嵌入式实时操作系统 VxWorks

实验教程

王 韬　　林　芹

印 勇　　黄扬帆

编著

重庆大学通信工程学院

2003 年 10 月

目　　录

1 嵌入式实时系统概述 1

1.1　嵌入式系统.....1

1.2　实时操作系统.....7

2 嵌入式微处理器MPC860 16

3 嵌入式实时操作系统VxWorks及其集成开发环境Tornado 25

3.1嵌入式实时操作系统VxWorks 25

3.2 Tornado集成开发环境 43

4 VxWorks的BootRom 48

5 实验 55

5.1 实验一、Tornado的使用 55

5.2 实验二、嵌入式实时系统概念的建立.....74

5.3 实验三、实时多任务程序的编写.....78

5.4 实验四、设备及文件管理.....101

5.5 实验五、嵌入式系统软件的交叉编译.....110

5.6 实验六、嵌入式系统软件的交叉调试.....116

附录A　hwa-xpc860实验板..... 120

5 实 验

5.1 实验一、Tornado的使用

① 实验目的

1、理解嵌入式系统交叉开发的概念

2、熟悉嵌入式软件开发的过程

3、理解使用 VxWorks 创建的工程文件类型

4、熟悉 Tornado 开发环境的使用以及开发环境集成的各种工具的作用

② 实验要求

1、正确连接和配置宿主机 Host 和目标机 Target（通过超级终端配置目标机启动参数和配置 Target Server 并启动它），实现交叉开发和调试。

2、创建 Bootable 工程生成 vxWorks 映像文件。

3、创建 downloable 工程，动态加载和卸载目标机上的目标文件，调试应用程序。

③ 实验原理

1、嵌入式交叉开发的含义

由于嵌入式应用系统的软件受时间和空间的限制，一般通过交叉开发来实现。交叉开发环境是指实现、编译、链接、调试应用程序代码的环境与运行应用程序的环境不同。分散在不同设备上，开发过程中，二者之间存在某种通信连接。提供实现、编译、链接、调试应用程序代码的环境的设备称为宿主机（Host），运行应用程序的设备称为目标机（Target）。

TornadoII 开发环境是嵌入式实时领域里最新一代的开发调试环境，是实现嵌入式实时应用程序的完整的软件开发平台，是交叉开发环境运行在主机上的部分，是开发和调试 VxWorks 系统不可缺少的组成部分。TornadoII 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。

TornadoII 开发系统包含三个高度集成的部分：

- 运行在宿主机和目标机上的强有力的交叉开发工具和实用程序；
- 运行在目标机上的高性能、可裁剪的实时操作系统 VxWorks；
- 连接宿主机和目标机的多种通讯方式，如：以太网，串口线，ICE 或 ROM 仿真器等。

Torando II 嵌入式集成开发系统结构图

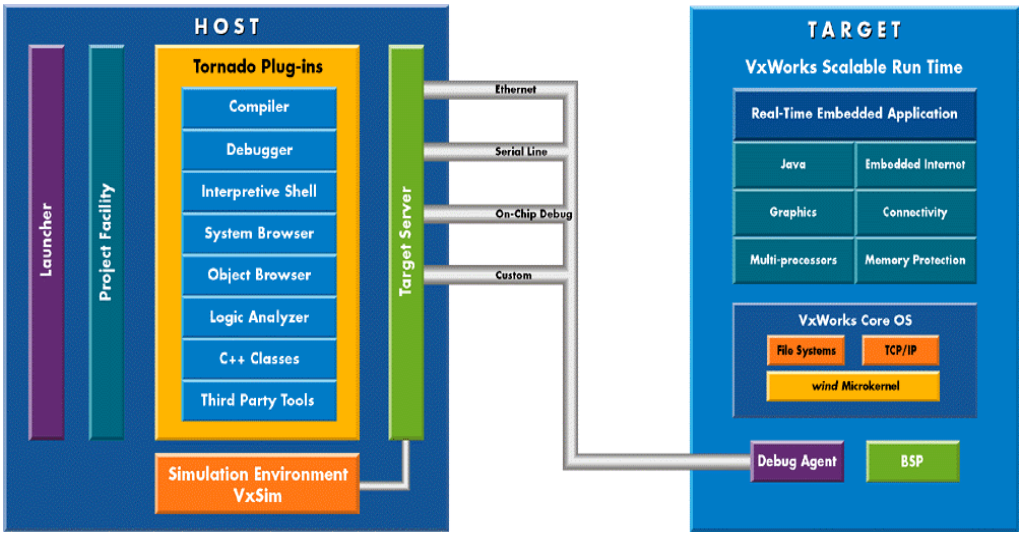

图 1　Torando II 嵌入式集成开发系统结构图

2、VxWorks 的工程类型及映像文件的类型

工程类型：

（1）、Bootable Project——用于为一个专门的 BSP 配置并编译链接接（Build）VxWorks image。应用程序的代码静态地与 VxWorks 链接。

（2）DownLoadable Project——用于管理和建立应用程序模块，应用程序的模块可以动态地下载到目标机上，可动态地与运行载目标机上的 VxWorks 连接。

映像文件的类型：

有三种类型的 VxWorks 映像文件类型：

- Loadable image：可以被 VxWorks BootROM 装载到目标机 RAM 中的 VxWorks image。
- ROM－Based image－Compressed/Uncompressed。这一类的 image 是 ROM 类型的 vxworks image，它通过编程器被烧录在目标板的 ROM 或 Flash 中。它包含 bootstrap 代码，bootstrap 的作用是将 vxworks 从 ROM 拷贝到 RAM 中去，如果是压缩的，就解压到 RAM 运行。
- ROM－Resident image－这也是一种 ROM 类型的 image，被烧录在目标板的 ROM 或 Flash 中，它也包含启动代码，与上一类型相比，该启动代码仅将 vxworks 的数据段从 ROM 中拷贝到 RAM 中的 RAM_LOW_ADRS 处，代码段仍然在 ROM 中执行。这种类型的 image 需要较小的 RAM 空间，但运行速度较慢。

3、Tornado 集成的开发工具介绍

可参见第三章的部分内容。

④ 实验步骤

1、创建 bootable 工程，生成可下载到目标板的 RAM 中运行的 vxWorks 映像文件。

（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：

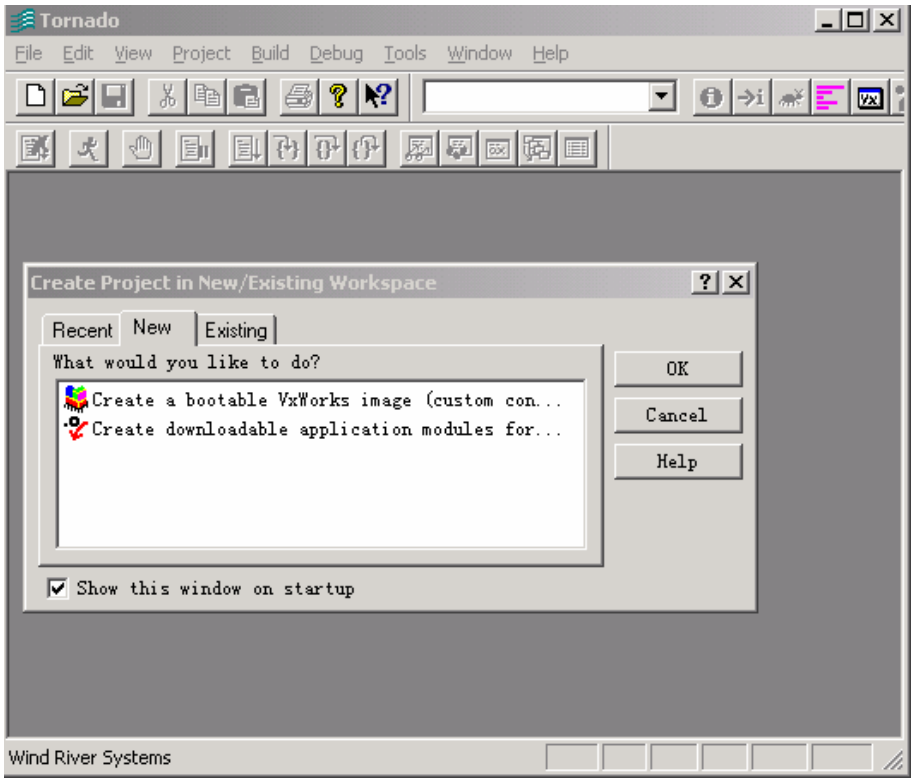

图 2 创建工程的对话框

（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

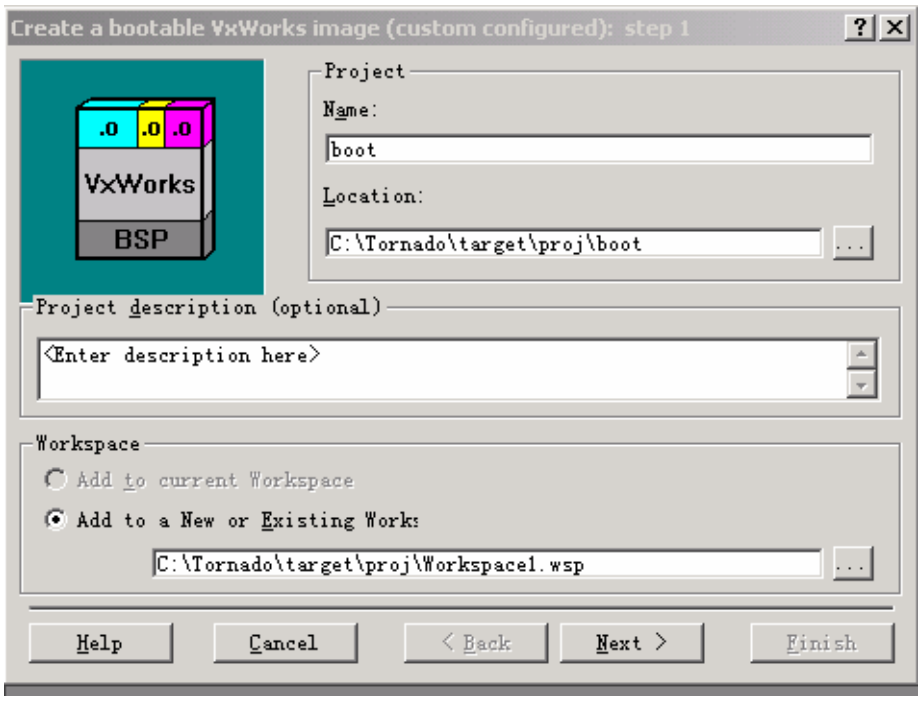

图 3 工程属性对话框

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：

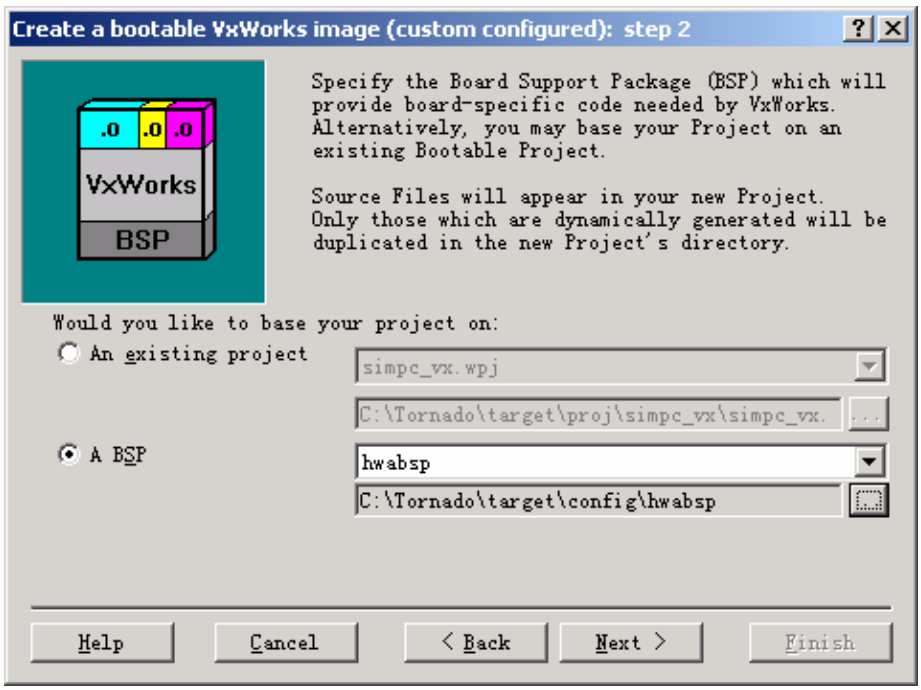

图 4　BSP 选择对话框

（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：

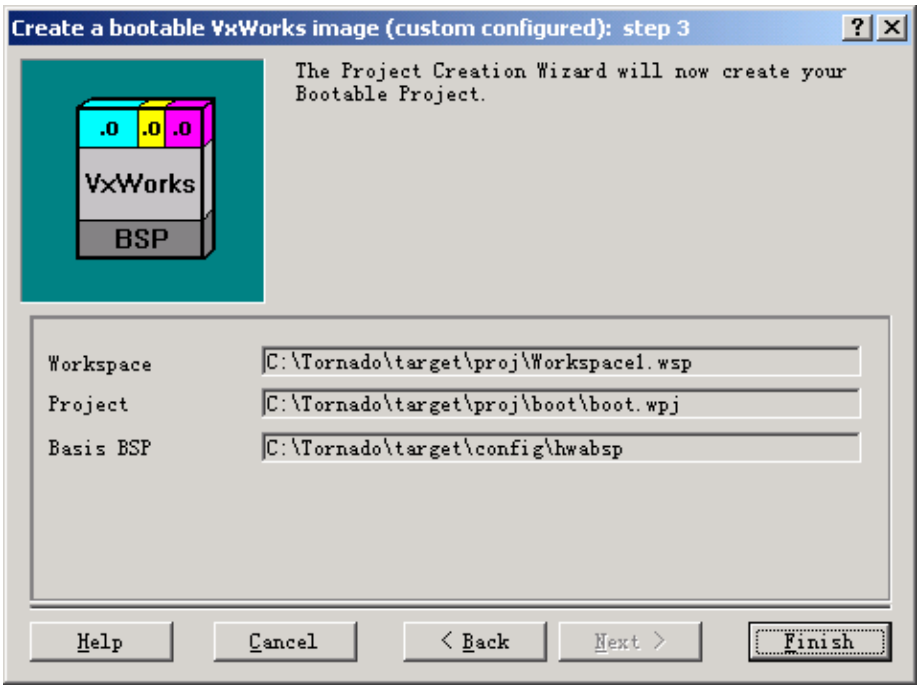

图 5 工程属性确认对话框

（5）点击 Finish，继续下一步，这时将出现工作空间窗口。

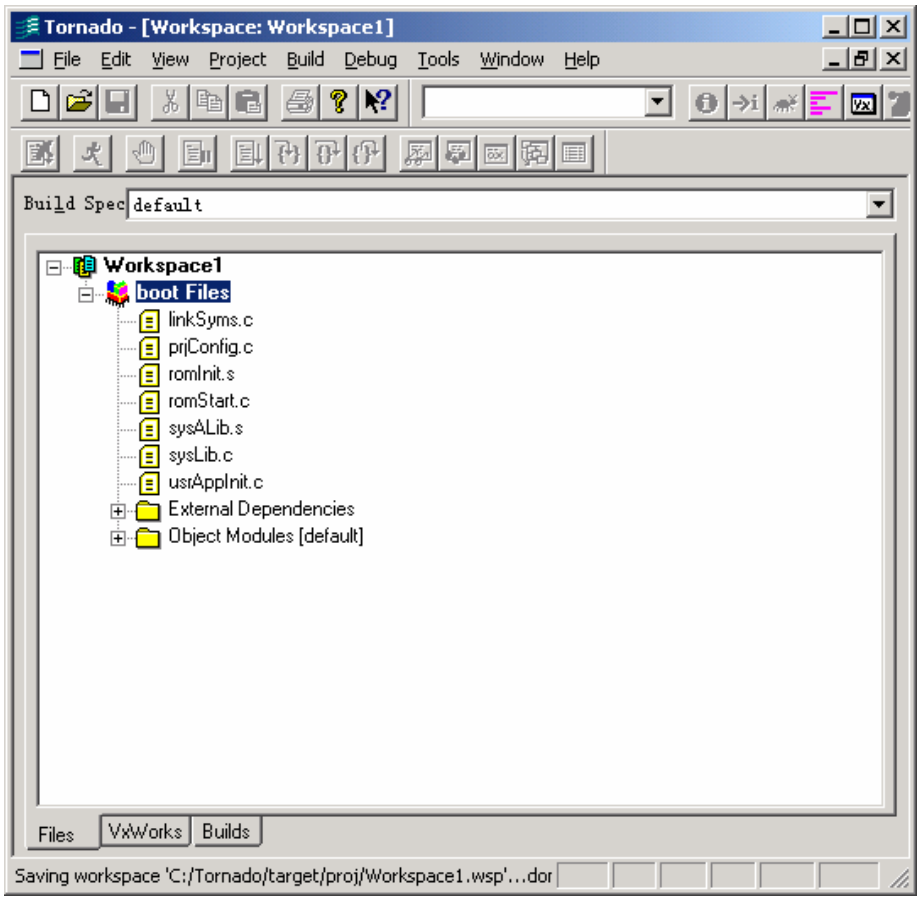

图 6 工程创建完成后的界面

（6）点击窗口左下方的 Builds 图标，出现下图界面：

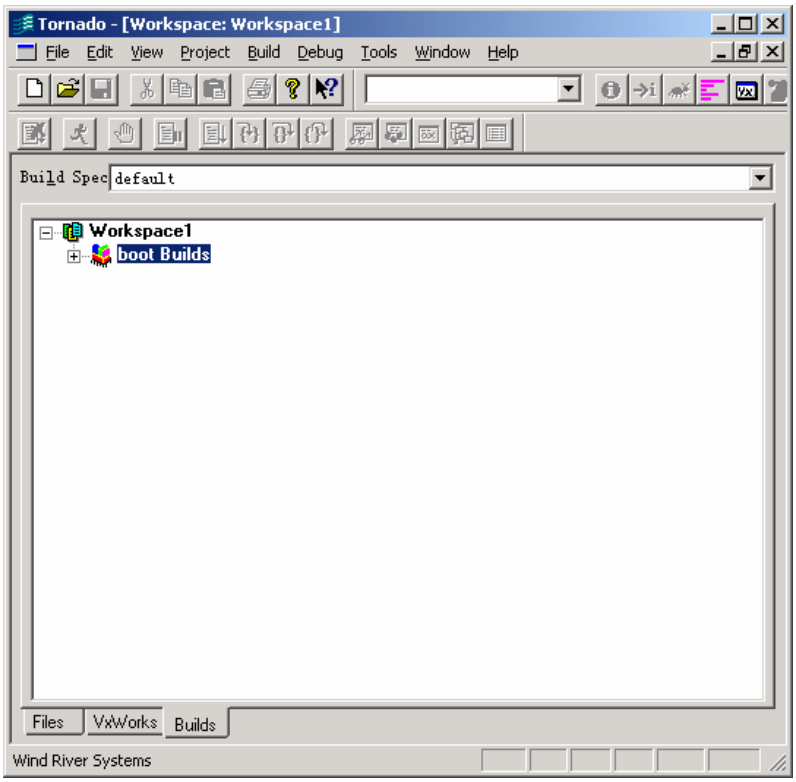

图 7 点击 Builds 图标后的界面

（7）点击“boot Builds”前的“＋”号，出现如下的画面：

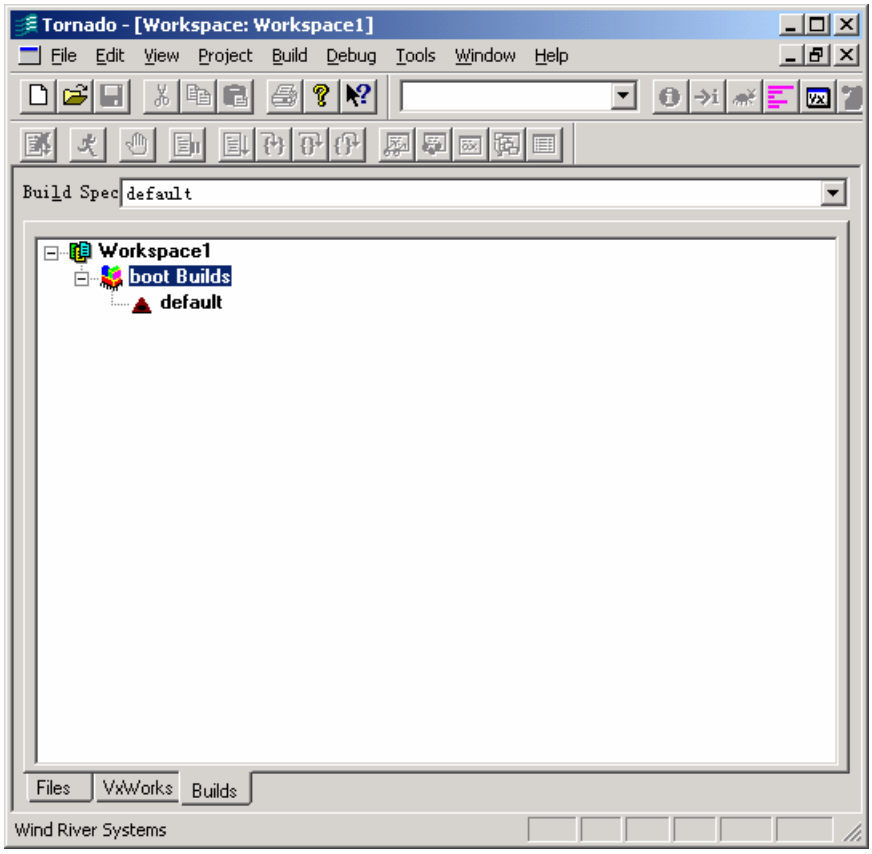

图 8 设置 default 属性

（8）双击 default 图标，出现编译规则对话框。

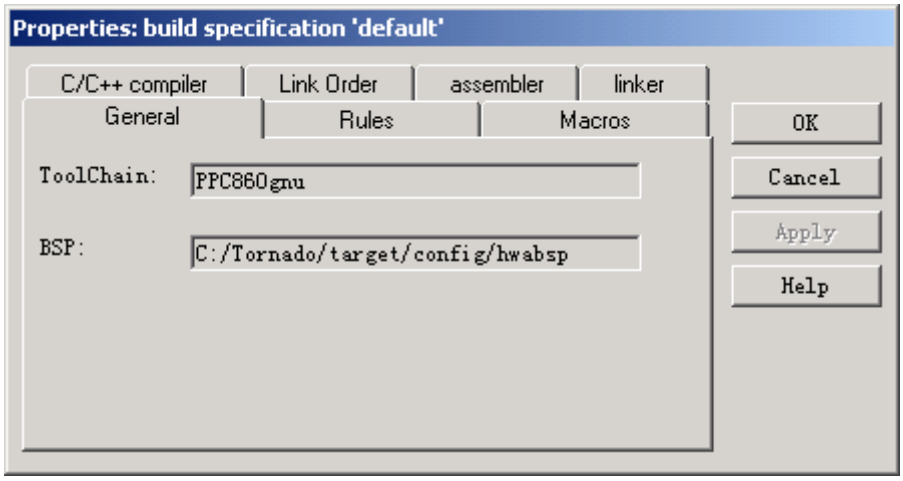

图 9 编译器及 BSP 属性对话框

（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。

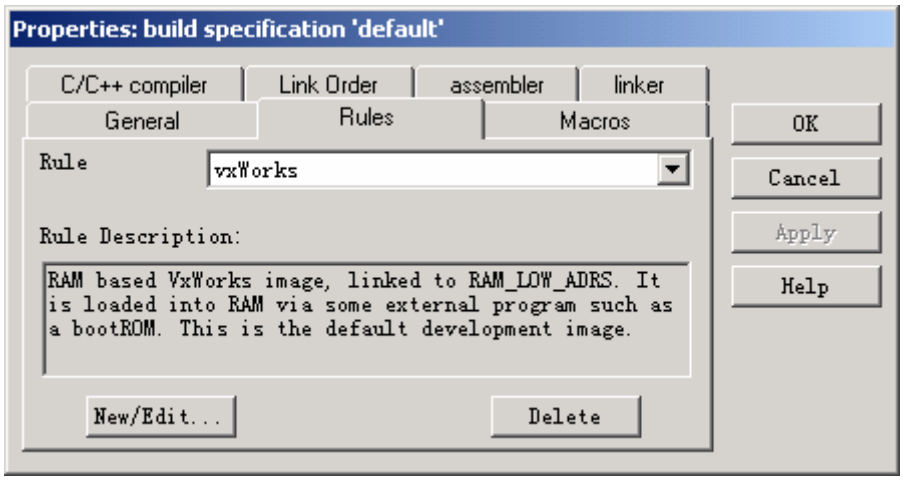

图 10 缺省目标映象文件选择属性框

（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：

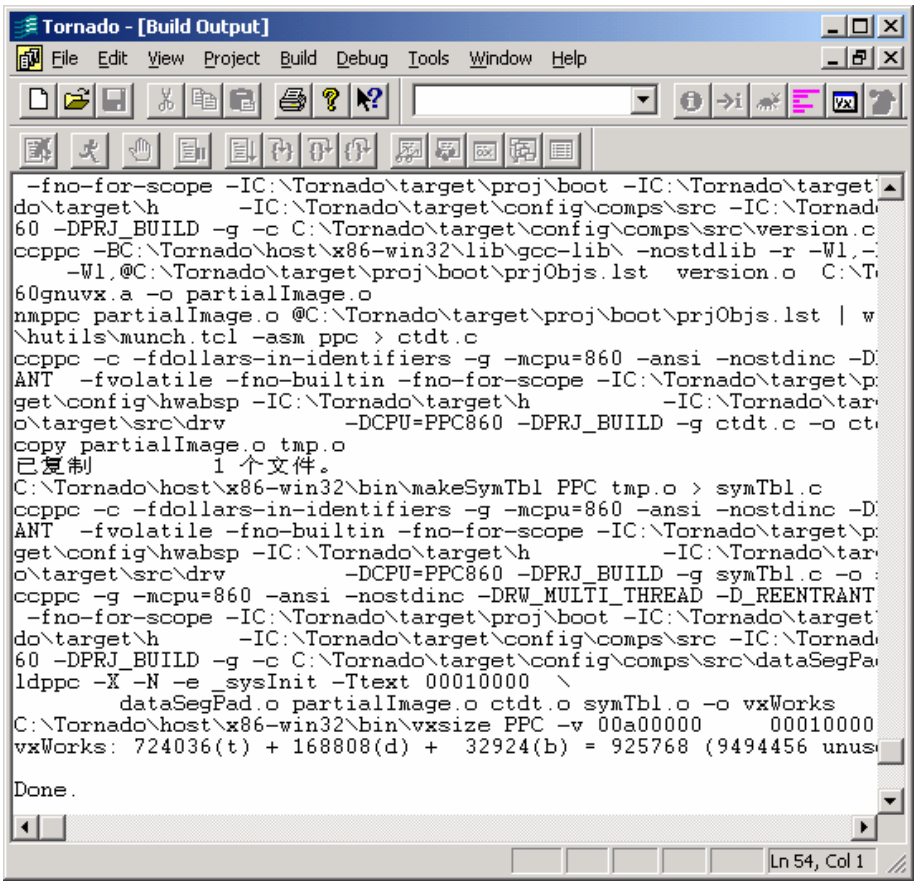

图 11 编译过程显示的信息

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

2、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。

3、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：

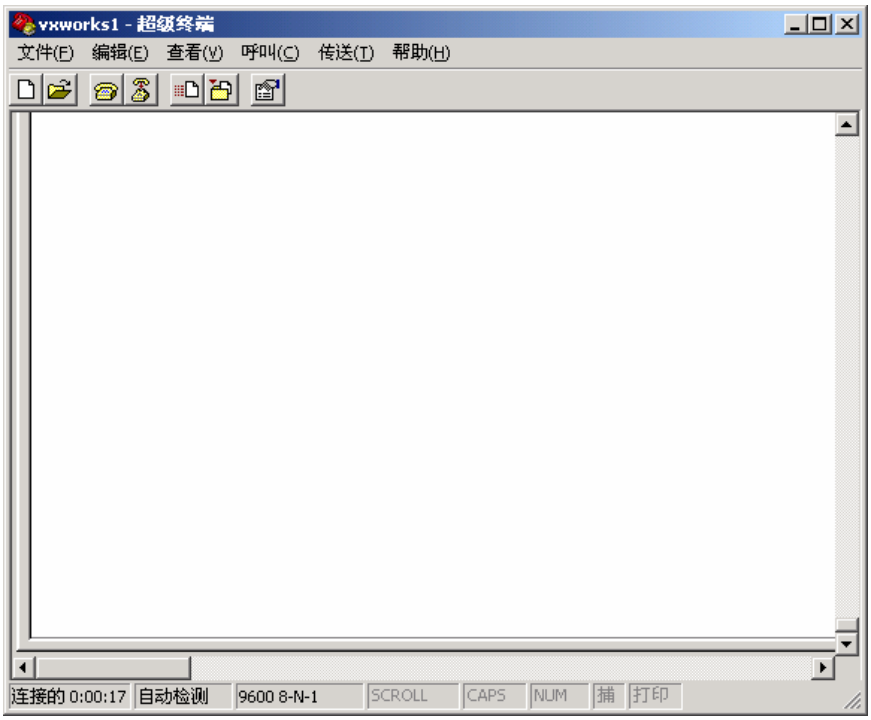

图 12 启动超级终端

4、目标板加电，这是串口的出现的信息如下：

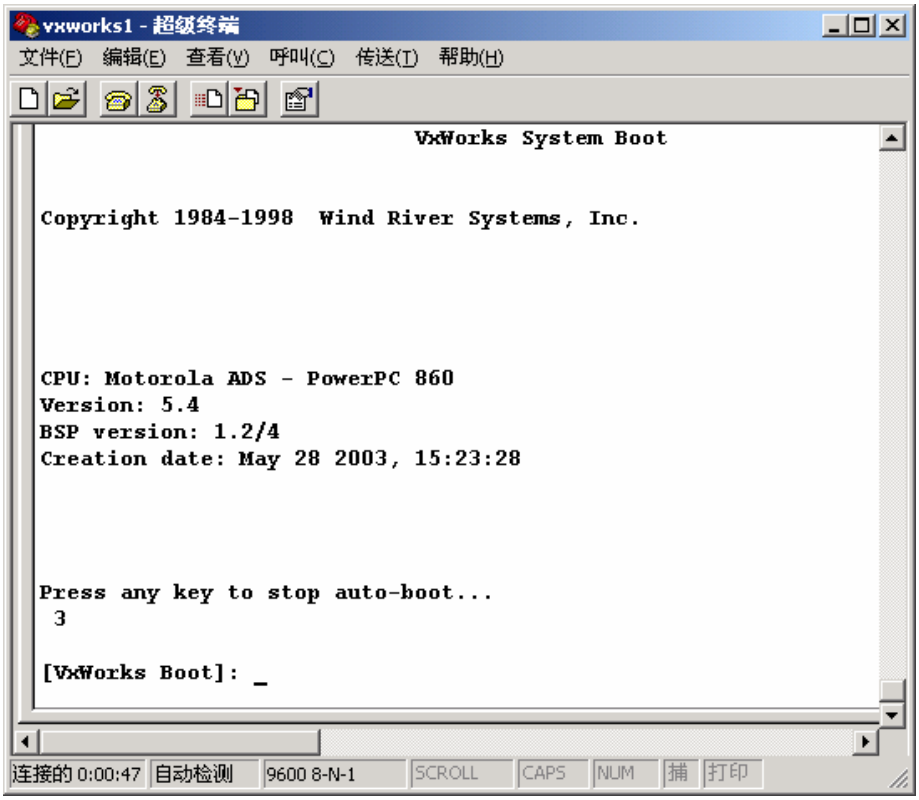

图 13 启动目标板串口超级终端打印的信息

5、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。

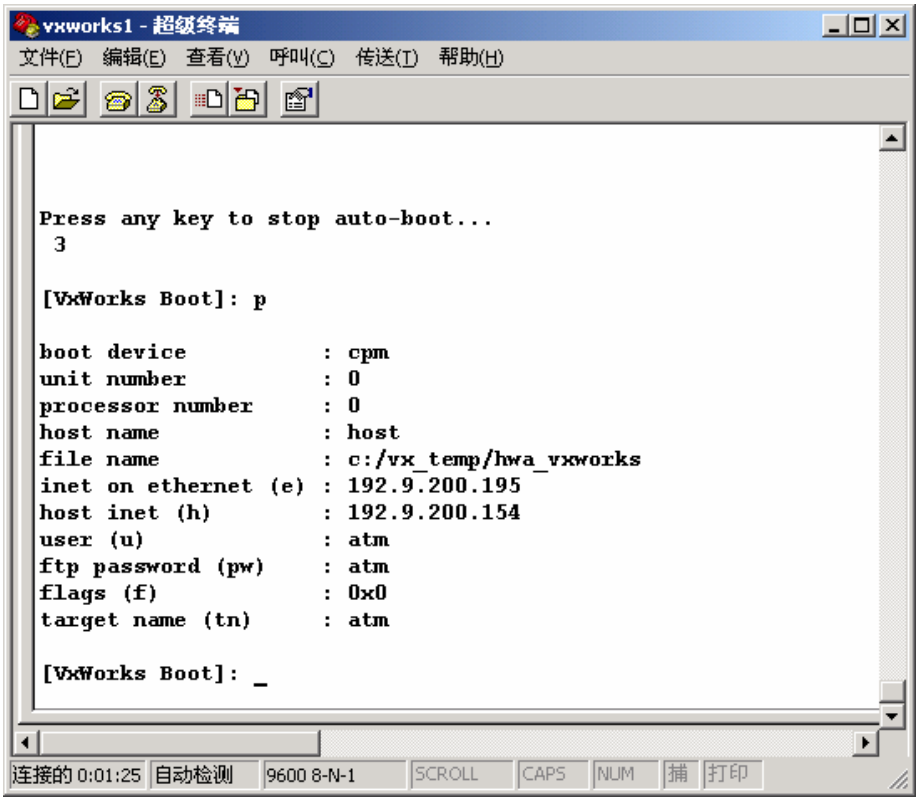

图 14 目标板缺省的启动配置信息

6、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。

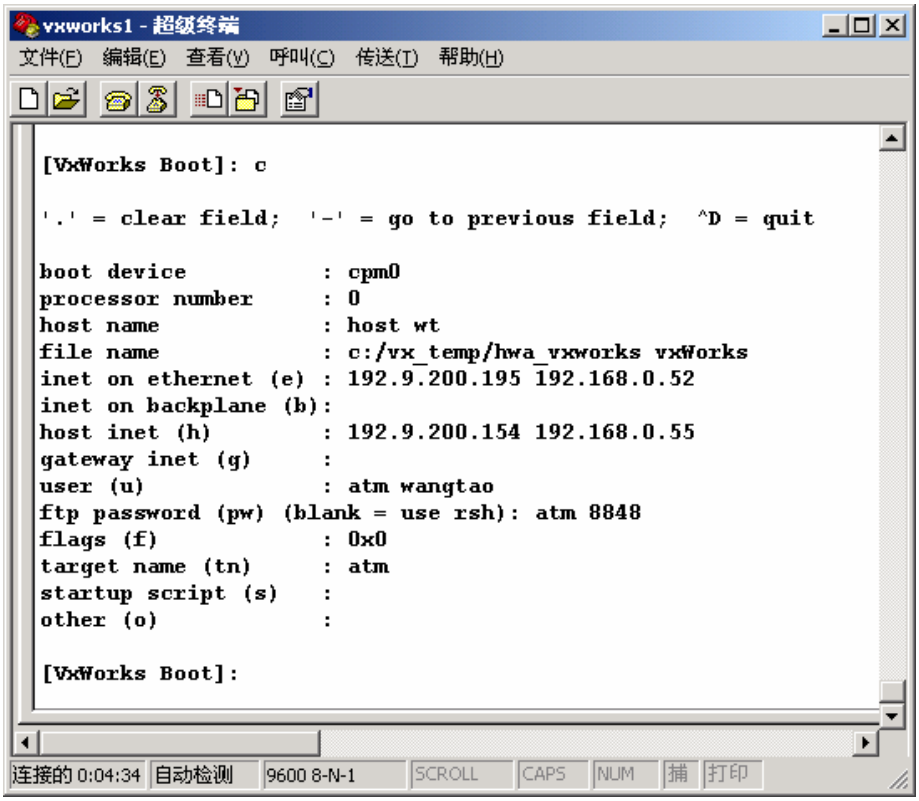

图 15 修改目标板缺省的启动配置信息

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

7、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 17 启动 FTP Server

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

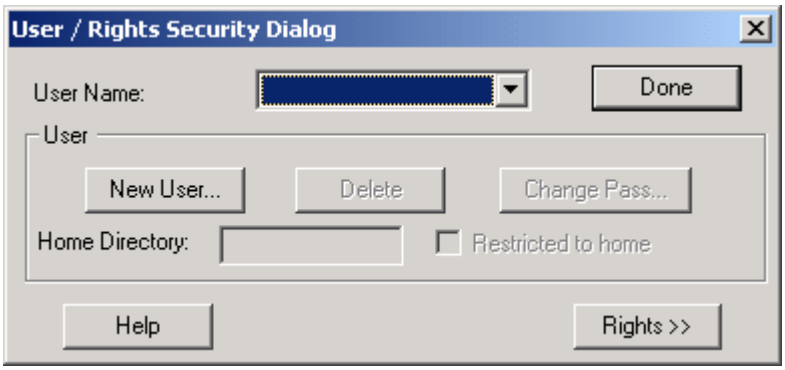

图 18 设置 User/Rights Security Dialog

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 19 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）。

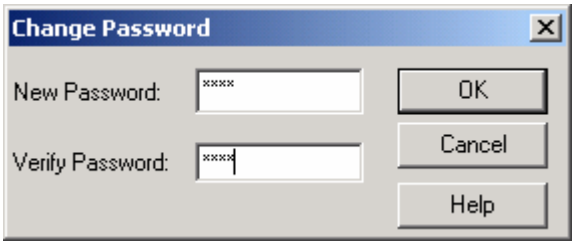

图 20 输入用户密码

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。

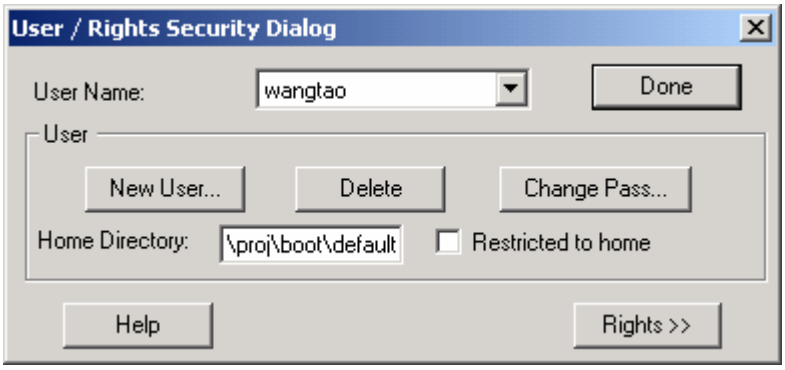

图 21 输入文件下载路径

8、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：

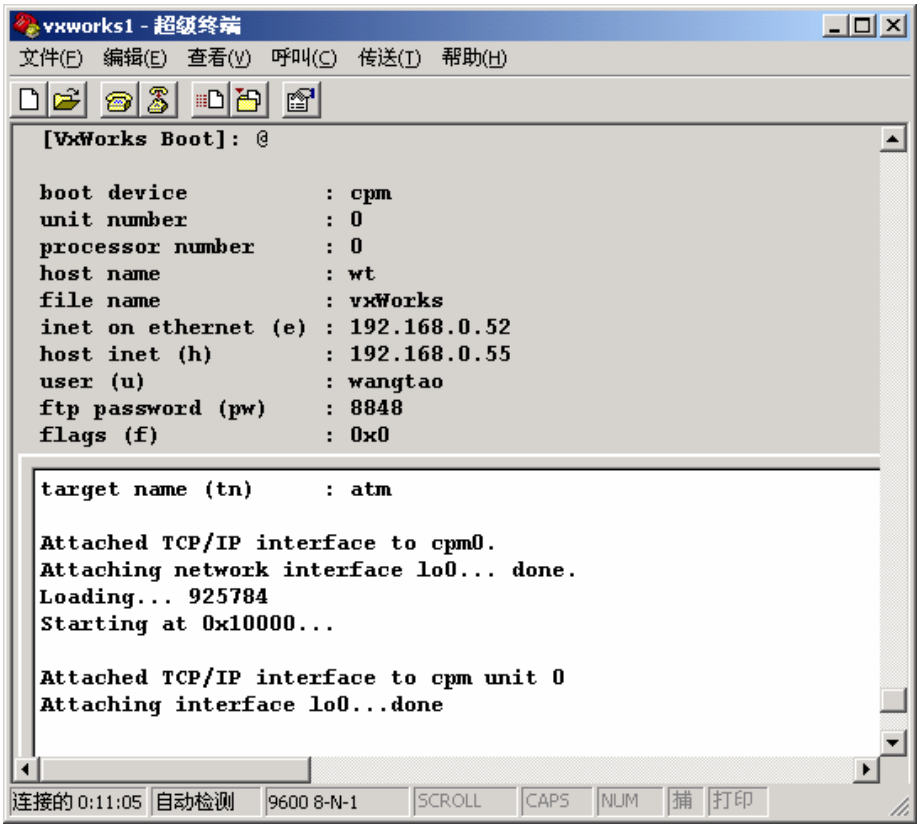

图 22 执行下载操作

9、创建 downloadable 工程，调试应用程序。

（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 …

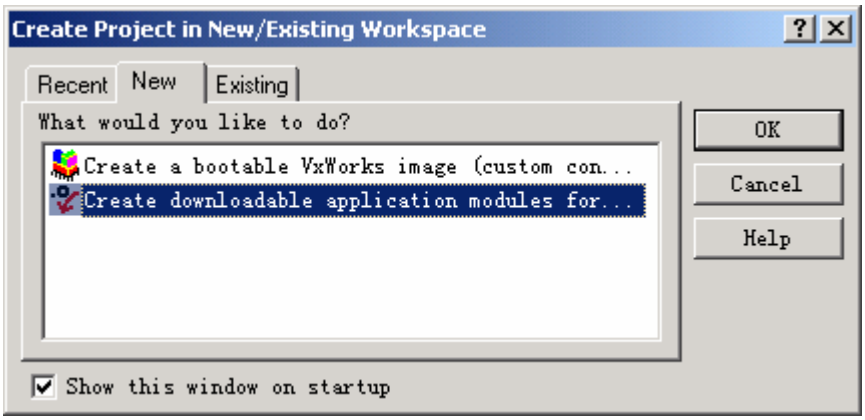

图 23 创建 Downloadable 工程

（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。

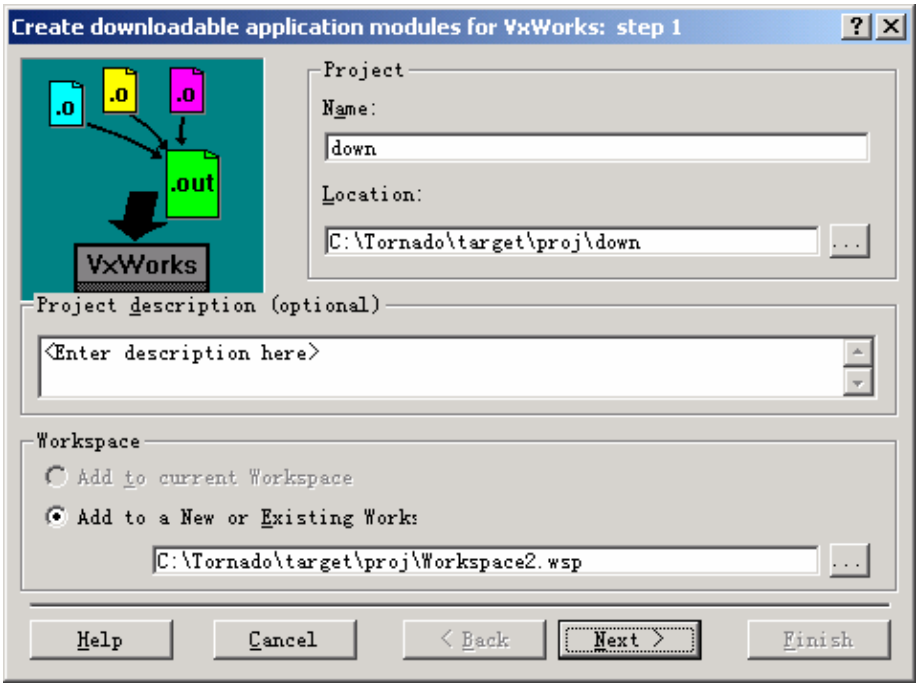

图 24 设置 Downloadable 工程的属性

（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。

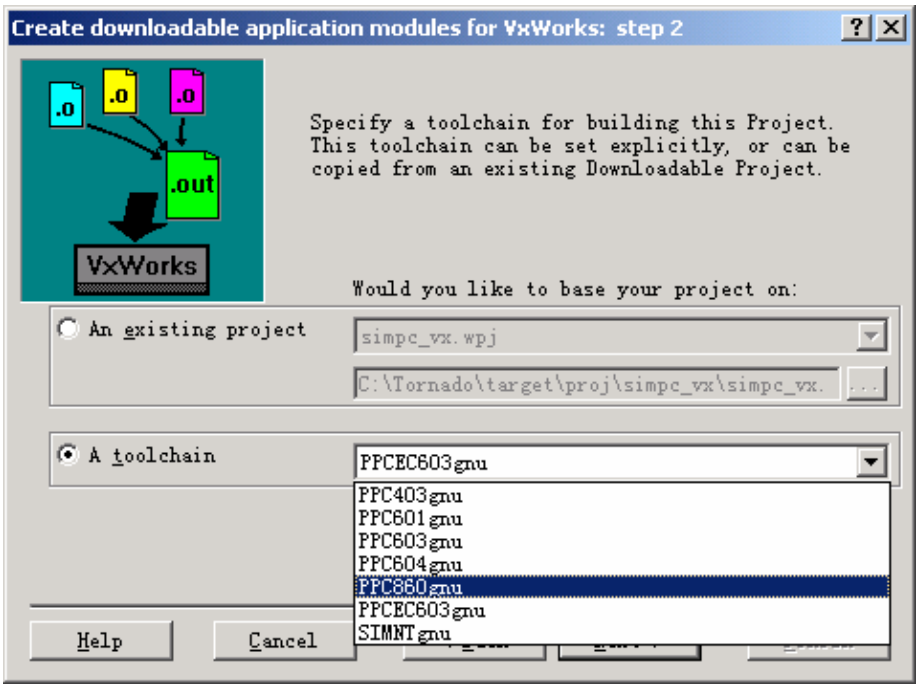

图 25 选择编译器

（4）单击 Next，弹出确认界面。

（5）单击 Finish，完成工程的创建。

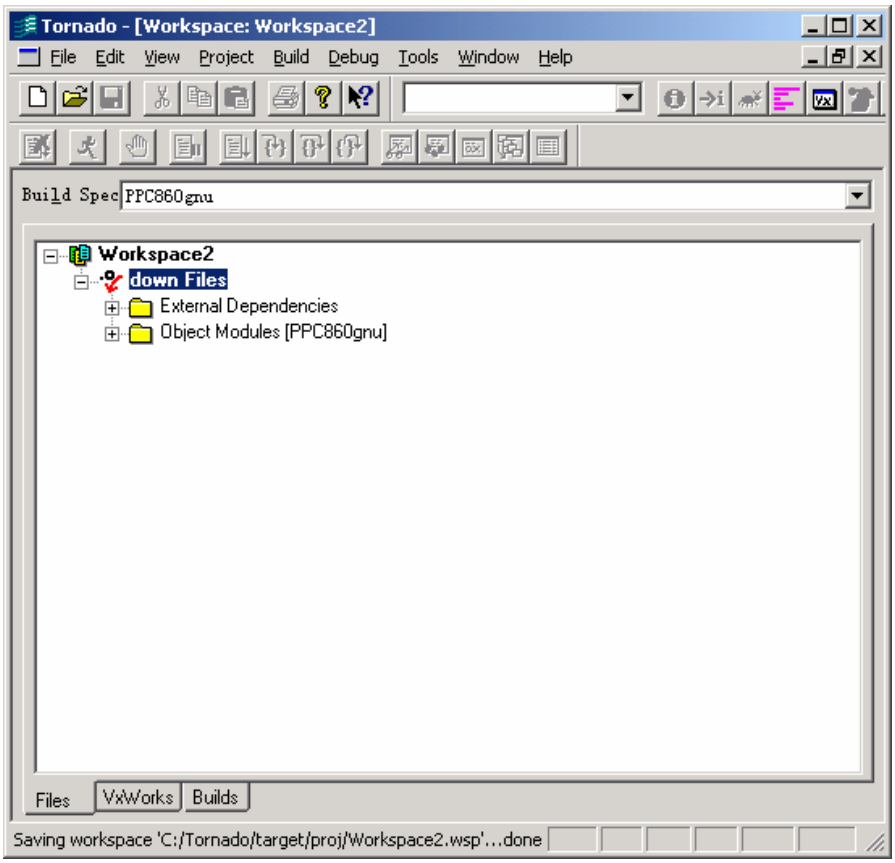

图 26 工程创建完毕后的界面

（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：

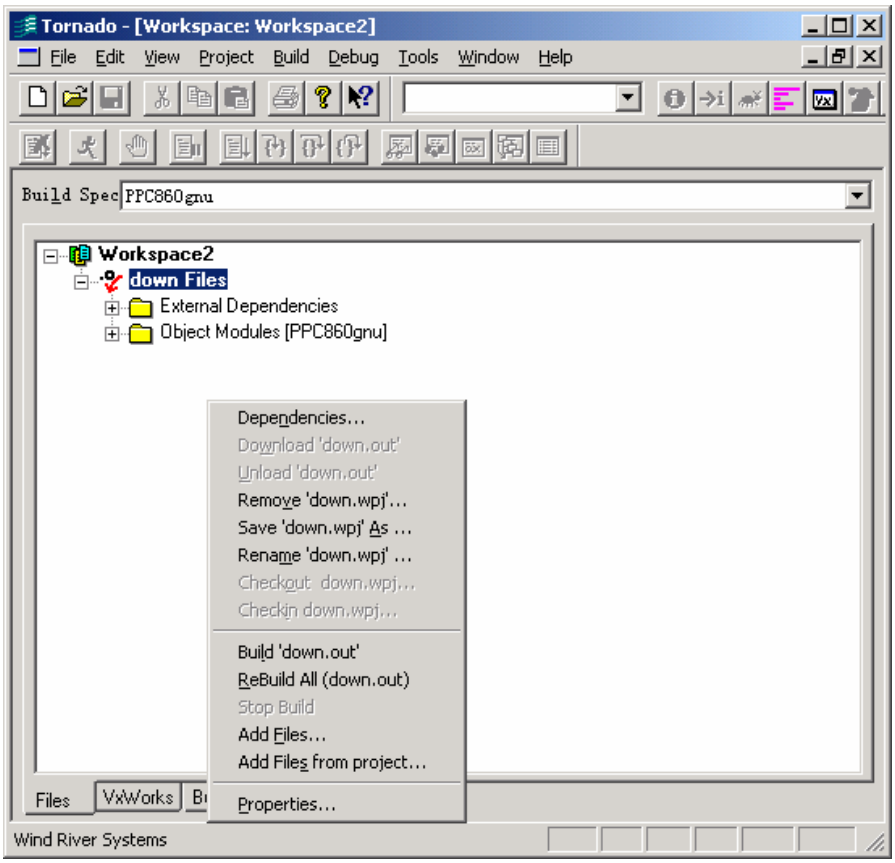

图 27 工程的上下文菜单

（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。

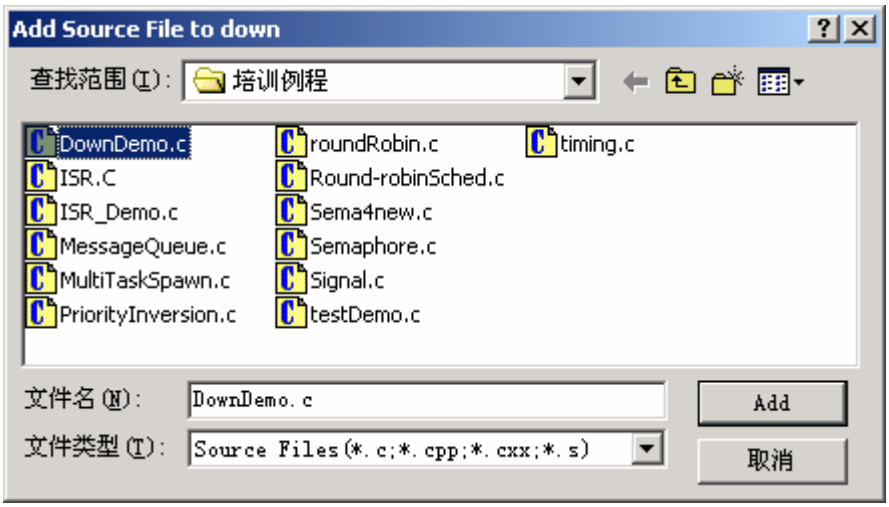

图 28 加入新的源文件

（8） 在加入源文件后，Tornado 的工作环境如下图所示：

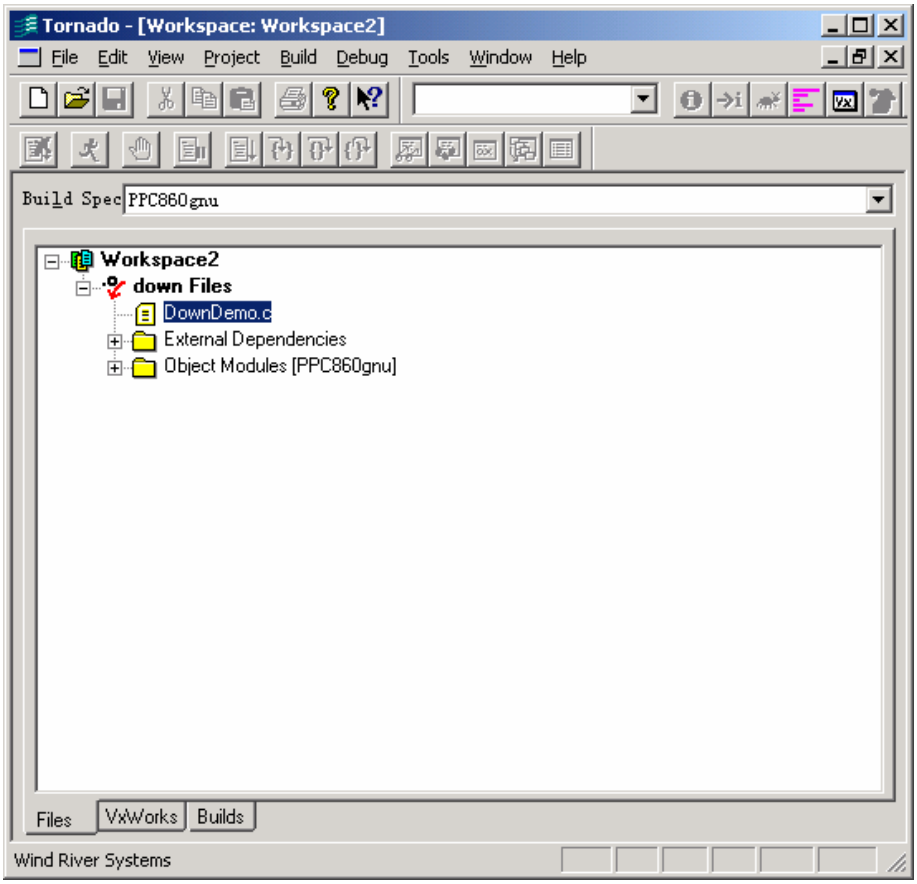

图 29 加入源文件后的界面

（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。

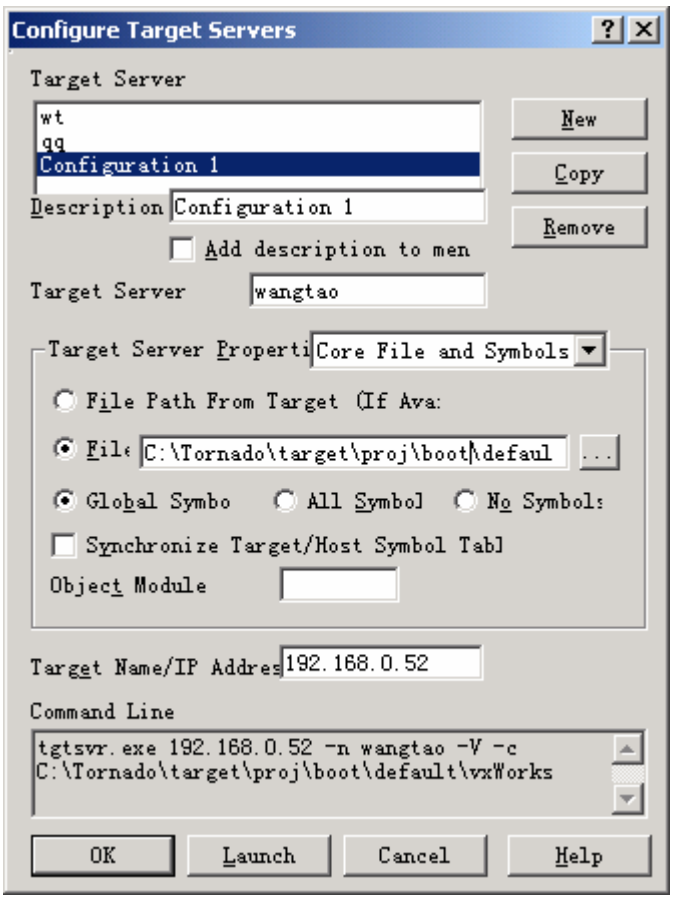

图 30 设置 Target Server

单击 New 命令新建一个 Target Server，输入 Target Server 名，在 Target Server Proerti 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。

10、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。

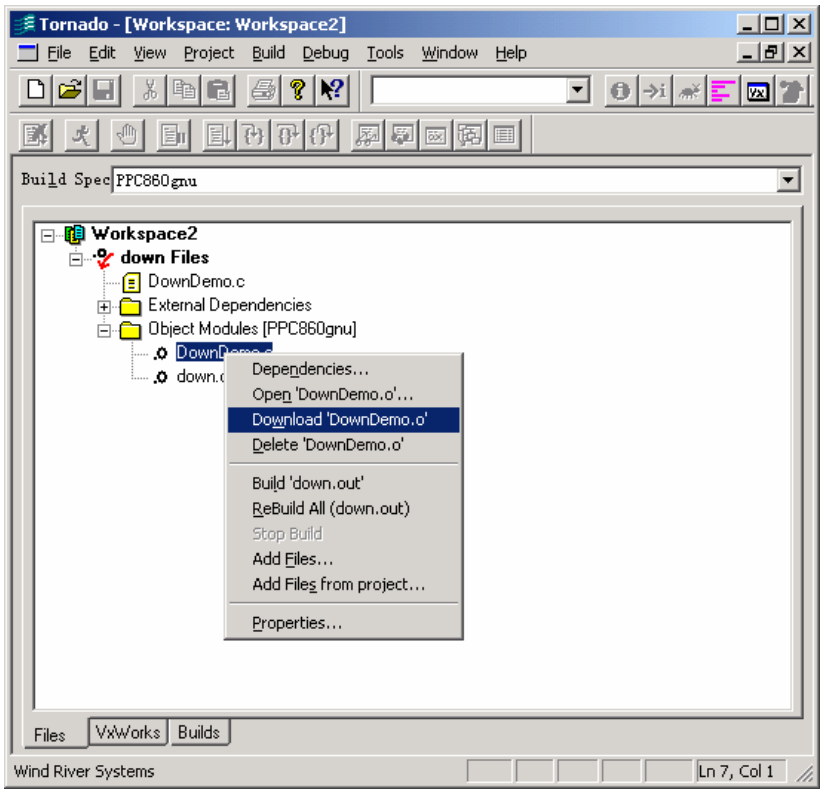

图 31 下载目标文件到目标机中

11、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：

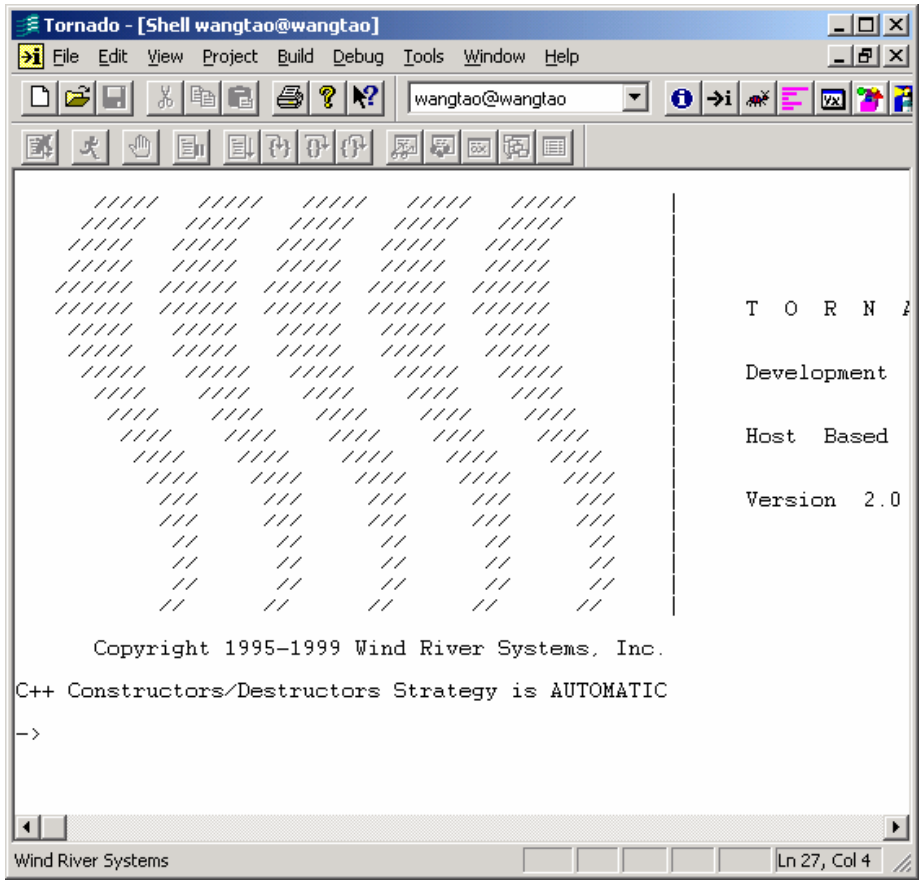

图 32 启动 WindSh

12、在 WindSh 的一>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。

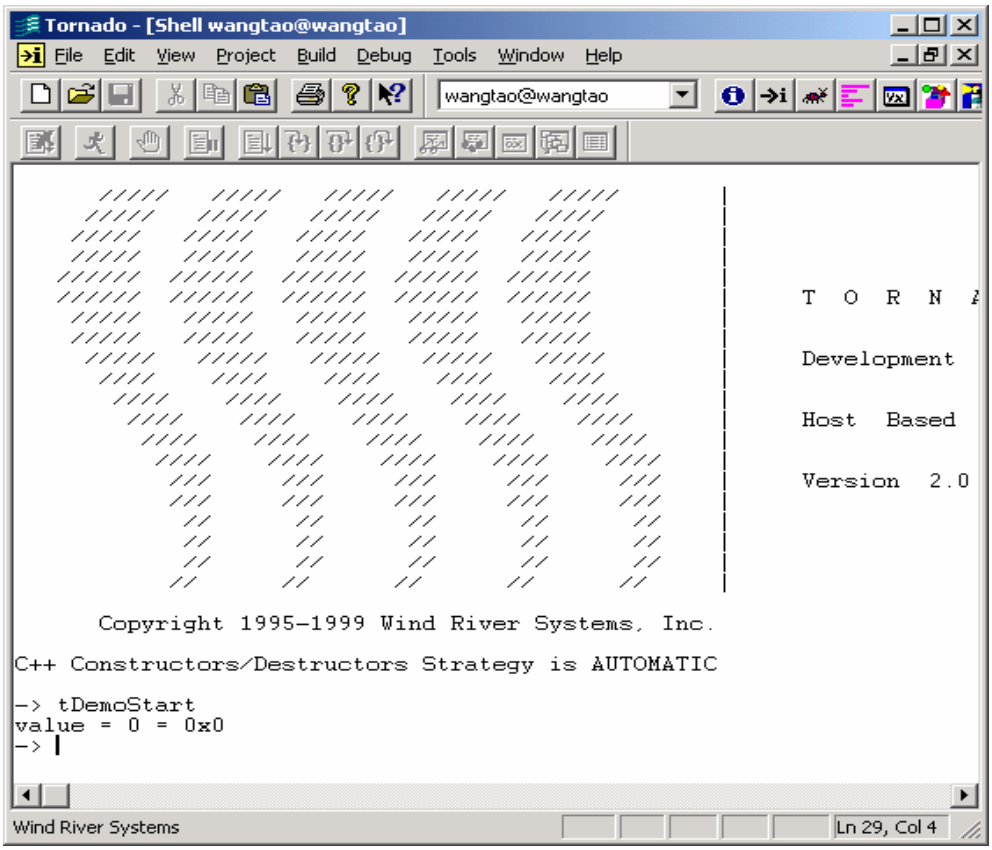

图 33 执行源文件中的 tDemoStart 函数

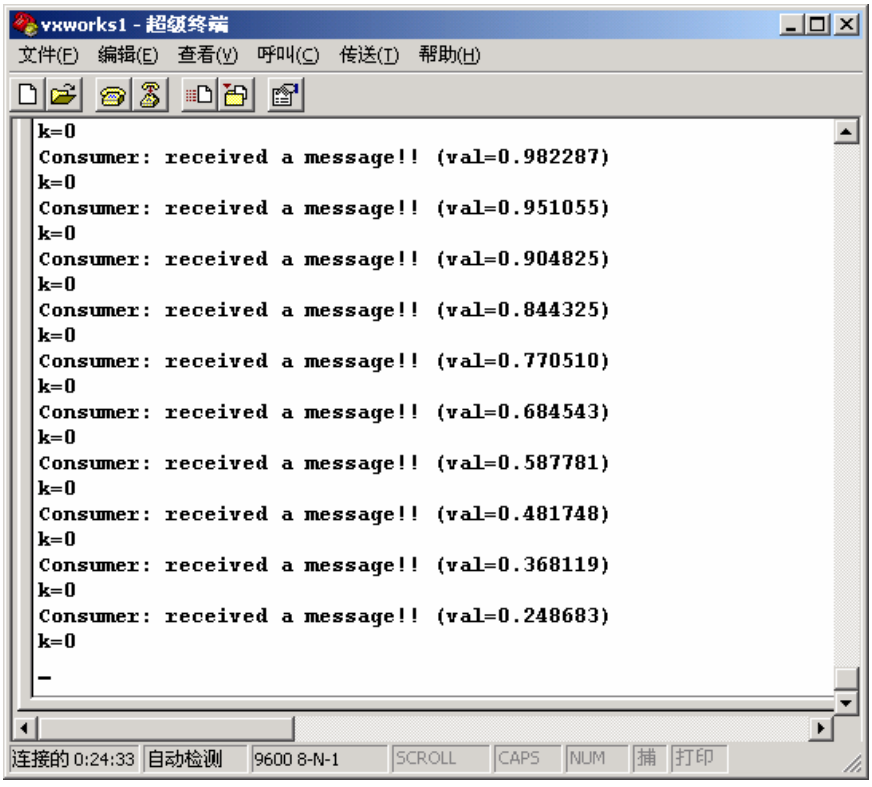

图 34 超级终端打印的函数执行信息

⑤ 思考题

1、如何卸载加载到目标机上的应用程序模块？

2、若要形成最终产品，即烧录到目标板的程序存储器中的映像文件，则应创建什么类型的工程，选择什么样的映像文件作为默认的编译规则？

5.3 实验三、实时多任务程序的编写

① 实验目的

1) 在 VxWorks 下如何创建并启动多个任务
2) 掌握任务间常用的通信机制
3) 掌握任务间常用的同步和互斥机制
4) 掌握实时操作系统中的常用调度方式

② 实验要求

1) 使用 VxWorks 提供的任务管理函数 taskSpawn（）创建十个任务，每个任务打印出各自的 ID 号
2) 使用二进制信号量来；使用互斥信号量
3) 使用 kernelTimeSlice（）函数来允许轮转调度。设计一段程序来实现基于优先级的轮转调度。
4) 设计一段程序来实现任务间使用消息队列进行通信。

③ 实验原理

1) 现代实时系统基于多任务和任务间通信的概念，—个多任务环境允许实时应出构造成多个独立任务组成的集合，每个任务单独执行，拥有自己的一套资源。任务间通信功能允许这些任务间同步以协调他们的活动。多任务制造一种多个线程同时执行的假象。事实上，内核基于调度算法插入到这些任务的执行中。每个单独执行的程序称作一个“任务”。每个任务拥有自己的上下文，包括 CPU 环境和系统资源，这些是内核调度该任务运行时所必需的。

任务的创建和激活可以通过系统调用 taskSpawn 来实现。函数 taskSpawn() 创建一个新的任务上下文，包括为程序执行分配和建立任务环境，传递参数，新任务的入口由第五个参数指定。其他参数包括任务名、优先统、一个“option”选项、堆栈大小以及传递给入口函数的 10 个整形参数。返回值为任务 ID 号。其语法是：

id＝taskSpawn（name，ptiority，options，stacksize，function，arg1…arg10）

2) VxWorks 在单 CPU 中，多任务通信的主要机制是消息队列。消息队列允许以 FIFO 或基于优先级方式排队消息，消息的数目可变，消息的长度可变。任何任务都可以向消息队列发送消息，也可以从消息队列接收消息。多个任务允许从同一个消息队列收发消息。但是，两个任务间的双向通信通常需要两个消息队列，各自用于一个方向。

VxWorks 消息队列的创建、发送、接收和删除的系统调用如下：

（1） msgCreate（int maxMsgs，int maxMsgLength，int options）：分配

并初始化一个消息队列。消息的最大数目以及一个消息的最大长度由其参数决定。

（2） msgDelete（MSG_ID msgQId）：终止并释放一个消息队列。

（3） msgQSend（MSG_Q_ID msgQId,char *Buffer，UINT nBytes，int timeout，int prioty）：向一个消息队列发送一个消息。如果没有任务在等待该队列的消息，那么这条消息增加到该队列的消息缓冲中，如果有任务在等待，那么该消息立即提供给第一个等待的任务。

（4） msgQReceive（MSG_Q_ID msgQId,char *Buffer，UINT nBytes，int timeout）从一个消息队列接收消息。任务如果需要从一个消息队列接收一条消息，它应该调用 msgQreceive（）。如果该消息队列中已有消息可用，那么队列中的第一条消息立即出队，并提交给调用任务；如果没有消息可用，那么调用任务阻塞，并且加入到等待该消息的任务队列中。等待任务队列可以按两种方式排队：基于任务优先级或基于 FIFO 方式，由消息队列创建时指定。

- **Timeouts**：函数 msgQSend（）和 msgQreceive（）都可以说明一个超时参数，规定任务等待的时间（tick 数）：发送消息任务等待队列空间可用，接收消息任务等待消息可用。
- **Urgent Messages**：函数 msgQSend（）可以指定欲发送消息的优先级：正常 MSG_PRI_NORMAL 或紧急 MSG_PRI_URGENT。正常优先级的消息将加入到消息队列的尾部，而紧急优先级的消息增加到消息队列的头部。

3)、信号量允许多个任务相互协调其活动。任务间最直接的通信方式是共享各式各样的数据结构。由于 VxWorks 中，所有任务存在于一个单一的线性地址空间，共享数据结构也就非常容易实现。全局的变量、线性缓冲、环形缓冲、连结链和指针都可以被运行在不同上下文的代码直接引用。

然而，共享地址空间简化数据交换的同时，需要保证这块内存的互斥访问。vxWorks 提供了许多实现共享临界区互斥访问的机制，信号量就是其中一种。VxWorks 信号量是任务通信最优选择，同时也提供了最快的通信方式，信号量是实现任务间同步和互斥最直接的方式。

Wind 内核提供三种类型的信号量：用于解决不问类型的问题；

- 二进制信号量（binary）：最快的、最普通的信号量，适合于实现同步，也适合于互斥。
- 互斥信号量（mutual exclusion）：一种特殊的二进制信号量，适合于解决具有内在互斥问题：优先级集成、删除安全和递归。
- 计数信号量（counting）：类似于二进制信号量，但是可以记录信号量释放的

次数。适合于对一类包含多个数目的资源的访问。在实际使用中，很少采用。

信号量的控制

Wind 内核对上述三类信号量提供统一的调用接口。信号量的类型在创建时说明。

- semBCreate（int options，SEM_B_STATE initialState）：分配并初始化一个二进制信号量。
- semMCreate（int options）：分配并初始化一个互斥信号量。
- semCCreate（int options，SEM_B_STATE initialCount）：分配并初始化一个计数器信号量。
- semDelete（SEM_ID semID）：终止并释放一个信号量
- semTake（SEM_ID semID）：取一个信号量
- semGive(SEM_ID，semID)：释放一个信号量。
- semFlush（SEM_ID semID）：解锁所有等待该信号量的任务。

4)、任务调度策略

所谓任务调度就是基于某种规则约束，给一个任务集合中的每个任务分配开始和结束时间。约束一般包括时间约束和资源约束。在一个时间共享(timing)操作系统中，系统按照时间片依次轮流每个任务(或进程)，因此，制造一种多个任务在单个处理器上同时执行的假象。

Wind 内核内核调度默认使用基于优先级抢占式调度，但是问时也允许使用轮转调度。下面介绍以下两种调度方式。

基于时间片的轮转调度：

轮转调度算法目的是使相同优先级的所有就绪任务共享 CPU。如果不使用轮转调度，当多个相同优先级的任务需要共享处理器时，其中的一个任务因为永不阻塞，可能霸占处理器，造成其他同等优先级的任务得不到运行的机会。

时间片（timing slicing）是轮转调度在多个相同优先级的任务间实现 CPU 公平分配的基础。每个任务执行一段确定的时间(一个时间片)；然后另一个任务执行同样大小的一段时间，如此循环下去。这种分配是相当公平的：

在这个优先级相同的任务组中的其他就绪任务没得到一个时间片之前，不会出现一个得到第二个时间片的情形。

在 Wind 内核中，调用函数 kernelTimeSlice（）将允许轮转调度，时间片的大小由参数传给它，规定了每个任务一次占用 CPU 的最长时间。

函数的原型：

kernelTimeSlice（int ticks）控制轮转调度，参数是时间片，以系统 tick 为单位。

基于优先级的抢占式调度：

使用基于优先级的抢占式调度，每个任务有一个优先级，任一时刻，内核保证将 CPU

分配给处于就绪状态的优先级最高的任务执行。之所以说这种调度算法式抢占的是因为：如果在某一时刻，一个优先级比当前正在运行的任务的优先级高的任务变为就绪，那么内核立即保存当前任务的上下文，然后切换到这个最高优先级任务的上下文。

Wind 内核有 256 个优先级(0—255)，优先数 0 对应着最高优先级，优先数 255 对应着最低优先级。任务的优先级在其创建时指定，VxWorks 也允许任务在执行时调用由 taskPriority（）改变自身的优先级。

函数：taskspawn()用于创建并激活一个新任务。原型如下

id＝taskSpawn（name，priority，options，stacksize，function，arg1，…，arg10）。

④ 实验步骤

1、多任务程序的创建和启动

（1）、编写一个程序，该程序创建是个任务，每个任务打印输出子的任务 ID 号。

参考源程序如下：

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

#define TaskNumber 10

void printTaskInfo(void);

STATUS spawnTenTask(void)
{
    int i, taskId;
    for(i=0; i<TaskNumber; i++)
    {
        taskId=taskSpawn( "tPrint",
90+i,0x100,2000,(FUNCPTR)printTaskInfo,0,i,0,0,0,0,0,0,0,0);
    }
}

void printTaskInfo(void)
{
```

```
    printf("Hello, I am task%d\n",taskIdSelf() );/* print the task id */
};
```

（2）创建一个 downloadable application 工程，在该工程中，加入该程序代码，并编译生成对应的 xx.o 文件。

（3）启动目标机，将该 xx.o 文件下载到目标板中，启动 Tornado 集成开发环境中 WindSh 工具，在出现的命令提示符下输入 spawnTenTask，运行代码，观察运行结果。

2、时间片轮转调度

（1）编写程序代码，实现三个优先级相同的任务向控制台输出它们的任务 id 号和任务名。源代码如下：

```
/* include files*/

#include "vxWorks.h"
#include "taskLib.h"
#include "kernelLib.h"
#include "sysLib.h"
#include "logLib.h"
#include "stdio.h"

/* function defined */
void taskOne(void);
void taskTwo(void);
void taskThree(void);

/* global varibles*/
#define ITER1    5
#define ITER2    1
#define PRIORITY 103
#define TIMESLICE 3
#define LONG_TIME 100

void sched(void)/*function to create three tasks*/
```

```
{
    int taskIdOne,taskIdTwo,taskIdThree;

    if( kernelTimeSlice(TIMESLICE)==OK)
        printf("\n\n\n\t\t\tTIMESLICE=%dseconds\n\n",TIMESLICE);

    if(                                                    taskIdOne=taskSpawn("task1",
PRIORITY,0x100,20000,(FUNCPTR)taskOne,0,0,0,0,0,0,0,0,0,0)==ERROR)
        printf("\nTaskSpawn taskone failed\n");
    if(                                                    taskIdTwo=taskSpawn("task2",
PRIORITY,0x100,20000,(FUNCPTR)taskTwo,0,0,0,0,0,0,0,0,0,0)==ERROR)
        printf("\nTaskSpawn tasktwo failed\n");
    if(                                                  taskIdThree=taskSpawn("task3",
PRIORITY,0x100,20000,(FUNCPTR)taskThree,0,0,0,0,0,0,0,0,0,0)==ERROR)
        printf("\nTaskSpawn taskthree failed\n");
}

void taskOne(void)
{
    int i,j;
    for(i=0;i<ITER1;i++)
    {
        for(j=0;j<ITER2;j++)
            logMsg("This is task 1111111\n",0,0,0,0,0,0);
        for(j=0;j<LONG_TIME;j++);/*Allow time to context switch*/
    }
}

void taskTwo(void)
{
    int i,j;
    for(i=0;i<ITER1;i++)
    {
        for(j=0;j<ITER2;j++)
```

```
            logMsg("This is task 2222222\n",0,0,0,0,0,0);
        for(j=0;j<LONG_TIME;j++);/*Allow time to context switch*/
    }
}

void taskThree(void)
{
    int i,j;
    for(i=0;i<ITER1;i++)
    {
        for(j=0;j<ITER2;j++)
            logMsg("this is task 3333333\n",0,0,0,0,0,0);
        for(j=0;j<LONG_TIME;j++);/*Allow time to context switch*/
    }
}
```

（2）在创建的 downloadable application 工程中，加入该源代码，编译生成对应的 xx.o 文件。

（3）将该文件下载到目标机中，在 Tornado 中启动 WindSh，在 windSh 命令行提示符下，输入 sched，运行代码，观察实验结果。

3、基于优先级抢占式调度

（1）编写源程序，在程序中有三个优先级各不相同的任务。程序运行的结果将是拥有最高优先级的任务 taskThree 首先运行完成，紧跟着优先级次高的任务 taskTwo 运行，等它完成后，优先级最低的任务 taskone 才获得运行。源程序如下：

```
/****include*************************/
#include "vxWorks.h"
#include "taskLib.h"
#include "logLib.h"
/**function prototypes******/

void taskOne(void);
void taskTwo(void);
void taskThree(void);
```

```
/****gloals****/
#define ITER1 100
#define ITER2 1
#define LONG_TIME 1000000
#define HIGH 100 /*high prioty */
#define MID 101 /**medium prioty****/
#define LOW 102 /***low prioty**/


void sched(void)      /***function to creat the two task*/
{
int taskIdOne,taskIdTwo,taskIdThree;

printf("\n\n\n\n\n");

/*Spawn the three tasks*/
if((taskIdOne=taskSpawn("task1",LOW,0x100,20000,(FUNCPTR)taskOne,0,0,0,0,0,
0,0,0,0,0))==ERROR)
     printf("taskSpawn taskOne failed\n");
if((taskIdTwo=taskSpawn("task2",MID,0x100,20000,(FUNCPTR)taskTwo,0,0,0,0,0,
0,0,0,0,0))==ERROR)
     printf("taskSpawn taskTwo failed\n");
if((taskIdThree=taskSpawn("task3",HIGH,0x100,20000,(FUNCPTR)taskThree,0,0,0,
0,0,0,0,0,0,0))==ERROR)
     printf("taskSpawn taskThree failed\n");
}


void taskOne(void)
{
int i,j;
for(i=0;i<ITER1;i++)
```

```
    {
    for(j=0;j<ITER2;j++)
        logMsg("\n",0,0,0,0,0,0,0)
        for(j=0;j<LONG_TIME;j++);
    }
}


void taskTwo(void)
{
int i,j;
for(i=0;i<ITER1;i++)
    {
    for(j=0;j<ITER2;j++)
        logMsg("\n",0,0,0,0,0,0,0)
        for(j=0;j<LONG_TIME;j++);
    }
}



void taskThree(void)
{
int i,j;
for(i=0;i<ITER1;i++)
    {
    for(j=0;j<ITER2;j++)
        logMsg("\n",0,0,0,0,0,0,0)
        for(j=0;j<LONG_TIME;j++);
    }
}
```

（2）在创建的 downloadable application 工程中，加入该源代码，编译生成对应的 xx.o 文件。

（3）将该文件下载到目标机中，在 Tornado 中启动 WindSh，在 windSh 命令行

提示符下，输入 sched，运行代码，观察实验结果。

4、消息队列实现任务间的通信

（1）编写源代码，使用消息队列在生产者任务和消费者任务之间实现消息通信。源代码如下：

```
/* includes */

#include "vxWorks.h"
#include "taskLib.h"
#include "msgQLib.h"
#include "msgQDemo.h"
#include "sysLib.h"
#include "stdio.h"

/* function prototypes */

LOCAL STATUS producerTask ();                /* producer task */
LOCAL STATUS consumerTask ();                /* consumer task */

/*******msgQDemo - Demonstrates intertask communication using Message
Queues ******************************************/
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);
    }
```

（2）在创建的 downloadable application 工程中，加入该源代码，编译生成对应的 xx.o 文件。

（3）将该文件下载到目标机中，在 Tornado 中启动 WindSh，在 windSh 命令行提示符下，输入 msgQDemo，运行代码，观察实验结果。

5、使用二进制信号量来实现任务间的同步

（1）编写源代码，使用二进制信号量来实现。源程序如下：

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

```
#include "semLib.h"
#include "stdio.h"
#include "sysLib.h"

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

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

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

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);
    }
```

（2）在创建的 downloadable application 工程中，加入该源代码，编译生成对应的 xx.o 文件。

（3）将该文件下载到目标机中，在 Tornado 中启动 WindSh，在 windSh 命令行提示符下，输入 synchronizeDemo，运行代码，观察实验结果。

7、使用互斥信号量来实现任务对共享资源的互斥访问

（1）编写源程序，在程序中创建一个互斥信号量用于生产者任务和消费者任务之间的互斥操作。生产者任务和消费者任务都对共享的全局数据结构访问和操作。为了避免对共享的全局数据结构的破坏，使用互斥信号量。发起消费者任

务，该任务生产消息，把它放到共享的数据结构中；发起消费者任务，它从共享数据结构中消费消息 并更新 CONSUMED 的状态域，以便生产者任务能接着将生产的消息放入共享的数据结构中。在所有的消息消费完毕后，删除互斥信号量。

源程序如下：

```
#include "vxWorks.h"
#include "semLib.h"
#include "taskLib.h"
#include "mutexSemDemo.h"
#include "logLib.h"
#include "sysLib.h"
#include "stdio.h"

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

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

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

    /* Create the mutual exclusion semaphore*/
    if ((mutexSemId = semMCreate (SEM_Q_PRIORITY | SEM_DELETE_SAFE
| SEM_INVERSION_SAFE)) == NULL)
```

```
        {
        perror ("Error in creating mutual exclusion semaphore");
        return (ERROR);
        }

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

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

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

    while (notFinished)
        taskDelay (sysClkRateGet ());

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

```
    }

  return (OK);
  }

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

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 that
consumerTask can access the global shared data structure.  */
            }
        else
            notDone = FALSE;
        }

    return (OK);
    }

/*** consumerTask -  consumes the message from the global shared data
structure and updates the status 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);
    }
```

（2）在创建的 downloadable application 工程中，加入该源代码，编译生成对应的 xx.o 文件。

（3）将该文件下载到目标机中，在 Tornado 中启动 WindSh，在 windSh 命令行提示符下，输入 mutexSemDemo，运行代码，观察实验结果。

⑤ 思考题

1、 在多任务的程序中，当对每个任务分配不同的优先级时，输出结果将是怎样的次序？解释原因。

2、 在轮转调度算法的实验中，问为什么任务 taskOne、taskTwo 和 taskThree 必须比任务 sched 的优先级低？改变调度时间片的值，重新编译、运行，并观察结果。

3、 增加第四个任务，其优先级为 80，它和其他三个任务输出相同的信息，重新编译运行并观察结果。

4、 在基于优先级抢占式调度程序中，如何修改程序，使得任务的执行顺序变为 taskOne 先执行，然后是 taskTwo，最后是 taskThree。修改程序，使得任务

taskOne 具有最高的优先级，并且它和任务 taskTwo 以相同的优先级运行，观察输出结果。