第一章 Wind River

1.1 风河系统公司简介

风河系统公司（Wind River）是全球领先的嵌入式软件与服务商，也是业界唯一提供面向行业市场的嵌入式软件平台的厂商。它所提供的嵌入式软件平台包括集成化的实时操作系统、开发工具和技术。风河的产品和专业服务已经在许多市场领域得到认可，主要包括空间技术及国防、汽车、消费电子、工业制品和网络基础设施领域。世界各地的电子设备制造商普遍把风河公司的嵌入式软件产品作为行业标准。包括 NASA 的“勇气号”火星探测器在内的航空航天设备也采用风河嵌入式软件。

1.2 实时操作系统 Vxworks 简介

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发的一种嵌入式实时操作系统（RTOS），是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一席之地。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，如卫星通讯、军事演习、弹道制导、飞机导航等。

VxWorks操作系统是实时多任务操作系统. 实时多任务操作系统能在确定的时间内执行其功能，并对外部的异步事件作出响应的计算机系统。多任务环境允许一个实时应用作为一系列独立任务来运行，各任务有各自的线程和系统资源。VxWorks系统提供多处理器间和任务间高效的信号灯、消息队列、管道、网络透明的套接字。 实时系统的另一关键特性是硬件中断处理。为了获得最快速可靠的中断响应，VxWorks系统的中断服务程序ISR有自己的上下文。

VxWorks实时操作系统由400多个相对独立的、短小精炼的目标模块组成，用户可根据需要选择适当模块来裁剪和配置系统，这有效地保证了系统的安全性和可靠性。系统的链接器可按应用的需要自动链接一些目标模块。这样，通过目标模块之间的按需组合，可得到许多满足功能需求的应用。

VxWorks操作系统的基本构成模块包括以下部分：

- 高效的实时内核Wind

VxWOrks实时内核（Wind）主要包括基于优先级的任务调度、任务同步和通信、中断处理、定时器和内存管理。

- 兼容实时系统标准POSIX

VxWOrks提供接口来支持实时系统标准P.1003.1b.

- I/O系统

VxWOrks提供快速灵活的与ANSI-C相兼容的I/O系统，包括UNIX的缓冲I/O和实时系统标准POSIX的异步I/O。VxWOrks包括以下驱动：

网络 --- 网络设备（以太网、共享内存）

管道 --- 任务间通信

RAM --- 驻留内存文件

SCSI --- SCSI硬盘，磁碟，磁带

键盘 --- PC x86键盘（BSP仅支持x86）
显示器 --- PC x86显示器（BSP仅支持x86）
磁碟 --- IDE和软盘（BSP仅支持x86）
并口 --- PC格式的目标硬件

- 本机文件系统
- I/O系统

VxWorks的文件系统与MS-DOS、RT-11、RAM、SCSI等相兼容。

- 网络特性

VxWorks网络能与许多运行其它协议的网络进行通信，如TCP/IP、4.3BSD、NFS、UDP、SNMP、FTP等。VxWorks可通过网络允许任务存取文件到其它系统中，并对任务进行远程调用。

- 虚拟内存（可选单元VxVMI）

VxVMI主要用于对指定内存区的保护，如内存块只读等，加强了系统的健壮性。

- 共享内存（可选单元VxMP）

VxMP主要用于多处理器上运行的任务之间的共享信号量、消息队列、内存块的管理。

- 驻留目标工具

Tornado集成环境中，开发工具工作于主机侧。驻留目标外壳、模块加载和卸载、符号表都可进行配置。

- Wind基类

VxWorks系统提供对C++的支持，并构造了系统基类函数。

- 工具库

VxWorks系统向用户提供丰富的系统调用，包括中断处理、定时器、消息注册、内存分配、字符串转换、线性和环形缓冲区管理，以及标准ANSI-C程序库。

- 性能优化

VxWorks系统通过运行定时器来记录任务对CPU的利用率，从而进行有效地调整，合理安排任务的运行，给定适宜的任务属性。

- 目标代理

目标代理可使用户远程调试应用程序。

- 板级支持包

板级支持包提供硬件的初始化、中断建立、定时器、内存映象等。

- VxWorks仿真器（VxSim）

可选产品VxWorks仿真器，能模拟VxWorks目标机的运行，用于应用系统的分析。

第二章 Tornado

Tornado 2.2 是嵌入式实时领域里最新一代的开发调试环境。。Tornado 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 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。 在集成开发环境方面， VxWorks Tornado II 型提供了二次开发途径。集成开发环境是一种更直观的自动化环境，使得不同使用经验的开发人员可以快速、方便地在 VxWorks 上面开发应用

2.1 安装开发环境

2.2 Tornado 集成开发环境简述

Tornado 集成开发环境使用户创建和管理工程、建立和管理宿主机与目标机之间的通信以及运行、调试和监控 Vxworks 应用变得非常方便。

Tornado 集成开发环境具有如下的主要特点：

- 集成源码级编辑器；
- 工程管理工具；
- 包含 C 和 C++编译及 make 工具；.
- 包含目标机系统状态浏览器——Browser;
- 包含图形化源码级交叉调试器——CrossWmd;
- 包含 C 语言命令 shell 工具——WindSh；
- 包含 VxWorks 仿真器——Vxsim;
- 包含目标机软件逻辑分析仪——WmdView;
- 具有用户定制功能，包括编辑器和配置工具的定制，Tornado 本身图形用户界面的定制。

2.2.1 Tomado 编辑器

Tornado 源码级编辑器具有以下特点：

- 标准的文档处理功能；
- C 和 C++语法关键字的突出显示；
- 调试程序时追踪代码执行；
- 编译连接程序时错误及警告信息显示。
-

2.2.2 工程管理

Tornado 工程管理工具使得组织、配置、构造 VxWorks 应用变得简单化。这些处理过程是以图形化方式来完成，如图 2.1 所示的组织、配置及构造应用工程。构造 VxWorks 包括配置编译、连接选项等操作通过选择图 2.2 所示对话框来实现。

2.2.3 编译

Tornado 包含 GNU 编译器以及一系列完整的开发工具，具体如下：

- cpp，C 预处理程序；
- gcc，C 和 C++编译器；
- make，构造程序自动操作工具；
- ld,目标代码连接器；
- 有效的二进制文件；

这些工具都是自由软件，来源于自由软件组织（Free Software Foundation. FSF)。Wind River 公司经过测试和验证把它们集成到 Tornado 开发环境中，在这里不作详细介绍，更多的信息请阅读《Tornado user's guide》、《GNU ToolKit User's Guide》和《GNU Make》等书籍。

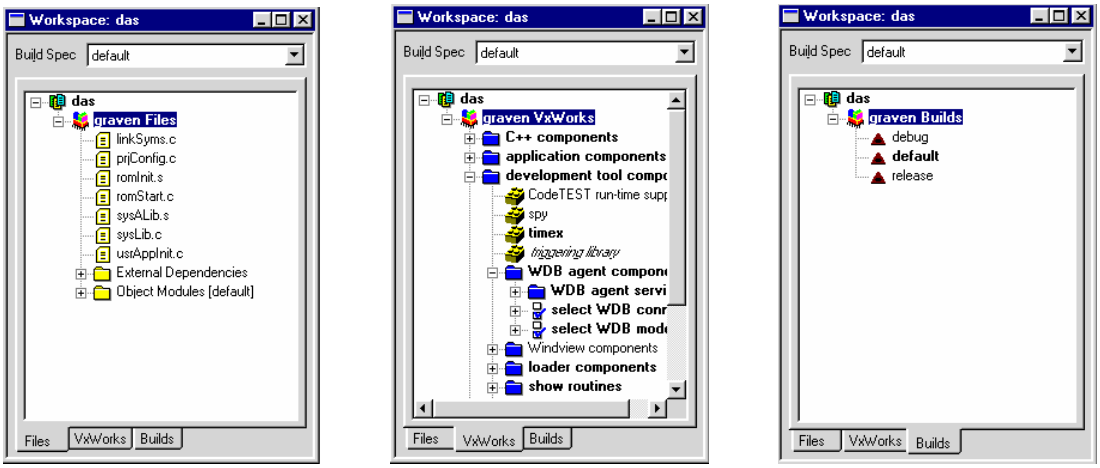


图 2.1 组织、配置及构造应用程序

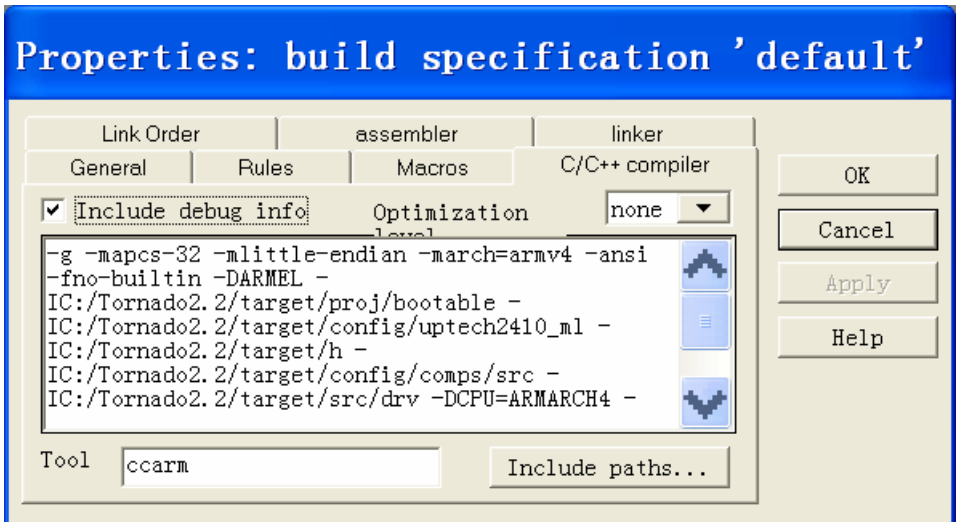


图 2.2 编译、连接选项配置

2.2.4 目标机系统状走浏览器 Browser

Browser 是 Tornado 命令解释器 WindSh 的图形化工具。与命令解释器一样，它可以在需要的时候显示目标机信息。

Browser 的主窗口显示目标系统的整个状态，它允许用户请求显示个别的目标机操作系统对象的状态，例如任务、信号量、消息队列、内存块和看门狗定时器。这些显示会根据用户命令更新或者按照开发者的设定周期性地更新。

2.2.5 文叉调试器——Crosswind

源码级调试器 CrossWind 提供所有调试功能的图形化和命令行的接口。功能包括符号反汇编、任务或系统断点、单步、系统状态显示和异常处理。

CrossWind 是 GNU 源码级调试器（GDB）的一个扩充版本。最明显的对 GDB 的扩充就是图形接口。另外 CrossWind 提供了许多 Tcl 脚本接口，允许开发者创建熟悉的调试器宏或者为自己的需求扩充 CrossWind。为了最大的灵活性，CrossWind 的命令接口集成了 GDB 的命令行接口和 WindSh(Tornado 命令解释器）的功能。

在宿主机上使用 Crosswind，可以创建、运行和调试目标机运行系统中的任务。Crosswind 也可以捕获已经由应用程序、Tornado 命令解释器或者目标机驻留的命令解释器创建、运行的任务。

Crosswind 支持应用程序断点和系统断点。开发者可以使用它在 c 和 C++源语言级、汇编语言级或混合模式下调试应用程序代码。

Crosswind 除了可以与 Tornado 目标机代理通信，也可以与为基于目标机的工具配置过的目标机直接通信。

2.2.6 C 语言命令 shell 工具 WindSh

Tornado 统一的命令解释器接口 windSh 允许用户与目标机的组件相互作用。不同于其他的"shell”工具，Tornado 命令解释器提供了一个简单、强大的功能：它能解释和执行几乎所有的 C 语言表达式，包括函数的调用和名字在系统符号表中的变量引用。被解释的 C 声明提供了到目标环境的易于使用的接口和有用的调试性能。可以引发装载到内存中的任何子例程，包括 VxWorks 系统和应用模块，这样可以加快代码生成和增量代码开发。通过动态函数的执行实现的动态测试可以加快开发。命令解释器的符号调试使下面的操作更简易：

- 任务断点；
- 任务单步；
- 符号反汇编；
- 符号产生和变量观察；
- 内存观察和修改；
- 异常陷入；
- 任务跟踪；
- 堆栈检查。’

通过在命令解释器创建运行任务、删除任务、挂起和唤醒任务可以实现性能分析和调试，报告系统信息和任务信息。

2.2.7 VxWorks 仿真器——VxSim

VxSim 是一个全面的 VxWorks 原型仿真器。提供与真实目标机一致的调试和仿真运行环境。支持 CrossWind、WindView, Browser 等工具。

VxSim 作为 Tornado 核心工具包含在软件包中，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，基于 vxSim 迅速开始软件开发工作，缩短开发时间，提高开发效率。

2.2.8 目标机软件逻辑分析——WindView

WindView 是一个图形化的可视诊断和分析用户目标机系统的工具，使用户非常容易地观察任务、中断程序之间的相互作用，它是在嵌入式系统应用开发期间，前所未有的可视工具。

WindView 主要是向开发者提供目标机硬件上应用程序实际运行的详细情况。这种系统级的诊断分析工具可以与 VxSim 一起使用。

通常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，而这种全功能版本的 WindMew 提供了 VxWorks 应用程序的详细动态行为，图形化显示了任务、中断和系统对象相互作用的复杂关系。

2.2.9 用户定制功能

Tornado 集成开发环境是一个开放的环境，它使用户可以根据自己的需要对 Tornado 进行定制。这种定制主要包括三个方面：

（1）Tornado 工具选项设置；

（2）定制 Tornado tools 菜单；

（3）通过 TCL 语言定制 Tornado 环境.

2.3 一个基于 VxSim 的简单工程

2.3.1 开始 tornado

选择开始－>程序－>Tornado2.2,然后点击 tornado 图标，启动 Tornado 以后，是 Tornado 的主窗口，是带有创建工程的默认窗口。如图 2.3

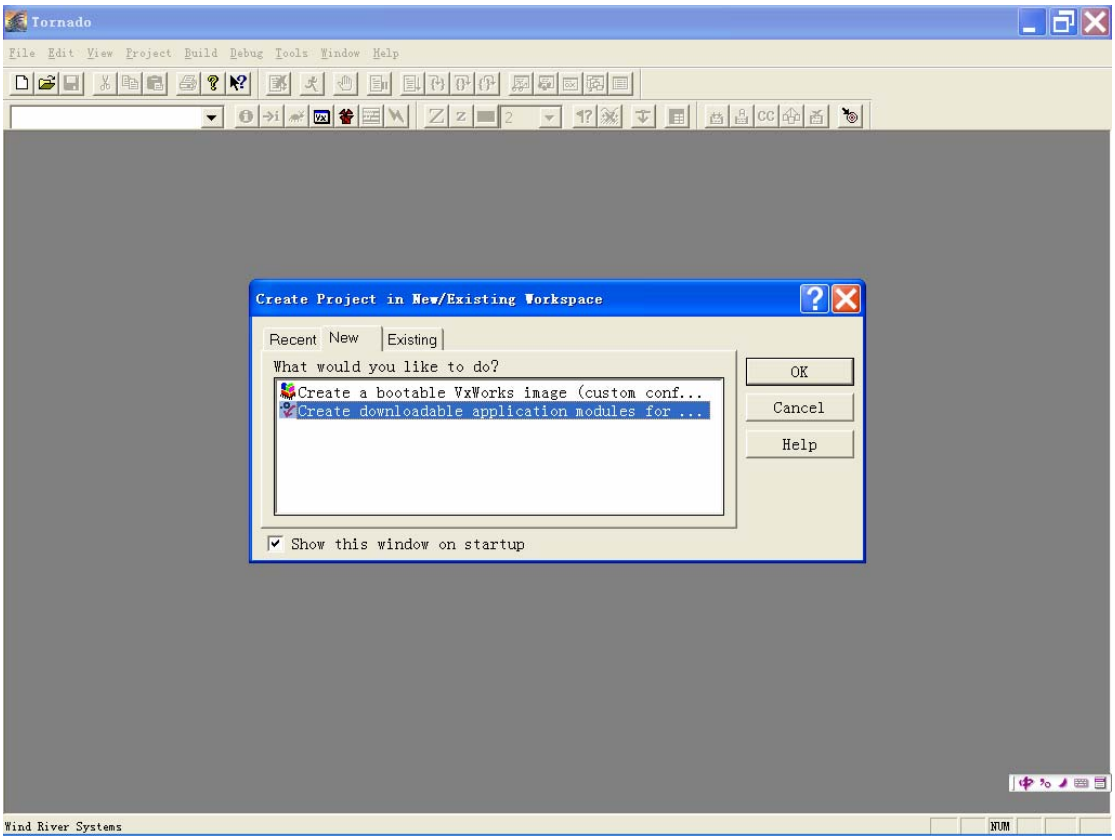


图 2.3 创建工程

2.3.2 创建工程

选择图 2.3 中的 creat downloadable application modules for VxWorks,点击 ok，创建可下载工程，如图 2.4,按照向导的提示做如下修改：

- 工程名是 gizmo
- 工程目录是 c:\projects\gizmo.
- 工程描述是 lightning gizmo
- 工程路径和文件名是 c:\projects\lightning.wsp

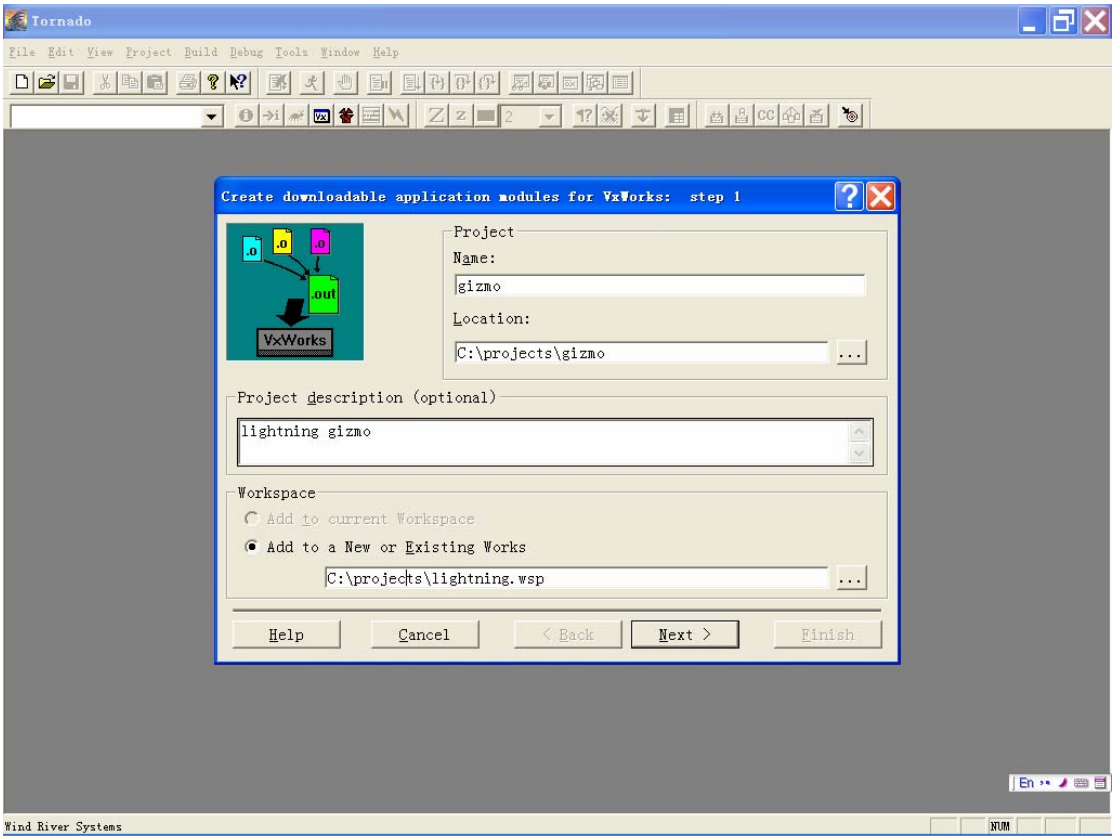


图 2.4

点击如图 2.4 中的 next>进入下一步向导如图 2.5,为工程选择工具链，从下拉列表中选择 SIMNTgnu，点击 next>进入确认选择界面图 2.6，点击 finish 出现工作空间窗口如图 2.7

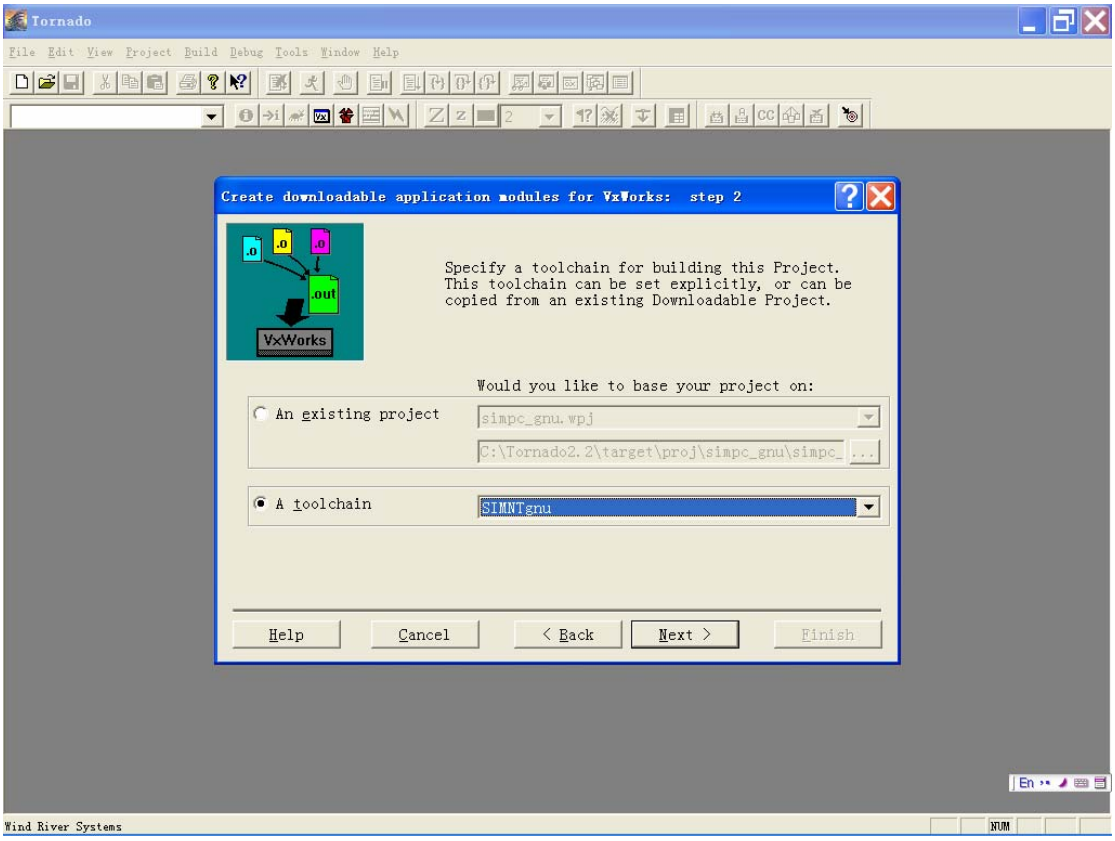

Tornado
File Edit View Project Build Debug Tools Window Help
Create downloadable application modules for VxWorks: step 2
.o
.o
.o
.out
VxWorks
Specify a toolchain for building this Project.
This toolchain can be set explicitly, or can be
copied from an existing Downloadable Project.
Would you like to base your project on:
An existing project
simpc_gnu.wpj
C:\Tornado2.2\target\proj\simpc_gnu\simpc_
A toolchain
SIMNTgnu
Help
Cancel
< Back
Next >
Finish
Wind River Systems
NUM

图 2.5

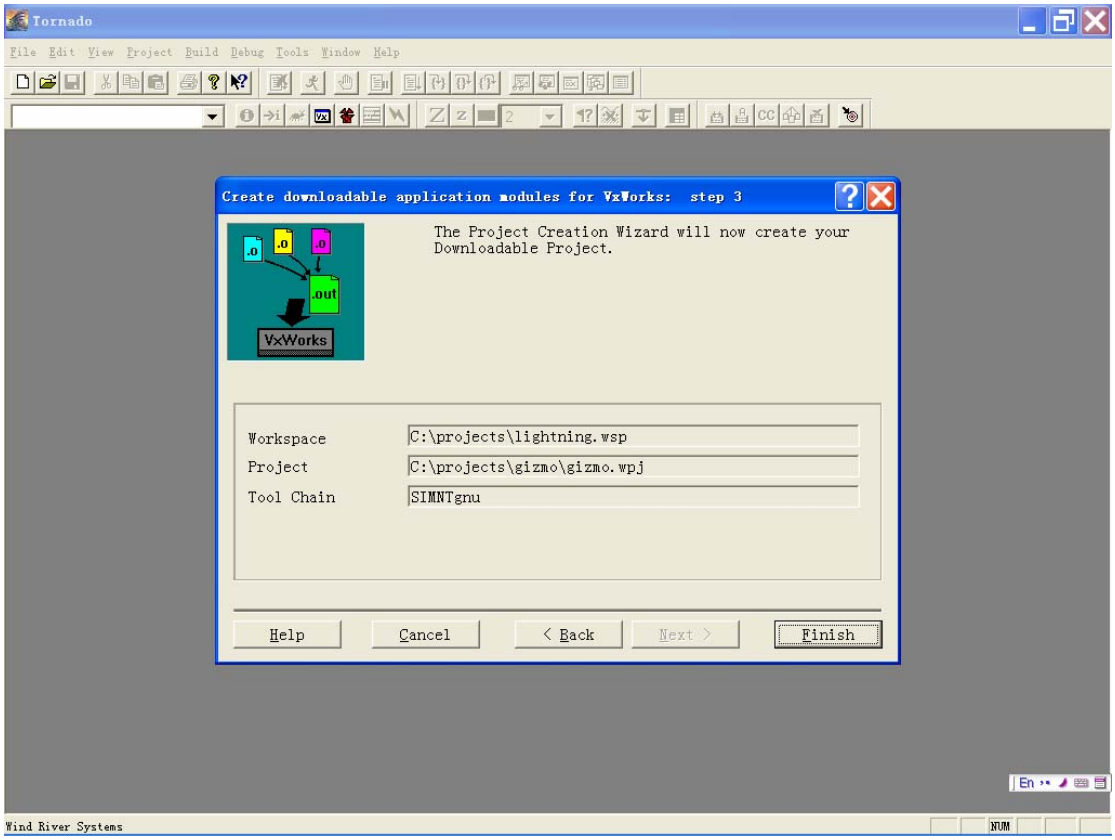

Tornado
File Edit View Project Build Debug Tools Window Help
Create downloadable application modules for VxWorks: step 3
.o
.o
.o
.out
VxWorks
The Project Creation Wizard will now create your
Downloadable Project.
Workspace
C:\projects\lightning.wsp
Project
C:\projects\gizmo\gizmo.wpj
Tool Chain
SIMNTgnu
Help
Cancel
< Back
Next >
Finish
Wind River Systems
NUM

图 2.6

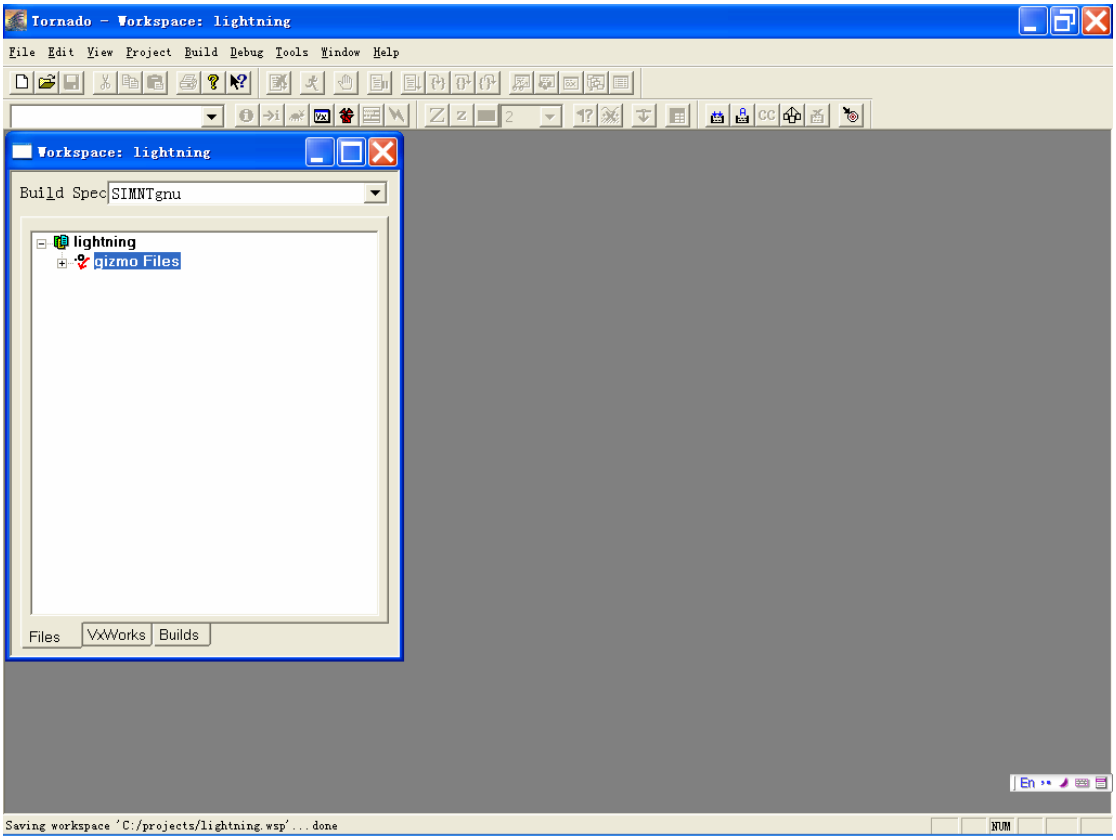


图 2.7

2.3.3 向工程加例子源代码

例子的程序是 cobble.c,这是一个简单的多任务应用程序，模拟采集系统数据，数据来源于外部设备，cobble.c 的四个任务如下：

- tCosmos, 模拟中断服务程序（ISR）产生新的数据。
- tSchlep, 采集数据。
- tCrunch, 计算数据获得结果值。
- tMonitor, 监控结果,如果超出安全值打印警告信息。

Cobble 程序包括 progStart( ) 和 progStop( )，用来开始和停止程序。把例子源码复制到你建工程的目录，例子源码在安装目录\target\src\demo\start，复制完例子程序以后将其加入到工程中，添加方法：如图 2.8，在工作空间的 file 窗口中选择 gizmofiles 点击右键选择 Add files，将文件添加到工程以后如图 2.9

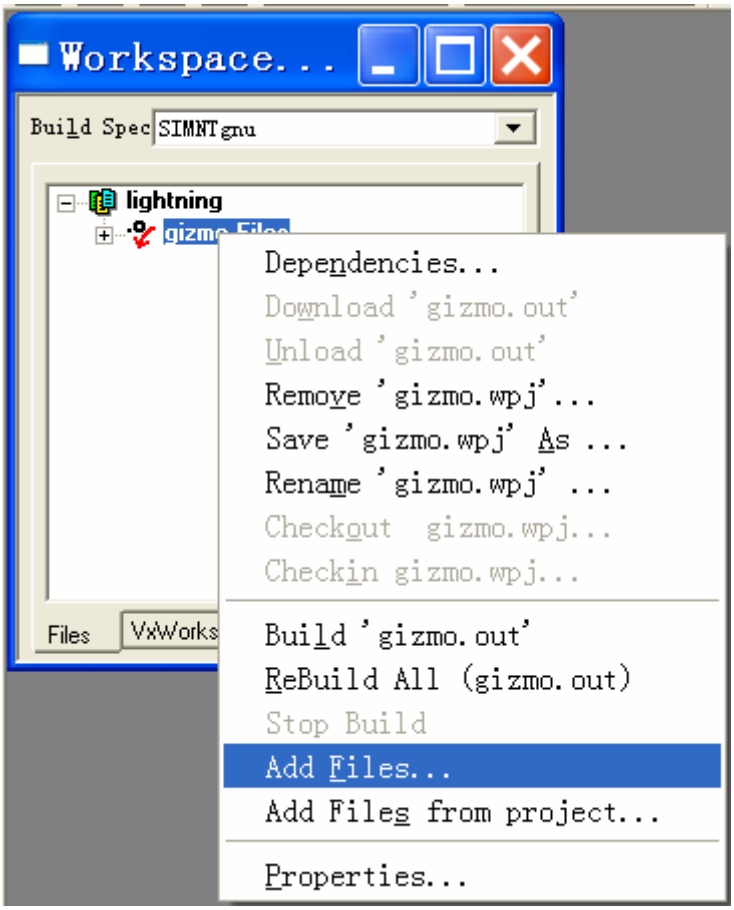

Workspace...
Build Spec SIMNTgnu
lightning
Dependencies...
Download 'gizmo.out'
Unload 'gizmo.out'
Remove 'gizmo.wpj'...
Save 'gizmo.wpj' As ...
Rename 'gizmo.wpj' ...
Checkout gizmo.wpj...
Checkin gizmo.wpj...
Build 'gizmo.out'
ReBuild All (gizmo.out)
Stop Build
Add Files...
Add Files from project...
Properties...
Files
VxWorks

图 2.8

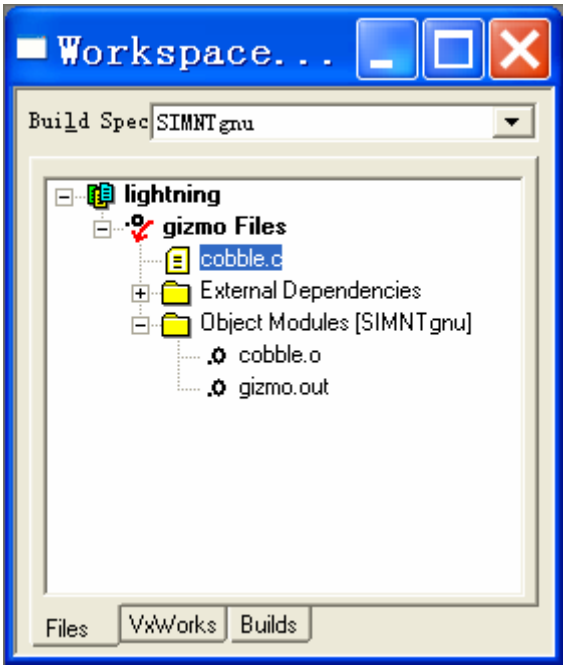

Workspace...
Build Spec SIMNTgnu
lightning
gizmo Files
cobble.c
External Dependencies
Object Modules [SIMNTgnu]
cobble.o
gizmo.out
Files
VxWorks
Builds

图 2.9

2.3.4 编译工程

为了查看在编译工程时的默认设置，在工作空间选择build标签如图3.0，然后双击SIMNTgnu出现编译属性菜单如图 3.1，可以查看 makefile 规则和宏，编译器，汇编器和连接器选项。

图 3.0

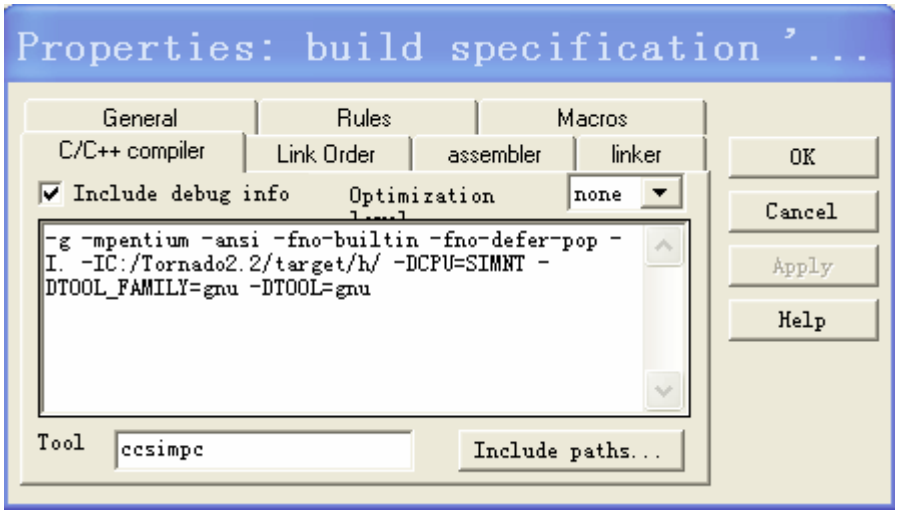


图 3.1

在 C/C++编译器选项可以看到默认包括调试信息，这可以确保工程在编译的时候优化选项是关闭的。点击 cancel 关闭属性窗口。从上下文菜单选择 Build ‘gizmo.out’ 编译工程，如图 3.2。这样编译所有工程模块，编译好的模块可以下载到目标板运行。Tornado 在编译工程之前，依赖关系对话诓会警告 cobble.c 的 makefile 没有形成依赖，如图 3.2.点击 ok 继

续。Tornado 会计算 makefile 的依赖关系并编译这个工程，如果发现外部的依赖关系就会自动包含在工程中并且在工作空间的文件查看窗口的External Dependencies 文件夹列出如图 3.3,同时在编译输出窗口会显示错误和警告，在这个工程中没有出现错误和警告如图 3.4,关闭编译输出窗口。

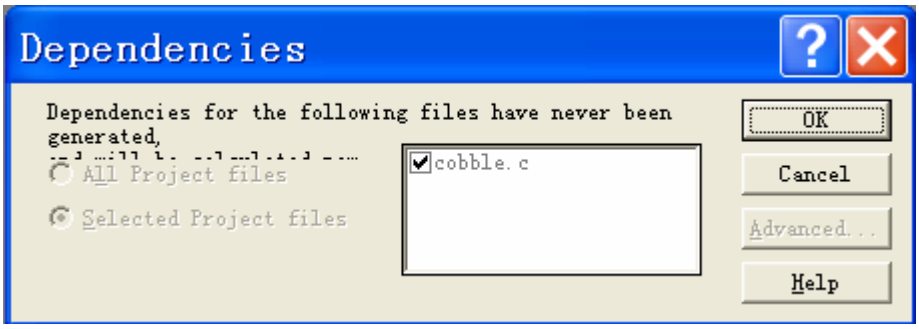


图 3.2

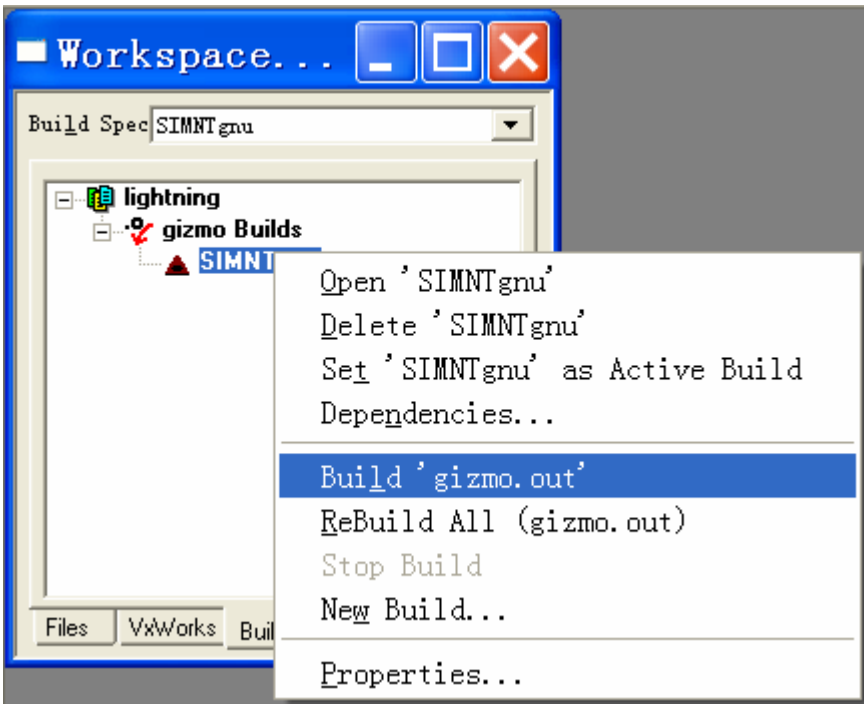


图 3.2

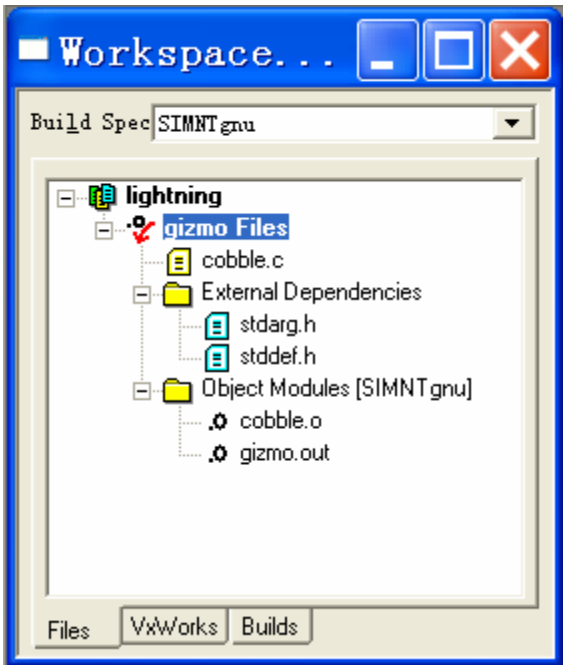


图 3.3

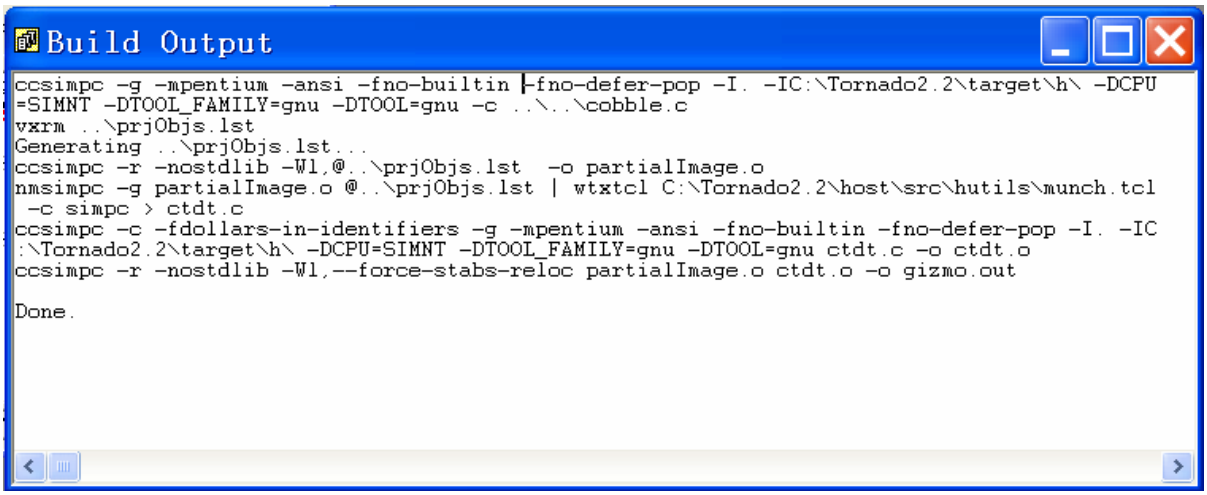


图 3.4

2.3.5 下载工程到 vxworks 目标模拟器

从工作空间的文件查看窗口下载工程如图 3.5 开始模拟器。Tornado 会提示开始模拟器如图 3.6, 点击 ok 继续会出现 VxSim 对话框如图 3.7.
使用默认的设置点击 ok 出现目标模拟器窗口如图 3.8。Tornado 会提示开始一个 Target Server，点击 ok 继续。若要关闭模拟器点击模拟器的窗口。

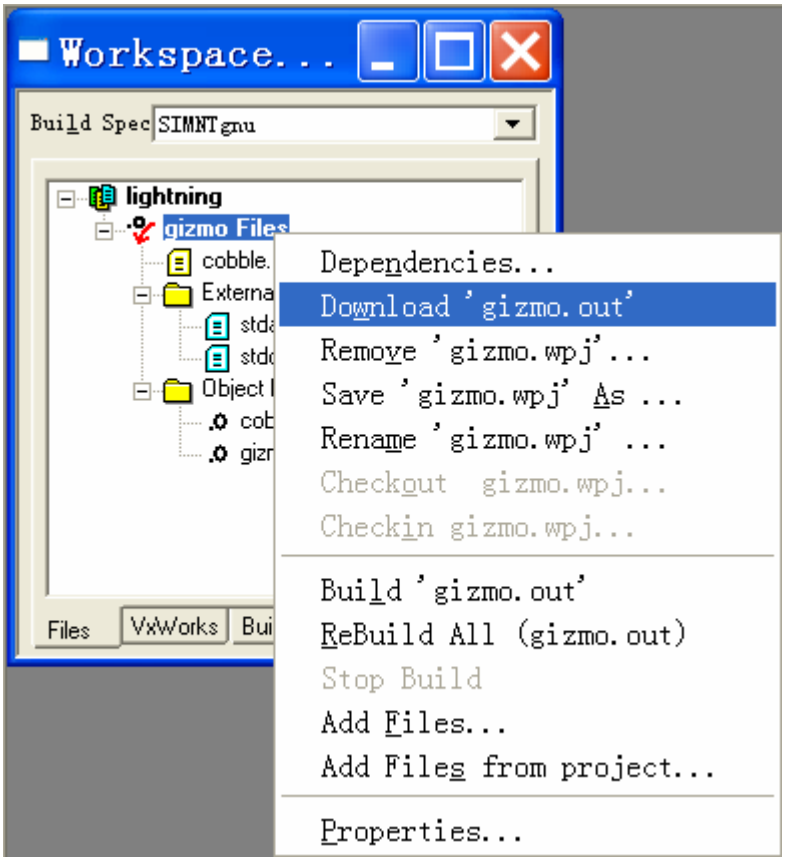

Workspace...
Build Spec SIMNTgnu
lightning
gizmo Files
cobble.
Externa
Object
Files
VxWorks
Dependencies...
Download 'gizmo.out'
Remove 'gizmo.wpj'...
Save 'gizmo.wpj' As ...
Rename 'gizmo.wpj' ...
Checkout gizmo.wpj...
Checkin gizmo.wpj...
Build 'gizmo.out'
ReBuild All (gizmo.out)
Stop Build
Add Files...
Add Files from project...
Properties...

图 3.5

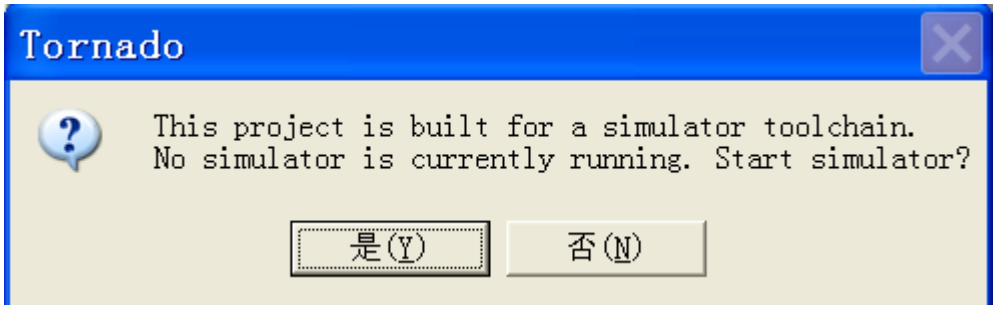

Tornado
This project is built for a simulator toolchain.
No simulator is currently running. Start simulator?
是(Y)
否(N)

图 3.6

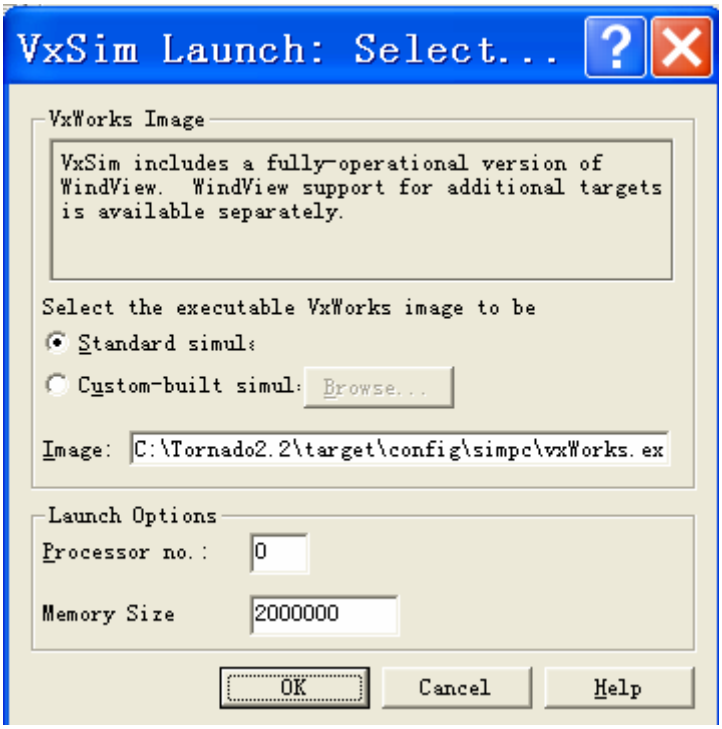


图 3.7

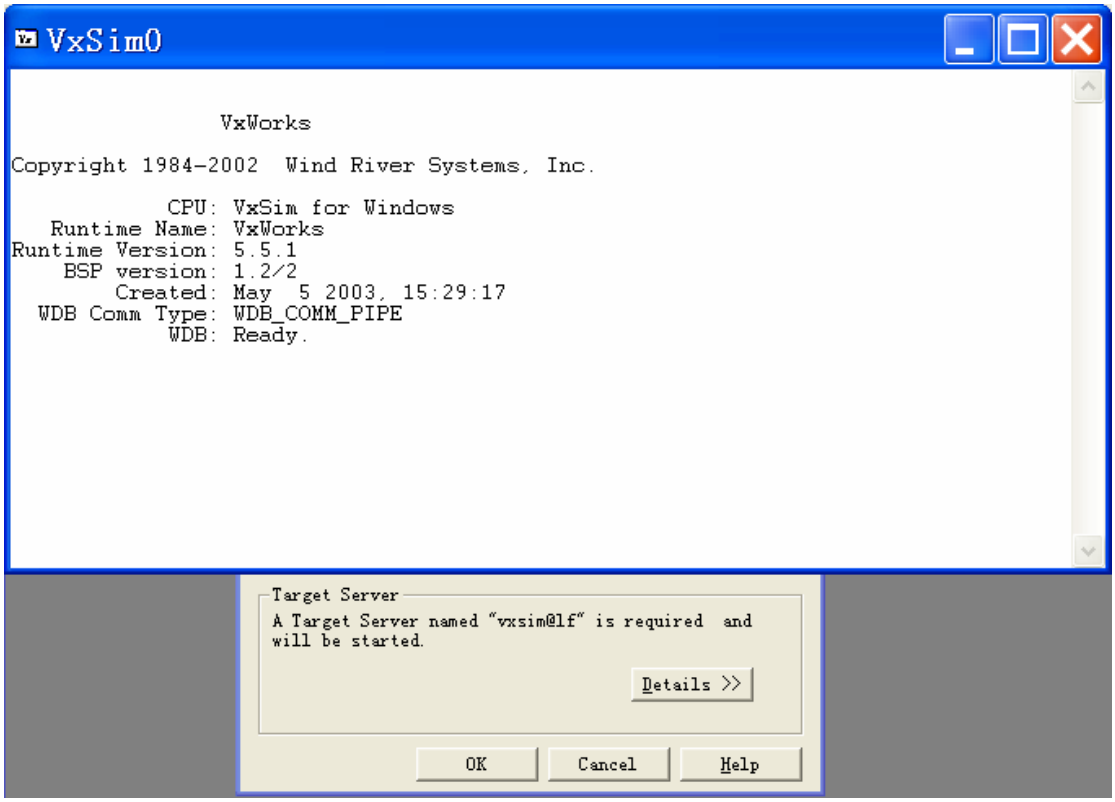


图 3.8

Target server 管理着 Tornado 主机工具（例如 debugger）和目标机的通信，习惯上目标服务器的名子是 targetName@hostName。这里目标名是 vxsin（模拟器默认），pc 机的名字是

lf。目标服务器的名字显示在 Tornado 开始工具条如图 3.9,这个工具条包括 browser, shell 和 debugger.从目标服务器开始以后这些按钮就可以使用了。带有—>的 i 是 shell 命令，令一个 i 是用来查看目标信息。

图 3.9

2.3.6 从 Tornado Shell 运行应用程序

运行应用程序之前配置 Tornado debugger 可以使 debugger 对程序的异常自动作出反映。Tornado debugger（CrossWind）提供两种调试接口：图形接口和命令行接口。常用的调试手段如设置断点和控制程序异常等用图形接口实现方便。

如果要配置 debugger，从主 Tornado 窗口选择 Tools>Options>Debugger，当 Options 对话框出现时在 Auto attach to tasks 下选择 Always，如图 3.10,这样当异常发生时，debugger 将会自动连接任务。点击 ok 继续，更多的信息参考 Tornado User's Guide: Debugger。

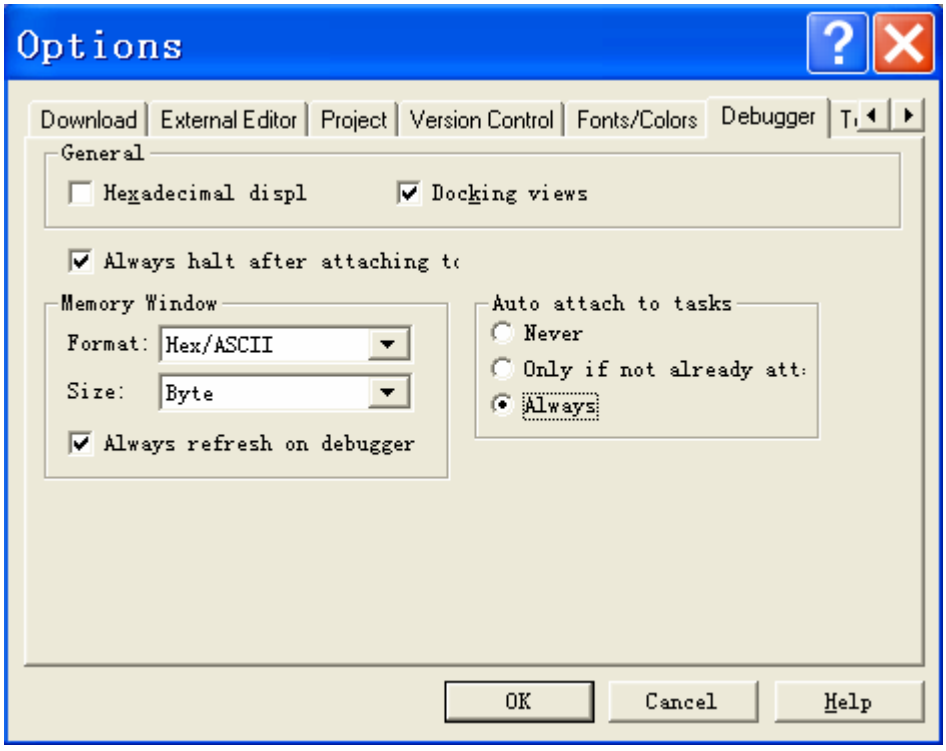


图 3.10

点击 debugger 按钮  开始 debugger。在主 Tornado 窗口的下面的状态行显示 debugger 正在运行如图 3.11.

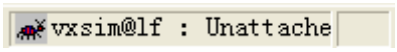


图 3.11

Tornado shell(也叫 WindSh)是 c 语言的命令解释程序，用它可以在 shell 命令行调用任何下载到目标板上的程序。Shell 含有一套自己的命令用于管理任务、访问系统信息、调试等等。

用 shell 命令行运行程序前要点击 shell 按钮  ，当 shell 窗口打开以后在命令行键入主程序的名字 progStart 运行程序。如图 3.12.

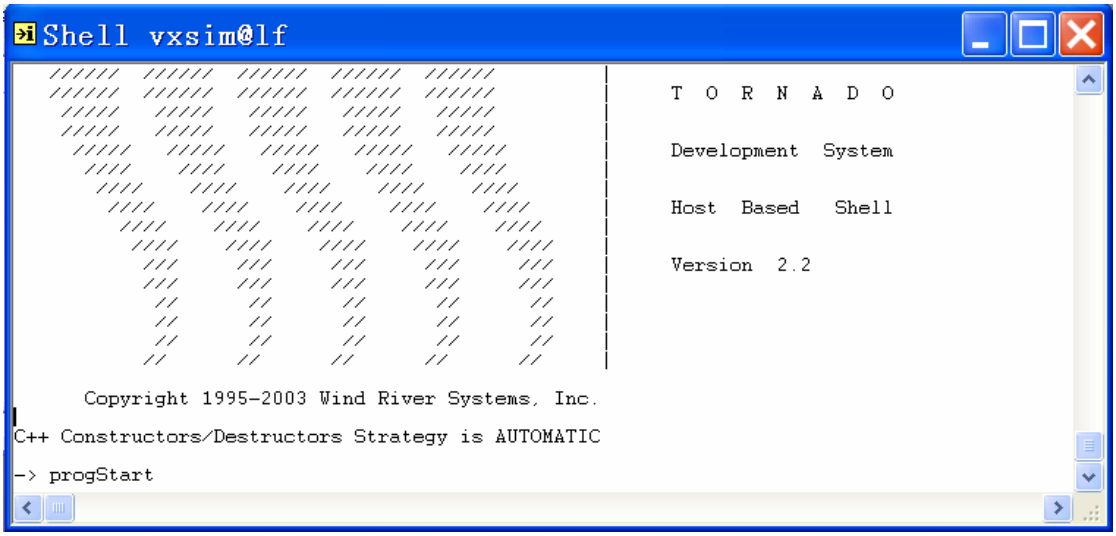


图 3.12

2.3.7 查看目标内存使用情况

Tornado browser 用来查看系统目标，它提供显示设备来监控目标系统的状态如内存分配等。点击 browser 按钮  开始 browser，当 browser 出现的时候，从下拉列表选择 Memory Usage 并且点击定时刷新按钮  如图 3.13,这样就会几秒钟更新一次显示，查看 bobble.c 的内存使用变化情况。用标题栏的窗口控制 browser 的关闭。

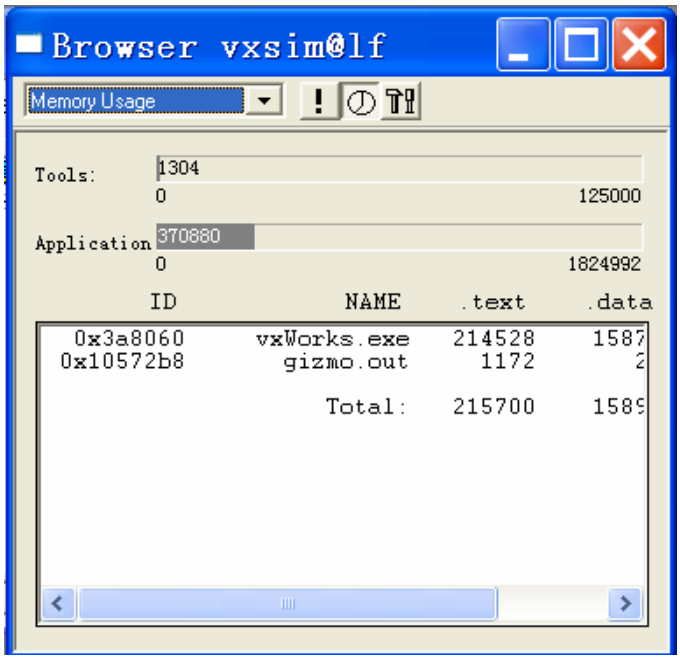


图 3.13

2.3.8 查看任务

WindView 是一个用于实时应用程序的 Tornado 逻辑分析器，它可以动态的反映上下文切换、事件以及信号量、消息队列和看门狗定时器的信息。

点击 WindView 按钮出现 WindView Control 窗口，同时还有 Collection Configuration 对话框。在对话框中从下拉列表选择 Additional Instrumentaion 点击 ok，点击 GO 按钮开始数据的收集。如图 3.14.等几秒钟点击更新按钮更新数据收集的状态。当在 Current

Content 下的 Buffers 值大于等于 2 时点击停止按钮结束数据收集。

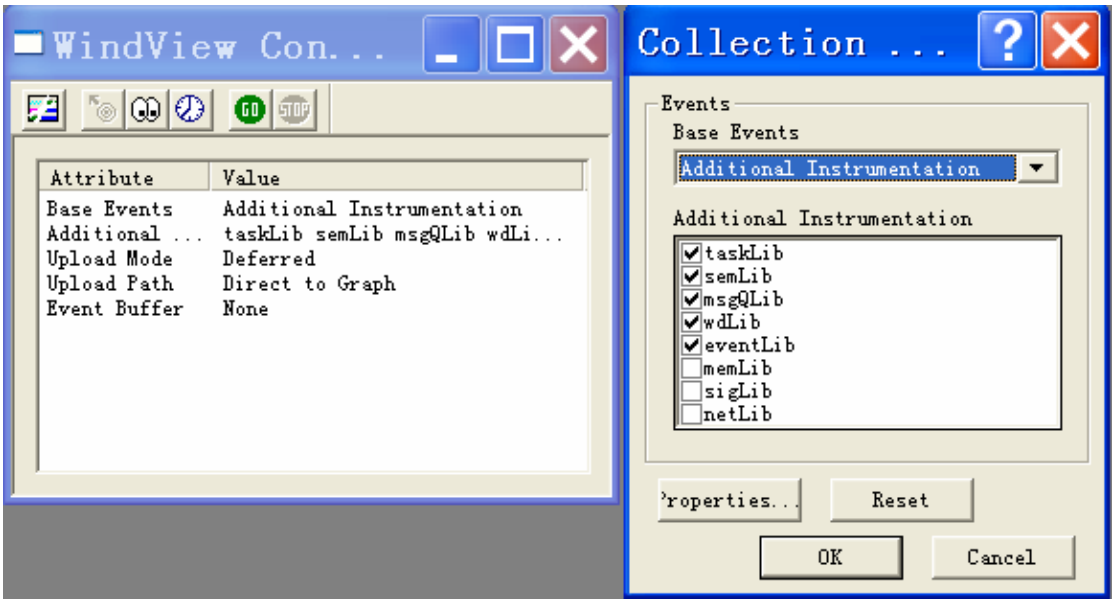


图 3.14

在上传模拟器目标的 WindView 数据到主机前，在 shell 命令行键入 progStop 停止采样程序。然后按上传按钮，当数据上传以后出现一个图标，当最大化以后如图 3.15,可以使用缩放按钮来放大和缩小显示的区域。(按钮一屏显示所有的数据)。图标表示信号的获取与释放，水平线表示任务的状态（执行，挂起，就绪等)。其他的 WindView 图标的使用参见 help—>WindView help—>Legend。

注意：tCrunch 任务负责处理数据和移除连接列表的节点，从不运行。具体参见代码。

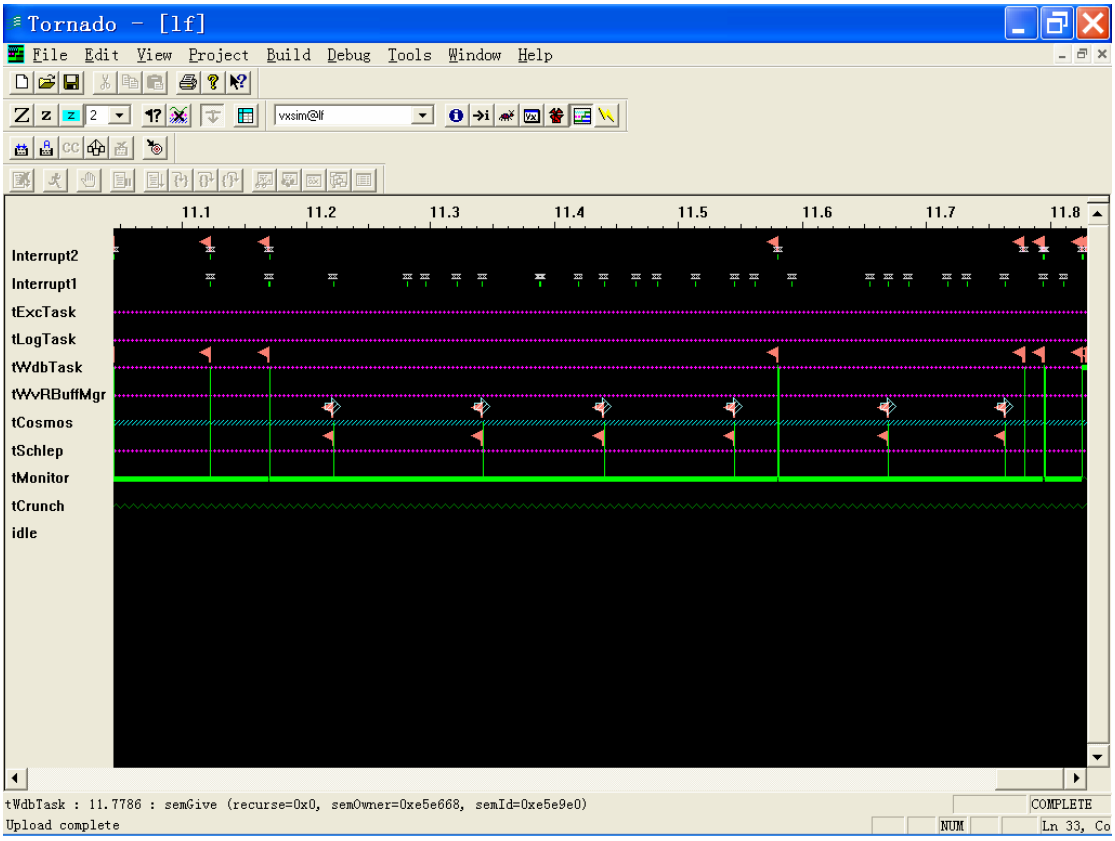


图 3.15

2.3.9 修改任务的优先级和查找错误

在工作空间的 Files 窗口双击 cobble.c 打开源码。找到 progStart()程序，这里 tCrunch 分配的优先级（240）比 tMonitor 的优先级（230）低，因此 tCrunch 任务从不运行。编辑源码改变任务 tCrunch 的优先权到 230，tMonitor 的优先级到 240，就是将两个任务的优先级颠倒。然后保存文件。在工作空间 Files 窗口用上下文菜单选择 ReBuild All（gizmo.out）选项重新编译工程，编译结束以后，关闭编译输出窗口。用上下文的 Download‘gizmo.out’选项下载工程到目标板。从 Tornado
Shell 用命令 progStart 再次开始运行程序。模拟器和 shell 窗口会声明异常和。。。。。 如图 3.16

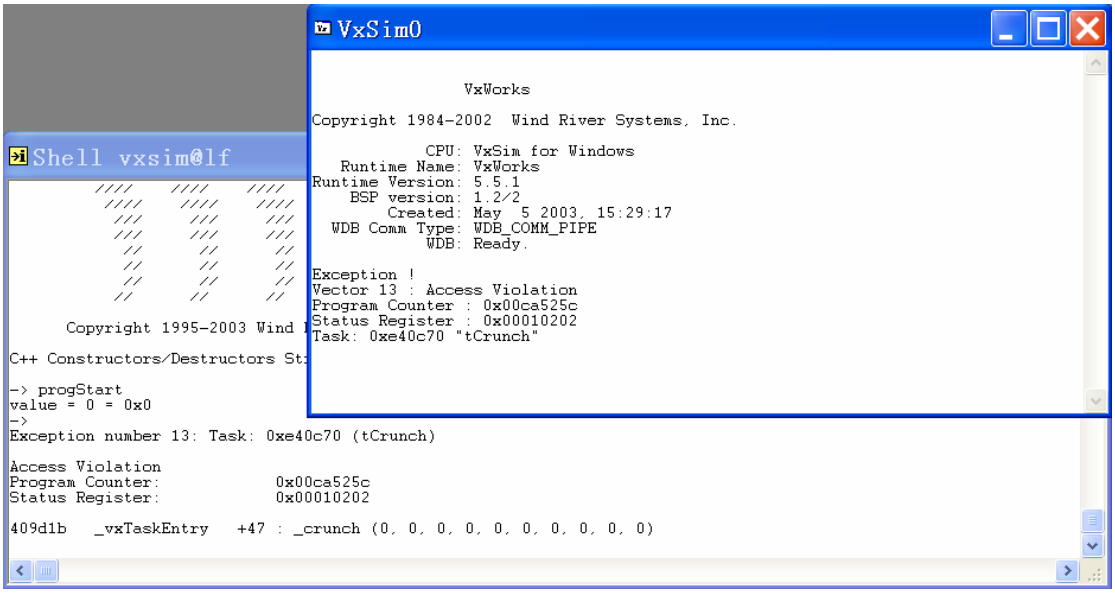

VxSim0
VxWorks
Copyright 1984-2002 Wind River Systems, Inc.
CPU: VxSim for Windows
Runtime Name: VxWorks
Runtime Version: 5.5.1
BSP version: 1.2/2
Created: May 5 2003, 15:29:17
WDB Comm Type: WDB_COMM_PIPE
WDB: Ready.
Exception !
Vector 13 : Access Violation
Program Counter : 0x00ca525c
Status Register : 0x00010202
Task: 0xe40c70 "tCrunch"
Shell vxsim@lf
Copyright 1995-2003 Wind
C++ Constructors/Destructors St
-> progStart
value = 0 = 0x0
->
Exception number 13: Task: 0xe40c70 (tCrunch)
Access Violation
Program Counter: 0x00ca525c
Status Register: 0x00010202
409d1b _vxTaskEntry +47 : _crunch (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

图 3.16

第三章 基本工程实践

3.1 Bootable 工程实践

首先把 bsp 的源码文件夹复制到安装目录\target\config 下，这样在建 bootable 工程的时候就能找到这个 bsp 了，然后启动 Tornado 方法如前面 vxsim 工程。不同的是在创建工程的页面选择 Creat a bootable Vxworks image 如图 3.17,点击 ok 继续，出现图 3.18，工程的名字、位置、和工作空间如图所示。点击 next，出现图 3.19,选择 A BSP，从下拉列表中选择 uptech2410，就是我们复制的那个 bsp，Tool 选择 gnu，点击 next 继续，最后点击 finish 完成出现图 3.20，

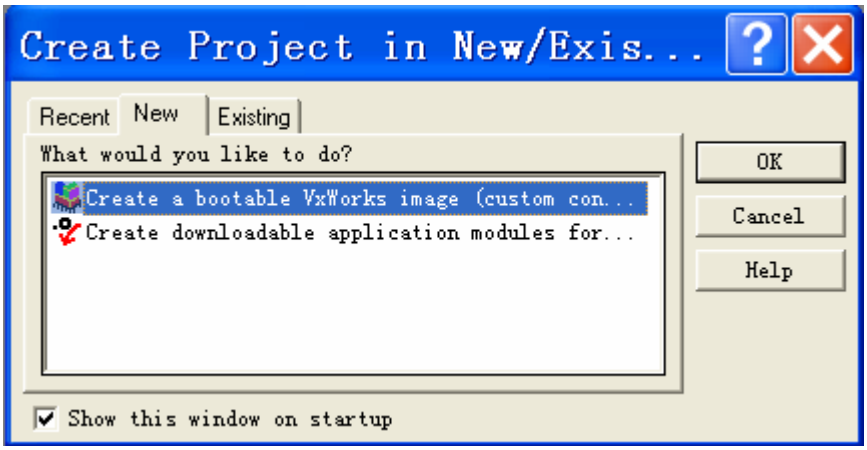


图 3.17

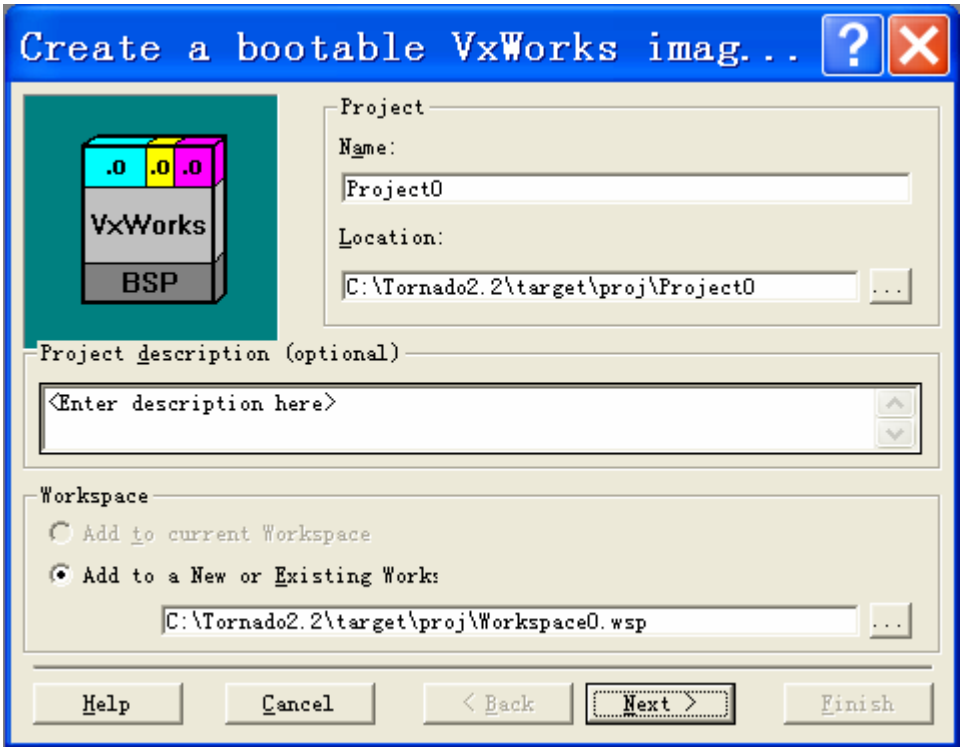


图 3.18

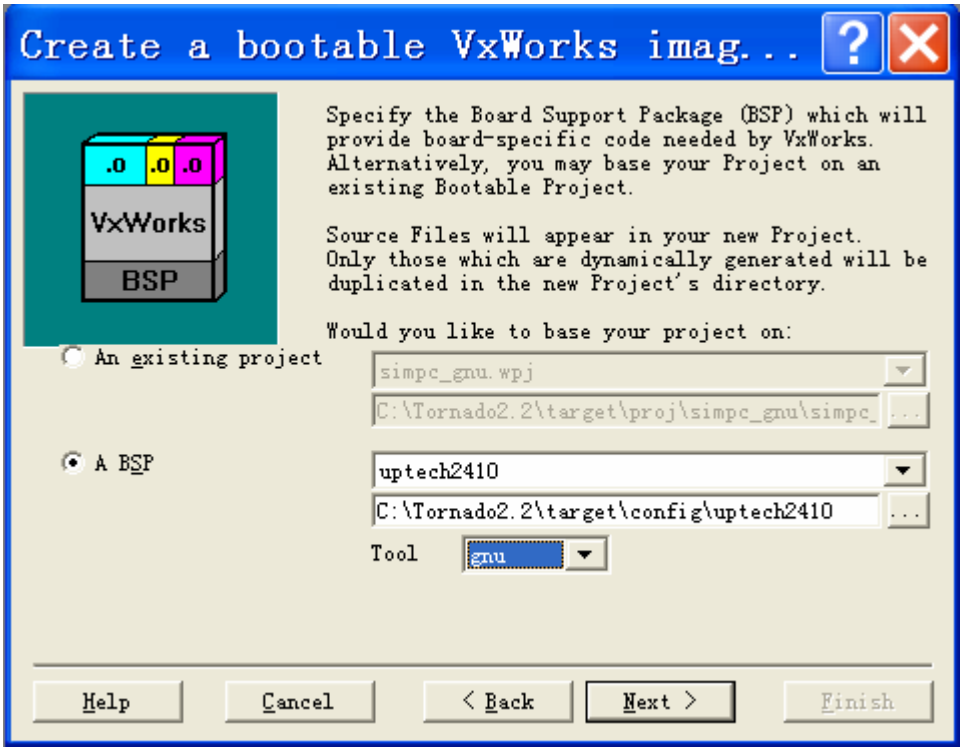


图 3.19

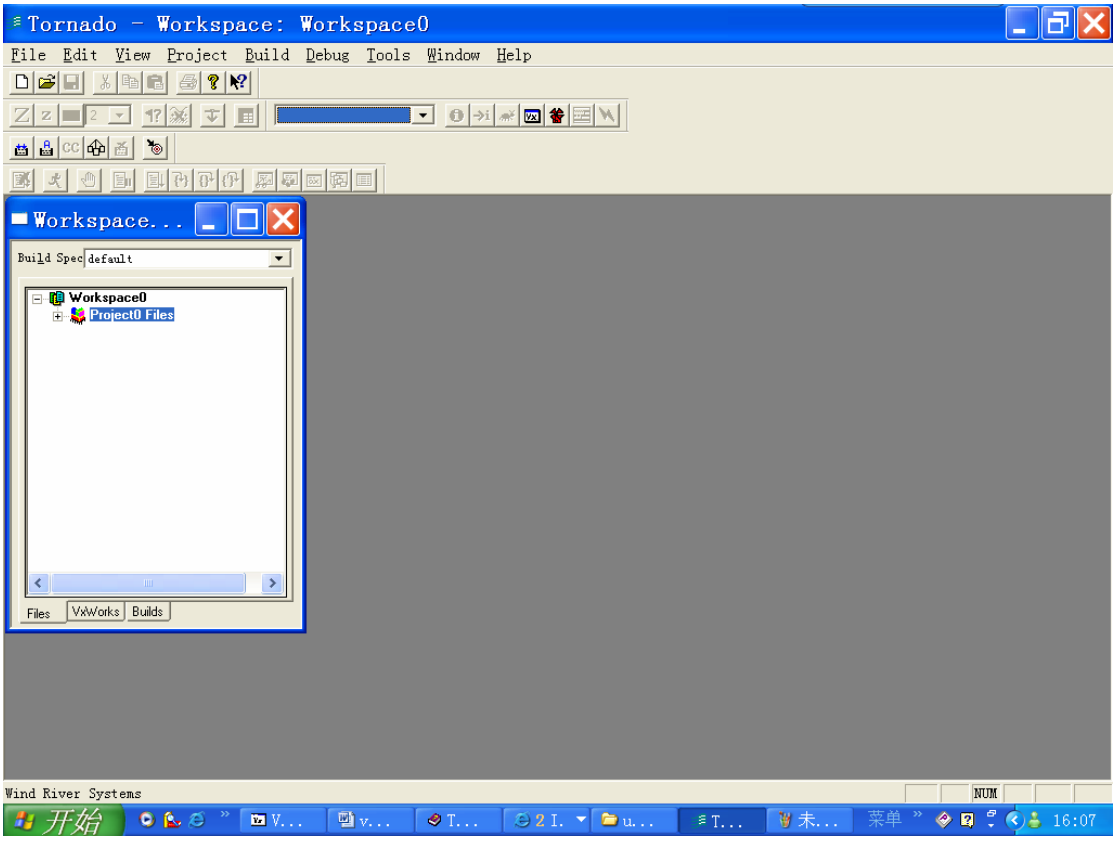


图 3.20

在工作空间的 Files 窗口中有几个创建工程时自动添加的文件：

- romInit.s：存放了 Vxworks BootRom 的入口，即是上电以后执行初始化代码，在完成处理器初始化以后转入函数 romStart。
- RomStart.c：主要功能是将 Rom 中的数据段和代码段拷贝到内存中去，然后将一部分空闲内存清零，如果需要进行解压缩。
- sysAlib.s:位于 RAM 中的 Vxworks 操作系统的启动入口。它禁止中断和 Cache,建立堆栈，然后跳入 usrConfig.c 文件中的 usrInit（）函数。
- sysLib.c：在不同板子的硬件环境中，此文件都可以向高层软件提供相同的接口。他们包含了硬件的初始化，中断产生和处理硬件时钟和定时器管理，映射本地和总线内存空间，内存保护模式等。
- linkSyms.c：工程自动生成的符号连接表。
- prjConfig.c:存放了系统启动时一些固定初始化函数调用。
- usrAppInit.c:存放了用户程序入口代码。

在工作空间的 vxwokrs 窗口是所有的组件，可以根据需要添加和裁减组件，默认模式下这里选择了 WDB network connection。如果要进行串口调试，就要选择 WDB serial connection，这里我们使用网络连接组件。其他的组件在后面的实验中遇到的会有说明。

在工作空间的 Builds 窗口单击 Project0 Builds 前的加号，然后右键选择 default_rom（此时会在所建的工程文件夹下生成一个名为 default_rom 的文件夹），在上下文菜单中先选择 Set’default_rom’as Active Build 再选择 Properties 出现图 3.21。在 Properties 窗口中选择 Rulse，从下拉列表中选择 vxWorks_rom.bin（表中其他的文件格式说明如下）然后点击 ok，在菜单栏点击编译按钮  编译或右键点击 default_rom 选择 Build’vxWorks’编译。编译结束后如图 3.22,编译生成的映象文件在所建的工程文件夹下（本工程名是 project0）的 default_rom 文件夹中。

缩写	含义	举例
hex	文件后缀，表示伟 hex 文件格式，多用于烧录 ROM; 对于 ROM 启动映象都有上两种格式	vxWroks_rom vxWorks_rom.hex
uncmp	Uncompressed,未压缩映象	bootrom_uncmp.hex
Res	Rom_resident,ROM 驻留执行映象	Bootrom_res vwWorks_romResident
nosym	映象不带 build-in 符号表	vxWorks.res_rom_nosym
rom	BootRom 和 VxWorks 合二为一的映象	vxWorks_rom
st	Standalone 映象	vxWorks.st

表 3.1

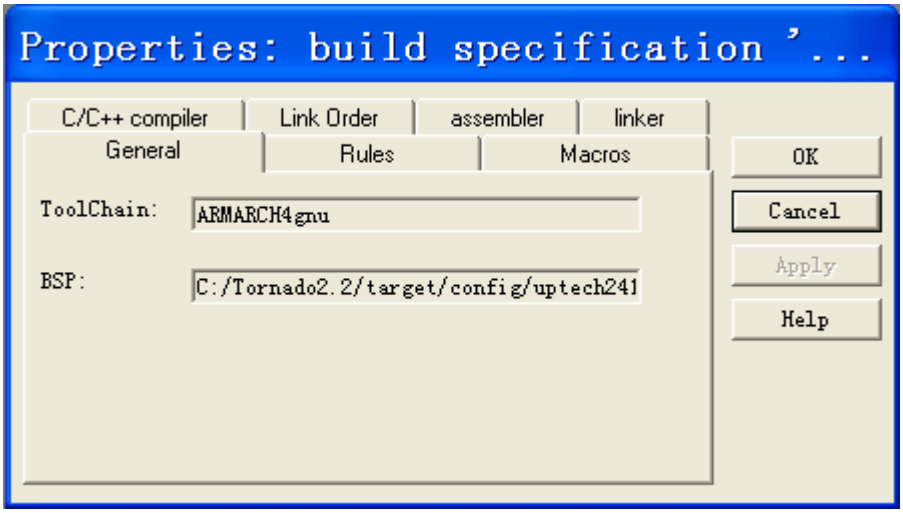


图 3.21

图 3.22

建立超级终端将编译生成的映象文件下载到目标板中，然后在 vivi 的提示符下输入 boot 启动系统，在超级终端输出如图 3.23，bootable 工程建立成功。建立超级终端的方法见附件。

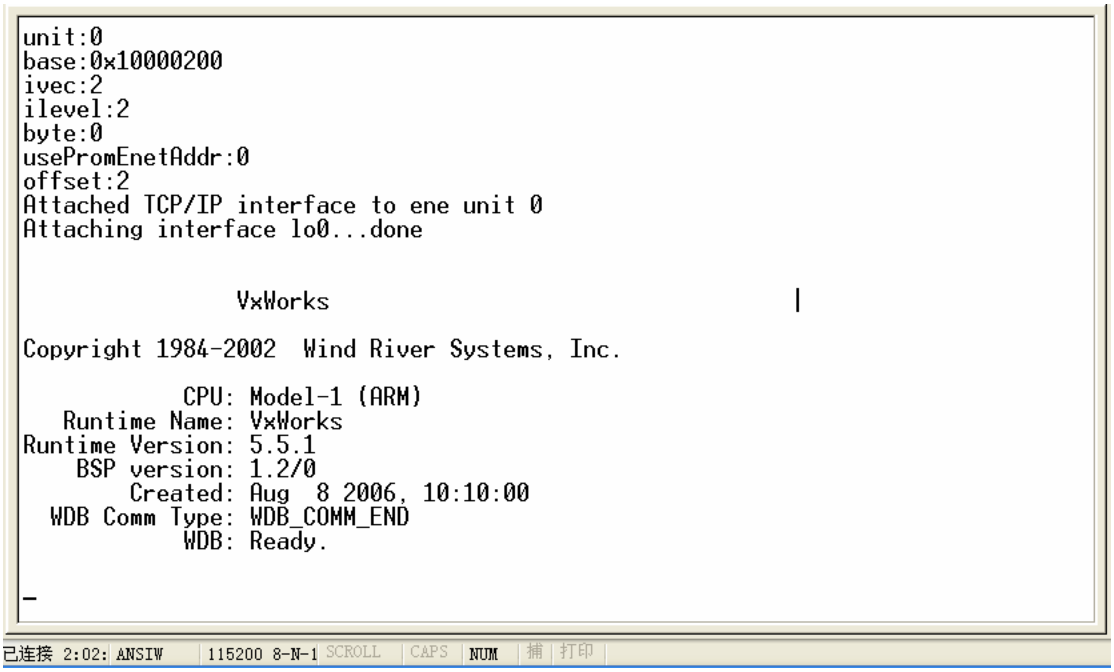


图 3.23

3.2 Downloadable 工程实践

启动 Tornado 方法同前，在创建工程窗口选择 Creat downloadable 如图 3.24，点击 ok 继续。出现图 3.25，在这里确定工程的名字、位置和工作空间。这里我们采用默认设置，用户可以自行修改。单击 next 选择工具链 ARMARCH4gnu 如图 3.26,再点击 finish 完成如图 3.27。在菜单栏中选择 file->new 出现如图 3.28,在 file name 中输入 hello.c 点击 ok 出现 hello.c 源文件的编辑窗口，输入下列代码：

```
#include <ioLib.h>
hello(){
int serialFd;
serialFd = open("/tyCo/0",O_RDWR,0);
write(serialFd,"Hello,World!\n",13);
close(serialFd);
}
```

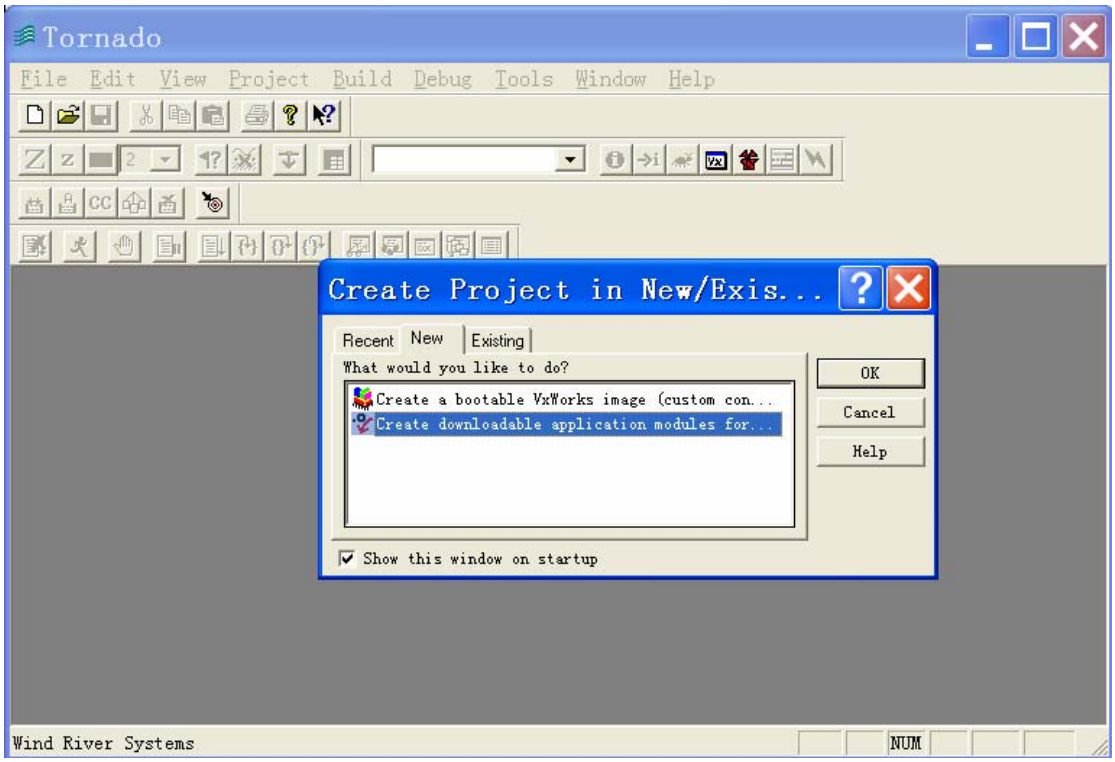

Tornado
File Edit View Project Build Debug Tools Window Help
Create Project in New/Exis...
Recent New Existing
What would you like to do?
Create a bootable VxWorks image (custom con...
Create downloadable application modules for...
OK
Cancel
Help
Show this window on startup
Wind River Systems
NUM

图 3.24

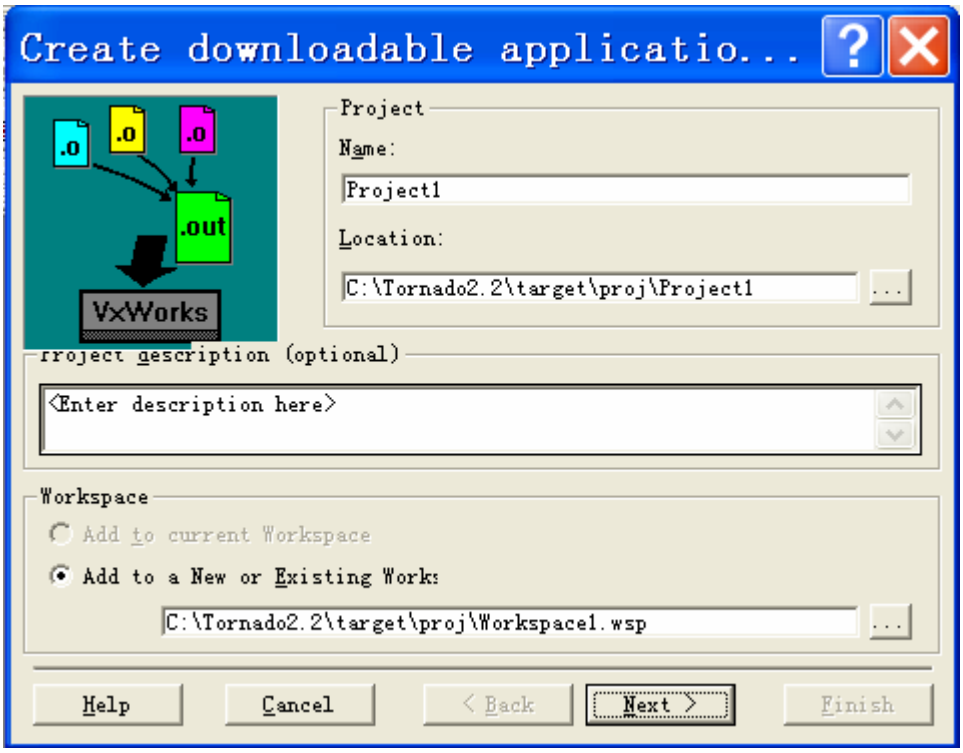

Create downloadable applicatio...
.o
.o
.o
.out
VxWorks
Project
Name:
Project1
Location:
C:\Tornado2.2\target\proj\Project1
...
Project description (optional)
<Enter description here>
Workspace
Add to current Workspace
Add to a New or Existing Works
C:\Tornado2.2\target\proj\Workspace1.wsp
...
Help
Cancel
< Back
Next >
Finish

图 3.25

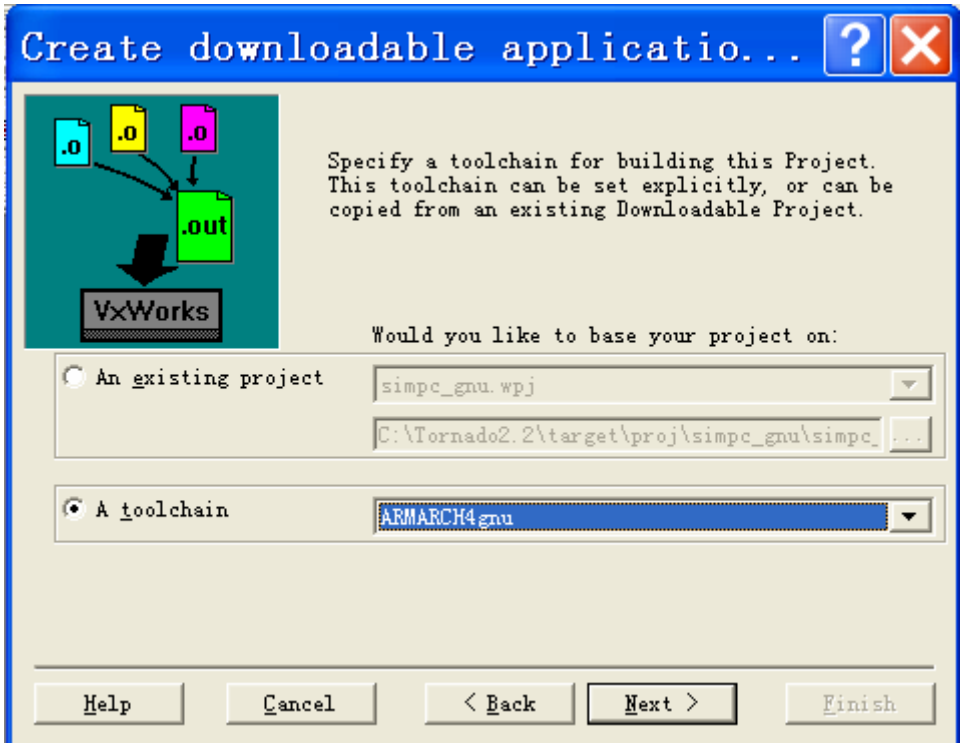

Create downloadable applicatio...
.o
.o
.o
.out
VxWorks
Specify a toolchain for building this Project.
This toolchain can be set explicitly, or can be
copied from an existing Downloadable Project.
Would you like to base your project on:
An existing project
simpc_gnu.wpj
C:\Tornado2.2\target\proj\simpc_gnu\simpc_
A toolchain
ARMARCH4gnu
Help
Cancel
< Back
Next >
Finish

图 3.26

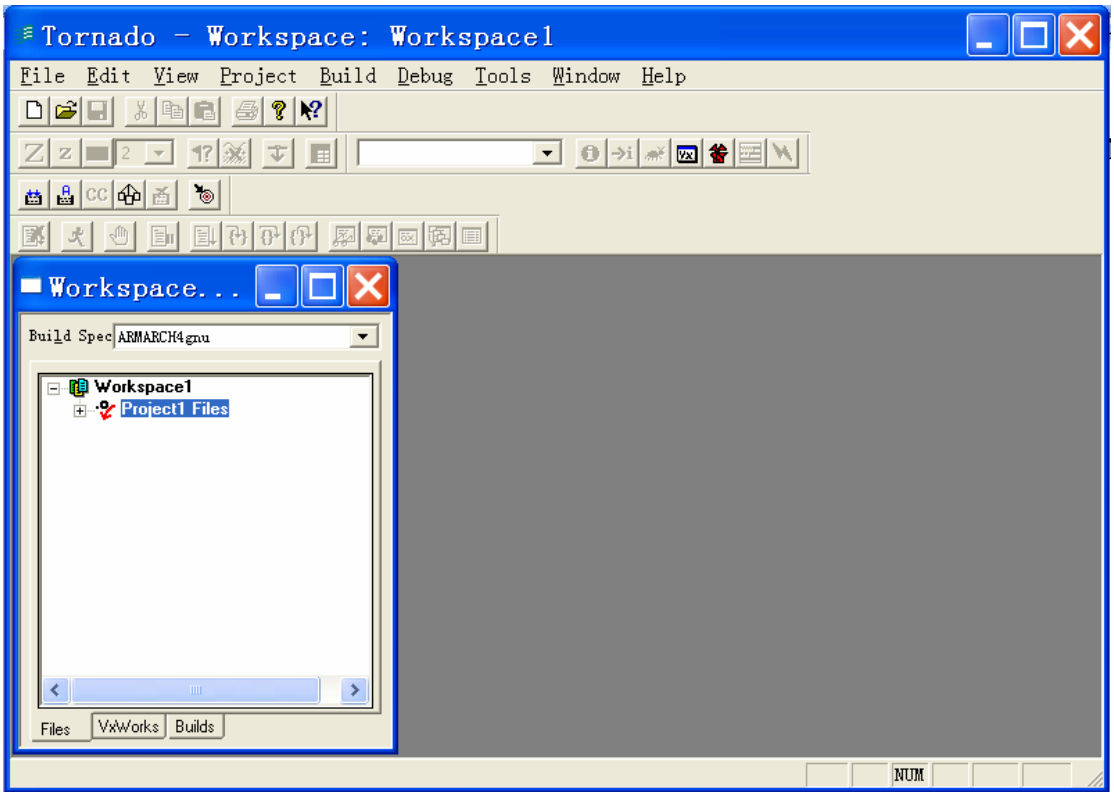

Tornado - Workspace: Workspace1
File Edit View Project Build Debug Tools Window Help
Workspace...
Build Spec ARMARCH4gnu
Workspace1
Project1 Files
Files
VxWorks
Builds
NUM

图 3.27

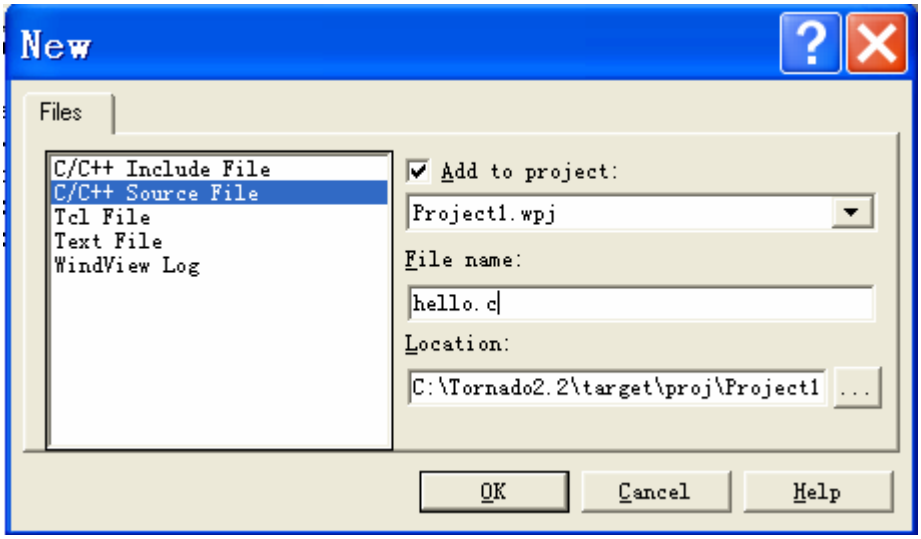


图 3.28

然后保存，在工作空间的 file 窗口右键选择刚添加的 hello.c 源文件，在上下文菜单中选择 compile’ hello.c’ 编译，如图 3.29。编译完成以后，配置 Target server（配置方法见附录），在工作空间的 files 窗口中的 Object Module 文件夹下右键选择 Project1.out，从上下文菜单中选择 Download‘Project1.out’， 下载完成后起 shell,在命令行输入 hello 运行程序，在超级终端显示 hello 如图 3.30,

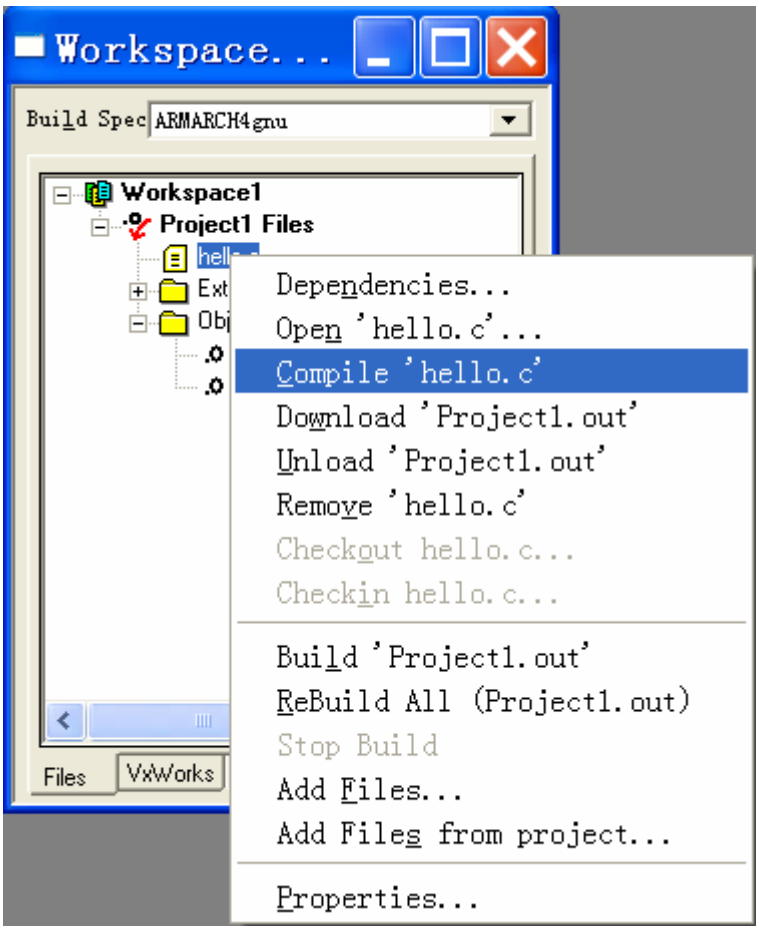


图 3.29

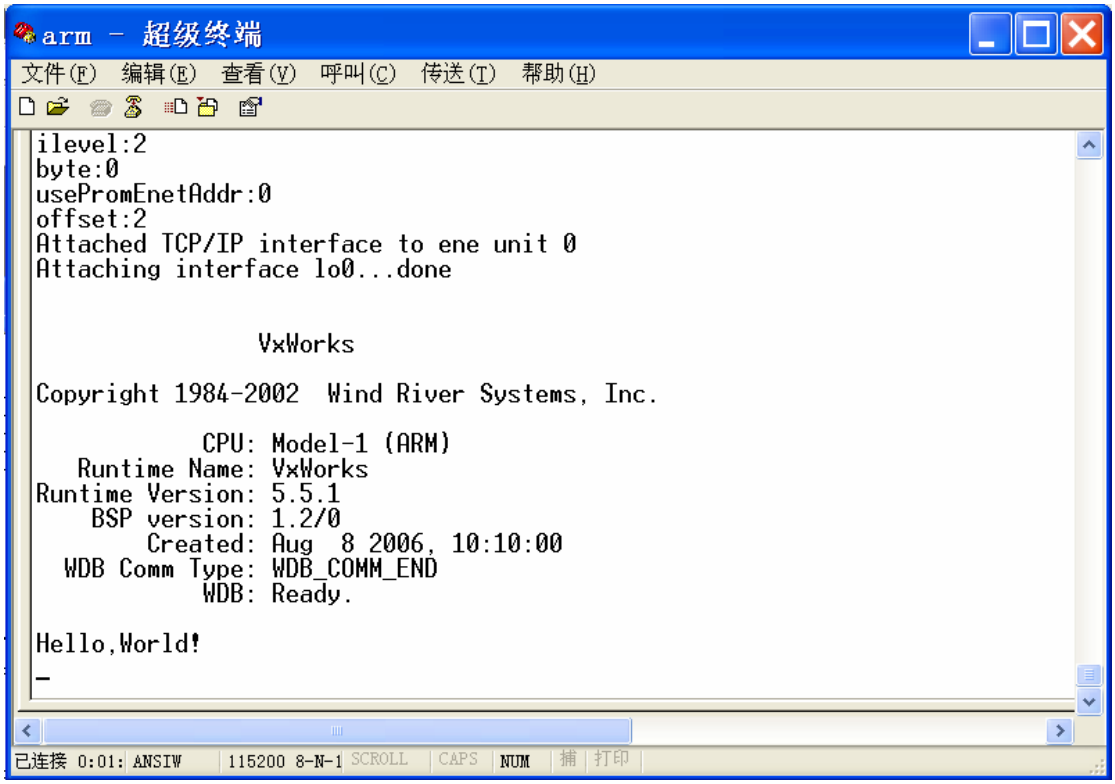

arm - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
ilevel:2
byte:0
usePromEnetAddr:0
offset:2
Attached TCP/IP interface to ene unit 0
Attaching interface lo0...done
VxWorks
Copyright 1984-2002 Wind River Systems, Inc.
CPU: Model-1 (ARM)
Runtime Name: VxWorks
Runtime Version: 5.5.1
BSP version: 1.2/0
Created: Aug 8 2006, 10:10:00
WDB Comm Type: WDB_COMM_END
WDB: Ready.
Hello,World!
已连接 0:01: ANSIW 115200 8-N-1 SCROLL CAPS NUM 捕 打印

图 3.30

第四章 驱动实验

WindML 3.0.3 开发

4.1、WindML 简介

Wind River Multimedia Library（WindML）是支持嵌入式系统的多媒体软件开发包。WindML 软件开发包支持图形、视频和音频，并且是支持用户开发各种设备标准驱动的架构。

WindML 支持的主机开发环境有 Windows 2000，NT 4.0，和 XP，还有 Solaris 2.7，2.8，和 2.9。

图形设备和驱动列表：

设备	处理器	Endian	编 译器	Tornado 版本	图形模式	可选组件
Assiliant CT6555x, CT690x0	ARMARCH4, ARMARCH4_T	BE，LE	GNU or Diab	2.1.x，2.2.x	640x480x8 640x480x16 800x600x8 1024x768x8 1024x768x16	JPEG ， Double Buffer
Asiliant CT6555x CT690x0	PPC603, PPC604	BE	GNU or Diab	2.1.x，2.2.x	640x480x8 640x480x16 800x600x8 1024x768x8 1024x768x16	JPEG ， Double Buffer
Asiliant CT6555x CT690x0	Pentium ， Pentium2 ， Pentium3 ， Pentium4	LE	GNU	2.1.x，2.2.x	640x480x8 640x480x16 800x600x8 1024x768x8 1024x768x16	JPEG ， Double Buffer
Asiliant CT6555x CT690x0	PPC405	BE	GNU or Diab	2.1.x，2.2.x	640x480x8 640x480x16 800x600x8 1024x768x8 1024x768x16	JPEG ， Double Buffer
Asiliant CT6555x CT690x0	MIPS32sf ， MIPS64	BE，LE	GNU or Diab	2.1.x，2.2.x	640x480x8 640x480x16 800x600x8 1024x768x8 1024x768x16	JPEG ， Double Buffer
CyrixMediaGX Compatible	Pentium	LE	GNU	2.0.x	640x480x8 640x480x16 800x600x8	JPEG ， Double Buffer ，

					1024x768x8 1280x1024x8	Software Cursor
Epson S1D13506 , S1D13806	MIPS32sf , MIPS64	LE	GNU or Diab	2.1.x, 2.2.x	640x480x4 640x480x8 640x480x16 800x600x4 800x600x8 800x600x16	JPEG
Epson S1D13506 , S1D13806	Pentium , Pentium2 , Pentium3 , Pentium4	LE	GNU	2.0.x, 2.2.x	640x480x4 640x480x8 640x480x16 800x600x4 800x600x8 800x600x16	JPEG
Epson S1D13506 , S1D13806	SH7750	LE	GNU or Diab	2.1.x, 2.2.x	640x480x4 640x480x8 640x480x16 800x600x4 800x600x8 800x600x16	JPEG
Hitachi MS13806	SH7750	BE, LE	GNU or Diab	2.1.x, 2.2.x	640x480x4 640x480x8 640x480x16 800x600x4 800x600x8 800x600x16	JPEG
Hitachi Q2SD	SH7750	BE, LE	GNU or Diab	2.1.x, 2.2.x	640x480x8	JPEG
HitachiSH7727 on-chip LCD with Hitachi STN LCD parnel	SH7750	BE, LE	GNU or Diab	2.1.x, 2.2.x	250x320x8 256x320x16	JPEG
Intel 81x	Pentium	LE	GNU	2.0.x, 2.2.x	320x400x8 320x400x16 352x480x16 640x480x16 800x600x8 800x600x16	JPEG
Intel PXA250 on-chip LCD with Wind River LCD panel	XSCALE	BE, LE	GNU or Diab	2.2.x	640x480x8	JPEG

Intel SA1100 on-chip LCD with Assabet (LQ03Q2DS01 LCD panel)	STRONGARM	BE, LE	GNU or Diab	2.1.x, 2.2.x	320x240x8 320x240x16	JPEG
Motorola MPC82x on-chip LCD with Sharp LQ10D367	PPC860	BE	GNU or Diab	2.0.x, 2.2.x	640x480x8	JPEG
Tvia CyberPro 5050	ARMARCH4 , ARMARCH4_T	BE, LE	GNU or Diab	2.1.x, 2.2.x	640x480x8 640x480x16 800x600x8 800x600x16 1024x768x8 1024x768x16	Video , JPEG , Overlays, Software Cursor
Tvia CyberPro 5050	MIPS32sf , MIPS64	BE, LE	GNU or Diab	2.1.x, 2.2.x	640x480x8 640x480x16 800x600x8 800x600x16 1024x768x8 1024x768x16	Video , JPEG , Overlays, Software Cursor
Tvia CyberPro 5050	Pentium , Pentium2 , Pentium3 , Pentium4	LE	GNU	2.0.x, 2.2.x	640x480x8 640x480x16 800x600x8 800x600x16 1024x768x8 1024x768x16	Video , JPEG , Overlays, Software Cursor
Tvia CyberPro 5050	PPC603, PPC604	BE	GNU or Diab	2.0.x, 2.2.x	640x480x8 640x480x16 800x600x8 800x600x16 1024x768x8 1024x768x16	Video , JPEG , Overlays, Software Cursor
Vesa BIOS 2.1 compatible	Pentium , Pentium2 , Pentium3 , Pentium4	LE	GNU	2.0.x, 2.2.x	640x480x8 640x480x16 800x600x8 800x600x16 1024x768x8 1024x768x16	JPEG Double Buffer
VGA Compatible	I80486	LE	GNU	2.0.x	640x480x8 640x480x16 800x600x8 800x600x16	JPEG

					1024x768x8 1024x768x16	
VGA Compatible	Pentium , Pentium2 , Pentium3 , Pentium4	LE	GNU	2.0.x, 2.2.x	320x200x8 640x480x4	JPEG
VxSim Windows host	SIMNT	LE	GNU	2.0.x, 2.2.x	640x480x8 640x480x16 800x600x8 800x600x16 1024x768x8 1024x768x16	JPEG Double Buffer
VxSim Solaris host	SIMPARCSOLARIS	BE	GNU	2.0.x, 2.1.x, 2.2.x	320x200 640x480 800x600 1024x768 （Depth as per host）	JPEG Double Buffer

输入设备列表：

设备	处理器	Endian	编译器	Tornado 版本
Assabet Touchscreen	STONGARM	BE, LE	GNU	2.1.x, 2.2.x
PS2 Keyboard	MIPS32sf , MIPS64, PPC603, PPC604, PPC860, SH7750, SH7700	BE, LE	GNU or Diab	2.0.x, 2.2.x
PS2 Keyboard	Pentium , Pentium2 , Pentium3 , Pentium4,	BE, LE	GNU	2.0.x, 2.2.x
PS2 Mouse	Pentium , Pentium2 , Pentium3 , Pentium4,	BE, LE	GNU	2.0.x, 2.2.x
Serial Mouse （Microsoft）	ARMARCH4 , ARMARCH4_T , MIPS32sf , MIPS64, PPC603, PPC604, PPC860, SH7750, SH7700	BE, LE	GNU or Diab	2.0.x, 2.1.x, 2.2.x
Serial Mouse （Microsoft）	Pentium , Pentium2 , Pentium3 ,	BE, LE	GNU	2.0.x, 2.2.x

	Pentium4,			
USB Keyboard	Pentium , Pentium2 , Pentium3 , Pentium4,	BE, LE	GNU	2.2.x
USB Mouse	ARMARCH4 , ARMARCH4_T , MIPS32sf , MIPS64, PPC603, PPC604, PPC860, SH7750, SH7700	BE, LE	GNU , LE	2.2.x
USB Mouse	Pentium , Pentium2 , Pentium3 , Pentium4,	BE, LE	GNU	2.2.x

音频设备列表；

设备	处理器	编译器	Tornado 版本
Tvia CyberPro 5060	MIPS32sf, MIPS64	GNU or Diab	2.0.x, 2.1.x, 2.2.x
Tvia CyberPro 5060	Pentium, Pentium2, Pentium3,Pentium4,	GNU	2.0.x, 2.2.x

4.2、安装和配置

WindML 是以源代码形式的提供给客户的。当您安装完 WindML 以后，必须使用 Tornado 的工具对这些源代码进行编译。

安装过程和安装 Tornado 一样，这里不做赘述。需要注意的是要先安装 WindML3.0.2，然后安装 WindML3.0.3 的补丁。

4.3、WindML 体系

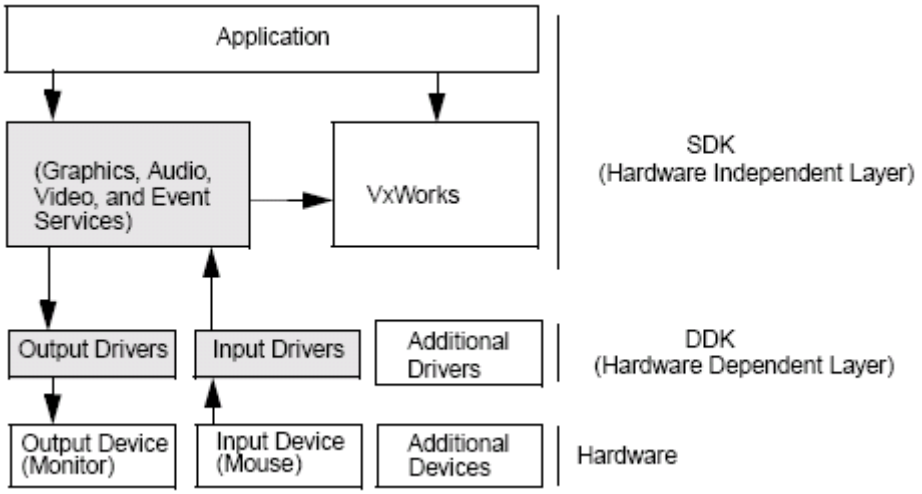
WindML 包括两个部分

（1） Software Development Kit（SDK）

SDK 是用来开发应用程序。SDK 提供图形、输入、多媒体、字体和存储器管理的 API，可以用来开发硬件平台无关的应用程序。

（2） Driver Development Kit（DDK）

DDK 是用来开发硬件驱动的。DDK 提供参考驱动，可以让用户方便快捷的开发图形、输入设备等的驱动程序。



SDK 定义了应用程序和驱动程序之间的接口。SDK 提供了下面的 API 组：

- 图形设备的资源初始化和清除；
- 多媒体 API，包括 2D 图形，区域管理，窗口、颜色管理，图像，JPEG 和音频；
- 事件服务；
- 内存管理；
- 扩展 API；
- 设备管理；

DDK 提供多媒体层和硬件直接的接口。包括显示器、视频、音频、键盘和鼠标。

- 图形驱动；
- 视频驱动；
- 字体驱动；
- 音频驱动；
- 窗口管理；

4.4、WindML 开发流程简介

本章节简单的介绍 WindML 的开发流程。

VxWorks 的 BSP，通常会提供 WindML 的支持。在 WindRiver 的 Tornado2.2 以上的版本提供的 BSP 中都会 sysWindML.c，这个文件是提供 WindML 所需的功能。

在安装完成的目录/target/src/ugl/bsp/config/bspName 中已经提供了一些 BSP 的支持。但是并不是所有的 BSP 都支持。所以 **WindML 的第一步是要在 BSP 添加 sysWindML.c**。sysWindML.c 的内容可以参考已有 sysWindML.c。

并且在 BSP 中的 config.h 里面定义 #define INCLUDE_WINDML。如果需要，在 sysLib.c 里面定义 WindML 所需要的内存区域。

第二步，选择驱动程序，驱动程序在/target/src/ugl/driver/里面，包括 keyboard、pointer、graphics、font。确定驱动程序存在并且正确。如果 WindML 没有合适的驱动程序，那么需要我们在已有的驱动程序的基础上开发新的。

第三步，配置驱动程序并且编译。

在 Tornado 菜单中启动 Tools>WindML， 会出现 WindML 的配置选择画面。然后选择合适配置文件，如果没有，则需要重新建立一个。

然后会出现配置画面，选择 Build 项，选择处理器器类型，编译器类型，以及编译 debug 或者 example。

选择 Devices 项，选择图形设备，选择输出、输入设备，比如 CRT 或者平面输出 LCD 等。选择图形设备的属性，比如分辨率、刷新频率和颜色模式。选择指针设备，比如鼠标等。选择 keyboard、audio 等设备。

选择 Advanced 选项，比如存储池类型以及窗口管理。

选择 Bitmap Font 选项，可以选择字体和 Bitmap、JPEG 等选项。

然后保存配置，就可以进行编译。编译完成，WindML 的库就可以使用了，同时也编译了很多例子程序。就可以下载例子程序进行测试。

4.5、UPTECH2410 的 LCD 开发流程详解

4.5.1 WindML 的 BSP 修改

第一步：在 BSP 里面的 config.h 里面添加 WindML 的支持。

在 config.h 里面，添加 `#define INCLUDE_WINDML`

第二步：建立 sysWindML.c

在 sysWindML.c 里面需要添加的函数包括

- sysWindMLDevGet
- sysWindMLDevCtrl
- sysWindMLDevRelease
- sysWindMLIntConnect
- sysWindMLIntEnable
- sysWindMLIntDisable
- sysWindMLHwInit

下面我们一一介绍这些函数的作用和写法。

SysWindMLDevGet 函数是最重要的一个函数，他会被 WindML 的初始化函数调用，来获取硬件设备的信息。主要包括 vendorID（提供商 ID），instance（实例），devType（设备类型），deviceID（设备 ID），并且返回这个设备结构的指针，这个指针中包含了设备的各种信息。对于图形设备来说最重要的就是物理地址。由于 2410 里 LCD Controller，所以我们的信息要和实际的硬件相匹配。对于 2410 的 LCD Controller 最重要的一部就是建立 LCD 的缓冲区，并且把缓冲区的地址传入到设备结构指针中去。

下面逐句分析 sysWindMLDevGet 函数。

```
WINDML_DEVICE * sysWindMLDevGet
(
    UINT32 devType,      /* Device Type */
    UINT32 instance,     /* The instance of the device */
    UINT32 vendorID,     /* The identifier for the device manufacturer */
    UINT32 deviceID      /* The identifier for the device */
)
```

```
{
    WINDML_DEVICE * pDev = NULL;
    int busno, slotno, funcno;
    UINT16 data;
    switch (devType) {
    case WINDML_GRAPHICS_DEVICE : /* 表示是图形设备，也是我们要开发的设备 */
    {
        if (vendorID == 0 || deviceID == 0)
        {
            vendorID = data;
            deviceID = data;
        }
        else
        {
            printf("verdorID and deviceID are not 0.\n");
        }

        pDev = (WINDML_DEVICE *)calloc(1, sizeof(WINDML_DEVICE)); /* 分配内存 */

        if (pDev == NULL)
        {
            return (pDev);
        }

        /* memRgn = */
        /* 添加相关信息 */
        pDev->vendorID = vendorID;
        pDev->deviceID = deviceID;
        pDev->instance = instance;
        pDev->devType = devType;

        /* LCD buffer address */
        pDev->pPhysBaseAdrs0 = (void *)cacheDmaMalloc(640 * 480 * 16 / 8); /* 分
配 LCD 缓冲区，缓冲区大小 = 宽 x 高 x 颜色深度 / 8 */

        pDev->pPhysBaseAdrs1 = 0;
        pDev->pPhysBaseAdrs2 = 0;
        pDev->pPhysBaseAdrs3 = 0;
        pDev->regDelta = 0;
        pDev->intLevel = 0;
        pDev->intVector = NULL;
        pDev->pRegBase = 0;

        break;
```

```
    }

    case WINDML_KEYBOARD_DEVICE : /* 目前并没有实现 */
    {
        printf("Keyboard device do not support right now.\n");
        break;
    }

    case WINDML_POINTER_DEVICE : /* 目前并没有实现 */
    {
        printf("Pointer device do not support right now.\n");
        break;
    }

    case WINDML_AUDIO_DEVICE : /* 目前并没有实现 */
    {
        /* not implemented yet */
        printf("Audio device do not support right now.\n");
        break;
    }

    default:
    {
        printf("Error switch item.\n");
        break;
    }
    }
    return (pDev);
}
```

与对应

```
STATUS sysWindMLDevRelease
(
    WINDML_DEVICE *pDev /* Device to release */
)
{
    if(pDev != NULL)
    free(pDev);

    return(OK);
}
```

与 sysWindMLDevGet 对应， sysWindMLDevRelease 要释放结构指针的内存。

```
STATUS sysWindMLDevRelease
(
```

```
    WINDML_DEVICE *pDev /* Device to release */
)
{
    if(pDev != NULL)
    free(pDev);

    return(OK);
}
```

其他的函数在 2410 的 LCD 开发中并没有使用，所以可以不用完成，但是在不同的硬件或者 LCD 的情况下，有可能需要完成。

另外，需要注意的一点就是大多数 WindML 支持的 LCD 或者 CRT 都是基于 PCI 总线的，所以在 WindML 中需要 PCI 相关的函数，虽然这些函数并没有实际作用，仍然需要。所以我们 sysWindML.c 的最后添加了 pci 的读写函数的模型，但并没有实现。

比如下面的例子：

```
STATUS pciConfigInByte
(
    int bus,         /* bus number */
    int slot,        /* slot number */
    int function,    /* function number */
    int offset,      /* offset into the configuration space */
    char *data       /* data read from the offset */
)
{
    printf("pciConfigInByte.\n");
}
```

然后新建 VxWorks Image 的时候要把 sysWindML.c 加入，加入方法有两种，一种是在图形开发环境中建立 Bootable Project 以后，通过 Add file 的方法加入。另外一种是修改 BSP 的 Makefile，把 sysWindML.c 加入。

4.5.2 LCD 配置文件的建立

在\host\resource\windML\config\database 目录下面是 WindML 的配置文件。我们需要修改或详细阅读的是 windML_OUTPUT_DB.cfg 文件。这个文件中详细的列出了 WindML 所支持的各种设备名称，以及相应的代码位置。我们的 LCD 类型是 LQ080V3DG01，这个类型在 WindML 软件开发包中并没有支持，所以我们要在\host\resource\windML\config\database\windML_OUTPUT_DB.cfg 添加。

```
OUTPUTTYPE=genericoutputtype \
sharplq039q2sd01 \
acttechnoemvue \
hitachistn \
necnl6448bc20 \
LQ080V3DG01 \    /* 这个是我们新添加的 */
nooutput
```

并且要把 LQ080V3DG01 的信息详细列出，填入以下内容：

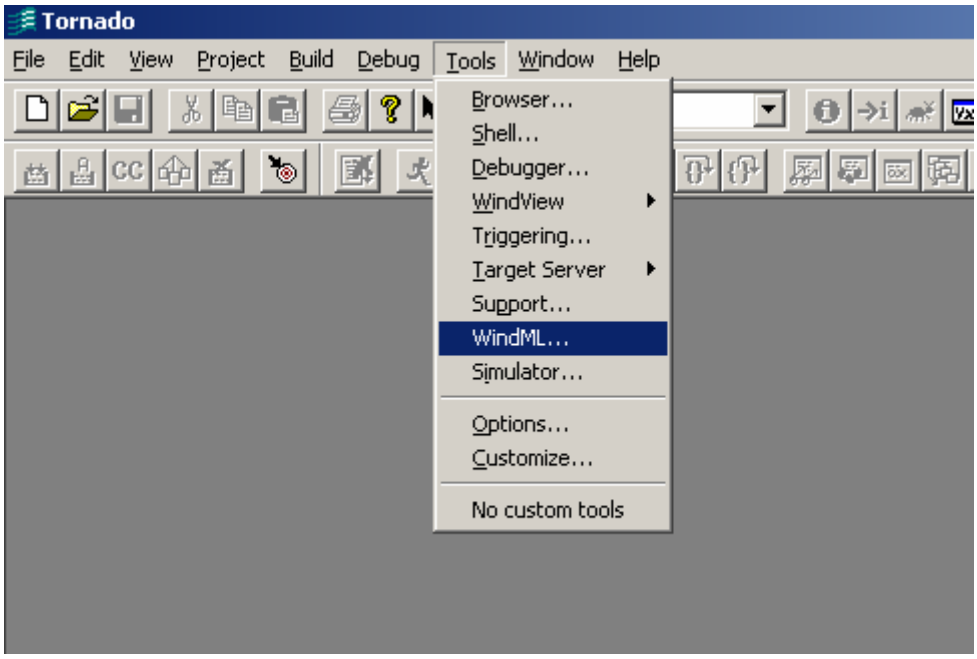
```
LQ080V3DG01.NAME=Sharp LQ080V3DG01 Flat Panel
LQ080V3DG01.SELECT=INCLUDE_UGL_SHARP_LQ080V3DG01
LQ080V3DG01.TYPE=UGL_MODE_FLAT_PANEL
```

这样，WindML 的配置工具才能识别到 LQ080V3DG01 这个设备。

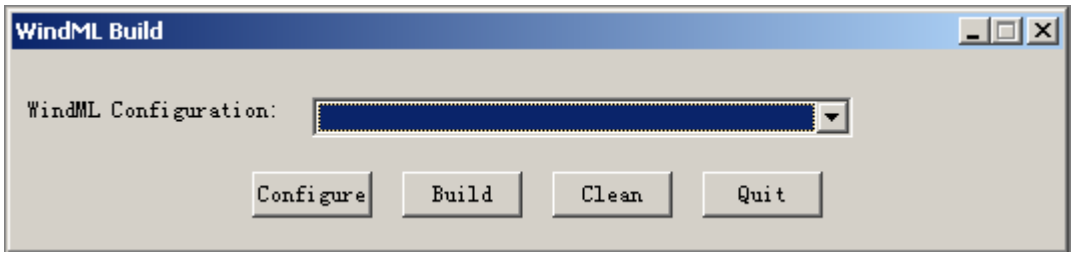
但是，只有这样还是不行的，我们只定义了输出设备的类型，但是 WindML 并不知道这个设备的 CPU、工作模式等属性。所以需要重新建立一个配置文件来完成这些属性的配置。这样我们建立\host\resource\windML\config\database\windML_UPTECH2410_DB.cfg，在这个文件中我们详细的定义 UP2410 种的 LCD 的配置。

```
GRAPHICSDEVICE=Uptech2410

Uptech2410.NAME=Uptech2410
Uptech2410.ARCH=armarch4le
Uptech2410.SELECT=INCLUDE_UPTECH2410_GRAPHICS
Uptech2410.OUTPUTTYPE=LQ080V3DG01 /* LCD 名称，必须和我们上面定义的一致 */
Uptech2410.DIR=uptech2410 /* 驱动程序的位置 */
Uptech2410.HEADER=ugl/driver/graphics/uptech2410/LQ080V3DG01.h /* 驱动程序头文件 */
Uptech2410.MODE=indexed8,640x480,60 rgb565,640x480,60 /* LCD 的工作模式 */
Uptech2410.OPTION=JPEG
Uptech2410.GRAPHICSDEV_PARAM1=0
Uptech2410.GRAPHICSDEV_PARAM2=0
```

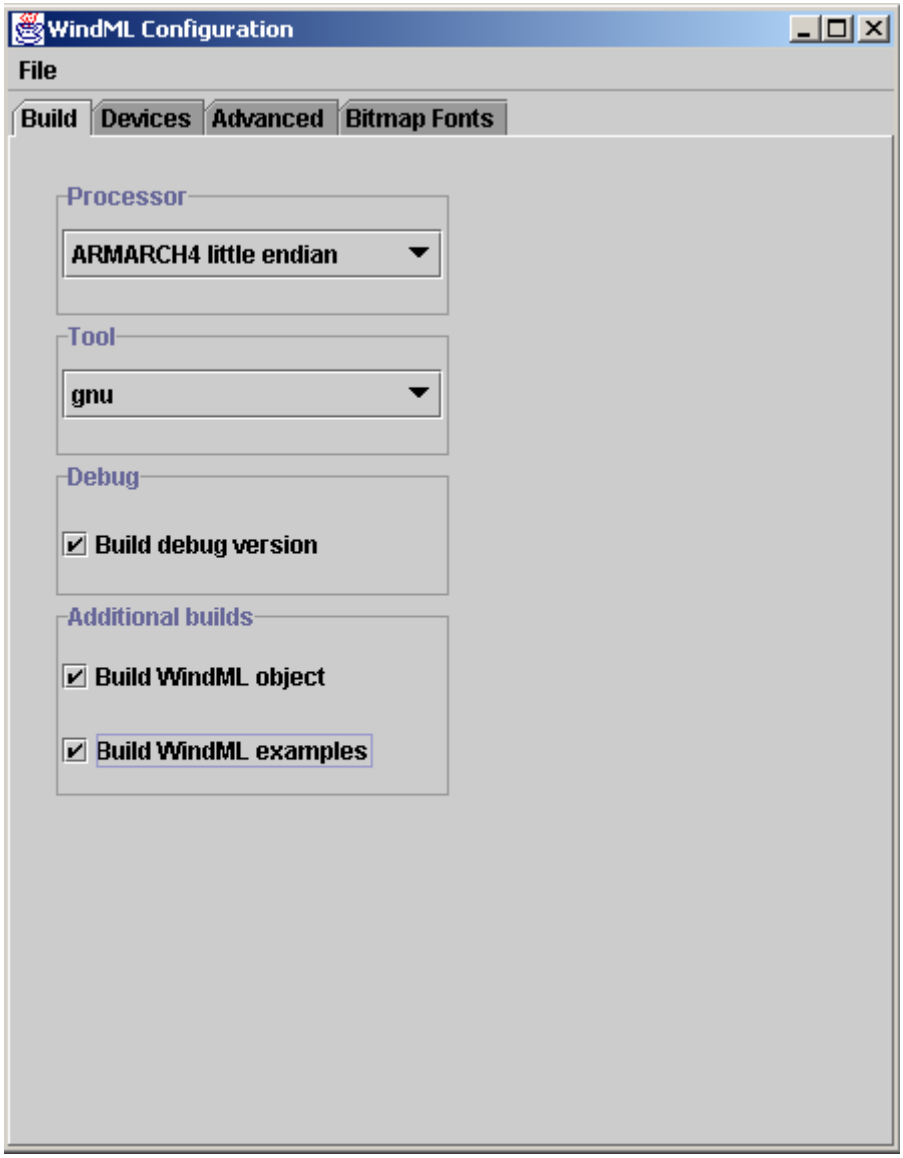
这样，我们对 WindML 的配置工作就基本完成了。然后启动 WindML 的配置工具。



点击 Tools>WindML，会出现 WindML 配置工具的画面，如下图：

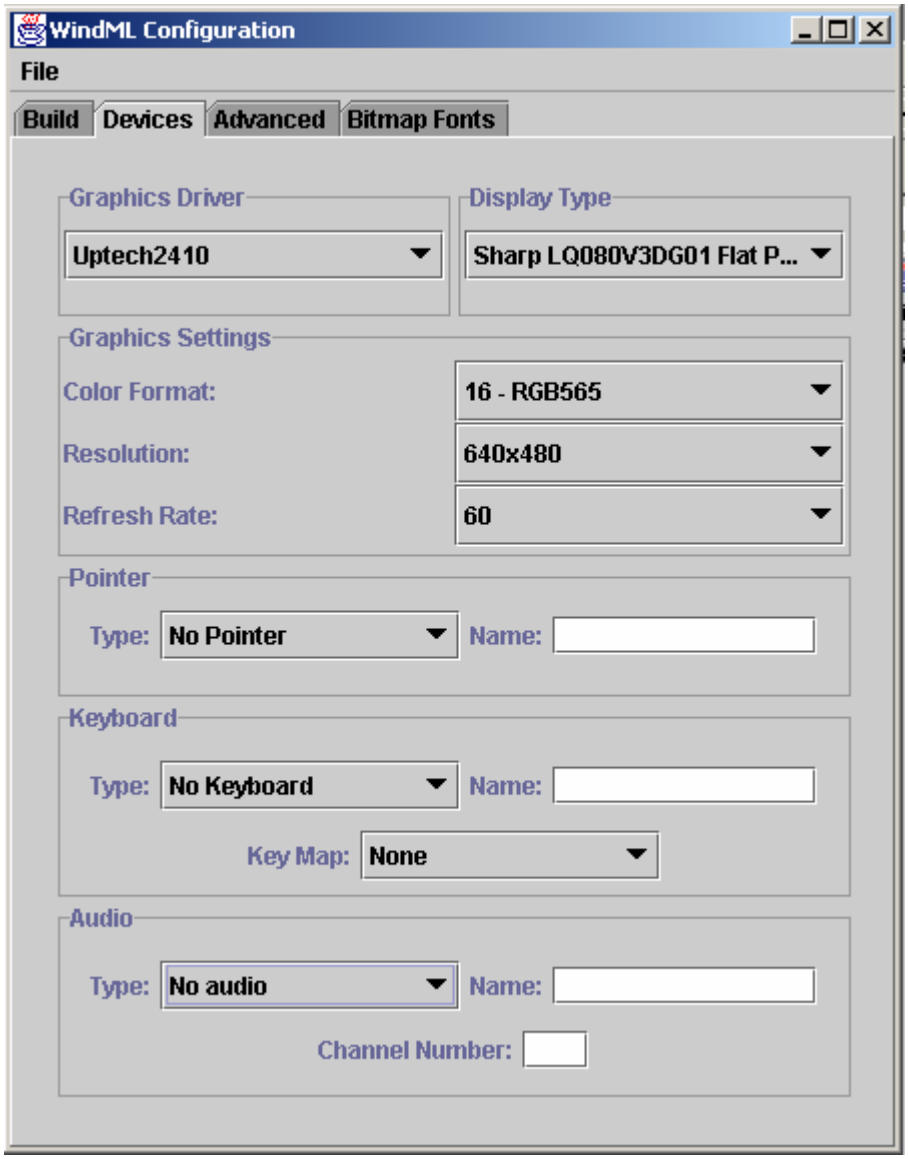


然后点其 Configure，这会出现配置的详细流程。



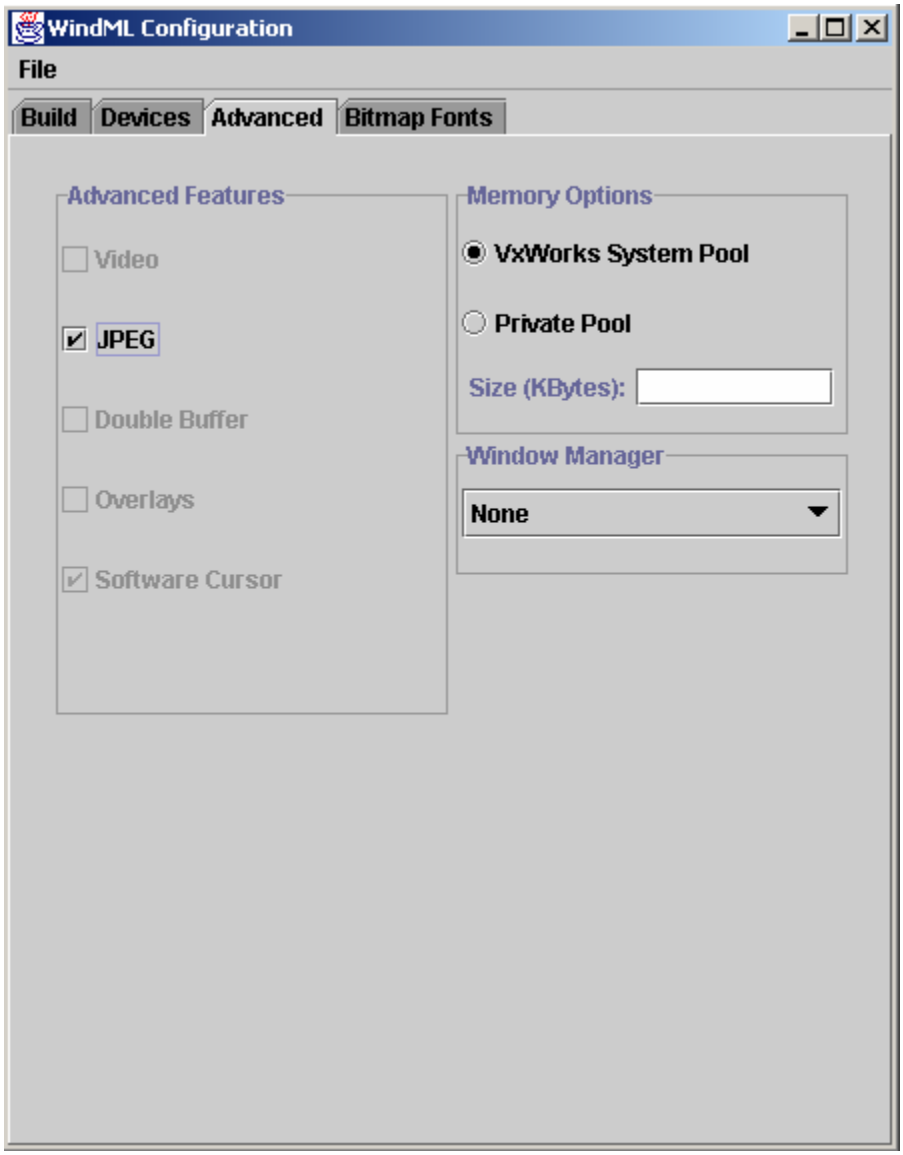
根据我们的实际情况，选择 Processor、Tool、Debug 和 Additional builds 选项，在开发的阶段，建议大家把 Debug 和 example 选择上，并且在实际产品中去掉。

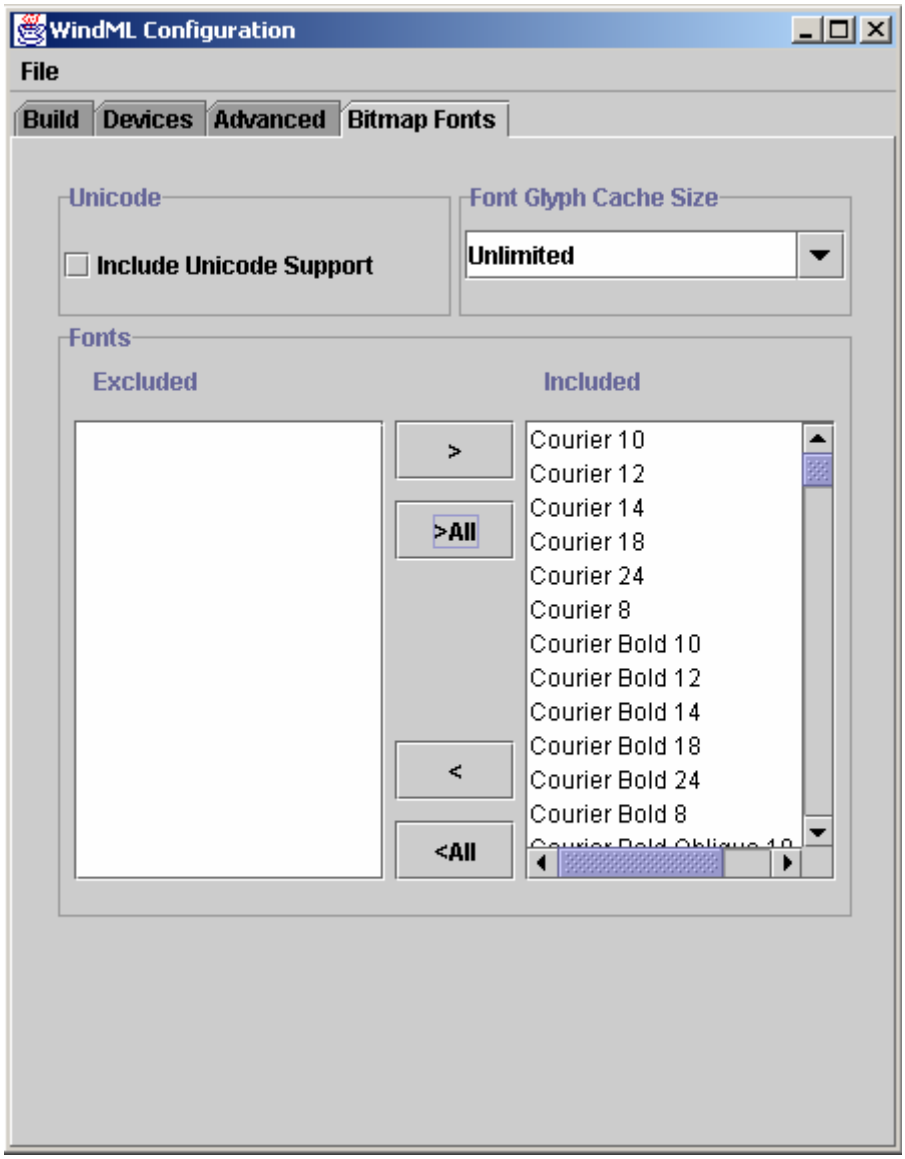
下面选择 Devices 的选项。



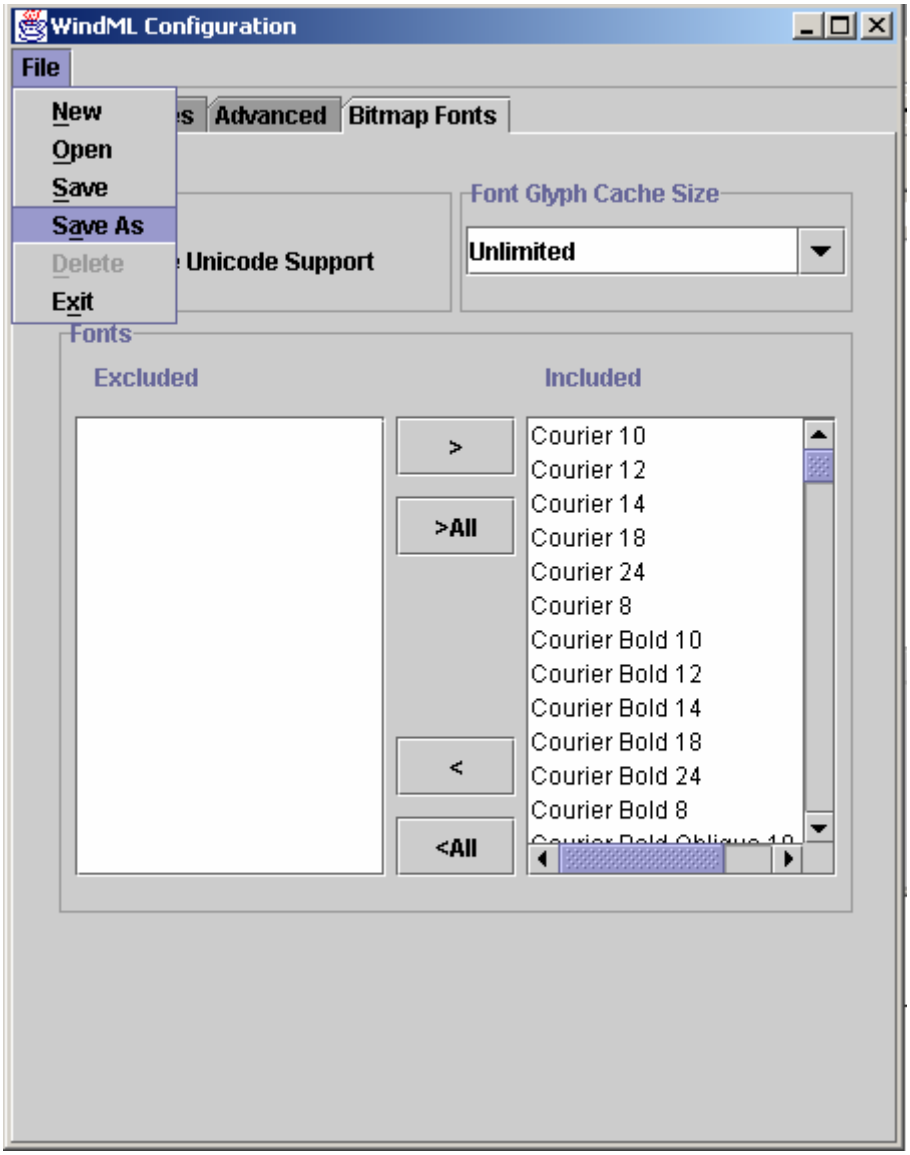
由于我们已经添加配置文件，所以在 Graphics Driver 中我们将可以看到 Uptech2410 这个选择项，选择以后，在 Display Type 中会显示我们定义的 LCD 类型的名称。在 Color Format 中我们选择 RGB565，这个是和硬件的工作模式相匹配的。其他的地方由于目前不支持鼠标、键盘和音频，所以都选择没有就可以。

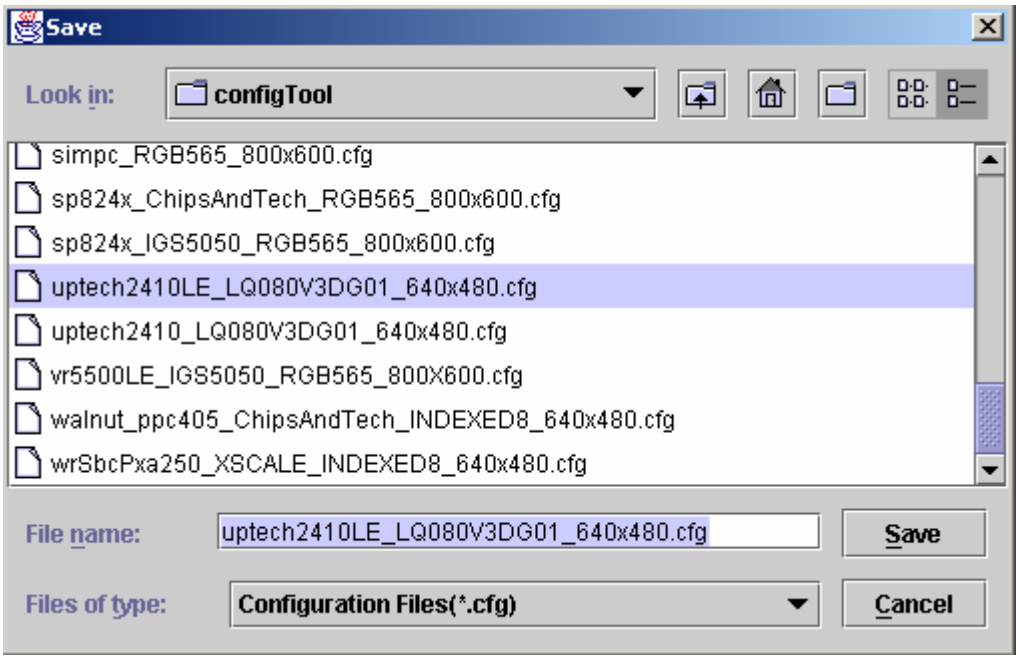
然后配置 Advanced 和 Bitmap Font，这个步骤相对简单，按照下面的画面进行选择就可以了。


WindML Configuration
File
Build
Devices
Advanced
Bitmap Fonts
Advanced Features
Video
JPEG
Double Buffer
Overlays
Software Cursor
Memory Options
VxWorks System Pool
Private Pool
Size (KBytes):
Window Manager
None

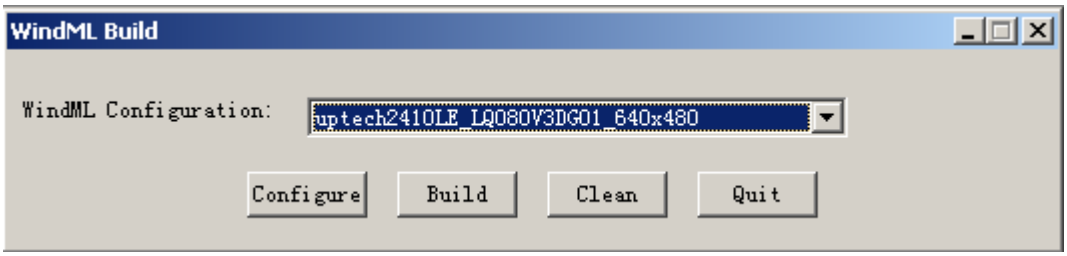


至此，我们就完成对 WindML 的配置，然后保存我们的选择。


WindML Configuration
File
New
Open
Save
Save As
Delete
Exit
Advanced
Bitmap Fonts
Unicode Support
Font Glyph Cache Size
Unlimited
Fonts
Excluded
Included
>
>All
<
<All
Courier 10
Courier 12
Courier 14
Courier 18
Courier 24
Courier 8
Courier Bold 10
Courier Bold 12
Courier Bold 14
Courier Bold 18
Courier Bold 24
Courier Bold 8



这样我们就保存了一个名称为 uptech2410LE_LQ080V3DG01_640x480.cfg 的配置文件，在下一次调用 Tools>WindML 的时候不需要在进行 Configure，直接可以进行编译了。



目前为止，我们已经对 WindML 成功地进行了配置。

4.5.3 LCD 驱动程序开发

上面的过程成功通过以后，我们会发现，其实我们在 windML_UPTECH2410_DB.cfg 文件中填写的驱动程序的位置下面并没有我们的驱动程序。下面我们就要开发驱动程序。

第一步：在 target\src\ugl\driver\graphics\建立 uptech2410 目录；

第二步：在 target\h\ugl\driver\graphics\建立 uptech2410 目录；

第三步：在 target\h\ugl\driver\graphics\uptech2410 建立 LQ080V3DG01.h 文件；

```
#define WINDML_GRAPHICS_CPU_BUS  /* 由于我们没有使用 PCI 总线，必须定义为 CPU_BUS */

#define INCLUDE_UGL_RGB565 /* 颜色模式 */

#ifndef INCLUDE_UPTECH2410_GRAPHICS
#define INCLUDE_UPTECH2410_GRAPHICS  /*  选择 UPTECH2410 的设备 */
```

```
#define BUILD_DRIVER
#include <uglInit.h>
#endif /* INCLUDE_CHIPS_GRAPHICS */

#ifdef INCLUDE_UGL_RGB565
/* 定义设备名称，设备建立函数名称，颜色深度，颜色模式 */
#define UGL_GRAPHICS_NAME       "LQ080V3DG01 - 16 Bit"
#define UGL_GRAPHICS_CREATE     uglLQ16BitDevCreate
#define UGL_COLOR_DEPTH         16
#define UGL_MODE_FLAGS          UGL_MODE_DIRECT_COLOR
#endif /* INCLUDE_UGL_RGB565 */

/* LCD 的驱动数据结构 */
typedef struct ugl_lq_driver
{
    UGL_GENERIC_DRIVER pGenDriver;
    UGL_UINT32 busType;
    UGL_UINT32 chipType;
} UGL_LQ_DRIVER;


/*  通用的图片结构头  */
/* General bitmap header type */
typedef struct ugl_lq_ddb
{
    UGL_GEN_DDB genDdb;
    UGL_BOOL vram;
}UGL_LQ_DDB;


/* 下面是函数的申明 */
/* 16 Bit driver */
    UGL_UGI_DRIVER * uglLQ16BitDevCreate(UGL_UINT32 instance, UGL_UINT32 notUsed0,
UGL_UINT32 notUsed1); /* 建立 */
    UGL_STATUS uglLQ16BitDevDestroy (UGL_DEVICE_ID devId); /* 释放 */
    UGL_STATUS uglLQ16BitInfo (UGL_DEVICE_ID devId, UGL_INFO_REQ infoRequest,
                void *info); /* 获取信息 */
    UGL_STATUS uglLQ16BitModeAvailGet (UGL_UGI_DRIVER * pDriver,
                    UGL_UINT32  * pNumModes,
                    const UGL_MODE ** pModeArray ); /* 获取模式 */
    UGL_STATUS uglLQ16BitModeSet (UGL_UGI_DRIVER * pDriver, UGL_MODE * pMode); /*
模式设置 */

/* 下面的函数并没有用到，可以忽略 */
```

```
/* 16 Bit Bitmap Support */
UGL_DDB_ID uglLQ16BitBitmapCreate(UGL_DEVICE_ID devId, UGL_DIB  * pDib,
                                  UGL_DIB_CREATE_MODE createMode,
                                  UGL_UINT32    initValue,    UGL_MEM_POOL_ID
poolId);
UGL_STATUS uglLQ16BitBitmapBlt (UGL_DEVICE_ID devId, UGL_DDB_ID srcDdbId,
                                UGL_RECT * pSourceRect, UGL_DDB_ID destDdbId,
                                UGL_POINT * pDestPoint);

/* Page Managment */
UGL_STATUS uglCTPageDrawSet (UGL_UGI_DRIVER * pDriver, UGL_PAGE * pPage);
UGL_STATUS uglCTPageVisibleSet (UGL_UGI_DRIVER * pDriver, UGL_PAGE * pPage);

/* Graphics Processor Wait */
UGL_STATUS uglLQgpWait(UGL_GENERIC_DRIVER * pDriver);
```

在我们的这个目录下面还有其它的文件，他们是和 2410 的硬件相关的，这个设计的硬件的一些东西，需要的话可以参考，但是这里并不作详细的说明。

第四步：在\target\src\ugl\driver\graphics\uptech2410 建立 LQ080V3DG01.c 文件；LQ080V3DG01.c 文件是 LCD 驱动程序。

主要包括

- uglLQ16BitDevCreate
- uglLQ16BitDevDestroy
- uglLQ16BitInfo

```
UGL_UGI_DRIVER * uglLQ16BitDevCreate
(
    UGL_UINT32 instance,
    UGL_UINT32 param1,  /* intLevel */
    UGL_UINT32 param2   /* intVector */
)
{
    …, …
    /* Set Standard Driver API Functions */
    /* 填写驱动要调用的各种函数
       这些函数大部分都是 WindML 提供的，并且这些函数都支持 TFT/STN 的 LCD */
        pDriver->bitmapBlt      = uglGeneric16BitBitmapBlt;
        pDriver->bitmapCreate       = uglGeneric16BitBitmapCreate;
        pDriver->bitmapDestroy      = uglGeneric16BitBitmapDestroy;
        pDriver->bitmapRead     = uglGeneric16BitBitmapRead;
        pDriver->bitmapWrite        = uglGeneric16BitBitmapWrite;
        pDriver->bitmapStretchBlt   = uglGeneric16BitBitmapStretchBlt;

        /* lcd special function */
        pDriver->modeAvailGet       = uglLQ16BitModeAvailGet;
```

```
        pDriver->modeSet        = uglLQ16BitModeSet;

        pDriver->monoBitmapBlt      = uglGeneric16BitMonoBitmapBlt;
        pDriver->monoBitmapCreate   = uglGeneric16BitMonoBitmapCreate;
        pDriver->monoBitmapDestroy  = uglGeneric16BitMonoBitmapDestroy;
        pDriver->monoBitmapRead     = uglGeneric16BitMonoBitmapRead;
        pDriver->monoBitmapWrite    = uglGeneric16BitMonoBitmapWrite;
        pDriver->monoBitmapStretchBlt   =
uglGeneric16BitMonoBitmapStretchBlt;
        pDriver->transBitmapBlt     = uglGeneric16BitTransBitmapBlt;
        pDriver->transBitmapCreate  = uglGenericTransBitmapCreate;
        pDriver->transBitmapDestroy = uglGenericTransBitmapDestroy;
        pDriver->transBitmapRead    = uglGenericTransBitmapRead;
        pDriver->transBitmapWrite   = uglGenericTransBitmapWrite;
        pDriver->transBitmapStretchBlt  =
uglGeneric16BitTransBitmapStretchBlt;
        pDriver->transBitmapCreateFromDdb                                   =
uglGenericTransBitmapCreateFromDdb;
        pDriver->colorAlloc     = uglGenericColorAllocDirect;
        pDriver->colorFree      = UGL_NULL;
        pDriver->clutSet        = UGL_NULL;
        pDriver->clutGet        = UGL_NULL;
        pDriver->colorConvert           = uglGeneric16BitColorConvert;
        pDriver->cursorBitmapCreate = uglGenericCursorBitmapCreate;
        pDriver->cursorBitmapDestroy    = uglGenericCursorBitmapDestroy;
        pDriver->cursorInit     = uglGenericCursorInit;
        pDriver->cursorDeinit       = uglGenericCursorDeinit;
        pDriver->cursorHide     = uglGenericCursorHide;
        pDriver->cursorImageGet     = uglGenericCursorImageGet;
        pDriver->cursorImageSet     = uglGenericCursorImageSet;
        pDriver->cursorMove     = uglGenericCursorMove;
        pDriver->cursorPositionGet  = uglGenericCursorPositionGet;
        pDriver->cursorOff      = uglGenericCursorOff;
        pDriver->cursorOn       = uglGenericCursorOn;
        pDriver->cursorShow     = uglGenericCursorShow;

        /* lcd special functin */
        pDriver->destroy        = uglLQ16BitDevDestroy;

        pDriver->ellipse        = uglGenericEllipse;
        pDriver->gcCopy         = uglGenericGcCopy;
        pDriver->gcCreate       = uglGenericGcCreate;
        pDriver->gcDestroy      = uglGenericGcDestroy;
        pDriver->gcSet          = uglGenericGcSet;
```

```
        /* lcd special functin */
        pDriver->info           = uglLQ16BitInfo;

        pDriver->line           = uglGenericLine;
        pDriver->pixelGet       = uglGeneric16BitPixelGet;
        pDriver->pixelSet       = uglGeneric16BitPixelSet;
        pDriver->polygon        = uglGenericPolygon;
        pDriver->rectangle      = uglGenericRectangle;
        pDriver->pageCopy       = uglGenericPageCopy;
        pDriver->pageCreate     = uglGenericPageCreate;
        pDriver->pageDestroy        = uglGenericPageDestroy;
        pDriver->pageDrawGet        = uglGenericPageDrawGet;

        /* lcd special functin */
        pDriver->pageDrawSet        = NULL; /* uglLQPageDrawSet; */

        pDriver->pageVisibleGet     = uglGenericPageVisibleGet;

        /* lcd special functin */
        pDriver->pageVisibleSet     = NULL; /* uglLQPageVisibleSet; */

        pDriver->memPoolCreate      = uglGenericMemPoolCreate;
        pDriver->memPoolDestroy     = uglGenericMemPoolDestroy;

        /* Set Generic Driver Functions */
        pGenDriver->bresenhamLine   = uglGeneric16BitBresenhamLine;
        pGenDriver->fbPixelGet      = uglGeneric16BitFbPixelGet;
        pGenDriver->fbPixelSet      = uglGeneric16BitFbPixelSet;
        pGenDriver->fill        = uglGeneric16BitFill;
        pGenDriver->hLine       = uglGeneric16BitHLine;
        pGenDriver->rectFill        = uglGeneric16BitRectFill;
        pGenDriver->vLine       = uglGeneric16BitVLine;

        /* lcd special functin */
        pGenDriver->gpWait      = uglLQgpWait;
        }
    ..., ...
    return (pDriver);
    }
```

第五步：在\target\src\ugl\driver\graphics\uptech2410 建立 LQ080V3DG01ini.c 文件；

LQ080V3DG01ini.c 是 LCD 的硬件初始化程序。

```
UGL_MODE uglLQ16BitModes[] =
```

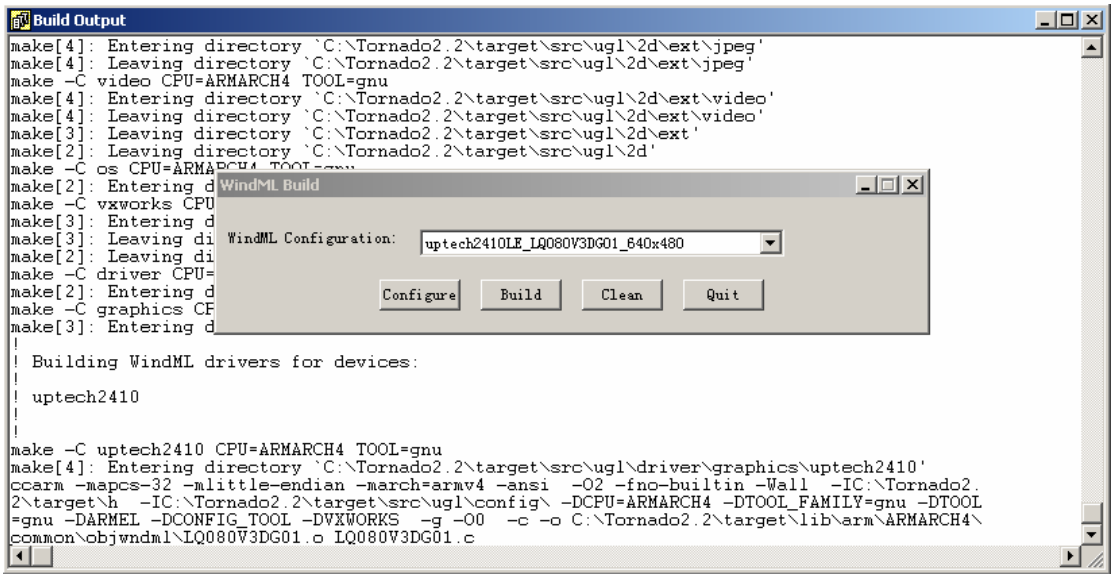
```
{
    /* 640, 480, 16Bit Color */
    {640, 480, 16, 60, UGL_MODE_FLAT_PANEL, UGL_MODE_DIRECT_COLOR},
};

UGL_INT32 numLQ16BitElements = NELEMENTS(uglLQ16BitModes);
/* 上面这段程序定义 LCD 的工作模式 */

UGL_STATUS uglLQ16BitModeSet
(
    UGL_UGI_DRIVER * pDriver,
    UGL_MODE * pMode
)
{
    UGL_GENERIC_DRIVER * pGenDriver = (UGL_GENERIC_DRIVER *)pDriver;
    UGL_LQ_DRIVER * pCTDriver = (UGL_LQ_DRIVER *)pDriver;
    UGL_UINT32 videoMemSize;
    void * videoMemAddress;
    void * address;
    UGL_UINT32 saveValue;
    UGL_INT32 index;
    UGL_BOOL allocate;
    UGL_LQ_DDB *pDdb;
    UGL_UINT8 byteValue;
    int dsplySize;
    void* dsplyBuffer;
    /* First time through, pMode will be NULL */
    if (pDriver->pMode == UGL_NULL)
    {
    allocate = UGL_TRUE;
    }
    else
    allocate = UGL_FALSE;
    /* 获取工作模式，就是从上面定义的 uglLQ16BitModes 里面提取宽、高、颜色深
度的信息 */
    index = uglGenericModeFind(uglLQ16BitModes, pMode, numLQ16BitElements);

    if (index > -1)
    {
    pDriver->pMode = &uglLQ16BitModes[index];
    }
    else
    return(UGL_STATUS_ERROR);
     lcd_ctrl_init(pGenDriver->fbAddress); /* 把在 sysWindML.c 里面建立的 LCD
```

```
内存地址填入驱动的数据结构中 */
    pGenDriver->videoMemPoolId = UGL_NULL;
    /* 后面的程序可以参考 WindRiver 提供的代码，并没有区别 */
    …, …
    }
```

其中 lcd_ctrl_init 函数是针对 2410 的 LCD Controller 编写的实际的 2410 的 LCD 相关寄存器的配置函数，在这里就不作详细的描述了。

5.4 编译 WindML

在 Tools>WindML 下面点击 Build 就会编译 WindML 的库和例子程序了。



在\target\lib\objARMARCH4gnuApps 目录下面可以看到编译好的例子程序，后面带有_ugl 的是把 WindML 的库函数集成在一起的目标文件，而没有带_ugl 的则是单纯的例子目标文件。可以在系统启动以后，通过 Target Server 把目标文件下载到目标机。

4.6、WindML 例程分析

通过下面两个例子的分析，可以对 WindML 应用程序的开发流程有个一个了解。

4.6.1 wexbasic 实例分析

target\src\ugl\example\basic\wexbasic.c

```
int windMLExampleBasic (void)
{
/* For registry lookups. */
/* 为了查询注册的多媒体设备 */
UGL_REG_DATA *pRegistryData;
```

```
    /* An int for miscellaneous counters,etc. */

    int i;

    /*
    * The UGL Graphics Context (GC) is an important concept in UGL.
    * The GC contains data that various UGL functions can use.
    * If there were no such thing as a GC, most function calls
    * would require several more parameters to communicate that
    * information.
    */
    /* GC 是图形环境， 它包括了各种可以使用的图形函数 */
    UGL_GC_ID gc;

    /*
    * This structure is filled in with data about the frame buffer
    * by the uglInfo() function.
    */
    /* LCD 缓冲信息 */
    UGL_MODE_INFO modeInfo;

    /*
    * The device ID is critical to the operation of UGL.  It
    * identifies individual "devices" (display adapter, keyboard,
    * font engine, etc.) to functions that may be able to work
    * with more than one device.
    */
    /* 图形设备 ID */
    UGL_DEVICE_ID devId;

    /*
    * As mentioned in the comments to patternData, the fill pattern
    * is defined as a monochrome bitmap for certain drawing primitives.
    * To create the bitmap (UGL_MDDB_ID), a Device Independent Bitmap
    * (DIB) is used; in particular a monochrome DIB (UGL_MDIB).  The
    * "DDB" in UGL_MDDB_ID stands for Device Dependent Bitmap.  Often
    * the DDB is simply referred to as a "bitmap" and the DIB is
    * called a "dib" (it's easier to pronounce dib than ddb).
    */

    UGL_MDIB patternDib;
    UGL_MDDB_ID patternBitmap;

    /*
```

```
    * Initialize UGL.  Must do this before trying to do anything
    * else within UGL/WindML.
    */
    /* 初始化 UGL */
    uglInitialize();

    /* Obtain the device identifier for the display */
    /* 获取注册的图形设备信息 */
    pRegistryData = uglRegistryFind (UGL_DISPLAY_TYPE, 0, 0, 0);
    if (pRegistryData == UGL_NULL)
        {
        uglDeinitialize();
        return (UGL_STATUS_ERROR);
        }
    devId = (UGL_DEVICE_ID)pRegistryData->id;

    /* Obtain the input service */
    /* 获取注册的输入设备信息 */
    pRegistryData = uglRegistryFind (UGL_INPUT_SERVICE_TYPE, 0, 0, 0);
    if (pRegistryData == UGL_NULL)
        {
        uglDeinitialize();
        return (UGL_STATUS_ERROR);
        }
    inputServiceId = (UGL_INPUT_SERVICE_ID)pRegistryData->id;

    /*
    * Obtain the dimensions of the display.  We will use these
    * to center some of our objects.
    */
    /* 获取显示设备的信息， 包括高、宽等 */
    uglInfo(devId, UGL_MODE_INFO_REQ, &modeInfo);
    displayWidth = modeInfo.width;
    displayHeight = modeInfo.height;

    /*
    * Create a graphics context.  Default values are set during
    * the creation.
    */
    /* 建立图形环境 */
    gc = uglGcCreate(devId);

    /*
    * Initialize colors. UGL maintains a Color Look-Up Table (CLUT)
```

```
    * for devices that do not represent colors directly.  Essentially
    * some hardware is only able to represent a subset of colors at
    * any given time.  To manage which colors will be available for
    * rendering, UGL uses color allocation.  If the hardware is able
    * to represent colors directly, then the uglColorAlloc() function
    * still works, but it is then essentially a no-op.
    *
    * We have set up for 16 colors, so here we will "allocate" them
    * within UGL's CLUT (sometimes referred to as a "palette").  Colors
    * can also be de-allocated, or freed, with uglColorFree.
    *
    * Since the ARGB's are intermingled with the UGL_COLORs in
    * the colorTable, we must allocate each color individually.
    * If the ARGB's had a contiguous array of ARGBs, and likewise for
    * the UGL_COLORs, a single uglColorAlloc call could be made.
    * (see the windows example).
    */
    /*  初始化颜色  */
    uglColorAlloc (devId, &colorTable[BLACK].rgbColor, UGL_NULL,
                   &colorTable[BLACK].uglColor, 1);
    uglColorAlloc(devId, &colorTable[BLUE].rgbColor, UGL_NULL,
                  &colorTable[BLUE].uglColor, 1);
    uglColorAlloc(devId, &colorTable[GREEN].rgbColor, UGL_NULL,
                  &colorTable[GREEN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[CYAN].rgbColor, UGL_NULL,
                  &colorTable[CYAN].uglColor, 1);

    uglColorAlloc(devId, &colorTable[RED].rgbColor, UGL_NULL,
                  &colorTable[RED].uglColor, 1);
    uglColorAlloc(devId, &colorTable[MAGENTA].rgbColor, UGL_NULL,
                  &colorTable[MAGENTA].uglColor, 1);
    uglColorAlloc(devId, &colorTable[BROWN].rgbColor, UGL_NULL,
                  &colorTable[BROWN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTGRAY].rgbColor, UGL_NULL,
                  &colorTable[LIGHTGRAY].uglColor, 1);

    uglColorAlloc(devId, &colorTable[DARKGRAY].rgbColor, UGL_NULL,
                  &colorTable[DARKGRAY].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTBLUE].rgbColor, UGL_NULL,
                  &colorTable[LIGHTBLUE].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTGREEN].rgbColor, UGL_NULL,
                  &colorTable[LIGHTGREEN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTCYAN].rgbColor, UGL_NULL,
                  &colorTable[LIGHTCYAN].uglColor, 1);
```

```
    uglColorAlloc(devId, &colorTable[LIGHTRED].rgbColor, UGL_NULL,
                  &colorTable[LIGHTRED].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTMAGENTA].rgbColor, UGL_NULL,
                  &colorTable[LIGHTMAGENTA].uglColor, 1);
    uglColorAlloc(devId, &colorTable[YELLOW].rgbColor,  UGL_NULL,
                  &colorTable[YELLOW].uglColor, 1);
    uglColorAlloc(devId, &colorTable[WHITE].rgbColor,  UGL_NULL,
                  &colorTable[WHITE].uglColor, 1);

    /*
    * Here is where we actually create the bitmap for the fill
    * pattern.  We are not going to do much with bitmaps here, just
    * use one for a fill pattern.  Bitmaps are a huge topic, so for
    * more information about bitmaps and associated operations, look
    * at the bitmap example and WindML documentation.
    */

    patternDib.width = patternDib.stride = patternData.width;
    patternDib.height = patternData.height;
    patternDib.pImage = patternData.data;
    patternBitmap = uglMonoBitmapCreate(devId, &patternDib,
                                        UGL_DIB_INIT_DATA, 0, UGL_NULL);

    /* Get the screen ready for some drawing. */
    /* 清屏 */
    ClearScreen(gc);

    /* ---------------------- Draw a line. --------------------------*/
    /* 画线 */
    /*
    * The batch commands are bracketing each of the different drawing
    * areas.  Since we are really just drawing everything and then
    * pausing, only one batch start at the beginning and then a
    * batch stop at the end is needed.  We'll keep them around each
    * section as a reminder.  The batch commands are optional; they
    * can enhance performance by locking resources for the UGL
    * functions called within their scope.  Otherwise each UGL function
    * will do its own resource locking and un-locking which adds over-
    * head.
    */
    /* 获得使用 LCD 的权限，从此开始，别的程序不能再访问 LCD */
    uglBatchStart(gc);
```

```
    /*
    * The foreground color is the color of the object being drawn.
    * There is a background color that is used to define the color
    * of solid fills (no pattern is defined).  Both of these values
    * are part of the Graphics Context (GC).  So here we are changing
    * the GC.  If the GC didn't contain the color, we would have to
    * pass them to the line drawing function as parameters (since the
    * GC does contain the colors, passing them as parameters is not
    * an option in UGL).
    */
    /* 设置线背景色 */
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);

    /* Draw the line!  Diagonal from upper left to lower right. */
    /* 画线：从坐标 0,0 到 LCD 的右下角 */
    uglLine(gc, 0, 0, displayWidth - 1, displayHeight - 1);
    /* 使用 LCD 资源结束 */
    uglBatchEnd(gc);

    /* ------------------ Draw a dashed line. ----------------------    */
    /* 画虚线 */
    uglBatchStart(gc);

    /*
    * This is another change to the a GC value.  Change the style of
    * line; there are two options from which to choose: solid and dashed.
    */
    /* 设置线的类型为虚线，设置线的颜色 */
    uglLineStyleSet(gc, UGL_LINE_STYLE_DASHED);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);

    /* Diagonal from upper right to lower left */
    /* 画线：从右上角到左下角 */
    uglLine(gc, displayWidth - 1, 0, 0, displayHeight -1);
    uglBatchEnd(gc); /* 释放 LCD 资源 */


    /* ------------------ Draw a thick line. ----------------------    */
    /* 画粗线 */
    uglBatchStart(gc);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);
    uglLineStyleSet(gc, UGL_LINE_STYLE_SOLID);
    uglLineWidthSet(gc, 7);
    uglLine(gc, displayWidth / 2, 0, displayWidth / 2, displayHeight -1);
```

```
    uglBatchEnd(gc);

    /* -------------- Draw a thick dashed line. --------------------   */
/* 画粗虚线 */

    uglBatchStart(gc);
    uglLineStyleSet(gc, UGL_LINE_STYLE_DASHED);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);
    uglLineWidthSet(gc, 7);
    uglLine(gc, 0, displayHeight / 2, displayWidth - 1, displayHeight / 2);
    uglBatchEnd(gc);


    /* ------------------- Draw a rectangle. -------------------------    */
    /* 画矩形 */
    uglBatchStart(gc);

    /*
    * Here we are setting the pattern bitmap in the GC. Since it is in
    * the GC all subsequent drawing that does a fill will use this
    * pattern.
    */

    uglFillPatternSet(gc, patternBitmap);
    uglForegroundColorSet(gc, colorTable[WHITE].uglColor);
    uglBackgroundColorSet(gc, colorTable[GREEN].uglColor);
    uglLineStyleSet(gc, UGL_LINE_STYLE_SOLID);
    uglLineWidthSet(gc, 7);
    uglRectangle(gc, displayWidth / 8, displayHeight / 8, displayWidth / 4,
displayHeight / 4);
    uglBatchEnd(gc);

    /* ------------------- Draw polygons. ---------------------------    */
    /*画多边形 */
    uglBatchStart(gc);
    {
    int j;

    /*
    * poly[] is and array of points.  The uglPolygon() function actually
    * is looking for a list of position values and assumes they are
    * organized as alternating x and y coordinates (x, y, x, y, x, ...).
    * I am using a list of points since it creates a compatible list
    * of positions and I can then manipulate the coordinates a little
```

```
    * more conveniently.  It does require a cast in uglPolygon's parms
    * though.  The polygon definition must be a closed path; the first
    * point and the last point are equal.  Polygon lines may cross
    * without a problem.
    */

    UGL_POINT                                   poly[]                                  =
{{0,17},{30,0},{30,10},{70,0},{66,6},{90,3},{60,10},{64,5},{24,17},{24,10},{0,1
7}};

    /*
    * Since the fill pattern from before is still in the GC, these
    * polygons will have the same pattern.  These polygons are rather
    * flat, so the pattern doesn't show as more than a few pixels inside.
    * Also note that the colors are set differently and that the colors
    * are applied to the fill pattern as well.
    */

    uglForegroundColorSet(gc, colorTable[LIGHTCYAN].uglColor);
    uglBackgroundColorSet(gc, colorTable[BLACK].uglColor);
    uglLineWidthSet(gc, 1);
    uglLineStyleSet(gc, UGL_LINE_STYLE_SOLID);

    /*
    * Adjust the position of the entire polygon.  Add the same x
    * coordinate adjustment to all of the x-coordinates.  The same
    * is done for all of the y coordinates.
    */

    for (j=0;j<11;j++)
        {
        poly[j].x += displayWidth / 2 - 60;
        poly[j].y += displayHeight / 2 - 40;
        }

    /*
    * Just to make it interesting, draw five of these
    * polygons, spaced 15 pixels apart vertically.
    */

    for (i = 0; i < 5;i++)
        {
        int j;
        uglPolygon(gc, 11, (UGL_POS *)poly);
```

```
        for (j=0;j<11;j++)
            {
            poly[j].x += 10;
            poly[j].y += 15;
            }
        }
    }
    uglBatchEnd(gc);

    /* ---------------- Draw solid filled polygons. -------------------
     */

    uglBatchStart(gc);
    {
    UGL_POINT                                   poly[]                                  =
{{0,17},{30,0},{30,10},{70,0},{66,6},{90,3},{60,10},{64,5},{24,17},{24,10},{0,1
7}};

    int j, k;

    for (j=0;j<11;j++)
        {
        poly[j].x += displayWidth / 2 - 200;
        poly[j].y += displayHeight / 2 + 45;
        }

    /*
    * By setting the GC's fill pattern to UGL_NULL uglPolygon() will
    * assume a solid pattern for its fill.  This would be the default,
    * but since we used a different fill for the rectangle, we need to
    * set it back again now.
    */

    uglFillPatternSet(gc, UGL_NULL);
    uglForegroundColorSet(gc, colorTable[LIGHTCYAN].uglColor);
    uglBackgroundColorSet(gc, colorTable[CYAN].uglColor);
    uglLineWidthSet(gc, 1);
    uglLineStyleSet(gc, UGL_LINE_STYLE_SOLID);

    /* To make it even more interesting, draw three sets of 5 polygons. */

    for (k = 0; k < 3;k++)
        {
        for (i = 0; i < 5;i++)
```

```
            {
            uglPolygon(gc, 11, (UGL_POS *)poly);
            for (j=0;j<11;j++)
                {
                poly[j].x += 10;
                poly[j].y += 15;
                }
            }

        /* Move each of the 3, five polygon, sets to a different position. */

        for (j=0;j<11;j++)
            {
            poly[j].x += 100;
            poly[j].y -= 60;
            }

        }
    }
    uglBatchEnd(gc);

    /* --------------------- Draw an ellipse. ------------------------    */
    /* 画椭圆 */
    /*
    * With the drivers that ship with WindML, the ellipse only draws
    * single pixel width borders.
    */

    uglBatchStart(gc);
    {
        uglForegroundColorSet(gc, colorTable[BLUE].uglColor);
        uglBackgroundColorSet(gc, colorTable[RED].uglColor);
        uglEllipse(gc, 5 * displayWidth / 8, displayHeight / 6, 15 * displayWidth
/ 16, displayHeight / 3, 0, 0, 0, 0);
    }
    uglBatchEnd(gc);

    /* ---------------- Draw a pie sliced ellipse. -------------------    */
    /* 画部分椭圆 */
    /*
    * The coordinates chosen here put this pie slice on top of the
    * previous ellipse.
    */
```

```
    uglBatchStart(gc);
    {
        uglForegroundColorSet(gc, colorTable[YELLOW].uglColor);
        uglBackgroundColorSet(gc, colorTable[GREEN].uglColor);
        uglEllipse(gc, 5 * displayWidth / 8, displayHeight / 6, 15 * displayWidth
/ 16, displayHeight / 3, 5 * displayWidth / 8, displayHeight / 6, 5 * displayWidth
/ 8, displayHeight / 3);
    }
    uglBatchEnd(gc);

    /* -------- Wait for the signal to quit from basicStop. -------------
    */

    wexPause();

    /* ---------------- Clean everything up and exit. -----------------
    */

    uglMonoBitmapDestroy(devId, patternBitmap);
    uglGcDestroy (gc);

    uglDeinitialize();

    return(0);
    }
```

4.6.2 ugldemo 实例分析

target\src\ugl\example\demo\guldemo.c

通用的 ugl 应用程序流程如下：

```
    /* 第一步： 初始化 UGL 模块 */
    /* Initialize UGL */
    if (uglInitialize() == UGL_STATUS_ERROR)
    {
        printf("uglInitialize error.\n");
        return;
    }

    /* 第二步：获取显示设备 ID */
    /* Obtain display device identifier */
    pRegistryData = uglRegistryFind (UGL_DISPLAY_TYPE, 0, 0, 0);
```

```
    if (pRegistryData == UGL_NULL)
    {
        printf("Display not found. Exiting.\n");
        uglDeinitialize();
        return;
    }
    devId = (UGL_DEVICE_ID)pRegistryData->id;

    /* 第三步：建立图形环境 */
    /* Create a graphics context */
    gc = uglGcCreate(devId);


    /* 第四步：建立字体 */
    /* Create Fonts */
    pRegistryData = uglRegistryFind (UGL_FONT_ENGINE_TYPE, 0, 0, 0);
    if (pRegistryData == UGL_NULL)
    {
        printf("Font engine not found. Exiting.\n");
        uglDeinitialize();
        return;
    }
    fontDrvId = (UGL_FONT_DRIVER_ID)pRegistryData->id;

    /* 第五步：设置字体驱动的参数 */
    uglFontDriverInfo(fontDrvId, UGL_FONT_TEXT_ORIGIN, &textOrigin);

    /* 第六步：查询字体库，找到相应的字体 */
    uglFontFindString(fontDrvId, "familyName=Lucida; pixelSize = 12",
&systemFontDef);

    if ((fontId = uglFontCreate(fontDrvId, &systemFontDef)) == UGL_NULL)
    {
        printf("Font not found. Exiting.\n");
        return;
    }
    /* 第七步：获取 LCD 显示缓冲区，也就是 LCD 的尺寸 */
    /* Obtain the dimensions of the display */
    uglInfo(devId, UGL_FB_INFO_REQ, &fbInfo);
    displayWidth = fbInfo.width;
    displayHeight = fbInfo.height;
    printf("LCD display width = %d, height = %d.\n", displayWidth,
displayHeight);
```

```
    /*第八步：初始化颜色 */
    /*  Initialize colors. */
    uglColorAlloc (devId, &colorTable[BLACK].rgbColor, UGL_NULL,
                   &colorTable[BLACK].uglColor, 1);
    uglColorAlloc(devId, &colorTable[BLUE].rgbColor, UGL_NULL,
                  &colorTable[BLUE].uglColor, 1);
    uglColorAlloc(devId, &colorTable[GREEN].rgbColor, UGL_NULL,
                  &colorTable[GREEN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[CYAN].rgbColor, UGL_NULL,
                  &colorTable[CYAN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[RED].rgbColor, UGL_NULL,
                  &colorTable[RED].uglColor, 1);
    uglColorAlloc(devId, &colorTable[MAGENTA].rgbColor, UGL_NULL,
                  &colorTable[MAGENTA].uglColor, 1);
    uglColorAlloc(devId, &colorTable[BROWN].rgbColor, UGL_NULL,
                  &colorTable[BROWN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTGRAY].rgbColor, UGL_NULL,
                  &colorTable[LIGHTGRAY].uglColor, 1);
    uglColorAlloc(devId, &colorTable[DARKGRAY].rgbColor, UGL_NULL,
                  &colorTable[DARKGRAY].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTBLUE].rgbColor, UGL_NULL,
                  &colorTable[LIGHTBLUE].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTGREEN].rgbColor, UGL_NULL,
                  &colorTable[LIGHTGREEN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTCYAN].rgbColor, UGL_NULL,
                  &colorTable[LIGHTCYAN].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTRED].rgbColor, UGL_NULL,
                  &colorTable[LIGHTRED].uglColor, 1);
    uglColorAlloc(devId, &colorTable[LIGHTMAGENTA].rgbColor, UGL_NULL,
                  &colorTable[LIGHTMAGENTA].uglColor, 1);
    uglColorAlloc(devId, &colorTable[YELLOW].rgbColor,  UGL_NULL,
                  &colorTable[YELLOW].uglColor, 1);
    uglColorAlloc(devId, &colorTable[WHITE].rgbColor,  UGL_NULL,
                  &colorTable[WHITE].uglColor, 1);
```

然后就可以进行相应的图形的开发工作了。后面的程序主要是进行画线、画圆等等。这些函数可以在 wind_medialib_sdk_programmers_guide_3.0.2.pdf 文件找到详细的说明。并且还可在 wind_medialib_api_reference_3.0.2.pdf 和在线帮助中找到说明。

4.7、2410 LCD WindML 软件使用方法

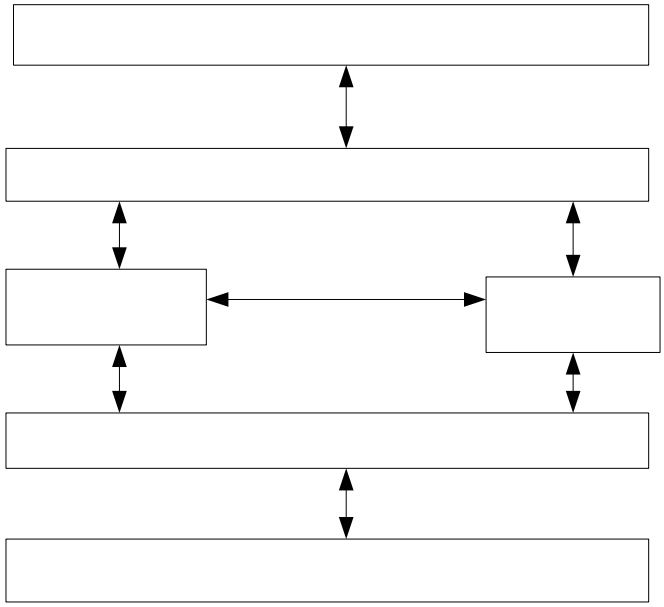
1 拷贝 BSP 目录下面的 sysWindML.c 到 2410 的 BSP 目录下面。
2 修改 Makefile，添加
EXTRA_MODULES += sysWindML.o
或者
通过 Tornado 图形开发环境建立 Bootable Project 的时候手动添加 sysWindML.c 这个文件。
3 备份 target\src\ugl\example 目录，拷贝 host 和 targe 两个目录到 Tornado 安装目录，提示是否覆盖时，选择 Yes。
4 运行 Tools->WindML，选择 uptech2410LE_LQ080V3DG01_640x480，可以按 Config 进行配置，按 Build 进行编译。编译 WindML 的结果是
target\h\ugl\config\configTool\uptech2410LE_LQ080V3DG01_640x480_windml.o 和
target\h\ugl\config\configTool\uptech2410LE_LQ080V3DG01_640x480_windml.a
5 编译的例子程序在 target\lib\objARMARCH4gnuApps\目录下面

第五章 应用实验

5.1 串口

5.1.1 串口概述

在 VxWorks 中，串口是一种特殊的字符型设备，它的驱动程序并不是直接挂在 I/O 系统中，而是通过在其上构建的 tty 驱动及其支持库 tyLib 接入，如下图所示。



ttyDrv 是一个虚拟的驱动，在 tyLib 库的支持下，统一管理 xxDrv(硬件驱动)，按照字符设备驱动接入到 I/O 系统中，并提供读写缓冲区，完成各种与硬件无关的功能，提供标准的 I/O 接口接入 I/O 系统，并统一处理相关的终端功能，如命令行编辑、字符回显等。xxDrv 作为具体的硬件驱动，除了完成硬件相关的操作外，还要对上一层提供操作接口。此部分内容可参考 VxWorks Programmer’s Guaid : 4.7.1 Serial I/O Devices (Terminal and Pseudo-Terminal Devices).和 tyLib 和中的相关内容。

在 VxWorks 中，使用其串行设备，必须遵循串行设备与其驱动程序对应规则，如下所示：

设备	驱动	设备名	描述
Tty	ttyDrv	“/tyCo/”	串行设备

5.1.2 串口操作

与 UNIX 相同，VxWorks 所有的 I/O 设备都被当作文件来存取。这里的文件指的是 I/O 系统中的操作对象，可能是一个物理设备，或一个任务管道，或文件系统的一个文件。这里以串口为例，”/tyCo/0”表示一个物理串口，”/tyCo/”是设备名（设备名称是 VxWorks 系统规定的），表明此设备为串行设备，”0”是设备号。

在基本的 I/O 调用中，文件通过***文件描述符***来引用，当一个文件被打开后，一个文件描

述符被分配并返回；当文件被关闭后，文件描述符被释放。用户可以通过文件描述符来访问设备，进行读写以及 I/O 控制等操作。

在 VxWorks 中，串口作为 I/O 设备，支持基本的 I/O 调用：open, close, read, write, ioctrl.

5.1.2.1 open

open	
目标	打开一个文件，得到文件描述符（fd）
头文件	# include <ioLib.h>
函数原型	int fd = open(const char *name, int flags, int mode)
参数	name 文件名 flags 文件存取标志 mode 文件属性（当用 open 建立新文件时，需要填写，一般不用）
返回值	ERROR 错误 fd 返回文件描述符

Open 可用的 Flags 如下表所示：

Flags	值	说明
O_RDONLY	0	只读打开
O_WRONLY	1	只写打开
O_RDWR	2	读写打开
O_CREAT	0x200	建立新文件

Flag 可以用对应的宏，也可以直接赋值，比如：

- serialFd = open(“/tyCo/0”,O_RDWR,0);
- serialFd = open(“/tyCo/0”, 2,0);

这两条语句的结果是一样的，都是以读写方式打开串口。

5.1.2.2 close

close	
目标	关闭文件，释放文件描述符（fd）
头文件	# include <ioLib.h>
函数原型	int result = close(int fd)
参数	fd 欲关闭的文件描述符
返回值	ERROR 错误 OK 成功

例如：

close(serialFd); 关闭打开的串口描述符。

5.1.2.3 read

read	
目标	把数据读到缓冲区
头文件	# include <ioLib.h>
函数原型	int nBytes = read(int fd, char* buffer, size_t maxbytes)
参数	fd 文件描述符 buffer 用来存放数据的目的缓冲区 maxbytes 最大读取的字节数
返回值	ERROR 错误 nBytes 实际所读取的字节数

例如：

```
int nBytes = read (serialFd, buf, 10);  /*从串口中最多读取 10 个字符到 buf 缓冲区*/
```

5.1.2.4 write

write	
目标	将缓存中的数据写入文件
头文件	# include <ioLib.h>
函数原型	int nBytes = read(int fd, char* buffer, size_t maxbytes)
参数	fd 文件描述符 buffer 缓存数据 maxbytes 要写的字节数
返回值	ERROR 错误 nBytes 实际所写入的字节数

例如：

```
write(serialFd,"Hello,World!\n",13);                    /* 向 串 口 写 入 数 据 "Hello,World!"*/
```

5.1.2.5 ioctl

ioctl 对文件（设备）执行各种 I/O 控制操作。

ioctl	
目标	对文件（设备）执行各种 I/O 控制操作
头文件	# include <ioLib.h>
函数原型	int result = ioctl(int fd, int function,

	int arg)
参数	fd 文件描述符 function 功能码 arg 参数
返回值	ERROR 错误 OK 成功

串口支持的功能码（function）如下表所示：

功能	**描述**
FIOBAUDRATE	设置波特率
FIOCANCEL	取消读或写
FIOFLUSH	丢弃所有输入/输出缓冲区里的字符
FIOGETNAME	取得文件描述符对应的文件名
FIOGETOPTIONS	取得当前设备的控制字
FIOSETOPTIONS	设置设备控制字
FIONREAD	取得输入缓冲区内未读的字符个数
FIONWRITE	取得输出缓冲区内的字符个数

通过 ioctl 的功能码 FIOSETOPTIONS,可以设置串口的各种模式。

模式	描述
OPT_LINE	选择行模式
OPT_ECHO	回显输入的字符
OPT_CRMOD	支持输入的字符 RETURN 为 NEWLINE(\n)；转换输出的 NEWLINE 为 RETURN-LINEFEED
OPT_TANDEM	支持 X-on/X-off 协议（Ctrl+Q 和 Ctrl+S）
OPT_MON_TRAP	特别的 ROM 驻留程序字符有效，缺省 Ctrl+X
OPT_7_BIT	将输入的字符转换为 7 位 ASCII 码
OPT_ABORT	支持 Shell 停止字符，缺省为 Ctrl+C
OPT_TERMINAL	设置上述所有的终端属性
OPT_RAW	取消上述的所有终端属性

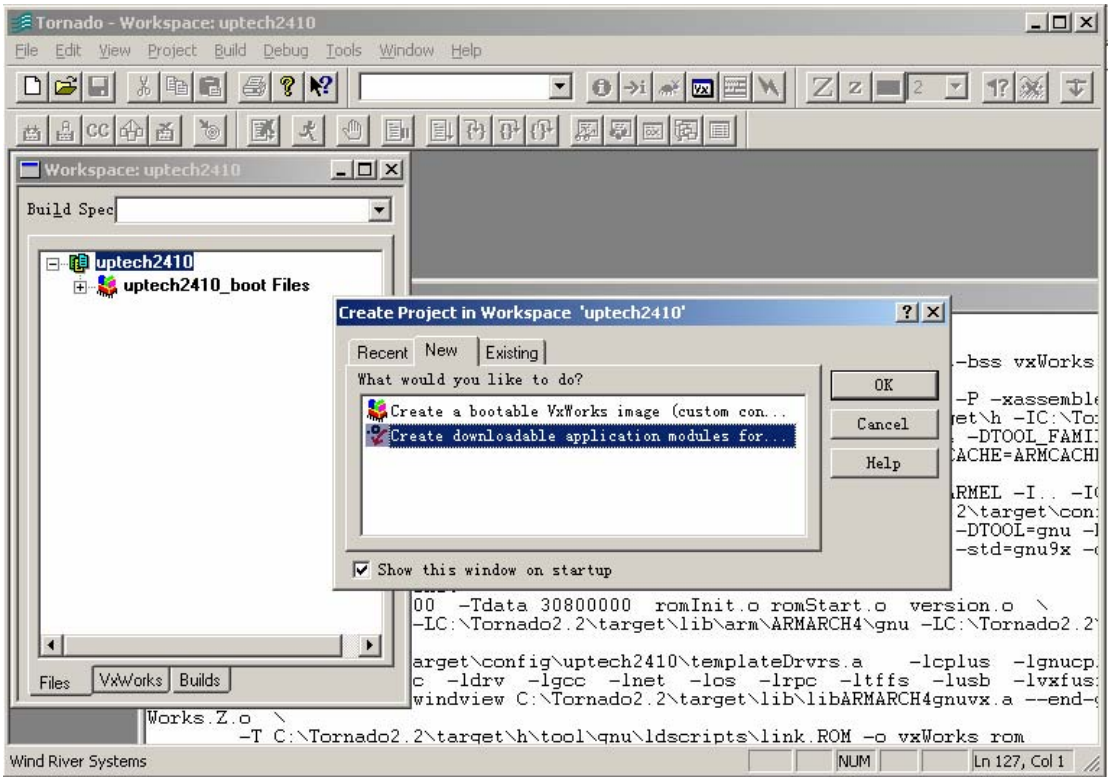
例如：

```
ioctl(serialFd, FIOBAUDRATE, 38400);  /*设置串口速率为 38400*/
ioctl(serialFd, FIOSETOPTIONS, OPT_RAW);     /*设置串口模式为 RAW 模式*/
ioctl (serialFd, FIONREAD, (int) &nNreadBytes); /* nNreadBytes 被赋值为输入缓冲区内未读的字符数*/
```

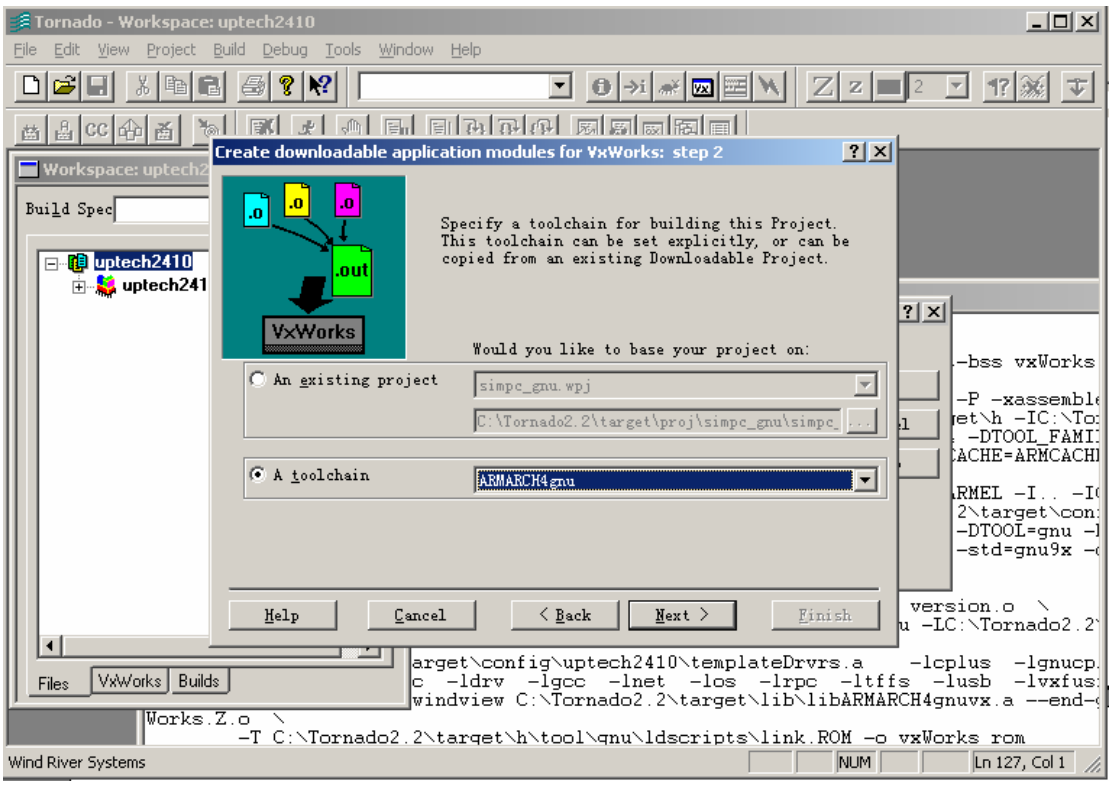
5.1.3 实验

5.1.3.1 使用 wirte 函数对串口进行写操作

第一步：建立可下载工程并且填入工程名称和路径。



第二步：选择编译工具选项，S3C2410 的编译选项应该是 ARMARCH4gnu。



第三步：在工程中添加文件，\src\Serial\serial.c

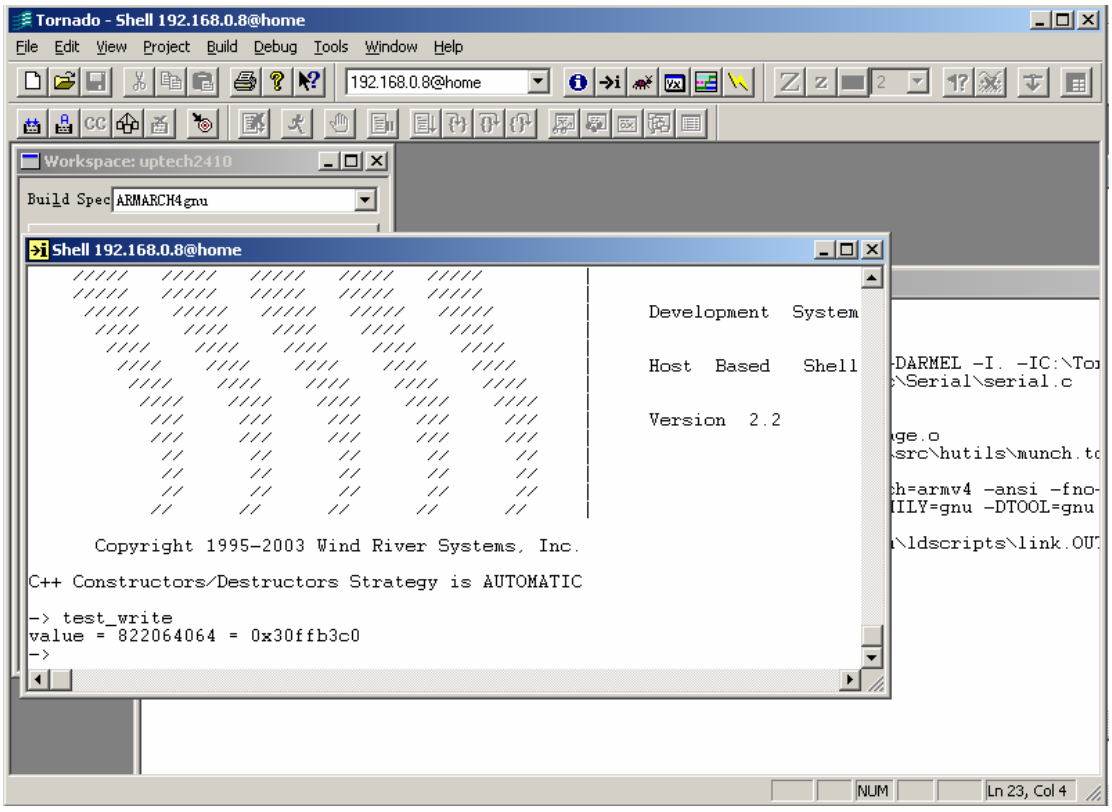
test_write（）函数。

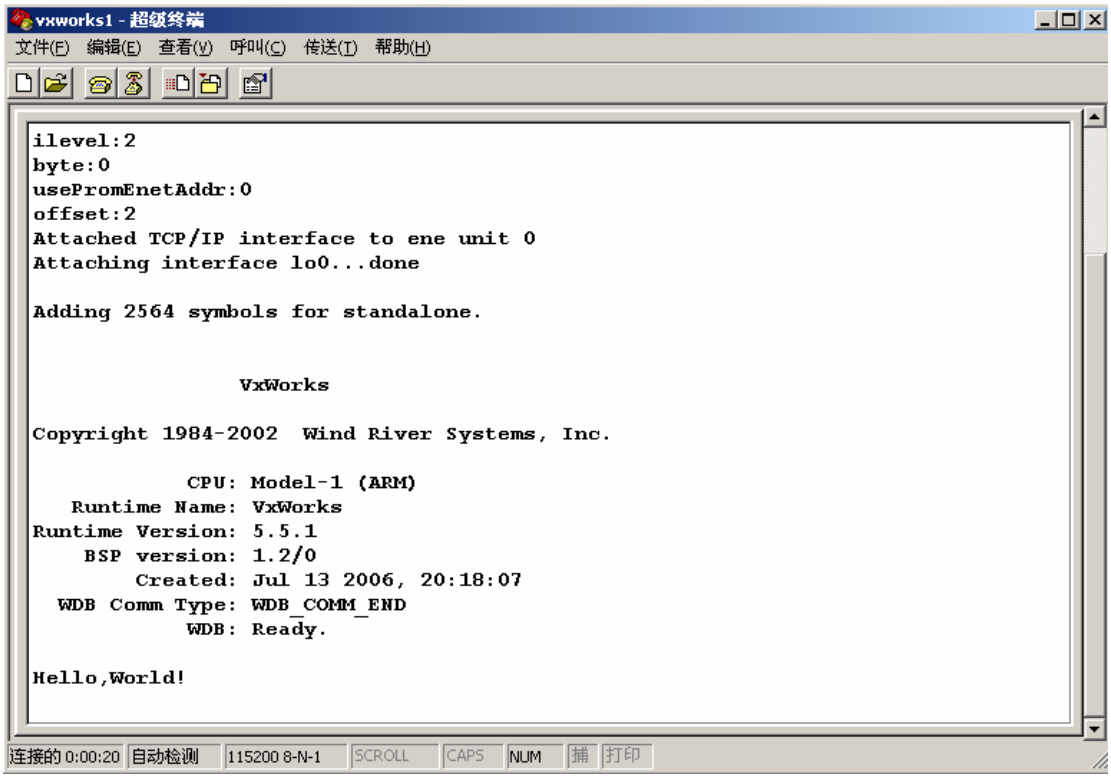
```
void test_write(void)
{
    int serialFd;
```

```
    serialFd = open("/tyCo/0",O_RDWR,0);/*以读写方式打开串口 1，得到相应的文件描述符 SerialFd*/
    write(serialFd,"Hello,World!\n",13);    /* 向 串 口 写 入 数 据 "Hello,World!"*/
    close(serialFd);                        /*关闭串口*/

}
```

在 Tornado 界面里面编译。

第三步：连接 Target Server，并且把编译好的结果下载到目标板中。在 Target Shell 或者 WindSh 下输入 test_write，屏幕上打印出”Hello,World!”,如下图所示：





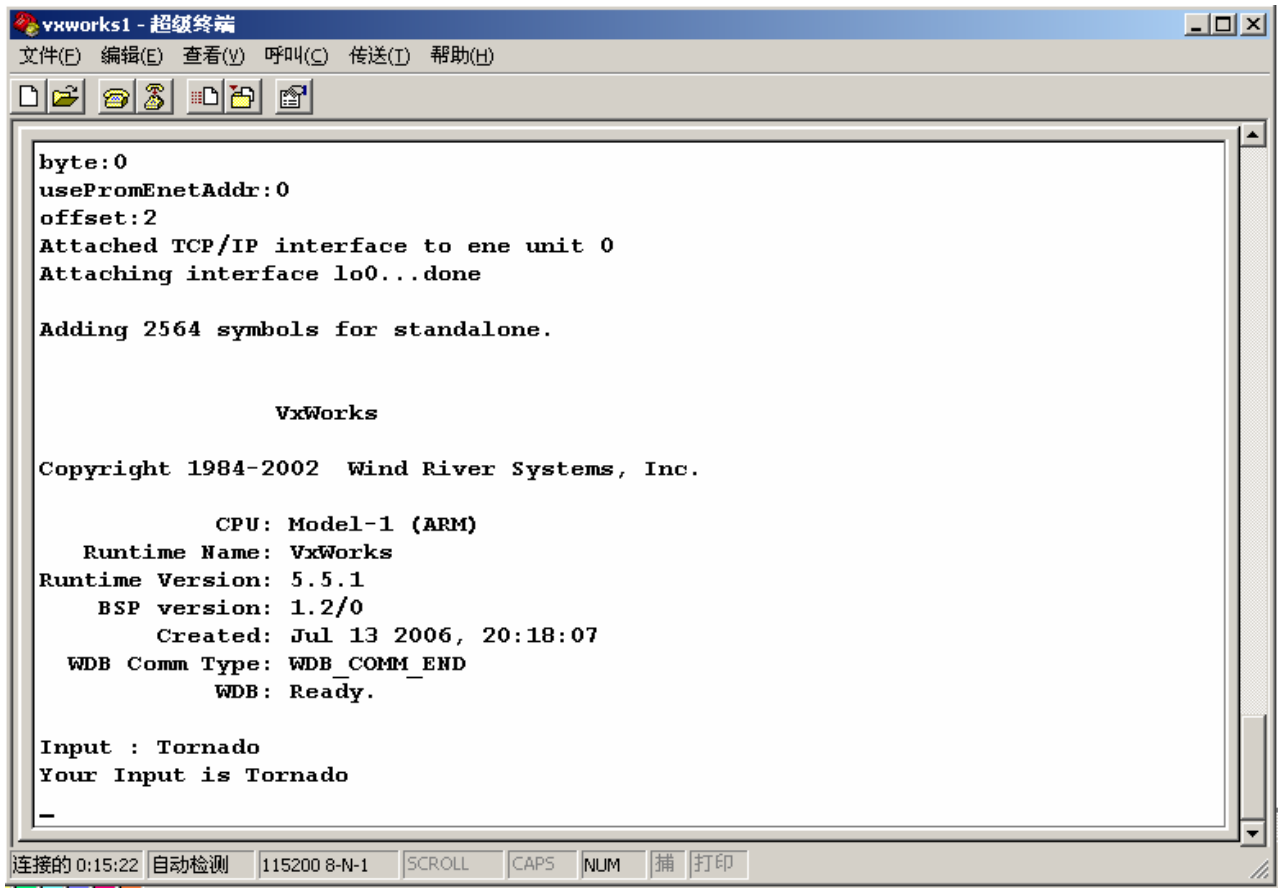
5.1.3.2 使用 read 函数对串口进行读操作

具体代码在\src\Serial\serial.c 文件中，test_read（）函数。由于这个函数和上一个 write 的粒子在同一个文件中，上次编译的结果就可以用在这个实验中。

```
void test_read(void)
{
    int serialFd;
    int n;
    char buf[BUFFSIZE];
    serialFd = open("/tyCo/0",O_RDWR,0);/*以读写方式打开串口 1，得到相应的文件描述符 SerialFd*/
    if(write(serialFd,"Input :  ", 8)  == ERROR) /*输出提示语 Input: */
        perror("Write 1 ");
    if((n = read(serialFd, buf, BUFFSIZE)) > 0) /*等待串口输入*/
        {
            if(write(serialFd,"Your Input is ", 15)  == ERROR)
                perror("Write 2 ");
            else
                if(write(serialFd, buf, n) != n) /*将从串口读到的数据写回串口*/
                    perror("Write 3 ");
        }
    else
        perror("read");
```

```
    close(serialFd);
}
```

在 Target Shell 或者 WindSh 下输入 test_read，函数把从串口读到的数据送到输出进行显示,如下图所示：



5.1.3.3 使用 ioctl 函数对串口进行控制

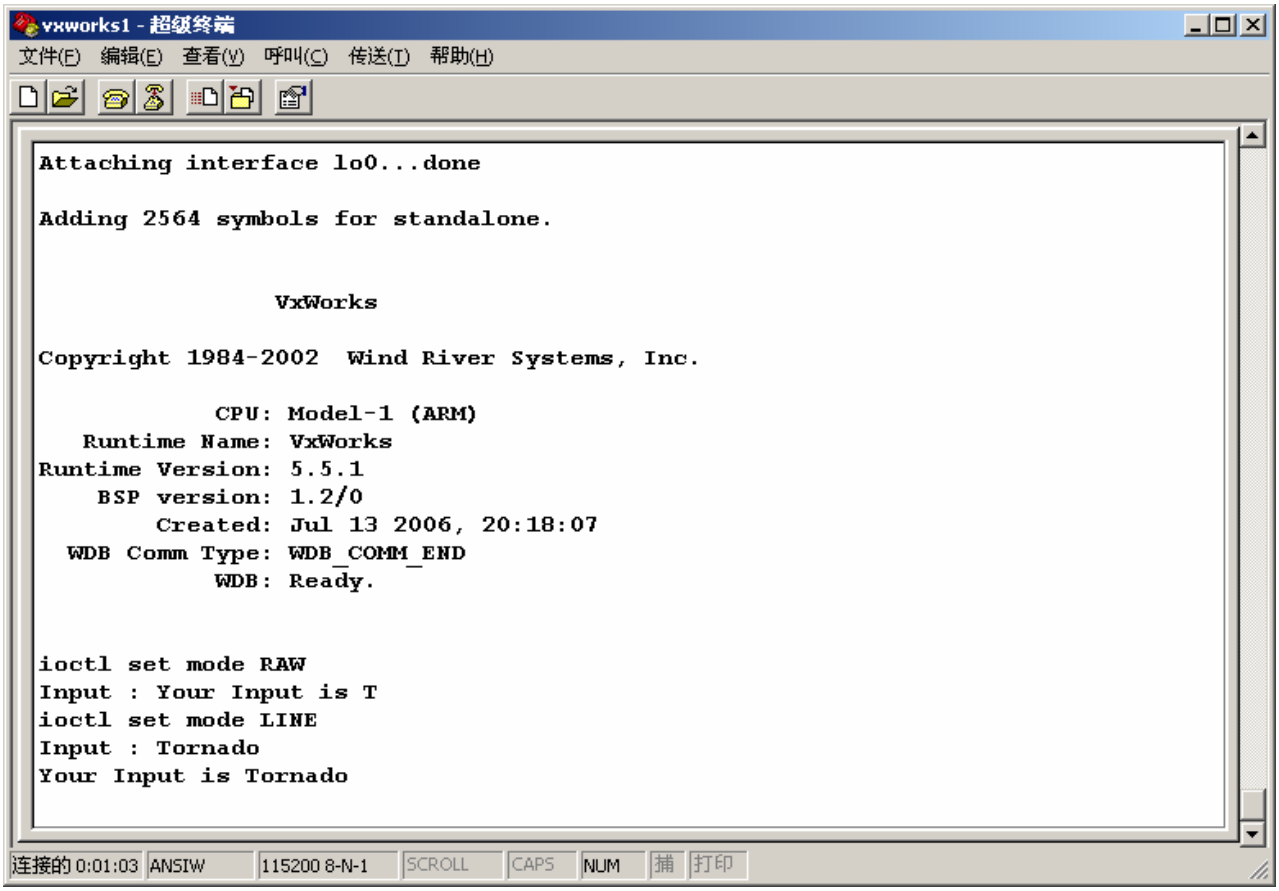
具体代码在\src\Serial\serial.c 文件中由于这个函数和上一个 write 的粒子在同一个文件中，上次编译的结果就可以用在这个实验中

```
void test_ioctl()
{
    int serialFd;
    serialFd = open("/tyCo/0",O_RDWR,0);

    /*设置串口速率为 38400*/
    if(ioctl(serialFd, FIOBAUDRATE, 38400)== ERROR)
        perror("ioctl 1");
    /*设置串口模式为 RAW 模式*/
```

```
    if(ioctl(serialFd, FIOSETOPTIONS, OPT_RAW)== ERROR)
        perror("ioctl 2");
    else
        printf("\n\rioctl set mode RAW\n\r");
    /*用 test_read 测试*/
    test_read();
    /*设置串口恢复成 LINE 模式,以及恢复其他终端属性*/
    if(ioctl(serialFd, FIOSETOPTIONS, OPT_TERMINAL)== ERROR)
        perror("ioctl 3");
    else
        printf("\nioctl set mode LINE\n");

    /*用 test_read 测试*/
    test_read();
    close(serialFd);

}
```

在 Target Shell 或者 WindSh 下输入 test_ioctl，串口首先被设置为 RAW 模式，然后恢复成 LINE 模式。

在 RAW 模式下每个输入的字符在用户输入时就立即生效,所以用户无法对正在进行的输入作修改，而对于 LINE 模式，所有的输入字符都被保存在缓冲区中直到一个 NEWLINE 字符被输入，而且在输入的过程中可以使用特殊字符（比如 Delete,Backspace 等）来修改输入。

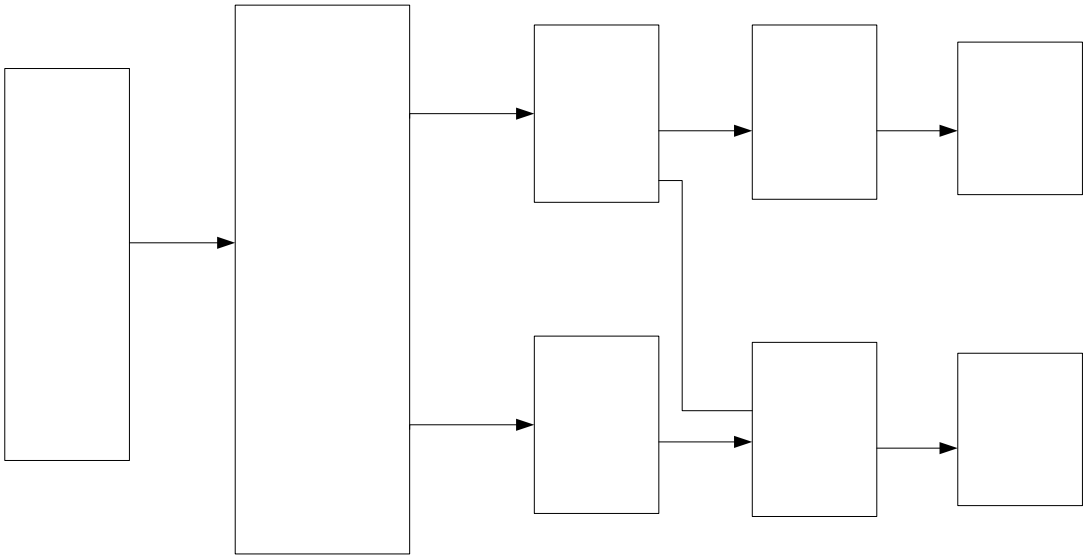


5.2 基于块设备的文件系统

块设备是 VxWorks 系统中重要的一种设备，具有下列特性：

- 支持文件结构；
- 随机访问能力；
- 使用多字节的数据块传输数据。

块设备的驱动程序不是直接挂接在 VxWorks 的 I/O 系统中。由于在 VxWorks 中，文件系统是作为驱动程序挂接在 I/O 系统中，而用户通过访问文件系统访问块设备。VxWorks 的 I/O 系统、文件系统及块设备驱动程序之间如下图所示：



5.2.1 VxWorks 支持的文件系统

■ 与 MS_DOS 文件系统兼容的 dosFs

dosFs 有诸多优点，它兼容于 MS-DOS5.0 文件系统，可以有效地利用磁盘空间，同时又保证文件访问的效率。与 MS-DOS 文件系统相同，dosFs 支持目录分级结构，在目录结构中可以包含更深的目录结构，出于访问效率的考虑，WindRiver 还对其进行了优化，dosFs 允许用户指定文件是否连续，从而加快文件访问速度。

■ 原始文件系统 rawFs

原始文件系统将块设备当作一个单独的文件，不支持目录结构和多文档结构，但原始文件系统使访问效率得到最大的提高。原始文件系统为用户提供了有效的磁盘管理办法，但是如果对方问没有什么特殊要求的话首选还是 dosfs。

■ 其他文件系统

VxWorks 同时还支持其他的文件系统：专门面向磁带系统的 tapeFs、面向 CDROM 的文件系统 cdromFs、网络文件系统 NFS、目标机服务器文件系统 TSFS 等，每一种文件系统都可以

和其他文件系统并存，而每一种块设备驱动程序既可以支持其中的一种也可以支持其中的多种格式。

5.2.2 文件系统的配置

不论对于何种文件系统，在使用前都要初始化文件系统、初始化块设备驱动程序并将其挂接到文件系统中。用户可以使用文件系统的指定函数初始化文件系统，使用 xxDrv（）安装块设备驱动程序，使用 xxDevCreate()创建并初始化设备。步骤如下：

1) 调用 xxDrv()函数安装块设备驱动程序；
2) 执行 pBlkDev = xxDevCreate()操作，创建设备。在这里 pBlkDev 是由 xxDecCreate()返回的指向 BLK_DEV 的指针。
3) 接下来就可以用 pBlkDev 作为参数初始化文件系统了。以 dosFs 为例，初始过程如下：
   dosFsMkfs(fsName, pBlkDev);
4) 随后用户就可以用正常的办法访问文件了。
   fd = open(fsName, O_RDWR, 0);

■ BLK_DEV 数据结构

```
typedef struct          /*BLK_DEV*/
    {
    FUNCPTR     bd_blkRd;           /*驱动程序读函数的入口地址*/
    FUNCPTR     bd_blkwrt;          /*驱动程序写函数的入口地址*/
    FUNCPTR     bd_ioctl;               /*驱动程序功能函数的入口地址*/
    FUNCPTR     bd_reset;               /*驱动程序重置函数的入口地址*/
    FUNCPTR     bd_statusChk;           /*驱动程序检查磁盘状态的函数的
入口地址*/
    BOOL        bd_removable;           /*可移动媒介标志*/
    ULONG       bd_nBlocks;         /*分区（磁盘）块数*/
    ULONG       bd_bytesPerBlk;     /*每块中包含的块的数目*/
    ULONG       bd_blksPerTrack;        /*每一个磁道包含的块的数目*/
    ULONG       bd_nHeads;          /*磁头数*/
    int             bd_retry;               /*遇到 I/O 错误后重试的错误
*/
    int         bd_mode;                /*O_RDONLY,O_WRONLY,O_RDWR*/
    BOOL        bd_readyChanged;        /*设备变更准备好*/

      }
```

■ dosFs 库函数

dosFsMkfs	
目标	创建一个 dosFs 文件系统并初始化设备
头文件	#include “dosFsLib.h”
函数原型	DOS_VOL_DESC* pDosVol = dosFsMkfs(char* volName, BLK_DEV*

	pBlkDev)
参数	volName 为设备指定的文件名 pBlkDev 指向设备的 BLK_DEV 结构体的指针
返回	NULL 错误 pDosVol 指向对应卷描述符的指针

例如：

DOS_VOL_DESC* pVolDesc = dosFsMkfs ("/ram0/", pBlkDev);/*加载 dosFs 文件系统*/

5.2.3 ramDrv

RAM 存储器是最容易使用的块设备，在物理上没有块的分界，可以使用任意大小的块。对 RAM 的操作是很方便的，RAM 也是最快的存储设备，但数据的组织比较麻烦。VxWorks 灵活的文件系统分层功能，让用户可以在任意能抽象成块设备的存储器上加载需要的文件系统。ramDrv 库就是用来完成 RAM 存储器的这种抽象工作的。上层文件系统将 RAM 的传地址空间抽象为名称空间，就可以用文件的标准操作来访问 RAM 空间，而不用管内存的分配和释放，不用管数据数据存放在何处。

RAM 块设备的提供了和其他块设备类似的创建函数：

ramDevCreate	
目标	创建一个 RAM 块设备
头文件	#include “ramDrv.h”
函数原型	BLK_DEV* pBlkDev = ramDevCreate (char* ramAddr, int bytesPerBlk, int blksPerTrack, int nBlocks, int blkOffset);
参数	ramAddr RAM 设备的起始地址，0 表示由系统帮助分配 bytesPerBlk 逻辑快的大小，0 表示缺省值 512 blksPerTrack 每个磁道的数据块数,0 表示一个磁道包括所有的块 nBlocks RAM 设备的块数 blkOffset 为偏移块数
返回	NULL 错误 pBlkDev 指向对应块设备结构体(BLK_DEV)的指针

例如：

pBlkDev = ramDevCreate (0, 512, 400, 400, 0); /*创建一个大小为 512*400 的 RAM 块设备*/

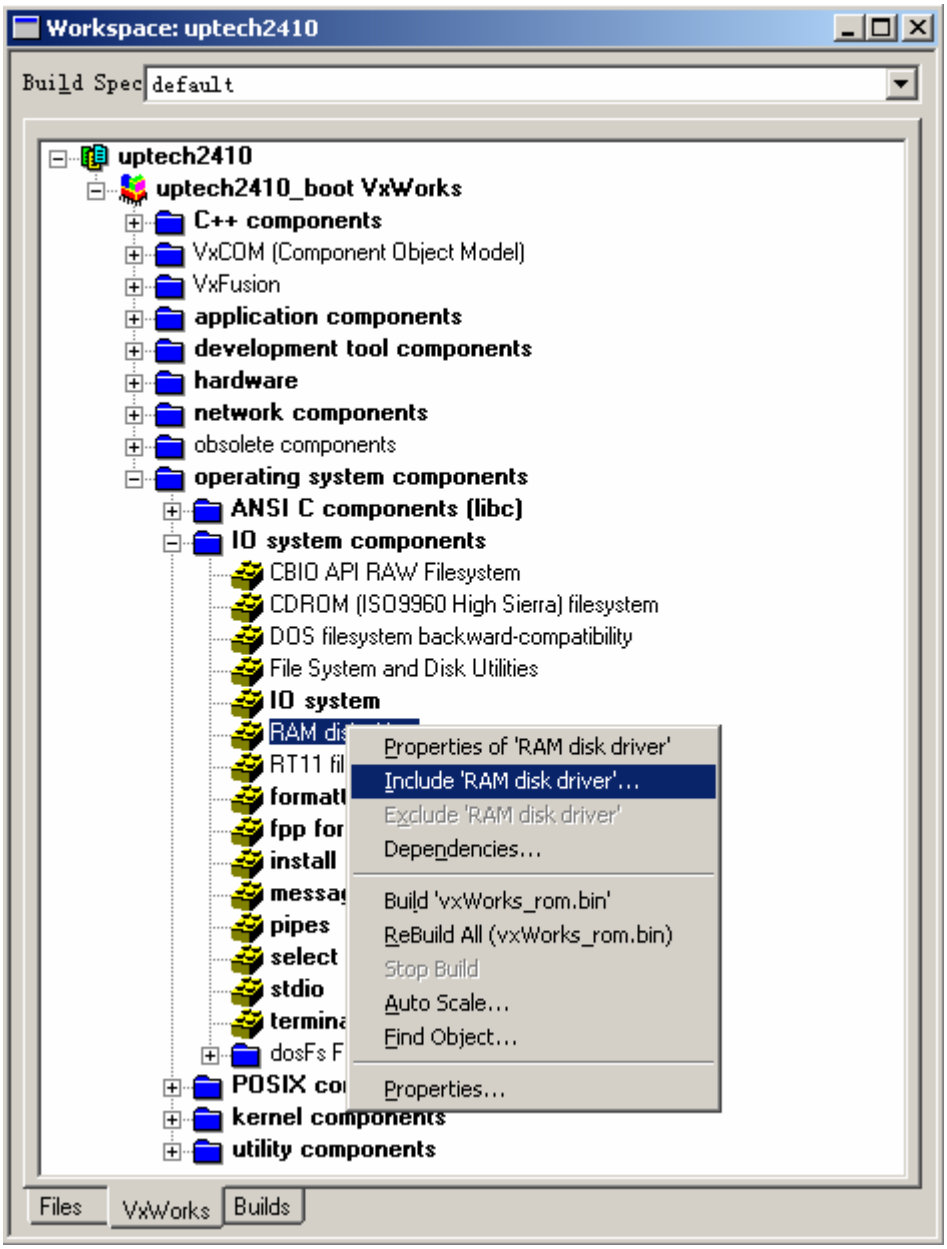
下面的例子创建了一个大小为”512×400”的 RAM 块设备，并加载 dosFs 文件系统，具体程序在 RamDrv 文件夹下的 ramDisk.c 文件。

在 target shell 下运行 sp ramDiskInit 执行 ramDiskInit 函数后，调用 devs，可以显示系统中的设备列表及对应的驱动程序号，可以看到”/ram0/”块设备已经建立成功，接着可以调用 cd 等命令对”/ram0/”进行操作。

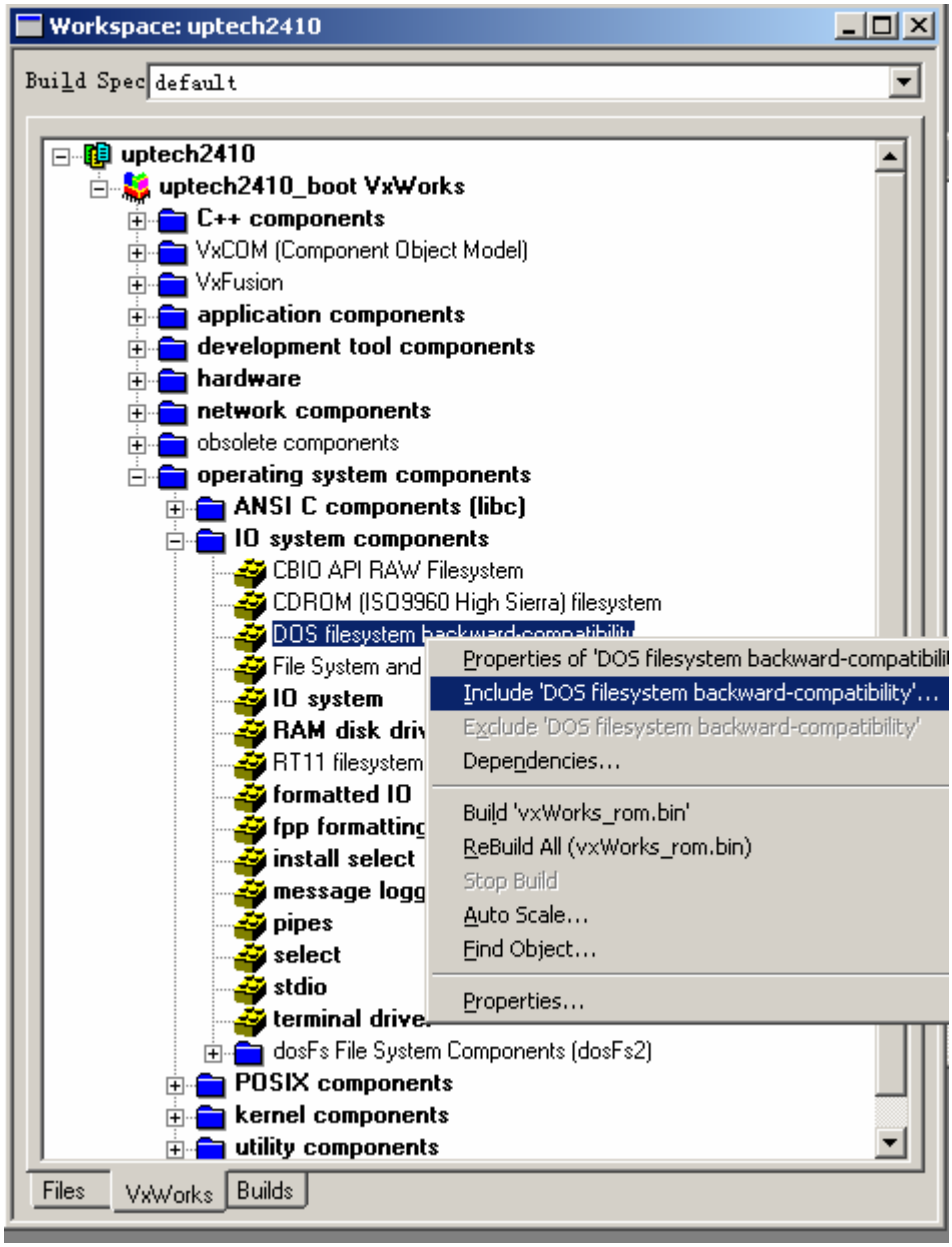
```
STATUS ramDiskInit(void)
{
    BLK_DEV                 *pBlkDev;
```

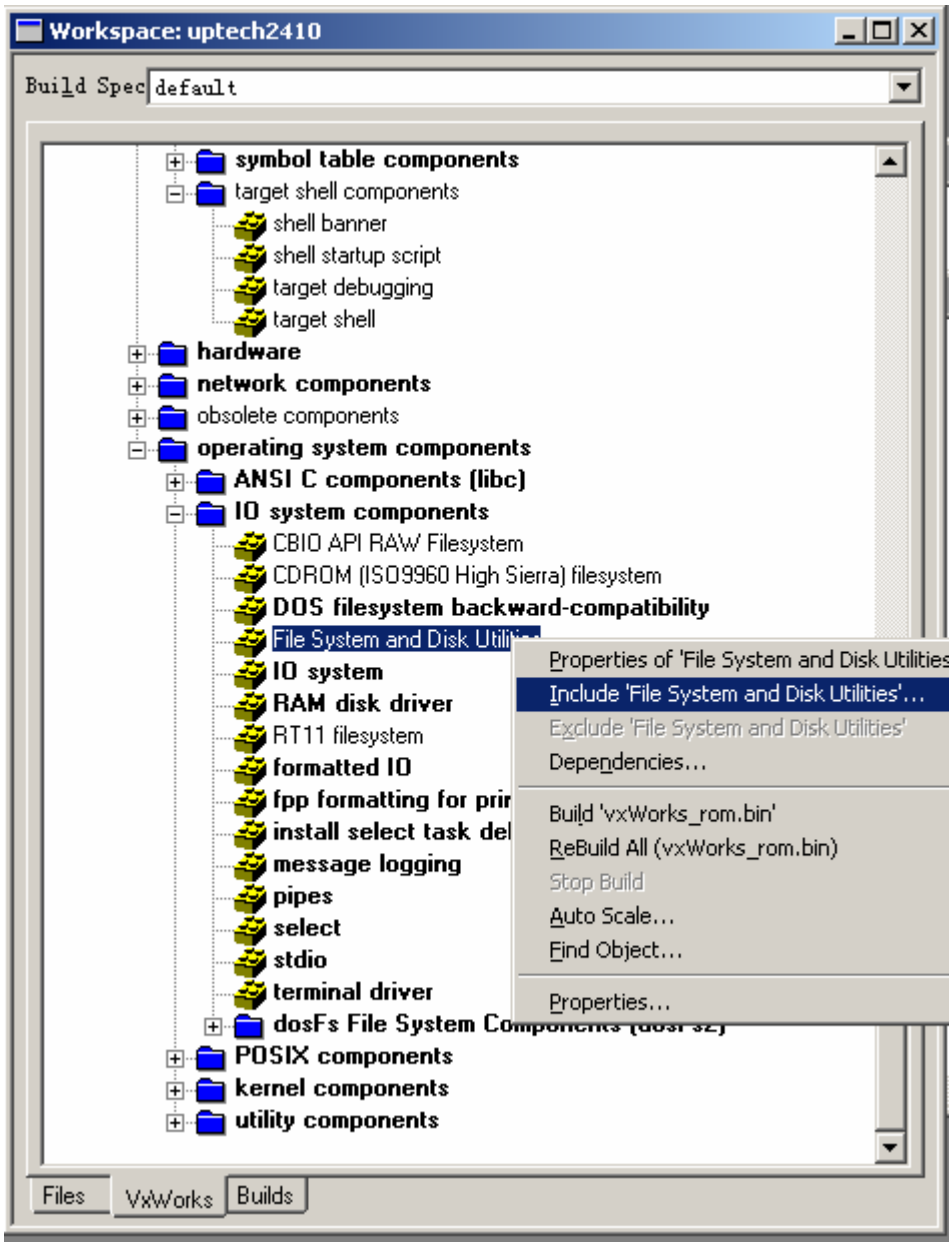
```
    DOS_VOL_DESC          *pVolDesc;
     char *ramDiskDevName = "/ram0/";

    pBlkDev = ramDevCreate (0, 512, 400, 400, 0); /*创建一个大小为 512*400
的 RAM 块设备*/
     if(pBlkDev == NULL)
        perror("ramDevCreate");

    pVolDesc = dosFsMkfs (ramDiskDevName, pBlkDev);/*加载 dosFs 文件系统*/
    if(pVolDesc == NULL)
        perror("dosFsMkfs");

return OK;

} /* end of ramDiskInit() */
```

第一步：在 Bootable 工程中添加 RAM DISK 组件。



第二步：包含 DOS 文件系统组件和 File System and Utilities。

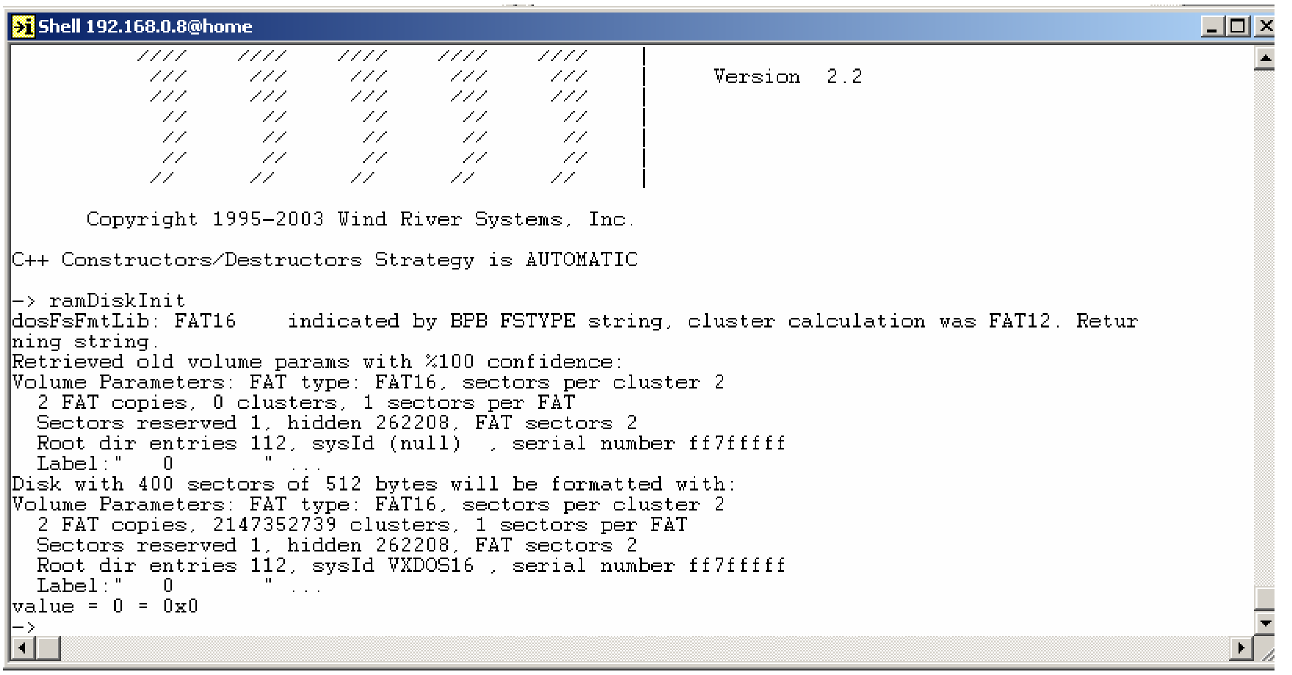

Workspace: uptech2410
Build Spec
default
uptech2410
uptech2410_boot VxWorks
C++ components
VxCOM (Component Object Model)
VxFusion
application components
development tool components
hardware
network components
obsolete components
operating system components
ANSI C components (libc)
IO system components
CBIO API RAW Filesystem
CDROM (ISO9960 High Sierra) filesystem
DOS filesystem backward-compatibility
File System and
IO system
RAM disk driv
RT11 filesystem
formatted IO
fpp formatting
install select
message logg
pipes
select
stdio
terminal drive
dosFs File System Components (dosFs2)
POSIX components
kernel components
utility components
Files
VxWorks
Builds
Properties of 'DOS filesystem backward-compatibili
Include 'DOS filesystem backward-compatibility'...
Exclude 'DOS filesystem backward-compatibility'
Dependencies...
Build 'vxWorks_rom.bin'
ReBuild All (vxWorks_rom.bin)
Stop Build
Auto Scale...
Find Object...
Properties...



第三步：编译 Bootable 工程，下载或烧入 VxWorks Image。

第四步：建立 Downloadable 工程，添加\src\RamDrv\ramDisk.c 文件，编译并下载。

第五步：在 Target Shell 或者 WinSh 运行 ramDiskInit。



```
Shell 192.168.0.8@home
                                                Version  2.2

      Copyright 1995-2003 Wind River Systems, Inc.

C++ Constructors/Destructors Strategy is AUTOMATIC

-> ramDiskInit
dosFsFmtLib: FAT16    indicated by BPB FSTYPE string, cluster calculation was FAT12. Retur
ning string.
Retrieved old volume params with %100 confidence:
Volume Parameters: FAT type: FAT16, sectors per cluster 2
  2 FAT copies, 0 clusters, 1 sectors per FAT
  Sectors reserved 1, hidden 262208, FAT sectors 2
  Root dir entries 112, sysId (null)  , serial number ff7fffff
  Label:"    0        " ...
Disk with 400 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT16, sectors per cluster 2
  2 FAT copies, 2147352739 clusters, 1 sectors per FAT
  Sectors reserved 1, hidden 262208, FAT sectors 2
  Root dir entries 112, sysId VXDOS16 , serial number ff7fffff
  Label:"    0        " ...
value = 0 = 0x0
->
```

显示文件系统已经建立成功。

5.3 网络通信

5.3.1 概述

VxWorks 具有强大的网络功能。标准的 VxWorks 采用了与 4.4 BSD TCP/IP 兼容的实时网络协议栈，使得网络开发变得易于进行和方便移植。

VxWorks 网络栈包含了对以下协议和工具的支持：

- 串行线路接口协议（SLIP）和可压缩串行线路接口协议（CSLIP）
- 网际协议（IP）
- 传输控制协议（TCP）和用户数据报协议（UDP）
- 动态主机配置协议（DHCP）
- 自举协议（BOOTP）
- 域名系统（DNS）
- 地址解析协议（ARP）和代理地址解析协议（Proxy ARP）
- 路由信息协议（RIP）
- 套接字（传输控制协议、用户数据报协议、组播、路由选择和 ZBUF）
- 远程过程调用（RPC）
- 远程命令解释程序（RSH）、远程登录（rlogin、telnet）
- 文件传输协议（FTP）和小文件传输协议（TFTP）
- 网络文件系统（NFS）

本章先介绍了 VxWorks 下的套接字工具，以及相应的系统函数。然后给出了如何使用标

准的 BSD 流套接字和数据报套接字实现接口实现 TCP、UDP 服务器/客户端的实验，以及如何利用 VxWorks 的网络组件实现网络协议和网络应用程序的实现，如 Ping，FTP 等。

5.3.2 VxWorks 套接字

套接字是与网络节点的 UDP 或 TCP 端口捆绑在一起的通信接口。在 VxWorks 中，应用程序使用套接字接口可以访问网际协议报的功能。VxWorks 套接字是与 UNIX BSD 4.4 套接字相兼容的。然而，VxWorks 并不支持套接字的信号功能。

比较常用的套接字有流套接字和数据报套接字。流套接字使用 TCP 协议捆绑某一端口。在两个 TCP 套接字连接后，他们之间将建立一条虚电路用以实现可靠的套接字之间的通信。这种通信的方式是对话式的。数据报套接字使用 UDP 协议捆绑某一端口。与 TCP 相比，UDP 提供了一个相对简单但适应性很强的通信方式。在 UDP 通信中，数据经由数据报进行传送，它们是分离、无连接的且是单独寻址的。数据报套接字采用的是邮信方式，每个数据报都含有目的地抵制和发送者地址。与 TCP 相比，UDP 是不可靠的。

5.3.3 Socket 函数

5.3.3.1 socket

socket	
目标	建立一个 socket
头文件	#include ” sockLib.h”
函数原型	int sockid = socket(int domain, int type, int protocol)
参数	domain 地址域 AF_INET 表示因特网地址域 type socket 的类型 protocol 指定协议 通常为 0
返回指	ERROR 错误 sockid 成功，返回 socket 的文件描述符

套接字的类型 type 如下：

SOCK_STREAM	流式套接字	提供顺序的、可靠的、双向的、基于连接的字节流，使用 TCP 协议
SOCK_DGRAM	数据报式套接字	提供无连接的、不可靠的数据报，使用 UDP 协议
SOCK_RAW	原始套接字	允许对较低层协议直接访问

例如：

```
int sFd = socket(AF_INET,SOCK_STREAM,0) ; /*生成流式套接字，用于 TCP*/
int sFd = socket(AF_INET,SOCK_ DGRAM,0) ; /*生成数据报式套接字，用于 UDP*/
```

5.3.3.2 bind

bind	
目标	绑定一个地址到 socket
头文件	#include “sockLib.h”
函数原型	STATUS result = bind (int s, struct sockaddr* name, int namelen)
参数	s socket 的文件描述符 name 指向包含地址结构的指针 namelen 地址长度
返回值	ERROR 错误 OK 成功

bind 调用把一个地址分配给 socket，sockaddr 结构体如下：

```
struct sockaddr {
    u_char  sa_len;
    u_char  sa_family;
    char    sa_data[14];
};
```

在 AF_INET 地址域中，使用主机和端口来标志，name 的格式使用 sockaddr 结构体。

sockaddr_in 结构体中将 sockaddr 结构体中的成员 sa_data[14]细分为 sin_port，sin_addr，sin_zero[8]，但两个结构体的大小是相同的。

```
struct sockaddr_in{
    u_char sin_len;
    u_char sin_family;              /*设为 AF_INET*/
    u_short sin_port;               /*端口号*/
    struct in_addr sin_addr;        /*主机地址，INADDR_ANY 表示可以为主机任一有
    效地址*/
    char sin_zero[8];               /*保留字段，需设为 0*/
    }
struct in_addr{
    u_long s_addr;
}
```

注意：sin_port 和 sin_addr 均要以网络字节顺序表示。

在 socket 中常用的转换函数如下：

■ bzero

```
void bzero((char*)buffer, int nbytes)
```

此函数将以 buffer 为始地址的缓存中开始的 nbytes 个字节赋值为 0

■ htonl

```
unsighed long htonl(unsigned long hostlong)
```

此函数将一个 32 位的无符号整数从主机字节顺序转换成网络字节顺序。

函数返回网络字节顺序的 32 位无符号整数。

■ htons

```
unsighed short htons(unsigned short hostshort)
```

此函数将一个 16 位的无符号整数从主机字节顺序转换成网络字节顺序。
函数返回网络字节顺序的 16 位无符号整数。

■ inet_aton 和 inet_addr

```
STATUS inet_aton
(
char* pString ,                    /*字符串形式的网络地址，如“192.168.2.102”*/
struct in_addr* inetAddress        /*转换的值所存放的结构体指针*/
)
u_long inet_addr
(
Char* inetString                   /*字符串形式的网络地址，如“192.168.2.102”*/
)
```

这两个函数均是将所指的 C 字符串转换为 32 位的网络字节顺序的二进制值。
inet_aton 将转换的值存在 inetAddress 所指的结构中，成功返回 OK，失败返回 ERROR
inet_addr 是直接返回转换的结果，失败返回 ERROR。

■ inet_ntoa

```
char * inet_ntoa(struct in_addr inetAddress)
```

此函数将 32 位的网络字节顺序的二进制值转换为相应的点分的十进制字符串，便于显示。

Bind 函数的调用例如：

```
    #define SERVER_PORT_NUM 5001                    /*bind()绑定的服务器端口号*/
    int sockAddrSize;                                 /*套接字地址数据结构大小*/
    struct sockaddr_in serverAddr;                    /*服务器套接字地址*/
    /*设置本地地址*/
    sockAddrSize = sizeof(struct sockaddr_in);
    bzero((char*)&serverAddr , sockAddrSize);
    serverAddr.sin_len = (u_char) sockAddrSize;
    serverAddr.sin_family = AF_INET;                  /*地址域，AF_INET*/
    serverAddr.sin_port = htons(SERVER_PORT_NUM);/*端口号*/
    serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);/*地址，INADDR_ANY 表示
                                                  可以使用主机上的任一有效地址*/
    /*绑定套接字*/
    if(bind(sFd,(struct sockaddr *)&serverAddr, sockAddrSize) == ERROR)
        {
            perror("bind \n");/*这是打印错误的函数，可以将当前的错误打印在控制台
*/
            close(sFd);
            return(ERROR);

        }
```

5.3.3.3 listen

listen	
目标	监听 socket 上的连接
头文件	#include “sockLib.h”
函数原型	STATUS result = listen(int s, int backlog)
参数	s 接受请求的 socket backlog 允许接入连接的数目
返回值	OK 成功 ERROR 失败

listen 用于服务器端套接字的侦听连接。当 socket 创建一个流式套接口时，该套接口被假设为一个主动套接口，也就是说它是一个将调用 connect 发起连接请求的套接口；通过调用 listen，此套接口即被指定为被动模式，服务器端即可通过此套接口接受客户请求。所以它一般用在函数 socket 和 bind 之后，accept 之前。参见实验 2.4.2 流程图的服务器方。

第二个参数 backlog 指定了在此套接口上排队等待处理的连接请求的最大个数。

例如：

```
if(listen(sFd, 5) == ERROR)
    {
        perror("listen \n");
        close(sFd);
        return(ERROR);
    }
```

5.3.3.4 accept

accept	
目标	接收 socket 上的一个连接
头文件	#include “sockLib.h”
函数原型	int newfd = accept(int s, struct sockaddr* addr, int* addrlen)
参数	s 接收该服务器套接口上的连接 addr 存储客户端地址结构的指针 addrlen 结构的长度
返回值	ERROR 错误 newfd 已连接套接口的文件描述符

accept 阻塞当前进程，一直到指定 socket 上的接入连接被建立起来，然后 accept 将返回新的套接口文件描述符 newfd,并用该文件描述符来进行读写操作，而原先的监听套接口 s 仍然保持打开状态来继续监听客户端的连接，直到该服务器关闭。

例如：

```
FOREVER /* FOREVER 定义为 for(;;)，服务器要一直运行，检查是否有客户端试图连接
*/
```

```
    {
        if((newFd = accept(sFd, (struct sockaddr*)&clientAddr, &sockAddrSize))
== ERROR)
            {
                perror("accept \n");
                close(sFd);
                return(ERROR);
            }
        else
            {
                /*建立连接，处理客户端请求*/
            }
    }
```

5.3.3.5 connect

connect	
目标	连接到 socket
头文件	#include “sockLib.h”
函数原型	STATUS result = connect(int s, struct sockaddr* name, int namelen)
参数	s 用于建立连接的 socket name 用于连接的 socket 地址 namelen socket 地址的长度
返回值	ERROR 错误 newfd 已连接套接口的文件描述符

connect 调用试图把由 s 所标识的 socket 和由 name 所指向的 socket 地址相连接。如果连接成功的话，connect 返回 OK，而此时，s 是一个合法的文件描述符，可以用来进行读写操作。写入该文件描述符的数据被发送到连接的另一端的 socket，而从另一端写入的数据将从该文件描述符读取。

例如：

```
/* serverAddr 为服务器地址结构体*/
if(connect(sFd, (struct sockaddr*)&serverAddr, sockAddrSize) == ERROR)
    {
        perror("connect \n");
        close(sFd);
        return(ERROR);
    }
```

5.3.3.6 send 和 recv

send 和 recv 一般用来对已连接的流式套接字进行操作，进行数据的读写与发送。

send	
目标	向连接的流套接字上发送数据
头文件	#include “sockLib.h”
函数原型	int nchars = recv(int s,const char* buf, int bufLen, int flags)
参数	s 发送数据的套接字 buf 发送缓存 bufLen 缓存长度 flags 发送属性
返回值	ERROR 错误 nchars 实际发送的字符数

例如：

```
/*TCP 实验中服务器的写操作*/
if((send(sFd, replyMsg, sizeof(replyMsg), 0)) == ERROR)
    perror("send \n");
```

recv	
目标	从连接的流套接字上读取数据
头文件	#include “sockLib.h”
函数原型	int nchars = recv(int s, char* buf, int bufLen, int flags)
参数	s 读取数据的套接字 buf 接收缓存 bufLen 缓存长度 flags 接收属性
返回值	ERROR 错误 nchars 实际接收的字符数

例如：

```
/*TCP 实验中服务器的读操作*/
    nRead = recv(sFd, (char*)&clientRequest, sizeof(clientRequest),0);
    if(nRead < 0)
        {
        /*发生错误*/
        }
    else if(nRead = = 0)
        {
        /*连接被关闭*/
        }
    else
        {
        /*成功接收数据*/
        }
```

5.3.3.7 sendto 和 recvfrom

sendto 和 recvfrom 一般用来对数据报套接字进行操作，进行数据的读写与发送。

sendto	
目标	向数据报套接字上发送数据
头文件	#include “sockLib.h”
函数原型	int nchars = sendto(int s, char* buf, int bufLen, int flags, struct sockaddr* to, int toLen)
参数	s 读取数据的套接字 buf 接收缓存 bufLen 缓存长度 flags 接收属性 to 指向远端 socket 的地址指针 tolen 地址长度
返回值	ERROR 错误 nchars 实际接收的字符数

例如：

```
/*UDP 实验中客户端的写操作*/
struct sockaddr_in serverAddr;                /*服务器套接字地址*/
if(sendto(sFd,(caddr_t)&myRequest,sizeof(myRequest),0,
    (struct sockaddr*)&serverAddr,sockAddrSize) == ERROR)
    {
        perror("sendto");
        close(sFd);
        return(ERROR);
    }
```

recvfrom	
目标	从数据报套接字上读取数据
头文件	#include “sockLib.h”
函数原型	int nchars = recvfrom(int s, char* buf, int bufLen, int flags, struct sockaddr* from, int* pFromLen)
参数	s 读取数据的套接字 buf 接收缓存 bufLen 缓存长度 flags 接收属性 from 指向远端 socket 的地址指针 pFromLen 地址长度
返回值	ERROR 错误 nchars 实际接收的字符数

例如：

```
/*UDP 实验中服务器的读操作*/
struct sockaddr_in clientAddr;              /*客户端套接字地址*/
if(recvfrom(sFd,(char*)&clientRequest,sizeof(clientRequest),0,
  (struct sockaddr *)&clientAddr, &sockAddrSize) == ERROR)
  {
      perror("recvfrom \n");
      close(sFd);
      return(ERROR);
  }
```

5.3.4 实验

5.3.4.1 Ping

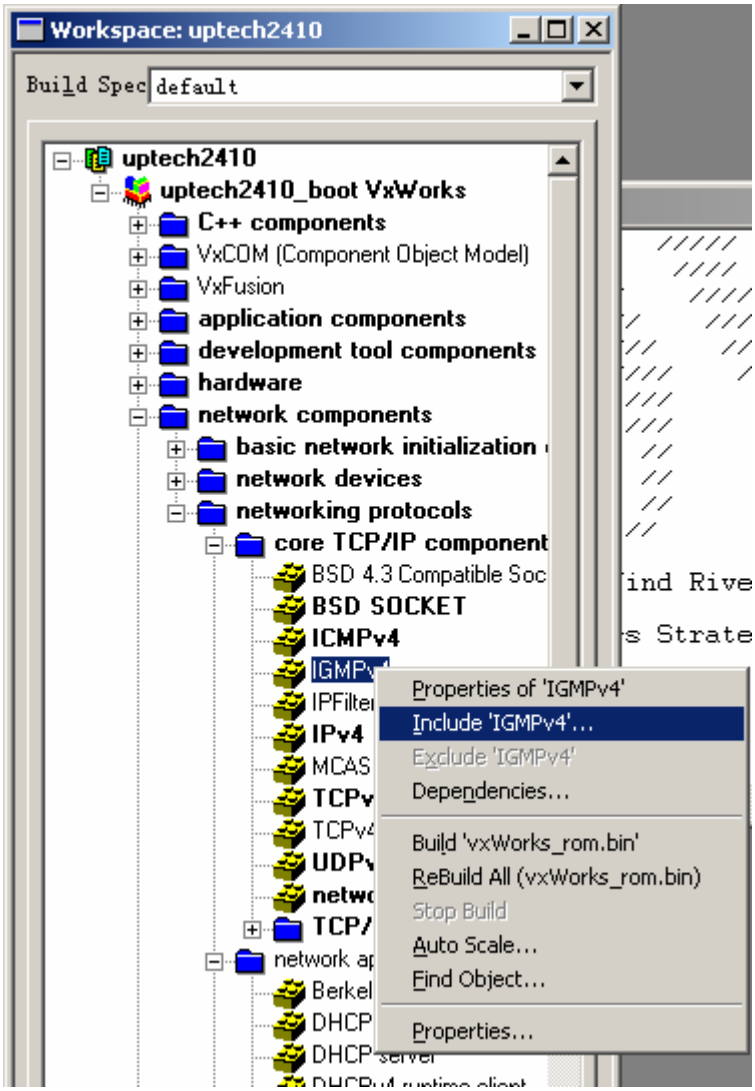
ping 程序的目的是为了测试网络中的主机是否可达。该程序发送一份 ICMP 回显请求报文给主机，并等待返回 ICMP 回显应答。我们称发送回显请求的 ping 程序为客户，而称被 ping 的主机为服务器。在 VxWorks 的网络组件中，默认包含了 ICMP 协议，即可支持 ping 服务器，而 ping 客户端的支持需要手动包含所需要的组件 INCLUDE_PING。

- Ping Server

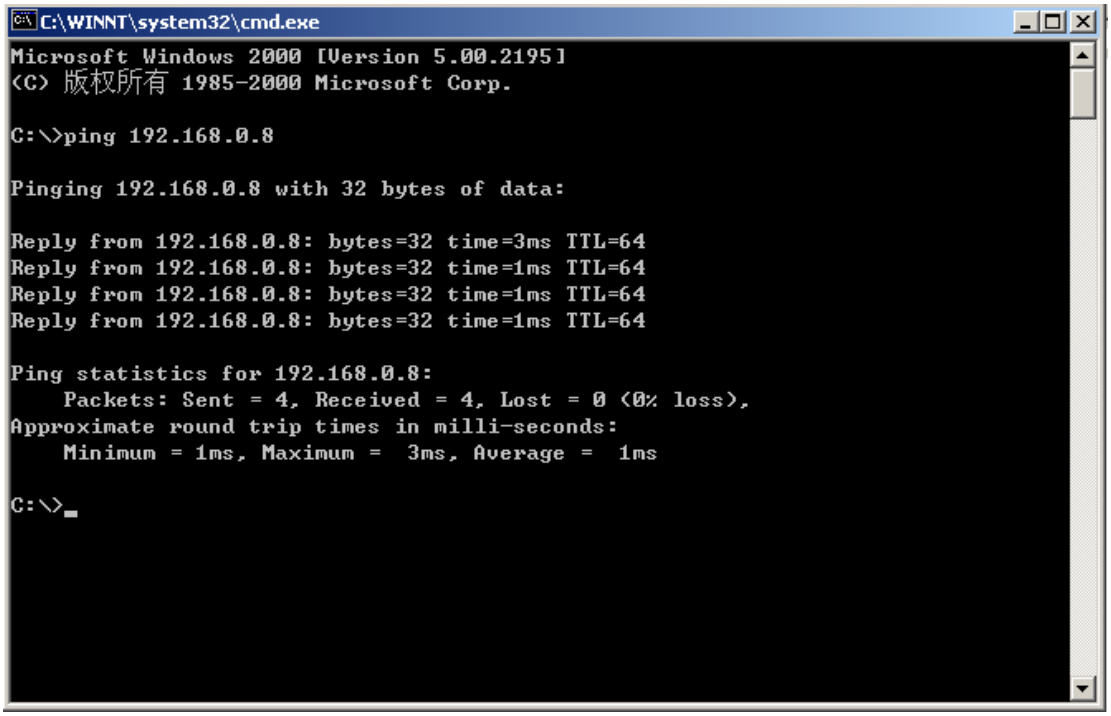
在 VxWorks 的网络组件中，默认包含了支持 ping 服务器的 ICMP 协议，可以在 Tornado 环境下察看

Network components->networking protocols->core TCP/IP components->ICMPv4 ping server 的实现需将此组件包含。

注意，在添加组件以后要重新编译 Bootable 工程，并下载 VxWorks Image 文件或重新烧入 Flash Image 文件。

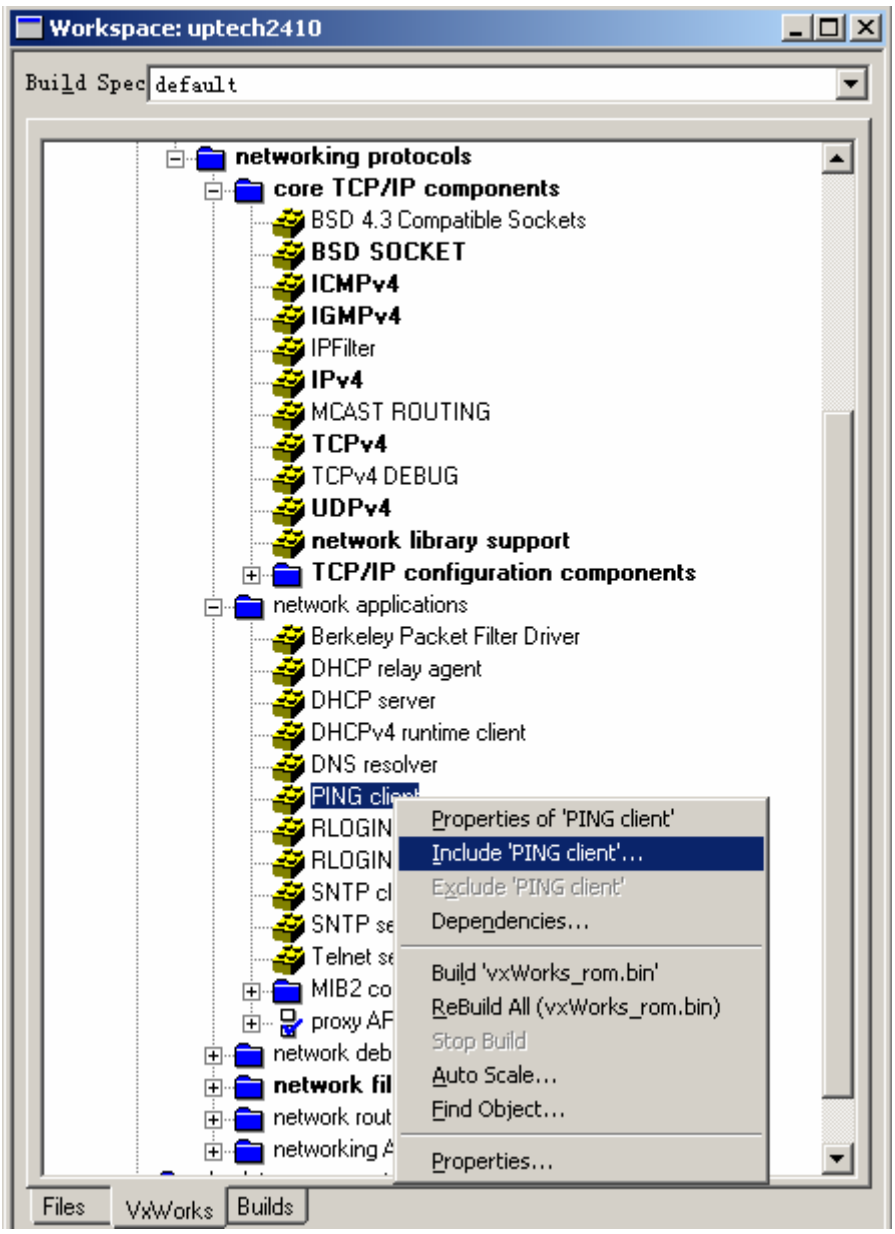


在我们的 PC 上去 ping 目标板，若网络连接正常，则显示：



■ Ping Client

ping client 的实现需包含相应的组件 INCLUDE_PING，在 Tornado 环境下 Network components->networking protocols->network applications->PING client



VxWorks 提供了 ping()公用程序可以在系统中使用。

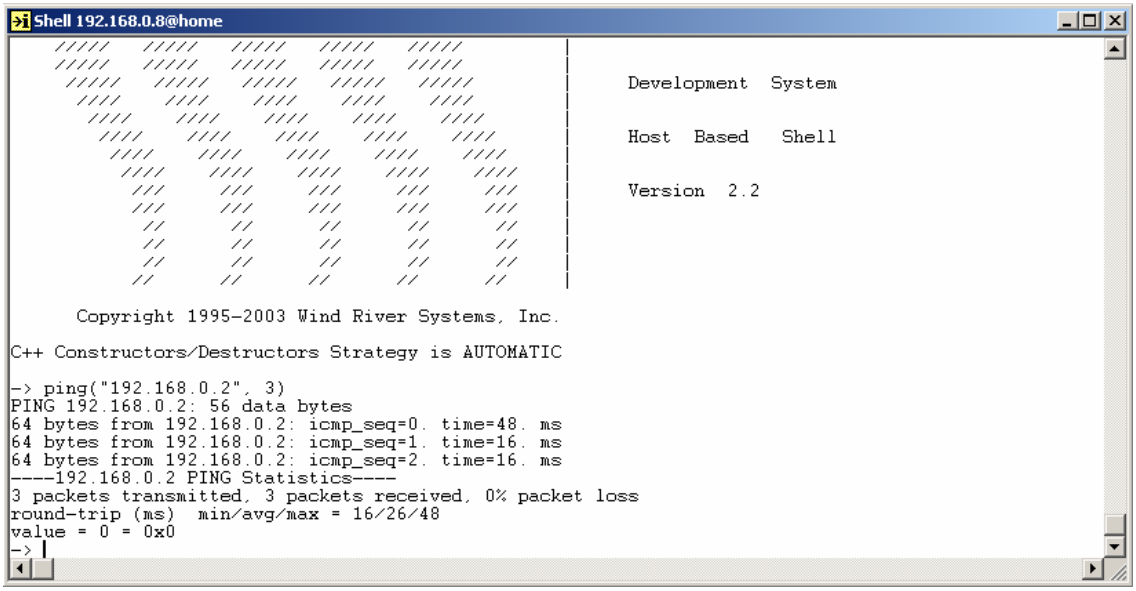
Ping	
目标	测试网络中的主机是否可达
头文件	#include “pingLib.h”
函数原型	STATUS nResult = ping(char* host, int numPackets, ulong_t options)
参数	host 被 ping 的主机 buf 返回数据报的多少（也是发送数据报的多少） options 属性，一般可以省略
返回值	ERROR 错误，主机不可到达 OK 成功，主机可到达

Ping 程序有如下属性 options

属性值	描述

PING_OPT_SILENT	抑制打印输出
PING_OPT_DONTROUTE	抑制 ping（）到达没有经过网关而直接相连的主机

添加组件以后重新编译 Bootable 工程，更新 VxWorks Image，重新启动后，ping PC 机。如下图：

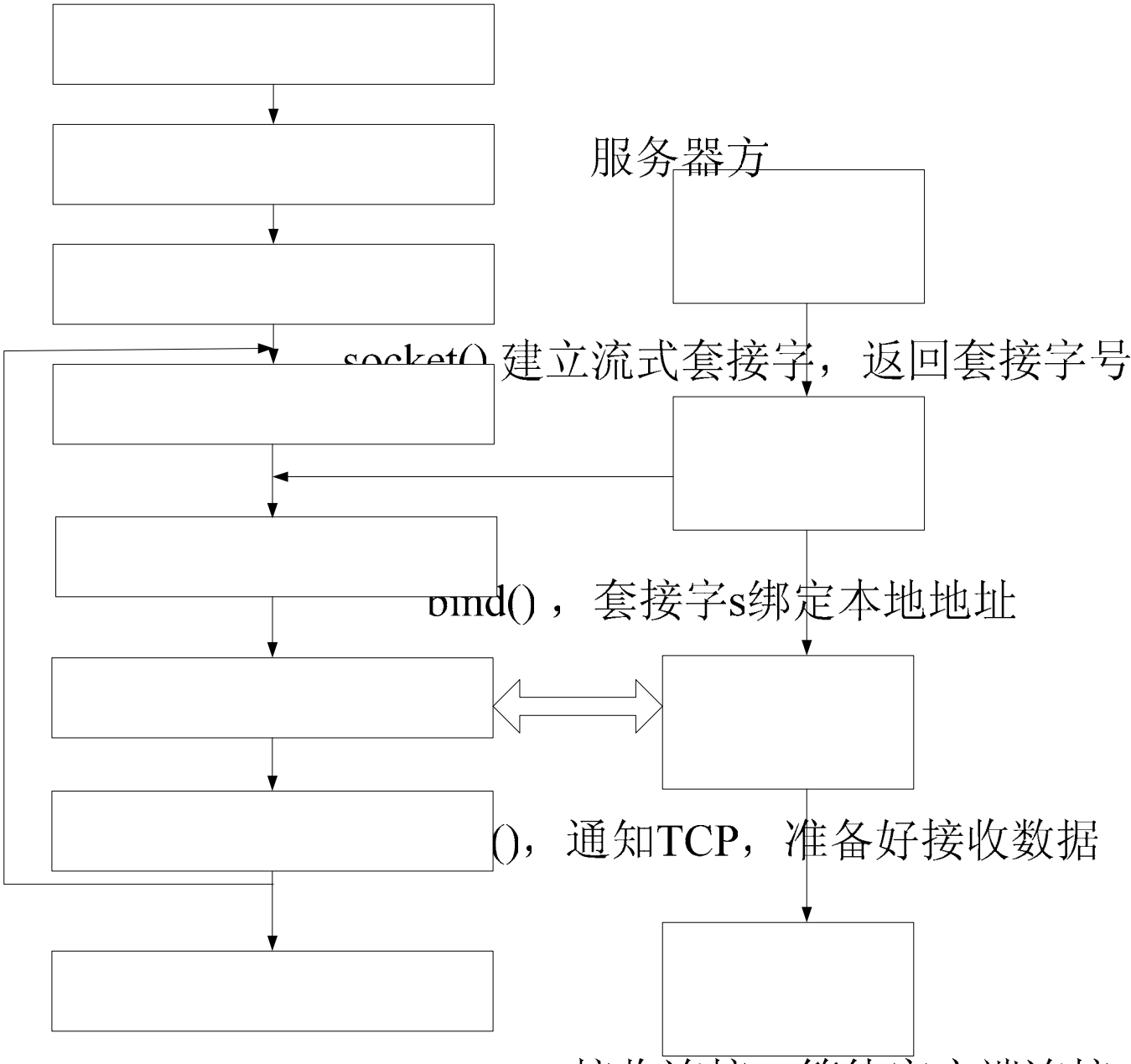


5.3.4.2 流套接字（基于 TCP）

在 TCP/IP 网络应用中，通信的两个任务间主要模式是客户机/服务器模式，即客户先向服务器提出服务请求，服务器接收到请求后，提供相应的服务。

面向连接的协议（如 TCP）服务器首先调用 socket 函数建立流式套接字，然后用 bind 将此套接字和本地地址绑定；调用 listen 准备接收客户端的连接；然后调用 accept 接收连接，当收到客户端的连接请求后，则连接建立，accept 返回新的套接字，就可以在这新套接字上读写数据；原来的套接字则可以继续通过 accept 调用等待另一个连接。

客户端也首先调用 socket 建立流式套接字，然后调用 connect 向服务器端发起连接请求，连接建立后就可以在此套接字上进行数据的读写了，具体流程如下：



TCP 文件夹下的三个文件演示了如何构建基于 TCP 的客户机/服务器模式。

tcpExample.h ： 头文件，定义了宏和结构体

tcpClient.c ： 客户端代码

tcpServer.c ： 服务器端代码

第一步：建立 Downloadable 工程，并且加入 \src\TCP\tcpServer.c 和 \src\TCP\tcpClient.c;并且把 tcpExample.h 放入 tcpServer.c 和 tcpClient.c 的目录中。

第二步：编译并且下载

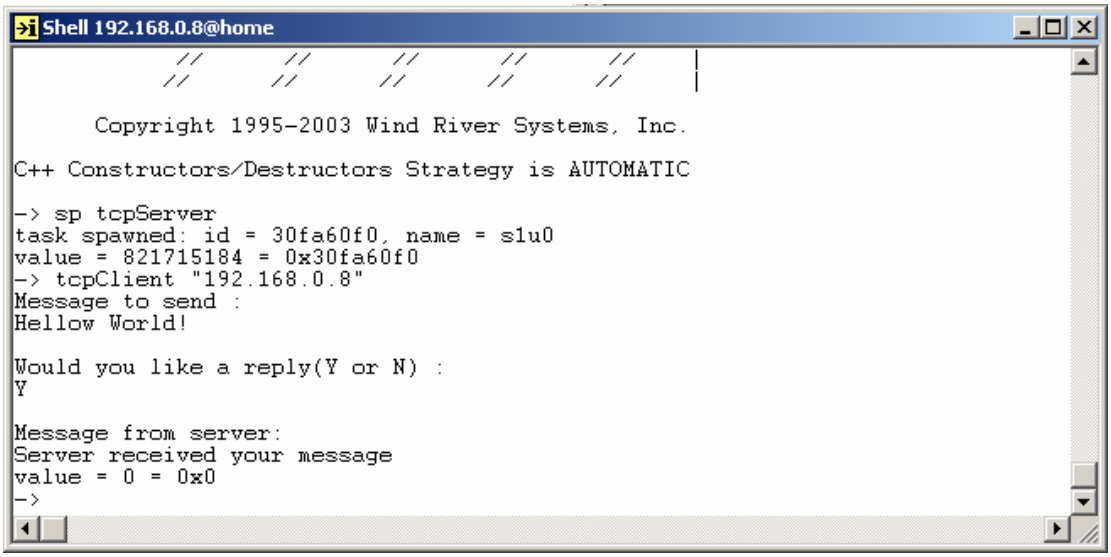
第三步：在 TargetShell 或者 WindSh 运行 tcpServer，把 Tcp Server 先运行起来。由于我们是在一个目标机上运行，也就是 Client 和 Server 都是在一台目标机上。所以接下来运行 Tcp Client，即在 TargetShell 或者 WindSh 运行 tcpClient <目标机 IP>

首先，在服务器端输入 sp tcpServer，发起 tcpServer 任务，等待客户端连接；

然后，在客户端调用 tcpClient（），参数为服务器地址。例如：tcpClient “192.168.2.105”，根据提示输入请求信息，然后选择服务器是否返回答复信息，这里输入 y 表示服务器需要返回，然后和服务器建立连接，将信息发送给服务器端，并得到服务器的应答。

Sp 是 taskSpawn 的意思，在一个目标机上运行一个任务，可以使用 sp <task name>，表示启动一个任务，并且不占用 Shell，也就是还可以在 Shell 运行其他的任务和命令。并且在 WinSh 下面，函数的调用可以不使用 ***函数名（函数参数）*** 的格式，可以直接使用***函数名 函数参数*** 。并且在 Shell 里面是区分大小写的。

如下图：



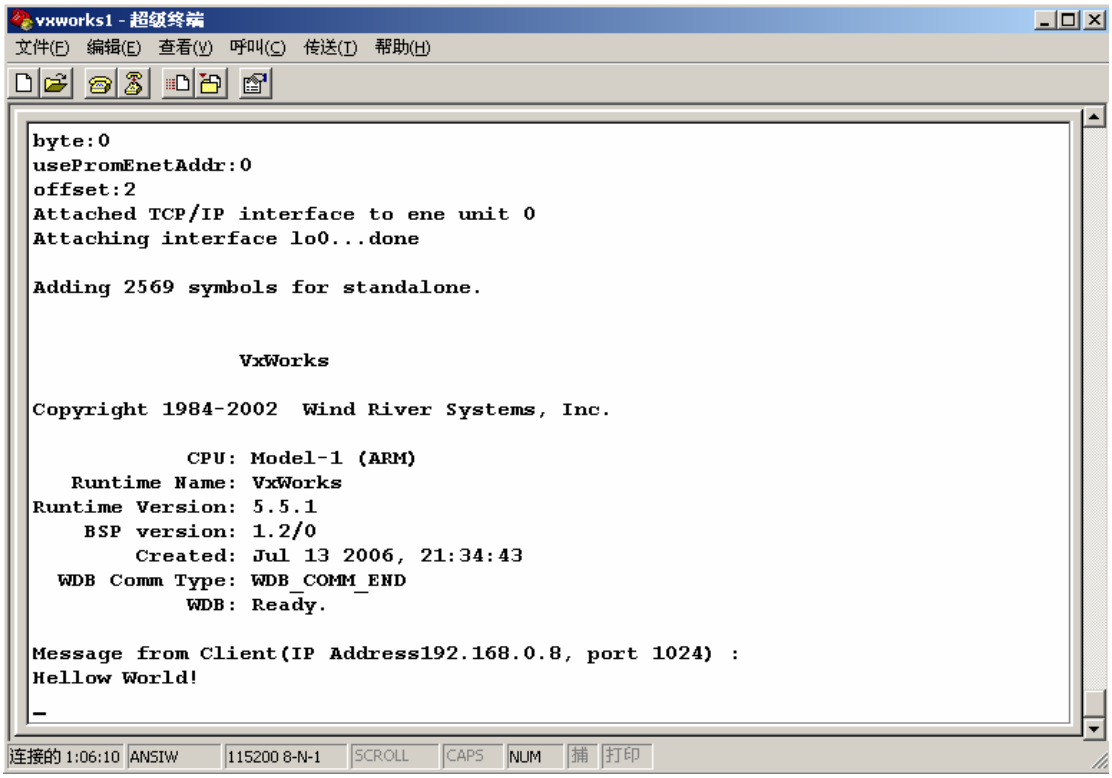
运行了 Tcp Client，然后输入要传输的字符，当问到是否要回答，属于 Y，会得到:

Message from server:

Server received your message

这是 Tcp Server 回传的信息，告诉 Client，Server 已经接收到了 Client 的数据。

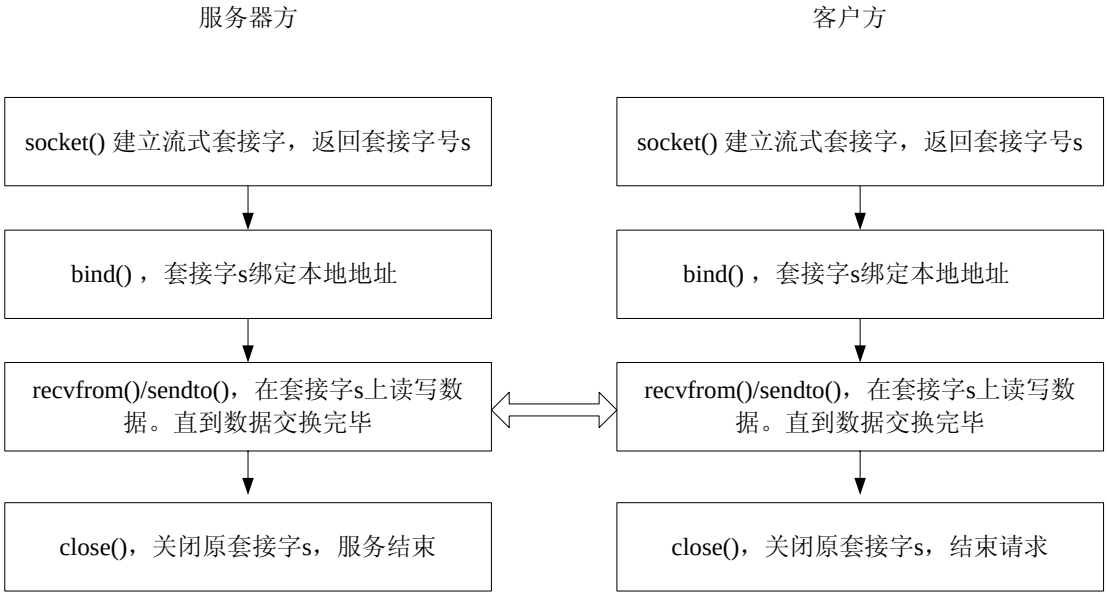
同时，在串口上的显示表示 Tcp Server 收到的 Client 的信息。



这个例子中，我们是把 Tcp Server 和 Client 放在同一个目标机上运行，当然把 Tcp Server 和 Client 分别运行在两个目标机上也是可以的。

5.3.4.3 数据报套接字（基于 UDP）

无连接服务器首先调用 socket 建立套接字，然后用 bind 绑定本地地址。与面向连接的不同的是它不需要侦听和建立连接，此时通过调用 sendto 和 recvfrom 就可以读写数据了，客户端与此相同，具体流程如下：



UDP 文件夹下的三个文件演示了如何构建基于 TCP 的客户机/服务器模式。

udpExample.h ： 头文件，定义了宏和结构体

udpClient.c　 ： 客户端代码

udpServer.c ： 服务器端代码

客户端调用 udpClient（），得到用户的输入信息，然后发送给服务器端，还可选择服务器端是否显示。

这个实验的过程和 TCP 试验基本相同。

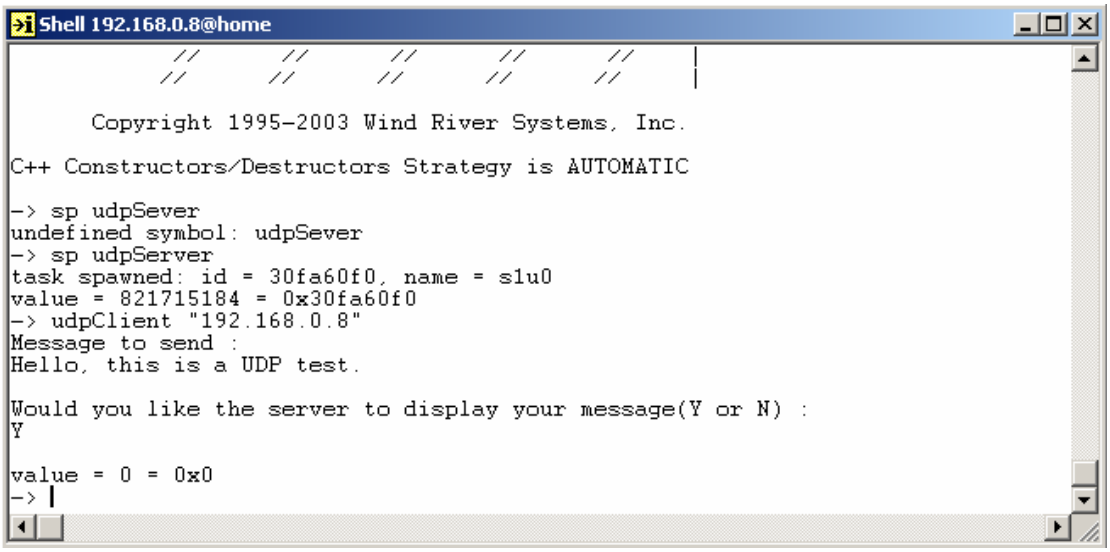
第一步：建立 Downloadable 工程，并且加入上面提到的 udpClient.c 和 udpServer.c 文件，把 udpExample.h 放入 udpClient.c 和 udpServer.c 文件的目录中。

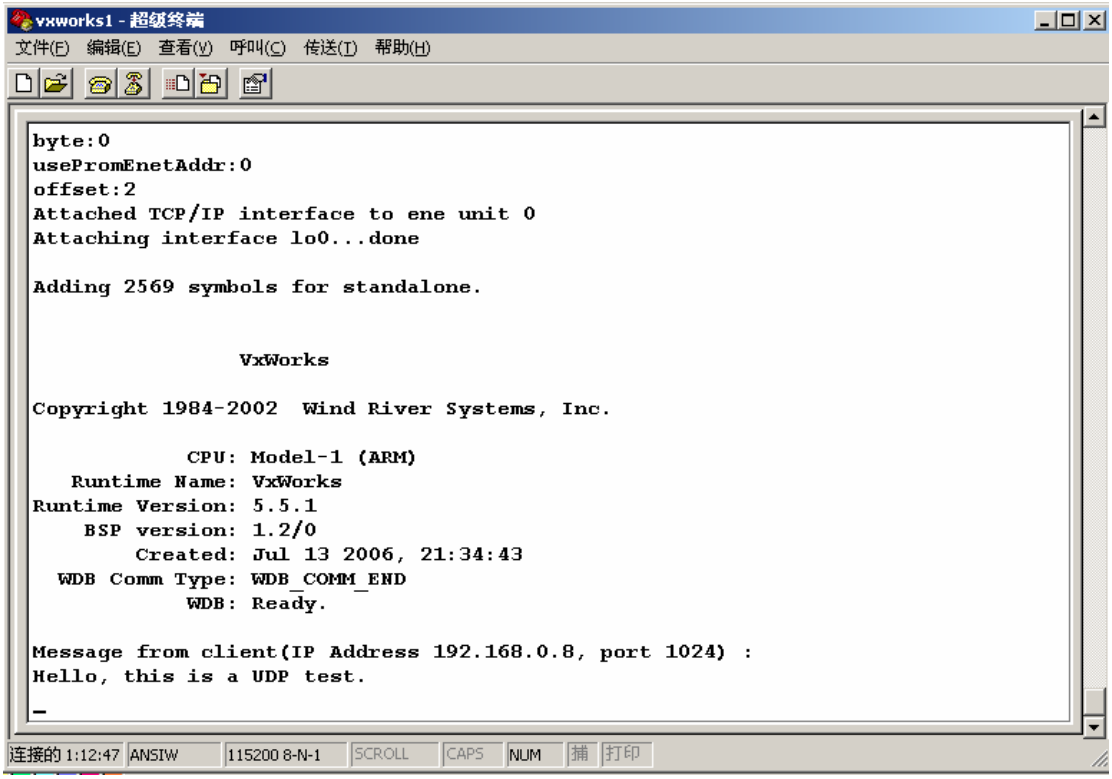
第二步：编译并且下载。

第三步：在 Target Shell 或者 WindSh 里面按照下图进行操作。先运行 udpServer，然后运行 udpClient，进行 UDP 通信。

首先，在服务器端输入 sp udpServer 发起 udpServer 任务，然后等待读取客户端信息，并根据客户的请求选择性显示客户信息。

然后，在客户端调用 udpClient（），参数为服务器地址，例如 udpClient（“192.168.2.105”），根据提示输入要发送的信息，并可选择服务器端是否显示，y 表示服务器端将接受的信息显示，n 则不显示。





5.3.4.4 FTP

FTP（File Transfer Protocol，文件传输协议），是用于文件传输的 Internet 标准。VxWorks 提供了对 FTP 协议的支持，所包含的 FTP 工具有：

- INCLUDE_FTP —— 包含 FTP 客户机，ftpLib
- INCLUDE_FTP_SERVER —— 包含 FTP 服务器，ftpdLib
- INCLUDE_FTPD_SECURITY —— 包含 FTP 服务器安全功能

■ FTP Server

包含组件 INCLUDE_FTP_SERVER，或在 Tornado 环境下选择

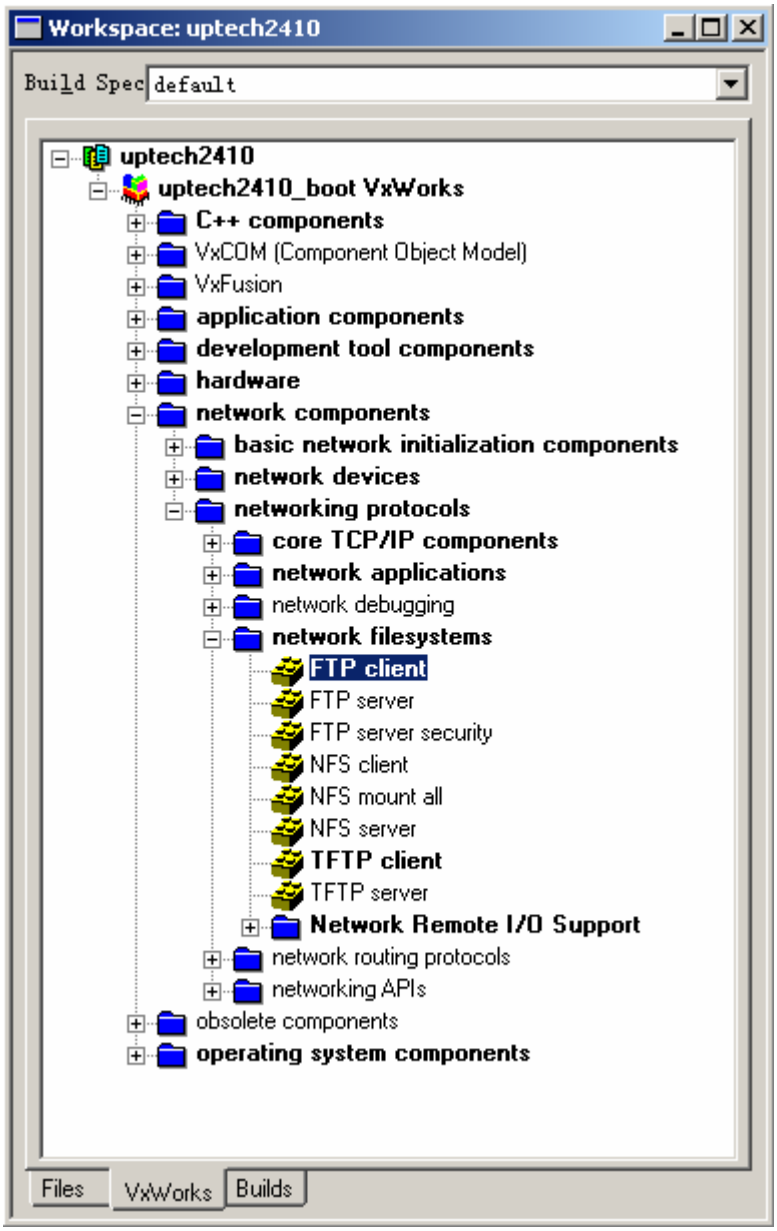
network components->networking protocols->network filesystems->FTP server

即可加载 FTP 服务器，使得系统启动时调用 ftpdInit 函数来启动 FTP 任务。

■ FTP Client

默认情况下，VxWorks 的网络组件包含 FTP Client 组件 INCLUDE_FTP，若没有包含，请在自己目标板 BSP 目录下的 config.h 文件中，加入：

```
#undef INCLUDE_FTP
#define INCLUDE_FTP
```



ftpXfer 时使用 FTP 协议进行 FTP 操作的主要函数。

ftpXfer	
目标	通过 FTP 协议传输文件
头文件	#include “ftpLib.h”
函数原型	STATUS nResult = ftpXfer (char * host, char * user, char * passwd, char * acct, char * cmd, char * dirname, char * filename,

	int * pCtrlSock, int * pDataSock)
参数	host FTP Server 的地址 user FTP 用户名 passwd FTP 密码 acct 说明 cmd 命令 dirname 目标目录 filename 待传输的文件名 pCtrlSock 控制套接字描述符 pDataSock 数据套接字描述符
返回	ERROR 若套接字无法被分配或无法连接 OK 传输成功

命令参数 cmd：

值	描述
“STOR %s”	向 FTP Server 上传文件
“RETR %s”	从 FTP Server 下载文件

ftpCommand	
目标	发送 FTP 命令并得到一个回应
头文件	#include “pingLib.h”
函数原型	int nReply = ftpCommand(int ctrlSock, char *fmt, int arg1…arg6)
参数	ctrlSock 控制套接字，**ftpXfer** 的参数 pCtrlSock 指向的套接字 fmt 命令字符 arg1…arg6 参数
返回值	ERROR 错误 nReply 返回信息

命令字符 fmt 有以下常用值：

值	描述
ABOR	放弃先前的 FTP 命令和数据传输
LIST filelist	列表显示文件或目录
PASS password	服务器上的口令
PORT n1,n2,n3,n4,n5,n6	客户端 IP 地址（n1,n2,n3,n4）和端口(n5×256+n6)
QUIT	从服务器注销
RETR filename	检索（取）一个文件
STOR filename	存取（放）一个文件
Type type	说明文件类型：A 表示 ASCII 码；I 表示图像
USER username	服务器上用户名

nReply 返回信息有以下值：

宏	值	描述
FTP_PRELIM	1	肯定预备应答。它仅仅是在发送另一个命令前期待另一个应答时启动
FTP_COMPLETE	2	肯定完成应答。一个新命令可以发送
FTP_CONTINUE	3	肯定中介应答。该命令已被接受，但另一个命令必须被发送
FTP_TRANSIENT	4	暂态否定完成应答。请求的动作没有发生、但差错状态是暂时的，所以命令可以过后在发
FTP_ERROR	5	永久性否定完成应答。命令不被接受，并且不再重试

下面的例子演示了如何将远程 FTP 服务器上的文件下载到本地文件系统/ram0/上的过程，具体程序在 FTP 文件夹下的 ftpCopy.c 中。需要注意的是这里用到了 5.3 ramDrv 的相关内容，必须将 RamDrv 文件夹下的 ramDisk.c 文件加入到工程里。具体的过程请参考 5.3 的内容。

```
/************************* ftpCopy.c ****************************
将远程 FTP 服务器上的文件下载到本地文件系统上
(需要在目标机上建立本地文件系统,
比如基于 RamDisk 的 dosFs 等)
*****************************************************************/

/*****************************************************************
 *  INCLUDE FILES
 *****************************************************************/
#include "vxWorks.h"
#include "ftpLib.h"
#include "stdio.h"
#include "ioLib.h"

#define SYS_BASE_FS          "/ram0/"
/*****************************************************************
 *  FUNCTION
 *****************************************************************/

/*****************************************************************
 *  ftpCopy()
 *
 *  参数
 *  serverAddr - FTP Server 的地址
 *  ftpUser - FTP 用户名
 *  ftpPwd - FTP 密码
 *  pSrcFileName - 存在本地文件系统的文件名
 *  pDesFileName - 要从服务器端下载的文件名
 *
 *  返回值
```

```
*  OK
*  ERROR
*
**********************************************************************/

STATUS ftpCopy ( char* serverAddr, char* ftpUser, char *ftpPwd, char * pSrcFileName,
char * pDestFileName )
{
    int     fd;                                                    /*数据传输套接
字*/
   int  errFd;                                                  /* 控制套接字*/

    int     localFd;                                               /*本地文件描述
符*/
   char     localFname[50];
    int maxSize = 0x10000;                          /*接收文件的最大值 64K*/
    char   *compBuf = NULL;                         /*接收缓存指针*/
    int    compSize = 0;                                    /*接收文件大小*/
    int    r = 0;
    STATUS rc = OK;
    memset( localFname , '\0' , 50 );
    printf("starting connect host,please waiting...\n\r");
    /*向 FTP Server 发送请求,建立连接,传输文件*/
    if (ftpXfer (serverAddr, ftpUser, ftpPwd, "", "RETR %s", "", pSrcFileName,
                        &errFd, &fd) == ERROR)
        {
            printf("when downing file , found error.\n\r");
           return ERROR;
        }
    if ((compBuf = malloc(maxSize)) == NULL)            /*分配 64K 内存,用以接收
文件*/
    {
        printErr("Not enough memory for image buffer\n");
        rc = ERROR;
        goto errorDo;
    }

    memset( compBuf , '\0' , maxSize );
    compSize = 0;
        while ((r = read(fd,    /* 循环读取数据套接字中的数据*/
                compBuf + compSize,
                maxSize - compSize)) > 0)
        compSize += r;
        if (r < 0)                      /*读出错*/
```

```
        {
        printErr("Read failed: errno = %d\n", errnoGet());
        rc = ERROR;
        goto errorDo;
        }
        if (compSize == maxSize)            /*传输文件大于 64K*/
        {
        printErr("Compressed image too large\n");
        rc = ERROR;
        goto errorDo;
        }
        /*把传入的数据写入/ram0/desFile*/
        printf("Saving image to %s ... ",localFname);
       sprintf( localFname , "%s%s" , SYS_BASE_FS, pDestFileName);

        if ((localFd = open(localFname, O_RDWR , 0)) != ERROR)      /*  如  果
/ram0/desFile 文件存在*/
        {
            close(localFd);
            printf("This file is already exist!\n\r");
            rc = ERROR;
        }
       else if ((localFd = open(localFname, O_RDWR  | O_CREAT, 0644)) != ERROR)
    /*创建文件/ram0/desFile*/
       {
                write(localFd,compBuf,compSize);
                close(localFd);
       }
       else
       {
                rc = ERROR;
       }
errorDo:
    if( compBuf )                                          /*释放内存*/
        free( compBuf );
    ftpCommand ( errFd, "QUIT",0,0,0,0,0,0);     /*退出 FTP 服务器*/

    /*关闭套接字*/
    if( fd )
        close( fd );
    if( errFd )
        close( errFd );

    if( rc != ERROR )
```

```
        printf("\n\rsuccess.\n\r");
    else
        printf("\n\rfailure.\n\r");

    return rc;
}
```

首先必须保证自己的主机上已启动了 FTP 服务器。FTP 服务器的建立方法见附件

第一步：建立包含 FTP Client 和 RamDisk 的 Bootable 工程，编译并且更新 VxWorks Image；

第二步：建立 Downloadable 工程，包换 RamDisk 文件和 FTP 文件，编译并且下载；

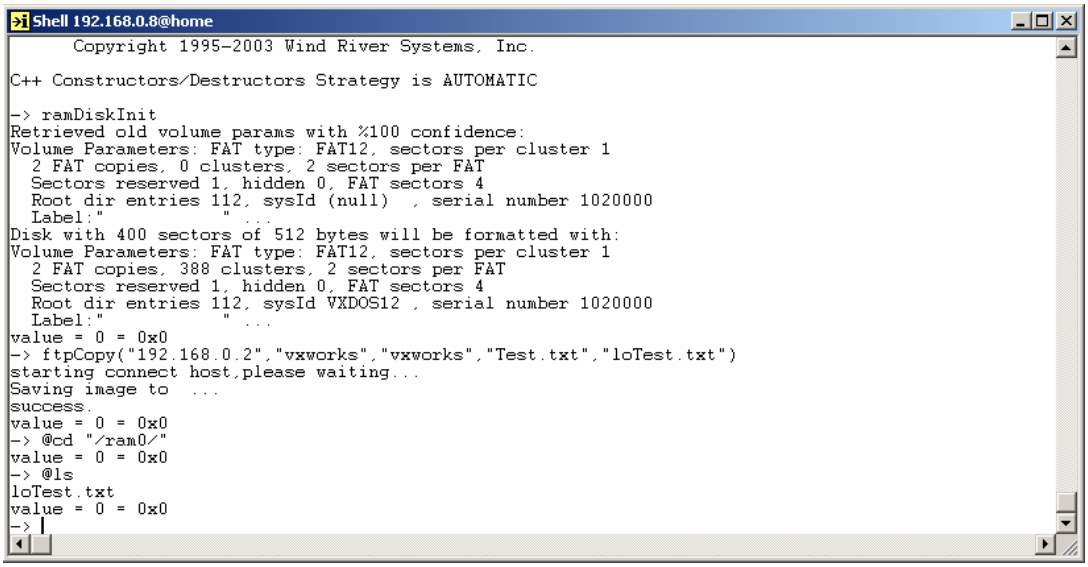
第三步：需先调用 ramDiskInit（）建立基于 RamDisk 块设备的 dosFs 文件系统（具体内容请参见 4.3 ramDrv），输入：`sp ramDiskInit` ，接着调用 ftpCopy(),从 FTP Server 上下载相应的文件存储到文件系统中。

例如：

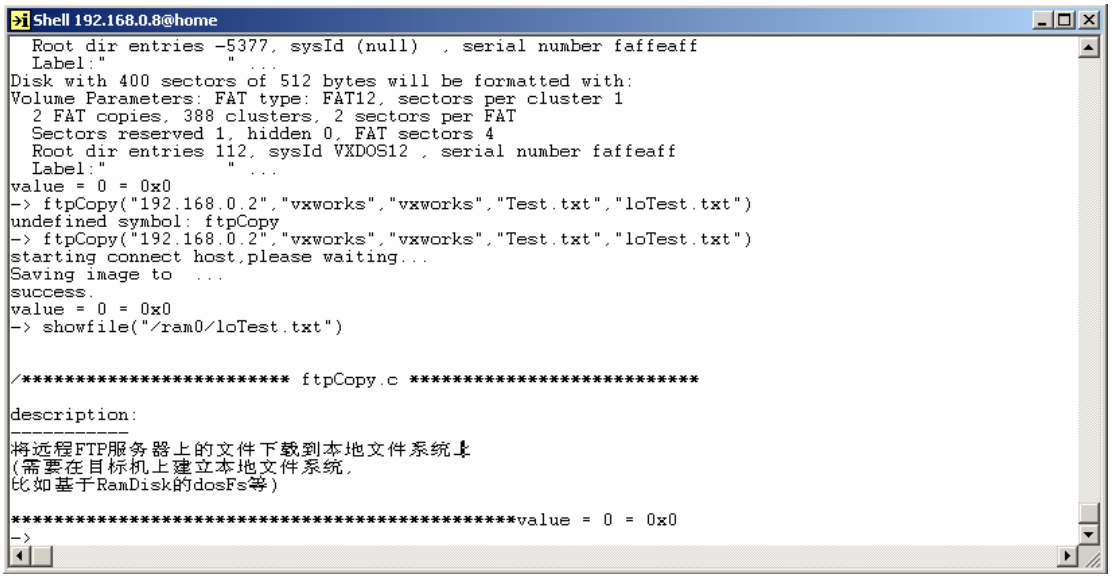
```
ftpCopy("192.168.0.2","vxworks","vxworks","Test.txt","loTest.txt");
```

其中，"192.168.2.12" FTP 服务器的地址，"vxworks" 用户名，"vxworks" 密码，"Test.txt"服务器上欲下载文件的文件名，"loTest.txt" 本地目的文件的文件名。

此时，利用 cd 进入/ram0/目录下，调用 ls 即可以看到保存文件 loTest.txt：



然后运行 showfile("/ram0/loTest.txt")，就可以显示 loTest.txt 文件的前 256Byte 的内容。



5.3.4.5 以太网包的截取与解析

以太网包的截取有很多种方法，我们这里介绍一种最直接，最底层的方法：就是在以太网驱动程序中直接把数据包截取并且打印出来。

在 BSP 的目录里面，ne2000end.c 是以太网的驱动文件。ne2000HandleRcvInt 函数就是以太网收包的函数。（BSP 的知识在这里也不做具体的介绍）。

```
LOCAL void ne2000HandleRcvInt ( NE2000END_DEVICE *pDrvCtrl )
{
    int oldLevel;
    char *pBuf;
    int i;
    /* END_RECV_HANDLING_FLAG set by ISR */
    ENDLOGMSG (DRV_DEBUG_RX, ("ne2000HandleRcvInt(%x): enter (flags=%x)\n",
        pDrvCtrl,pDrvCtrl->flags,0,0,0,0));
    pBuf = NULL;
    while (pDrvCtrl->flags & END_RECV_HANDLING_FLAG)
    {
    int len;
    CL_BLK_ID   pClBlk;
    M_BLK_ID    pMblk;      /* MBLK to send upstream */
    ENDLOGMSG (DRV_DEBUG_RX,
                    ("ne2000HandleRcvInt: flags=%x imask=%x cur=%d next=%d\n",
            pDrvCtrl->flags, pDrvCtrl->imask,
            pDrvCtrl->current, pDrvCtrl->nextPacket,
            0, 0));
    /* check if all packets removed */
    if (pDrvCtrl->nextPacket == pDrvCtrl->current)
```

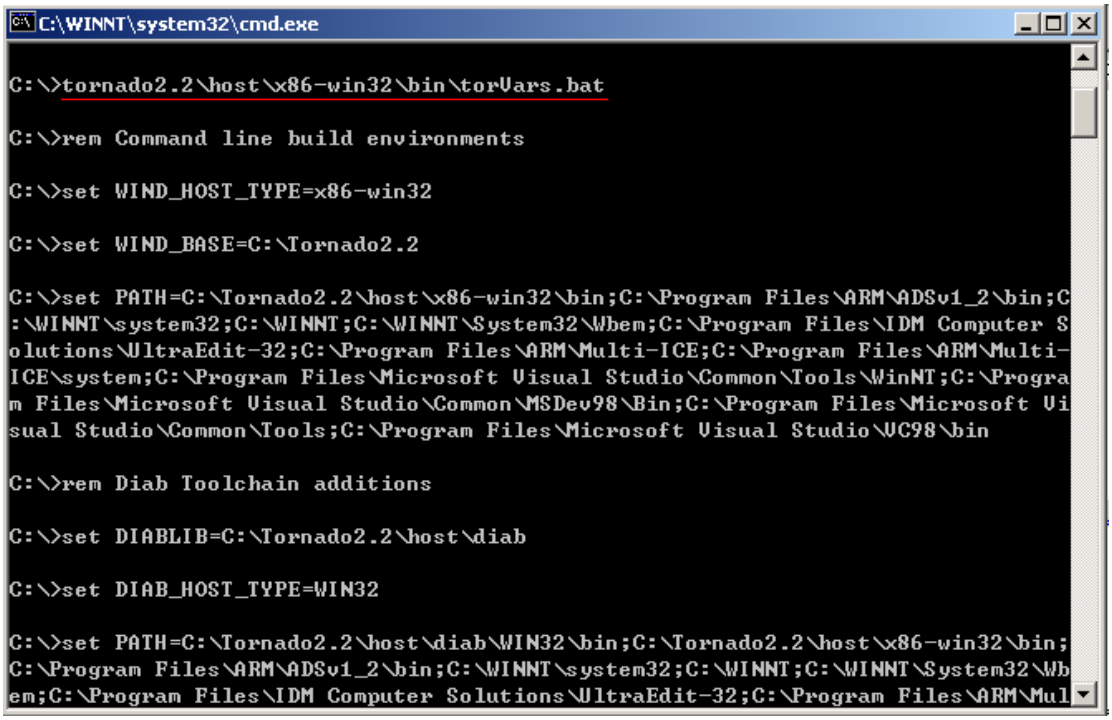
```
        break;
    /* Break out if we get an overwrite condition */
    if (pDrvCtrl->flags & END_OVERWRITE)
        break;
    /* Allocate an MBLK, and a replacement buffer */
    if (!pBuf)
    {
        pBuf = netClusterGet (pDrvCtrl->endObj.pNetPool, pDrvCtrl->clPoolId);
        if (!pBuf)
        {
        ENDLOGMSG (DRV_DEBUG_ERROR,("ne2000HandleRcvInt: Out of clusters!\n", 0,
0, 0, 0, 0, 0));                                /* Tell the mux to give back some
clusters */
        pDrvCtrl->lastError.errCode = END_ERR_NO_BUF;
                muxError(&pDrvCtrl->endObj, &pDrvCtrl->lastError);

        break;
        }
        }
    /* Read packet in offset so IP header is long-aligned */
    len = ne2000PacketGet (pDrvCtrl, pBuf + pDrvCtrl->offset);
    if (len <= 0)
        {
        ENDLOGMSG    (DRV_DEBUG_ERROR,    ("ne2000HandleRcvInt:    bad    packet!
(len=%d)\n",
            len, 0, 0, 0, 0, 0));
        END_ERR_ADD (&pDrvCtrl->endObj, MIB2_IN_ERRS, +1);
        break;
        }
    pMblk = mBlkGet (pDrvCtrl->endObj.pNetPool,
             M_DONTWAIT, MT_DATA);
    if (!pMblk)
        {
        ENDLOGMSG (DRV_DEBUG_ERROR,
                        ("ne2000HandleRcvInt: Out of M Blocks!\n",
            0, 0, 0, 0, 0, 0));
        END_ERR_ADD (&pDrvCtrl->endObj, MIB2_IN_ERRS, +1);
        break;
        }
    pClBlk = netClBlkGet (pDrvCtrl->endObj.pNetPool, M_DONTWAIT);
    if (!pClBlk)
        {
        ENDLOGMSG (DRV_DEBUG_ERROR,
                        ("ne2000HandleRcvInt: Out of CL Blocks!\n",
```

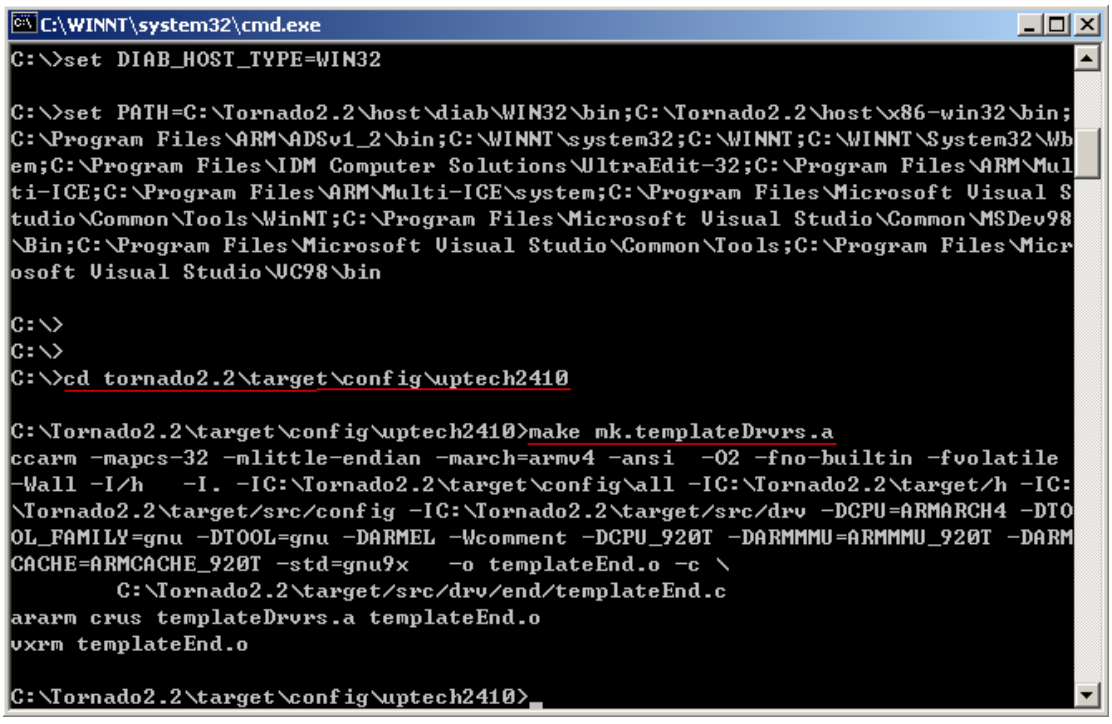
```
            0, 0, 0, 0, 0, 0));
        netMblkFree (pDrvCtrl->endObj.pNetPool, (M_BLK_ID)pMblk);
        break;
        }
    /* Associate the data pointer with the MBLK */
    netClBlkJoin (pClBlk, pBuf, NE2000_BUFSIZ, NULL, 0, 0, 0);

    /* Associate the data pointer with the MBLK */
    netMblkClJoin (pMblk, pClBlk);

    pMblk->mBlkHdr.mFlags |= M_PKTHDR;
    pMblk->mBlkHdr.mLen    = len;
    pMblk->mBlkPktHdr.len  = len;
    /* Adjust mData to match n23000PacketGet() above */
    pMblk->mBlkHdr.mData   += pDrvCtrl->offset;
    #define ETH_DEBUG
    #ifdef ETH_DEBUG
    for(i = 0;i < len; i++)
    printf("0x%02x  \n", pMblk->mBlkHdr.mData[i]);
    #endif
    /* record received packet */
    END_ERR_ADD (&pDrvCtrl->endObj, MIB2_IN_UCAST, +1);


    /* Call the upper layer's receive routine. */
    END_RCV_RTN_CALL (&pDrvCtrl->endObj, pMblk);

    /* buffer was used, will need another next time around */
    pBuf = NULL;
    }

    /* If we still have an unused buffer, free it */
    if (pBuf)
    netClFree (pDrvCtrl->endObj.pNetPool, (UCHAR *) pBuf);

    /* Re-enable the receive interrupt */
    oldLevel = intLock ();
    pDrvCtrl->flags &= ~END_RECV_HANDLING_FLAG;
    pDrvCtrl->imask |= IM_PRXE;
    SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, pDrvCtrl->imask);

    ENDLOGMSG (DRV_DEBUG_RX,
                ("ne2000HandleRcvInt: done (flags=%x, imask=%x)\n",
        pDrvCtrl->flags,pDrvCtrl->imask,0,0,0,0));
```

```
    intUnlock (oldLevel);
    }
```

红色的部分就是我们添加的以太网包的打印代码。在以太网的驱动中，会把收到的数据填入 ppMblk->mBlkHr.mDatad 这个数据结构中，我们只需要把它打印出来就行了。

修改完代码，在 DOS 命令行中进行库的更新。具体操作如下图：torVars.bat 是设置 Tornado 编译环境变量的命令。



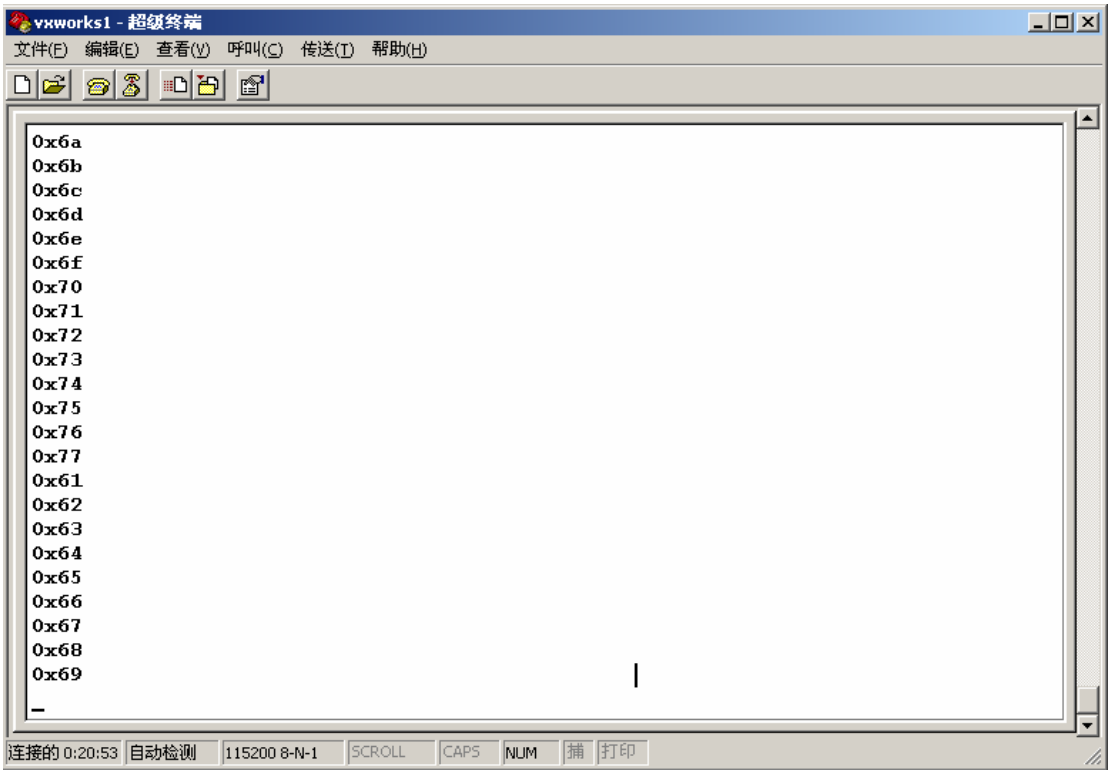


执行完 make mk.templateDrvrs.a，就把 ne2000end.c 文件编译到了库文件，然后重新编译

Bootable 工程，并且更新 VxWorks Image。重新启动。

在 PC 机上 Ping 目标机，就会把目标机收到的以太网数据包显示出来。

如下图：



以太网的数据包格式，这个不多作介绍，如果需要，请查阅相关的书籍。就可以知道每一个字节代表什么意思了。

5.4 多任务环境

当前的实时操作系统大多基于多任务和任务间通信的概念。多任务的环境允许实时应用程序由一组任务构成，每一个任务都将有自己的线程和一组属于自己的系统资源。任务间的通信机制允许任务间同步以及任务间协同工作。在 VxWorks 中，任务间通信包括信号量、消息队列、管道、信号和事件等。

5.4.1 任务

5.4.1.1 概述

VxWorks 支持多任务，其中包括 VxWorks 基本的系统任务和用户创建的任务。用户任务是实行应用目的的执行线程，在本质上和系统任务没有区别，所有任务处于统一地址空间，都运行在 CPU 最高级别的系统模式（具有更好的系统性能）。

VxWorks 的任务调度策略为***优先级抢占式调度***为基础，辅以同级***时间片循环调度***算法。

这一调度算法使得 VxWorks 能够及时地响应高优先级的任务，而同级任务间则可选择同级时间片循环法使多个同优先级的任务并发执行。

5.4.1.2 任务函数库

■ taskSpawn

TaskSpawn	
目标	创建和启动任务
头文件	#include “taskLib.h”
函数原型	int taskID = taskSpawn(char* name, int priority, int options, int stackSize, FUNCPTR entryPt, int arg1…arg10)
参数	name 任务名，字符串形式 priority 任务优先级，从 0（最高）到 255（最低） options 任务选项，一般可以不用，置 0 即可。 stackSize 任务栈大小 entrpPt 任务的入口函数 arg1…arg10 传给 entryPt 函数 10 个 4 字节参数
返回值	ERROR 错误 taskID 任务标志符

例如：

```
#define  TASK_PRI                    120     /* 优先级 */
#define  TASK_STACK_SIZE           5000   /* 任务栈 */
taskId = taskSpawn(“tTest”, TASK_PRI, 0, TASK_STACK_SIZE,
                (FUNCPTR) taskTest,0,0,0,0,0,0,0,0,0,0,0);
```

5.4.2 任务间的通信机制

VxWorks 提供了灵活丰富的任务间通信机制，用于协调任务间的行为，包括信号量、消息队列、管道、信号和事件等。

5.4.2.1 信号量

VxWorks 下的信号量经过优化后已经成为任务之间通信最快的机制。信号量机制可以用于互斥临界区以及任务之间的同步。

VxWorks 提供 3 种信号量：二进制[Binary](同步)信号量、计数[Counting]信号量和互斥[Mutex]信号量。

- 二进制信号量

  二进制信号量也称为开关信号量，可看作计数值为 1 的特殊计数信号量，可用于共享资源互斥和任务同步。由于 VxWorks 有专门的互斥信号量用于资源保护，二进制信号量多用于任务间同步。

- 互斥信号量

  用于资源保护，防止多个任务同时访问连接资源。

  二进制信号量和计数信号量也可以用于互斥操作，而互斥信号量是专门设计用来解决互斥中出现的 3 个常见问题。

  1) 互斥信号量可以被嵌套获取。也就是拥有该信号量的任务可以多次调用 semTake，而不会被阻塞。但放弃使用权时，任务必须调用同样数目的 semGive。
  2) 当任务 tA 拥有删除安全互斥信号量时，任务 tB 删除信号量时会被阻塞，直到 tA 放弃该信号量。因此 tA 正在操作资源时是安全的。用 SEM_DELETE_SAFE 选项创建这种信号量。
  3) 反转安全互斥信号量提供优先级继承属性，能防止所谓的优先级反转问题。用”SEM_Q_PRIORITY|SEM_INVERSION_SAFE”组合选项创建这种信号量。

  注意：互斥信号量有如下限制：只能用于互斥操作，而不能作为同步机制；只能由获取信号量的任务来释放它；不能在中断上下文中使用；不能使用 semFlush 函数。

- 计数信号量

  计数信号量是信号量的一般形式，每一个计数信号量可以被多个任务获取直到计数的限制。

  当用 semGive 操作计数信号量时，如果信号量上没有等待的任务，计数加 1；如果有等待的任务，队列头上的任务转为 ready 状态，而信号量计数维持为 0。

  当用 semTake 操作计数信号量时，如果计数大于 0，semTake 立即返回 OK，任务不会阻塞而继续运行，计数值减 1；如果计数为 0，任务会阻塞在 semTake 调用上。

- 信号量函数

  VxWorks 的 3 种信号量有类似的操作接口，仅在创建函数略有不同：

semBCreat	
目标	为二进制信号量分配空间并将其初始化
头文件	#include “semBLib.h”
函数原型	SEM_ID semID =semBCreate(int options, SEM_B_STATE initialState)
参数	options 选项 SEM_B_STATE 信号量初始状态
返回值	NULL 错误 semID 信号量标志符

二进制信号量的初始状态 SEM_B_STATE

宏	描述
SEM_EMPTY	资源不可用或事件未发生
SEM_FULL	资源可用或事件已发生

semMCreat	
目标	为互斥信号量分配空间并将其初始化
头文件	#include “semMLib.h”
函数原型	SEM_ID semID =semMCreate(int options)

参数	options 选项
返回值	NULL 错误 semID 信号量标志符

互斥信号量用于控制临界资源访问，初始时都假设资源可用。

semCCreat	
目标	为计数信号量分配空间并将其初始化
头文件	#include “semCLib.h”
函数原型	SEM_ID semID =semBCreate(int options, int initialState)
参数	options 选项 initialState 初始化的计数值
返回值	NULL 错误 semID 信号量标志符

三种信号量的第一个参数 options 定义：

Options	值	描述
SEM_Q_FIFO	0x0	等待任务按先进先出方式排列
SEM_Q_PRIORITY	0x1	等待任务按优先级方式排列
SEM_DELETE_SAFE	0x4	避免删除信号量占有任务（互斥信号量专用）
SEM_INVERSION_SAFE	0x8	允许优先级继承，必须配合 SEM_Q_PRIORITY 使用（互斥信号量专用）

SemShow	
目标	显示信号量信息
头文件	#include “semLib.h”
函数原型	STATUS semShow(SEM_ID semId, Int level)
参数	SEM_ID semId,要显示的信号量 ID int level,0 表示显示概要信息 1 表示显示详细信息
返回值	NULL 错误 semID 信号量标志符

例如：

/*创建二进制信号量，按优先级排队，初始状态为可用*/

```
SEM_ID semId = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

/* 创建互斥信号量，属性为优先级方式，避免删除信号量占有任务,允许优先级继承*/

```
SEM_ID mutexSemId = semMCreate (SEM_Q_PRIORITY | SEM_DELETE_SAFE
```

```
                                                | SEM_INVERSION_SAFE);
```

/*创建计数器信号量，按优先级排队，初始状态为不可用*/

```
SEM_ID semId = semBCreate(SEM_Q_PRIORITY, SEM_EMPTY);
```

下面是常用的信号量操作函数：

semDelete	
目标	终止并释放指定的信号量
头文件	#include “semLib.h”
函数原型	STATUS nResult =semDelete(SEM_ID semId)
参数	semId 信号量标志符
返回值	ERROR 错误 OK 成功

semTake	
目标	获取指定的信号量
头文件	#include “semLib.h”
函数原型	STATUS nResult =semTake(SEM_ID semId, int timeout)
参数	semId 信号量标志符 timeout 超时时间，单位为 tick 其中 WAIT_FOREVER(-1) NO_WAIT(0)
返回值	ERROR 错误 OK 成功

例如：

```
semTake (semId, WAIT_FOREVER); /*永远等待 semId 变为可用*/
```

SemGive	
目标	释放指定的信号量
头文件	#include “semLib.h”
函数原型	STATUS nResult =semGive(SEM_ID semId)
参数	semId 信号量标志符
返回值	ERROR 错误 OK 成功

SemFlush	
目标	使所有等待指定信号量的任务处于就绪态
头文件	#include “semLib.h”
函数原型	STATUS nResult =semFlush(SEM_ID semId)
参数	semId 信号量标志符
返回值	ERROR 错误 OK 成功

注意：互斥信号量不可以用 semFlush()。

5.4.2.2 消息队列

虽然信号量为任务间同步提供了高性能的机制，但是交换的信息有限，在 VxWorks 中，提供了消息队列的机制用于任务间信息的交换。

VxWorks 所提供的消息队列允许传递那些长度、数目可变的消息，而且允许消息排队。一个任务既可以发送消息，同时也能接受消息。在通常情况下，可以把消息队列看成一个单方向的设备，对同一个任务来说，这个设备只能读取或者写入。当然，也可以实现全双工的消息队列，这样就需要两个消息队列来实现，一个用于发送，一个用于接收。

有关消息队列的函数调用如下：

msgQCreate	
目标	为消息队列分配空间并将其初始化
头文件	#include “msgQLib.h”
函数原型	MSG_Q_ID msgId =msgQCreate(int maxMsgs, int maxMsgLength, int options)
参数	maxMsgs 队列中允许的最大消息个数 maxMsgLength 最长消息的字节数 options 消息队列类型 MSG_Q_FIFO 先进先出类型 MSG_Q_PRIORITY 基于优先级类型
返回值	ERROR 错误 msgID 消息队列标识符

例如：

```
struct msg {                                    /* 传递的消息结构体 */
        int tid;                                /* 任务号*/
        int value;                               /* 传递的值 */
        };
int numMsg = 3;
/*创建一个消息队列,最大消息个数为 3，先入先出类型*/
msgQId = msgQCreate (numMsg, sizeof (struct msg), MSG_Q_FIFO);
```

MsgQDelete	
目标	删除消息队列并将其释放
头文件	#include “msgQLib.h”
函数原型	STATUS nResult = msgQDelete(MSG_Q_ID msgId)
参数	msgId 消息队列标识符
返回值	ERROR 错误 OK 成功

MsgQSend	
目标	向一个消息队列发送消息
头文件	#include “msgQLib.h”
函数原型	STATUS nResult = msgQSend(MSG_Q_ID msgId, char* buffer,

	UINT nBytes, int timeout, int priotity)
参数	msgId 消息队列标识符 buffer 消息数据缓冲区地址 nBytes 传送消息的长度 timeout 消息队列满情况下任务等待延时，单位 tick 特殊地， WAIT_FOREVER(-1) NO_WAIT(0) priority 消息等级 MSG_PRI_NORMAL（0） 平常等级，新消息添加到队列尾部 MSG_PRI_URGENT（1） 紧急消息，新消息添加到队列头部
返回值	ERROR 错误 OK 成功

例如：

```
struct msg producedItem;            /* 发送缓存 */
 /* 发送消息*/
 if ((msgQSend (msgQId, (char *) &producedItem, sizeof (producedItem),
                WAIT_FOREVER,  MSG_PRI_NORMAL)) == ERROR)
     {
         perror ("Error in sending the message");
         return (ERROR);
     }
```

MsgQReceive	
目标	从一个消息队列接收消息
头文件	#include “msgQLib.h”
函数原型	int mLen = msgQReceive(MSG_Q_ID msgId, char* buffer, UINT maxNBytes, int timeout)
参数	msgId 消息队列标识符 buffer 接收缓存指针 maxNBytes 接收缓存大小 timeout 消息队列为空时任务等待延时，单位 tick 特殊地， WAIT_FOREVER(-1) NO_WAIT(0)
返回值	ERROR 错误 OK 成功

例如：

```
struct msg consumedItem;            /* 接收缓存 */
/* 接受消息 */
if ((msgQReceive (msgQId, (char *) &consumedItem,
                  sizeof (consumedItem), WAIT_FOREVER)) == ERROR)
    {
    perror ("Error in receiving the message");
    return (ERROR);
```

```
    }
```

5.4.2.3 管道

管道是消息队列的替代品，与消息队列不同的是，管道使用了 VxWorks 系统的 I/O 系统。管道是由驱动 pipeDrv 提供的虚拟 I/O 设备，而在底层实现使用了消息队列，所以我们可以通过 VxWorks 标准接口 open()、read()、write()、ioctl()来访问管道。

作为一个 I/O 设备，管道提供了一个消息队列所不能提供的支持，即是对 select 的支持。通过使用 select 函数，任务可以同时等待不同的 I/O 设备的数据。

在创建管道设备时，需要调用 pipeDevCreate()函数，然后就可以通过标准 I/O 接口函数操作管道了。

pipeDrvCreate	
目标	创建一个管道设备
头文件	#include “pipeDrv.h”
函数原型	STATUS nResult = pipeDrvCreate(char* name, int nMessages, int nBytes)
参数	name 管道设备名 nMessages 队列中允许的最大消息个数 nBytes 最长消息的字节数
返回值	ERROR 错误 OK 成功

例如：

```
#define PIPE_NAME       "/pipe/pipe0"  /* 管道设备的文件名*/
#define MAX_MSGNO       3              /*最大消息数*/
#define MAX_MSGLEN  sizeof (int)       /*消息大小*/
 /*创建管道*/
if (pipeDevCreate (PIPE_NAME, MAX_MSGNO, MAX_MSGLEN) == ERROR)
    {
    perror ("Error in creating pipe");
    }
```

5.4.2.4 信号

VxWorks 支持软件信号功能。信号是软件中断，可以异步的改变任务的控制流程。在实际操作中，中断服务程序或者任务向指定的任务发送信号，接收到信号的任务立即挂起当前执行的线程，在下一次调度执行的时候执行指定的信号处理程序。与其他的任务间通信机制比较，信号机制更适合于错误和异常的处理。信号程序可以看作是一种”软”中断的服务程序，所以任何可能导致任务阻塞的函数都不能用在信号处理程序中。

wind 内核中支持了两种类型的信号：UNIX BSD 接口和 POSIX 接口的信号。

以 UNIX BSD 接口为例，VxWorks 中缺省的包含了 UNIX BSD 接口的信号组件 INCLUDE_SIGNALS，

在 Tornado 环境下是 operating system components->kernel components->signal.

常用函数如下：

signal	
目标	指定一个信号的处理函数
头文件	#include “sigLib.h”
函数原型	void * pResult = signal(int signo, void (*pHandler)())
参数	Signo 要处理的信号 pHandler 处理函数
返回值	ERROR 错误 pResult 成功,返回前一个处理函数

常用的信号 sig 有以下值：

宏	描述
SIGBUS	指示一个实现定义的硬件故障
SIGILL	指示执行一条非法的硬件指令
SIGINT	当用户按下中断键(Ctrl-C)时，终端驱动程序产生此信号
SIGEGV	进行了一次无效的存储访问
SIGUSR1	用户自定义信号 1
SIGUSR2	用户自定义信号 2

处理函数 pHandler 可以有以下三种选择：

值	描述
SIG_IGN	忽略信号
SIG_DFL	默认处理
pHandler	用户自定义的处理函数

例如：

```
/*将信号处理函数与 SIGUSR1 信号绑定*/
signal (SIGUSR1, (VOIDFUNCPTR)sigHandler);
```

kill	
目标	向一个任务发送一个信号
头文件	#include “sigLib.h”
函数原型	int nResult = kill(int tid, int signo)
参数	tid 目标任务 ID signo 要处理的信号
返回值	ERROR 错误 OK 成功

例如：

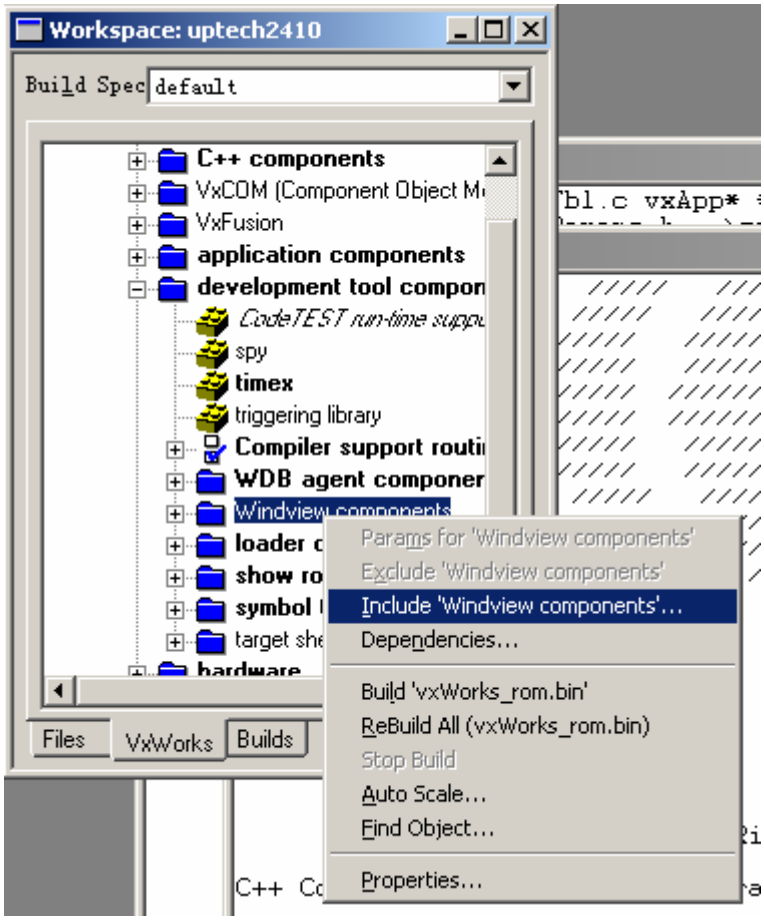
```
/*向 syncTask 发送信号 SIGUSR1*/
kill(syncTaskTid,SIGUSR1);
```

5.4.4 实验

5.4.4.1 创建任务，利用 WindView 观察任务调度

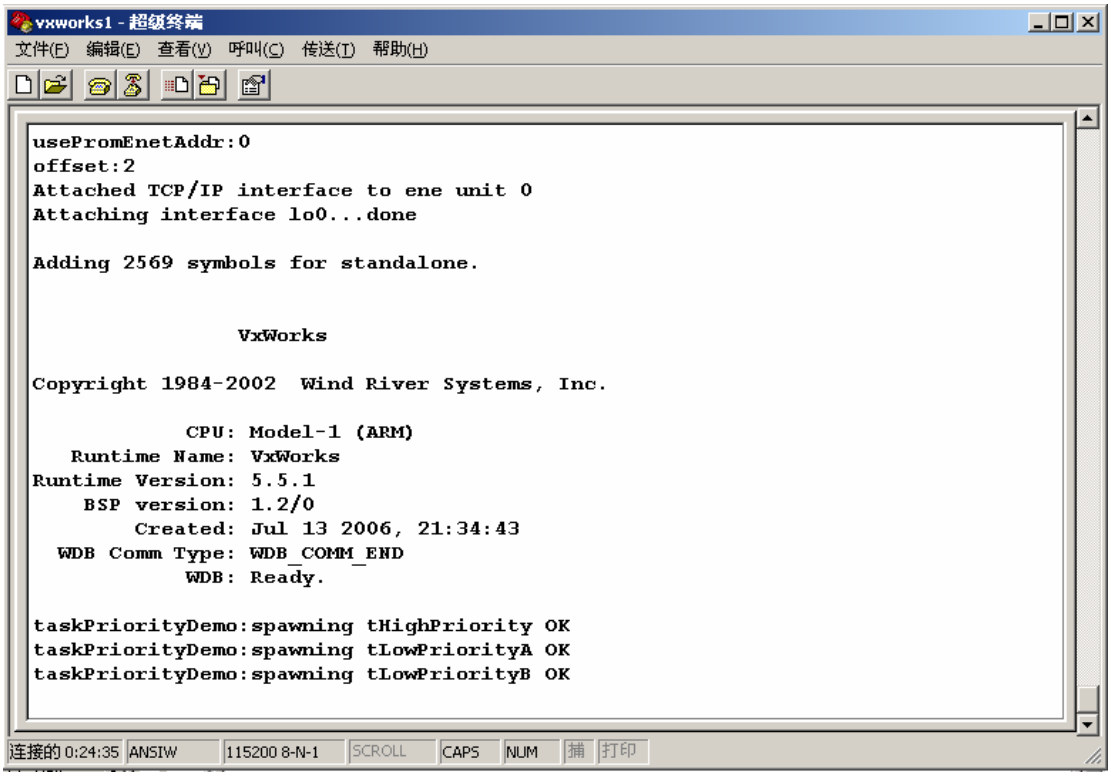
*本实验的具体程序见 Multi-Tasking 文件夹下的 taskPriorityDemo.c。*主函数为 taskPriorityDemo，创建了三个任务，tHighPriority 优先级较高,tLowPriorityA 和 tLowPriorityB 优先级相同，三个任务都是执行 LOOP_TIMES 次循环后退出，这样保证了三个任务在执行过程中均处于 ready 状态，便于利用 WindView 观察。

要使用 WindView 察看 Target 中的任务运行情况，必须在建立 Bootable 工程的时候把 WindView 包含进入，如下图：



然后编译，并且下载或烧入目标机中。

实验步骤和上面的一样，这里简单介绍一下：建立 Downloadable 工程，然后编译，最后下载到目标机，然后在 Target Shell 或者 WindSh 下输入：sp taskPriorityDemo ：

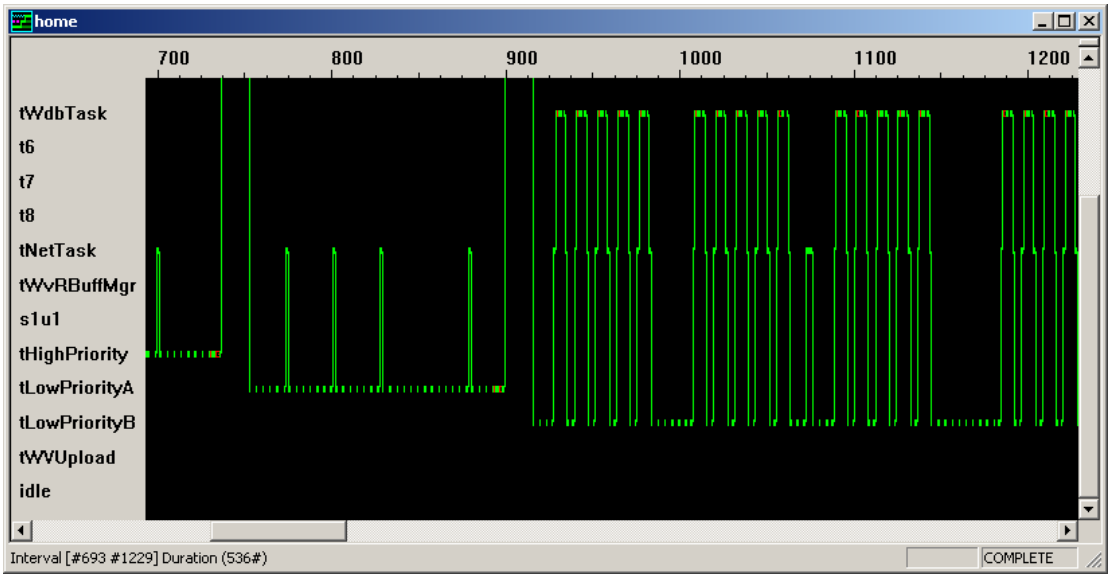


此时用 WindView 观察，VxWorks 默认采用优先级抢占任务调度算法，根据它的原则，高级 ready 状态的任务可以抢占正在运行的低级任务，idle 为空闲任务，优先级最低。

根据优先级抢占算法的原则，同级任务之间是不允许抢占的，这时只遵循 FIFO 的基本原则进行调度：

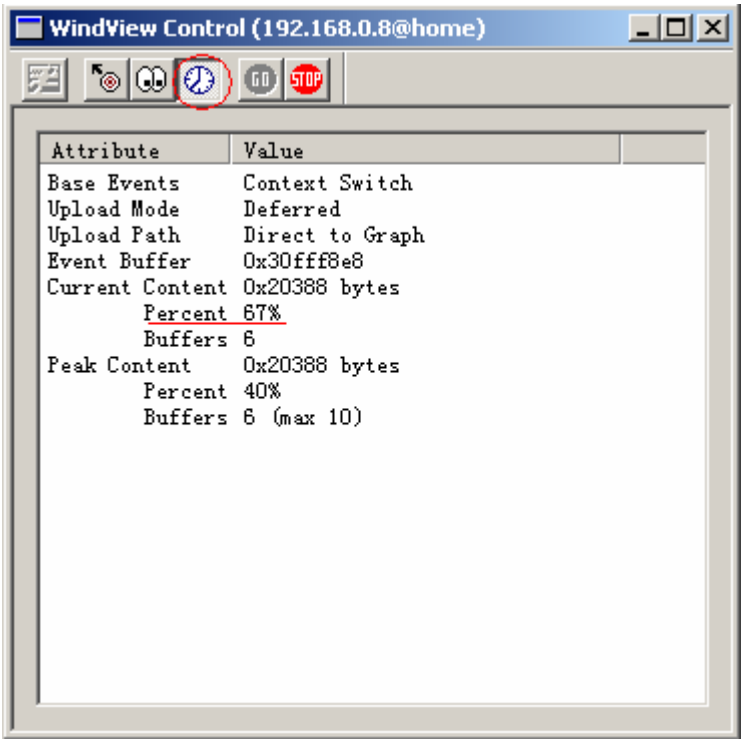
同级任务不可抢占，必须等正执行的同级任务完成后，等待的任务才能运行；如果不断有同级任务转为 ready 状态，这些任务按状态变化的时间顺序排入等待队列，先进入队列的先运行。如图，tLowPriorityA 和 tLowPriorityB 为同级任务，因先创建的 tLowPriorityA，所以 tLowPriorityB 要等到 tLowPriorityA 执行完毕后才执行。

具体的 WindView 的使用方法，在这里不作具体的介绍。请参考《WindView User’s Guide》。



从上面这个图中看出，tHighPriority 的任务先运行，然后才是另外两个低优先级的任务。由于先创建的 tLowPriorityA，所以他先运行，然后才是 tLowPriorityB。这些任务运行的时候，其他的高优先级的任务都可以打断他们的运行。

注意：在使用 WindView 时候，点击类似钟表的图标，直到 Perncent 到 50%以上的时候再点击 Upload 的图标，这样才会看到比较全面的 WindView 图示。



5.4.4.2 利用二进制信号量同步任务

具体程序见 Multi-Tasking 文件夹下的 binarySemDemo.c。主函数 synchronizeDemo，它创建了两个二进制信号量(semId1 和 semId2)，和两个任务(TaskA 和 TaskB)，TaskA 需要执行 EventA，TaskB 需要执行 EvernB，利用二进制信号量同步。

```
/* synchronizeDemo.c - 用二进制信号量同步任务
 */
#include "vxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "stdio.h"
#include "sysLib.h"

#define  TASK_PRI                98      /* 任务优先级 */
#define  TASK_STACK_SIZE         5000    /* 任务栈 */

LOCAL SEM_ID semId1;            /*binary semaphore 1 的 ID*/
LOCAL SEM_ID semId2;            /* binary semaphore 2 的 ID*/
```

```
LOCAL int numTimes = 2;          /* 反复次数*/
LOCAL BOOL notDone;               /*结束标志*/
LOCAL STATUS taskA ();
LOCAL STATUS taskB ();

/*******************************************************************************
 *  synchronizeDemo - 主函数
 *
 *  描述
 *  创建两个二进制信号量(semId1 和 semId2).同步两个任务(TaskA 和 TaskB)
 *
 *  返回值:
 *  OK
 *  ERROR
 */

STATUS synchronizeDemo ()
    {
    notDone = TRUE;

    /* 创建二进制信号量 semId1,并使之可用*/
    if ((semId1 = semBCreate (SEM_Q_PRIORITY, SEM_FULL)) == NULL)
        {
        perror ("synchronizeDemo: Error in creating semId1 semaphore");
        return (ERROR);
        }

    /* 创建二进制信号量 semId2,并使之不可用*/
    if ((semId2 = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY)) == NULL)
        {
        perror ("synchronizeDemo: Error in creating semId2 semaphore");
        return (ERROR);
        }

    /* 创建 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);
        }

    /* 创建 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);
        }

    /* 轮循检查是否完成，如果循环完毕，则删除信号量 */
    while (notDone)
        taskDelay (sysClkRateGet());  /* 延时 1 秒,sysClkRateGet()返回 1 秒钟的
tick 值*/
        /* 删除信号量 */
    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 - 等待 semId1 变为可用, 执行 EventA,执行完毕后,释放 semId2,
 *             以实现任务同步
 *
 *  返回值: OK or ERROR
 *
 */

LOCAL STATUS taskA ()
    {
    int count;

    for (count = 0; count < numTimes; count++)
        {
        /*等待 semId1 变为可用,永远等待*/
        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()));
       /*taskName (taskIdSelf()) 得到任务名:tTaskA*/
        printf("taskA:  I'm  done,  taskB  can  now  proceed;  Releasing  semId2
semaphore\n\n");
         /*释放 semId2 的控制权*/
        if (semGive (semId2) == ERROR)
            {
            perror ("taskA: Error in semGive");
            return (ERROR);
            }
        }
        return (OK);
    }


/*******************************************************************************
 *  taskB - 等待 semId2 变为可用, 执行 EventB,执行完毕后,释放 semId1,
 *             以实现任务同步
 *
 *  返回值: OK or ERROR
 *
 */
LOCAL STATUS taskB()
    {
    int count;

    for (count = 0; count < numTimes; count++)
        {
        /*等待 semId2 变为可用,永远等待*/
        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()));
   /*taskName (taskIdSelf()) 得到任务名:tTaskB*/
        printf("taskB:  I'm  done,  taskA  can  now  proceed;  Releasing  semId1
semaphore\n\n\n");
```

```
        /*释放 semId2 的控制权*/
        if (semGive (semId1) == ERROR)
            {
            perror ("taskB: Error in semGive");
            return (ERROR);
            }
        }
        notDone = FALSE;
        return (OK);
    }
```

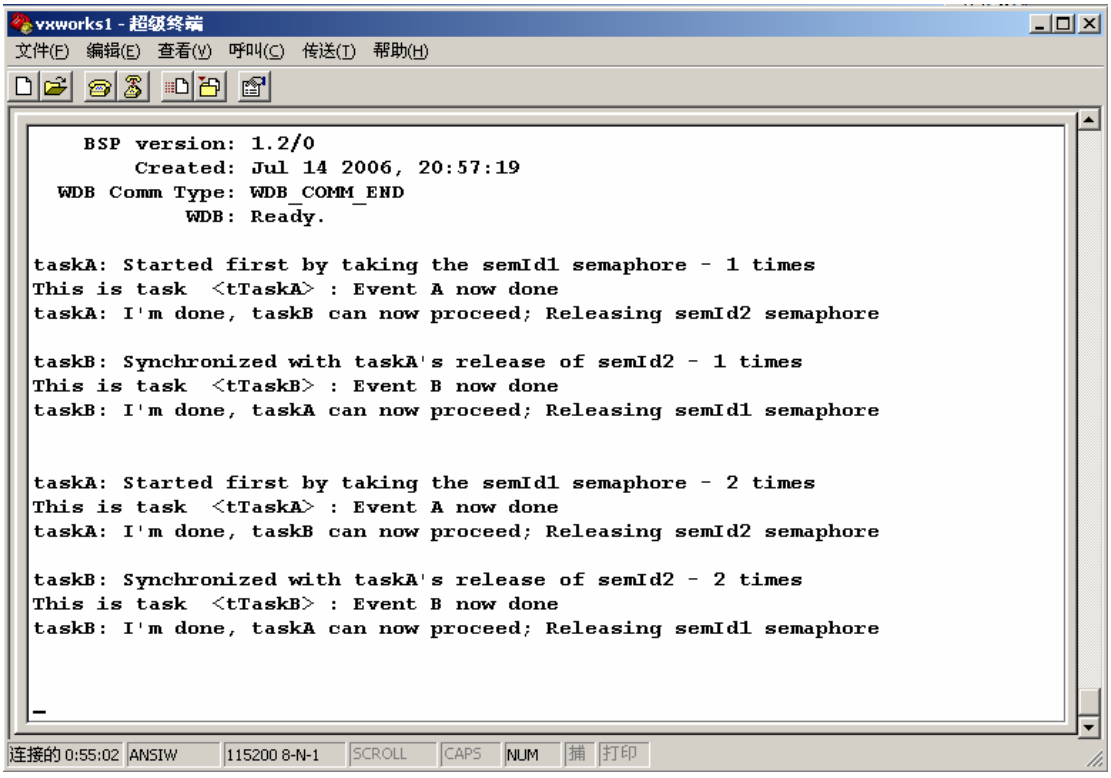
说明，主程序创建了两个信号量，其中一个即 semId1，是一个创建以后就可以是被获得的，而 semId2 是创建以后是空的，即不能被获得，但是可以被释放。

TaskA 是先获得 semId1，然后释放 semId2；

TaskB 是先获得 semId2，然后释放 semId1。

这样执行的顺序应该是，TaskA 先获得 semId1，然后释放 semId2，接着 TaskB 获得 semId2，然后释放 semId1。加上循环功能，则 TaskA 必须等到 TaskB 释放 semId1 才能继续下次循环。既能出现下面的结果。

在 Target Shell 或者 WindSh 下输入 synchronizeDemo。



5.4.4.3 利用计数器信号量同步任务

后边的例子的过程，都是建立 Downloadable 工程，编译并且下载，方法同前

用二进制信号量同步任务时,如果事件发生的过快,有可能任务来不及处理，（因为可能

获取信号量的任务运行的过快，可能会阻止释放信号量的任务运行）使用计数器信号量可以解决这个问题. 这个实验可以选择用二进制信号量或是计数器信号量, 从而理解两种类型的区别。具体程序在 Multi-Tasking 文件夹下的 countingSemDemo.c 文件。主函数是 countingSemDemo，它允许用户选择信号量的类型来实现运行中的任务和模拟的外部中断之间的同步。这里用到了看门狗定时器来模拟外部中断，每隔 1tick 模拟发送一个中断，由 syncTask 接收处理。看门狗定时器的使用请参见 3.3.2 节的有关介绍以及相关实验。

```
/* countingSemDemo.c - 利用计数信号量同步任务
 */


/*
 * DESCRIPTION
 * 用二进制信号量同步任务时,如果事件发生 的过快,
 * 有可能会造成任务来不及处理.    使用计数器信号量可
 * 以解决这个问题.
 * RETURNS: OK or ERROR
 *
 * EXAMPLE

 *
 */

/* include files */

#include "vxWorks.h"
#include "wdLib.h"
#include "stdio.h"
#include "semLib.h"
#include "taskLib.h"
#include "usrLib.h"
#include "sysLib.h"

/* defines */

#define TASK_PRIORITY               101
#define TASK_STACK_SIZE                 5000
#define TIME_BETWEEN_INTERRUPTS                 1   /* 1 tick */
#define TASK_WORK_TIME              2   /* 2 ticks */
#define NUM_OF_GIVES                3

/* globals */

LOCAL SEM_ID semId = NULL;      /* 信号量 ID*/
LOCAL WDOG_ID wdId = NULL;              /* 看门狗定时器 ID */
```

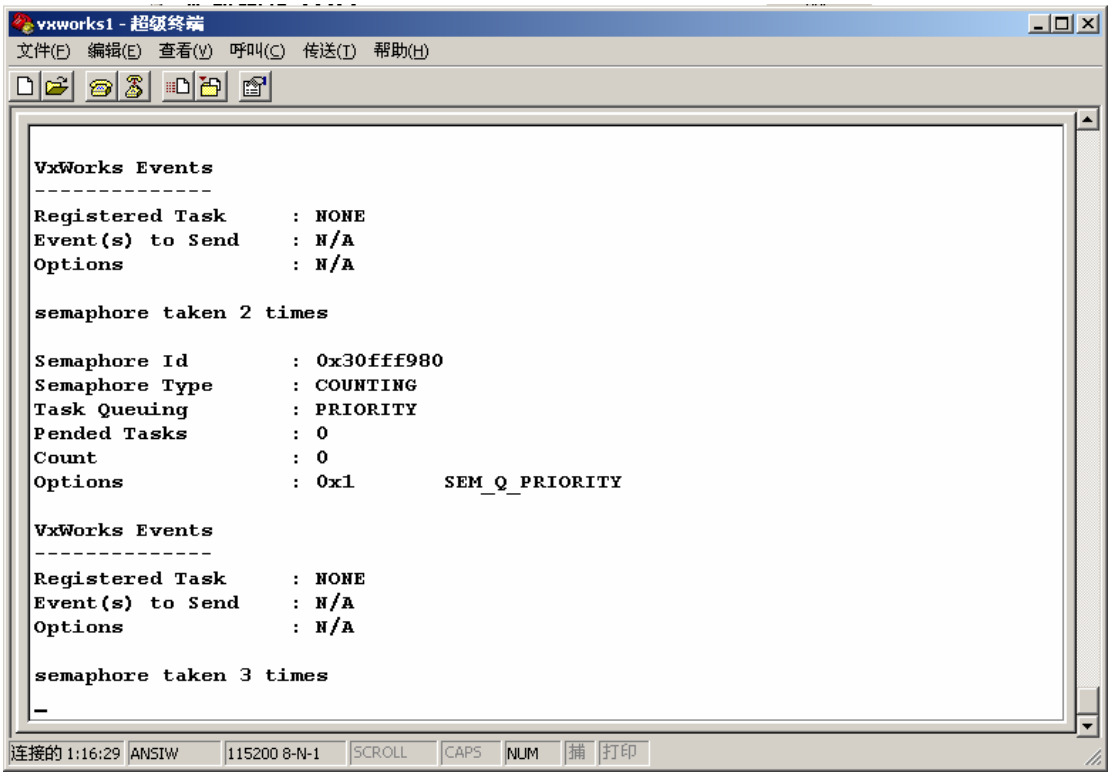
```
LOCAL int syncTaskTid = 0;                /*syncTask 的 tid */
LOCAL int numToGive = NUM_OF_GIVES;       /* semGive 被调用的次数 */

/* forward declaratiuon */
void syncISR(int);
void cleanUp ();
void syncTask ();


/*******************************************************************************
 * countingSemDemo - 主函数. 可以让用户选择信号量的类型
 *
 *  RETURNS: OK or ERROR
 *
 */

STATUS countingSemDemo (void )
    {
   /*创建一个计数器 信号量,初始值为 0,不可用*/
    if ((semId = semCCreate (SEM_Q_PRIORITY, 0)) == NULL)
     {
                perror ("semCCreate");
                return (ERROR);
     }

     /*创建一个看门狗定时器,模拟外部中断*/
    if ((wdId = wdCreate()) == NULL)
        {
        perror ("wdCreate");
        cleanUp ();
        return (ERROR);
        }

     /*创建任务*/
    if    ((syncTaskTid    =    taskSpawn    ("tsyncTask",    TASK_PRIORITY,
0,TASK_STACK_SIZE,(FUNCPTR) syncTask, 0,0,0,0,0,0,0,0,0,0)) == ERROR)
        {
   perror ("taskSpawn");
   cleanUp();
   return (ERROR);
   }

    /* 启动看门狗定时器 */
    if (wdStart (wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, numToGive)
```

```
                    == ERROR)
        {
        perror ("wdStart");
        cleanUp ();
        return (ERROR);
        }

    /* 延时,然后结束演示，延时的时间是根据任务的运行时间和信号量使用的次数来定，本来应该 TASK_WORK_TIME * numToGive ，为了保证任务能正确运行，我们多加一些，就变成 TASK_WORK_TIME + 2) * numToGive 。前面的 sysClkRateGet() ，也是为了保证这点，其实可以去掉*/
    taskDelay (sysClkRateGet()  + ((TASK_WORK_TIME + 2) * numToGive));

    cleanUp();
    return (OK);
    }


/*******************************************************************************
 *  syncTask -运行的任务, 和模拟的外部中断同步
 */

void syncTask (void)
    {
    int eventCount = 0;

    FOREVER
        {
   if (semTake (semId, WAIT_FOREVER) == ERROR)
        {
            perror ("syncTask semTake");
            return;
            }

   taskDelay (TASK_WORK_TIME);
   semShow (semId,1);

   eventCount++;
   printf ("semaphore taken %d times\n", eventCount);
   }

    }
```

```
/*******************************************************************************
 * syncISR - 模拟的外部中断,和运行中的任务同步
 */
void syncISR
    (
    int times
    )
    {
    semGive (semId);
    times--;
    if (times > 0)
        wdStart (wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, times);
    }

/*******************************************************************************
 * cleanUP - 删除任务和模拟的中断,以及相关的资源,结束演示
 */
void cleanUp ()
    {
    if (syncTaskTid)
        taskDelete (syncTaskTid);
    if (semId)
        semDelete (semId);
    if (wdId)
        wdDelete (wdId);

    }
```

从上面的程序中看分析出，在看门狗的函数中，会定期是放信号量，一共释放三次。Task 会循环的获得信号量。

在 Target Shell 或者 WindSh 下输入 sp countingSemDemo 。



从上面结果中看到，一共释放了三次信号量，而 syncTask 都获得了。

5.4.4.4 利用互斥信号量保护共享资源

互斥信号量主要用于资源保护，以防止多个任务同时访问临界资源。本实验创建两个任务 producerTask 和 consumerTask,两个任务是通过共享内存的方式通信,都可能读写全局变量 shMemResource,为了防止两个任务同时访问 shMemResource，使用互斥信号量进行保护。*具体程序见 Multi-Tasking 文件夹下的 mutexSemDemo.c*

```
/* mutexSemDemo.c -  使用互斥信号量保护共享资源的例子
 */


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

#define  CONSUMER_TASK_PRI          98
#define  PRODUCER_TASK_PRI          99
#define  TASK_STACK_SIZE          5000
```

```
#define PRODUCED  1
#define CONSUMED  0
#define NUM_ITEMS 3                         /* 总共处理的次数 */
struct shMem                                /* 共享内存结构体 */
    {
    int tid;                                /* 任务号*/
    int count;                              /*处理的次数 */
    int status;                             /* 状态
                                                    0 : consumed
                                                    1 :  produced*/
    };


LOCAL STATUS protectSharedResource ();
LOCAL STATUS releaseProtectedSharedResource ();
LOCAL STATUS producerTask ();
LOCAL STATUS consumerTask ();
LOCAL struct shMem shMemResource;           /*共享内存*/
LOCAL SEM_ID mutexSemId;                    /* 互斥信号量的 ID*/
LOCAL BOOL notFinished;                     /* 完成的标志 */


/*******************************************************************************
    *  mutexSemDemo - 主函数
    *
    *  DESCRIPTION
    *  创建两个任务 producerTask 和 consumerTask,两个任务是通过共享内存的方式通信,
    *   都可能 读写全局变量 shMemResource,为了防止两个任务同时访问 shMemResource,
    *  使用互斥信号量进行保护.
    *
    *  RETURNS: OK or ERROR
    *
    *  EXAMPLE
    *
    *  -> sp mutexSemDemo
    *
    */

STATUS mutexSemDemo()
    {
    notFinished = TRUE;
```

```
    /* 创建互斥信号量, 属性为优先级方式, 避免删除信号量占有任务,允许优先级
继承*/
    if ((mutexSemId = semMCreate (SEM_Q_PRIORITY | SEM_DELETE_SAFE
                                  | SEM_INVERSION_SAFE)) == NULL)
        {
        perror ("Error in creating mutual exclusion semaphore");
        return (ERROR);
        }

    /* 创建任务 consumerTask  */
    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);
        }

     /* 创建任务 producerTask  */
    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);
        }

    /* 轮询结束标志 */

    while (notFinished)
        taskDelay (sysClkRateGet ());

    /* 删除信号量*/
    if (semDelete (mutexSemId) == ERROR)
        {
       perror ("Error in deleting mutual exclusion semaphore");
       return (ERROR);
        }

     return (OK);
     }
```

```
/*******************************************************************************
 *  producerTask - 产生信息的任务,向共享内存 shMemResource 赋值
 *
 *  RETURNS: OK or ERROR
 *
 */

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

    while (notDone)
        {

        /* 循环 NUM_ITEMS 次
         */

        if (count < NUM_ITEMS)
            {

            /* 等待共享资源变为可用，即获得信号量 */
            if (protectSharedResource() == ERROR)
                return (ERROR);

            /* 赋值，访问共享资源 */
            if (shMemResource.status == CONSUMED)
                {
                count++;
                shMemResource.tid = taskIdSelf ();
                shMemResource.count = count;
                shMemResource.status = PRODUCED;
                }
            /* 释放控制权 */
            if (releaseProtectedSharedResource () == ERROR)
                return (ERROR);

            logMsg ("ProducerTask: tid = %#x, producing item = %d\n",
                                                    taskIdSelf      (),
count,0,0,0,0);

            taskDelay (sysClkRateGet()/6);
```

```
                }
            else
                notDone = FALSE;
            }

        return (OK);
        }


/*******************************************************************************
    *  consumerTask -  通过共享内存 shMemResource 取得信息,并将 status 变为
CONSUMED
    *
    *  RETURNS: OK or ERROR
    *
    */

    LOCAL STATUS consumerTask ()
        {
        int notDone = TRUE;

        while (notDone)
            {
            taskDelay (sysClkRateGet()/6);

            /* 等待共享资源变为可用，获得信号量 */
            if (protectSharedResource() == ERROR)
                return (ERROR);

            /* 访问共享资源 */
            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;

            /* 释放控制权 */
            if (releaseProtectedSharedResource () == ERROR)
                return (ERROR);
```

```
        }

    notFinished = FALSE;
    return (OK);
    }


/*******************************************************************************
 *  protectSharedResource -  semTake 得到共享资源的访问权
 *
 *  RETURNS: OK or ERROR
 *
 */
LOCAL STATUS protectSharedResource ()
    {
    if (semTake (mutexSemId, WAIT_FOREVER) == ERROR)
        {
        perror ("protectSharedResource: Error in semTake");
        return (ERROR);
        }
    else
        return (OK);
    }


/*******************************************************************************
 *  releaseProtectedSharedResource -  semGive 释放共享资源的控制权
 *
 *  RETURNS: OK or ERROR
 *
 */
LOCAL STATUS releaseProtectedSharedResource ()
    {
    if (semGive (mutexSemId) == ERROR)
        {
        perror ("protectSharedResource: Error in semTake");
        return (ERROR);
        }
    else
        return (OK);
    }
```

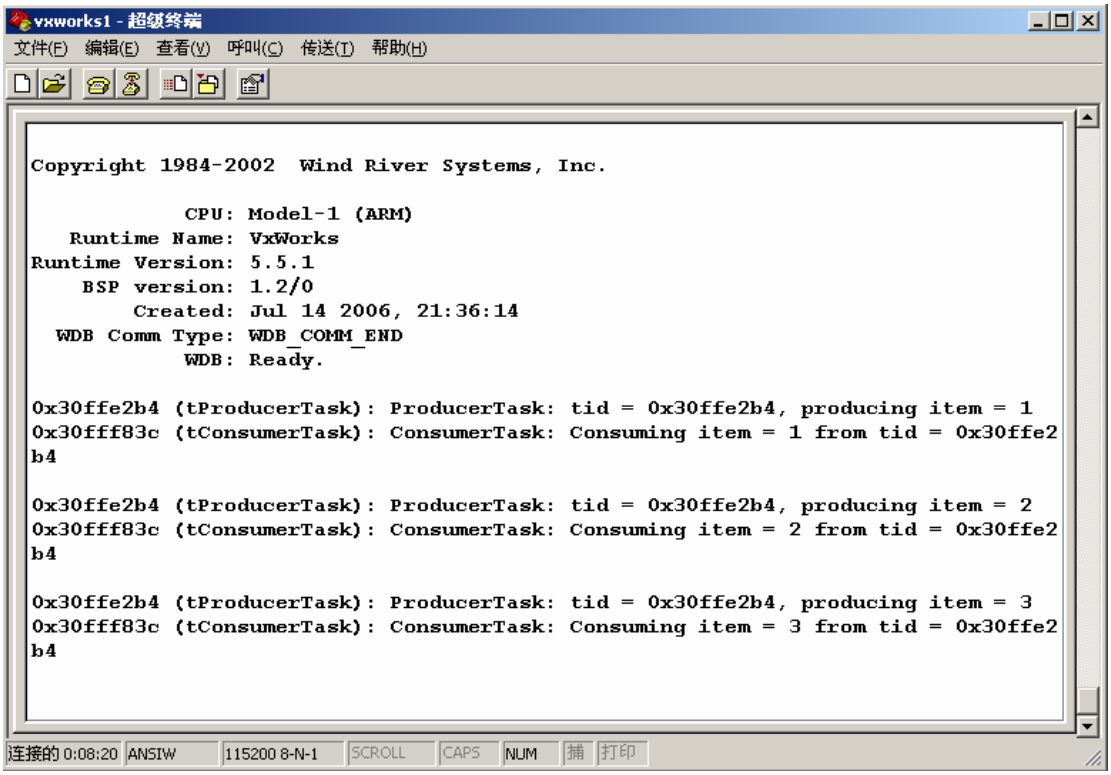
这里简单介绍一下 logMsg 的用法：

```
int logMsg
(
char * fmt,                /* format string for print */
int    arg1,               /* first of six required args for fmt */
int    arg2,
int    arg3,
int    arg4,
int    arg5,
int    arg6
)
```

第一个参数和 printf 的参数一样，是一个打印的字符串格式，后面的 6 个参数也和 printf 一样是要打印的值。LogMsg 是不独占 CPU 资源的，可以在中断函数使用。

分析上面程序，producerTask 是获得共享资源，并且付值，然后释放。consumerTask 是获得共享资源，打印信息，然后释放。由于 producerTask 优先级高，所以先运行。

在 Target Shell 或者 WindSh 下输入 sp mutexSemDemo ：



5.4.4.5 利用消息队列进行任务间通信

具体程序见 Multi-Tasking 文件夹下的 msgQDemo.c。两个任务(producerTask 和 consumerTask)，并在 producerTask 和 consumerTask 之间创建一个消息队列，producerTask 通过消息队列发送消息， consumerTask 通过消息队列接收消息. 当

consumerTask 接收完所有消息,关闭消息队列。

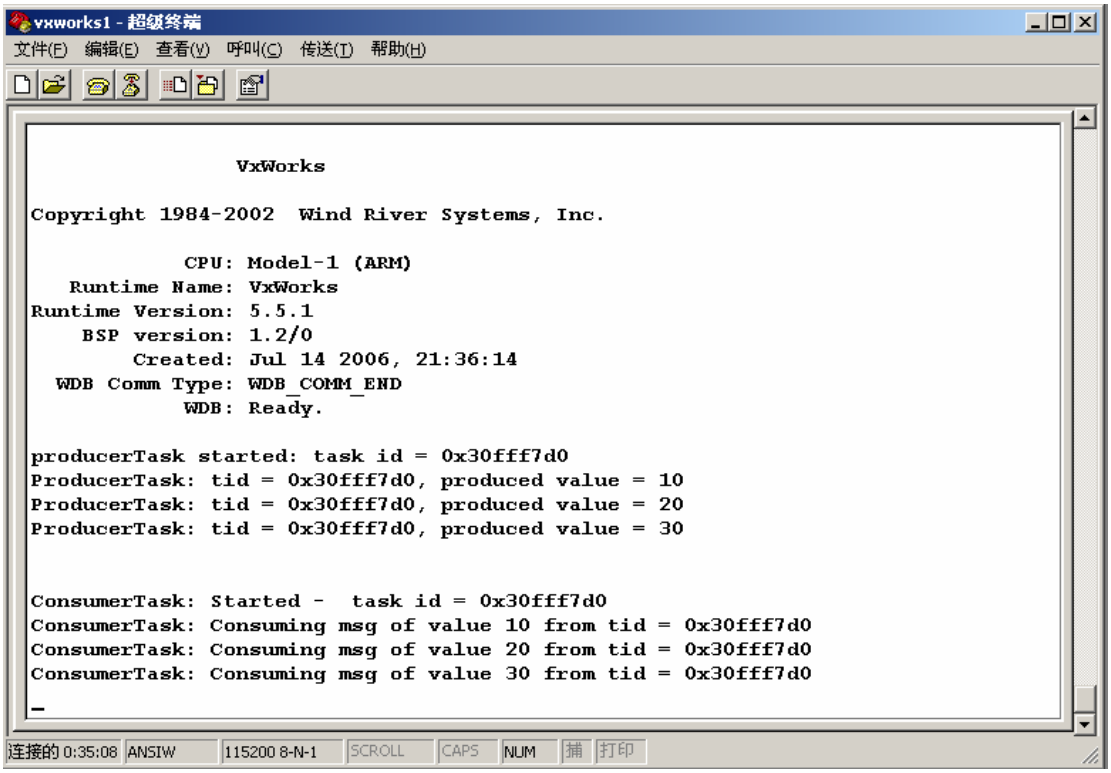
```
/* msgQDemo.c - 利用消息队列通信的例子*/


/* includes */

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

#define  CONSUMER_TASK_PRI          99   /* consumer task 优先级*/
#define  PRODUCER_TASK_PRI          98   /* producer task 优先级*/
#define  TASK_STACK_SIZE          5000   /*任务栈大小 */

struct msg {                             /* 传递的消息结构体 */
        int tid;                              /* 任务号*/
        int value;                          /* 传递的值 */
        };

LOCAL MSG_Q_ID msgQId;                   /* 消息队列号*/
LOCAL int numMsg = 3;                    /*队列中 最大消息个数*/
LOCAL BOOL notDone;                      /* 完成标志*/



/* function prototypes */

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


/*******************************************************************************
 *  msgQDemo - 利用消息队列通信的主程序
 *
 *  DESCRIPTION
 *  在 producerTask 和 consumerTask 之间创建一个消息队列,
 *  producerTask 通过消息队列发送消息,
 *  consumerTask 通过消息队列接收消息.
 *  当 consumerTask 接收完所有消息,关闭消息队列
 *
 *  RETURNS: OK or ERROR
 *
```

```
 *  EXAMPLE
 *
 *  -> sp msgQDemo
 *
 */

STATUS msgQDemo()
    {
    notDone = TRUE;  /* initialize the global flag */

    /* 创建一个消息队列,先入先出类型*/
    if ((msgQId = msgQCreate (numMsg, sizeof (struct msg), MSG_Q_FIFO))
                                                                    ==
NULL)
        {
        perror ("Error in creating msgQ");
        return (ERROR);
        }

     /* 创建 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);
        }

    /* 创建 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);
        }

    /* 轮询,等待 consumerTask 接收完所有消息*/
    while (notDone)
        taskDelay (sysClkRateGet ());

     /*关闭消息队列*/
    if (msgQDelete (msgQId) == ERROR)
        {
```

```
        perror ("Error in deleting msgQ");
        return (ERROR);
        }

    return (OK);
    }


/*******************************************************************************
    *  producerTask - 通过消息队列发送消息
    *
    *  RETURNS: OK or ERROR
    *
    */

 LOCAL STATUS producerTask (void)
    {
    int count;
    int value;
    struct msg producedItem;            /* 发送缓存 */

    printf ("producerTask started: task id = %#x \n", taskIdSelf ());

    /*产生  numMsg 个消息并发送*/

    for (count = 1; count <= numMsg; count++)
        {
        value = count * 10;

        /* 给消息结构体赋值 */
        producedItem.tid = taskIdSelf ();       /*任务号*/
        producedItem.value = value;

        /* 发送消息*/
        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 - 通过消息队列接收消息
    *
    *  RETURNS: OK or ERROR
    *
    */

    LOCAL STATUS consumerTask (void)
    {
    int count;
    struct msg consumedItem;            /* 接收缓存 */

    printf ("\n\nConsumerTask: Started -  task id = %#x\n", taskIdSelf());

    /* 接收 numMsg 个消息*/
    for (count = 1; count <= numMsg; count++)
        {
        /* 接受消息 */
        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;    /*设置 notDone,通知主任务关闭消息队列*/
    return (OK);
    }
```

在 Target Shell 下运行 sp msgQDemo 执行。

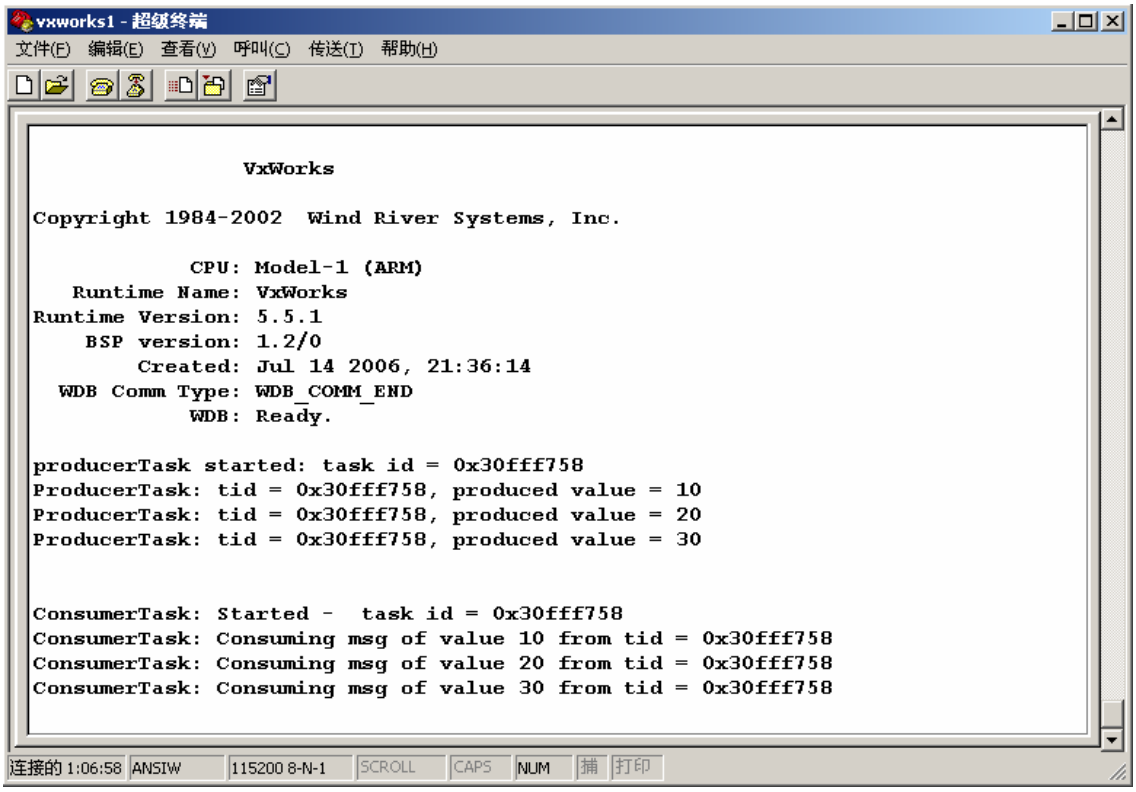


5.4.4.6 利用管道进行任务间通信

具体程序见 Multi-Tasking 文件夹下的 pipeDemo.c。它的流程和 3.4.5 基本一致，只不过任务间的通信机制变为管道。所以在这里不多作解释，具体可以参考代码和注释。

在 Target Shell 或者 WindSh 下运行 sp pipeDemo 执行。

管道的操作函数中，除了上面介绍的 **pipeDrvCreate** 以外，其他的操作和文件操作一样，可以使用 open/close/read/write 进行操作。



5.4.4.8 信号的使用

这个实验说明了如何利用 kill 给另一个任务发送信号，和如何利用 signal()为信号绑定处理函数。具体程序见 Multi-Tasking 文件夹下的 sigDemo.c 文件。主函数 sigDemo（）发起一个任务 syncTask 和一个看门狗定时器。syncTask 每隔一秒向终端打印"Hello",并设定了用户自定义信号 SIGUSR1 处理函数,看门狗定时器模拟外部异常事件,每发生一次异常向 syncTask 发送用户自定义信号 SIGUSR1,调用处理函数，信号处理函数运行完后,转回到 syncTask 任务的上下文继续执行。

```
/*sigDemo.c - 信号处理的例子*/

/*include*/
#include "vxworks.h"
#include "stdio.h"
#include "sigLib.h"
#include "taskLib.h"
#include "wdLib.h"
/* * DESCRIPTION
  * 发起一个任务 syncTask 每隔一秒向终端打印"Hello",
  * 并设定了用户自定义信号 SIGUSR1 处理函数,
  * 利用 watchdog 定时器模拟外部异常事件,每发生一次异常
  * 向 syncTask 发送用户自定义信号 SIGUSR1,调用处理函数
```

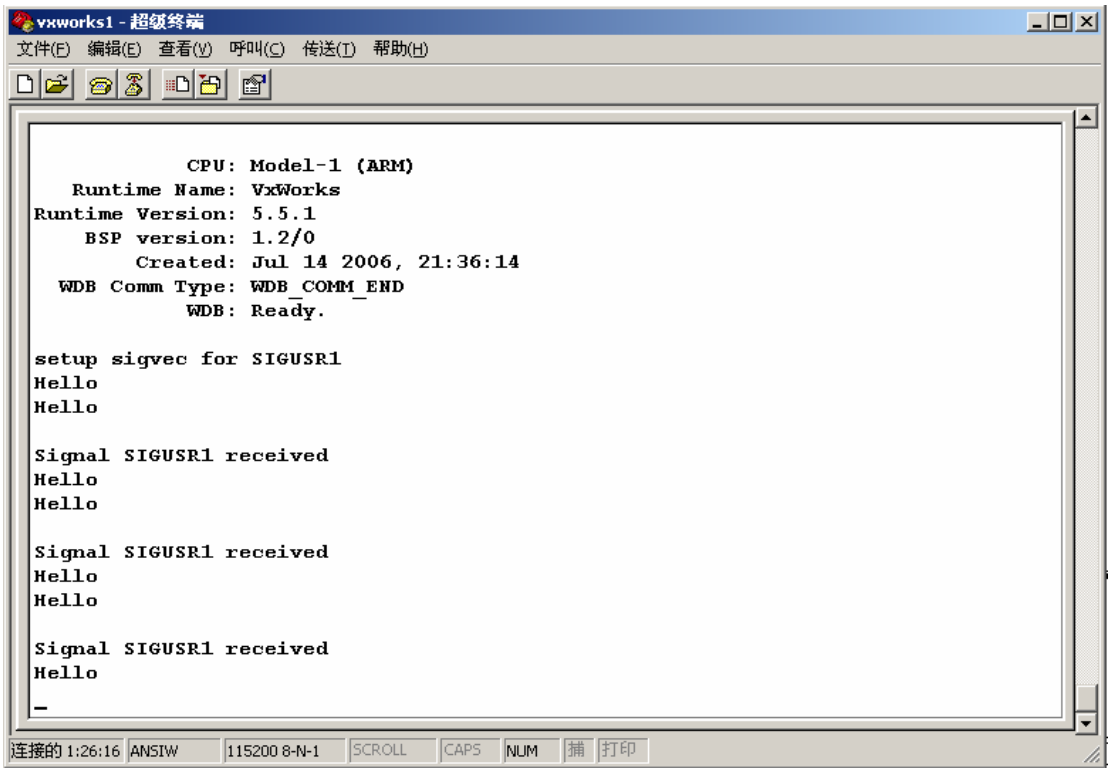
```
 * 信号处理函数运行完后,转回到 syncTask 任务的上下文继续执行
 * EXAMPLE
 *  -> sp sigDemo
 */

/* defines */
#define TASK_PRIORITY               101
#define TASK_STACK_SIZE                 5000
#define TIME_BETWEEN_INTERRUPTS  sysClkRateGet()*2   /* 2 秒 */
#define NUM_OF_GIVES                3

/* globals */
LOCAL WDOG_ID wdId = NULL;              /* 看门狗定时器 ID */
LOCAL int syncTaskTid = 0;                      /*syncTask 的 tid */
LOCAL int numToGive = NUM_OF_GIVES;     /* 信号被发送的次数 */

/* forward declaratiuon */
LOCAL void syncISR(int);
LOCAL void cleanUp ();
LOCAL void syncTask ();

/*******************************************************************************
 * sigDemo - 主函数. 创建任务 syncTaskTid,并创建和启动 watchdog 定时器
 *   模拟外部异常事件
 *  RETURNS: OK or ERROR
 */

STATUS sigDemo (void)
    {

    /*创建一个看门狗定时器,模拟外部中断*/
    if ((wdId = wdCreate()) == NULL)
        {
        perror ("wdCreate");
        cleanUp ();
        return (ERROR);
        }

    /*创建任务*/
    if    ((syncTaskTid    =    taskSpawn    ("tsyncTask",    TASK_PRIORITY,
0,TASK_STACK_SIZE,(FUNCPTR) syncTask,  0,0,0,0,0,0,0,0,0,0)) == ERROR)
        {
        perror ("taskSpawn");
        cleanUp();
```

```
            return (ERROR);
            }

    /* 启动开门狗定时器 */
    if (wdStart (wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, numToGive)
                    == ERROR)
        {
        perror ("wdStart");
        cleanUp ();
        return (ERROR);
        }

    /* 延时,然后结束演示*/
    taskDelay (sysClkRateGet()+TIME_BETWEEN_INTERRUPTS* numToGive);

    cleanUp();
    return (OK);
    }


/*********************************************************************
 * sigHandler -- 信号处理函数
 *
 */

LOCAL  void sigHandler
      (
      int sig,      /* signal 值*/
      int code,     /* 附加值*/
      SIGCONTEXT *sigContext /* 任务上下文 */
      )
 {

      switch (sig)
      {
          case SIGUSR1:
          printErr ("\nSignal SIGUSR1 received\n");
          break;

            default:
           printErr ("\nOther Signal received\n");
          break;
        }

```

```
    }

/*******************************************************************************
 *  syncTask -运行的任务, 响应异常任务发出的信号
 */

LOCAL  void syncTask (void)
    {
     int val = 0;

     /*将信号处理函数与 SIGUSR1 信号绑定*/
    signal (SIGUSR1, (VOIDFUNCPTR)sigHandler);
    printf ("setup sigvec for SIGUSR1\n");

    /*每隔一秒在终端上打印"Hello"*/
    FOREVER
        {

           printf("Hello\n");
           taskDelay (sysClkRateGet());
       }

    }

/*******************************************************************************
 * syncISR - 模拟的外部异常事件
 */
LOCAL  void syncISR
    (
    int times
    )
    {
       /*向 syncTaskId 发送信号 SIGUSR1*/
        kill(syncTaskTid,SIGUSR1);
       times--;
       if (times > 0)
           wdStart (wdId, TIME_BETWEEN_INTERRUPTS, (FUNCPTR) syncISR, times);

    }

/*******************************************************************************
 * cleanUP - 删除任务和定时器
 */
LOCAL  void cleanUp ()
```

```
    {
    if (syncTaskTid)
        taskDelete (syncTaskTid);
    if (wdId)
        wdDelete (wdId);

    }
```

在 Target shell 或者 WindSh 下运行 sp sigDemo:



5.5 中断

除了任务外，中断处理是另外一个执行线程单元。中断处理是整个运行系统中优先级最高的，可以抢占任何任务级代码运行。某些 CPU 还支持中断分级，高级中断可以抢占低级中断运行。中断机制是多任务环境运行的基础，是系统实时性的保证。

5.5.1 硬件中断

硬件中断处理是实时系统的重要组成部分，系统通常都是通过中断与外部世界进行交互的。为了尽可能快地响应中断，VxWorks 中的中断服务程序运行在一个不同于任何任务的上下文中。因此，中断的处理不涉及到任务上下文的切换，所有中断服务程序必须共享一个单独的堆栈。由于这个原因，中断服务程序(ISR)的编写有以下限制：

1) ISR 中不能有任何可能直接或间接导致阻塞的代码。例如，ISR 不能取信号量。
2) 内存操作函数 malloc()和 free()的内部实现都涌到了信号量，因此这两个函数不能由 ISR 直接或间接引用。
3) 在 ISR 不要进行通过 VxWorks 驱动程序的 I/O 操作，因为大多数 I/O 设备驱动程序将会阻塞的等待设备的调用者。一个特例是管道设备，其设计允许由 ISR 进行写操作。同时，为了给中断服务程序提供输出功能，VxWorks 提供了 logLib 库，这个库中的函数允许系统任务打印文本信息，它是终端服务程序中最常用的输出手段。如 logMsg()。

下表简要列出了可以在 ISR 中所调用的函数：

库	函 数
bLib	所有函数
errnoLib	errnoGet(), errnoSet()
fppArchLib	FppSave(), fppRestore()
intLib	intContext(), intCount(), intVecSet(), intVecGet()
intArchLib	IntLock(), intUnlock()
logLib	Logmsg()
lstLib	除了 lstFree()的所有函数
mathALib	如果使用了 fppSave()/fppRestore(),就可以使用全部函数
msgQLib	MsgQSend
pipeDrv	Write
rngLib	除了 rngCreate()和 rngDelete()都可以使用
selectLib	selWakeup(), selWakeupAll()
semLib	semGive()(不能用于互斥信号量),semFlush
sigLib	kill()
taskLib	taskSuspend(), taskResume(), taskePrioritySet(), taskPriorityGet(), taskIdVerify(), taskIdDefault(), taskIsReady(), taskIsSuspended(), taskTcb()
tickLib	tickAnnounce(), tickSet(), tickGet()
tyLib	tyIRd(), tyITx()
vxLib	vxTas(), vxMemProbe()
wdLib	WdStart(), wdCancel()

中断相关的函数：

intConnect	
目标	为一个中断指定中断服务程序
头文件	#include “intArchLib.h”
函数原型	STATUS result = intConnect(VOIDFUNCPTR* vector, VOIDFUNCPTR* routine, int parameter)
参数	vector 中断向量 routine 中断服务程序 parameter 传入中断服务程序的参数
返回值	ERROR 错误 OK 成功

5.5.2 看门狗定时器（Watchdog）

VxWorks 为定时功能提供了看门狗定时器(watchdog)机制。这个机制允许在一定时间延迟之后执行一段特定的函数代码。在系统内部，watchdog 是系统时钟中断处理程序的延伸，是作为系统时钟中断的一部分来维护的。

wdLib 库中提供如下函数接口供用户使用：

wdCreate	
目标	创建并初始化 Watchdog 定时器，但未启动
头文件	#include “wdLib.h”
函数原型	WDOG_ID wdId = wdCreate(void)
参数	
返回值	ERROR 错误 wdId Watchdog 定时器标志符

例如：

```
/* 创建一个看门狗定时器 */
if ((wdId = wdCreate ()) == NULL)
    {
    perror ("cannot create watchdog");
    return (ERROR);
    }
```

wdDelete	
目标	停止并释放 Watchdog 定时器
头文件	#include “wdLib.h”
函数原型	STATUS nResult = wdDelete(WDOG_ID wdId)
参数	wdId Watchdog 定时器标志符
返回值	ERROR 错误 OK 成功

wdStart	
目标	启动一个已创建的 Watchdog 定时器
头文件	#include “wdLib.h”
函数原型	STATUS nResult = wdStart(WDOG_ID wdId, int delay, FUNCPTR pRoutine, int parameter)
参数	wdId Watchdog 定时器标志符 delay 延时时间，单位为 tick pRoutine 延时执行函数，即 Watchdog 定时器到时执行的函数 parameter 传递给 pRoutine 的参数
返回值	ERROR 错误 OK 成功

例如：

```
#define DEADLINE_TIME       5
/*启动看门狗定时器，定时 5S，并绑定一个处理函数*/
if ((wdStart (wdId, sysClkRateGet ()* DEADLINE_TIME,
            (FUNCPTR) wdHandler, 0)) == ERROR)
    {
    perror ("Error in starting watchdog timer");
    return (ERROR);
    }
```

注意：定时器由 WdStart 函数启动，每启动一次，都执行一次指定的函数 pRoutine，而不会多次执行。如果需要循环触发，需要延时执行函数 pRoutine 中用 wdStart 再次启动定时器。

WdCancel	
目标	停止正在运行的 Watchdog 定时器
头文件	#include “wdLib.h”
函数原型	STATUS nResult = wdCancel(WDOG_ID wdId)
参数	wdId　　Watchdog 定时器标志符
返回值	ERROR　错误 OK　　成功

5.5.3 实验

5.5.3.1 硬件中断

实验中采用触摸屏的中断，所以在程序中要先初始化触摸屏。本试验给出了中断服务程序的一个模板。其他的中断服务程序的流程都可以参考。

中断的设置流程如下：

第一步：初始化中断相关寄存器

第二步：连接中断服务程序

第三步：中断使能

中断服务程序的流程如下：

第一步：设置中断相关寄存器，屏蔽中断

第二步：进行中断的相关操作

第三步：设置中断相关寄存器，打开中断

```
#include "vxWorks.h"
#include "taskLib.h"
#include "intLib.h"
#include "iv.h"
#include "logLib.h"
```

```
#include "S3C2410.h"
#include "irqs.h"

/*复用功能管脚定义宏*/
#define nYPON 0x3
#define YMON 0x3
#define nXPON 0x3
#define XMON 0x3
/*ACDCON 宏*/
#define ECFLG_END 1                    // End of A/D conversion
#define PRSCEN_Enable 1          //prescaler Enable
#define PRSCVL_49 49                        //A/D converter in process
#define SEL_MUX 7                // xp
#define STDBM_NORMAL 0                //normal mode
#define STDBM_STANDBY 1          //Standby mode
#define READ_START 0                  //Disable start by read operation
#define ENABLE_START 0                    //No operation
/*ADC touch screen control register 宏*/
#define YM_SEN_Hi_Z 0                 //YM=Hi-Z
#define YM_SEN_GND 1                  //YM=GND
#define YP_SEN_External_voltage 0   //YP=External voltage
#define YP_SEN_AIN5 1                 //conntected with AIN5
#define XM_SEN_Hi_Z 0
#define XM_SEN_GND  1
#define XP_SEN_External_voltage 0
#define xp_SEN_AIN7 1
#define PULL_UP_ENABLE 0              //ENABLE
#define PULL_UP_DISABLE 1             //DISABLE
#define AUTO_PST_Normal 0             //Normal ADC conversion
#define AUTO_PST_AUTO   1             //Auto(sequential) x/y positioin conversion
mode
#define XY_PST 0                      //No operation mode
#define XY_PST_INT 0x3                //waiting for interrupt Mode

/*ADC start or interval delay register*/
#define DELAY
/*ADC conversion data0 register*/
#define UPDOWN_DOWN_0 0
#define UPDOWN_UP_0 1
#define AUTO_PST_sequencing_0 1
#define XY_PST_0 0
/*ADC conversion data1 register*/
#define UPDOWN_DOWN_1 0
#define UPDOWN_UP_1 1
```

```
#define AUTO_PST_sequencing_1 1
#define XY_PST_1 0

#define LCDWIDTH 320
#define LCDHEIGHT 240
int TchScr_Xmin=145,TchScr_Xmax=902,
    TchScr_Ymin=142,TchScr_Ymax=902;

 void TchScr_init()
{
    /*复用管脚功能定义*/
    GPGCON &= ~((0x03 << 30)|(0x03 << 28)|(0x03 << 26)|(0x03 << 24));
    GPGCON|=(nYPON<<30)|(YMON<<28)|(nXPON<<26)|(XMON<<24);
    /*set ACDCON*/
    ADCCON=(PRSCEN_Enable<<14)|(PRSCVL_49<<6)|(SEL_MUX<<3);
    /*ADC start or interval delay register*/
    ADCDLY=0xffff;
    /*set ADC touch screen control register*/
    ADCTSC = (0<<8)|(1<<7)|(1<<6)|(0<<5)|(1<<4)|(0<<3)|(0<<2)|(3);
}


void testIsr(void)
{
    /* 第一步：设置中断相关寄存器，屏蔽中断 */
    INTMSK |= 0x80000000;
    INTSUBMSK  |= 0x00000200;

    /* 第二步：进行中断的相关操作 */
    logMsg("testIsr run...\n", 1, 2, 3, 4, 5, 6);

    /* 第三步：设置中断相关寄存器，打开中断 */
    SRCPND |= 0x80000000;
    SUBSRCPND |= 0x00000200;
    INTPND |= 0x80000000;
    INTSUBMSK &= (~0x00000200);
    INTMSK &= (~0x80000000);
}
void intTest(void)
{
    TchScr_init(); /* 初始化触摸屏设置 */
    /* 第一步：初始化中断相关寄存器 */
    SRCPND |= 0x80000000;
    printf("SRCPND = 0x%x\n", SRCPND);
```

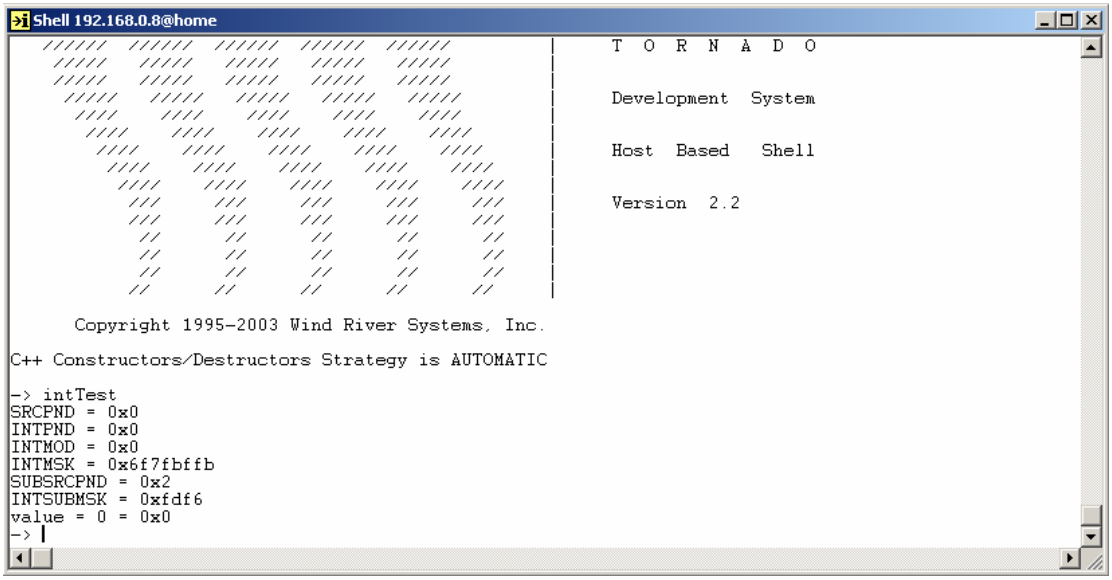
```
    INTPND |= 0x80000000;
    printf("INTPND = 0x%x\n", INTPND);
    INTMOD &= (~0x80000000);
    printf("INTMOD = 0x%x\n", INTMOD);
    INTMSK &= (~0x80000000);
    printf("INTMSK = 0x%x\n", INTMSK);
    SUBSRCPND |= 0x00000200;
    printf("SUBSRCPND = 0x%x\n", SUBSRCPND);
    INTSUBMSK &= (~0x00000200);
    printf("INTSUBMSK = 0x%x\n", INTSUBMSK);

    /* 第二步：连接中断服务程序 */
    intConnect((IRQ_TC), (FUNCPTR)testIsr, 0);

    /* 第三步：中断使能 */
    intEnable(IVEC_TO_INUM(IRQ_TC));
}
```

试验步骤：

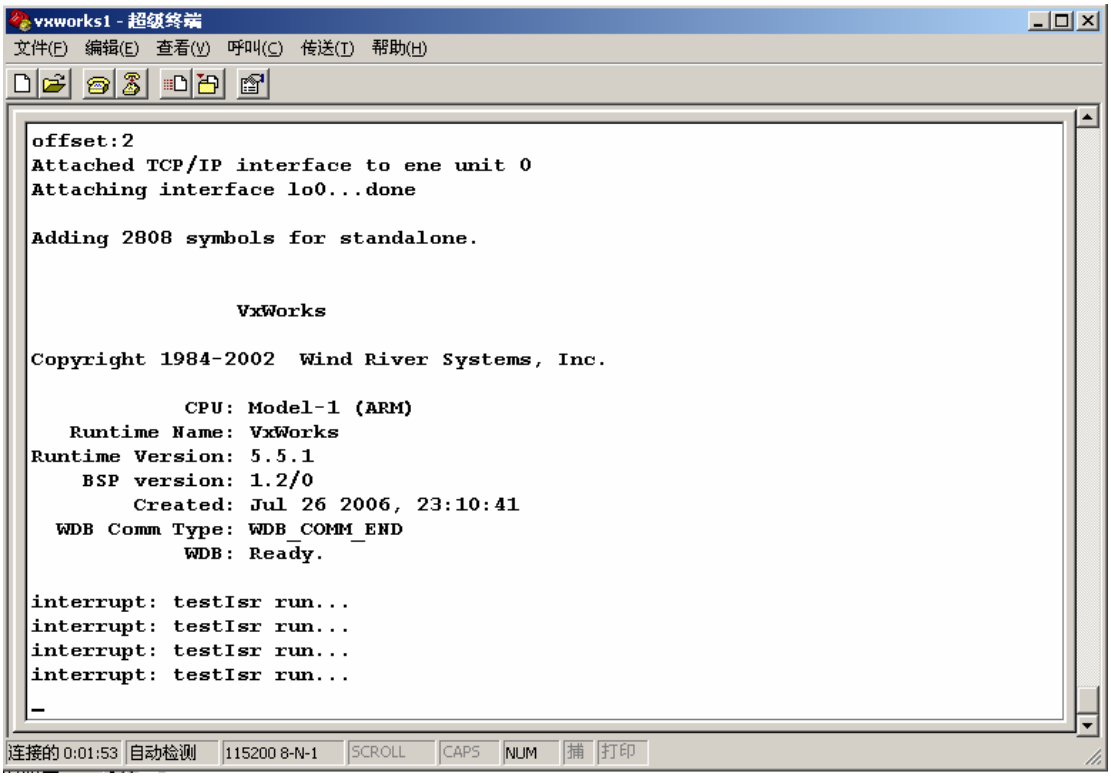
第一步：建立 downloadable 工程；添加 testIsr.c，并且把头文件 S3C2410.h、bitfield.h 和 irqs.h 放在和 testIsr.c 相同的目录下面；

第二步：编译并且下载程序；

第三步：在 Target Shell 或者 WinSh 下运行 intTest()；



点击触摸屏，会在串口看到中断服务程序执行。



5.5.3.2 看门狗定时器的使用

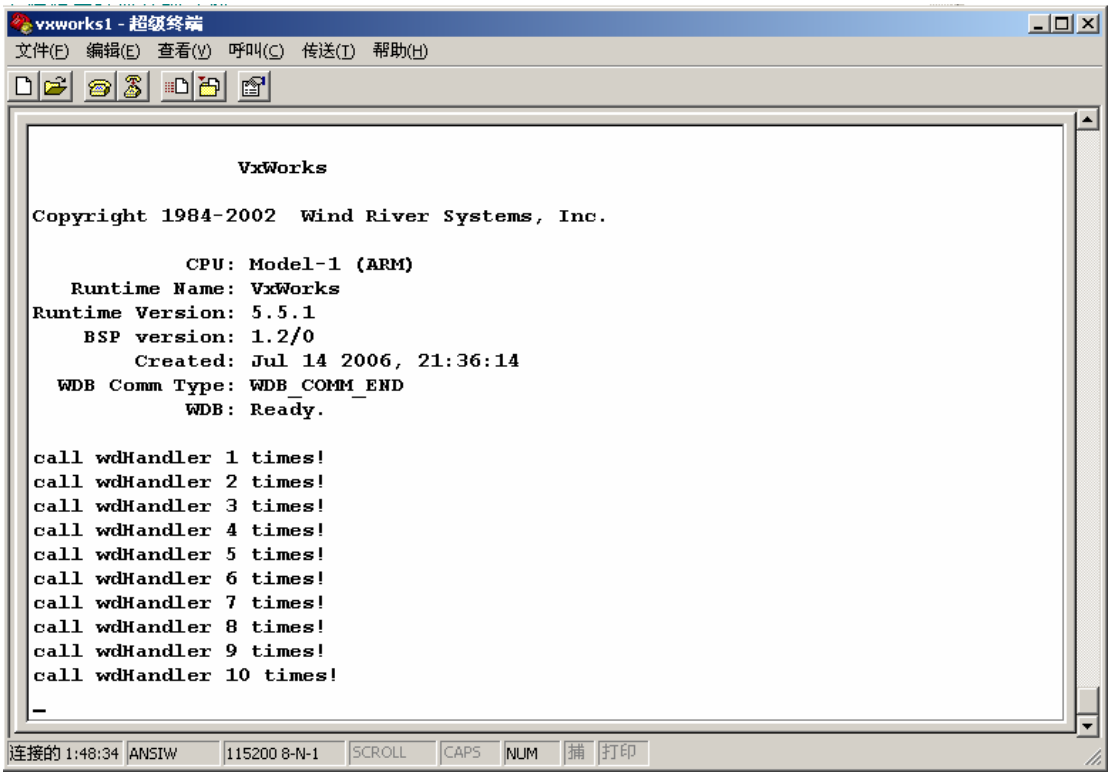
具体程序见 Multi-Tasking 文件夹下的 wtDogDemo.c。在 wdHandler 这个看门狗定时器处理程序中再次启动定时器，实现了看门狗定时器的循环触发。

```
/*WdDemo.c - 利用看门狗的例子*/

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

 /* defines */
#define TIMEOUT         10
#define DEADLINE_TIME 1

 LOCAL WDOG_ID   wdId;       /*看门狗定时器 ID*/
/*
    wdHandler - 看门狗定时器处理函数，
                        处理结束后再次启动定时器
                        循环处理
```

```
    RETURN : OK or ERROR
*/

STATUS wdHandler()
{
    static int count = 0;
    if(count < TIMEOUT )
        {
            count++;
            printf("call wdHandler %d times!\n",count);
            /*再次启动 wdId*/
            if ((wdStart (wdId, sysClkRateGet ()* DEADLINE_TIME,
                        (FUNCPTR) wdHandler, 0)) == ERROR)
                {
                perror ("Error in starting watchdog timer");
                return (ERROR);
                }

        }
    return (OK);
}

/*
    wdDemo   - 看门狗定时器主函数，
                        创建并发起一个看门狗定时器
    RETURN : OK or ERROR
*/

STATUS wdDemo()
{
     /* 创建一个看门狗定时器 */
     if ((wdId = wdCreate ()) == NULL)
         {
         perror ("cannot create watchdog");
         return (ERROR);
         }
    /*启动看门狗定时器，并绑定一个处理函数*/
    if ((wdStart (wdId, sysClkRateGet ()* DEADLINE_TIME,
                (FUNCPTR) wdHandler, 0)) == ERROR)
        {
        perror ("Error in starting watchdog timer");
        return (ERROR);
        }

```

```
        return (OK);

}
```

在 Target shell 下运行 sp wdDemo:



附录

1. 建立超级终端

1.1、运行 Windows 系统下的超级终端（HyperTerminal）应用程序，新建一个通信终端。如果要求输入区号、电话号码等信息请随意输入，出现如图 2-12 所示对话框时，为所建超级终端取名为 arm，可以为其选一个图标。单击“确定”按钮。

图 2-12 创建超级终端

2、在接下来的对话框中选择 ARM 开发平台实际连接的 PC 机串口（如 COM1），按确定后出现如图 2-13 所示的属性对话框，设置通信的格式和协议。这里波特率为 115200，数据位 8，无奇偶校验，停止位 1，无数据流控制。按确定完成设置。

图 2-13 设置串行口

3、完成新建超级终端的设置以后，可以选择超级终端文件菜单中的保存，将当前设置保存为一个特定超级终端到桌面上，以备后用。用串口线将 PC 机串口和平台 UART0 正确连接后，就可以在超级终端上看到程序输出的信息了。

2. 目标机服务器－Target Server

宿主机和目标机可以通过网络或者串口连接起来，这样宿主机上运行的开发工具 tornado 就可以与目标机上运行的程序进行通信，获取目标板上的信息。Tornado 下个工具与目标机系统的通信都是通过目标机服务器来实现的，在开发工程中使用 Tornado 工具，首先要配置目标机服务器。

1.启动目标机服务器

在启动目标机服务器之前先要对它进行配置，其操作过程是单击 Tools-> Target Server->Configure 菜单项，屏幕出现图 2.1 所示的目标机服务器配置对话框。

首先点击 new 新建一个 Target Server，在 Description 中输入 network，也可不添此项。Target Server 中也可不添，在 Target Server Properties 的下拉列表中选择 Back End，在 Available Back 中选择 wdbrpc，Target Name/IP address 中输入目标板的 ip 地址，目标板默认的 ip 地址是 192.168.0.8，这个 ip 地址要和 bsp 中的 config.h 中的 DEFAULT_BOOT_LINE 宏定义中设置的目标板的 ip 地址一致。如图 2.1 再从 Target Server Properties 的下拉列表中选择 Core File and Symbols,选则 file 选项，从右侧选择 vxworks 文件所在的路径，默认路径如下：

C:\Tornado2.2\target\proj\Project0\default_rom\vxWorks

单击此窗口下面的 Launch 按钮登陆。此时在 windows 的任务栏右侧出现一个靶形图标  ，单击图标出现图 2.2,表明 Target Server 配置成功，在菜单栏内的下拉列表中选择 Target Server 服务器 192.168.0.8@lf,这里的 ip 地址是目标板的 ip 地址，lf 是主机的名字，主机名可以在 windows 环境自行设置。至此就可以通过网络下载目标到目标板了

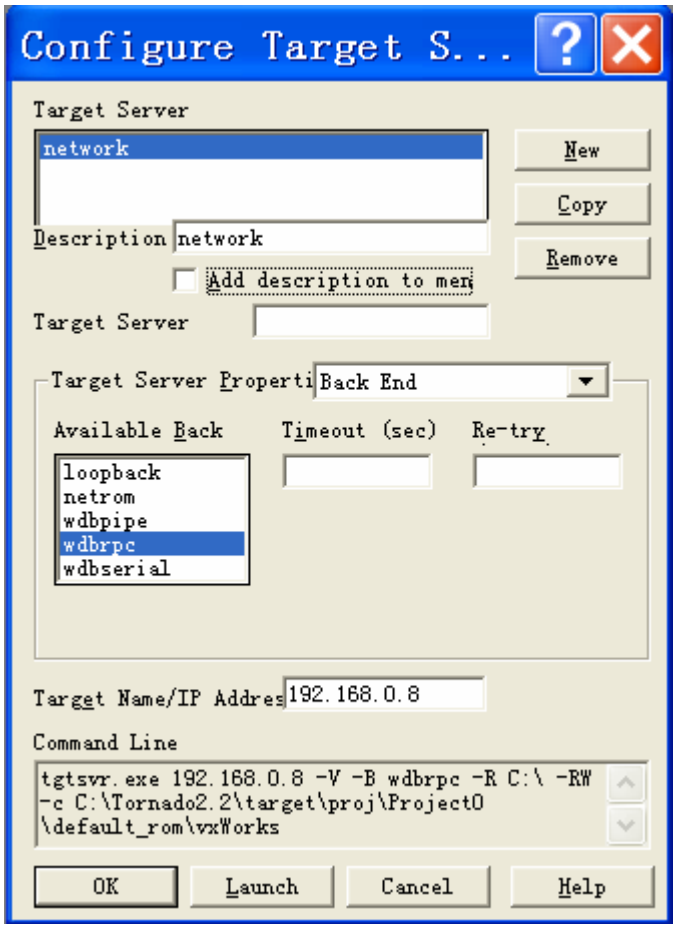


图 2.1

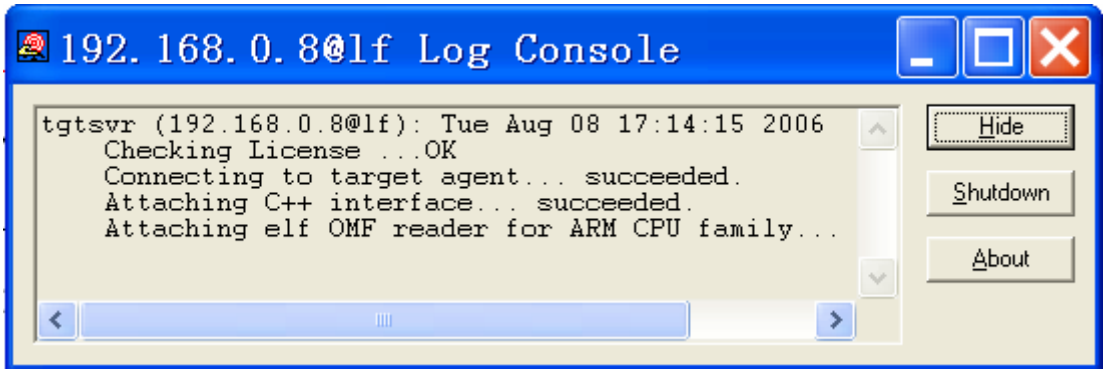


图 2.2

2. 关闭目标机服务器

双击 windows 任务栏的靶形图标，在弹出的窗口中点击 shutdown 关闭目标机服务器

3. FTP 服务器

这里只介绍在和前面章节做实验相关的 ftp 的配置方法，关于 ftp 更多的说明参见 ftpLib 和 ftpdLib，创建 FTP 服务器的方法很多，如果宿主机的操作系统是 windows NT，

windows 2000 的服务器版本，可以利用操作系统自身的 FTP 服务，如果是 windows95，windows98 或者是其他没有 FTP 服务的操作系统，则可以采用 VxWorks 自带的 WFTPD 程序来提供 FTP 服务。下面以 vxWorks 自带的 FTP 为例说明配置方法。

1. 启动 WFTPD，开始->所有程序->Tornado2.2->FTP Server,启动以后如图 3.1。

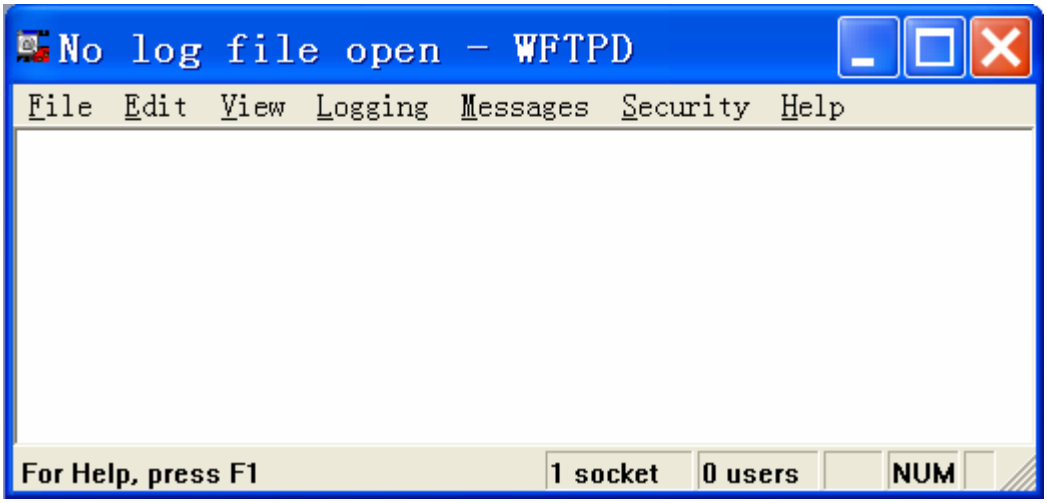


图 3.1

2. 点击 Security 选择 Users/rights 出现如图 3.2,点击 Uew User 创建新用户，用户名为 my2410 点击 ok 确定，不设密码。在 Home Directory 中输入将从 ftp 下载的文件所在目录。在 ftp 实验中使用的目录是 C:\Tornado2.2。在 Rights 中可设置操作权限，这里只用读的权限即可如图 3.3

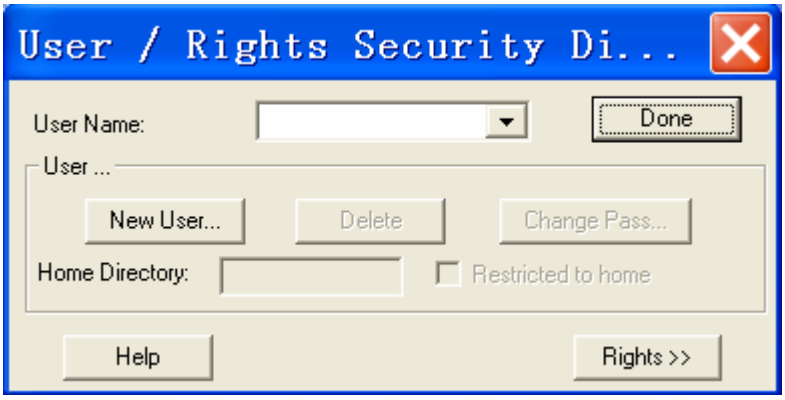


图 3.2

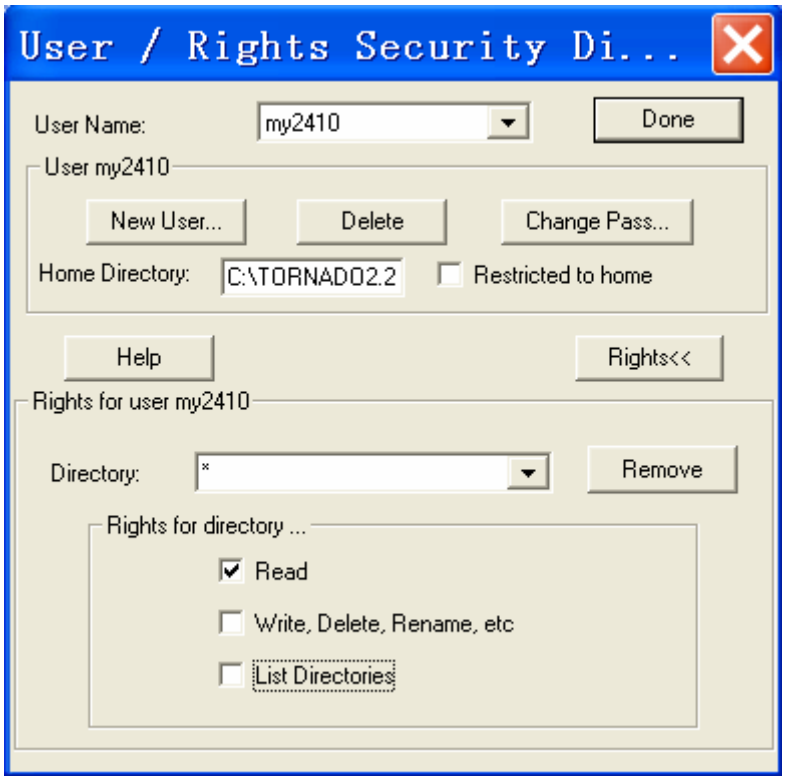


图 3.3

3.关闭 WFTPD，作完实验以后可以直接点击 WFTPD 主窗口右上角的关闭窗口按钮关闭 WFTPD 服务，或者单击 WFTPD 主窗口中的 File->Exit 菜单项关闭 WFTPD 服务。

4 WindSH

WindSh 是运行在宿主机上的命令行解释器，它可以解释用户在 WindSh 主窗口中命令提示符下输入的命令，并实现相关的功能，如下载应用程序模块、控制程序的运行、调用 VxWorks 操作系统的系统服务、调用应用程序模块的子程序、实时获取目标系统的运行信息、解释 C 语言表达式、解释 C++语言表达式。解释 Tcl 语言表达式等。

WindSh 的命令很多，功能也很强大，通过这些命令可以帮助开发人员交互式地探测目标系统的运行情况，如果能够灵活运用，就可以大大提高开发和调试的效率。

4.1 启动和关闭 WindSh

4.1.1.启动 WindSh

可以通过单击 Tornado 的 Tools->Shell 菜单项或者单击工具栏中的  按钮来启动 WindSh。

4.1.2.中断 WindSh 命令

如果 WindSh 执行了一个命令后没有响应，这时候可以键入 Ctrl+Break 或者 Crtl+C 来终止 WindSh 命令的执行，返回到命令提示符状态下。

4.1.3.关闭 WindSh

关闭 WindSh 的方法很多，可以在 WindSh 主窗口中输入 exit 或者 quit 命令；也可以键入 Ctrl+D；当然还可以通过鼠标单击窗口右上角的关闭按钮来关闭 WindSh。

4.2 WindSh 的使用

在 WindSh 主窗口中可以执行很多的内部命令。因为是内部命令，而不是对目标板系统函数的调用，所以的目标系统的影响很小。根据命令实现的功能一般可以分为：与任务管理相关的命令、用来获取任务信息的命令、与系统相关的命令、与网络相关的命令等。

4.2.1 任务管理命令

命令名	功能描述
sp()	以默认的参数创建一个任务
sps()	创建一个任务并挂起这个任务
tr()	恢复一个挂起的任务
ts()	挂起一个任务
td()	删除一个任务
period()	创建一个任务，这个任务周期性的调用一个函数
repeat()	创建一个任务，这个任务重复不断地调用一个函数
taskIdDefault()	设置或返回默认任务的 ID 号

4.2.2 任务信息命令

命令名	功能描述
i()	显示系统信息（系统中运行的任务列表，各个任务的状态，如 pc、sp、tcb 地址等）。为了节省内存，这个命令多次查询目标系统，每次获得部分信息，这样可能导致显示的信息前后不一致
iStrict()	显示系统信息，与命令 i 功能相同。但是这个命令只查询目标系统一次，这样要花费更多的内存，但是可以获得精确信息
ti()	显示任务信息，这个命令显示所有包含在任务 TCB 表中的信息
w()	显示每个任务的挂起信息
tw()	显示导致任务挂起的对象的信息
checkStack()	显示栈的使用情况

tt()	显示任务的堆栈
taskIdFigure()	显示任务的 ID 号

4.2.3 系统命令

命令名	功能描述
devs()	显示目标系统上的所有设备
lkup()	在符号表中查询一个符号
lkAddr()	查询特定值对应的符号
d()	显示目标系统内存
l()	反汇编一段程序
pirntErrno()	描述最近一次错误的状态值
version()	显示 VxWorks 版本信息
cd()	改变宿主机工作目录
ls()	显示宿主机工作目录的文件名
pwd()	显示宿主机当前工作的绝对路径
help()	显示 WindSh 命令的帮助列表
h()	显示历史命令
shellHistory()	设置或者显示 WindSh 历史命令
shellPromptSet()	改变 C 语言解释器命令提示符
pirntLogo()	显示 WindSh 的徽标

4.2.4 网络状态显示

命令名	功能描述
hostShow()	显示宿主机列表
icmpstatShow()	显示所有 ICMP 协议的信息
ifShow()	显示网络接口
inetstatShow()	显示所有的活动连接
ipstatShow()	显示所有 IP 协议的信息
routestatShow()	显示所有路由协议信息
tcpstatShow()	显示所有 TCP 协议的信息
tftpInfoShow()	获得 TFTP 状态信息
udpstatShow()	显示所有 UDP 协议的信息

4.3 运行目标机程序

用户可以从 WindSh 运行目标机系统上所有的 VxWorks 系统函数和应用程序函数，通过这种方式来测试和调试应用程序，举例如下

4.3.1VxWorks 系统函数的调用

VxWokrs 的系统函数 taskSpawn()的功能是创建一个任务，下面将在 WindSh 中调用 taskSpawn()创建一个名为 myTask 的任务，假定已有一个用户函数 myFunction(int,int)，创建过程如下所示：

—>taskSpawn(“myTask”,100,0,1024,myFunction,800,300)

Value= …

这个命令运行的结果是创建了一个名为myTask的任务,它的优先级为100,栈的大小是1024，任务的入口是 myFunction 函数，800 和 300 是函数 myFunction 的参数。

4.3.1.1 应用程序函数的调用

假定应用程序中包含两个函数，名字分别是 myFunc01()和 myFunc02(),如下所示调用，假定函数 myFunc01 有一个参数是 parameter01,函数 myFunc02 没有参数。

a.->myFunc01(parameter01)

value=…

b.->myfunc01()

4.3.1.2.重新启动目标机系统

有时候目标机系统的状态不受控制，需要重新启动，可以在 WindSh 中输入 reboot 命令，如

->reboot

Rebooting…