Tornado® 2.2

USER'S GUIDE

UNIX VERSION

WindRiver®

Corporate Headquarters
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): 800/545-WIND
telephone: 510/748-4100
facsimile: 510/749-2010

For additional contact information, please visit the Wind River URL:

http://www.windriver.com

For information on how to contact Customer Support, please visit the following URL:

http://www.windriver.com/support

Tornado User's Guide, 2.2

30 Jul 02
Part #: DOC-14208-2D-01

Contents

1 Overview

1.1 Introduction

Tornado is an integrated environment for software cross-development. It provides an efficient way to develop real-time and embedded applications with minimal intrusion on the target system. Tornado consists of the following elements:

- VxWorks, a high-performance real-time operating system.
- Application-building tools (compilers and associated programs).
- A development environment that facilitates managing and building projects, establishing and managing host-target communication, and running, debugging, and monitoring VxWorks applications.

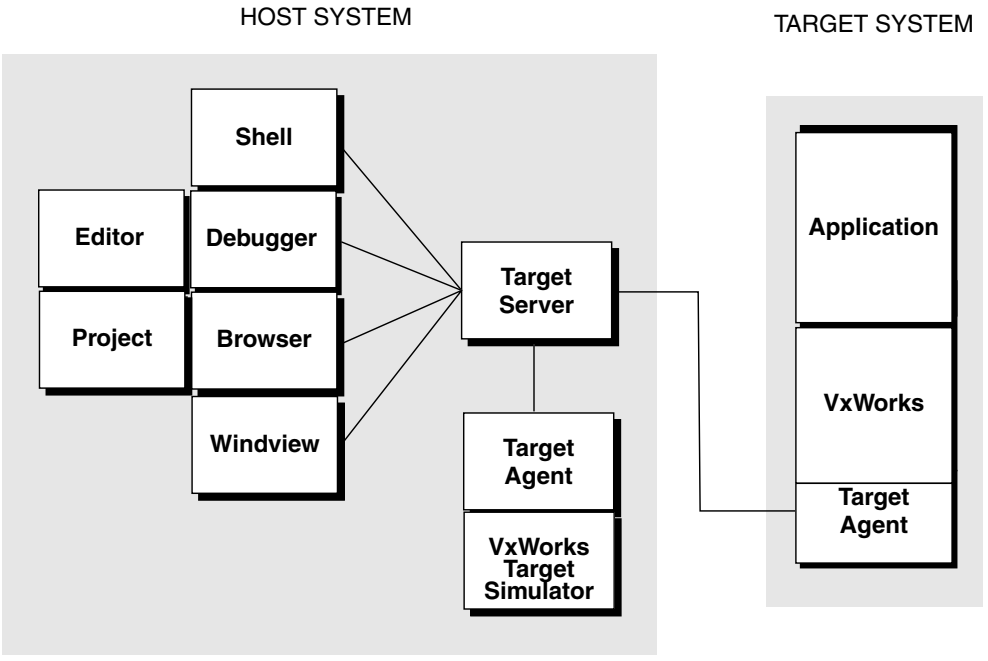
The Tornado interactive development tools include:

- The *launcher*, an integrated target-management utility.
- A project management facility.
- Integrated C and C++ compilers and **make**.
- The *browser*, a collection of visualization aids to monitor the target system.
- CrossWind, a graphically enhanced source-level debugger.
- WindSh, a C-language command shell that controls the target.
- An integrated version of the VxWorks target simulator, VxSim.
- An integrated version of the WindView logic analyzer for the target simulator.

The Tornado environment is designed to provide this full range of features regardless of whether the target is resource-rich or resource-constrained. Tornado facilities execute primarily on a host system, with shared access to a host-based dynamic linker and symbol table for a remote target system. Figure 1-1 illustrates the relationships between the principal interactive host components of Tornado

and the target system. Communication between the host tools and VxWorks is mediated by the target server and target agent.

Figure 1-1 **Tornado Development Environment**



The *run-time system* (often called simply the *run-time*) is the code that is intended for the final application, as distinguished from the complete Tornado cross-development environment. The run-time includes the real-time kernel, and typically also includes some selection of VxWorks library code as well as application-specific code. It does not usually include the target agent, although in some cases the target agent can be included to provide field debugging.

With Tornado, the cycle between developing an idea and observing its implementation is minimized. Fast incremental downloads of application code are linked dynamically with the VxWorks operating system and are thus available for symbolic interaction with minimal delay.

1.2 Cross-Development with Tornado

The Tornado cross-development environment ensures the smallest possible difference between the target system during development and the system after deployment. This is accomplished by segregating most development facilities on the host system, while providing minimally intrusive access to the target. The facilities of the run-time and the development environment are as independent of each other as possible, regardless of the scale of the target application. You can use the cross-development host to manage project files, to edit, compile, link, and store real-time code, and to configure the VxWorks operating system. Application modules in C or C++ are compiled with the Tornado cross-compiler. These application modules can draw on the VxWorks run-time libraries to accelerate application development. You can also run and debug real-time code on the target while under host-system control.

The hardware in a typical development environment includes one or more networked development host systems and one or more embedded target systems. A number of alternatives exist for connecting the target system to the host, but usually the connection is either an Ethernet or a serial link. If hardware or hardware-specific code is not initially available, the integrated VxSim target simulator can be used to begin application development.

A typical host development system is equipped with large amounts of RAM and disk space, backup media, printers, and other peripherals. In contrast, a typical target system has only the resources required by the real-time application, and perhaps some small amount of additional resources for testing and debugging.

A fundamental advantage of the Tornado environment is that the application modules do not need to be linked with the run-time system libraries or even with each other. Tornado loads the relocatable object modules directly, using the symbol tables in each module to resolve external symbol references dynamically. In Symbol table resolution is done by the target server (which executes on the host).

Tornado minimizes object-module sizes during development because there is no requirement to link the application fully. This shortens the development cycle because less data is downloaded, thus shortening the development cycle. Even partially completed modules can be downloaded for incremental testing and debugging. The host-resident Tornado shell and debugger can be used interactively to invoke and test either individual application routines or complete tasks.

Tornado maintains a complete host-resident symbol table for the target. This symbol table is incremental: the server incorporates symbols as it downloads each object module. You can examine variables, call subroutines, spawn tasks,

disassemble code in memory, set breakpoints, trace subroutine calls, and so on, all using the original symbolic names.

In addition, the Tornado development environment includes the CrossWind debugger, which allows developers to view and debug applications in the original source code. Setting breakpoints, single-stepping, examining structures, and so on, is all done at the source level, using a convenient graphical interface.

1.3 VxWorks Target Environment

The complete VxWorks operating-system environment is included in Tornado. This includes a multitasking kernel that uses an interrupt-driven, priority-based task scheduling algorithm. Run-time facilities include POSIX interfaces, intertask communication, extensive networking, file system support, and many other features.

Target-based tools analogous to some of the Tornado tools are included as well: a target-resident command shell, symbol table, and dynamic linker. In some situations the target-resident tools are appropriate, or even required, for a final application.

⚠ **CAUTION:** When you run the VxWorks target-based tools, avoid concurrent use of the corresponding tools that execute on the host. There is no technical restriction forbidding this, but an environment with—for example—two shells, each with its own symbol table, can be quite confusing. Most users choose either host-based tools or target-based tools, and seldom switch back and forth

In addition to the standard VxWorks offering, Tornado is compatible with the features provided by the optional component VxVMI. VxVMI provides the ability to make text segments and the exception vector table read-only, and includes a set of routines for developers to build their own virtual memory managers. When VxVMI is in use, Tornado's target-server loader/unloader takes account of issues such as page alignment and protection.

Tornado is also compatible with the VxWorks optional components VxMP and VxFusion. VxMP provides for synchronization of tasks on different CPUs over a back plane, while VxFusion allows that synchronization to take place over any kind of connection including Ethernet.

For detailed information on VxWorks and on its optional components, see the *VxWorks Programmer's Guide* and the *VxWorks Network Programmer's Guide*. For information on exactly what functions your architecture supports, see the appropriate *Architecture Supplement*.

1.4 Tornado Host Tools

Tornado integrates the various aspects of VxWorks programming into a single environment for developing and debugging VxWorks applications. Tornado allows developers to organize, write, and compile applications on the host system; and then download, run, and debug them on the target. This section summarizes the major features of Tornado tools.

Launcher

The launcher lets you start, manage, and monitor target servers, and connects the remaining interactive Tornado tools to the target servers of your choice. When you select a particular target server, the launcher shows information about the hardware and software environment on the target, as well as monitoring and reporting on what Tornado tools are currently attached to that target. You can reserve target servers for your own use with the launcher, or allow others to use them as well.

In many ways the launcher is the central Tornado control panel. Besides providing a convenient starting point to run the other tools, the launcher can also:

- Aid in the installation of additional Tornado components.
- Provide access to Wind River publications on the Internet.
- Help you prepare and transmit support requests to the customer support group at Wind River.

The launcher is described in *3. Launcher*.

Project Management

The Tornado project facility simplifies organizing, configuring, and building VxWorks applications. It includes graphical configuration of the build environment (including compiler flags), as well as graphical configuration of VxWorks (with dependency and size analysis). The project facility also provides for basic integration with common configuration management tools such as ClearCase.

The project facility is described in *4. Projects*.

Compiler

Tornado includes the GNU compiler for C and C++ programs, as well as a collection of supporting tools that provide a complete development tool chain:

- **cpp**, the C preprocessor
- **gcc**, the C and C++ compiler
- **make**, the program-building automation tool
- **ld**, the programmable static linker
- **as**, the portable assembler
- binary utilities

These tools are supported, commercial versions of the GNU tools originally developed by the Free Software Foundation (FSF). The Tornado project facility provides a GUI for the GNU tools that is powerful and easy to use.

For more information, see *4. Projects*, *GNU ToolKit User's Guide*, and *GNU Make User's Guide*.

In addition, the Diab compiler for C and C++, available as an optional product, is fully integrated with Tornado. For more information, see *4. Projects*, *5. Command-Line Configuration and Build*, and the *Diab C/C++ Compiler User's Guide* for your target architecture.

WindSh Command Shell

WindSh is a host-resident command shell that provides interactive access from the host to all run-time facilities. The shell can interpret and execute almost all C-language expressions. It supports C++, including *demangling* to allow developers to refer to symbols in the same form as used by the original C++ source code. The Tornado shell also includes a complete Tcl interpreter.

The shell can be used to call run-time system functions, call any application function, examine and set application variables, create new variables, examine and modify memory, and even perform general calculations with all C operators. The shell also provides the essential symbolic debugging capabilities, including breakpoints, single-stepping, a symbolic disassembler, and stack checking.

The shell interpreter maintains a command history and permits command-line editing. The shell can redirect standard input and standard output, including input and output to the virtual I/O channels supported by the target agent.

The shell is described in *7. Shell*.

CrossWind Debugger

The remote source-level debugger, CrossWind, is an extended version of the GNU source-level debugger (GDB). The most visible extension to GDB is a straightforward graphical interface. CrossWind also includes a comprehensive Tcl scripting interface that allows you to create sophisticated macros or extensions for your own debugging requirements. For maximum flexibility, the debugger console window synthesizes both the GDB command-line interface and the facilities of WindSh, the Tornado shell.

From your development host, you can use CrossWind to do the following:

- Spawn and debug tasks on the target system.
- Attach to already-running tasks, whether spawned from your application, from a shell, or from the debugger itself.
- Use breakpoints and other debugging features at either the application level or the system level.
- View your application code as C or C++ source, as assembly-level code, or in a mixed mode that shows both.

The debugger is described in *9. Debugger*. Also see the *GDB User's Guide*.

Browser

The Tornado browser is a system-object viewer, a graphical companion to the Tornado shell. The browser provides display facilities to monitor the state of the target system, including the following:

- Summaries of active tasks (classified as system tasks or application tasks).

- The state of particular tasks, including register usage, priority, and other attributes.
- Comparative CPU usage by the entire collection of tasks.
- Stack consumption by all tasks.
- Memory allocation.
- Summary of modules linked dynamically into the run-time system.
- Structure of any loaded object module.
- Operating-system objects such as semaphores, message queues, memory partitions, and watchdog timers.

The browser is described in *8. Browser*.

WindView Software Logic Analyzer

WindView is the Tornado logic analyzer for real-time software. It is a dynamic visualization tool that provides information about context switches, and the events that lead to them, as well as information about instrumented objects.

Tornado includes an integrated version of WindView designed solely for use with the VxSim target simulator. WindView is available as an optional product for all supported target architectures.

WindView is described in the *WindView User's Guide*.

VxSim Target Simulator

The VxSim target simulator is a port of VxWorks to the host system that simulates a target operating system. No target hardware is required. The target simulator facilitates learning Tornado usage and embedded systems development. More significantly, it provides an independent environment for developers to work on parts of applications that do not depend on hardware-specific code (BSPs) and target hardware.

Tornado includes an integrated version of the target simulator that runs as a single instance per user, without networking support. Optional networking products such as SNMP are not available for this version.

The VxSim full simulator is available as an optional product. It supports multiple-instance use, networking, and most other optional products.

See the *Tornado Getting Started Guide* for a introductory discussion of target simulator usage, and *4. Projects* for information about its use as a development tool.

1.5 Host-Target Interface

The elements of Tornado described in this section provide the link between the host and target development environments:

- The target agent is a scalable component of VxWorks that communicates with the target server on the host system.
- The target server connects Tornado tools such as the shell and debugger with the target agent.
- The Tornado registry provides access to target servers, and may run on any host on a network.

Target Agent

On the target, all Tornado tools are represented by the target agent. The target agent is a compact implementation of the core services necessary to respond to requests from the Tornado tools. The agent responds to requests transmitted by the target server, and replies with the results. These requests include memory transactions, notification services for breakpoints and other target events, virtual I/O support, and task control.

The agent synthesizes two modes of target control: *task mode* (addressing the target at application level) and *system mode* (system-wide control, including ISR debugging). The agent can execute in either mode and switches between them on demand.

The agent is independent of the run-time operating system, interfacing with run-time services indirectly so that it can take advantage of kernel features when they are present, but without requiring them. The agent's driver interface is also independent of the run-time, avoiding the VxWorks I/O system. Drivers for the agent are raw drivers that can operate in either a polling or an interrupt-driven mode. A polling driver is required to support system-level breakpoints.

Run-time independence means that the target agent can execute before the kernel is running. This feature is valuable for the early stages of porting VxWorks to a new target platform.

A key function of the agent is to service the requests of the host-resident object-module loader. If the agent is linked into the run-time and stored in ROM. The target server automatically initializes the symbol table from the host-resident image of the target run-time system as it starts. From this point on, all downloads are incremental in nature, greatly reducing download time.

The agent itself is scalable; you can choose what features to include or exclude. This permits the creation of final-production configurations that still allow field testing, even when very little memory can be dedicated to activities beyond the application's purpose.

NOTE: The target agent is not required. A target server can also connect to an ICE back end, which requires less target memory, but does not support task mode debugging.

Tornado Target Server

The target server runs on the host, and connects the Tornado tools to the target agent. There is one server for each target; all host tools access the target through this server, whose function is to satisfy the tool requests by breaking each request into the necessary transactions with the target agent. The target server manages the details of whatever connection method to the target is required, so that each tool need not be concerned with host-to-target transport mechanisms.

In some cases, the server passes a tool's service request directly to the target agent. In other cases, requests can be fulfilled entirely within the target server on the host. For example, when a target-memory read hits a memory region already cached in the target server, no actual host-to-target transaction is needed.

The target server also allocates target memory from a pool dedicated to the host tools, and manages the target's symbol table on the host. This permits the server to do most of the work of dynamic linking—address resolution—on the host system, before downloading a new module to the target.

A target server need not be on the same host as the Tornado tools, as long as the tools have network access to the host where the target server is running.

Target servers can be started from the Tornado launcher, from the UNIX command line, or from scripts. See *2.7 Connecting a Tornado Target Server*, p.56 for a discussion

of starting a server from the UNIX command line, and see *3.5 Managing Target Servers*, p.72 for details on using graphical facilities in the launcher. For reference information on target servers, see the **tgtsvr** entry in in Help>Manuals Contents>Tornado Reference/Tornado Tools.

Tornado Registry

Tornado provides a central target server registry that allows you to select a target server by a convenient name. The registry associates a target server's name with the network address needed to connect with that target server. You can see the registry indirectly through the list of available targets. The Tornado registry need not run on the same host as your tools, as long as it is accessible on the network.

To help keep server names unique over a network of interacting hosts, target-server names have the form *targetName@host*, where *targetName* is a target-server name selected by the user who launches a server (with the network name of the target as a default). The registry rejects registration attempts for names that are already in use.

It is recommended that a single registry be used at a development site, to allow access to all targets on the network. To ensure that the registry starts up automatically in the event of a server reboot, it should be invoked from a UNIX system initialization file. A registry should never be killed; without a registry, target servers cannot be named, and no Tornado tool can connect to a target.

For more information, see *2.2 Setting up the Tornado Registry*, p.19.

Virtual I/O

Virtual I/O is a service provided jointly by the target agent and target server. It consists of an arbitrary number of logical devices (on the VxWorks end) that convey application input or output through standard C-language I/O calls, using the same communication link as other agent-server transactions.

This mechanism allows developers to use standard C routines for I/O even in environments where the only communication channel is already in use to connect the target with the Tornado development tools.

From the point of view of a VxWorks application, a standard I/O channel is an ordinary character device with a name like **/vio/0**, **/vio/1**, and so on. It is managed using the same VxWorks calls that apply to other character devices, as described

in the *VxWorks Programmer's Guide: I/O System*. This is also the developer's point of view while working in the Tornado shell.

On the host side, virtual I/O is connected to the shell or to the target server console, which is a window on the host where the target server is running. See *Target-Server Configuration Options*, p.76 for information about how to configure a target server with a virtual console.

1.6 Customer Services

Wind River is committed to meeting the needs of its customers. As part of that commitment, Wind River provides a variety of services, including training courses and contact with customer support engineers, along with a Web site containing the latest advisories, FAQ lists, known problems lists, and other valuable information resources.

Customer Support

For customers holding a maintenance contract, Wind River offers direct contact with a staff of software engineers experienced in Wind River products. A full description of the Customer Support program is described in the *Customer Support User's Guide* available at the following Web site:

http://www.windriver.com/support

The *Customer Support User's Guide* describes the services that Customer Support can provide, including assistance with installation problems, product software, documentation, and service errors.

You can reach Customer Support using either of the following methods:

- **E-mail.** You can contact Wind River Customer Support by sending e-mail to **support@windriver.com**.
- **1-800-872-4977 (1-800-USA-4WRS) .** Within North America, you can contact Customer Support with a toll-free voice telephone call. For telephone access outside North America, see the Support Web site shown above.

For Customer Support contact information specific to your products, please visit the Support Web site.

WindSurf

Wind River Customer Services also provides WindSurf, an online support service available under the Support Web site. WindSurf offers basic services to all Wind River customers, including advisories, publications such as the *Customer Support User's Guide*, and a list of training courses and schedules. For maintenance contract holders, WindSurf also provides access to additional services, including known problems lists, available patches, answers to frequently asked questions, and demo code.

2

Setup and Startup

2.1 Introducing Tornado

This chapter describes how to set up your host and target systems, how to boot your target, and how to establish communications between the target and host. It assumes that you have already installed Tornado.

NOTE: For information about installing Tornado, as well as an introductory tutorial using the integrated VxWorks target simulator, see the *Tornado Getting Started Guide*.

You do not need much of this chapter if all you want to do is connect to a target that is already set up on your network. If this is the case, read *2.3 The Tornado Host Environment*, p.20 and then proceed to *2.6 Booting VxWorks*, p.46.

The process of setting up a new target has the following steps (described in detail in the remainder of this chapter). Some of these steps are only required once, when you begin using Tornado for the first time; some are required when you install a new target; and only the last two are repeated frequently.

Tornado Configuration (once only)

1. Make sure that there is a Tornado registry running at your site.
2. Make sure your host environment includes the right definitions, on the host system where you attach the target.

Target Configuration (once for each new target)

3. Modify your host network tables so that you can communicate with your target.
4. Create and install the VxWorks boot ROM (or equivalent) in your target.
5. Set up physical connections (serial, Ethernet) between your target and your host.
6. Define a Tornado target server to connect to the new target.

Normal Operation (repeat to re-initialize target during development)

7. Boot VxWorks on the target. (VxWorks includes a target agent, by default.)
8. Launch or restart a Tornado target server on the host.

NOTE: In general, this manual refers to Tornado directories and files with path names prefixed by *installDir*. Use the actual path name chosen on your system for Tornado installation.

Target Servers and Target Agents

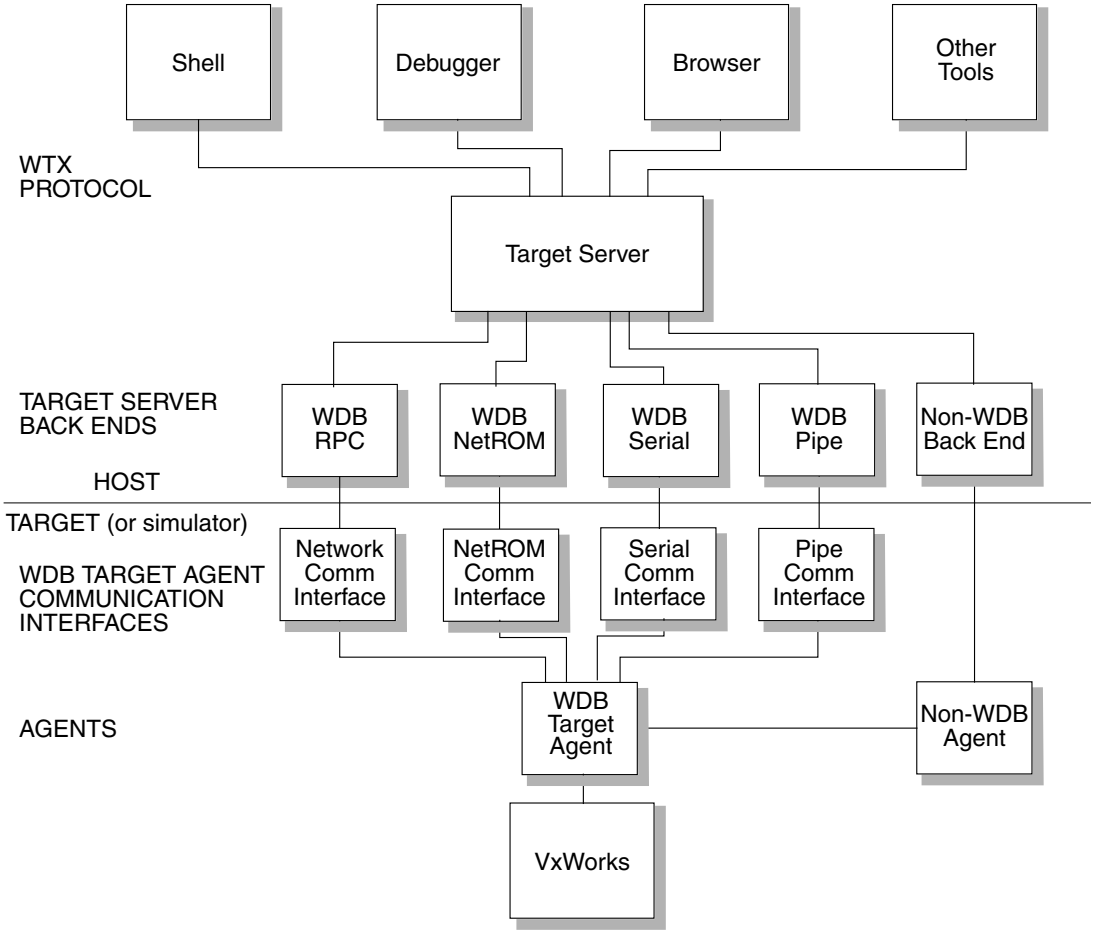
Tornado host tools such as the shell and the debugger communicate with the target system through a target server. A target server can be configured with a variety of back ends, which provide for various modes of communication with the target agent. On the target side, VxWorks can be configured and built with a variety of target agent communication interfaces.

Your choice of target server back end and target agent communication interface is based on the mode of communication that you establish between the host and target (network, serial, and so on). In any case, the target server *must* be configured with a back end that matches the target agent interface with which VxWorks has been configured and built. See Figure 2-1 for a detailed diagram of host-target communications.

Target Agent Modes

All of the standard back ends included with Tornado connect to the target through the WDB target agent. Thus, in order to understand the features of each back end, you must understand the modes in which the target agent can execute. These modes are called *task mode*, *system mode*, and *dual mode*.

Figure 2-1 **Tornado Host-Target Communication**



- In *task mode*, the agent runs as a VxWorks task. Debugging is performed on a per-task basis: you can isolate the task or tasks of interest without affecting the rest of the target system.
- In *system mode*, the agent runs externally from VxWorks, almost like a ROM monitor. This allows you to debug an application as if it and VxWorks were a single thread of execution. In this mode, when the target run-time encounters a breakpoint, VxWorks and the application are stopped and interrupts are locked. One of the biggest advantages of this mode is that you can single-step through ISRs; on the other hand, it is more difficult to manipulate individual

tasks. Another drawback is that this mode is more intrusive: it adds significant interrupt latency to the system, because the agent runs with interrupts locked when it takes control (for example, after a breakpoint).

- In *dual mode*, two agents are configured into the run-time simultaneously: a task- mode agent, and a system-mode agent. Only one of these agents is active at a time; switching between the two can be controlled from either the debugger (see *9.5 System-Mode Debugging*, p.362) or the shell (*7.2.7 Using the Shell for System Mode Debugging*, p.267). In order to support a system-mode agent, the target communication path must work in polled mode (because the external agent needs to communicate to the host even when the system is suspended). Thus, the choice of communication path can affect what debugging modes are available.

Communication Paths

The most common VxWorks communication path—both for server-agent communications during development, and for applications—is IP networking over Ethernet. That connection method provides a very high bandwidth, as well as all the advantages of a network connection.

Nevertheless, there are situations where you may wish to use a non-network connection, such as a serial line without general-purpose IP, or a NetROM connection. For example, if you have a memory-constrained application that does not require networking, you may wish to remove the VxWorks network code from the target system during development. Also, if you wish to perform system-mode debugging, you need a communication path that can work in polled mode. Older versions of VxWorks network interface drivers such at **netif** do not support polled operations and so cannot be used as a connection for system-mode debugging.

Note that the target-server back end connection is not always the same as the connection used to load the VxWorks image into target memory. For example, you can boot VxWorks over Ethernet, but use a serial line connection to perform system-mode debugging. You can also use a non-default method of getting the run-time system itself into your target board. For example, you might burn your VxWorks run-time system directly into target ROM, as described in *VxWorks Programmer's Guide: Configuration and Build*. Alternatively, you can use a ROM emulator such as NetROM to quickly download new VxWorks images to the target's ROM sockets. Another possibility is to boot from a disk locally attached to the target; see *VxWorks Programmer's Guide: Local File Systems*. You can also boot from a host disk over a serial connection using the Target Server File System; see *2.6.7 Booting a Target Without a Network*, p.55. Certain BSPs may provide other alternatives, such as flash memory. See the reference information for your BSP; Help>Manuals contents>BSP Reference in the Tornado Launcher.

2.2 Setting up the Tornado Registry

Before anyone at your site can use Tornado, someone must set up the *Tornado target server registry*, a daemon that keeps track of all available targets by name. The registry daemon must always run; otherwise Tornado tools cannot locate targets.

Usage of the Tornado registry is initially determined during the software installation process, based on the installer's choice of options for the registry. See the *Tornado Getting Started Guide* for information about installation.

Only one registry is required on your network, and it can run on any networked host. It is recommended that a development site use a single registry for the entire network; this provides maximum flexibility, allowing any Tornado user at the site to connect to any target.

If there is already a registry running at your site, you do not need the remainder of this section; just make sure you know which host the registry is running on, and proceed to *2.3 The Tornado Host Environment*, p.20.[1]

No privilege is required to start the registry, and it is not harmful to attempt to start a registry even if another is already running on the same host—the second daemon detects that it is not needed, and shuts itself down.

To start the registry daemon from a command line, execute **wtxregd** in the background. For example, on a Sun-4 running Solaris 2.x:

```
% installDir/host/sun4-solaris2/bin/wtxregd -V >/tmp/wtxregd.log &
```

This example uses the **-V** (verbose) option to collect diagnostic output in a logging file in **/tmp**. We recommend this practice, so that status information from the registry is available for troubleshooting.

To ensure that the registry remains available after a system restart, run **wtxregd** from a system startup file. For example, on Sun hosts, a suitable file is **/etc/rc2**. Insert lines like the following in the appropriate system startup file for your registry host. The example below uses conditionals to avoid halting system startup if **wtxregd** is not available due to some unusual circumstance such as a disk failure.

```
#
# Start up Tornado registry daemon
#
if [ -f /usr/wind/host/host-os/bin/wtxregd ]; then
    WIND_HOST_TYPE=host-os
```

1. Note that the same registry can serve both UNIX and Windows developers, as long as they share a local network. Either flavor of host may run the registry; see the *Tornado User's Guide (Windows version)* for instructions on setting up a registry on a Windows host.

```
        export WIND_HOST_TYPE
        WIND_BASE=/usr/wind
        export WIND_BASE
        /usr/wind/host/host-os/bin/wtxregd -V -d /var/tmp >/tmp/wtxregd.log &
        echo -n 'Tornado Registry started'
    fi
```

The Tornado tools locate the registry daemon through the environment variable **WIND_REGISTRY**; each Tornado user must set this variable to the name of whatever host on the network runs **wtxregd**.

In some cases, you may wish to segregate some collections of targets; to do this, run a separate registry daemon for each separate set of targets. Developers can then use the **WIND_REGISTRY** environment variable to select a registry host.

One of the more exotic applications of Tornado is to set this environment variable to a remote site; this allows the Tornado environment to execute remotely. Using a remote registry can bridge two separate buildings, or even enable concurrent development on both sides of the globe! As a support mechanism, it allows customer support engineers to wire themselves into a remote environment. This application often requires setting **WIND_REGISTRY** to a numeric Internet address, since the registry host may not be mapped by domain name. For example (using the C shell):

```
% setenv WIND_REGISTRY 127.0.0.1
```

If **WIND_REGISTRY** is not set at all, the Tornado tools look for the registry daemon on the local host.

You can query the registry daemon for information on currently-registered targets using the auxiliary program **wtxreg**. See the online *Tornado API Reference* for more information about both **wtxreg** and **wtxregd.**

2.3 The Tornado Host Environment

Tornado requires host-system environment variables for the following purposes:

- Tornado-specific environment variables reflect your development environment: what sort of host you are using, where Tornado was installed on your system, and where on your network to find the Tornado registry for development targets.

- Your shell search path must specify how to access Tornado tools.

2.3.1 Environment Variables for Tornado Components, p.21 discusses all Tornado environment variables.

NOTE: A shortcut to setting these variables is to source either **torVars.csh** or **torVars.sh**, which can be found in *installDir***/host/sun4-solaris2/bin**.

You can also set X Window System resources to allow the Tornado tools to benefit from color or grayscale displays; see *2.3.4 X Resource Settings*, p.23.

2.3.1 Environment Variables for Tornado Components

Specify the location of Tornado facilities by defining the following environment variables on your development host:

WIND_BASE
installation directory for Tornado, also shown as *installDir*

WIND_HOST_TYPE
name of host type, also shown as *hostType*

WIND_REGISTRY
registry host; see *2.2 Setting up the Tornado Registry*, p.19

WIND_HELP_SEPARATE_PROCESS
control whether to use an existing Netscape window or to launch a new window each time help is invoked; set to 0 to use the existing window (the default) or 1 to open a separate window.

PATH
shell search path; add *installDir***/host/sun4-solaris2/bin** directory

LD_LIBRARY_PATH
dynamic library search path; add Tornado *installDir***/host/sun4-solaris2/lib** directory

WIND_PROJ_BASE
WIND_SOURCE_BASE
These variables *are not* needed for most Tornado installations. If you are using sub-projects, you *may* need to use them. For more information, see Example 4-2.

Example Environment Setup Using C Shell

If you use the C shell, add lines like the following to your **.cshrc** to reflect your Tornado development environment. After you modify the file, be sure to source it and execute the **rehash** command.

The following example is for a Sun-4 host running Solaris 2.*x*, in a network whose Tornado registry is on host **mars**:

```
setenv WIND_BASE  /usr/wind
setenv WIND_HOST_TYPE sun4-solaris2
setenv WIND_REGISTRY mars
setenv PATH ${WIND_BASE}/host/sun4-solaris2/bin:${PATH}
setenv LD_LIBRARY_PATH ${WIND_BASE}/host/sun4-solaris2/lib:${LD_LIBRARY_PATH}
```

Example Environment Setup Using Bourne Shell (or Compatible)

If you are using the Bourne shell (or a compatible shell, such as the Korn shell or Bash), add lines like the following to your **.profile** to reflect your Tornado development environment. Be sure to source the file (using the "**.**" command) after you modify the file.

The following example is for an Solaris host in a network whose Tornado registry is on host **venus**:

```
WIND_BASE=/usr/wind; export WIND_BASE
WIND_HOST_TYPE=sun4-solaris2; export WIND_HOST_TYPE
WIND_REGISTRY=venus; export WIND_REGISTRY
PATH=$WIND_BASE/host/sun4-solaris2/bin:$PATH; export PATH
SHLIB_PATH=$WIND_BASE/host/sun4-solaris2/bin:$SHLIB_PATH; export SHLIB_PATH
```

2.3.2 Environment Variable For Solaris Hosts

If your development host runs Solaris 2, you must also modify the value of **LD_LIBRARY_PATH** to include the shared libraries in **/usr/dt/lib**, **/usr/openwin/lib**, and *installDir*/**host/sun4-solaris2/lib**.

If you use the C shell, include a line like the following in your **.cshrc**:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/usr/dt/lib:/usr/openwin/lib
```

If you use the Bourne shell (or a compatible shell), include lines like the following in your **.profile**:

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/dt/lib:/usr/openwin/lib
export LD_LIBRARY_PATH
```

2.3.3 Environment Variables for Convenience

Certain other environment variables, though they are not required for Tornado, can make the tools fit in better with your site or with your habits. The following environment variables are in this category:

EDITOR
When you request an activity in a Tornado tool that involves editing text files, the Tornado tools refer to this variable to determine what program to run. The default is **vi**, if **EDITOR** is not defined.

PRINTER
When any Tornado tool generates a printout at your request, it directs the printout to the printer name specified in this variable. The default is **lp**, if **PRINTER** is not defined.

2.3.4 X Resource Settings

Tornado has resource definitions to cover the range of X Window System displays. For better use of color or grayscale displays with Tornado, set **customization** resources in your X-resource initialization file (usually a file named **.Xdefaults** or **.Xresources** in your home directory). There are three possible values for these resources:

undefined
The general-purpose default; suitable for monochrome displays.

-color
For color displays.

-grayscale
For grayscale displays.

NOTE: X servers consult the resource-initialization file automatically only when they begin executing. To force your display to use new properties immediately, invoke the utility **xrdb**. For example, after modifying X resources in **.Xdefaults**, execute the following:

```
% xrdb -merge .Xdefaults
```

The following example (for a color display) shows **customization** settings specified explicitly for each of the Tornado tools:

```
Browser*customization: -color
CrossWind*customization: -color
Dialog*customization: -color
Launch*customization: -color
Tornado*customization: -color
```

Alternately, you can set **customization** globally for all tools that use this property. The following example does this for a grayscale display:

```
*customization: -grayscale
```

WARNING: If you set the **customization** property globally, it may affect applications from other vendors, as well as the Tornado tools.

For more information about X resources in Tornado, see *E. X Resources*.

2.4 Setting Up the Default Target Hardware

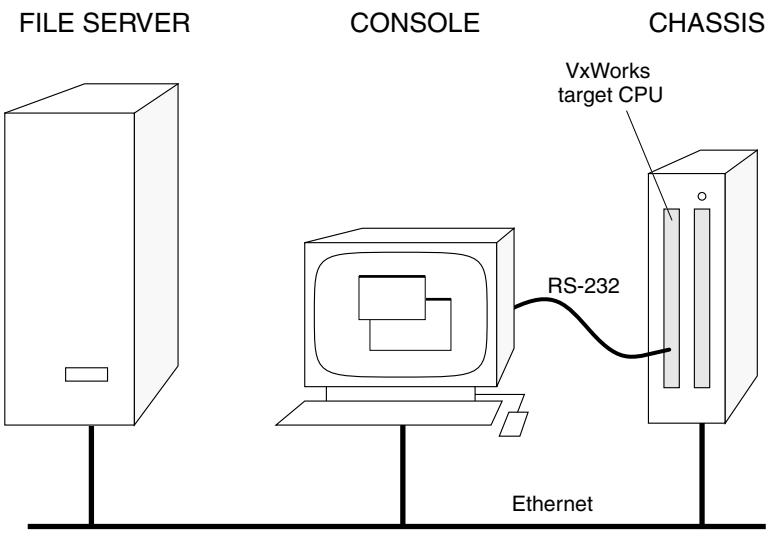
This section covers bringing up VxWorks on a hardware target with the relatively simple configuration matching the default software image. The *VxWorks Programmer's Guide* elaborates on more advanced options, such as gateways, NFS, multiprocessor target systems, and so on.

NOTE: Before you set up your target hardware, you may find it productive to use Tornado with the integrated target simulator. See the *Tornado Getting Started Guide* for a tutorial introduction.

2.4.1 Default Target Configuration

VxWorks is a flexible system that has been ported to many different hardware platforms.The default VxWorks run-time development configuration is shown in Figure 2-2. The pre-built VxWorks images shipped with your BSP include all the necessary components to run on this hardware configuration.

Figure 2-2 **A Resource-Rich Tornado Configuration**



The configuration in Figure 2-2 consists of the following:

Chassis
: A card cage with backplane and power supply.

Target CPU
: A single-board computer (target) where VxWorks is to run.

Console
: An ASCII terminal or a serial port on a workstation (required by the boot program for initial setup).

File Server
: A networked host where VxWorks binaries reside on disk; often the same workstation used as the console.

For more detailed information about your particular target Board Support Package (BSP), see Help>Manuals contents>BSP Reference in the Tornado Launcher.

2.4.2 Networking the Host and Target

IP networking over Ethernet is the most desirable way to connect a development target to your host, because of the high bandwidth it provides. This section

describes setting up simple IP connections to a target over Ethernet. To read about other communication strategies, see *2.5 Host-Target Communication Configuration*, p.31.

Before VxWorks can boot an executable image obtained from the host, the network software on the host must be correctly configured. There are three main tasks in configuring the host network software for VxWorks:

- Initializing the host network software.
- Establishing the VxWorks system name and network address on the host.
- Giving the VxWorks system appropriate access permissions on the host.

The following sections describe these procedures in more detail. Consult your system administrator before following these procedures: some procedures may require root permissions, and some UNIX systems may require slightly different procedures.

NOTE: If your UNIX system is running the Network Information Service (NIS), the "hosts" database is maintained by NIS facilities that are beyond the scope of this introduction. If you are running NIS, consult your UNIX system administration manuals.

Initializing the Host Network Software

Most UNIX systems automatically initialize the network subsystem and activate network processes in the startup files **/etc/rc2** and **/etc/rc.boot**. This typically includes configuring the network interface with the **ifconfig** command and starting various network daemons. Consult your UNIX system manuals if your UNIX startup procedure does not initialize the network.

Establishing the VxWorks System Name and Address

The UNIX host system maintains a file of the names and network addresses of systems accessible from the local system. This database is kept in the ASCII file **/etc/hosts**, which contains a line for each remote system. Each line consists of an Internet address and the name(s) of the system at that address. This file must have entries for your host UNIX system and the VxWorks target system.

For example, suppose your host system is called **mars** and has Internet address 90.0.0.1, and you want to name your VxWorks target **phobos** and assign it address 90.0.0.50. The file **/etc/hosts** must then contain the following lines:

```
90.0.0.1      mars
90.0.0.50     phobos
```

Giving VxWorks Access to the Host

The UNIX system restricts network access through remote login, remote command execution, and remote file access. This is done for a single user with the **.rhosts** file in that user's home directory, or globally with the **/etc/hosts.equiv** file.

The **.rhosts** file contains a list of system names that have access to your login. Thus, to allow a VxWorks system named **phobos** to log in with your user name and access files with your permissions, create a **.rhosts** file in your home directory containing the line:

```
phobos
```

The **/etc/hosts.equiv** file provides a less selective mechanism. Systems listed in this file are allowed login access to any user defined on the local system (except the super-user **root**). Thus, adding the VxWorks system name to **/etc/hosts.equiv** allows the VxWorks system to log in with any user name on the system.

Table 2-1 **Accessing Host from Target**

Target listed in:	Access
/etc/hosts.equiv	Any user can log in.
.rhosts file in user's home directory	Only this user can log in.

2.4.3 Configuring the Target Hardware

Configuring the target hardware may involve the following tasks:

- Setting up a boot mechanism.
- Jumpering the target CPU, and any auxiliary (for example, Ethernet) boards.
- Installing the boards in a chassis, or connecting a power supply.
- Connecting a serial cable.
- Connecting an Ethernet cable, if the target supports networking.

The following general procedures outline common situations. Select from them as appropriate to your particular target hardware. Refer also to the specific information in the target-information reference entry for your BSP; see Help>Manuals contents>BSP Reference in the Tornado Launcher.

Setting Up a Boot Mechanism

Tornado is shipped with the following VxWorks images.

Table 2-2 **VxWorks Images Shipped with Tornado**

Compiled with GNU	Compiled with Diab
vxWorks	**vxWorks**
vxWorks_rom	**vxWorks_rom**
vxWorks_romCompress	**vxWorks_romCompress**
vxWorks_romResident	**vxWorks_romResident**

In every case, you will need to create your own boot medium. Your board will require one of the following media:

Boot ROM
: Most BSPs include boot ROMs.

Floppy Disk
: Some BSPs for systems that include floppy drives use boot diskettes instead of a boot ROM. For example, the BSPs for PC386 or PC486 systems usually boot from diskette.

Flash Memory
: For boards that support flash memory, the BSP may be designed to write the boot program there. In such cases, an auxiliary program is supplied to write the boot program into flash memory is supplied by the board vendor.

Open Boot Prom
: Some targets use the "Open Boot Prom" protocol developed by Sun Microsystems. This is particularly common on (but not limited to) SPARC-based BSPs.

For specific information on a BSP's booting method, see Help>Manuals contents>BSP Reference in the Tornado Launcher. Instructions for

making a floppy disk for booting a Pentium target are in the *VxWorks for Pentium Architecture Supplement*.

You may also wish to replace a boot ROM, even if it is available, with a ROM emulator. This is particularly desirable if your target has no Ethernet capability, because the ROM emulator can be used to provide connectivity at near-Ethernet speeds. Tornado includes support for one such device, NetROM.[2] For information about how to use NetROM on your target, refer to *2.5.4 The NetROM ROM-Emulator Connection*, p.35. Contact the nearest Wind River office (see copyright page) for information about support for other ROM emulators.

For cases where boot ROMs are used to boot VxWorks, install the appropriate set of boot ROMs on your target board(s). When installing boot ROMs, be careful to:

- Install each device only in the socket indicated on the label.
- Note the correct orientation of pin 1 for each device.
- Use anti-static precautions whenever working with integrated circuit devices.

See *4.8 Configuring and Building a VxWorks Boot Program*, p.164 for instructions on creating a new boot program with parameters customized for your site.

Setting Board Jumpers

Many CPU and Ethernet controller boards still have configuration options that are selected by hardware jumpers, although this is less common than in the past. These jumpers must be installed correctly before VxWorks can boot successfully. You can determine the correct jumper configuration for your target CPU from the information provided in the target-information reference for your BSP; see Help>Manuals contents>BSP Reference in the Tornado Launcher.

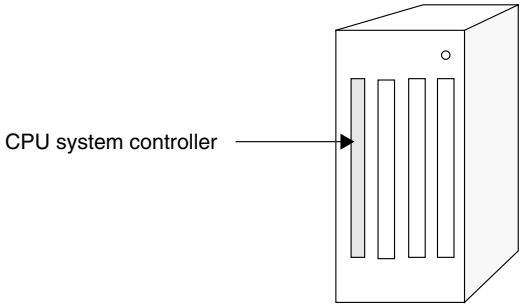
Board Installation and Power

For bare-board targets, use the power supply recommended by the board manufacturer (often a PC power supply).

If you are using a VME chassis, install the CPU board in the first slot of the backplane. See Figure 2-3.

2. NetROM is a trademark of Applied Microsystems Corporation.

Figure 2-3 **Assembling VME Targets**



On a VMEbus backplane, there are several issues to consider:

P1 and P2 Connectors
: The P1 connector must be completely bussed across all the boards in the system.

  Many systems also require the P2 bus. Some boards require power on the P2 connector, and some configurations require the extended address and data lines of the B row of the P2 bus.

System Controller
: The VME bus requires a *system controller* to be present in the first slot. Many CPU boards have a system controller on board that can be enabled or disabled by hardware jumpers. On such boards, enable the system controller in the first slot and disable it in all others. The diagrams in the target-information reference indicate the location of the system controller enable jumper, if any.

  Alternatively, a separate system controller board can be installed in the first slot and the CPU and Ethernet boards can be plugged into the next two slots.

Empty Slots
: The VME bus has several daisy chained signals that must be propagated to all the boards on the backplane. If you leave any slot empty between boards on the backplane, you must jumper the backplane to complete the daisy chain for the BUS GRANT and INT ACK signals.

Connecting the Cables

All supported VxWorks targets include at least one on-board serial port. This serial port must be connected to an ASCII terminal (or equivalent device) for the default

configuration. After the initial configuration of the boot parameters and getting started with VxWorks, you may wish to configure VxWorks to boot automatically without a terminal. Refer to the CPU board hardware documentation for proper connection of the RS-232 signals.

For the Ethernet connection, a transceiver cable must be connected from the Ethernet controller to an Ethernet transceiver.

2.5 Host-Target Communication Configuration

Connecting the target server to the target in a configuration other than the default requires a little work on both the host and target. The next few subsections describe the details for network connections, END connections, serial line connections, the NetROM Emulator, and the transparent mode driver.

2.5.1 Network Connections

A network connection is the easiest to set up and use, because most VxWorks targets already use the network (for example, to boot); thus, no additional target set-up is required. Furthermore, a network interface is typically a board's fastest physical communication channel.

When VxWorks is configured and built with a network interface for the target agent (the default configuration), the target server can connect to the target agent using the default *wdbpipe* back end (see *Target-Server Configuration Options*, p.76).

The target agent can receive requests over any device for which a VxWorks network interface driver is installed. The typical case is to use the device from which the target was booted; however, any device can be used by specifying its IP address to the target server.

Configuring the Target Agent for Network Connection

The default VxWorks system image is configured for a networked target. See *4.7 Configuring the Target-Host Communication Interface*, p.156 for information about configuring VxWorks for various target agent communications interfaces.

2.5.2 END Connections

An END (Enhanced Network Driver) connection supports dual mode agent execution. This connection can only be used if the BSP uses an END driver (which has a polled interface). With an END connection, the agent uses an END driver directly, rather than going through the UDP/IP protocol stack.

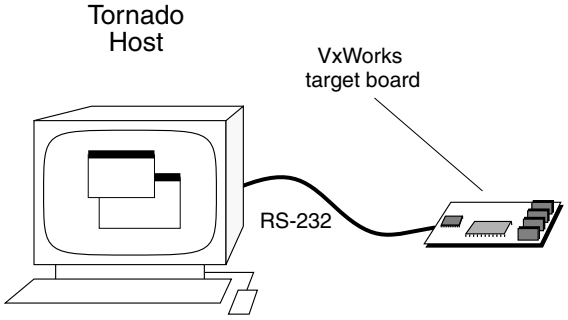
Configuring the Target Agent for END Connection

See *Configuration for an END Driver Connection*, p.157 for information about configuring the VxWorks target agent for an END connection.

2.5.3 Serial-Line Connections

Figure 2-4 illustrates a minimal cross-development configuration: the target is a bare board, connected to the host development system by a single serial line. For a configuration of this sort, use a combination of a boot mechanism that does not require a network and an alternative Tornado communications back end.

Figure 2-4 **A Minimal Tornado Configuration**



Tornado can operate over a raw serial connection between the host and target systems, and can operate on standalone systems that have no network connection to other hosts.

When you connect the host and target exclusively over serial lines, you must:

- Configure and build a boot program for the serial connection, because the default boot configuration uses an FTP download from the host.

- Reconfigure and rebuild VxWorks with a target agent configuration for a serial connection.
- Configure and start a target server for a serial connection.

For more information, see *4.7 Configuring the Target-Host Communication Interface*, p.156.

A raw serial connection has some advantages over an IP connection. The raw serial connection allows you to scale down the VxWorks system (even during development) for memory-constrained applications that do not require networking: you can remove the VxWorks network code from the target system.

When working over a serial link, use the fastest possible line speed. The Tornado tools—especially the browser and the debugger—make it easy to set up system snapshots that are periodically refreshed. Refreshing such snapshots requires continuing traffic between host and target. On a serial connection, the line speed can be a bottleneck in this situation. If your Tornado tools seem unresponsive over a serial connection, try turning off periodic updates in the browser, or else closing any debugger displays you can spare.

Configuring the Target Agent for Serial Connection

To configure the target agent for a raw serial communication connection, reconfigure and rebuild VxWorks with a serial communication interface for the target agent (see *Configuration for Serial Connection*, p.160).

Configuring the Boot Program for Serial Connection

When you connect the host and target exclusively over serial lines, you must configure and build a boot program for the serial connection because the default boot configuration uses an FTP download from the host (see *4.8 Configuring and Building a VxWorks Boot Program*, p.164). The simplest way to boot over a serial connection is by using the Target Server File System. See *2.6.7 Booting a Target Without a Network*, p.55.

Testing the Connection

Be sure to use the right kind of cable to connect your host and target.Use a simple Tx/Tx/GND serial cable because the host serial port is configured not to use handshaking. Many targets require a null-modem cable; consult the target-board

documentation. Configure your host-system serial port for a full-duplex (no local echo), 8-bit connection with one stop bit and no parity bit. The line speed must match whatever is configured into your target agent.

Before trying to attach the target server for the first time, test the serial connection to the target. To help verify the connection, the target agent sends the following message over the serial line when it boots (with **WDB_COMM_SERIAL**):

```
WDB READY
```

To test the connection, attach a terminal emulator[3] to the target-agent serial port, then reset the target. If the WDB READY message does not appear, or if it is garbled, check the configuration of the serial port you are using on your host.

As a further debugging aid, you can also configure the serial-mode target agent to echo all characters it receives over the serial line. This is not the default configuration, because as a side effect it stops the boot process until a target server is attached. If you need this configuration in order to set up your host serial port, edit *installDir*/**target/src/config/usrWdb.c**.

Look for the following lines:

```
#ifdef  INCLUDE_WDB_TTY_TEST
    /* test in polled mode if the kernel hasn't started */

    if (taskIdCurrent == 0)
        wdbSioTest (pSioChan, SIO_MODE_POLL, 0);
    else
        wdbSioTest (pSioChan, SIO_MODE_INT, 0);
#endif  /* INCLUDE_WDB_TTY_TEST */
```

In both calls to **wdbSioTest( )**, change the last argument from 0 to 0300.

With this configuration, attach any terminal emulator on the host to the **tty** port connected to the target to verify the serial connection. When the serial-line settings are correct, whatever you type to the target is echoed as you type it.

WARNING: Because this configuration change also prevents the target from completing its boot process until a target server attaches to the target, it is best to change the **wdbSioTest( )** third argument back to the default 0 value as soon as you verify that the serial line is set up correctly.

3. Commonly available terminal emulators are **tip**, **cu**, and **kermit**; consult your host reference documentation.

Starting the Target Server

After successfully testing the serial connection, you can connect the target server to the agent by following these steps:

1. Close the serial port that you opened for testing (if you do not close the port, then it will be busy when the target server tries to use it).
2. Start the target server with the serial back end to connect to the agent. Use the **tgtsvr -B** option to specify the back end, and also specify the line speed to match the speed configured into your target:

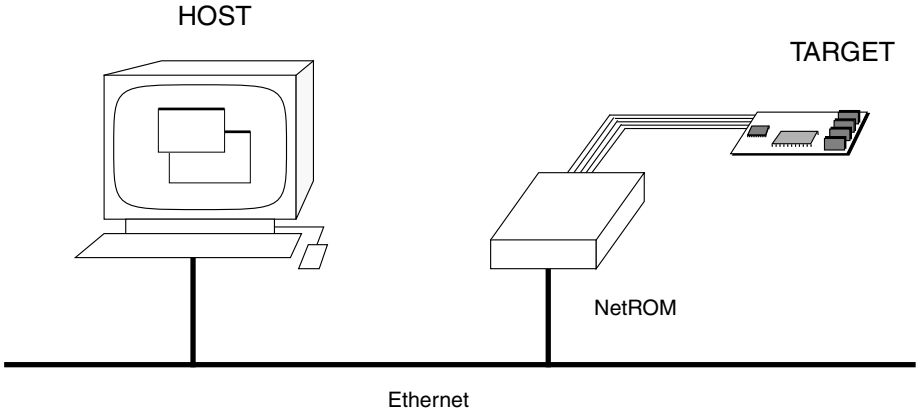
```
% tgtsvr -V targetname -B wdbserial -bps 38400 &
```

You can also use the Tornado GUI to configure and start a target server (see *3.5 Managing Target Servers*, p.72).

2.5.4 The NetROM ROM-Emulator Connection

The agent can be configured to communicate with the target server using the target board's ROM socket. Tornado supports this configuration for NetROM, a ROM emulator produced by Applied Microsystems Corporation. Contact your nearest Wind River office (listed on the back cover) for information about support for other ROM emulators. Figure 2-5 illustrates this connection method.

Figure 2-5 **Connecting a Target through NetROM**



The NetROM acts as a liaison between the host and target. It communicates with the host over Ethernet, and with the target through ROM emulation pods that are plugged into the target board's ROM sockets. The NetROM allows you to download new ROM images to the target quickly. In addition, a 2 KB segment of the NetROM's emulation pod is dual-port RAM, which can be used as a communication path. The target agent uses the NetROM's read-only protocol to transfer data up to the host. It works correctly even on boards that do not support write access to the ROM banks.

This communication path has many benefits: it provides a connection which does not intrude on any of your board's I/O ports, it supports both task-mode and system-mode debugging, it is faster than a serial-line connection, and it provides an effective way to download new VxWorks images to the target.

NOTE: The information about NetROM in this section is a summary of NetROM documentation, with some supplementary remarks. This section is not a replacement for the NetROM documentation. In particular, refer to that documentation for full information about how to connect the NetROM to the network and to your target board.

For information about booting a target without a network, see *2.6.7 Booting a Target Without a Network*, p.55.

Configuring the Target Agent for NetROM

To configure the target agent for a NetROM communication connection, reconfigure and rebuild VxWorks with a NetROM interface for the target agent. Several configuration macros are used to describe a board's memory interface to its ROM banks. You may need to override some of them for your board. See *Configuration for NetROM Connection*, p.158.

Configuring the NetROM

Before a target server on your host can connect to the target agent over NetROM, some hardware and software configuration is necessary. The following steps outline this process.

1. Configure the NetROM IP address from your host system.

   When it powers up, the NetROM knows its own Ethernet address, but does not know its internet (IP) address.

There are two ways of establishing an IP address for the NetROM:

- Connect a terminal to the NetROM serial console, and specify the IP address manually when you power up the NetROM for *Step 4*. This solution is simple, but you must repeat it each time the NetROM is powered up or restarted.
- Configure a network server to reply to RARP or BOOTP requests from the NetROM. On power-up, the NetROM automatically broadcasts both requests. This solution is preferable, because it permits the NetROM to start up without any interaction once the configuration is working.

Since the RARP and BOOTP requests are broadcast, any host connected to the same subnet can reply. Configure only one host to reply to NetROM requests.

2. Prepare a NetROM startup file.

   After the NetROM obtains its IP address, it loads a startup file. The pathname for this file depends on which protocol establishes the IP address:

   - BOOTP: A startup-file name is part of the BOOTP server's reply to the BOOTP request. Record your choice of startup-file pathname in the BOOTP table (typically **/etc/bootptab**).
   - RARP: When the IP address is established by a reply to the RARP request, no other information accompanies the IP address. In this case, the NetROM derives a file name from the IP address. The file name is constructed from the numeric (dot-decimal) IP address by converting each address segment to two hexadecimal digits. For example, a NetROM at IP address 147.11.46.164 expects a setup file named **930B2EA4** (hexadecimal digits from the alphabet are written in upper case). The NetROM makes three attempts to find the startup file, with each of the following pathnames: **./***filename*, **/tftpboot/***filename*, and *filename* without any other path information.

   The startup file contains NetROM commands describing the emulated ROM, the object format, path and file names to download, and so on. The following example NetROM startup file configures the Ethernet device, adds routing information, records the object-file name to download and the path to it, and establishes ROM characteristics.

Example 2-1 **Sample NetROM Startup File**

```
begin
    ifconfig le0 147.11.46.164 netmask 255.255.255.0 broadcast 147.11.46.0
    setenv filetype srecord
    setenv loadpath /tftpboot
```

```
    setenv loadfile vxWorks_rom.hex
    setenv romtype 27c020
    setenv romcount 1
    setenv wordsize 8
    setenv debugpath readaddr
    set udpsrcmode on
    tgtreset
end
```

NOTE: The environment variable **debugpath** should be set to **dualport** (rather than to **readaddr**) if you are using the 500-series NetROM boxes.

When you create a NetROM startup file, remember to set file permissions to permit the TFTP file server to read the file.

For more information regarding NetROM boot requirements, refer to NetROM documentation. Consult your system administrator to configure your host to reply to RARP or BOOTP requests (or see host-system documentation for **bootpd** or **rarpd**).

3. Connect NetROM to Ethernet network; plug NetROM pods into target ROM sockets.

WARNING: Do not power up either the NetROM or the target yet. Pod connections and disconnections should be made while power is off on both the NetROM and the target board.

WARNING: Some board sockets are designed to support either ROM or flash PROM. On this kind of socket, a 12V potential is applied to pin 1 each time the processor accesses ROM space. This potential may damage the NetROM. In this situation, place an extra ROM socket with pin 1 removed between the NetROM pod and the target-board socket.

WARNING: Take great care when you plug in NetROM pod(s). Double check the pod connections, especially pin 1 position and alignment. A pod connection error can damage either the NetROM itself, the target board, or both.

The pins coming out of the NetROM's DIP emulation pods are very easy to break, and the cables are expensive to replace. It is a good idea to use a DIP extender socket, because they are much cheaper and faster to replace if a pin breaks.

NetROM pod 0 differs from other pods because it implements the dual-port RAM. This special port is used by NetROM both to send data to the board and

to receive data from the board: that is, the dual port is the communication path between the NetROM and the board.

4. Power up the NetROM (but not the target).

   Once the required NetROM address and boot information is configured on a host, the NetROM can be powered up. To verify that the NetROM has obtained its IP address and loaded and executed the startup file, you can connect to a NetROM command line with a **telnet** session.

   The following example shows the expected response from a NetROM at IP address 147.11.46.164:

```
% telnet 147.11.46.164
Trying 147.11.46.164
Connected to 147.11.46.164
Escape character is '^]'

NETROM TELNET
NetROM>
```

   At the NetROM prompt, you can display the current configuration by entering the command **printenv** to verify that the startup file executed properly.

5. Download test code to the NetROM.

   One method is to type the **newimage** command at the NetROM prompt. This command uses the TFTP protocol to download the image specified by the **loadfile** environment variable from the path specified by the **loadpath** environment variable (which is **/tftpboot/vxWorks_rom.hex** if you use the startup script in Example 2-1). After the NetROM configuration is stable, you can include this command in the startup file to download the image automatically. Wait to be certain the image is completely downloaded before you power up your target. This method takes about 30 seconds to transfer the image.

   A faster method is to use two host utilities from AMC: **rompack** packs a ROM image into a compact file (with the default name **outfile.bin**); **download** ships the packed file to the NetROM. This method takes only about five seconds to transfer a new image to the target. This UNIX shell script shown in uses these utilities to send an image to the NetROM whose IP address is recorded in the script variable **ip**:

```
#! /bin/sh
if [ $# != 1 ]; then
    echo "Usage: $0 <filename>"
    exit 1
fi
```

```
file=$1
ip=t46-154

if [ -r "$file" ];  then
    echo "Downloading $file to the NetROM at $ip."
    rompack -c 1 -r 27c020 -x $file 0 0
    download outfile.bin $ip
else
    echo "$0: \"$file\" not found"
    exit 1
fi

echo Done.
exit 0
```

The **rompack** option flags specify how to pack the image within the emulator pods. The **-c 1** option specifies a ROM count of one, which means that the image goes in a single ROM socket. The **-r 27c020** option specifies the type of ROM. The two trailing numbers are the base and offset from the start of ROM space. Both are typically zero.

6. Power up your target.

   The target CPU executes the object code in the emulated ROM. Make sure the code is running correctly. For example, you might want to have it flash an LED.

Starting the Target Server

Start the target server as in the following example, using the **-B** option to specify the NetROM back end.

```
% tgtsvr -V 147.11.46.164 -B netrom &
```

In this example, **147.11.46.164** is the IP address of the NetROM. (You can also use the Tornado GUI to configure and start a target server; see *Tornado Getting Started Guide*.)

If the connection fails, try typing the following command at the NetROM prompt:

```
NetROM> set debugecho on
```

With this setting, all packets sent to and from the NetROM are copied to the console. You may need to hook up a connector to the NetROM serial console to see the **debugecho** output, even if your current console with NetROM is attached through Telnet (later versions of NetROM software may not have this problem). If you see packets sent from the host, but no reply from the target, you must modify

the target NetROM configuration parameters described in section *Configuration for Network Connection*, p.159.

NOTE: With a NetROM connection, you must inform the NetROM when you reboot the target. You can do this as follows at the NetROM prompt:

```
NetROM> tgtreset
```

Troubleshooting the NetROM ROM-Emulator Connection

If the target server fails to connect to the target, the following troubleshooting procedures can help isolate the problem.

Download Configuration

It is possible that the NetROM is not correctly configured for downloading code to the target. Make sure you can download and run a simple piece of code (for example, to blink an LED — this code should be something simpler than a complete VxWorks image).

Initialization

If you can download code and execute it, the next possibility is that the board initialization code is failing. In this case, it never reaches the point of trying to use the NetROM for communication. The code in **target/src/config/usrWdb.c** makes a call to **wdbNetromPktDevInit( )**. If the startup code does not get to this point, the problem probably lies in the BSP. Contact the vendor that supplied the BSP for further troubleshooting tips.

RAM Configuration

If the NetROM communication initialization code is being called but is not working, the problem could be due to a mis-configuration of the NetROM. To test this, modify the file **wdbNetromPktDrv.c**. Change the following line:

```
int wdbNetromTest = 0;
```

to:

```
int wdbNetromTest = 1;
```

NOTE: There are two versions of **wdbNetromPktDrv.c**. The one for the 400 series is located in **target/src/drv/wdb** and the one for the 500 series is located in **target/src/drv/wdb/amc500**. Be sure to edit the appropriate one.

When you rerun VxWorks with this modification, the **wdbNetromPktDevInit( )** routine attempts to print a message to NetROM debug port. The initialization code halts until you connect to the debug port (1235), which you can do by typing:

```
% telnet NetROM_IPaddress 1235
```

If the debug port successfully connects, the following message is displayed in the **telnet** window:

```
WDB NetROM communication ready
```

If you do not see this message, the NetROM dual-port RAM has not been configured correctly. Turn off the processor cache; if that does not solve the problem, contact AMC for further trouble shooting tips:

AMC web page:	**http://www.amc.com/**
AMC tech-support:	**1-800-ask-4amc**
	support@amc.com

If everything has worked up to this point, reset **wdbNetromTest** back to zero and end your **telnet** session.

Communication

Type the following at the NetROM prompt:

```
NetROM> set debugecho on
```

This causes data to be echoed to the NetROM console when packets are transmitted between the host and target. If you have a VxWorks console available on your target, edit **wdbNetromPktDrv.c** by changing the following line:

```
int wdbNetromDebug = 0;
```

to:

```
int wdbNetromDebug = 1;
```

This causes messages to be echoed to the VxWorks console when packets are transmitted between the host and target.

NOTE: You may need to hook up a connector to the NetROM serial console to see the **debugecho** output, even if your current console with NetROM is attached through **telnet**.

Retry the connection:

(1) Kill the target server.

(2) Type **tgtreset** at the NetROM prompt.

(3) Reboot your target.

(4) Start the target server using the **-Bd** option to log transactions between the target server and the agent to a log file. Use the target server **-Bt** option to increase the timeout period. (This is necessary whenever the NetROM debug echo feature is enabled, because **debugecho** slows down the connection.)

At this point, you have debugging output on three levels: the target server is recording all transactions between it and the NetROM box; the NetROM box is printing all packets it sees to its console; and the WDB agent is printing all packets it sees to the VxWorks console. If this process does not provide enough debug information to resolve your problems, contact Wind River technical support for more troubleshooting assistance.

2.5.5 The Transparent Mode Driver (TMD)

The TM driver provides the same connection capability as an Ethernet or serial cable would. However, the TM driver works through the Wind River visionICE II/visionPROBE II hardware debug tools. Physically, the connection is implemented over the BDM/JTAG/EJTAG emulation connection provided by the tools. This can be advantageous if the target being used does not have an Ethernet or serial port on it, or if the ports are required for something else. It can also be useful when the target ports are available, but the software that controls them is not yet working.

The Wind River TM driver supports both system and task level debugging. The TM driver also supports the /vio (virtual I/O) sub-channel of the WDB protocol.

Configuring the Target Agent for TMD

The TMD is added to the current build by selecting the VxWorks tab in the project dialog window. Expand the VxWorks entry associated with your project, and from

the list that appears, select development tool components>select WDB connection>WDB visionTMD connection.

For the TMD to be added to the current build, the WDB visionTMD connection entry must be made the active WDB connection. By default, when this project was built, the WDB END driver connection was included in the project. That entry now appears bolded because it is the active connection.

In order for the project to build correctly, only one WDB connection can be active. To include the WDB visionTMD connection, right-click on Select WDB connection and select Configure 'select WDB connection' on the pop-up menu. The properties dialog wind appears.

Click on the Components tab, scroll down the list, and click on the WDB visionTMD connection check box. The WDB END driver connection will automatically be deselected. Click the Apply button to select the TMD component. If the Include Component(s) dialog contains the correct information, click Ok to confirm it and close the dialog box. The WDB visionTMD connection is now the active connection.

NOTE: Only one connection can be active at a time. If more than one connection is made active in the list, then the names of the folders where the error is located turn red to alert you that there is a configuration error in the project. Making one of the active connections inactive will correct the error.

Now that you have specified the TMD, rebuild VxWorks to include the component in your image.

Configuring visionICE II/visionPROBE II

The debugger being used must be configured correctly to download the vxWorks image created in the previous steps to the target. The debugger will also be used to start the image running once it has been downloaded. Once the image is running on the target, the TM Driver will also be available since the WDB agent that is included in the vxWorks image uses the TM Driver as the connection mechanism.

Instructions for configuring the debugger and downloading and executing the vxWorks image on the target are provided for two of Wind Rivers debuggers in the *Transparent Mode Driver User's Guide*.

WARNING: Do not attempt to continue on to the next section without first downloading a valid, working image to the target and executing it. You will not be able to launch a target server or make use of the Tornado tools without this step being complete.

Information on configuring visionICE II for network operation is available in the visionICE II User's Manual. In addition, the UDP Console Port must be set to 17185.

1. In the **ethsetup** menu, accessible from the **NET>** prompt (as described in the manuals listed above), select option 5 to view the current port settings.
2. If **[7], UDPCNSL** is already set to 17185, no modifications are necessary, and you may exit this menu.
3. If **[7]** is not set to 17185, type **6** to allow the port values to be changed.
4. Type **7**, which will allow the **UDPCNSL** port setting to be changed, and change it to 17185.
5. Type **0** to exit the **Change Port Settings** menu.
6. Type **8** to save the changes.
7. Type **9** to exit the **ethsetup** menu.

Starting the Target Server

Once the VxWorks image is running on the target, the host will be able to communicate with the running WDB agent. To do this, a target server must be configured and activated. Follow the following steps:

1. With the VxWorks image running on the target, return to the Tornado project window. Select Tools>Target Server>Configuration from the main menu.
2. Click the New button at the top right side of the dialog window, enter a description, and check the Add description to menu check box. In the space beside Target Server Name, enter the same description as you placed in the Description space. This will result in a link being created in the Tornado Tools menu that will automatically launch the target server when selected.
3. Define Target Server Properties by selecting an item from the drop-down menu and then completing the related properties as shown in Table 2-3.

Table 2-3 **Target Server Properties Settings**

Target Server Properties	Parameter Values
Back End	Available back ends: **wdbrpc** TargetName/IP Address: If a visionICE II is being used, this is the IP address of the visionICE II unit. If a visionPROBE II is being used, this is the IP address of the host or a loopback address of 127.0.0.1 Keep defaults for other parameters
Core Files and Symbols	Select the File option and enter the path to the VxWorks file associated with the project.
Target Server File System	Check Enable File System Select Read/Write option

To launch a correctly configured target server using the command line, enter the following:

```
% tgtsvr.exe -n 127.0.0.1 -V -B wdbrpc -R C:/Tornado/2.2 -RW -c myCoreFile
```

For more information about using either the GUI or the command line to configure target servers, see *2.7 Connecting a Tornado Target Server*, p.56.

2.6 Booting VxWorks

Once you have correctly configured your host software and target hardware, establish a terminal connection from your host to the target, using the serial port that connects the two systems.[4] For example, the following command starts a **tip** session for the second serial port at 9600 bps:

```
% tip /dev/ttyb -9600
```

See your BSP documentation for information about the bps rate (Help>Manuals contents>BSP Reference in the Tornado Launcher, or see the file *installDir*/**docs/BSP_Reference.html**).

You are now ready to turn on the target system power and boot VxWorks.

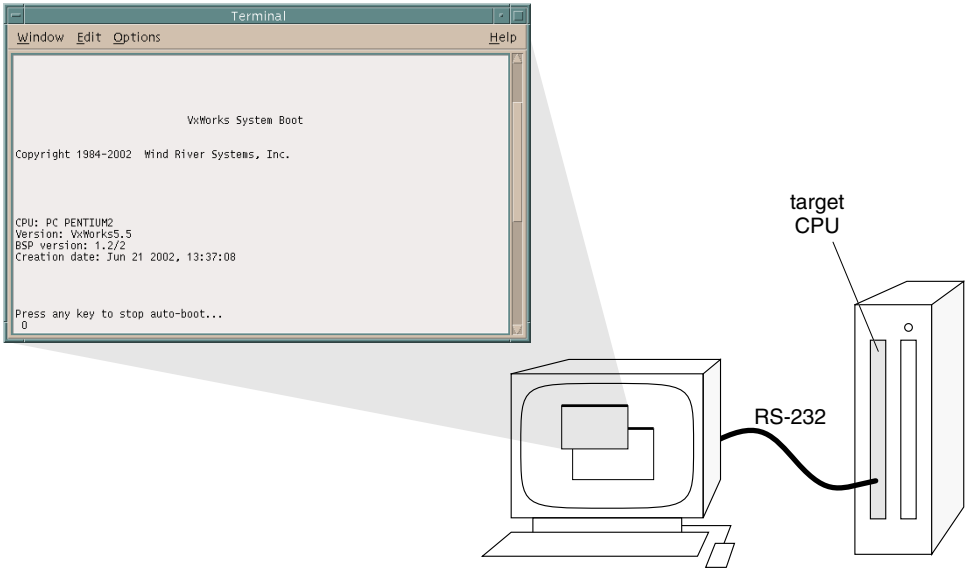
4. Commonly available terminal emulators are **tip**, **cu**, and **kermit**; consult your host reference documentation.

2.6.1 Default Boot Process

When you boot VxWorks with the default boot program (from ROM, diskette, or other medium), you must use the VxWorks command line to provide the boot program with information that allows it to find the VxWorks image on the host and load it onto the target. The default boot program is designed for a networked target, and needs to have the correct host and target network addresses, the full path and name of the file to be booted, the user name, and so on.[5]

When you power on the target hardware (and each time you reset it), the target system executes the boot program from ROM; during the boot process, the target uses its serial port to communicate with your terminal or workstation. The boot program first displays a banner page, and then starts a seven-second countdown, visible on the screen as shown in Figure 2-6. Unless you press any key on the keyboard within that seven-second period, the boot loader will attempt to proceed with a default configuration, and will not be able to boot the target with VxWorks.

Figure 2-6 **Boot Program: Communication and Boot Banner Display**



5. Unless your target CPU has nonvolatile RAM (NVRAM), you will eventually find it useful to build a new version of the boot loader that includes all parameters required for booting a VxWorks image (see *4.8 Configuring and Building a VxWorks Boot Program*, p.164). In the course of your developing an application, you will also build bootable applications (see *4.5 Creating a Bootable Application*, p.147).

2.6.2 Entering New Boot Parameters

To interrupt the boot process and provide the correct boot parameters, first power on (or reset) the target; then stop the boot sequence by pressing any key during the seven-second countdown. The boot program displays the VxWorks boot prompt:

```
[VxWorks Boot]:
```

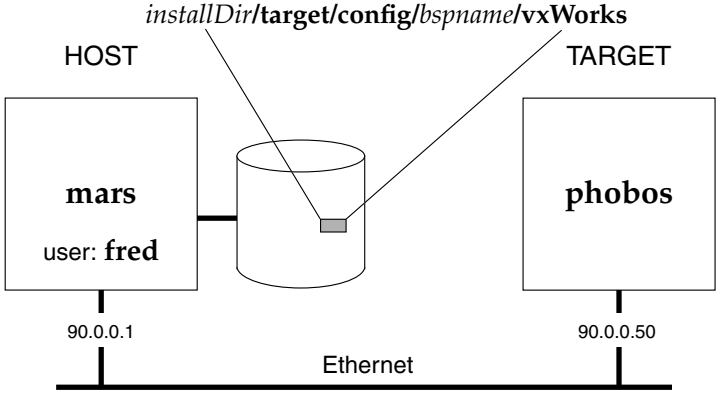
To display the current boot parameters, type **p** at the boot prompt, as follows:

```
[VxWorks Boot]: p
```

A display similar to the following appears; the meaning of each of these parameters is described in the next section. This example corresponds to the configuration shown in Figure 2-7. (The **p** command does not actually display blank fields, although this illustration shows them for completeness.)

```
boot device             : ln
processor number        : 0
host name               : mars
file name               : installDir/target/config/bspname/vxWorks
inet on ethernet (e)    : 90.0.0.50:ffffff00
inet on backplane (b)   :
host inet (h)           : 90.0.0.1
gateway inet (g)        :
user (u)                : fred
ftp password (pw)(blank=use rsh) :
flags (f)               : 0x0
target name (tn)        : phobos
startup script (s)      :
other (o)               :
```

Figure 2-7 **Boot Configuration Example**



To change the boot parameters, type **c** at the boot prompt, as follows:

```
[VxWorks Boot]: c
```

In response, the boot program prompts you for each parameter. If a particular field has the correct value already, press **RETURN**. To clear a field, enter a period (**.**), then **RETURN**. If you want to quit before completing all parameters, type **CTRL+D**.

Network information *must* be entered to match your particular system configuration. The Internet addresses must match those in **/etc/hosts** on your UNIX host, as described in *Establishing the VxWorks System Name and Address*, p.26.

If your target has nonvolatile RAM (NVRAM), boot parameters are retained there even if power is turned off. For each subsequent power-on or system reset, the boot program uses these stored parameters for the automatic boot configuration.

2.6.3 Boot Program Commands

The VxWorks boot program provides a limited set of commands. To see a list of available commands, type either **h** or **?** at the boot prompt, followed by **RETURN**:

```
[VxWorks Boot]: ?
```

Table 2-4 lists and describes each of the VxWorks boot commands and their arguments.

Table 2-4 **VxWorks Boot Commands**

Command	Description
h	Help command—print a list of available boot commands.
?	Same as **h**.
@	Boot (load and execute the file) using the current boot parameters.
p	Print the current boot parameter values.
c	Change the boot parameter values.
l	Load the file using current boot parameters, but without executing.
g *adrs*	Go to (execute at) hex address *adrs*.
d *adrs*[**,** *n*]	Display *n* words of memory starting at hex address *adrs*. If *n* is omitted, the default is 64.

Table 2-4 **VxWorks Boot Commands** *(Continued)*

Command	Description
m *adrs*	Modify memory at location *adrs* (hex). The system prompts for modifications to memory, starting at the specified address. It prints each address, and the current 16-bit value at that address, in turn. You can respond in one of several ways: **ENTER**: Do not change that address, but continue prompting at the next address. *number*: Set the 16-bit contents to *number*. **.** (dot): Do not change that address, and quit.
f *adrs*, *nbytes*, *value*	Fill *nbytes* of memory, starting at *adrs* with *value*.
t *adrs1*, *adrs2*, *nbytes*	Copy *nbytes* of memory, starting at *adrs1*, to *adrs2*.
s [**0** | **1**]	Turn the CPU system controller **ON** (**1**) or **OFF** (**0**) (only on boards where the system controller can be enabled by software).
e	Display a synopsis of the last occurring VxWorks exception.
n *netif*	Display the address of the network interface device *netif*.

2.6.4 Description of Boot Parameters

Each of the boot parameters is described below, with reference to the example in *2.6.2 Entering New Boot Parameters*, p.48. The letters in parentheses after some parameters indicate how to specify the parameters in the command-line boot procedure described in *2.6.6 Alternate Booting Procedures*, p.54.

boot device
The type of device to boot from. This must be one of the drivers included in the boot ROMs (for example, **enp** for a CMC controller). Due to limited space in the boot ROMs, only a few drivers can be included. A list of included drivers is displayed at the bottom of the help screen (type **?** or **h**).

processor number
A unique identifier for the target in systems with multiple targets on a backplane (zero in the example). The first CPU must be processor number 0 (zero).

host name
The name of the host machine to boot from. This is the name by which the host is known to VxWorks; it need not be the name used by the host itself. (The host name is **mars** in the example of *2.6.2 Entering New Boot Parameters*, p.48.)

file name
The full pathname of the VxWorks object module to be booted (**/usr/wind/target/config/***bspname***/vxWorks** in the example). This pathname is also reported to the host when you start a target server, so that it can locate the host-resident image of VxWorks.[6]

inet on ethernet (e)
The Internet address of a target system with an Ethernet interface (90.0.0.50 in the example).

inet on backplane (b)
The Internet address of a target system with a backplane interface (blank in the example).

host inet (h)
The Internet address of the host to boot from (90.0.0.1 in the example).

gateway inet (g)
The Internet address of a gateway node if the host is not on the same network as the target (blank in the example).

user (u)
The user name that VxWorks uses to access the host (**fred** in the example); that user must have read access to the VxWorks boot-image file. VxWorks must have access to this user's account, either with the FTP password provided below, or through the files **.rhosts** or **/etc/hosts.equiv** discussed in *Giving VxWorks Access to the Host*, p.27.

ftp password (pw)
The "user" password. This field is optional. If you provide a password, FTP is used instead of RSH. If you do not want to use FTP, then leave this field blank.

6. If the same pathname is not suitable for both host and target—for example, if you boot from a disk attached only to the target—you can specify the host path separately to the target server, using the Core file field (**-c** option). See *3.5 Managing Target Servers*, p.72.

flags (f)
Configuration options specified as a numeric value that is the sum of the values of selected option bits defined below. (This field is zero in the example because no special boot options were selected.)

0x01 = Do not enable the system controller, even if the processor number is 0. (This option is board specific; refer to your target documentation.)
0x02 = Load all VxWorks symbols, instead of just globals.
0x04 = Do not auto-boot.
0x08 = Auto-boot fast (short countdown).
0x20 = Disable login security.
0x40 = Use BOOTP to get boot parameters.
0x80 = Use TFTP to get boot image.
0x100 = Use proxy ARP.

target name (tn)
The name of the target system to be added to the host table (**phobos** in the example).

startup script (s)
If the target-resident shell is included in the downloaded image, this parameter allows you to pass to it the complete path name of a startup script to execute after the system boots. In the default Tornado configuration, this parameter has no effect, because the target-resident shell is not included.

other (o)
This parameter is generally unused and available for applications (blank in the example). It can be used when booting from a local SCSI disk to specify a network interface to be included.

2.6.5 Booting With New Parameters

Once you have entered the boot parameters, initiate booting by typing the @ command at the boot prompt:

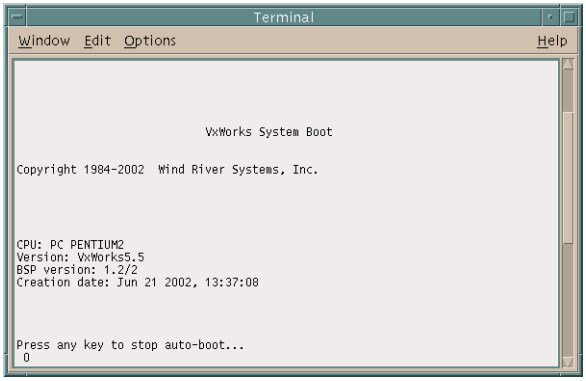
```
[VxWorks Boot]: @
```

Figure 2-8 shows a typical VxWorks boot display. The VxWorks boot program prints the boot parameters, and the downloading process begins. The following information is displayed during the boot process:

- The boot program first initializes its network interfaces.

- While VxWorks is booting, you can see the size of each VxWorks section (text, data, and bss) as it is loaded.
- After the system is completely loaded, the boot program displays the entry address and transfers control to the loaded VxWorks system.
- When VxWorks starts up, it begins just as the boot ROM did, by initializing its network interfaces; the network-initialization messages appear again, sometimes accompanied by other messages about optional VxWorks facilities.

Figure 2-8 **VxWorks Booting Display**



After that point, VxWorks is up and ready to attach to the Tornado tools, as discussed in *2.7 Connecting a Tornado Target Server*, p.56.

The boot display may be useful for troubleshooting. The following hints refer to Figure 2-8. For more troubleshooting ideas, see *2.10 Troubleshooting*, p.59.

- If the initial “Attaching network interface” is displayed without the corresponding “done,” verify that the system controller is configured properly and the Ethernet board is properly jumpered.
- If “Loading...” is displayed without the size of the VxWorks image, this may indicate problems with the Ethernet cable or connection, or an error in the network configuration (for example, a bad host or gateway Internet address).
- If the line “Starting at” is printed and there is no further indication of activity from VxWorks, this generally indicates there is a problem with the boot image.
- If “Attaching network interface” is displayed without the “done,” this may indicate there is a problem with the network driver in the newly loaded VxWorks image.

2.6.6 Alternate Booting Procedures

To boot VxWorks, you can also use the command line, take advantage of non-volatile RAM, or create new boot programs for your target.

Command-Line Parameters

Instead of being prompted for each of the boot parameters, you can supply the boot program with all the parameters on a single line at the boot prompt (**[VxWorks Boot]:**) beginning with a dollar sign character ("**$**"). For example:

```
$ln(0,0)mars:/usr/wind/target/config/bspname/vxWorks e=90.0.0.50 h=90.0.0.1 u=fred
```

The order of the assigned fields (those containing equal signs) is not important. Omit any assigned fields that are irrelevant. The codes for the assigned fields correspond to the letter codes shown in parentheses by the **p** command. For a full description of the format, see the reference entry for **bootStringToStruct( )** in **bootLib**.

This method can be useful if your workstation has programmable function keys. You can program a function key with a command line appropriate to your configuration.

Nonvolatile RAM (NVRAM)

As noted previously, if your target CPU has nonvolatile RAM (NVRAM), all the values you enter in the boot parameters are retained in the NVRAM. In this case, you can let the boot program auto-boot without having a terminal connected to the target system.

Customized Boot Programs

See *4.8 Configuring and Building a VxWorks Boot Program*, p.164 for instructions on creating a new boot program for your boot media, with parameters customized for your site. With this method, you no longer need to alter boot parameters before booting.

BSPs Requiring TFTP on the Host

Some Motorola boards that use Bug ROMs and that place boot code in flash require TFTP on the host in order to burn a new VxWorks image into flash. See your vendor documentation on how to burn flash for these boards.

2.6.7 Booting a Target Without a Network

You can boot a target that is not on a network most easily over a serial line with the Target Server File System (TSFS). The TSFS provides the target with direct access to the host's file system. Using TSFS is simpler than configuring and using PPP or SLIP.

To boot a target using TSFS, you must first reconfigure and rebuild the boot program, and copy it to the boot medium for your target (for example, burn a new boot ROM or copy it to a diskette). See *4.8 Configuring and Building a VxWorks Boot Program*, p.164.

Before you boot the target, configure a target server with the TSFS option and start it. See *Target-Server Configuration Options*, p.76.

The only boot parameters required to boot the target are **boot device** and **file name** (see *2.6.4 Description of Boot Parameters*, p.50). The **boot device** parameter should be set to **tsfs**. The **file name** parameter should be set relative to the TSFS root directory that is defined when you configure the target server for the TSFS. You can configure the boot program with these parameters, or enter them at the VxWorks prompt at boot time.

2.6.8 Rebooting VxWorks

When VxWorks is running, there are several way you can reboot VxWorks. Rebooting by any of these means restarts the attached target server on the host as well:

- Enter **CTRL+X** from the Tornado shell or a target console. (You can change this character to something else if you wish; see *7.7 Tcl: Shell Interpretation*, p.297.)
- Invoke **reboot( )** from the Tornado shell.
- Press the reset button on the target system.
- Turn the target's power off and on.

When you reboot VxWorks in any of these ways, the auto-boot sequence begins again from the countdown.

2.7 Connecting a Tornado Target Server

To make a VxWorks target ready for use with the Tornado development tools, you must start a target-server daemon for that target on one of your development hosts. One way to accomplish that is from the Tornado launcher; for that approach, see *3.5 Managing Target Servers*, p.72.

You may also want to start a server from the UNIX command line, so that your target is ready to use as soon as you enter the launcher. To start a target server this way, run the command **tgtsvr** in the background. You must specify the network name of your target (see *Establishing the VxWorks System Name and Address*, p.26) as an argument.

The following example starts a server for the target **phobos** using the default communications back end:

```
% tgtsvr -V vxsim0 &
tgtsvr.ex (vxsim0@seine): Mon Nov 30 14:09:46 1998
    Connecting to target agent... succeeded.
    Attaching C++ interface... succeeded.
    Attaching elf OMF reader for SIMSPARCSOLARIS CPU family... succeeded.
```

The **-V** (verbose) option shown above is not strictly necessary, but it is very useful for troubleshooting. With this option, **tgtsvr** produces informative messages if it cannot connect to the target.

For example, if you make an error in specifying the target name, **tgtsvr** exits when it cannot find that target. Without the **-V** option, **tgtsvr** exits silently. With the **-V** option, **tgtsvr** produces the following message for an unknown target:

```
% tgtsvr -V vxsim0 &
tgtsvr.ex (vxsim0@seine): Mon Nov 30 14:09:46 1998
    Error: Target vxsim0 unknown. Attach failed.
    Error: Backend initialization routine failed.
Problem during backend initialization.
```

There are a number of other **tgtsvr** command-line options to control the behavior of your target server. The most notable options are the following:

-B Chooses alternative methods of communicating with the target. See *2.5 Host-Target Communication Configuration*, p.31 to use other back ends.

-c Override the path to the VxWorks image on the host system.

For information on these and other command-line options, see the **tgtsvr** reference documentation (either online, or in the online *Tornado API Reference*). The easiest way to select and manage these options is with the Tornado launcher.

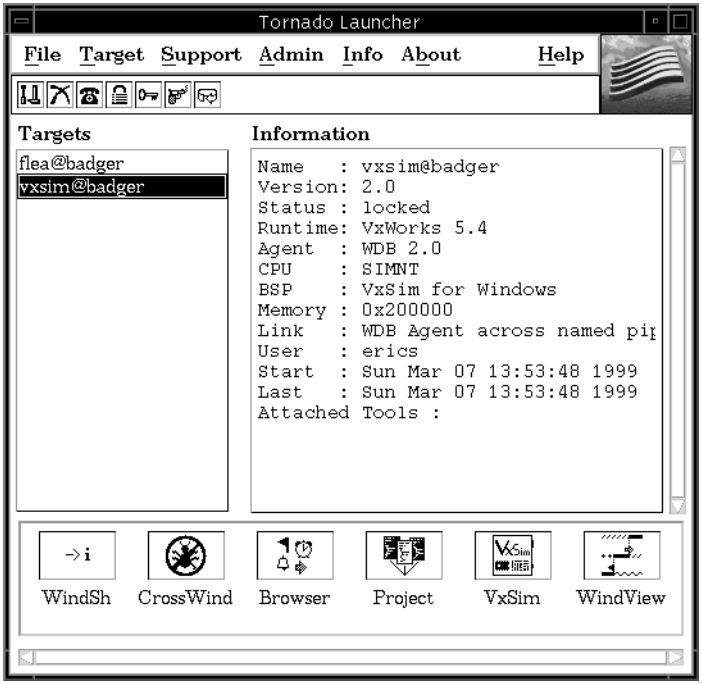
2.8 Launching Tornado

The launcher provides access to all other Tornado facilities. To start the launcher, execute the following command:

```
% launch &
```

The list on the left of the launcher window shows the targets currently available on your network. Click on one to select it, and you can see a display similar to Figure 2-9, summarizing the characteristics of that target. To explore the Tornado tools, click on any of the buttons along the bottom of the launcher screen.

Figure 2-9 **Launcher Listing Targets**



See *3. Launcher* for a detailed discussion of the launcher facilities. The remaining chapters in this guide discuss each of the other Tornado tools (which you can reach either from the command line or from the launcher).

2.9 Tornado Interface Conventions

The following conventions apply uniformly to all of the Tornado graphical tools (the launcher, the project facility, the browser, the debugger, and WindView):

Busy Box
: The Wind River logo appears in the top right of the main window of each tool. When the tool is busy, it indicates this by animating the logo.

Universal Menu Entries
: The following menu commands are always present:

  File>Quit
  : Shut down the tool.

  About>Tornado
  : Identify the version of Tornado.

  Help
  : Display online documentation; see *Online Documentation*, p.23.

Keyboard Selection from Menus
: Every Tornado menu has a one-letter abbreviation, shown by underlining that letter in the menu bar. Press the **META** shift and that letter to display the menu from the keyboard rather than using the mouse. While the menu is displayed, you can dismiss it without selecting a command by repeating the same **META**-letter shortcut.

  Within a menu, there are two ways of selecting and executing a command from the keyboard. Each command name also has an underlined letter; press that letter (no **META** shift at this level) to execute the command immediately. For example, the key sequence **META-F Q** selects Quit from any File menu. You can also use the arrow keys on your keyboard to highlight each successive menu command in turn; press **RETURN** (or **ENTER**) to execute the currently highlighted command.

Keyboard Operation of Forms (Dialogs)
: When a form is displayed, the **TAB** key selects each text or scrolling-list field in turn (shift-**TAB** selects them in reverse order). Type directly in a text field to change its value; in scrolling lists, select a new value with the arrow keys.

  When no scrolling list is selected, the arrow keys select in turn each of the toggles or buttons on the form; **RETURN** (**ENTER**) switches the highlighted toggle or presses the highlighted button.

Left Mouse Selects, Middle Mouse Drags
When there is selectable text in a Tornado display, use the left mouse button to select it. For objects that can be dragged, use the middle mouse button.

Folder Hierarchies
Whenever hierarchical data is presented graphically, a folder icon appears at each level of the hierarchy. Click on these folders to hide subordinate information; click again on the folder to reveal it once again.

2.10 Troubleshooting

If you encountered problems booting or exercising VxWorks, there are many possible causes. This section discusses the most common sources of error and how to narrow the possibilities. Please read *2.10.1 Things to Check*, p.59 before contacting the Wind River customer support group. Often, you can locate the problem just by re-checking the installation steps, your hardware configuration, and so forth.

2.10.1 Things to Check

Most often, a problem with running VxWorks can be traced to configuration errors in hardware or software. Consult the following checklist to locate a problem.

NOTE: Booting systems with complex network configurations is beyond the scope of this chapter. See *VxWorks Network Programmer's Guide: Booting over the Network*.

Hardware Configuration

- **Limit the number of variables.**

  Start with a minimal configuration of a single target CPU board and possibly an Ethernet board.

- **Be sure your backplane is properly powered and bussed.**

  For targets on a VMEbus backplane, most configurations require that the P2 B row is bussed and that there is power supplied to both the P1 and P2 connectors.

- **If you are using a VMEbus, be sure boards are in adjacent slots.**

  The only exception to this is if the backplane is jumpered to propagate the BUS GRANT and INT ACK daisy chains.

- **Check that the RS-232 cables are correctly constructed.**

  In most cases, the documentation accompanying your hardware describes its cabling requirements. One common problem: make sure your serial cable is a null-modem cable, if that is what your target requires.

- **Check the boot ROMs for correct insertion.**

  If the CPU board seems completely dead when applying power (some have front panel LEDs) or shows some error condition (for example, red lights), the boot ROMs may be inserted incorrectly. You can also validate the checksum printed on the boot ROM labels to check for defects in the ROM itself.

- **Press the RESET button if required.**

  Some system controller boards do not reset the backplane on power-on; you must reset it manually.

- **Make sure all boards are jumpered properly.**

  Refer to the target-information reference for your BSP to determine the correct jumper settings for your target and Ethernet boards.

Booting Problems

- **Check the Ethernet transceiver site.**

  For example, connect a known working system to the transceiver and check whether the network functions.

- **Verify Internet addresses.**

  An Internet address consists of a network number and a host number. There are several different classes of Internet addresses that assign different parts of the 32-bit Internet address to these two parts, but in all cases the network number is given in the most significant bits and the host number is given in the least significant bits. The simple configuration described in this chapter assumes that the host and target are on the same network—they have the same network number. (See *VxWorks Network Programmer's Guide: TCP/IP Under VxWorks* for a discussion

of setting up gateways if the host and target are not on the same network.) If the target Internet address is not on the same network as the host, the VxWorks boot program displays the following message:

```
NetROM> tgtreset
```

0x33 corresponds to **errno** 51 (decimal) **ENETUNREACH**. (This is one of the POSIX error codes, defined for VxWorks in **/target/h/errno.h**.)

If the target Internet address is not in **/etc/hosts** (or the NIS equivalent), then the host does not know about your target. The VxWorks boot program receives an error message from the host:

```
host name for your address unknown
Error loading file: status = 0x320001.
```

0x32 is the VxWorks module number for **hostLib** 50 (decimal). The digit "1" corresponds to **S_hostLib_UNKNOWN_HOST**. See the **errnoLib** reference manual entry for a discussion of VxWorks error status values.

- **Verify host file permissions.**

  The target name must be listed in either of the files *userHomeDir*/**.rhosts** or **/etc/hosts.equiv**. The target user name can be any user on the host, but do not use the user name **root** —special rules often apply to it, and circumventing them creates security problems on your host.

  Make sure that the user name you are using on the target has access to the host files. To verify that the user name has permission to read the **vxWorks** file, try logging in on the host with the target user name and accessing the file (for instance, with the UNIX **size** command). This is essentially what the target does when it boots.

  If you have trouble with access permissions, you might try using FTP (File Transfer Protocol) instead of relying on RSH (remote shell). Normally, if no password is specified in the boot parameters, the VxWorks object module is loaded using the RSH service. However, if a password is specified, FTP is used. Sometimes FTP is easier because you specify the password explicitly, instead of relying on the configuration files on the host. Also, some non-UNIX systems do not support RSH, in which case you must use FTP. Another possibility is to try booting using BOOTP and TFTP; see *VxWorks Network Programmer's Guide: File Access Applications*.

- **Check host account .cshrc file.**

  Unless you specify an FTP password in your boot parameters, or include NFS-client support in your VxWorks image, the default VxWorks access to

host-system files is based on capturing file contents through the **rcmd( )** interface to the UNIX host. For user accounts whose default shell is the C shell, this makes it imperative to avoid issuing any output from **.cshrc**. If any of the commands in **.cshrc** generates output, that output can interfere with downloading host files accurately through **rcmd( )**. This problem most often shows up while downloading the VxWorks boot image.

To check whether the **.cshrc** file is causing booting problems, rename it temporarily and try booting VxWorks again. If this proves to be the source of the problem, you may want to set up your **.cshrc** file to conditionally execute any commands that generate standard output. For example, commands used to set up interactive C shells could be grouped at the end of the **.cshrc** and preceded with the following:

```
# skip remaining setup if a non-interactive shell:
if (${?USER} == 0 || ${?prompt} == 0 || ${?TERM} == 0) exit
```

If **noclobber** is set in your **.cshrc**, be sure to un-set or move it to the section that is executed (as shown above) only if there is an interactive shell.

- **Helpful Troubleshooting Tools**

In tracking down configuration problems, the following UNIX tools can be helpful:

ping
: This command indicates whether packets are reaching a specified destination. For example, the following indicates this host is successful sending packets to **phobos**:

```
% ping phobos
phobos is alive
```

ifconfig
: This command reports the configuration of a specified network interface (for example, **ie0** or **le0** on a Sun system). It should report that the interface is configured for the appropriate Internet address and that the interface is up. The following example shows that interface **le0**, whose address is 137.10.1.3, is up and running:

```
% ifconfig -a
le0: flags=63<UP,BROADCAST,NOTRAILERS,RUNNING>
inet 137.10.1.3 netmask ffffff00 broadcast 137.10.1.0
lo0: flags=49<UP,LOOPBACK,RUNNING>
inet 127.0.0.1 netmask ff000000
```

arp -a
: This command displays the "address resolution protocol" tables that map Internet addresses to Ethernet addresses. Your target machine is listed if at

least one packet was transferred from your target to your host. The following example shows **saturn**'s Internet address (92.0.9.54) and Ethernet address (8:10:5:3:a5:c).

```
% arp -a
saturn (92.0.9.54) at 8:10:5:3:a5:c
```

etherfind

This command can be used on many UNIX systems to watch all traffic on a network. You must have **root** access to execute **etherfind**. For example, to monitor traffic between **mars** and **phobos** from a third machine, enter the following:

```
# etherfind between mars phobos
Using interface le0
                                    icmp type
lnth  proto  source   destination  src port  dst port
  60  tcp      mars        phobos      1022     login
  60  tcp    phobos          mars     login      1022
  60  tcp      mars        phobos      1022     login
   ...
```

etherfind displays the packet length, the protocol (for example, TCP, UDP), the source and destination machine names, and the source and destination ports.

netstat

This command displays network status reports. The **-r** option displays the network routing tables. This is useful when gateways are used to access the target. In the following example, this node sends packets for 91.0.10.34 through gateway **vx210**:

```
% netstat -r
Routing tables
Destination    Gateway   Flags   Refcnt  Use    Interface
91.0.10.34     vx210     UG      0       0      le0
```

Target-Server Problems

- **Check Back-End Serial Port**

  If you use a WDB Serial connection to the target, make sure you have connected the serial cable to a port on the target system that matches your target-agent configuration. The agent uses serial channel 1 by default, which is different from the channel used by VxWorks as a default console (channel 0). Your board's ports may be numbered starting at one; in that situation, VxWorks channel one corresponds to the port labeled "serial 2."

- **Verify Path to VxWorks Image**

  The target server requires a host-resident image of the VxWorks run-time system. By default, it obtains a path for this image from the target agent (as recorded in the target boot parameters). In some cases (for example, if the target boots from a local device), this default is not useful. In that situation, use the Core file field in the Create Target Server form (*3.5 Managing Target Servers*, p.72) or the equivalent **-c** option to **tgtsvr** (online *Tornado API Reference*) to specify the path to a host-resident copy of the VxWorks image.

2.10.2 Technical Support

If you have questions or problems with Tornado or with VxWorks after completing the above troubleshooting section, or if you think you have found an error in the software, contact the Wind River customer support organization. Your comments and suggestions are welcome as well. For information about customer support, see *1.6 Customer Services*, p.12.

3 Launcher

3.1 Introduction

This chapter discusses the *Tornado Launcher*, the control panel for Tornado. Once Tornado is configured and targets are set up on your network, all the information you need to connect Tornado tools to a target is in this chapter.

3.2 The Tornado Launcher

The Tornado Launcher is a central control panel that ties together the whole suite of Tornado tools and services. The launcher's mission is to bring together tools and targets; but, as the centerpiece of Tornado, the launcher also provides other services.

Through the launcher, you can

- inspect information about available targets and target servers
- launch any Tornado tool attached to any available target server
- start VxWorks target simulators
- select among available target servers
- create and manage target servers
- install new Tornado components
- consult Internet publications relating to Tornado or VxWorks
- transmit support requests to Wind River, and query their status

The Tornado registry (a daemon that keeps track of all target servers) must be in place on a host at your site before anyone can use Tornado. If the launcher finds no registry, it offers to start one on the current host.[1] For more information on the registry and on other host-configuration issues, see *2. Setup and Startup*.

To start the Tornado Launcher, invoke its name from the UNIX command line or from any shell script or window-manager menu:[2]

```
% launch &
```

Notice the **&** in the preceding example. Because the launcher runs in its own separate graphical window, it normally runs asynchronously from its parent shell.

NOTE: All tools started by the launcher inherit its working directory. You can select other directories when necessary from within each tool, but it is usually convenient to start the launcher from the directory where you expect to do most of your work.

To terminate the launcher, select Quit from the launcher File menu.

The launcher is a convenience, not a straitjacket. If you prefer, you can start Tornado tools and manage target servers directly from a UNIX shell or shell script.

3.3 Anatomy of the Launcher Window

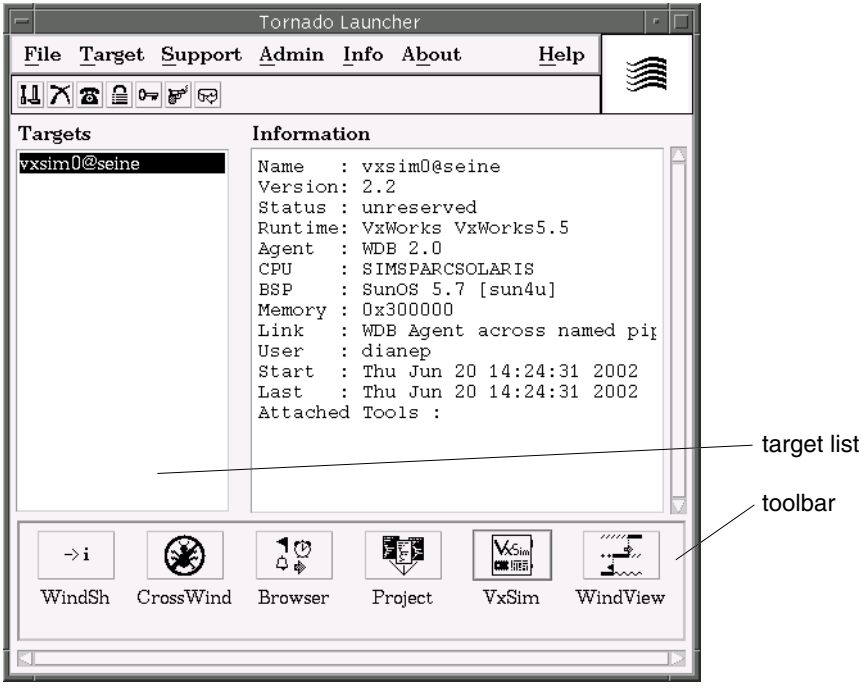
Most of the main launcher window (Figure 3-1) reflects the two main kinds of objects it links together—tools and target servers:

The *target list* shows all target servers currently available in your development network. The list scrolls vertically if its contents exceed the display area.

The *toolbar* has a button for every installed Tornado tool. The toolbar display area scrolls horizontally if its contents exceed the space available. The toolbar illustrated in Figure 3-1 displays the fundamental collection of Tornado tools:

1. By default the launcher has the registry create its database in *installDir***/.wind**. If that directory is not writable, the database is created in *homeDir***/.wind**.
2. If you have any trouble with this command, make sure that your host development environment is correctly configured, as described in *2.3 The Tornado Host Environment*, p.20.

Figure 3-1 **Tornado Launcher Main Window**



WindSh
: The Tornado shell, an interactive window to the target that includes both a C interpreter and a Tcl interpreter. The shell is described in detail in *7. Shell*.

CrossWind
: The Tornado graphical debugger, a powerful source-level debugger that provides both graphical and command-driven access to target programs. *9. Debugger* provides full documentation.

Browser
: A viewer to explore and monitor target system objects, described in *8. Browser*.

Project
: A graphical facility for managing application files, configuring VxWorks, and building applications and system images. See *4. Projects*.

VxSim
: The VxWorks target simulator. It is a port of VxWorks to the host system that simulates a target operating system. No target hardware is required. See the

Tornado Getting Started Guide for a introductory discussion of target simulator usage, and *4. Projects* for information about its use as a development tool.[3]

WindView
: The Tornado logic analyzer for real-time software. It is a dynamic visualization tool that provides information about context switches, and the events that lead to them, as well as information about instrumented objects. See the *WindView User's Guide*.[4]

3.4 Tools and Targets

One way to think of the Tornado launcher is as a central plugboard which allows you to connect any Tornado development tool to any networked target.

Figure 3-2 illustrates this concept. The launcher allows you to use targets just as easily regardless of their nature or their physical connection. Figure 3-2 shows several common variations on connections between a tool and a target:

– Tool 1 is connected to a target on the local Ethernet subnet.

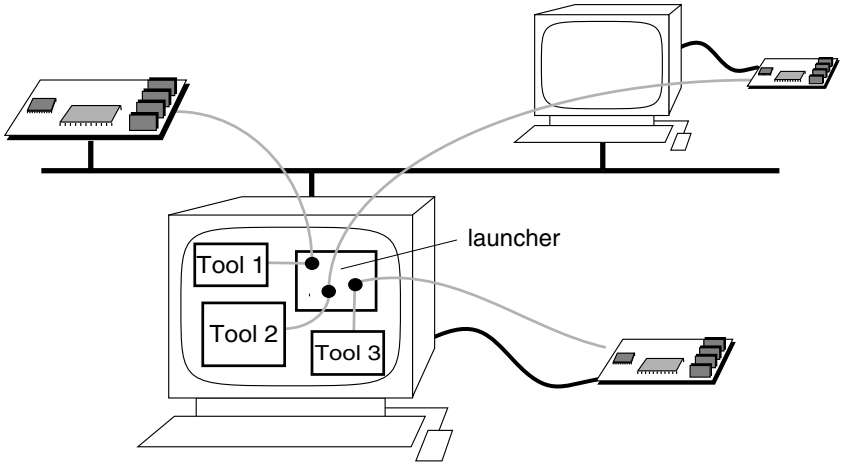
– Tool 2 is connected over the local Ethernet to a target that is physically attached to a remote host.

– Tool 3 is connected to a target that communicates directly with the local host over a serial line.

All this is possible thanks to the *target server*, a dedicated daemon which represents each development target to the development network. All details related to physical connectivity are handled by the target server. Someone must configure the target communications initially (see *2.4 Setting Up the Default Target Hardware*, p.24), but thereafter the target is immediately available to any authorized user on the local network, with no further cabling or configuration.

3. Tornado includes a version of the VxSim target simulator that runs as a single instance per user, without networking support (optional products such as VxMP are not available for this version). The full-scale version supports multiple-instance use and includes networking support. It is available as an optional product.
4. Tornado includes a version of WindView designed solely for use with the VxWorks target simulator. WindView is also available as an optional product for all supported target architectures.

Figure 3-2 **The Launcher as Network-Wide Plugboard**



For reference information about the target server, see the entry for **tgtsvr** in the online *Tornado API Reference* (Help>Manuals Contents>Tornado Reference>Tornado Tools>tgtsvr).

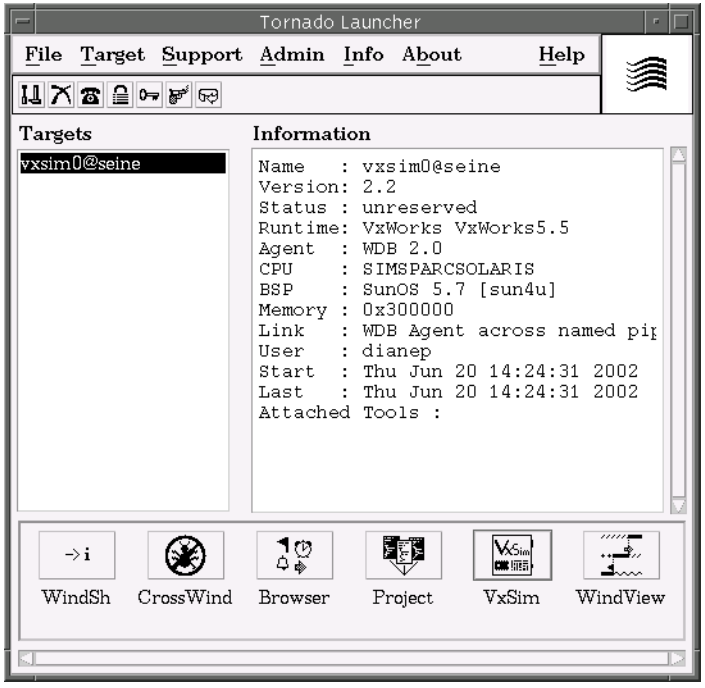
3.4.1 Selecting a Target Server

To select a target server, click on any of the server names in the target list. The launcher highlights the selected target name, and fills the Information panel with a scrollable description of the target configuration and target server. Figure 3-3 illustrates a launcher with a target server selected.

If no target servers are listed, or if none of the target servers listed represent the target you need, see *3.5.1 Configuring a Target Server*, p.73 below.

If you make a mistake, or if you wish to select another target, simply click on another target-server name. Any tools that you have already launched remain connected to the previous target (the plugboard analogy does not extend that far).

Figure 3-3 **Launcher with a Selected Target**



The information panel displays the following information about the selected target:

Name
: A unique string identifying the target server, which matches the selected entry in the target list. Servers are shown as *target@serverhost*, where *target* is an identifier (frequently the network name) for the target device itself, and *serverhost* is the network name of the host where the target server is running.

Version
: The target-server version number.

Status
: This field indicates whether a target is locked (restricted to the user ID that started the server), reserved (for the user shown below in the User field) or unreserved. Anyone[5] may connect to an unreserved target.

5. You can also restrict your target servers to permit connections only by a particular list of users; see *3.5.2 Sharing and Reserving Target Servers*, p.82.

Runtime
The name and version number of the operating system running on the target.

Agent
The name and version number of the agent program running on the target.

CPU
A string identifying the CPU architecture (and possibly other related information, such as whether this is a real target or a simulated one).

BSP
The name and version number of the Board Support Package linked into the run-time.

Memory
The total number of bytes of RAM available on this target.

Link
The physical connection mechanism to the target.

User
The user ID of the developer who launched this target server, or of the user who reserved it most recently.

Start
A timestamp showing when this target server was launched.

Last
The last time this target server received any transaction request.

Attached Tools
A list of all the tools currently attached to this target server. The list includes all Tornado tools attached to this target by any user on the network, not just your own tools.

3.4.2 Launching a Tool

Once you have selected a target server, click once on any button in the toolbar to launch a tool on that target. You can launch as many instances of a tool as you like, even attached to the same target. For instance, you may find it convenient to have one instance per application task of CrossWind, or to run different shells for different kinds of interaction.

You can also launch many of the Tornado tools from a UNIX shell (or shell script), specifying the target name as an argument. See the chapter that describes each tool for more information.

3.5 Managing Target Servers

The target-server architecture of Tornado permits great flexibility, but also introduces a number of housekeeping details to manage situations like the following:

- the target you need to use does not have a server running
- other developers keep interfering with your target over the net
- you want some other developers to have access to your target, but not everyone

The Target menu in the Tornado Launcher offers commands that allow you to manage these chores and related details to do with target servers. The small buttons immediately below the menu bar provide quick access to the same commands.

The following list describes each button and Target menu command:

Button	Menu	Description
	Create…	Define and start up a new target server. See *3.5.1 Configuring a Target Server*, p.73.
	Unregister	Remove the selected target server from the Tornado registry's list of available servers. *Do not use this command routinely.* Under most circumstances, the registry automatically removes the entry for any target server that has been killed (for example, due to a host system crash). This command can also be used to do so. The registry honors the Unregister command only if the server does not respond to the registry. **CAUTION:** Even if a target server is not responsive, it is not always appropriate to unregister it; the server may simply be too busy to respond, or a heavy network load may be interfering. The Unregister command reminds you of this possibility and requests confirmation before it takes effect. Make sure the server is really gone before you unregister it.
	Reattach	Reconnect the selected target server to the underlying target. This command is rarely necessary, because target servers attempt to reconnect automatically when required. Use this command after turning on or connecting a target that has been unavailable, if you want to reattach a running server explicitly (rather than by running a tool).

	Reserve	Restrict a target to your own use, or share it with others. See *3.5.2 Sharing and Reserving Target Servers*, p.82.
	Unreserve	Share a target with others. See *3.5.2 Sharing and Reserving Target Servers*, p.82.
	Kill	Kill the currently selected target server. **CAUTION:** Close any tool sessions that use a particular target before you kill that target server. Killing a target server does not immediately destroy any attached tools, but the tools lose the ability to interact with the target. There is no way to reconnect a new target server to such orphaned tool sessions.
	Reboot	Re-initialize the selected target server and reboot its target.

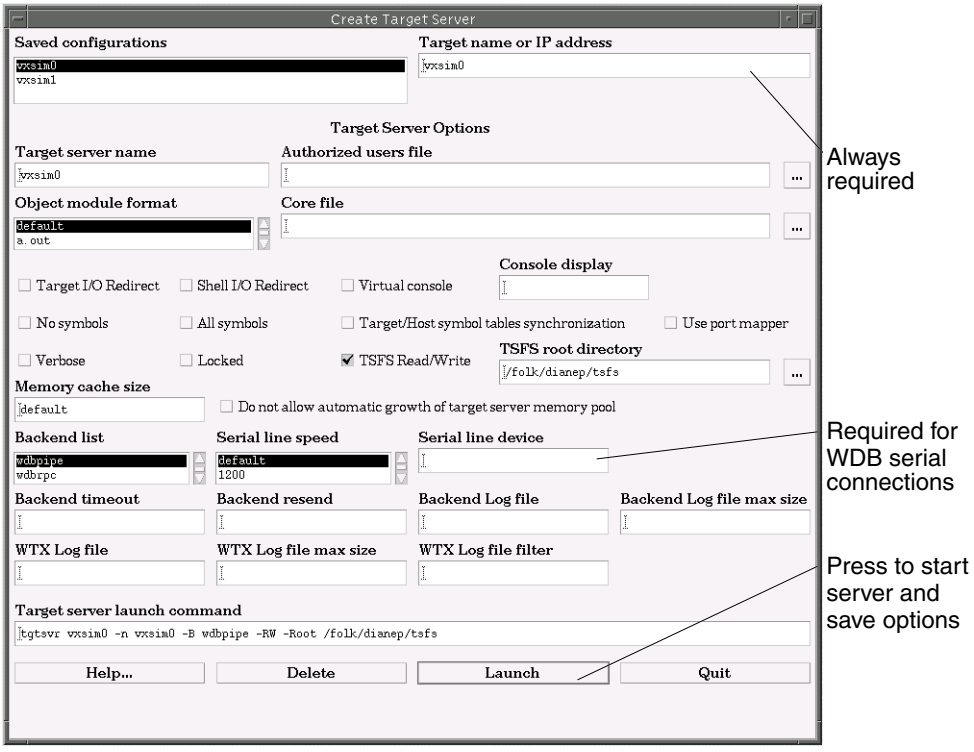
3.5.1 Configuring a Target Server

To use a new target, you must first ensure the host and target are connected properly. The details are unique to each target, but *2.4.2 Networking the Host and Target*, p.25 discusses some of the issues that are frequently involved. Your BSP also contains a target-information reference that describes what to do for that particular target. See Help>Manuals contents>BSP Reference.

To configure and launch a target server, select Create from the Target menu, or press the launcher's  button. The launcher displays the form shown in Figure 3-4. Many configuration options are available, but you can often skip all the optional parameters, specifying only the target name (and perhaps the serial device, if your target agent is configured for the WDB serial protocol).

Each time you specify a configuration option, the Target server launch command box near the bottom of the form is updated automatically to show the **tgtsvr** command options that capture your chosen configuration. (For text fields, the command line is updated when you select another field or press **RETURN**.)

Figure 3-4 **Form: Create Target Server**



The **tgtsvr** command is the underlying command that runs in the background for each target server as a UNIX daemon. The text in the Target server launch command box can be edited. Its display has the following uses:

- You can copy the text displayed, and insert it in any UNIX shell script to launch a target server with this configuration automatically.
- You can use the command-line display to explore the meanings of server options interactively, in conjunction with the **tgtsvr** reference documentation (located in the online *Tornado API Reference*).
- You can type **tgtsvr** options directly in this box. This allows you to add options that are not generated by the dialog boxes, such as those required for third-party back-ends.

To start a target server and save your server configuration, press the Launch button at the bottom of the Create Target Server form.

If a server does not respond when you select it, kill it () and try turning on the Verbose toggle near the middle of the Create Target Server form to display diagnostic messages when you start it again.

Simple Server Configuration for Networked Targets

For targets with network connectivity, only one field is required. Fill in the IP address or network name for the target, in the box headed Target name or IP address. After filling this in, you can launch a server immediately. The launcher saves each configuration automatically (identified with the target-server name); thus, you can retrieve a server's configuration later to add more options.

Simple Server Configuration for WDB Serial Targets

If your target agent is configured for the WDB serial protocol, you must specify what UNIX device name is connected to the target, in the Serial line device box (entering the device name automatically selects wdbserial as the back end in the Backend list field). You must also fill in a name for the target server in the Target name or IP address box; in this case, the name is completely arbitrary, and is used only to identify your target server.

Specifying the serial line speed is not required if you use the default speed of 9600 bps. However, it is best to use the fastest possible line speed when controlling your target over serial lines. Select the fastest speed your target hardware supports from the scrolling list headed Serial line speed. (The target agent must be compiled with the same speed selected; see *Configuration for Serial Connection*, p.160.)

Saved Configurations

Each time you press the Launch button, the launcher saves the server configuration. The configuration name is the same name used to register the target server: the contents of the Target name or IP address box, or the name specified under Target server name, if you use this box to define a different name for your server.[6]

6. Data for each saved configuration is stored in a file with the same name as the configuration, in the directory **.wind/tgtsvr** under your home directory.

The following controls are available to manage saved configurations:

Saved configurations scrolling list
: Select a configuration by clicking on a server name from this list (top left of the form). The fields of the Create Target Server form are filled in as last specified for that server name. (The last configuration you were working with is selected automatically when you open the form.)

Delete button
: Discard any configuration you no longer need by first selecting the configuration name, then pressing this button (in the row at the bottom of the form).

Target-Server Action Buttons

The command buttons at the bottom of the Create Target Server form perform the following functions (see Figure 3-4):

Help
: Display reference information for **tgtsvr**, using your default browser.

Delete
: Delete the selected configuration from the Saved configurations list.

Launch
: Start a target server using the currently specified configuration, and close the Create Target Server form.

Quit
: Discard the Create Target Server form without launching a server or saving.

Target-Server Configuration Options

This section describes all the configuration options you can specify in the Create Target Server form(Figure 3-4), in the order they appear (left to right and top to bottom).

Saved configurations
: Select a saved configuration by clicking on a server name from this list.

Target name or IP address
: The network name of the target hardware for networked targets, or an arbitrary target-server name for other targets. You must always specify this field.

Target server name

To give the target server its own name (distinct from the network name of the target), specify the name here. If you do not fill in this box, the target server is known by the same name as the target. Use this field to distinguish alternative configurations of a single target.

For serial targets, this box is never necessary, because the required Target name or IP address entry already specifies an arbitrary name for the server.

Authorized users file

To restrict this target server to a particular set of users, specify the name of a file of authorized user IDs here. If you do not specify an authorized-users file, any user on your network may connect to the target whenever it is not reserved. See *3.5.2 Sharing and Reserving Target Servers*, p.82 for more discussion of the authorized-users file.

Object module format

By default, the target server deduces the object-module format by inspecting the host-resident image of the run-time system. You can disable this by explicitly selecting an object format from this list.

Core file

A path on the host to an executable image corresponding to the software running on the target. This box is optional because the target agent reports the original path from where the executable was loaded to the server. However, if the file is no longer in the same location on the host as when your target downloaded it (or if host and target have different views of the file system), you can use this box to specify where to find the image on the host.

For example, if you are using a target programmed with a **vxWorks_rom.hex**, **vxWorks_romCompressed.hex**, or any other on-board VxWorks image, you must use the core file option to identify the location of a **vxWorks** file as the core file; otherwise the target server will not be able to identify the target symbols.

Target I/O Redirect

Turn on this toggle to redirect the target's standard input, output, and error. If Virtual console is selected, target I/O is redirected to the console window.

Shell I/O Redirect

Turn on this toggle to start a console window into which the target shell's standard input, output, and error will be directed. (This option is only available when Virtual console is selected.)

Virtual console
: Turn on this toggle to display the virtual console for this target server (a dedicated **xterm** where any output or input through virtual I/O channels takes place).[7] Examples in this manual that involve input and output streams from target programs assume the target server is running with this option set. See *Virtual I/O*, p.11 for a discussion of the role of the virtual console.

Console Display
: The name of an X Window System display to use as a target-server virtual console. Fill in this box with the display server name and screen number, in the usual X Window System format *hostname*:*N*. If the Display toggle is turned on but this box is not filled in, the virtual console appears on display 0 of the same host that runs this target server. (The alternative display must grant authorization for your host to use it; see your X Window System documentation.)

No symbols
: Turn on this toggle to avoid initial loading of the symbol table maintained on the host by the target server.

All symbols
: Turn on this toggle to include local symbols as well as global symbols in the target symbol table. The default is to include only global symbols, but during development it can be useful to see all symbols.

Target/Host symbol table synchronization
: Turn on this toggle to synchronize target and host symbol tables. Synchronizing the two symbol tables can be useful for debugging. The symbol table synchronization facility must be included in the target image to select this option. For more information see *4.4.3 Configuring VxWorks Components*, p.134 and the reference entry for **symSyncLib**.

  To use symbol and module synchronization, the **WIND_REGISTRY** environment variable must be set to a host name or an IP address that the VxWorks target can access. It *cannot* be left as the default value, **localhost**.

Use portmapper
: Turn on this toggle to register a target server with the RPC portmapper. While the portmapper is not needed for Tornado 2.2, this option is included for development environments in which both Tornado 2.2 and Tornado 1.0.1 are in use.

7. You can also create a virtual console from any Tornado tool using Tcl, with **wtxConsoleCreate**. See the online *Tornado API Reference: WTX TCL API*.

When both releases are in use, the portmapper must be used on the following:

- Any host running a Tornado 2.2 registry that will be accessed by any host running Tornado 1.0.1.
- Any host running a Tornado 2.2 target server that will be accessed by any host running Tornado 1.0.1.

To use the portmapper when either a Tornado registry or target server is started from the command line, the **-use_portmapper** option must be included. See the registry (**wtxregd**) and target server (**tgtsvr**) reference documentation in the online *Tornado API Reference: Tornado Tools Reference* for more information.

Verbose

Turn on this toggle to display target-server status information and error messages in a dedicated window.[8]

Use this display for troubleshooting. The same status and error information is saved in **~/.wind/launchLog.***servername*.

Locked

Turn on this toggle to restrict this target server to your own user ID. If you do not turn on this toggle, any authorized user may use or reserve the server after you launch it.

TSFS Read/Write

This is the default. Click the box to change this option to read only. The default allows you to run WindView. Because read/write also allows other users to access your host file system, you may with to set the TSFS option for your target server to read only when you are not using WindView.

The TSFS provides the most convenient way to boot a target over a serial connection (see *2.6.7 Booting a Target Without a Network*, p.55).

TSFS Root directory

Type the path to the files you want the target to be able to access through the target server in the Target Server File System root box. This is where WindView log files are saved. For example:

```
/usr/windview/logfiles
```

8. To disable the automatic display of log files by the launcher, insert "**set noViewers 1**" in your **~/.wind/launch.tcl** initialization file.

If you use the TSFS for booting a target, it is recommended that you use the base Tornado installation directory (*installDir*) or the root directory (*/*). If you do not do so, you must use the Core File configuration option to specify the location of the VxWorks image (see *Core file*, p.77).

Memory cache size

Specify the size of the target-memory cache (either in decimal or hexadecimal). The target server maintains a cache on the host system, in order to avoid excessive data-transfer transactions with the target. By default, this cache can grow up to a size of 1 MB.

A larger maximum cache size may be desirable if the memory pool used by host tools on the target is very large, because transactions on memory outside the cache are far slower. See *Scaling the Target Agent*, p.161 for more information about the memory pool managed by the server on the target.

Disable Automatic Growth of Target Server Memory Pool

By default, when there is not enough memory in the WDB pool to satisfy an allocation request from the target server, the WDB pool automatically grows to accommodate the request. You can disable automatic growth by checking the box, or by typing **-noG** or **-noGrowth** in the option box.

Backend list

If your BSP requires a special communications protocol, select the communications protocol here. The default, wdbrpc, is suitable for targets with IP connectivity. The standard back ends are described in Table 3-1; see also *4.7 Configuring the Target-Host Communication Interface*, p.156.

Table 3-1 **Communications Back Ends for Target Server**

Back End Name	Description
default	Initially selected; implicitly selects wdbrpc.
wdbrpc	Tornado WDB protocol. This back end is the default. It is the most frequently used back end, and supports any kind of IP connection (for example, Ethernet). Serial hardware connections are supported by this back end if your host has SLIP. On a serial connection, this back end supports either system-level or task-level views of the target, depending on the target-agent configuration.
wdbserial	A version of the WDB back end specialized for serial hardware connections; does not require SLIP on the host system. This back end supports either system-level or task-level views of the target, depending on the target-agent configuration.

Table 3-1 **Communications Back Ends for Target Server**

Back End Name	Description
netRom	A back end that communicates over a proprietary communications protocol for NetROM.
wdbpipe	WDB Pipe back end. The back end for VxWorks target simulators. It supports either system-level or task-level views of the target, depending on the configuration of the target agent.
loopback	Testing back end. This back end is not useful for connecting to targets; it is intended only to exercise the target-server daemon during tests.

Serial line speed

If you choose the wdbserial back end, use this scrolling list to specify the line speed (in bits per second) that your target uses to communicate over its serial line. The default speed is 9600 bps; use the highest possible speed available, in order to maximize the host tools' access to target information.

When you change the line speed, you must also re-compile the target agent with **WDB_TTY_BAUD** defined to the same speed (*Configuration for Serial Connection*, p.160).

Serial line device

If you choose the wdbserial back end, use this text box to specify the serial device on your host that is connected to the target. The default serial device is **/dev/ttya**.

Backend Timeout

How many seconds to wait for a response from the agent running on the target system (the default is 3 seconds). This option is supported by the standard wdbrpc, wdbserial, and netrom back ends, but may not have an effect on other back ends.

Backend Resend

How many times to repeat a transaction if the target agent does not appear to respond the first time. This option is supported by the standard wdbrpc, wdbserial, and netrom back ends, but may not have an effect on other back ends.

Backend log file

Log every WDB request sent to the target agent in this file. Back ends that are not based on WDB ignore this option. As with the Verbose toggle, a dedicated window appears to display the log.

Backend log file max size
: The maximum size of the backend log file, in bytes. If defined, the file is limited to the specified size and written to as a circular file. That is, when the maximum size is reached, the file is rewritten from the beginning. If the file initially exists, it is deleted. This means that if the target server restarts (for example, due to a reboot), the log file will be reset.

WTX Log file
: Log every WTX request sent to the target server in the specified file. If the file exists, log messages will be appended (unless a maximum file size is set in WTX Log file max size, in which case it is overwritten).

WTX Log file max size
: The maximum size of the WTX log file, in bytes. If defined, the file is limited to the specified size and written to as a circular file. That is, when the maximum size is reached, the file is rewritten from the beginning. If the file initially exists, it is deleted. This means that if the target server restarts (for example, due to a reboot), the log file will be reset.

WTX Log file filter
: Use this field to limit the amount of information written to a WTX log file. Enter a regular expression designed to filter out specific WTX requests. Default logging behavior may otherwise create a very large file, as all requests are logged.

3.5.2 Sharing and Reserving Target Servers

A target server may be made available to the following classes of user:

– the user who started the server

– a single user, who may or may not have started the server

– a list of specified users

– any user[9]

9. Strictly speaking, there is another layer of authorization defining who is meant by "any user". The file *installDir*/**.wind/userlock** is a Tornado-wide authorization file, used as the default list of authorized users for any target server without its own authorized-users file. The format of this file is the same format described below for individual target-server authorization files.

When a target server is available to anyone, its status (shown in the Information panel of the main launcher window; see Figure 3-3) is *unreserved*. Any user can attach a tool to the target, and any user can also restrict its use.

When you configure a target server, you can arrange for the server to be exclusively available to your user ID every time you launch it, by clicking the Lock toggle in the Create Target Server form. See *3.5.1 Configuring a Target Server*, p.73. Target servers launched this way have the status *locked*.

If a target server is not locked by its creator, and if no one else has reserved it, you can reserve the target server for your own use: click on Target>Reserve, or on the  launcher button. The target status becomes *reserved* until you release the target with the Unreserve command (). Unreserve on a target that is not reserved has no effect, nor does Unreserve on a target reserved or locked by someone else.

This simple reserve/unreserve locking mechanism is sufficient for many development environments. In some organizations, however, it may be necessary to further restrict some targets to a particular group of users. For example, a Q/A organization may need to ensure certain targets are used only for testing, while still using the reserve/unreserve mechanism to manage contention within the group of testers.

To restrict a target server to a list of users, create a list of authorized users in a file. The format for the file is the simplest possible: one user name per line. The user names are host sign-on names, as used by system files like **/etc/passwd** (or its network-wide equivalent). You can also use one special entry in the authorization file: a plus sign **+** to explicitly authorize any user to connect to the target server. (This might be useful to preserve the link between a target server and an authorization file when access to that target need only be restricted from time to time.)

To link an authorization file to a target server, specify the file's full pathname in the Authorized users file box of the Create Target Server screen (see Figure 3-4).

3.6 Tornado Central Services

Because the launcher is the control panel for Tornado, it performs a number of support functions as well as its central mission of connecting tools and targets.

Through the launcher menu bar, you can do the following:

- Authorize other developers at your site to use Tornado
- Install new Tornado product components
- Submit, manage, and query support requests to Wind River
- Point your World-Wide Web browser to Tornado- and VxWorks- related news and information on the Web

3.6.1 Support and Information

The About menu has a single command, Tornado, which displays version information for Tornado. This menu appears in all Tornado graphical tools.

The launcher's Support and Info menus are a gateway to Wind River's support, training, and sales services. See *1.6 Customer Services*, p.12 for more information on these launcher facilities.

3.6.2 Administrative Activities

The Admin menu provides a number of conveniences to automate Tornado administrative chores to the extent possible. The commands in this menu cover installing updates or optional products and managing your site's global authorization file for Tornado.

Install CD
: Begins by prompting you to mount a Tornado CD-ROM. Locate your installation keys and mount the CD-ROM as explained in the *Tornado Getting Started Guide*. The launcher runs the installation program for you.

FTP WRS
: Wind River maintains a small archive of auxiliary software and useful information available over the Internet by FTP. Click on this command to connect to the Wind River FTP server. Follow the usual conventions for anonymous FTP transfers: log in as **anonymous**, and provide your e-mail address at the **password:** prompt.

Authorize
: The Authorize command brings up an editor[10] on the file *installDir***/.wind/userlock**. This file controls overall access to Tornado host tools at your site. This file employs the same simple conventions described in

10. The editor specified in your **EDITOR** environment variable, or **vi**.

3.5.2 Sharing and Reserving Target Servers, p.82 for a file to restrict a target server to a list of users: the character **+** to indicate that all users are authorized, or the sign-on names of authorized users, one on each line.

3.7 Tcl: Customizing the Launcher

NOTE: If you are not familiar with Tcl, you may want to postpone reading this section (and other sections in this book beginning with "Tcl:") until you have a chance to read *C. Tcl* (and perhaps some of the Tcl references recommended there).

An important reference for these examples, even if you are familiar with Tcl, is the GUI Tcl Library reference available online from Help>Manuals contents>Tornado API Reference. It describes the building blocks for the user interface (GUI) shared by the Tornado tools.

All Tornado tools can be altered to your needs (and to your taste) by adding your own Tcl code. This section has a few examples of launcher customization.

When you consider modifications to the launcher, you may want to read related code in the standard form of the launcher. The Tcl code implementing the launcher is organized as follows:

*installDir***/host/resource/tcl/Launch.tcl**
The main launcher implementation file.

*installDir***/host/resource/tcl/app-config/Launch/01*.tcl**
Supporting procedures and definitions (grouped into separate files by related functionality) for the launcher.

*installDir***/host/resource/tcl/app-config/all/host.tcl**
Defaults for global settings; may be redefined for specific host types.

*installDir***/host/resource/tcl/app-config/all/***hostType***.tcl**
Host-specific overrides for global settings.

3.7.1 Tcl: Launcher Initialization File

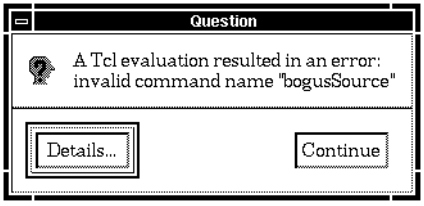
When the launcher starts up, it looks for a file called **.wind/launch.tcl** in your home directory. If that file is present, its contents are read with the Tcl **source** command

before the launcher puts up its initial display. Use this file to collect your custom modifications, or to incorporate shared customizations from a central repository of Tcl extensions at your site.

3.7.2 Tcl: Launcher Customization Examples

When you begin experimenting with any new system (or language), errors are to be expected. Any error messages from your launcher Tcl initialization code are captured by the launcher, and a summary of the error is displayed in a window similar to Figure 3-5.

Figure 3-5 **Tcl Error Display**



To see the full Tcl error display, click on the Details button in the error display; click Continue to dismiss the display.

The examples in this section use the Tcl extensions summarized in Table 3-2. For detailed descriptions of these and other Tornado graphical building blocks in Tcl, see Help>Manuals contents>Tornado API Reference>GUI Tcl Library.

Table 3-2 **Tornado UI Tcl Extensions Used in Launcher Customization Examples**

Tcl Extension	Description
noticePost	Display a popup notice or a file selector.
menuButtonCreate	Add a command to an existing menu.

Re-Reading Tcl Initialization

Because the launcher has no direct command-line access to Tcl, it is not as convenient as other tools (such as WindSh or CrossWind) for experimentation with Tcl extensions. The following example makes things a little easier: it adds a

command to the File menu that reloads the **.wind/launch.tcl** file. This avoids having to Quit the launcher and invoke it again, every time you change launcher customization.

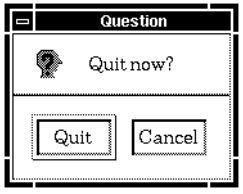
Example 3-1 **Tcl Reinitialization**

```
# "Reinitialize" command for Launcher.
# Adds item to File menu; calls Tcl "source" primitive directly.

menuButtonCreate File "Re-Read Tcl" T {
    source ~/.wind/launch.tcl
}
```

Quit Launcher Without Prompting

When you select the Quit command from the launcher File menu, the launcher displays the short form shown in Figure 3-6 to make sure you selected Quit intentionally.

Figure 3-6 **Form: Quit Confirmation**



This sort of safeguard is nearly universal in graphical applications, but some people find it annoying. If you would prefer to take your chances with an occasional unintended shutdown, for the sake of having the launcher obey you unquestioningly, this example may be of interest. It shows how to redefine the Quit command to shut down the launcher without first displaying a query.

To discover what procedure implements the Quit command, examine the launcher definitions in *installDir*/**host/resource/tcl/Launch.tcl**. Searching there for the string "Quit" leads us to the following **menuButtonCreate** invocation, which shows that the procedure to redefine is called **launchQuit**:

```
menuButtonCreate   File  Quit   Q  {launchQuit}
```

Example 3-2 **Alternate Quit Definition**

The following redefinition of the **launchQuit** procedure eliminates the safeguard against leaving the launcher accidentally:

```
##############################################################################
#
# launchQuit - abandon the launcher immediately
#
# This routine is a replacement for the launchQuit that comes with the
# launcher; it runs when Quit is selected from the File menu in place of
# the standard launchQuit, to avoid calling a confirmation dialog.
#
# SYNOPSIS:
#   launchQuit
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc launchQuit {} {
    exit
}
```

An Open Command for the File Menu

Because editing files is a common development activity, it may be useful to invoke an editor from the launcher. This example defines a File>Open command to run the editor specified by the **EDITOR** environment variable. The example is based on the file selector built into the **noticePost** Tcl extension.

The code in this example collects the launcher initialization (adding commands to the File menu, both for this example and for Example 3-1) in an initialization procedure as recommended in *D. Coding Conventions*. In the example, the launcher executes **launchExtInit**, which adds entries to the File menu. Of these two new entries, Open calls **launchFileOpen**, which in turn calls **launchEdit** if the user selects a file to open.

Example 3-3 **Open Command and Customized File Menu Initialization**

```
##############################################################################
#
#
# launchExtInit - collects personal launcher initialization
#
# This routine is invoked when the launcher begins executing, and collects
# all the initialization (other than global and proc definitions)
# defined in this file.
```

```
#
# SYNOPSIS:
#   launchExtInit
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc launchExtInit {} {

    # "Reinitialize" command for Launcher.
    # Adds item to File menu; calls Tcl "source" primitive directly.

    menuButtonCreate File "Re-Read Tcl" T {
        source ~/.wind/launch.tcl
    }

    # Add "Open" command to File menu

    menuButtonCreate File "Open..." O {
        launchFileOpen              ;# defined in launch.tcl
    }
}

##############################################################################
#
#
# launchFileOpen - called from File menu to run an editor on an arbitrary
file
#
# This routine supports an Open command added to the File menu.  It prompts
# the user for a filename; if the user selects one, it calls launchEdit to
# edit the file.
#
# SYNOPSIS:
#   launchFileOpen
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc launchFileOpen {} {
    set result [noticePost fileselect "Open file" Open "*"]
    if {$result != ""} {
        launchEdit $result
    }
}

##############################################################################
#
#
# launchEdit - run system editor on specified file
#
# This routine runs the system editor (as specified in the environment
```

```
# variable EDITOR, or vi if EDITOR is undefined) on the file specified
# in its argument.
#
# SYNOPSIS:
#   launchEdit fname
#
# PARAMETERS:
#   fname: the name of a file to edit
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc launchEdit {fname} {

    # we need to examine environment variables

    global env

    if { ([file readable $fname] && ![file isdirectory $fname]) ||
         ([file writable [file dirname $fname]] && ![file exists $fname])
       } then {

           # We have an editable file
           # Use the EDITOR environment variable, with vi default

           if [info exists env(EDITOR)] {
               set editor $env(EDITOR)
           } else {
               set editor vi
           }

           if [string match "emacsc*" $editor] {

               # looks like emacsclient.  Don't run an xterm; just put this
               # in the background.

               exec $editor $fname &
           } else {

               # Run an xterm with the editor in it.

               exec xterm -e $editor $fname &
           }
       } else {

           # fname was unreadable or a directory

           noticePost info "Cannot open: <<$fname>>"
       }
}

###############################################################################
#
#
```

```
# launch.tcl - initialization for private extensions to launcher
#
# The following line executes when the launcher begins execution; it
# calls all private launcher extensions defined in this file.
#

launchExtInit
```

4

Projects

4.1 Introduction

The project facility is a key element of the Tornado development environment. It provides graphical and automated mechanisms for creating applications that can be downloaded to VxWorks, for configuring VxWorks with selected features, and for creating applications that can be linked with a VxWorks image and started when the target system boots. The project facility provides mechanisms for:

- Adding application initialization routines to VxWorks.
- Organizing the files that make up a project.
- Grouping related projects into a workspace.
- Customizing and scaling VxWorks.
- Defining varied sets of build options.
- Building applications and VxWorks images.
- Downloading application objects to the target.

NOTE: For a tutorial introduction to the project facility and its use with the integrated version of the VxWorks target simulator and other Tornado tools, see the *Tornado Getting Started Guide*.

WARNING: Use of the project facility for configuring and building applications is largely independent of the methods used prior to Tornado 2.x. (These methods included manually editing the configuration files **config.h** or **configAll.h**, while the project tool uses **.cdf** files). The project facility provides the recommended and simpler means for configuring and building, although the configuration file method may still be used (see *5. Command-Line Configuration and Build*). To avoid confusion and errors, the two methods should rarely be used together for the same project.

One exception is for any configuration macro that is not accessible through the project facility GUI (which may be the case, for example, for some BSP driver parameters). You can use a Find Object dialog box to determine if a macro is accessible or not (see *Finding VxWorks Components and Configuration Macros*, p.136). If it is not accessible through the GUI, a configuration file must be edited, and the project facility will implement the change in the subsequent build.

The order of precedence for determining configuration is (in descending order):

project facility
config.h
configAll.h

For any macro that is exposed through the project facility GUI, changes made after creation of a project in either of the configuration files will not appear in the project.

A second exception may be building a project based on a BSP. If you have customized your BSP by modifying **config.h** and other configuration files, you can convert it to a project and combine it with your application in the project facility. See *5.7 Building Projects From a BSP*, p.214.

In general, changes to header files in the BSP or the BSP makefile are only carried over to projects by recreating the projects. However, changes to **.c** files are automatically picked up by existing projects.

Terminology

There are several key terms that you must understand before you can use the project facility documentation effectively:

Downloadable application
A downloadable application consists of one or more relocateable object modules,[1] which can be downloaded and dynamically linked to VxWorks, and then started from the shell or debugger. A novel aspect of the Tornado

development environment is the dynamic loader, which allows objects to be loaded onto a running system. This provides much faster debug cycles compared with having to rebuild and re-link the entire operating system. A downloadable application can consist of a single file containing a simple "hello world" routine, or a complex application consisting of many files and modules that are partially linked as a single object (which is created automatically by the project facility as *projectName*.**out**).

Bootable application

A bootable application consists of an application linked to a VxWorks image. The VxWorks image can be configured by including and excluding components of the operating system, as well as by resetting operating system parameters. A bootable application starts when the target is booted.

Project

A project consists of the source code files, build settings, and binaries that are used to create a downloadable application or a bootable application. The project facility provides a simple means of defining, modifying, and maintaining a variety of build options for each project. Each project requires its own directory.

When you first create a project, you define it as either a downloadable application or a bootable application. In this context, custom-configured VxWorks images can be considered bootable applications.

Workspace

A workspace is a logical and graphical "container" for one or more projects. It provides you with a useful means for working with related material, such as associating the downloadable application modules, VxWorks images, and bootable applications that are developed for a given product; or sharing projects amongst different developers and products; and so on.

Component

A component is a VxWorks facility that can be built into, or excluded from, a custom version of VxWorks or a bootable application. Many components have parameters that can be reset to suit the needs of an application. For example, various file system components can be included in, or excluded from, VxWorks; and they each include a parameter that defines the maximum number of open files.

---

1. The text and data sections of a relocateable object module are in transitory form. Because of the nature of a cross-development environment, some addresses cannot be known at time of compilation. These sections are modified (*relocated* or *linked*) by the Tornado object-module loader when it inserts the modules into the target system.

Toolchain

A toolchain is a set of cross-development tools used to build applications for a specific target processor. The default toolchains provided with Tornado are based on the GNU preprocessor, compiler, assembler, and linker (see the *GNU ToolKit User's Guide*) except for the ColdFire architecture. ColdFire uses the Diab toolchain. Diab is also available as an optional product for all architectures except Pentium and 68K. In addition, many third-party toolchains are also available. The tool options are exposed to the user through various elements of the project facility GUI.

BSP

A Board Support Package (BSP) consists primarily of the hardware-specific VxWorks code for a particular target board. A BSP includes facilities for hardware initialization, interrupt handling and generation, hardware clock and timer management, mapping of local and bus memory space, and so on.

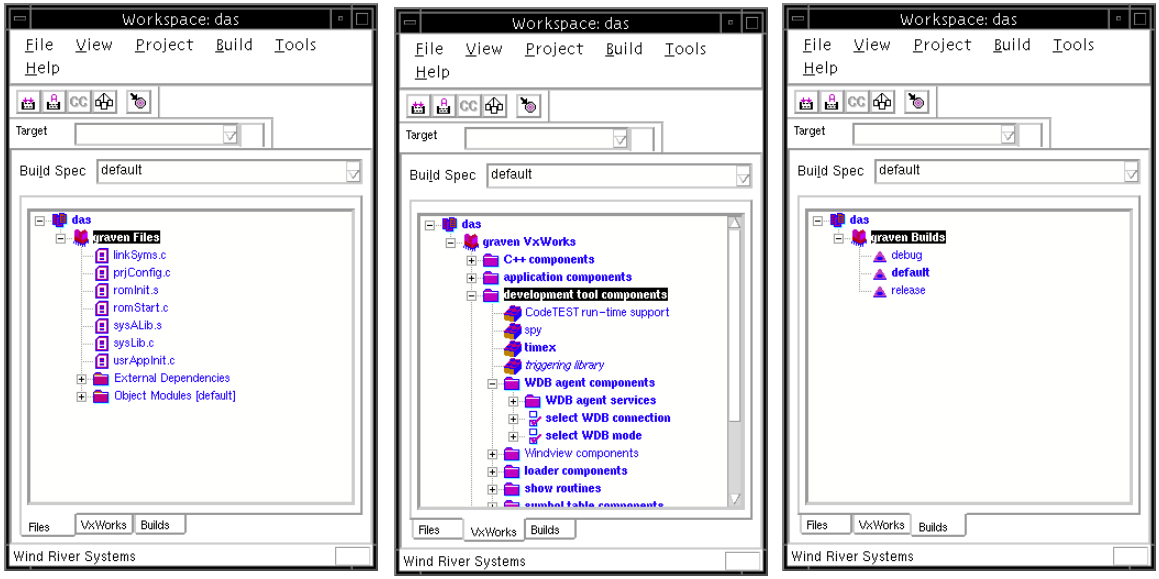
Project Facility GUI

The main components of the project facility GUI are:

- A project selection window, which allows you to begin creation of a new project, or open an existing project.
- An application wizard that guides you through creation of a new project.
- A workspace window, which provides you with a view of projects, and the files, VxWorks components, and build options that make them up. The workspace window also provides access to commands for adding and deleting project files, creating new projects, configuring VxWorks components, defining builds, downloading object files, and so on.
- A build toolbar, which provides access to all the major build commands.
- A target list, which allows you to specify the same target servers in the workspace that are available in the launcher.

As its name implies, the Workspace window provides the framework for the project facility. The window displays information about projects files, VxWorks components (if any), and build options in three tabbed views: Files, VxWorks, and Builds (Figure 4-1).

Figure 4-1 **Workspace Window Views: Files, VxWorks, and Builds**

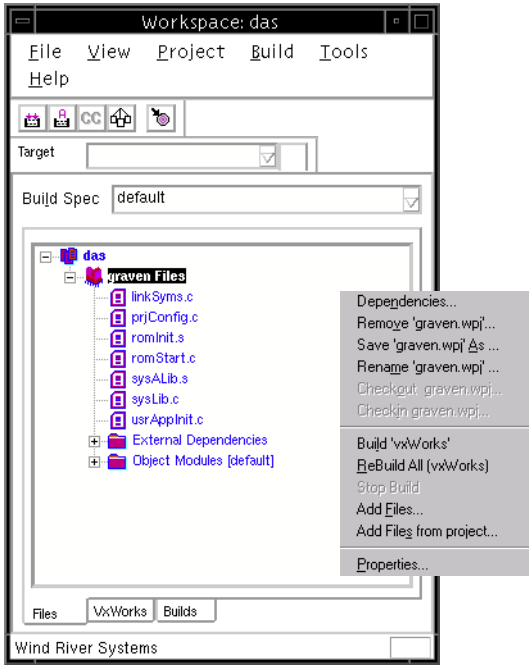


The workspace allows you to:

- Scale and customize VxWorks by adding and deleting components, as well as display component dependencies and view object sizes.
- Display information about the files, VxWorks components, and build options that make up a project, or set of projects.
- Add, open for editing, compile, and delete source code files.
- Download applications to the target.
- Specify and modify one or more builds for a project, display detailed build information, and modify build options.
- Add, delete, rename, or build a project.

A context-sensitive menu is available in each of the workspace views. A right-mouse click displays the menu. The first section of the menu provides commands relevant to the GUI object you have selected. The second section displays commands relevant to the current page of the window. And the third section displays global commands that are relevant to the entire workspace (Figure 4-2).

Figure 4-2 **Workspace Window Pop-up Menu**



Many of the pop-up menu options are also available under the File, Project, and Build menus.

Tornado will use your default editor. For information about using an alternate editor, integrating configuration management tools (such as ClearCase) with the project facility, and other customization options, see *11. Customization*.

Workspace Icons

As you expand the tree structure in each workspace pane, the icons by each tree element tell you what it is or what it contains.

Table 4-1 **Workspace Icons**

Icon	Location	Description
	All panes	Workspace
	Files and Builds panes	Downloadable application

Table 4-1 **Workspace Icons**

Icon	Location	Description
	All panes	Bootable system
	VxWorks pane	Component folder
	VxWorks pane	Selection folder
	VxWorks pane	Component
	Builds pane	Build specification
	Files pane	Source or object folder
	Files pane	Source file

4.2 Planning Your Projects

This section explains the steps necessary to get your product development underway. When you finish, you will be able to employ the features of the Tornado cross-development environment to their greatest utility.

To achieve full project facility support from Tornado, you will need to:

- Obtain or create a functioning BSP.
- Create a project from this BSP.
- Add your application code to this project, or to another in the same workspace.
- Create a new boot image (may not be required).

4.2.1 Getting a Functional BSP

To get a functioning BSP, you can:

- Use a Wind River- or third-party-supplied BSP (this includes the integrated or optional simulator).
- Create your own custom BSP to support custom hardware.

Using a Wind River or Third-Party BSP

Tornado 2.2 BSP

If your BSP was included with Tornado 2.2, you can create a bootable project from it directly. Use the project wizard for a bootable application to create a project based on your BSP or the pre-built project (*bspName*_**vx.wpj**) which is shipped with every Wind River-supplied BSP.

Tornado 2.0 BSP

For information on migrating a Tornado 2.0-compliant BSP to Tornado 2.2, see the *Tornado Migration Guide*.

Third-party or Tornado 1.0.1 BSP

If your BSP came from a third party or from Tornado 1.0.1, see the *Tornado Migration Guide* or the Tornado 2.0 documentation.

You may wish to enable the Tornado 1.0.1 compatibility mode, which exposes menu items to execute BSP builds in a BSP directory. (See *11. Customization*.) Once your BSP builds, you may proceed to create a bootable project from it immediately. See *4.5 Creating a Bootable Application*, p.147.

Using a Custom BSP For Custom Hardware

Creating a BSP

If you need to create your own BSP, refer to the *VxWorks BSP Developer's Guide* (a separate product available from Wind River). If you wish to develop the BSP and the application code in parallel, you may wish to begin application development on the VxWorks simulator. See *Using the Simulator BSP*, p.101.

Using a Pre-Existing BSP With the Project Facility

If you already have a custom BSP that is Tornado 2.0 compliant, see the *Tornado Migration Guide* for information on migrating from 2.0 to 2.2.

If you already have a custom BSP but it is not Tornado 2.0 compliant, you will need to modify it to conform to the guidelines outlined in the *VxWorks BSP Developer's Guide* in order to use it with the Tornado project facility. Once you have modified it, verify that it builds properly before creating a project for it.

NOTE: If you do not make your BSP Tornado 2.0 compliant, Tornado will not be able to provide project-based support for customizing, configuring, or building it.

Using a BSP Outside the Project Facility

You may still use a non-compliant BSP by managing its customization and configuration manually. For information on using manual methods, see *5. Command-Line Configuration and Build*. You can still create downloadable projects to hold your application code and download them to a target booted with a non-compliant BSP.

Using the Simulator BSP

You can use the target simulator if you want to develop the BSP and application code for your product in parallel, or if your target hardware is not yet ready. The integrated simulator contains default VxWorks functionality sufficient for supporting many applications. It does not have networking support; for this you can use the full simulator (VxSim), which is available as an optional product.

4.2.2 Creating a Bootable Project Based On a BSP

Using the VxWorks Simulator

Integrated Simulator With Basic Functionality

If you are using the integrated simulator and do not need to customize it by adding or removing VxWorks functionality, you need not create a bootable project until you have your production BSP ready.

Integrated Simulator With Added Functionality

If you need additional VxWorks functionality, you must create a bootable project immediately. Use the project wizard for a bootable application. You will use a different base depending on what additional functionality you need.

- **No networking:** If you do not need networking to support your application, you can create a bootable project based on the integrated simulator, configure it, and build it.

- **Networking:** If you need network support, you will need the VxWorks full simulator (VxSim), which can be purchased from Wind River as an optional product.

Creating a project and configuring it is identical for both the integrated and full simulator.

Base this project on the simulator BSP (either the integrated simulator or the optional product), or the pre-built, default simulator project (**sim***host*_**vx.wpj**. At this point, your project builds an image identical to the integrated simulator as you received it from Wind River. Now add any components you need using the VxWorks tab in the Workspace view (see *4.4.3 Configuring VxWorks Components*, p.134).

Using a Real Target

Create a bootable project using the project wizard. Base the project on your BSP. If project creation fails, your BSP is probably not Tornado 2.2-compliant. See the *VxWorks BSP Developer's Guide* for information on how to make it compliant.

Image Size Considerations

Use size information to make sure your image fits in your target memory space. The approximate image size information displayed in the Component Add Dialog reports the size of the VxWorks code in your configuration and the increase or decrease resulting from adding or removing components. This size is smaller than your actual image size, as it does not reflect your BSP support code or any application code you will be adding.

4.2.3 Developing and Adding Your Application Source Code

Adding Existing Application Source Code

Use Your Existing Build System

If you already have a working application, or if your application is very large, you may want to use your own build system. You can use the project facility to link the output of an external build into VxWorks and even start external builds (see *External Build System*, p.106).

Alternatively, you may want to use the project facility to configure your VxWorks image and produce a makefile. You can build your application outside Tornado and call the project facility-generated makefile from your build to produce a final image.

Integrate It With Your Bootable Project

Use this approach if your edit-compile-reboot cycle is relatively quick. Add the files to the bootable project using the Add File(s) to Project context menu available in the File tab of the Workspace view. Then edit the VxWorks initialization file, **usrAppInit.c**, adding calls to your application's initialization and startup routines. The VxWorks application initialization component is required, and is included by default. See *4.4 Creating a Custom VxWorks Image*, p.127 and *4.5 Creating a Bootable Application*, p.147.

Create a Separate, Downloadable Project For Your Code

Use this method if rebooting your target is inconvenient, and if your code is modular enough that it can be added to the running target without interrupting execution or if you have the means to start and stop your application. Create a downloadable project using the Downloadable Project Wizard. Add application files with the Add File(s) to Project context menu. Build your downloadable project. Boot the target using the appropriate image described in *4.2.2 Creating a Bootable Project Based On a BSP*, p.101. Download the partially linked and munched **.out** file produced by your project.[2] See *4.3 Creating a Downloadable Application*, p.112.

Creating New Application Source Code

Use File>New from the main Tornado menu bar to create a new file and specify the project into which it should be added.

Building With Custom Build Rules

If some of your source files require processing with tools not included with Tornado, you may want to add custom build rules to process them.

2. For information about munching, see the *VxWorks Programmer's Guide: C++ Development*.

You have two choices:

- Create a build rule specific to the source file

  This ensures that the custom rule will be invoked only to process the specified source file. For example, you may wish to add a custom rule to process a yacc file into a C source file. To create a custom rule, see *4.6 Working With Build Specifications*, p.148.

NOTE: If you migrate source files from one project to another, you will need to recreate the custom build rules for these files in the new project.

- Create a custom rule for the build

  A build-specific custom rule can invoke any command and reference any build dependencies. The rule can be selected as the current build rule to build the desired output explicitly or, if the Invoke this rule before building Project box is checked, it will be built implicitly prior to building any of the built-in rules (such as **vxWorks**, or *project*.**a**) for the project.

NOTE: Custom build rules cannot be copied between projects. If you will use either form of custom build rules, and know that you will be migrating files between projects, you may wish to put files with similar build settings into separate projects. These projects can then be built and linked together. For more information, see the hierarchical sub-project model discussed in *Sub-Projects*, p.106.

Developing Architecture-Independent Applications

The techniques for developing applications that are independent of target architecture are described below.

Migrating Files

You may migrate application source files between any two projects that coexist in the same workspace. Use the Add File(s) from Project context menu from the File tab. If you have defined custom build rules for any of your source files, you will have to replicate them in the destination project by hand.

CAUTION: It is important that you only migrate application source files between projects. BSP-specific files, and those synthesized by Tornado for your project, cannot be migrated. Only Tornado's project wizards can be used to create or reference these files.

Creating Sub-Projects

If you have a number of files that must be built with special build rules or flags, it may be easier to create a new project to build these particular files, and then build that project as a sub-project of your main project. For more information, see *Structuring Your Projects*, p.105.

Using Configuration Management

Tornado provides basic configuration management integration. To enable and configure it, see *11. Customization*.

Configuring VxWorks

VxWorks must be configured to support the calls your application makes to it, or you will not be able to link your image. If your BSP provides a "bare-bones" VxWorks configuration, you may wish to use Project>Auto Scale to detect and add most of the VxWorks functionality you require. Auto Scale will compile your code, analyze the symbols in your object modules, map them to components, and offer to include those components. There may be some components that Auto Scale does not detect. If you Auto Scale, build, and still get link errors, you will need to add the additional components from the workspace VxWorks tab. For information on using Auto Scale, see *4.5.1 Using Automated Scaling of VxWorks*, p.147.

Structuring Your Projects

You have three choices in how you organize the complete build of your application into VxWorks.

Single Project

Add all your application source code to one bootable project. This method is the simplest. All your source code is added to the bootable project, which already contains the BSP code and is linked to the VxWorks libraries.

External Build System

The Tornado workspace is very convenient for configuring VxWorks, building small applications, or building, downloading, and debugging small parts of a large application. It is not designed to handle a complete build of a large, modular application, which often requires sub-projects (though this can be achieved using custom rules and macros). For this reason, you may want to use an external build system to build your application, then link it to VxWorks using the **EXTRA_MODULES** or **LIBS** macros. You can write a custom rule to invoke your external build process; see *Sub-Projects*, p.106. Alternatively, your build can kick off a VxWorks build and link your application code as the final step.

Sub-Projects

Sub-projects allow you to create as many projects as are needed to hierarchically organize and build your product. This approach accommodates existing hierarchically-organized source code. You will want to use this approach if:

– some source files need different build settings or custom rules.
– a split of your code is desirable for organizational or structural reasons.

Tornado has only limited support for managing and building these hierarchical sub-projects. You must use macros and custom rules to create the hierarchy and structure the builds manually. For directions on how to organize your application code into sub-projects, see Example 1, below.

Example 1 **Using Sub-Projects**

This example illustrates how a master project can be used to build several sub-projects. The master project builds the sub-projects as **.pl** (partially linked) modules. Then they are linked with the master project and munched (integrated with code to call C++ static constructors and destructors) in the final build step.

In this example, the master project is a bootable project, and there are two sub-projects that are downloadable projects. However, a downloadable project can also serve as a master project. You could use this approach if you wanted to build several downloadable sub-projects and link them into a single downloadable project. You could also use this approach to integrate an external application build into VxWorks. You need to modify the custom rules in the example to invoke your external build (for example, using **make**).

Assumptions:

- The bootable project is called **Master** and resides in a directory of the same name. It contains a build specification called default based on the simpc (Windows host simulator) BSP.

- The two downloadable projects are called **Project1** and **Project2**, and they also reside in directories of the same name. Each contains a build specification called **SIMNTgnu** based on the PC simulator toolchain.
- **Project1** contains a C source file called **foo.c**, containing a function called ***Test*( )**.
- **Project2** contains a C source file **goo.c**, which in turn contains a function called ***Test2*( )**.
- ***Test*( )**, which calls ***Test2*( )**, is the main application entry point.
- Dependencies have been generated for each of the two downloadable projects, and they have been saved. This creates makefiles for them. Without the makefiles, the build fails.

Go first to the Build tab. Expand the project Master. Double-click on the build specification default to display the build property sheet. In the build property sheet, select the Rules page.

You enter a new rule by filling in the Target, Dependencies, and Commands fields of the Create or Edit Rule dialog box. For example, to add the **clean** rule, you type **clean** in the Target field, **CleanProject1 CleanProject2 vxWorks** in the Dependencies field, and nothing in the Commands field. For each rule, you must also uncheck the Invoke this rule before building project checkbox.

The required rules are listed below in makefile syntax. The first example shows the syntax. Fill in the appropriate boxes for each rule.

```
SYNTAX:
    target : dependencies
        commands

RULES:
    ../../Project1/SIMNTgnu :
        - mkdir $@

    Master : ../../Project1/SIMNTgnu/Project1.pl
    ../../Project2/SIMNTgnu/Project2.pl VxWorks

    ../../Project1/SIMNTgnu/Project1.pl : ../../Project1/SIMNTgnu
    wind_force_make
        $(MAKE) -C ../../Project1/SIMNTgnu -f ../../Project1/Makefile
    BUILD_SPEC=SIMNTgnu Project1.pl

    ../../Project2/SIMNTgnu :
        - mkdir $@

    ../../Project2/SIMNTgnu/Project2.pl : ../../Project2/SIMNTgnu
    wind_force_make
        $(MAKE) -C ../../Project2/SIMNTgnu -f ../../Project2/Makefile
```

```
BUILD_SPEC=SIMNTgnu Project2.pl

CleanProject1 : ../../Project1/SIMNTgnu
    $(MAKE) -C ../../Project1/SIMNTgnu -f ../../Project1/Makefile
BUILD_SPEC=SIMNTgnu clean

CleanProject2 : ../../Project2/SIMNTgnu
    $(MAKE) -C ../../Project2/SIMNTgnu -f ../../Project2/Makefile
BUILD_SPEC=SIMNTgnu clean

clean : CleanProject1 CleanProject2
```

CAUTION: The **clean** rule must have the correct case, and it cannot include any commands. In this example, **CleanProject1** and **CleanProject2** are added as dependencies to the default **clean** rule for VxWorks. The **clean** rule ensures that the ReBuild All command rebuilds **Project1**, **Project2**, and VxWorks.

If you wish your rules to be portable between architectures, substitute **(CPU)(TOOL)** for **SMNTgnu**.

In the Rules pane, set the default build rule for project **Master** to be the rule **Master**. Next, in the Build pane, in the MACROS tab for project **Master**, append to the **EXTRA_MODULES** macro "**../../Project1/SIMNTgnu/Project1.pl**" and "**../../Project2/SIMNTgnu/Project2.pl**" and click the Add/Set button.

NOTE: You can use **PRJ_LIBS** to link extra modules to downloadable projects, in the same way that **EXTRA_MODULES** is used for bootable projects.

Add to the source file **usrAppInit.c**, in project **Master**, a function prototype for, and a call to, the function ***Test*( )**:

```
void Test(void);

void usrAppInit (void)
    {
    #ifdef  USER_APPL_INIT
    USER_APPL_INIT; /* for backwards compatibility */
    #endif

    /* add application specific code here */
    Test();
    }
```

When you build **Master**, all three will be built, munched, and linked into one bootable image. (For information on munching, see the *VxWorks Programmer's Guide: C++ Development*.)

NOTE: The sub-project objects in the example (**Project1.pl** and **Project2.pl**) need not have been generated by Tornado downloadable projects. They could also have been the result of an external build system.

To modify this example to integrate an external application build system, you could, for instance:

- Replace all instances of **Project1.pl** with the partial link product of your external application build.
- Replace the **Project2.pl** rule with a rule appropriate for starting your external application build.

Example 4-2 **Avoiding Absolute Paths**

One problem that arises from using a version control system is that different users may extract projects and source files to different locations (for example, in Visual SourceSafe, or CVS), or map views to different drive letters (Clearcase on Windows). It helps greatly if projects do not have any absolute paths written into them.

If source files or subprojects are in a directory which is at the same directory level as the parent project directory or deeper, they are recorded in the parent project file with a path relative to the parent project directory. Organizing your source files and projects in this way is recommended to avoid absolute paths in project files and makefiles.

If you cannot organize your source files and projects in this way, we provide two environment variables to allow you to define the root of your source directory tree or your project directory tree: **WIND_SOURCE_BASE** and **WIND_PROJ_BASE**.

Below we give some examples showing how to avoid absolute paths to source files or to sub-projects.

1. Avoiding absolute paths to source files

   Suppose your project is in directory **t:\source_root\myproj** and your source files are organized as follows:

   ```
   t:\source_root\myproj\foo1.c
   t:\source_root\src\foo2.c
   t:\source_root\myproj\src\foo3.c
   ```

Your project will contain no absolute paths. Instead, the above files will be recorded in the project file as follows:

```
$(PRJ_DIR)/foo1.c
$(PRJ_DIR)/../src/foo2.c
$(PRJ_DIR)/src/foo3.c
```

Organizing your source files and projects in any of these ways is the recommended procedure for avoiding absolute paths in project files and makefiles.

If you cannot organize your source files and projects in one of these ways, the **WIND_SOURCE_BASE** environment variable allows you to define the root of your source directory tree. To illustrate the use of **WIND_SOURCE_BASE**, assume you wish to add the file:

```
t:\other_source_root\goo.c
```

If **WIND_SOURCE_BASE** is not defined, it appears in the project file with an absolute path (we do not support **$(PRJ_DIR)/../../**). However, before you add the file, you can use Tools->Options->Workspace to define:

```
WIND_SOURCE_BASE = t:\other_source_root
```

Then **t:\other_source_roo\goo.c** is recorded in the project file as:

```
$(WIND_SOURCE_BASE)/goo.c
```

The major drawback to the **WIND_SOURCE_BASE** variable is that all users of **myproj** must have **WIND_SOURCE_BASE** defined or the workspace cannot find **goo.c**.

2. Avoiding absolute paths to sub-projects

   This example refers to a master project called master and various sub projects called **sub1**, **sub2**,... The master project could be a bootable project and the sub-projects could refer to external build systems or downloadable projects.

   Assume the master project is in directory **t:\prj_root\master** and your sub-projects are organized as follows:

```
t:\prj_root\sub1
t:\prj_root\master\sub2
t:\prj_root\subprojects\sub3
t:\prj_root\master\subprojects\sub4
```

In this case, your master project will contain no absolute paths. Instead, the above directories will be recorded in the master project file as follows:

```
$(PRJ_DIR)/../sub1
$(PRJ_DIR)/sub2
$(PRJ_DIR)/../subprojects/sub3
$(PRJ_DIR)/subprojects/sub4
```

Organizing your projects in any of the above ways is the recommended procedure for avoiding absolute paths in project files and makefiles.

If you cannot organize your projects in any of these ways, we provide the **WIND_PROJ_BASE** environment variable to allow you to define the root of your project directory tree. To illustrate the use of **WIND_PROJ_BASE**, assume you wish to add the project:

```
t:\other_prj_root\sub5
```

If **WIND_PROJ_BASE** is not defined, then it appears in the master project file with an absolute path (we do not support **$(PRJ_DIR)/../../**). However, before you add the project you could use Tools->Options->Workspace to define:

```
WIND_PROJ_BASE=t:\other_prj_root
```

Now when you add **t:\other_prj_root\sub5**, this sub-project will be recorded in the master project file as:

```
$(WIND_PROJ_BASE)/sub5
```

The major drawback to the **WIND_PROJ_BASE** variable is that all users of the master project have to define **WIND_PROJ_BASE**, or the workspace is unable to find **sub5**.

NOTE: WIND_SOURCE_BASE and **WIND_PROJ_BASE** must be set before a source file is added or the variable is not recorded in the file path. The property page always reflects the path from which a source file was originally loaded, not the current path calculated from **WIND_SOURCE_BASE** (if it has changed). The build always uses the correct value, but the source editor launches against the stale copy of the file. To avoid confusion, reload the workspace whenever you change **WIND_SOURCE_BASE** and **WIND_PROJ_BASE**.

4.3 Creating a Downloadable Application

A downloadable application is a collection of relocateable object modules that can be downloaded and dynamically linked to VxWorks, and started from the shell or debugger. A downloadable application can consist of a single "hello world" routine or a complex application.

To create a downloadable application, you must:

1. Create a project for a downloadable application.
2. Write your application, or use an existing one.
3. Add the application files to the project.
4. Build the project.

You can then download the object module(s) to the target system and run the application.

4.3.1 Creating a Project for a Downloadable Application

All work that you do with the project facility, whether a downloadable application, a customized version of VxWorks, or a bootable application, takes place in the context of a project.

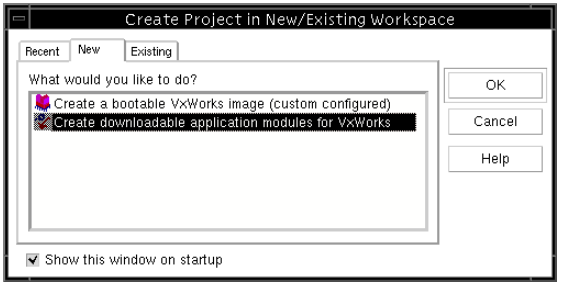
Open a project workspace by clicking the Project button in the Tornado Launch window. If the Create Project or Open Workspace window is open (the default when you first open the Tornado Project window[3]), click the New tab. Then choose the selection for a downloadable application, and click OK (Figure 4-3).

The application wizard appears (Figure 4-4). This wizard is a tool that guides you through the steps of creating a new project.

First, enter the full directory path and name of the directory you want to use for the project (only one project is allowed in a directory), and enter the project name. It is usually most convenient to use the same name for the directory and project, but it is not required.

3. You can modify the default behavior by un-checking the Show this window on startup box at the bottom of the window.

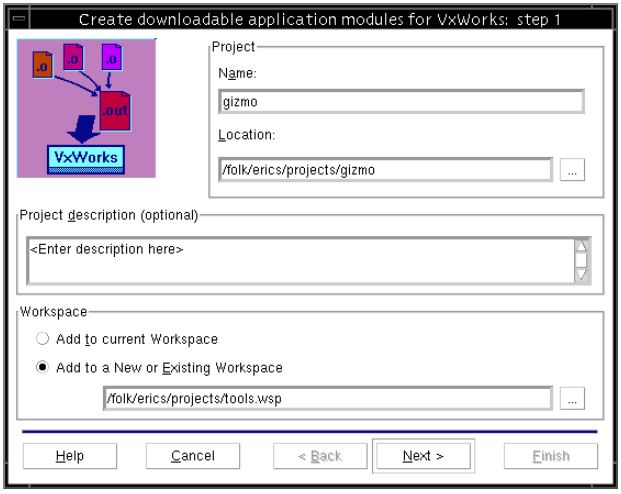
Figure 4-3 **Create Downloadable Application**



NOTE: You may create your projects anywhere on your file system. However, it is preferable to create them outside of the Tornado directory tree to simplify the process of future Tornado upgrades.

You may also enter a description of the project, which will later appear in the property sheet for the project. Finally, identify the workspace in which the project should be created. Click Next to continue.

Figure 4-4 **Application Wizard: Step One for Downloadable Application**

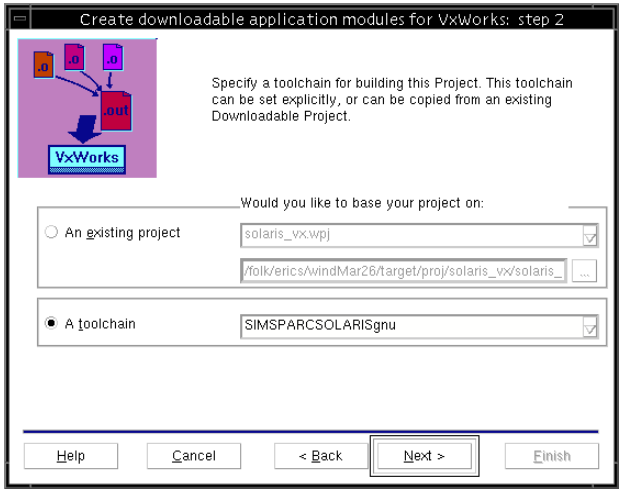


Identify the toolchain with which the downloadable application will be built. You can do so by referencing an existing project, or by identifying a toolchain.

Basing a project on an existing one means that the new project will reference the same source files and build specifications as the one on which it was based. Once the new project has been created, its build specifications can be modified without affecting the original project, but changes to any shared source files will be reflected in both.

For example, to create a project that will run on the target simulator, select A toolchain and select the default option for the target simulator from the drop-down list (Figure 4-5).[4] Click Next.

Figure 4-5 **Application Wizard: Step Two for Downloadable Application**



The wizard confirms your selections (Figure 4-6). Click Finish.

The Workspace window appears, containing a folder for the project. Note that the window title includes the name of the workspace (Figure 4-7).

NOTE: Pop-up menus provide access to all commands that can be used with the objects displayed in, and the pages that make up, the Workspace window (use the right mouse button).

4. The default toolchain names for target simulators take the form SIMSPARCSOLARISgnu.

Figure 4-6 **Application Wizard: Step Three for Downloadable Application**

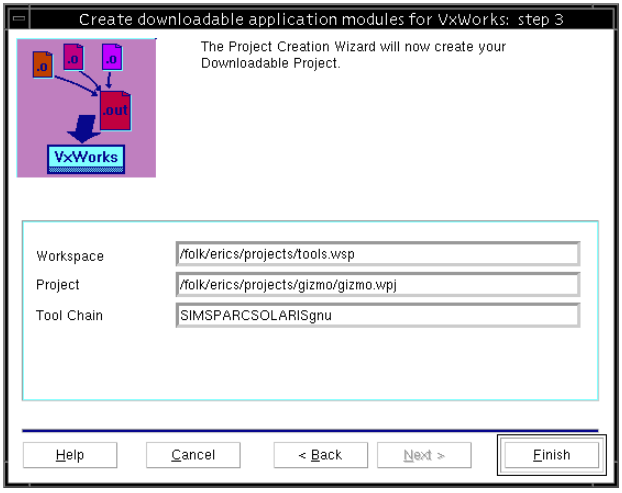
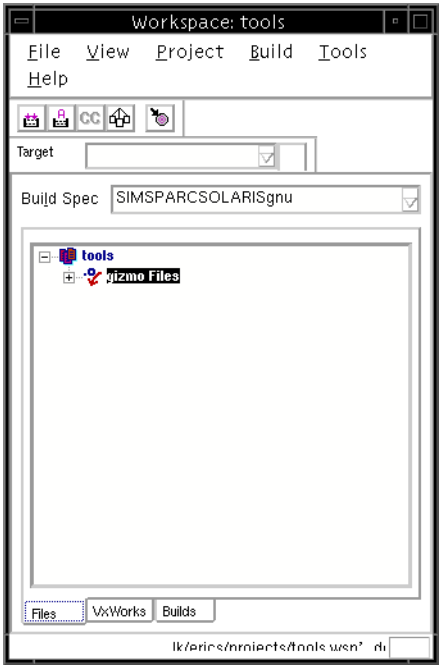


Figure 4-7 **Initial Workspace Window for a Downloadable Application**



4.3.2 Project Files for a Downloadable Application

The project facility generates a set of files whose contents are based on your selection of project type, toolchain, build options, and build configurations. During typical use of the project facility you need not be concerned with these files, except to avoid accidental deletion, to check them in or out of a source management system, or to share your projects or workspaces with others. The files are created in the directories you identify for the workspace and project. The files initially created are:

*projectName***.wpj**
: Contains information about the project used for generating the project makefile.

*workspaceName***.wsp**
: Contains information about the workspace, including which projects belong to it.

Both of these files contain information that changes as you modify your project, and add projects to, or delete projects from, the workspace.

When you build your application, a makefile is dynamically generated in the main project directory, and a subdirectory is created containing the objects produced by the build. The subdirectory is named after the selected build specification. If other build specifications are created and used for other builds, parallel directories are created for their objects.

4.3.3 Working With Application Files

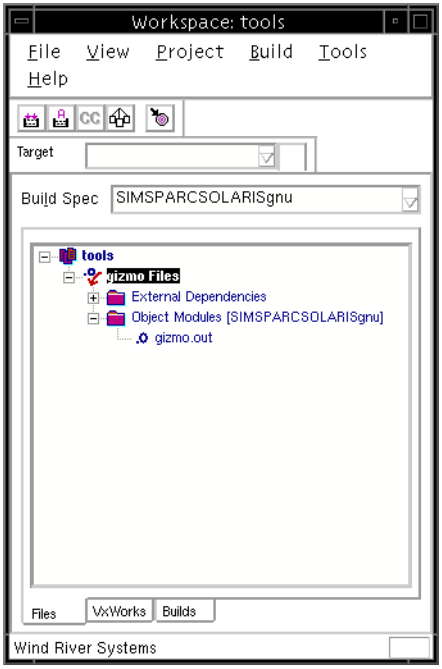
The Files view of the Workspace window displays information about projects, and the directories and files that make up each project (Figure 4-8).

The first level of folders in the Files view are projects. Each project folder contains:

- Project source code files, which are added to the project by the user.
- An Object Modules folder, which contains a list of objects that the project's build will produce. The list is automatically generated by the project facility.
- An External Dependencies folder, which contains a list of **make** dependencies. The list is automatically generated by the project facility.

Initially, there are only the default folders for Object Modules and External Dependencies, and the *projectName***.out** file. The file *projectName***.out** is created as a single, partially-linked module when the project is built. It comprises all of the

Figure 4-8 **Workspace Files View**



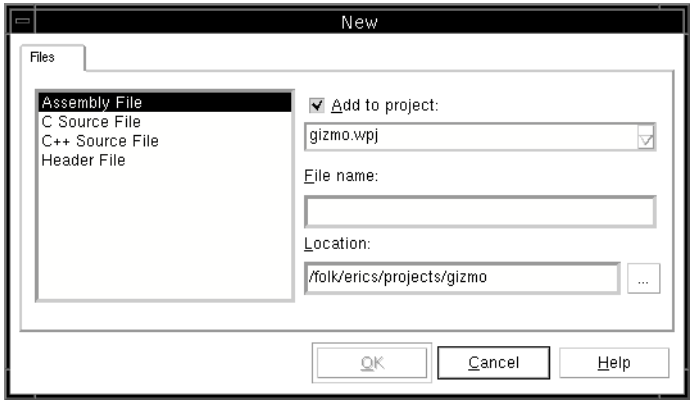
individual object modules in a project for a downloadable application, and provides for downloading them to the target simultaneously.

WARNING: Use of the *projectName*.**out** file is essential for downloading C++ modules, which require munching for proper static constructor initialization. You should also use the *projectName*.**out** file for downloading C modules to avoid any potential link order issues related to dynamic loading and linking.

Creating, Adding, and Removing Application Files

To create a new file, click File>New. Select the file type from the New dialog box. Then select the project to which the file should be added. Finally, enter the file name and directory, and click OK (Figure 4-9). The editor window opens, and you can write your code and save the file.

Figure 4-9 **New File Dialog Box**



Add existing files to a project by right-clicking in the Workspace window, selecting Add Files or Add FIles from project from the pop-up menu, and then using the associated dialog box to locate and select the file(s).

To link object files with your project, use the **PRJ_LIBS** macro or the Linker page of the build specification property sheet (see *Linker Options*, p.154). To link library (archive) files with your project, add the libraries to the list defined by the **LIBS** macro in the Macros page of the build specification property sheet (see *Makefile Macros*, p.150).

Remove files from the project by right-clicking on the file name and selecting Remove from the pop-up menu, or by selecting the file name and pressing **DELETE**.

⚠ **CAUTION:** Adding a file to a project or removing a file from a project does not affect its existence in the file system. The project facility does not copy, move, or delete user source files; merely the project facility's references to them. Removing a file from one workspace context does not affect references to it in any others, nor its existence on disk. However, if a file is included in more than one project or workspace, an edit made in one context will be reflected in all. (If this behavior is not desired, copy source files to another directory before adding them to a project.)

Displaying and Modifying File Properties

To display information about the properties of a file, right-click on the file name in the Workspace window, and select Properties from the pop-up menu. The extent of information displayed depends on the type of file and whether or not **make**

dependencies have been generated. In the case of source code, a Properties sheet for the file appears, displaying information about **make** dependencies; general file attributes such as modification date; and the associated make target, custom dependencies, and commands used for the build process (Figure 4-10).

Figure 4-10 **Source File Property Sheet**

See *Calculating Makefile Dependencies*, p.120, for information about how and when to calculate makefile dependencies. See *Compiler Options*, p.151 for information about overriding default compiler options for individual files.

Opening, Saving, and Closing Files

The File and pop-up menus provide options for opening, saving, and closing files. You can also use standard Windows-style shortcuts (such as double-clicking on a file name to open the file in the editor).

4.3.4 Building a Downloadable Application

The project facility uses the GNU **make** utility to automate compiling and linking an application.[5] It automatically creates a makefile prior to building the project. But before it can create a makefile, the makefile dependencies must be calculated. The calculation process, which is based on the project files' preprocessor **#include** statements, is also an automated feature of the project facility.

5. See the *GNU Make User's Guide* for more information about **make**.

Binaries produced by a given build are created in a project subdirectory with the same name as the name of the build specification (*projectName*/*buildName*).

NOTE: All source files in a project are built using a single build specification (which includes a specific set of makefile, compiler, and linker options) at a time. If some of your source requires a different build specification from the rest, you can create a project for it in the same workspace, and customize the build specification for those files. One project's build specification can then be modified to link in the output from the other project. See *Sub-Projects*, p.106.

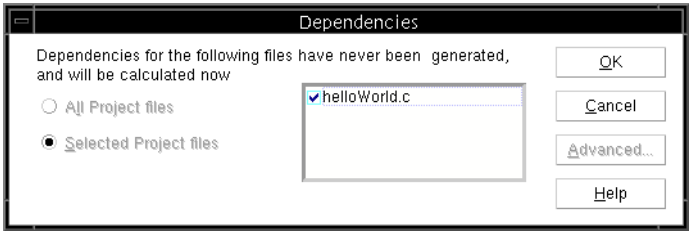
NOTE: The project facility allows you to create specifications for different types of builds, to modify the options for any one build, and to select the build specification you want to use at any given time easily. See *4.6 Working With Build Specifications*, p.148.

CAUTION: Different versions of C++ run-time support are provided for the GNU and Diab toolchains. For this reason, you cannot combine C++ objects compiled with GNU with C++ objects compiled with Diab. All C++ applications must be compiled with the same tool.

Calculating Makefile Dependencies

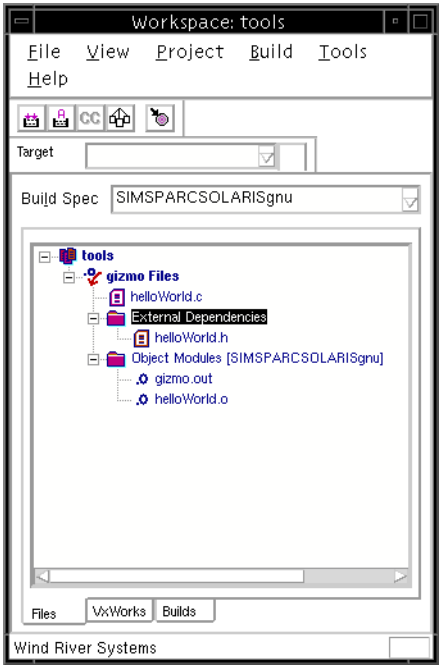
To calculate makefile dependencies, select Dependencies from the workspace pop-up menu. The Dependencies dialog box appears (Figure 4-11). Click OK.

Figure 4-11 **Dependencies Dialog Box**



After dependencies have been calculated, the files are listed in the External Dependencies folder (Figure 4-12).

Figure 4-12 **External Dependencies**



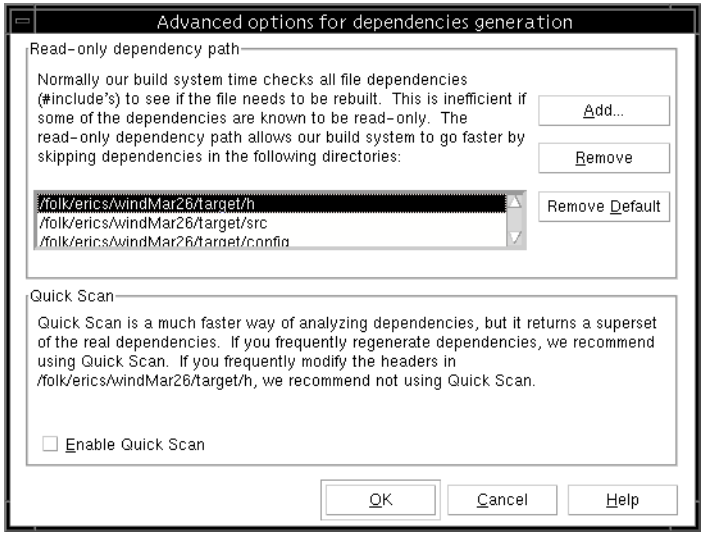
If you do not calculate dependencies before you start a build, Tornado prompts you to do so for any project files for which dependencies have not previously been calculated. Dependencies are *not* calculated for each build specification. If your dependencies change for different build specifications (for example, if they are CPU-dependent), then you may want to:

- Create a new project for each build specification.
- Regenerate dependencies when you switch build specifications.

Tornado assumes that your header files are in either your project directory or *installDir*/**target/h**. If you have placed files in other locations, you need to make two changes to your project build specification. Right-click on the name of your build specification (for example, SIMSPARCSOLARISgnu). Select Properties from the pop-up menu. On the C/C++ compiler tab, click the Include paths button. Add a separate entry for each directory path.

The Advanced option allows you to speed up the build process by specifying paths in which *none* of the dependencies could have changed since the last build. The timestamps for the files in the specified paths are *not* checked (Figure 4-13).

Figure 4-13 **Dependency Calculation Option**



Build Specifications

Each build specification for a downloadable application consists of a set of options for makefile rules and macros, as well as for the compiler, assembler, and linker. A default build specification is defined when you create your project. To display its property sheet, double-click on the build specification name in the Builds view of the workspace to display the property sheet for the build specification.

Rules Tab

The Rules page (Figure 4-14) allows you to select from the following build target options:

objects
: Objects for all source files in the project.

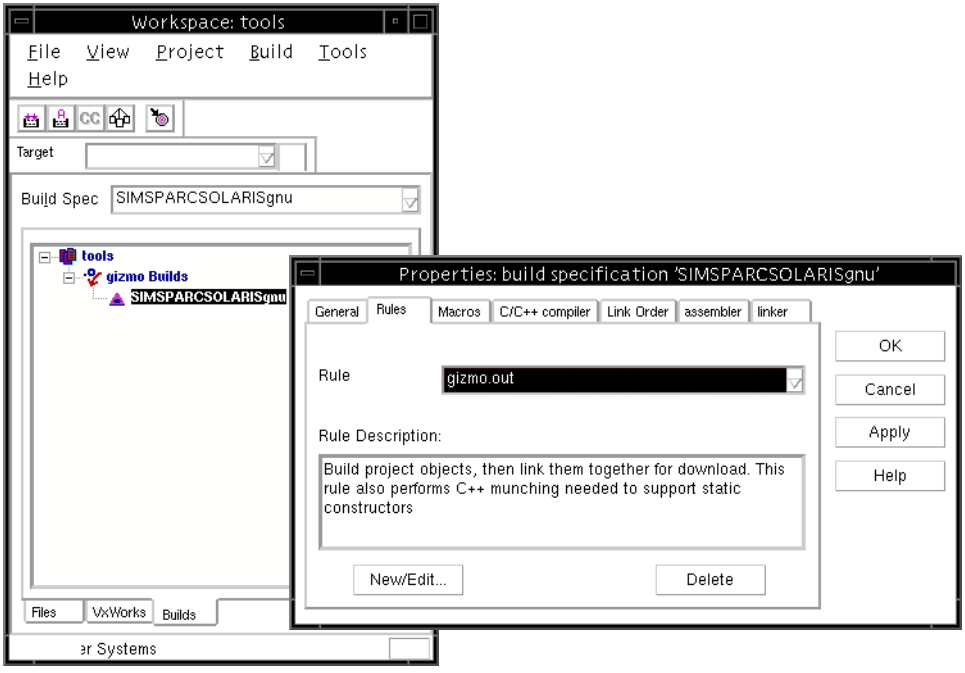
archive
: An archive (library) file.

projectName.out
: A single, partially-linked and munched object that comprises all of the individual object modules in a project. This is the correct module to download.

For information on munching, see the *VxWorks Programmer's Guide*: *C++ Development*. For more information on linking, see *Linker Options*, p.154.

projectName.pl
A single, partially-linked but *not* munched object that comprises all of the individual objects modules in a project. This file is provided for sub-project support. It is *not* intended for download since it has not been munched. See *Linker Options*, p.154.

Figure 4-14 **Build Specification Property Sheet**



You can use the project facility to change the options for a given build specification, create and save new build specifications, and select the specification to use for a build. You can, for example, create one build specification for your project that includes debug information, and another that does not. For more information, see *4.6 Working With Build Specifications*, p.148.

NOTE: It is sometimes useful to build an application for the target simulator, and then to create a new build specification to build it for a real target.

Macros Tab

The Macros tab contains pre-set build macros. Do not delete the pre-existing macros; while you can reenter them, the value will be lost. You can add and delete your own macros by typing in the Name window and then clicking the Add button.

Macros that are useful with bootable projects:

EXTRA_MODULES	Extra object modules to link into the VxWorks image.
LIBS	Libraries against which VxWorks is linked.
POST_BUILD_RULE	Shell commands to execute after the build has completed.
RAM_HIGH_ADRS	RAM address where the boot ROM data segment is loaded. It must be a high enough value to ensure loading VxWorks does not overwrite part of the ROM program.
RAM_LOW_ADRS	Beginning address to use for the VxWorks run-time in RAM.

WARNING: RAM_HIGH_ADRS and **RAM_LOW_ADRS** are also defined in **config.h**; the two definitions must match!

Macros that are useful with downloadable projects:

PRJ_LIBS	Libraries or modules against which a downloadable application is linked.
POST_BUILD_RULE	Shell commands to execute after the build has completed.

Environment Variables

If you are using the Diab tools, you must have two settings in place:

- Be sure that *installDir***/host/diab/WIN32/bin** is in the system path.
- Be sure that the environment variable **DIABLIB** is set to *installDir***/host/diab**.

There is a batch file called **torVars.bat** in *installDir***/host/x86-win32/bin** that will set **DIABLIB** for you.

Building an Application

To build a project with the default options, select the name of the project (or any subordinate object in its folder) and then select Build 'projectName.out' from the

pop-up menu. If you have created build specifications in addition to the default, you can select the build specification you want to use from the Build Spec drop-down list at the top of the workspace window before you start the build.

WARNING: Tornado only calculates dependencies upon the first use of a file in a build. Once an initial set of dependencies has been calculated, Tornado does not attempt to detect changes in dependencies that may have resulted from modification of the file. If you have changed dependencies by adding or deleting **#include** preprocessor directives, you should regenerate dependencies.

The Build Output window displays build messages, including errors and warnings (Figure 4-15). Any compiler errors or warnings include the name of the file, the line number, and the text of the error or warning text.

Figure 4-15 **Build Output**

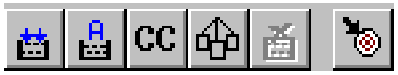
WARNING: The default compiler options include debugging information. Using debugging information with the optimization set to anything but zero may produce unexpected results. See *4.6 Working With Build Specifications*, p.148 for information about modifying builds and creating new build specifications.

To force a rebuild of all project objects, select Rebuild All from the pop-up menu (which performs a **make clean** before the build).

Build Toolbar

The Build toolbar provides quick access to build commands. Display of the toolbar (Figure 4-16) is controlled with the View>Build Toolbar menu option.

Figure 4-16 **Build Toolbar**



The Build toolbar commands (Table 4-2) are also available from the main menus and the Workspace pop-up menu.

Table 4-2 **Build Toolbar Buttons**

Button	Menu	Description
	Build>Build	Build project.
	Build>Rebuild All	Rebuild project (performing a **make clean** first).
	Build>Compile	Compile selected source file.
	Build>Dependencies	Update dependencies.
	Project>Download	Download object file (or boot image for target simulator).

4.3.5 Downloading an Application

Before you can download and run an application, you must boot VxWorks on the target system, have access to a Tornado registry, and configure and start a target server. For more information, see *2. Setup and Startup* and *3. Launcher*.

You can download an entire project from the project workspace by selecting Download 'projectName.out' from the pop-up menu for the Files view, or by using the download button on the Build toolbar. You can download individual object

modules by selecting the file name and then selecting the Download 'filename.o' option from the pop-up menu. However, you may inadvertently introduce errors by downloading individual object modules out of sequence. We strongly recommend that you *always* download the partially-linked *projectName***.out** file.

C++ projects must be downloaded as *projectName***.out** because this file is produced from application files and munched for proper static constructor initialization.

To run a downloaded application, use WindSh or CrossWind. For more information see *7. Shell* and *9. Debugger*.

To unload a project from the target, use the Unload 'projectName.out' option on the pop-up menu.

4.3.6 Adding and Removing Projects

New projects can be added to a workspace by selecting the menu options File>New Project and creating a new project when the workspace is open.

Existing projects can be added to a workspace by selecting File>Add Project to Workspace, and using the file browser to select a project file (*projectName***.wpj**).

Projects can be removed from a workspace by selecting the project name in the Files view, and then selecting the Remove option from the pop-up menu, or by selecting the project name and pressing **DELETE**.

NOTE: When you remove a project, you only remove it from the workspace. The project directory and its associated files are not removed from disk.

4.4 Creating a Custom VxWorks Image

The Tornado distribution includes a VxWorks system image for each target shipped. The *system image* is a binary module that can be booted and run on a target system. The system image consists of all desired system object modules linked together into a single non-relocateable object module with no unresolved external references. In most cases, you will find the supplied system image adequate for initial development. However, later in the cycle you may want to create a custom VxWorks image.

VxWorks is a flexible, scalable operating system with numerous facilities that can be tuned, and included or excluded, depending on the requirements of your application and the stage of the development cycle. For example, various networking and file system components may be required for one application and not another, and the project facility provides a simple means for either including them in, or excluding them from, a VxWorks application. In addition, it may be useful to build VxWorks with various target tools during development (such as the target-resident shell), and then exclude them from the production application.

Once you create a customized VxWorks, you can boot your target with it and then download and run applications. You can also create a bootable application simply by linking your application to VxWorks and adding application startup calls to the VxWorks system initialization routines (see *4.5 Creating a Bootable Application*, p.147).

4.4.1 Creating a Project for VxWorks

All work that you do with the project facility, whether a downloadable application, a customized version of VxWorks, or a bootable application, takes place in the context of a project.

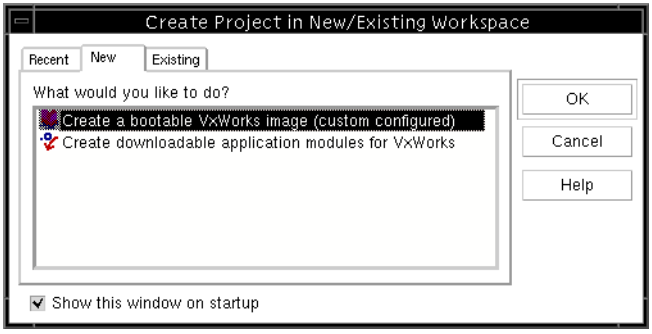
Open a project workspace by clicking the Project button in the Tornado Launch window. If the Create Project or Open Workspace window is open (the default when you first open the Tornado Project window[6]), click the New tab. Otherwise, click File>New Project. Then choose the selection for a bootable application, and click OK (Figure 4-17).

The application wizard appears (Figure 4-18). This wizard is a tool that guides you through the steps of creating a new project.

First, enter the full directory path and name of the directory you want to use for the project (only one project is allowed in a directory), and enter the project name. It is usually most convenient to use the same name for the directory and project, but it is not required.

6. You can modify the default behavior by un-checking the Show this window on startup box at the bottom of the window.

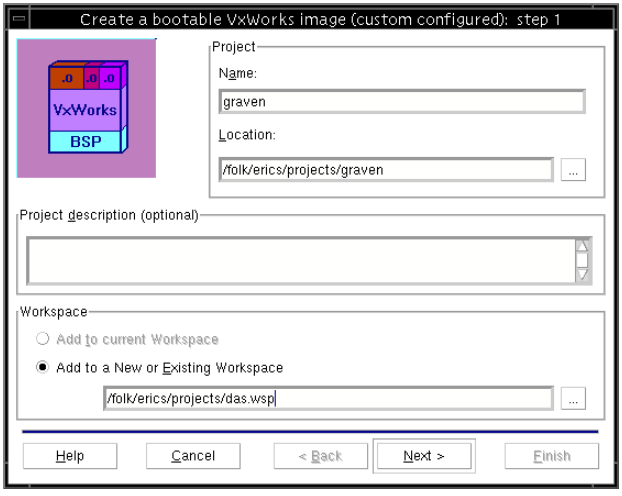
Figure 4-17 **Create Bootable Application**



NOTE: You may create your projects anywhere on your file system. However, it is preferable to create them outside of the Tornado directory tree to simplify the process of future Tornado upgrades.

You may also enter a description of the project, which will later appear in the property sheet for the project. Finally, identify the workspace in which the project should be created. Click Next to continue.

Figure 4-18 **Application Wizard: Step One for Bootable Application**



Then you identify the BSP with which you will build the project. You can do so by referring to an existing project, or by identifying a BSP that you have installed.

NOTE: If you are creating a customized VxWorks image or a bootable application, the project will be generated faster if you base it on an existing project rather than on a BSP. This is because the project facility does not have to regenerate configuration information from BSP configuration files. All Tornado 2.x BSPs include both GNU and Diab project files for this purpose. Options for BSP projects are available in the drop-down list for existing projects. For example, the mbx860 BSP project files are:

installDir/**target/proj/mbx860_gnu/mbx860_gnu.wpj**
installDir/**target/proj/mbx860_diab/mbx860_diab.wpj**

Basing a project on an existing project means that the new project will reference the *same* source files as the one on which it was based, but it will start with *copies* of the original project's VxWorks configuration and build specifications. The build specifications and VxWorks configuration for the new project can be modified without affecting the original project, but changes to any shared source files will be reflected in both.

For example, to create a project for a module that will run on a mbx860 target, select An existing project and then select mbx860_gnu.wpj (or mbx860_diab.wpj if you have purchased the Diab tools) from the drop-down list (Figure 4-19). Click Next.

If, on the other hand, you must base your project on a BSP, select A BSP and choose one of the BSPs you installed or the appropriate simulator. If your BSP offers both GNU and Diab toolchains, select a toolchain as well. Click Next.

The wizard confirms your selections (Figure 4-20). Click Finish.

The Workspace window appears.

Figure 4-19 **Application Wizard: Step Two for Bootable Application**

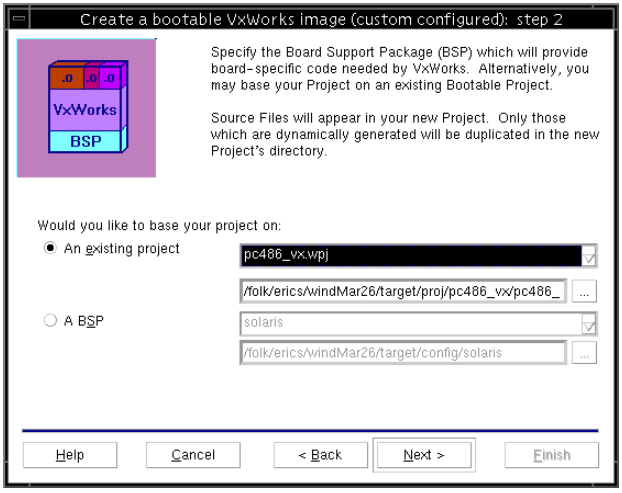
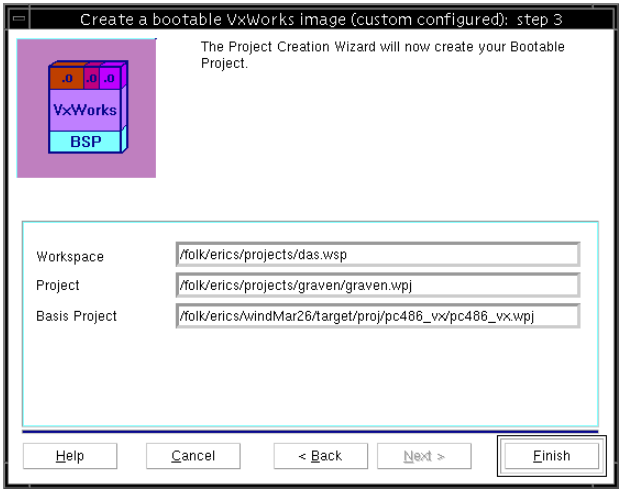


Figure 4-20 **Application Wizard: Step Three for Bootable Application**

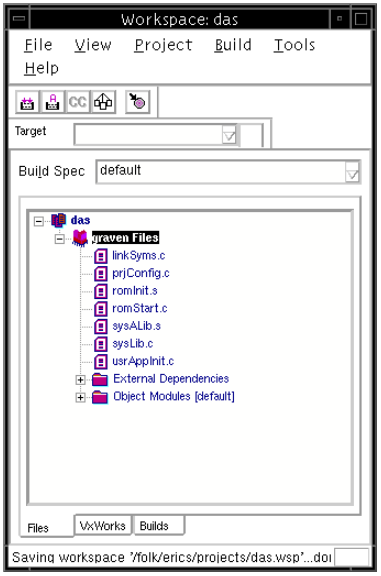


4.4.2 Project Files for VxWorks

The project facility generates, or includes copies of, a variety of files for a VxWorks project. The names of the files that you may need to work with are displayed in the workspace File view (Figure 4-21).

NOTE: Pop-up menus provide access to all commands that can be used with the objects displayed in, and the pages that make up, the Workspace window (use the right mouse button).

Figure 4-21 **VxWorks Project Files**



During typical use of the project facility you do not need to be concerned with these files, except to avoid accidental deletion, to check them in or out of a source management system, or to share your projects or workspaces with others. You will need to edit **userAppInit.c**, however, when you create a bootable application (see *4.5 Creating a Bootable Application*, p.147).

CAUTION: Do *not* check in **linkSyms.c**, **prjConfig.c**, **prjComps.h**, or **prjParams.h**; these files are regenerated whenever the project file changes.

The VxWorks project files serve the following purposes:

linkSyms.c
A dynamically generated configuration file (therefore not to be checked in) that includes code from the VxWorks archive by creating references to the appropriate symbols. It contains symbols for components that do not have initialization routines.

prjConfig.c
A dynamically generated configuration file (therefore not to be checked in) that contains initialization code for components included in the current configuration of VxWorks.

romInit.s
Contains the entry code for the VxWorks boot ROM.

romStart.c
Contains the routines to load VxWorks system image into RAM.

sysALib.s
Contains system startup code, the first code executed after booting (which is the entry point for VxWorks in RAM).

sysLib.c
Contains board-specific routines.

userAppInit.c
Contains a stub for adding user application initialization routines for a bootable application.

The following files are created in the main project directory as well, but are not visible in the workspace:

prjComps.h
A dynamically generated configuration file (therefore not to be checked in) that contains the preprocessor definitions (macros) used to include VxWorks components.

Makefile
The makefile used for building an application or VxWorks. Created when the project is built, based on the build specification selected at that time.

prjParams.h
A dynamically generated configuration file (therefore not to be checked in) that contains component parameters.

projectName.**wpj**
Contains information about the project used for generating the project makefile, as well as project source files such as **prjConfig.c**.

workspaceName.**wsp**
Contains information about the workspace, including which projects belong to it.

When you build the project from the GUI, a makefile is dynamically generated in the main project directory, and a subdirectory is created containing the objects produced by the build. The subdirectory is named after the selected build specification. If other build specifications are created and used for other builds, parallel directories are created for their objects.

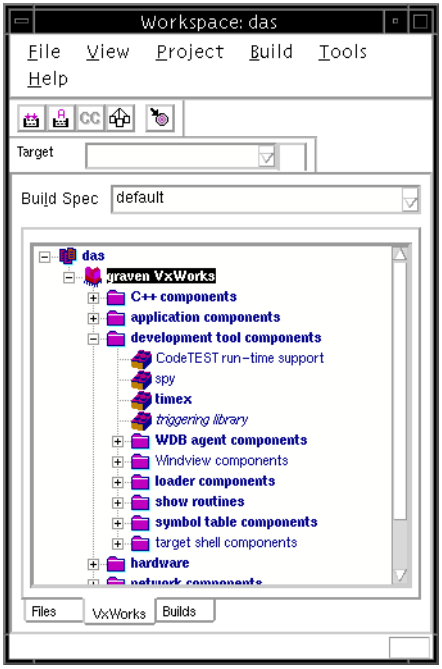
You can also build your project from the command line. When you do so, you must create the makefile first. See *Using the Command Line*, p.146.

The Files view can also display the default list of objects that would be built, and the external dependencies that make up the new project, in the Object Modules and External Dependencies folders, respectively.

4.4.3 Configuring VxWorks Components

The VxWorks view of the Workspace displays all VxWorks components available for the target. The names of components that are selected for inclusion appear in **bold** type. The names of components that are excluded appear in plain type. The names of components that have not been installed appear in *italics*. Note that the names of folders appear in bold type if any (but not necessarily all) of their components are included. (Figure 4-22.)

Figure 4-22 **VxWorks Components**

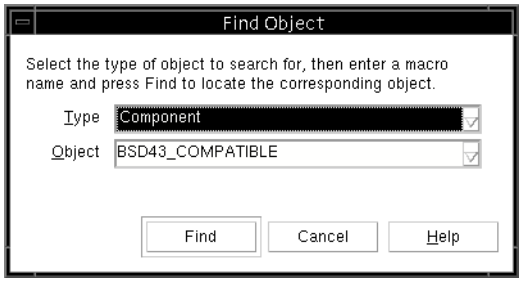


NOTE: See the *VxWorks Programmer's Guide* for detailed information about the use of VxWorks facilities, target-resident tools, and optional components.

Finding VxWorks Components and Configuration Macros

You can locate individual components and configuration parameters in the component tree, based on their macro names, with the Find Object dialog box. Open the dialog box with the pop-up menu for the VxWorks view (Figure 4-23).

Figure 4-23 **Find Object**



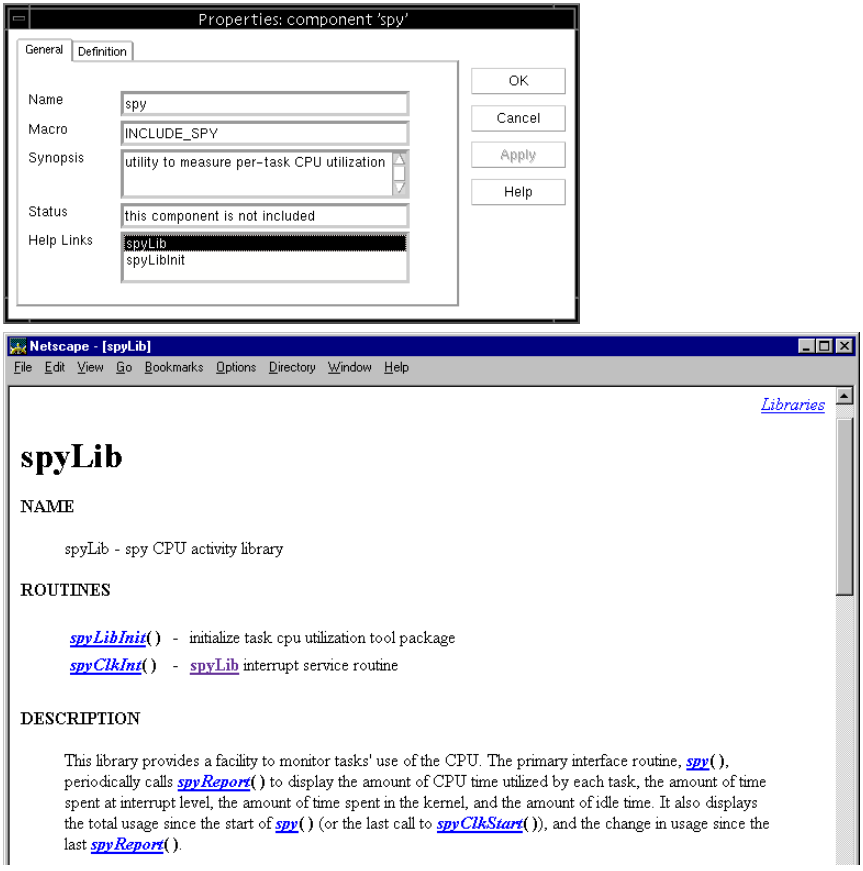
NOTE: The Find Object dialog box is particularly helpful in conjunction with VxWorks documentation, which discusses VxWorks configuration in terms of preprocessor symbols, rather than the descriptive names used in the project facility GUI.

Displaying Descriptions and Online Help for Components

The component tree in the VxWorks view provides descriptive names for components. You can display a component description property sheet, which includes the name of the preprocessor macro for the component, by double-clicking on the component name (Figure 4-24).

To display online reference documentation, double-click on the topic of your choice displayed in the Help Link box of the property sheet. The corresponding HTML reference material is displayed in a Web browser (Figure 4-24).

Figure 4-24 **VxWorks Component Properties and HTML Reference**



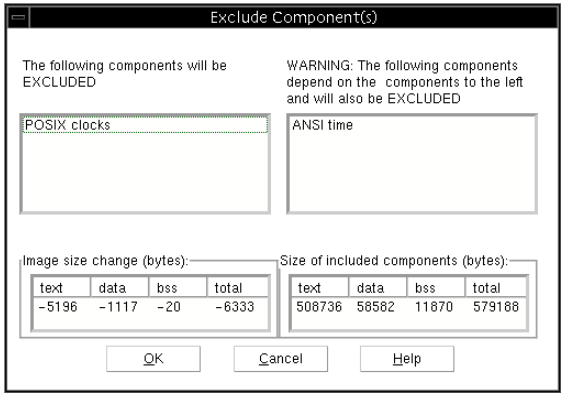
Including and Excluding Components

VxWorks components that are not needed for a project can be excluded, and components that have been excluded can be included again. The pop-up menu provides Include and Exclude options for components you select in the VxWorks view. You can also use the **DELETE** key to exclude options.

Tornado automatically determines component dependencies each time a component is included or excluded. That is, it determines if a component you want to include is dependent upon other components that have not been included in the project, or if a component that you are deleting is required by other components.

When a component is included, any components it requires are automatically included. When a component is excluded, any dependent components are also excluded. In either case, a dialog box provides information about dependencies and the option of cancelling the requested action. For example, if you exclude POSIX clocks, the dialog box informs you that the ANSI time component would be excluded (Figure 4-25).

Figure 4-25 **Exclude VxWorks Component**



WARNING: The results of calculating dependencies is not necessarily identical for inclusion and removal. Including a component you previously excluded does not automatically include the components that were dependent on that component, and that were therefore excluded with it. For example, excluding the POSIX clocks component automatically excludes the ANSI time component, which is *dependent on* it. But if the POSIX clocks component is subsequently included, there are no components *required by* it, so the ANSI time component is not automatically included (Figure 4-26).

You can also include folders of components. However, not all components in a folder are necessarily included by default (nor would it always be desirable to do so, as there might be conflicts between components). Tornado offers a choice about what components to include. For example, if you include target shell components, not all of the components are included by default, and you are prompted to accept or modify the default selection (Figure 4-27).

Tornado automatically calculates an estimate of the change in the size of the image resulting from the inclusion or exclusion, as well as the new image size. The Include

Figure 4-26 **Include VxWorks Component**

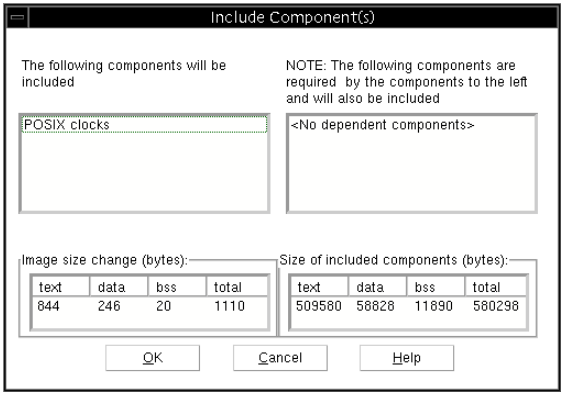
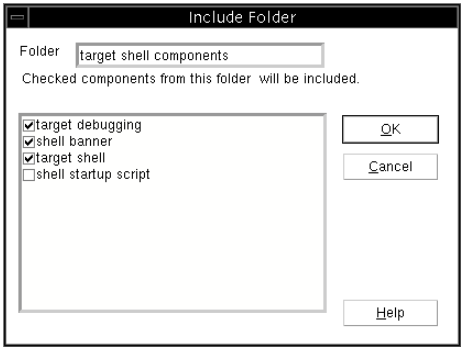


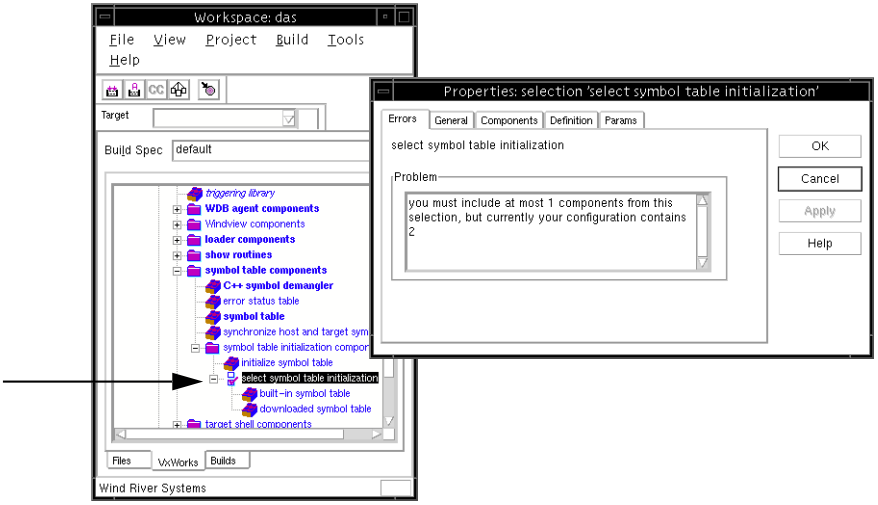
Figure 4-27 **Including a Component Folder**



and Exclude dialog boxes display this information. (Also see *Estimating Total Component Size*, p.142).

Some folders contain component options that are explicitly combinative or mutually exclusive (in the sense of being potentially in conflict). These folders are called selections, and their names are preceded by a checkbox icon in the folder tree. You can make your selection or change either by opening the folder and performing an include or exclude operation on individual components, or by displaying the property sheet for the folder and making selections with the check boxes on the Components page (Figure 4-27).

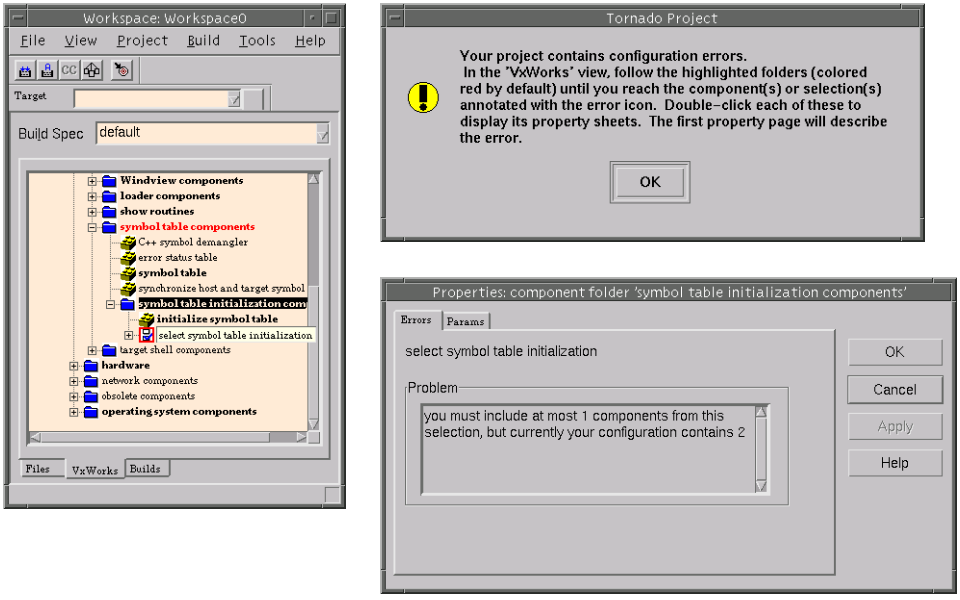
Figure 4-28 **Including Conflicting Components**



Component Conflicts

If you include components that potentially conflict, or are missing a required component, Tornado warns you of the conflict by displaying a message box with a warning, and by highlighting the full folder path to the source of the conflict. The property sheet for the folder also displays error information in its Errors page. For example, if you attempt to include both symbol table initialization components, a warning is first displayed. Once you acknowledge the warning, the folder names development tool components, symbol table components, select symbol table initialization are highlighted. You can display the property sheet for the folder for a description of the problem and how to correct it. (See Figure 4-29 for all GUI elements.)

Figure 4-29 **Component Conflicts**



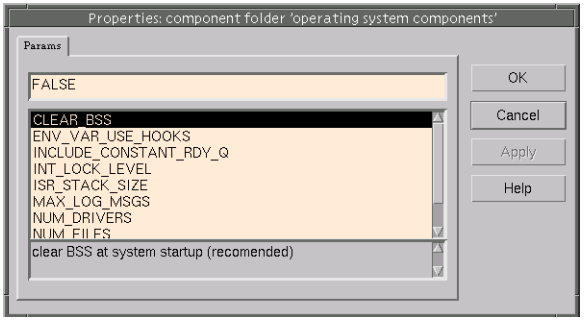
WARNING: You can build VxWorks even if there are conflicts between the components you have selected, but you may have linker errors or the run-time results may be unpredictable.

Changing Component Parameters

In the VxWorks view, the pop-up menu provides access to component parameters (preprocessor macros). For example, selecting the operating system components folder, then Params for 'operating system components' from the pop-up menu (or double-clicking on the folder name), displays a dialog box that allows you to change the values of the parameters defined for the operating system components (Figure 4-30). Parameters specific to individual components can be accessed similarly.

For more information about component parameters, see the *VxWorks Programmer's Guide* and the *VxWorks Network Programmer's Guide*.

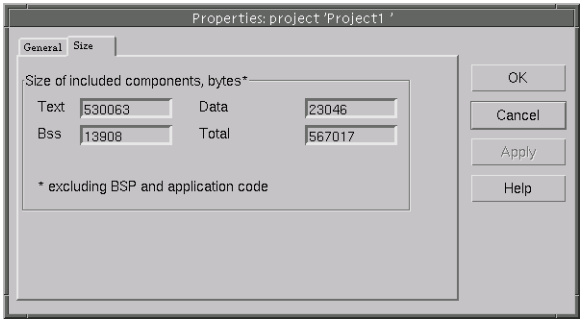
Figure 4-30 **Component Parameters**



Estimating Total Component Size

To calculate and display the estimated size of the components included in an image, select the project name (in any of the workspace views), then select Properties from the pop-up menu, and select the Size tab in the property sheet that appears (Figure 4-31). Note that this estimate is for the components only, and does not include the BSP or any application code.

Figure 4-31 **Total Component Size**



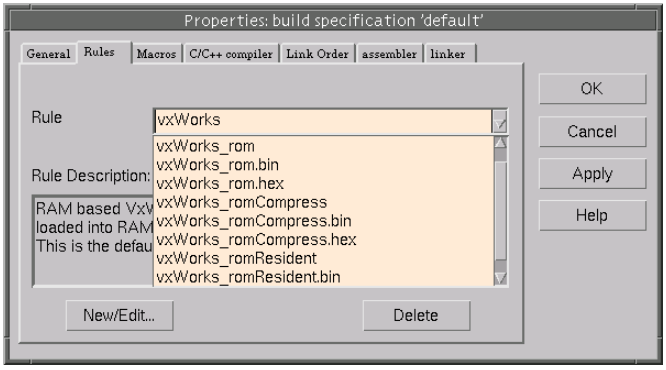
4.4.4 Selecting the VxWorks Image Type

The default VxWorks is a RAM-based image. If you want to create something other than the default images, use one of the other build specifications created when you created your project:

- **default_rom**
- **default_romCompress**
- **default_romResident**

These build specifications are identical to the default, except that instead of building the **vxWorks** rule, **default_rom**, for example, is set up to build the **vxWorks_rom** rule. (Figure 4-32).

Figure 4-32 **Build Rules for VxWorks Images**



The options available for a VxWorks image are:

vxWorks
: A RAM-based image, usually loaded into memory by a VxWorks boot ROM. This is the default development image.

default_rom
: A ROM-based image that copies itself to RAM before executing. This image generally has a slower startup time, but a faster execution time than vxWorks_romResident. It is also available in alternate formats as vxWorks_rom.bin and vxWorks_rom.hex. The .hex options are variants of the main options with Motorola S-Record output. The .bin options are variants of the main options with binary output.

default_romCompress
: A compressed ROM image that copies itself to RAM and decompresses before executing. It takes longer to boot than vxWorks_rom but takes up less space than other ROM-based images (nearly half the size). The run-time execution is the same speed as vxWorks_rom. It is also available in .bin and .hex formats.

default_romResident
: A ROM-resident image. Only the data segment is copied to RAM on startup. It has the fastest startup time and uses the smallest amount of RAM. Typically, however, it runs slower than the other ROM images because ROM access is slower. It is also available in .bin and .hex formats.

NOTE: Project files used only for a ROM-based image can flagged as such, so that they are only used when a ROM-based image is built. See *Compiler Options*, p.151.

4.4.5 Building VxWorks

VxWorks projects are built in the same manner as downloadable applications. To build a project with the default options, select the name of the project (or any subordinate object in its folder) and then select the Build option from the pop-up menu. The name of the build specification that will be used is displayed in the Build Spec drop-down list at the top of the workspace window.

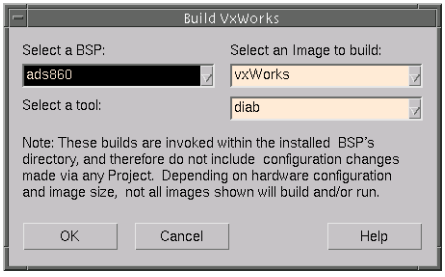
For more information about a generic build, see *4.3.4 Building a Downloadable Application*, p.119. For information about modifying builds and creating new build configurations. see *4.6 Working With Build Specifications*, p.148.

Using the Build Menu

The build menu allows you to choose boot ROMs or BSPs to build. Figure 4-33 illustrates the Build dialog box in a Tornado system that has a PowerPC BSP and the Diab compiler installed.

Select a BSP
: The drop-down list includes all BSPs you have installed. You will have at least one integrated simulator BSP, and you have probably installed at least one additional BSP.

Figure 4-33 **Rebuilding VxWorks from the Build Menu**



Select an Image to build

The available BSP **make** targets appear in the drop down list. For information on the targets, see *Build Specifications*, p.122.

The standard make target **clean** (which erases all objects that can be built by the BSP makefile) is also in this list.

WARNING: Be sure *not* to use **make clean** in your VxWorks tree, in other words, in *installDir*/**target/src**. The **make clean** command is designed to force a complete rebuild of your project files. If you use it in *installDir*/**target/src**, you will remove VxWorks objects for which you do not have source and which you will therefore be unable to recreate.

Select a tool

If you have installed both the GNU and Diab compilers, you can select which one to use.

When you click OK in the build dialog box, Tornado builds the corresponding object in the BSP directory. Output from the build goes to a Build Output window, which you can use as a diagnostic aid.

To rebuild VxWorks, select the vxWorks target name in the Select an Image list. For example, Figure 4-33 shows the vxWorks target selected for the ads860 BSP.

NOTE: All source files in a project are built using a single build specification (which includes a specific set of makefile, compiler, and linker options) at a time. If some of your source requires a different build specification from the rest, you can create a project for it in the same workspace, and customize the build specification for those files. One project's build specification can then be modified to link in the output from the other project. See *Linker Options*, p.154.

WARNING: The default compiler options include debugging information. Using debugging with the optimization set to anything but zero may produce unexpected results. See *4.6 Working With Build Specifications*, p.148 for information about modifying builds and creating new build configurations.

Using the Command Line

Using the command line allows you to automate builds. Projects must be created and configured in the GUI, and dependencies must be generated there as well. However, makefile generation and building are possible from the command line. To generate the makefile, use the **makegen** utility and to build, use **make**:

- Change to the build directory and type **make** with flags, for example:

```
% cd installDir/target/proj/Project1/mbx860_gnu
% makegen
% mkdir default_rom
% cd default_rom
% make -f ../Makefile BUILDSPEC=default_rom
```

  Your build output will be in the **default_rom** directory and the build will use the **default_rom** build specification.

4.4.6 Booting VxWorks

For information about booting VxWorks (and bootable applications), see *2.6 Booting VxWorks*, p.46. VxWorks images for the target simulator can be downloaded and booted with the pop-up menu Start command.

4.5 Creating a Bootable Application

A bootable application is completely initialized and functional after a target has been booted, without requiring interaction with Tornado development tools. For information about developing your application, see *4.2.3 Developing and Adding Your Application Source Code*, p.102.

4.5.1 Using Automated Scaling of VxWorks

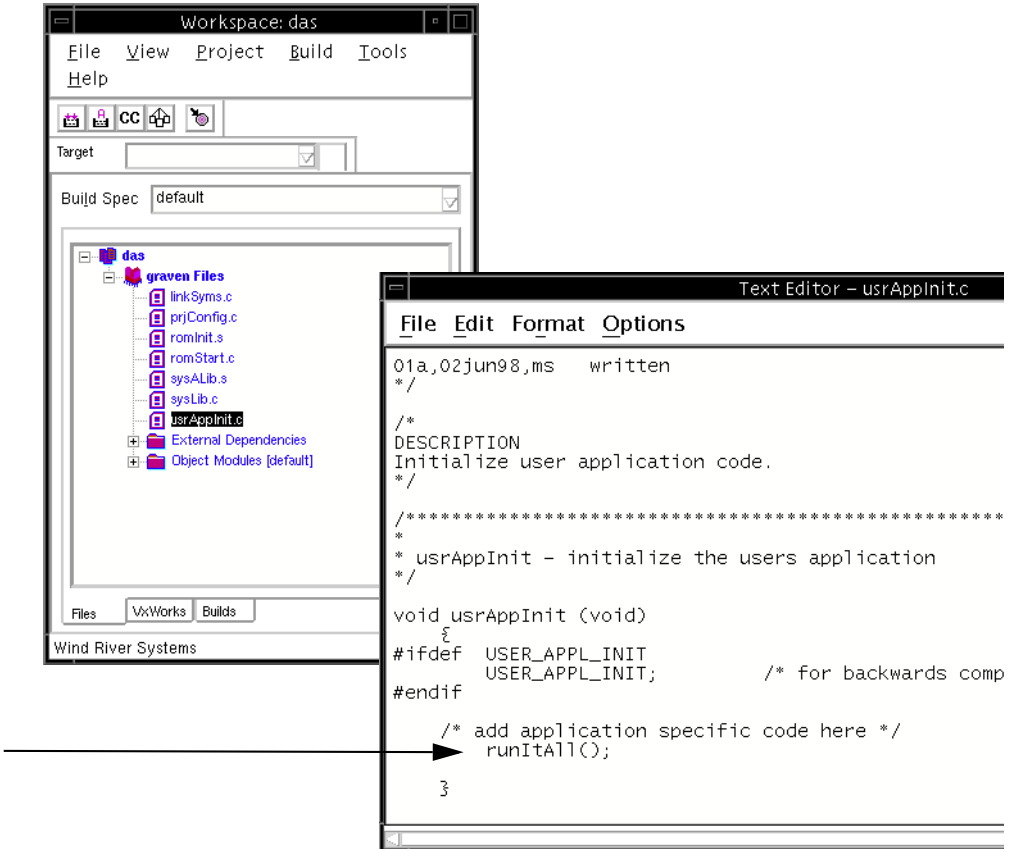
The auto scale feature of the project facility determines if your code, or your custom version of VxWorks, requires any components that are not included in your VxWorks project, and adds them as required. It also provides information about components that *may* not be required for your application. To automatically scale VxWorks, select Auto Scale from the pop-up menu in the VxWorks view of the workspace window to display the Auto Scale dialog box, and click OK.

NOTE: The auto scale feature detects only statically calculable dependencies between the application code and VxWorks. Some components may be needed even if they are not called by your application. This is especially true for servers such as WDB, NFS, and so on.

4.5.2 Adding Application Initialization Routines

When VxWorks boots, it initializes operating system components (as needed), and then passes control to the user's application for initialization. To add application initialization calls to VxWorks, double-click on **userAppInit.c** to open the file for editing, and add the call(s) to **usrAppInit( )**. Figure 4-34, for example, illustrates the addition of a call to **runItAll( )**, the main routine in the application file **helloWorld.c**.

Figure 4-34 **Adding Application Initialization Calls to VxWorks**



4.6 Working With Build Specifications

The project facility allows you to create, modify, and select specifications for any number of builds. Default build specifications are defined when you create your project.

- While a BSP is usually designed for one CPU, you can create build specifications for different image types and optimization levels, specifications

for builds that include debugging information and builds that do not, and so on.

- A downloadable project can have build specifications for different CPUs, for example, for a simulator and a real target.

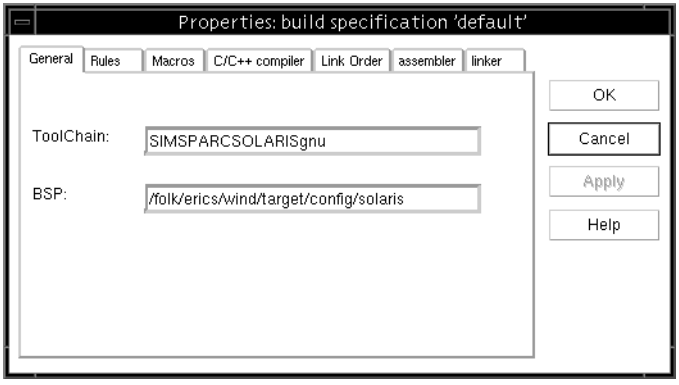
4.6.1 Changing a Build Specification

Each build specification consists of a set of options that define the VxWorks image type (for VxWorks and bootable application projects), makefile rules, macros, as well as compiler, assembler, and linker options.

NOTE: For detailed information about compiler, assembler, and linker options, see the *GNU ToolKit User's Guide* or the *Diab C/C++ Compiler User's Guide*.

You can change default or other previously defined build options by double-clicking on the build name in the Builds view of the workspace window. The build's property sheet appears (Figure 4-35).

Figure 4-35 **Build Property Sheet**



You can use the property sheet to modify:

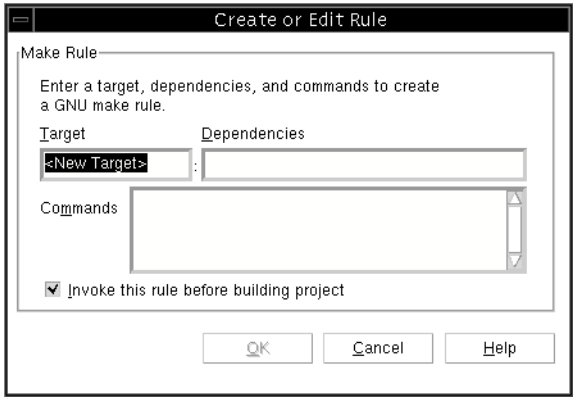
- build targets
- makefile rules
- makefile macros for the compiler and linker
- compiler options
- assembler options
- linker options

For information about build targets for downloadable applications, see *Build Specifications*, p.122. For information about build targets for bootable applications, see *4.4.4 Selecting the VxWorks Image Type*, p.143. Other features of the build property sheet are covered in the following sections.

Custom Makefile Rules

The buttons at the bottom of the build property sheet allow you to create, edit, or delete makefile rules (default project entries cannot be deleted; only those created by a user can be deleted). When you click the New/Edit button, the Create or Edit Rule dialog box appears (Figure 4-36). Once you have created or edited an entry, click OK. Note that the default is to invoke the rule before building the project (see the checkbox). If the default is not selected, the rule is only invoked if it is the rule currently selected for the build (with the drop-down list in the Rules page of the build property sheet) or if it is a dependency of the currently selected rule. New rules are added to the *projectName***.wpj** file and written to the makefile prior to a build.

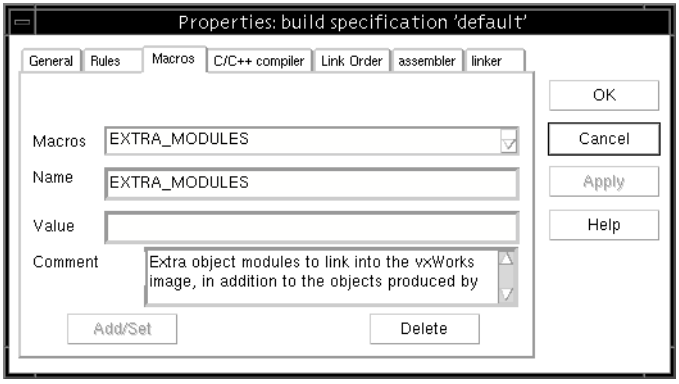
Figure 4-36 **Makefile Rule**



Makefile Macros

Select the Macros tab of the build specification property sheet to view the makefile macros associated with the current project, build specification, and rules (Figure 4-37).

Figure 4-37 **Makefile Macros**



You can use the Macros page to modify the values of existing makefile macros, as well as to create new rules to be executed at the end of the build. Use the Delete button to delete a macro from the build. To change an existing macro, modify the value and click the Add/Set button. To add a macro, change the name and value of an existing macro, and click the Add/Set button.

The recommended way to link library (archive) files to your bootable project is to add the libraries to the list defined by the **LIBS** macro and the modules to the list defined by the **EXTRA_MODULES** macro. Use the **PRJ_LIBS** macro for downloadable projects.

Compiler Options

The C/C++ compiler page of the build specification property sheet displays compiler options. You can edit the options displayed in the text box (Figure 4-38). You can also click the Include Paths button and use the dialog box to enter and order your include paths.

Figure 4-38 **Compiler Options**

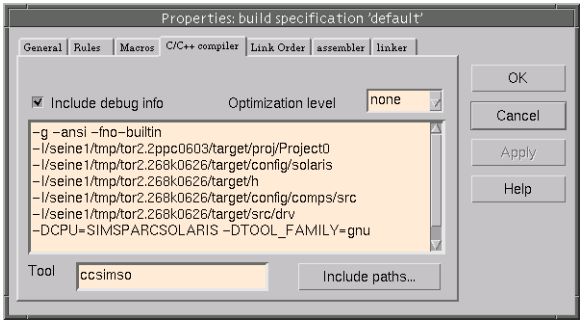
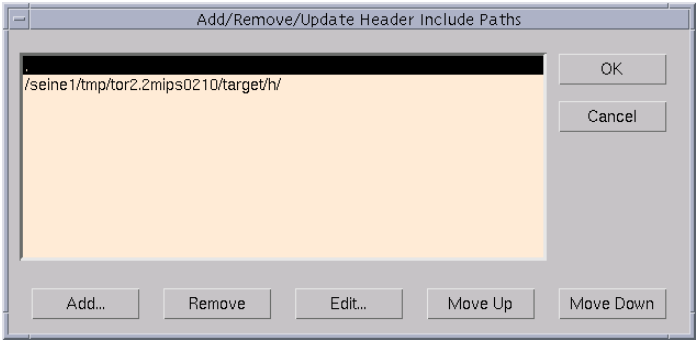


Figure 4-39 **Include Paths Dialog Box**

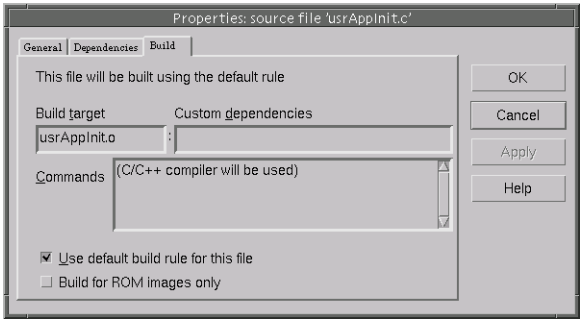


WARNING: The default compiler options include debugging information. Using debug information with the optimization option set to anything but zero may produce unpredictable results. Selecting Include debug info automatically sets optimization to zero. This can be changed by editing the option.

You can override the default compiler flags for individual files by right-clicking on the file name in the Files view, selecting Properties from the pop-up menu, and specifying a new set of options in the Build page of the property sheet. Un-checking the Use default build rule for this file box allows you to edit the fields in this page (Figure 4-40).

If the file should be used only when building a ROM-based image, check the Build for ROM images only box. See *4.4.4 Selecting the VxWorks Image Type*, p.143.

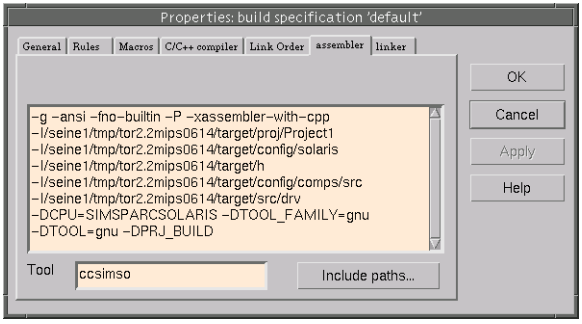
Figure 4-40 **Compiler Options for Individual Files**



Assembler Options

Select the assembler tab of the build specification property sheet to view assembler options. You can edit the options displayed in the text box (Figure 4-41).

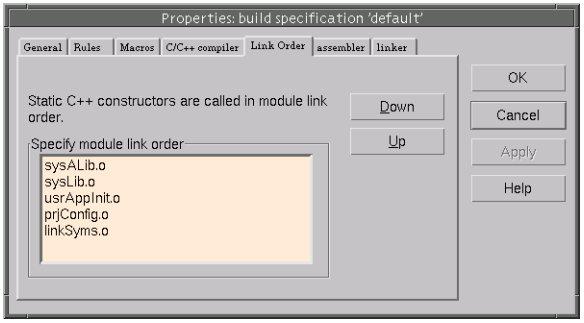
Figure 4-41 **Assembler Options**



Link Order Options

Select the Link Order tab of the build specification property sheet to view module link order (Figure 4-42). You can change the link order using the Down and Up buttons to ensure that static C++ constructors and destructors are invoked in the correct order.

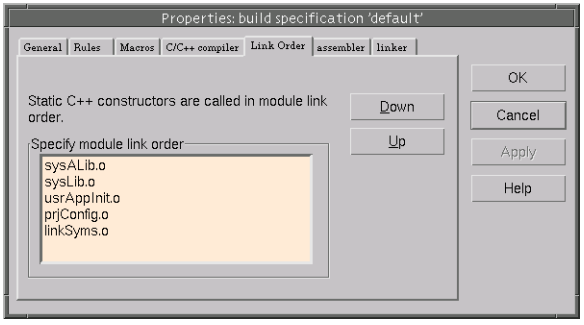
Figure 4-42 **Link Order Options**



Linker Options

Select the linker tab of the build specification property sheet to view linker options. You can edit the options displayed in the text box (Figure 4-43).

Figure 4-43 **Linker Options**



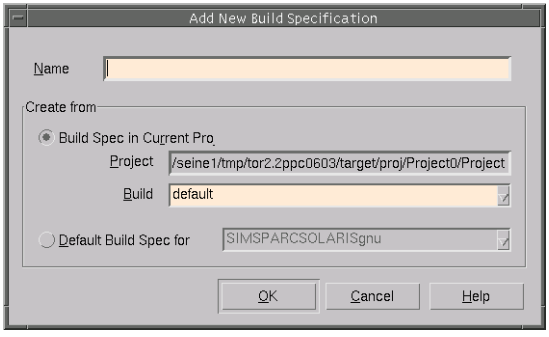
To link an object or library (archive) file with a project, you can list the full path to the file here. However, the recommended way to link library (archive) files is to add the libraries to the list defined by the **LIBS**, **EXTRA_MODULES**, or **PRJ_LIBS** macros on the Macros tab (see *Makefile Macros*, p.150).

WARNING: You cannot link another project object file (*projectName***.out**) with the project you are building. You must compile the other project as a library (*Build Specifications*, p.122) or as a partial link (*projectName***.pl**), and then link it with the current project.

4.6.2 Creating New Build Specifications

You can create new build specifications for a project with the Add New Build Specification window, which is displayed with the New Build option on the pop-up menu. For example, one build specification can be created that includes debug information, and another that does not; specifications can be created for different image types, optimization levels, and so on. You can create a new build specification by copying from an existing specification, or by creating it as a default specification for a given toolchain (Figure 4-44).

Figure 4-44 **New Build Specification**



Once you have created a new build specification, use the build specification property sheet to define it (see *4.6.1 Changing a Build Specification*, p.149).

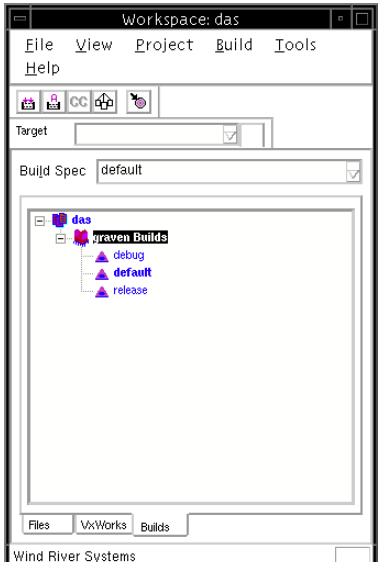
NOTE: Within a bootable project, you are restricted to the toolchains that support the CPU required by the BSP. You can still create multiple build specifications (for example, with different optimization levels or rules).

4.6.3 Selecting a Specification for the Current Build

When you want to build your project, select the build specification from the Build Spec drop-down list (Figure 4-45).

Binaries produced by a build are created in the *buildName* subdirectory of your project directory.

Figure 4-45 **Build Specification Selection**



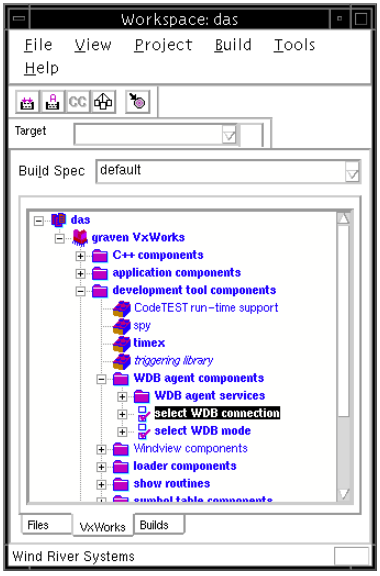
4.7 Configuring the Target-Host Communication Interface

WARNING: During development you must configure VxWorks with the target agent communication interface required for the connection between your host and target system (network, serial, NetROM, and so on). By default, VxWorks is configured for a network connection. Also note that before you use Tornado tools such as the shell and debugger, you must start a target server that is configured for the same mode of communication. See *2.4.2 Networking the Host and Target*, p.25; and *3.5.1 Configuring a Target Server*, p.73.

To display the options for the communication interface for the target agent in the VxWorks view, select development tool components>WDB agent components>select WDB connection (Figure 4-46).

To select an interface, select it from the list and select the Include '*component name*' option from the pop-up menu. (You can also make a selection by double-clicking on the select WDB connection option to display the property sheet, and then making the selection from the Components page.)

Figure 4-46 **Target Agent Connection Options**



To display general information about a component, or to change its parameters, simply double-click on its name, which displays its property sheet (see Figure 4-47). The options for the target agent communication interface are described below.

NOTE: Also see *Scaling the Target Agent*, p.161 and *Starting the Agent Before the Kernel*, p.162.

Configuration for an END Driver Connection

When VxWorks is configured with the standard network stack, the target agent can use an END (Enhanced Network driver) connection. Add the WDB END driver connection component. This connection has the same characteristics as the network connection, but also has a polled network interface that allows system and task mode debugging.

Configuration for Integrated Target Simulators

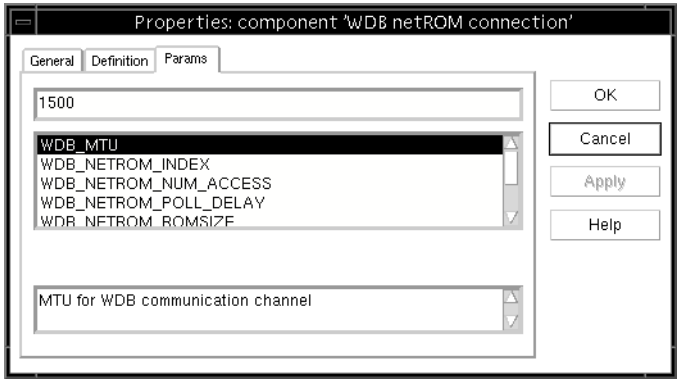
To configure a target agent for an image that will run with the VxWorks integrated target simulator, add the WDB simulator pipe connection component.

Configuration for NetROM Connection

To configure the target agent to use a NetROM communication path, add the WDB netROM connection component. (See *2.5.4 The NetROM ROM-Emulator Connection*, p.35).

Several configuration macros are used to describe a board's memory interface to its ROM banks. You may need to override some of the default values for your board. To do this, display the component property sheet, and select the Params tab to display and modify macro values (Figure 4-47).

Figure 4-47 **NetROM Connection Macros**



WDB_NETROM_MTU
The default is 1500 octets.

WDB_NETROM_INDEX
The value 0 indicates that pod zero is at byte number 0 within a ROM word.

WDB_NETROM_NUM_ACCESS
The value 1 indicates that pod zero is accessed only once when a word of memory is read.

WDB_NETROM_POLL_DELAY
The value 2 specifies that the NetROM is polled every two VxWorks clock ticks to see if data has arrived from the host.

WDB_NETROM_ROMSIZE
The default value is **ROM_SIZE**, a makefile macro that can be set for a specific build. See *Makefile Macros*, p.150.

WDB_NETROM_TYPE
The default value of 400 specifies the old 400 series.

WDB_NETROM_WIDTH
The value 1 indicates that the ROMs support 8-bit access. To change this to 16- or 32-bit access, specify the value 2 or 4, respectively.

The size of the NetROM dual-port RAM is 2 KB. The NetROM permits this 2 KB buffer to be assigned anywhere in the pod 0 memory space. The default position for the NetROM dual-port RAM is at the end of the pod 0 memory space. The following line in *installDir*/**target/src/config/usrWdb.c** specifies the offset of dual-port RAM from the start of the ROM address space.

```
dpOffset = (WDB_ROM_SIZE - DUALPORT_SIZE) * WDB_NETROM_WIDTH;
```

If your board has more than one ROM socket, this calculation gives the wrong result, because the VxWorks macro **ROM_SIZE** describes the total size of the ROM space—not the size of a single ROM socket. In that situation, you must adjust this calculation.

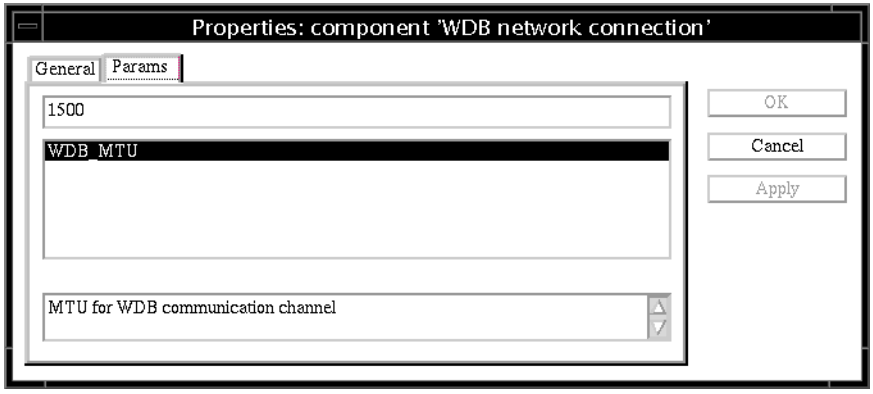
Refer to the NetROM documentation for more information on the features governed by these parameters.

Configuration for Network Connection

To configure the target agent for use with a network connection, add the WDB network connection component. (See *2.5.1 Network Connections*, p.31).

The default MTU is 1500 octets. To change it, display the component property sheet, select the Params tab, select the WDB_MTU item and change the value associated with it (Figure 4-48).

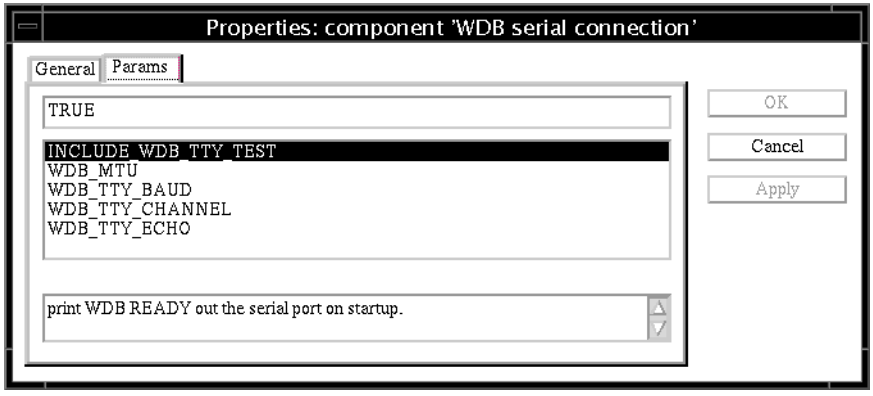
Figure 4-48 **Network Connection Macro**



Configuration for Serial Connection

To configure the target agent to use a raw serial communication path, add the WDB serial connection component. (See *2.5.3 Serial-Line Connections*, p.32).

Figure 4-49 **Serial Connection Macros**



By default, the agent uses serial channel 1 at 9600 bps.[7] For better performance, use the highest line speed available, which is often 38400 bps. Try a slower speed if you suspect data loss. To change the speed, display the component property sheet, select the Params tab, select WDB_TTY_BAUD and change the value associated with it.

If your target has a single serial channel, you can use the target server virtual console to share the channel between the console and the target agent. You must configure your project with the **CONSOLE_TTY** parameter set to **NONE** and the **WDB_TTY_CHANNEL** parameter set to 0. See *Target-Server Configuration Options*, p.76 for more information regarding the target server virtual console.

When multiplexing the virtual console with WDB communications, excessive output to the console may lead to target server connection failures. The following actions may help resolve this problem:

- Decrease the amount of data being transmitted to the virtual console from your application.
- Increase the timeout period of the target server (see *Target-Server Configuration Options*, p.76).
- Increase the baud rate of the target agent and the target server connection.

Configuration for tyCoDrv Connection

To configure a target agent to use a serial connection, add the WDB tyCoDrv connection component. Display the component property sheet and select the Params tab to display and modify macro values.

Scaling the Target Agent

In a memory-constrained system, you may wish to create a smaller agent. To reduce program text size, you can remove the following optional agent facilities:

- WDB banner (**INCLUDE_WDB_BANNER**)
- VIO driver (**INCLUDE_WDB_VIO**)
- WDB task creation (**INCLUDE_WDB_START_NOTIFY**)

7. VxWorks serial channels are numbered starting at zero. Thus Channel 1 corresponds to the second serial port if the board's ports are labeled starting at 1. If your board has only one serial port, you must change **WDB_TTY_CHANNEL** to 0 (zero).

- WDB user event (**INCLUDE_WDB_USER_EVENT**)

These components are in the development tool components>WDB agent components>WDB agent services folder path.

You can also reduce the maximum number of WDB breakpoints with the **WDB_BP_MAX** parameter of the WDB breakpoints component. If you are using a serial connection, you can also set the **INCLUDE_WDB_TTY_TEST** parameter to **FALSE**.

If you are using a communication path which supports both system and task mode agents, then by default both agents are started. Since each agent consumes target memory (for example, each agent has a separate execution stack), you may wish to exclude one of the agents from the target system. You can configure the target to use only a task-mode or only a system-mode agent with the WDP task debugging or WDB system debugging options (which are in the folder path development tool components>WDB agent components>select WDB mode).

Configuring the Target Agent for Exception Hooks

If your application (or BSP) uses **excHookAdd( )** to handle exceptions, host tools will still be notified of *all* exceptions (including the ones handled by your exception hook). If you want to suppress host tool notifications, you must exclude the component WDB exception notification. When this component is excluded, the target server is not notified of target exceptions, but the target will still report them in its console. In addition, if an exception occurs in the WDB task, the task will be suspended and the connection between the target server and the target agent will be broken.

Starting the Agent Before the Kernel

By default, the target agent is initialized near the end of the VxWorks initialization sequence. This is because the default configuration calls for the agent to run in task mode and to use the network for communication; thus, **wdbConfig( )** must be called after **kernelInit( )** and **usrNetInit( )**. (See *F. VxWorks Initialization Timeline* for an outline of the overall VxWorks initialization sequence.)

In some cases (for example, if you are doing a BSP port for the first time), you may want to start the agent before the kernel starts, and initialize the kernel under the control of the Tornado host tools. To make that change, perform the following steps when you configure VxWorks:

1. Choose a communication path that can support a system-mode agent (NetROM or raw serial). The END communication path cannot be used as it requires that the system be started before it is initialized.
2. Change your configuration so that only WDB system debugging is selected (in folder path development tool components>WDB agent components>select WDB mode). By default, the task mode starts two agents: a system-mode agent and a task-mode agent. Both agents begin executing at the same time, but the task-mode agent requires the kernel to be running.
3. Create a configuration descriptor file called *fileName***.cdf** (for example, **wdb.cdf**) in your project directory that contains the following lines:

```
InitGroup usrWdbInit {
    INIT_RTN    usrWdbInit (); wdbSystemSuspend ();
    _INIT_ORDER usrInit
    INIT_BEFORE INCLUDE_KERNEL
}
```

   This causes the project code generator to make the **usrWdbInit( )** call earlier in the initialization sequence. It will be called from **usrInit( )**, just before the component kernel is started.[8]

After the target server connects to the system-mode target agent, you can resume the system to start the kernel under the agent's control. (See *7.2.7 Using the Shell for System Mode Debugging*, p.267 for information on using system mode from the shell, and *9.5 System-Mode Debugging*, p.362 for information on using it from the debugger.)

After connecting to the target agent, set a breakpoint in **usrRoot( )**, then continue the system. The routine **kernelInit( )** starts the multi-tasking kernel with **usrRoot( )** as the entry point for the first task. Before **kernelInit( )** is called, interrupts are still locked. By the time **usrRoot( )** is called, interrupts are unlocked.

Errors before reaching the breakpoint in **usrRoot( )** are most often caused by a stray interrupt: check that you have initialized the hardware properly in the BSP **sysHwInit( )** routine. Once **sysHwInit( )** is working properly, you no longer need to start the agent before the kernel.

8. The code generator for **prjConfig.c** is based on a component descriptor language that specifies when components are initialized. The component descriptors are searched in a specific order, with the project directory last in the search path. This allows the **.cdf** files in the project directory to override default definitions in the generic **.cdf** files.

⚠ **CAUTION:** When the agent is started before the kernel, there is no way for the host to get the agent's attention until a breakpoint occurs. There are two reasons for this: 1) For the NetROM connection, the agent cannot spawn the NetROM polling task to check periodically for incoming packets from the host. 2) For other types of connections, only system mode is supported and the WDB communication channel is set to work in polled mode only. On the other hand, the host does not really need to get the agent's attention: you can set breakpoints in **usrRoot( )** to verify that VxWorks can get through this routine. Once **usrRoot( )** is working, you can start the agent after the kernel (that is, within **usrRoot( )**), after which the polling task is spawned normally.

⚠ **WARNING:** If you are using the serial connection, take care that your serial driver does not cause a stray interrupt when the kernel is started, because the serial-driver interrupt handlers are not installed until after **usrRoot( )** begins executing: the calling sequence is **usrRoot( )** ➠ **sysClkConnect( )** ➠ **sysHwInit2( )** ➠ **intConnect( )**. You may want to modify the driver so that it does not set a channel to interrupt mode until the hardware is initialized. This can be done by setting a flag in the BSP after serial interrupts are connected.

4.8 Configuring and Building a VxWorks Boot Program

The default boot image included with Tornado for your BSP is configured for a networked development environment. The boot image consists of a minimal VxWorks configuration and a boot loader mechanism. You need to configure and build a new boot program (and install it on your boot medium) if:

- You are working with a target that is not on a network.
- You do not have a target with NVRAM, and do not want to enter boot parameters at the target console each time it boots.
- You want to use an alternate boot process, such as booting over the Target Server File System (TSFS).

WARNING: Configuration of boot programs is handled independently of the project facility so normally you can configure your boot ROM differently than your other images. However, any changes you make to **config.h** that are not masked by project facility selections may be absorbed by your projects (see the warning in *4.1 Introduction*, p.93). To prevent this, copy your BSP and create your boot image from the copy.

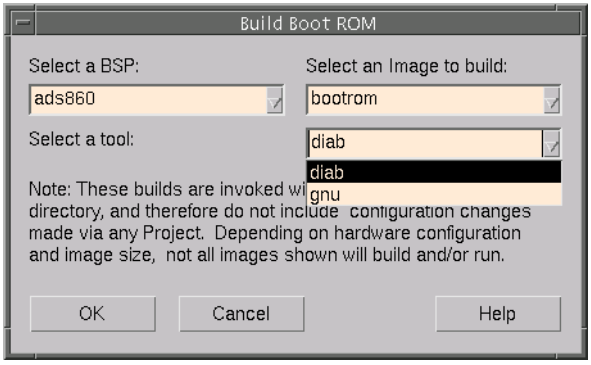
Configuring Boot Parameters

To customize a boot program for your development environment, you must edit *installDir*/**target/config/***bspname*/**config.h** (the configuration file for your BSP). The file contains the definition of **DEFAULT_BOOT_LINE**, which includes parameters identifying the boot device, IP addresses of host and target, the path and name of the VxWorks image to be loaded, and so on. For information about the boot line parameters defined by **DEFAULT_BOOT_LINE**, see *2.6.4 Description of Boot Parameters*, p.50 and Help>Manuals contents>VxWorks Reference Manual> Libraries>bootLib.

Building a Boot Image

To build the new boot program, select Build>Build Boot ROM from the Workspace window. Select the BSP for which you want to build the boot program and the type of boot image in the Build Boot ROM dialog box (Figure 4-50). Then click OK.

Figure 4-50 **Build Boot ROM**



The three main options for boot images are:

bootrom
: A compressed boot image.

bootrom_uncmp
: An uncompressed boot image.

bootrom_res
: A ROM-resident boot image.

TSFS Boot Configuration

The simplest way to boot a target that is not on a network is over the TSFS (which does not involve configuring SLIP or PPP). The TSFS can be used to boot a target connected to the host by one or two serial lines, or a NetROM connection.

WARNING: The TSFS boot facility is not compatible with WDB agent network configurations. See *4.7 Configuring the Target-Host Communication Interface*, p.156.

To configure a boot program for TSFS, edit the boot line parameters defined by **DEFAULT_BOOT_LINE** in **config.h** (or change the boot parameters at the boot prompt). The "boot device" parameter must be **tsfs**, and the file path and name must be relative to the root of the host file system defined for the target server (see *Configuring Boot Parameters*, p.165 and *Target-Server Configuration Options*, p.76).

Regardless of how you specify the boot line parameters, you must reconfigure (as described below) and rebuild the boot image.

If two serial lines connect the host and target (one for the target console and one for WDB communications), **config.h** must include the following lines:

```
#undef CONSOLE_TTY
#define CONSOLE_TTY         0
#undef WDB_TTY_CHANNEL
#define WDB_TTY_CHANNEL     1
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_SERIAL
#define INCLUDE_TSFS_BOOT
```

If one serial line connects the host and target, **config.h** must include the following lines:

```
#undef CONSOLE_TTY
#define CONSOLE_TTY         NONE
#undef WDB_TTY_CHANNEL
#define WDB_TTY_CHANNEL     0
```

```
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_SERIAL
#define INCLUDE_TSFS_BOOT
```

For a NetROM connection, **config.h** must include the following lines:

```
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_NETROM
#define INCLUDE_TSFS_BOOT
```


With any of these TSFS configurations, you can also use the target server console to set the boot parameters by defining the **INCLUDE_TSFS_BOOT_VIO_CONSOLE** macro in **config.h**. This disables the auto-boot mechanism, which might otherwise boot the target before the target server could to start its virtual I/O mechanism. (The auto-boot mechanism is similarly disabled when **CONSOLE_TTY** is set to **NONE**, or when **CONSOLE_TTY** is set to **WDB_TTY_CHANNEL**.) Using the target server console is particularly useful for a single serial connection, as it provides an otherwise unavailable means of changing boot parameters from the command line.

When you build the boot image, select bootrom.hex for the image type (*Building a Boot Image*, p.165).

See the *VxWorks Programmer's Guide: Local File Systems* for more information about the TSFS.

4.9 Building a Custom Boot ROM

If your boot strategy utilizes a boot ROM, and this boot ROM requires a new driver, you will need to rebuild the boot ROM. Boot ROMs are not yet fully supported as projects in Tornado 2.2. To build a boot ROM, Select Build>Build Boot Rom from the Tornado main menu bar. From the dialog, select the BSP and boot ROM target you wish to build.

If the boot ROM you wish to build is not shown, do the following:

1. Enable extended build options by using the Tools>Options menu from the main menu bar to bring up the Tools Options dialog box. Select the Project pane and select the appropriate check box.
2. Invoke the Build>Customize menu item to bring up the custom build dialog. Click the Add button to bring up a template dialog. Enter menu text (for

example, "Build My boot ROM"), the name of the boot ROM image (for example, **bootrom.hex**), and the BSP directory (for example, *installDir*/**target/config/mv162** for a Windows host).

3. Close the dialog, return to the Build menu, and invoke the newly created menu item. This will build the boot ROM image in the BSP directory.

You can also use the command line as described in *Using the Command Line*, p.146.

5

Command-Line Configuration and Build

5.1 Introduction

The Tornado distribution includes several VxWorks system images for each target shipped. (See *4.4.4 Selecting the VxWorks Image Type*, p.143.) A *system image* is a binary module that can be booted and run on a target system. The system image consists of all desired system object modules linked together into a single non-relocatable object module with no unresolved external references.

In most cases, you will find the supplied system image entirely adequate for initial development. However, later in the cycle you may want to configure the operating system to reflect your application requirements.

This chapter describes in detail the manual cross-development procedures used to create and run VxWorks systems and applications as well as how to configure the system by directly editing configuration files.

The following topics are included:

- Building, loading, running, and unloading VxWorks applications manually.
- Using VxWorks configuration files and configuration options and parameters.
- Creating common alternative configurations of VxWorks.
- Rebuilding VxWorks system images, bootable applications, and ROM images using manual methods.

There are two approaches to system configuration in Tornado 2.2:

- Use the project facility and the GUI

  You can use the project facility for configuring and building, with or without a command-line or automated build. If this is your choice, you do not need any of the information in this chapter. See *4. Projects*.

- Edit configuration files and build from the command line

  The remainder of this chapter summarizes the steps and issues involved in this choice.

WARNING: Use of the project facility for configuring and building applications is largely independent of the methods used prior to Tornado 2.x (which included manually editing the configuration file **config.h**). The project facility provides the recommended and simpler means for configuration and build, although the manual method as described in this chapter may still be used.

To avoid confusion and errors, the two methods should not be used together for the same project. The one exception is for any configuration macro that is not exposed through the project facility GUI (which may be the case, for example, for some BSP driver parameters). In this case, a configuration file must be edited, and the project facility will implement the change in the subsequent build.

Note that the project facility overrides any changes made to a macro in **config.h** that is also exposed through the project facility. If you are using the project facility, only edit macros in **config.h** which can not be configured through the project facility.

VxWorks has been ported to numerous target systems and can support many different hardware configurations. Some of the cross-development procedures discussed in this chapter depend on the specific system and configuration you are running. The procedures in this chapter are presented in generic form, and may differ slightly on your particular system.

For information specific to an architecture family, see the appropriate *VxWorks Architecture Supplement*. Information specific to particular target boards is provided with each BSP.

5.2 Building, Loading, and Unloading Application Modules

In the Tornado development environment, application modules for the target system are created and maintained on a separate development host. First, the source code, generally in C or C++, is edited and compiled to produce a relocatable object module. Application modules use VxWorks facilities by virtue of including header files that define operating- system interfaces and data structures. The

resulting object modules can then be loaded and dynamically linked into a running VxWorks system over the network.

The procedure for configuring a customized VxWorks image is described in *5.3 Configuring VxWorks*, p.186. In the interim, you can use the default images shipped with Tornado.

The following sections describe in detail the procedures for carrying out cross-development manually (without using the project facility).

5.2.1 Using VxWorks Header Files

Many application modules make use of VxWorks operating system facilities or utility libraries. This usually requires that the source module refer to VxWorks *header files*. The following sections discuss the use of VxWorks header files.

VxWorks header files supply ANSI C function prototype declarations for all global VxWorks routines. The ANSI C prototypes are conditionally compiled; to use them, the preprocessor constant **__STDC__** must be defined. ANSI C compilers define this constant by default. VxWorks provides all header files specified by the ANSI X3.159-1989 standard.

VxWorks system header files are in the directory *installDir*/**target/h** and its subdirectories.

NOTE: The notation **$(WIND_BASE)** is used in makefiles to refer to the Tornado installation directory (*installDir*).

VxWorks Header File: vxWorks.h

The header file **vxWorks.h** contains many basic definitions and types that are used extensively by other VxWorks modules. Many other VxWorks header files require these definitions. Thus, this file must be included first by every application module that uses VxWorks facilities. Include **vxWorks.h** with the following line:

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

Other VxWorks Header Files

Application modules can include other VxWorks header files as needed to access VxWorks facilities. For example, an application module that uses the VxWorks linked-list subroutine library must include the **lstLib.h** file with the following line:

```
#include "lstLib.h"
```

The API reference entry for each library lists all header files necessary to use that library.

ANSI Header Files

All ANSI-specified header files are included in VxWorks. Those that are compiler-independent or more VxWorks-specific are provided in *installDir*/**target/h** while a few that are compiler-dependent (for example **stddef.h** and **stdarg.h**) are provided by the compiler installation. Each toolchain knows how to find its own internal headers; no special compile flags are needed.

Many familiar UNIX header files are available under VxWorks. In one case the VxWorks file name differs from the usual UNIX name: **a_out.h** is the VxWorks equivalent of the UNIX **a.out.h**.

ANSI C++ Header Files

Each Wind River compiler has its own C++ libraries and C++ headers (such as **iostream** and **new**). The C++ headers are located in the compiler installation directory rather than in *installDir*/**target/h**. No special flags are required to enable the compilers to find these headers.

NOTE: In previous Tornado releases we recommended the use of the flag **-nostdinc**. This flag *should not* be used with the current release since it prevents the compilers from finding headers such as **stddef.h**. In this release, host header files will not be pulled in even though **-nostdinc** is not used.

The -I Compiler Flag

By default, the compiler searches for header files first in the directory of the source module and then in its internal subdirectories. In general, *installDir*/**target/h**

should always be searched before the compilers' other internal subdirectories; to ensure this, always use the following flag for compiling under VxWorks:

```
-I $(WIND_BASE)/target/h
```

Some header files are located in subdirectories. To refer to header files in these subdirectories, be sure to specify the subdirectory name in the include statement, so that the files can be located with a single **-I** specifier. For example:

```
#include "vxWorks.h"
#include "sys/stat.h"
```


VxWorks Nested Header Files

Some VxWorks facilities make use of other, lower-level VxWorks facilities. For example, the *tty* management facility uses the ring buffer subroutine library. The *tty* header file **tyLib.h** uses definitions that are supplied by the ring buffer header file **rngLib.h**.

It would be inconvenient to require you to be aware of such include-file interdependencies and ordering. Instead, all VxWorks header files explicitly include all prerequisite header files. Thus, **tyLib.h** itself contains an include of **rngLib.h**. (The one exception is the basic VxWorks header file **vxWorks.h**, which all other header files assume is already included.)

Generally, explicit inclusion of prerequisite header files can pose a problem: a header file could get included more than once and generate fatal compilation errors (because the C preprocessor regards duplicate definitions as potential sources of conflict). However, all VxWorks header files contain conditional compilation statements and definitions that ensure that their text is included only once, no matter how many times they are specified by include statements. Thus, an application module can include just those header files it needs directly, without regard to interdependencies or ordering, and no conflicts arise.

Internal Header Files

Internal header files are, for the most part, not intended for use by applications. The following subdirectories are exceptions, and are sometimes required by application programs:

- *installDir*/**target/h/net**, which is used by network drivers for specific network controllers.

- *installDir*/**target/h/rpc**, which is used by applications using the remote procedure call library.
- *installDir*/**target/h/sys**, which is used by applications using standard POSIX functions.

VxWorks Private Header Files

Some elements of VxWorks are internal details that may change and so should not be referenced in your application. The only supported uses of a module's facilities are through the public definitions in the header file, and through the module's subroutine interfaces. Your adherence ensures that your application code is not affected by internal changes in the implementation of a VxWorks module.

Some header files mark internal details using **HIDDEN** comments:

```
/* HIDDEN */
...
/* END HIDDEN */
```

Internal details are also hidden with *private* header files: files that are stored in the directory *installDir*/**target/h/private**. The naming conventions for these files parallel those in *installDir*/**target/h** with the library name followed by **P.h**. For example, the private header file for **semLib** is *installDir*/**target/h/private/semLibP.h**.

5.2.2 Compiling Application Modules Using GNU Tools

Tornado includes a full-featured C and C++ compiler and associated tools, collectively called the *GNU ToolKit*. Extensive documentation for this set of tools is available in the *GNU ToolKit User's Guide*. This section provides some general orientation about the source of these tools, and describes how the tools are integrated into the Tornado development environment.

NOTE: The GNU tools are not available for the ColdFire architecture; the Diab tools are the default toolset. Diab tools are available as an optional product for the ARM/StrongARM/XScale, MIPS, PowerPC, and Hitachi SH architectures. See *5.2.3 Compiling Application Modules Using Diab Tools*, p.179.

The GNU Tools

GNU ("GNU's Not UNIX!") is a project of the Free Software Foundation started by Richard Stallman and others to promote *free software*. To the FSF, free software is software whose source code can be copied, modified, and redistributed without restriction. GNU software is not in the public domain; it is protected by copyright and subject to the terms of the GNU General Public License, a legal document designed to ensure that the software remains free—for example, by prohibiting proprietary modifications and concomitant restrictions on its use. The General Public License can be found in the file **COPYING** that accompanies the source code for the GNU tools, and in the section titled *Free Software* at the back of the *GNU ToolKit User's Guide*.

It is important to be aware that the terms under which the GNU tools are distributed do not apply to the software you create with them. In fact, the General Public License makes no requirements of you as a software developer at all, as long as you do not modify or redistribute the tools themselves. On the other hand, it gives you the right to do both of these things, provided you comply with its terms and conditions. It also permits you to make unrestricted copies for your own use.

The Wind River GNU distribution consists of the GNU ToolKit, which contains GNU tools modified and configured for use with your VxWorks target architecture. The source code for these tools is available upon request.

Cross-Development Commands

The GNU cross-development tools in Tornado have names that clearly indicate the target architecture. This allows you to install and use tools for more than one architecture, and to avoid confusion with corresponding host native tools. A suffix identifying the target architecture is appended to each tool name. For example, the cross-compiler for the PowerPC processor family is called **ccppc**, and the assembler **asppc**. The suffixes used are shown in Table 5-1. Note that the *GNU ToolKit User's Guide* refers to these tools by their generic names (without a suffix).

Table 5-1 **Suffixes for Cross-Development Tools**

Architecture	Command Suffix
ARM, StrongARM, XScale	**arm**
MC680*x*0	**68k**
MIPS	**mips**

Table 5-1 **Suffixes for Cross-Development Tools** *(Continued)*

Architecture	Command Suffix
Pentium	**pentium**
PowerPC	**ppc**
SuperH	**sh**
VxSim Solaris, VxSim PC	**simso**, **simpc**

Defining the CPU Type

Tornado supports multiple target architectures. To accommodate this support, several VxWorks header files contain conditional compilation directives based on the definition of the variable **CPU**. When using these header files, the variable **CPU** must be defined in one of the following places:

– the source modules
– the header files
– the compilation command line

To define **CPU** in the source modules or header files, add the following line:

```
#define CPU cputype
```

To define **CPU** on the compilation command line, add the following flag:

```
-DCPU=cputype
```

The constants shown in Table 5-2 are supported values for *cputype*.

Table 5-2 **Values for *cputype* for GNU Tools**

Architecture	Value
ARM, StrongARM, XScale	**ARMARCH4**, **ARMARCH4_T**, **ARMARCH5**, **ARMARCH5_T**, **ARM7TDMI**, **ARM7TDMI_T**, **ARM710A**, **ARM810**, **ARMSA110**, **XSCALE**
MC680*x*0	**MC68000**, **MC68010**, **MC68020**,* **MC68040**, **MC68060**, **MC68LC040**†, **CPU32**
MIPS	**MIPS32**, **MIPS64**

Table 5-2 **Values for *cputype* for GNU Tools** *(Continued)*

Architecture	Value
Pentium	**PENTIUM2**, **PENTIUM3**, **PENTIUM4**
PowerPC	**PPC403**, **PPC405**, **PPC440**, **PPC603**, **PPC604**, **PPC860**
VxSim Solaris, VxSim PC	**SIMSPARCSOLARIS**, **SIMNT**
SuperH	**SH7600**, **SH7700**, **SH7750**

* **MC68020** is the appropriate value for both the MC68020 and the MC68030 CPUs.
† **MC68LC040** is the appropriate value for both the MC68LC040 and the MC68EC040.

With makefiles, the **CPU** definition can be added to the definition of the flags passed to the compiler (usually **CFLAGS**).

In the source code, the file **vxWorks.h** must be included before any other files with dependencies on the **CPU** flag.

As well as specifying the **CPU** value, you must usually run the compiler with one or more option flags to generate object code optimally for the particular architecture variant. These option flags usually begin with **-m**; see *Compiling C Modules With the GNU Compiler*, p.177.

Compiling C Modules With the GNU Compiler

The following is an example command to compile an application module for a VxWorks PowerPC 604 system:

```
% ccppc -mcpu=604 -mstrict-align -I ${WIND_BASE}/target/h -DCPU=PPC603 \
-DTOOL_FAMILY=gnu -DTOOL=gnu -c applic.c
```

This compiles the module **applic.c** into an object file **applic.o**.

Below are summary descriptions of the target-independent flags used in the example. Flags that are specific to a particular target architecture are described in the relevant architecture supplement. For more information on any of these flags, see the *GNU ToolKit User's Guide*.

-g
Generate debugging information.

-c
Compile only to produce a relocatable object file. The result is an object module with the suffix **.o**, in this case, **applic.o**.

-DCPU=*CPU*
Required; defines the CPU type.

-DTOOL_FAMILY=gnu
Optional; defines the compilation toolkit used to compile VxWorks. If not entered, it is derived from **-DTOOL=**.

-DTOOL=gnu
Required; specifies the compilation toolkit and the tool environment. For more information, see the *GNU ToolKit User's Guide*.

-I$(WIND_BASE)/target/h
Include VxWorks header files. (See *5.2.1 Using VxWorks Header Files*, p.171.)

-fno-builtin
Use library calls even for common library subroutines such as **memcpy**. Used by VxWorks for historical reasons. There is no need for application code to use this flag.

-Wall
Turn on all warnings. This flag is optional.

-ansi
Reject non-ANSI-compliant code. This flag is optional.

-O
Perform basic optimizations.

-O2
Perform most supported optimizations (except those involving a space-speed trade-off

Compiling C++ Modules

The GNU compiler drivers can be used to compile both C and C++ source files. C++ source files are recognized by their extension (typically **.cc**, **.cpp**, or **.C**). For complete information on using C++, including a detailed discussion of compiling C++ modules, see the *VxWorks Programmer's Guide*: *C++ Development*.

CAUTION: Different versions of C++ run-time support are provided for the GNU and Diab toolchains. For this reason, you cannot combine C++ objects compiled with GNU with C++ objects compiled with Diab. All C++ applications must be compiled with the same tool.

5.2.3 Compiling Application Modules Using Diab Tools

For more information about the Diab tools, see the *Diab C/C++ Compiler User's Guide*. The Diab tools are the only tools available for ColdFire. Diab tools are available as an optional product for the ARM/StrongARM/XScale, MIPS, PowerPC, and SuperH architectures.

The Diab Tools

The Diab C/C++ compiler suites are high performance programming tools. In addition to the benefits of state-of-the-art optimization, they reduce time spent creating reliable code because the compilers and other tools include many built-in, customizable, checking features which help detect problems earlier.

The compilers are particularly helpful in speeding up or reducing the size of existing programs developed with other tools.

With over 250 command-line options and special pragmas, and a powerful linker command language for arranging code and data in memory, the Diab C/C++ compiler suites can be customized to meet the needs of any embedded systems project. A number of options are specifically designed to be compatible with other tools to ease porting of existing code.

If you are using the Diab tools, you need to be sure that two settings are in place:

- Be sure that *installDir***/host/diab/WIN32\bin** is in your path.
- Be sure that the environment variable **DIABLIB** is set to *installDir***/host/diab**.

There is a batch file called **torVars.[c]sh** in *installDir***/host/***hostType***/bin** that will set **DIABLIB** for you.

Cross-Development Commands

The Diab cross-development tools in Tornado are always called by the same names: **dcc**, **dld**, and so forth. The architecture-specific version of the tool is specified by the **-t** option in the command line or makefile. For Tornado 2.2, the **-t** option always includes the architecture family and the VxWorks specifier, for example:

```
-tPPC403FS:vxworks55
```

When you install Diab in the Tornado tree, the defaults are set correctly for the architecture you installed. You can use the command **dcc -Xshow-target** to display the value and **dctrl -t** to change it.

You may need to change the architecture family and its characteristics (for example, **PPC403FS**). Detailed information is available in the *Diab C/C++ Compiler User's Guide: Selecting a Target and Its Components*.

Defining the CPU Type

Tornado supports multiple target architectures. To accommodate this, several VxWorks header files contain conditional compilation directives based on the definition of the variable **CPU**. When using these header files, the variable **CPU** must be defined in one of the following places:

– the source modules
– the header files
– the compilation command line

To define **CPU** in the source modules or header files, add the following line:

```
#define CPU cputype
```

To define **CPU** on the compilation command line, add the following flag:

```
-DCPU=cputype
```

The constants shown in Table 5-2 are supported values for *cputype*.

Table 5-3 **Values for *cputype* for Diab Tools**

Architecture	Value
ARM	**ARMARCH4**, **ARMARCH4_T**, **ARMARCH5_T**, **ARMARCH5_T**
ColdFire	**MCF5200**, **MCF5400**
SuperH	**SH7600**, **SH7700**, **SH7700**, **SH7750**
MIPS	**MIPS32, MIPS64**
PowerPC	**PPC403**, **PPC405**, **PPC440**, **PPC603**, **PPC604**, **PPC860**
StrongARM, XScale	**STRONGARM**, **XSCALE**

With makefiles, the **CPU** definition can be added to the definition of the flags passed to the compiler (usually **CFLAGS**).

In the source code, the file **vxWorks.h** must be included before any other files with dependencies on the **CPU** flag.

As well as specifying the **CPU** value, you must usually run the compiler with one or more option flags to generate object code optimally for the particular architecture variant. For detailed information, see *Diab C/C++ Compiler User's Guide: Selecting a Target and Its Components*.

Compiling C Modules With the Diab Compiler

The following is an example command to compile an application module for a VxWorks PowerPC 604 system:

```
% dcc -g -tPPC403FS:vxworks55 -Xmismatch-warning=2 \
-ew1554,1551,1552, 1086,1047,1547 -Xclib-optim-off -Xansi \
-Xstrings-in-text=0 -Wa,-Xsemi-is-newline-ei1516,1643,1604 \
-Xlocal-data-area-static-only -W:c++:,-Xexceptions -Xsize-opt \
-Wall -I${WIND_BASE}/target/h -DCPU=PPC604 -DTOOL=diab -c applic.c
```

This compiles the module **applic.c** into an object file **applic.o**.

Below are summary descriptions of the target-independent flags used in the example. Flags that are specific to a particular target architecture are described in the relevant architecture supplement. For more information on any of these flags see the *Diab C/C++ Compiler User's Guide*.

-g
Generate debugging information.

-tPPC403FS:vxworks55
Specifies the processor family and the compilation environment. See *Cross-Development Commands*, p.180.

-X*name_or_number***[=***value***]**
Control the compilation process when behavior other than the default is needed. Most **-X** options can be set either by name (**-X***name*) or by number (-**X***n*). Options control such behaviors as debugging, optimization, and syntax.

-ew*n***[,***n***,...]**
For each message number in the comma-separated list, change the severity level of the message to *warning*.

-Wa,*argument*
Pass *argument* to the assembler.

-ei*n***[,***n***,...]**
For each message number in the comma-separated list, change the severity level of the message to information (equivalent to *ignore*).

-W:c++:,-Xexception
Pass the argument **-Xexception** to the C++ compiler.

-D *arch family*
Specifies the architecture family.

-DCPU=*CPU*
Required; defines the CPU type.

-DTOOL_FAMILY=diab
Optional; defines the compilation toolkit used to compile VxWorks. If not entered, it is derived from **-DTOOL=**.

-DTOOL=diab
Required; specifies the tool and tool environment. For more information, see the *Diab C/C++ Compiler User's Guide*.

-I$(WIND_BASE)/target/h
Include VxWorks header files. (See *5.2.1 Using VxWorks Header Files*, p.171)

Compiling C++ Modules

The Diab compiler uses **dcc** to invoke the C compiler and **dplus** to invoke the C++ compiler. For complete information on using C++, including a detailed discussion of compiling C++ modules see the *VxWorks Programmer's Guide*: *C++ Development* and the *Diab C/C++ Compiler User's Guide*.

CAUTION: Different versions of C++ run-time support are provided for the GNU and Diab toolchains. For this reason, you cannot combine C++ objects compiled with GNU with C++ objects compiled with Diab. All C++ applications must be compiled with the same tool.

5.2.4 Static Linking (Optional)

After you compile an application module, you can load it directly into the target with the Tornado dynamic loader (through the shell or through the debugger).

In general, application modules do not need to be prelinked before being downloaded to the target. The exception is when several application modules cross reference each other. In this case, the modules should be linked to form a single downloadable module. When using C++, this prelinking should be done before the *munch* step (see the *VxWorks Programmer's Guide*: *C++ Development*).

The following example is a command to link several application modules, using the GNU linker for the PowerPC family of processors.

```
% ldppc -o applic.o -r applic1.o applic2.o applic3.o
```

Similarly, the following example is a command to link several application modules, using the Diab linker for the PowerPC family of processors.

```
% dld -o applic.o -r applic1.o applic2.o applic3.o
```

This creates the object module **applic.o** from the object modules **applic1.o**, **applic2.o**, and **applic3.o**. The **-r** option is required, because the object-module output must be left in relocatable form so that it can be downloaded and linked to the target VxWorks image.

Any VxWorks facilities called by the application modules are reported by the linker as unresolved externals. These are resolved by the Tornado loader when the module is loaded into VxWorks memory.

WARNING: Do not link each application module with the VxWorks libraries. Doing this defeats the load-time linking feature of Tornado, and wastes space by writing multiple copies of VxWorks system modules on the target.

5.2.5 Downloading an Application Module

After application object modules are compiled (and possibly linked by the host **ld***arch* command), they can be dynamically loaded into a running VxWorks system by invoking the Tornado module loader. You can do this either from the Tornado shell using the built-in command **ld( )**, or from the debugger using the Debug menu or the **load** command.

The following is a typical load command from the Tornado shell:

```
-> ld <applic.o
```

This relocates the code from the host file **applic.o**, linking to previously loaded modules, and loads the object module into the target's memory. Once an application module is loaded into target memory, any subroutine in the module can be invoked directly from the shell, spawned as a task, connected to an interrupt, and so on.

NOTE: The order in which modules are loaded using **ld( )** is important. A downloaded module can call into a previously downloaded module to resolve symbols. However, the opposite is not true. For example, given two modules, **applic1.o** and **applic2.o**, in which **applic1.o** can stand alone, but **applic2.o** relies on symbols that are defined in **applic1.o**; **ld( )** will perform the necessary linking only if **applic1.o** is loaded before **applic2.o**.

The shell **ld( )** command, by default, adds only global symbols to the symbol table. During debugging, you may want local symbols as well. To get all symbols loaded (including local symbols), you can use the GDB command **load** from the debugger. Because this command is meant for debugging, it always loads all symbols.

Alternatively, you can load all symbols by calling the shell command **ld( )** with a full argument list instead of the shell-redirection syntax shown above. When you use an argument list, you can get all symbols loaded by specifying a 1 as the first argument, as in the following example:

```
-> ld 1,0,"applic.o"
```

In the previous examples, the object module **applic.o** resides in the shell's current working directory. Normally, you can use either relative path names or absolute path names to identify object modules to **ld( )**. If you use a relative path name, the shell converts it to an absolute path (using its current working directory) before passing the download request to the target server. In order to avoid trouble when you call **ld( )** from a shell that is not running on the same host as its target server, Tornado supplies the **LD_SEND_MODULES** facility; see *7. Shell*. If you are using a remote target server and **ld( )** fails with a "no such file" message, be sure that **LD_SEND_MODULES** is set to "on."

For more information about loader arguments, see the discussion of **ld( )** in the reference entry for **windsh**.

For information about the target-resident version of the loader (which also requires the target-resident symbol table), see the *VxWorks Programmer's Guide: Target Tools* and the VxWorks reference entry for **loadLib**. For information on booting VxWorks, see *2.6 Booting VxWorks*, p.46.

5.2.6 Module IDs and Group Numbers

When a module is loaded, it is assigned a module ID and a group number. Both the module ID and the group number are used to reference the module. The module ID is returned by **ld( )** as well as by the target-resident loader routines. When symbols are added to the symbol table, the associated module is identified by the group number (a small integer). (Due to limitations on the size of the symbol table, the module ID is inappropriate for this purpose.) All symbols with the same group number are from the same module. When a module is unloaded, the group number is used to identify and remove all the module's symbols from the symbol table.

5.2.7 Unloading Modules

Whenever you load a particular object module more than once, using the target server (from either the shell or the debugger), the older version is unloaded automatically. You can also unload a module explicitly; both the Tornado shell and the target-resident VxWorks libraries include an unloader. To remove a module from the shell, use the shell routine **unld( )**; see the reference entry for **windsh**.

For information about the target-resident version of the unloader (which also requires the target-resident symbol table and loader), see the *VxWorks Programmer's Guide: Target Tools* and the VxWorks reference entry for **unldLib**.

After a module has been unloaded, any calls to routines in that module fail with unpredictable results. Take care to avoid unloading any modules that are required by other modules. One solution is to link interdependent files using the static linker **ld***arch* as described in *5.2.4 Static Linking (Optional)*, p.183, so that they can only be loaded and unloaded as a unit.

5.3 Configuring VxWorks

The configuration of VxWorks is determined by the configuration header files *installDir***/target/config/all/configAll.h** and *installDir***/target/config/***bspname***/config.h**. These files are used by the **usrConfig.c**, **bootConfig.c**, and **bootInit.c** modules as they run the initialization routines distributed in the directory *installDir***/target/src/config** to configure VxWorks.

The VxWorks distribution includes the configuration files for the default development configuration. You can create your own versions of these files to better suit your particular configurations; this process is described in the following subsections. In addition, if you need multiple configurations, environment variables can be set so you can move easily between them.

Including optional components in your VxWorks image can significantly increase the image size. If you receive a warning from **vxsize** when building VxWorks, or if the size of your image becomes greater than that supported by the current setting of **RAM_HIGH_ADRS**, be sure to see *Scaling Down VxWorks*, p.193 and *5.6 Creating Bootable Applications, p.210* for information on how to resolve the problem.

WARNING: Use of the project facility for configuring and building applications is largely independent of the methods used prior to Tornado 2.0 (which included manually editing the configuration files **config.h** or **configAll.h**). The project facility provides the recommended and simpler means for configuration and building; the manual method is described in this section.

To avoid confusion and errors, the two methods should not be used together for the same project. One exception is for any configuration macro that is not accessible through the project facility GUI (which may be the case, for example, for some BSP driver parameters). You can use a Find Object dialog box to determine if a macro is accessible or not (see *Finding VxWorks Components and Configuration Macros*, p.136). If it is not accessible through the GUI, a configuration file must be edited, and the project facility will implement the change in the subsequent build.

The order of precedence for determining configuration is (in descending order):

project facility
config.h
configAll.h

For any macro that is exposed through the project facility GUI, changes made after creation of a project in either of the configuration files will not appear in the project.

Another exception is that you may configure a BSP using manual methods and then use provided make targets to create a project for application development. See *5.7 Building Projects From a BSP*, p.214.

5.3.1 The Board Support Package (BSP)

The directory *installDir*/**target/config/***bspname* contains the *Board Support Package (BSP)*, which consists of files for the particular hardware used to run VxWorks, such as a VME board with serial lines, timers, and other devices. The files include: **Makefile**, **sysLib.c**, **sysALib.s**, **romInit.s**, *bspname*.**h**, and **config.h**.

Wind River-supplied BSPs conform to a standard, introduced with BSP Version 1.1. The standard is fully described in the *VxWorks BSP Developer's Guide*.

The System Library

The file **sysLib.c** provides the board-level interface on which VxWorks and application code can be built in a hardware-independent manner. The functions addressed in this file include:

- Initialization functions
  - initialize the hardware to a known state
  - identify the system
  - initialize drivers, such as SCSI or custom drivers
- Memory/address space functions
  - get the on-board memory size
  - make on-board memory accessible to the external bus (optional)
  - map local and bus address spaces
  - enable/disable cache memory
  - set/get nonvolatile RAM (NVRAM)
  - define the board's memory map (optional)
  - virtual-to-physical memory map declarations for processors with MMUs
- Bus interrupt functions
  - enable/disable bus interrupt levels
  - generate bus interrupts
- Clock/timer functions
  - enable/disable timer interrupts
  - set the periodic rate of the timer
- Mailbox/location monitor functions (optional)
  - enable mailbox/location monitor interrupts

The **sysLib** library does not support every feature of every board. Some boards may have additional features, others may have fewer, others still may have the same features with a different interface. For example, some boards provide some **sysLib** functions by means of hardware switches, jumpers, or PALs, instead of by software-controllable registers.

The configuration modules **usrConfig.c** and **bootConfig.c** in *installDir*/**target/config/all** are responsible for invoking this library's routines at the appropriate time. Device drivers can use some of the memory mapping routines and bus functions.

Virtual Memory Mapping

For boards with MMU support, the data structure **sysPhysMemDesc** defines the virtual-to-physical memory map. This table is typically defined in **sysLib.c**, although some BSPs place it in a separate file, **memDesc.c**. It is declared as an array of the data structure **PHYS_MEM_DESC**. No two entries in this descriptor can overlap; each entry must be a unique memory space.

The **sysPhysMemDesc** array should reflect your system configuration, and you may encounter a number of reasons for changing the MMU memory map, for example: the need to change the size of local memory or the size of the VME master access space, or because the address of the VME master access space has been moved. For information on virtual memory mapping, as well as an example of how to modify **sysPhysMemDesc**, see the *VxWorks Programmer's Guide: Virtual Memory Interface*.

CAUTION: A bus error can occur if you try to access memory that is not mapped.

Configuration Files

The file **config.h** specifies which VxWorks facilities are included in your system image. The file *bspname***.h** specifies BSP-specific capabilities.

BSP Initialization Modules

The following files initialize the BSP:

- The file **romInit.s** contains assembly-level initialization routines.
- The file **sysALib.s** contains initialization and system-specific assembly-level routines.

BSP Documentation

The file **target.nr** in the *installDir***/target/config/***bspname* directory is the source of the online reference entry for target-specific information. (You can also view the HTML version of this document from the Tornado IDE: Help>Manuals Contents>BSP Reference>*bspname*.) The **target.nr** file describes the supported board variations, the relevant jumpering, and supported devices. It also includes an

ASCII representation of the board layout with an indication of board jumpers (if applicable) and the location of the ROM sockets.

5.3.2 The Environment Variables

You can use Tornado environment variables to build variations of system configurations. In general, your Tornado environment consists of three parts: the host code (Tornado), the target code, and the configuration files discussed in this section. If you use the default environment, your UNIX environment variables are defined as follows:

Host code:	**$WIND_BASE/host/***hosttype***/bin**
Target code:	**TGT_DIR** = **$WIND_BASE/target**
Configuration code:	**CONFIG_ALL** = $**TGT_DIR/config/all**

To use different versions of **usrConfig.c**, **bootConfig.c**, and **bootInit.c**, store them in a different directory and change the value of **CONFIG_ALL**. To use different target code, point to the alternate directory by changing the value of **TGT_DIR**.

You can change the value of **CONFIG_ALL** by changing it either in your makefile or on the command line. The value of **TGT_DIR** must be changed on the command line.

NOTE: Changing **TGT_DIR** will change the default value of **CONFIG_ALL**. If this is not what you want, reset **CONFIG_ALL** as well.

To change **CONFIG_ALL** in your makefile, add the following command:

```
CONFIG_ALL = $WIND_BASE/target/config/newDir
```

To change **CONFIG_ALL** on the command line, do the following:

```
% make ... CONFIG_ALL = $WIND_BASE/target/config/newDir
```

To change **TGT_DIR** on the command line, do the following:

```
% make ... TGT_DIR = $ALT_DIR/target
```

5.3.3 The Configuration Header Files

You can control VxWorks's configuration by including or excluding definitions in the global configuration header file **configAll.h** and in the target-specific configuration header file **config.h**. This section describes these files.

The Global Configuration Header File: configAll.h

The **configAll.h** header file, in the directory *installDir*/**target/config/all**, contains default definitions that apply to all targets, unless they are redefined in the target-specific header file **config.h**. The following options and parameters are defined in **configAll.h**:

- kernel configuration parameters
- I/O system parameters
- NFS parameters
- selection of optional software modules
- selection of optional device controllers
- cache modes
- maximum number of different shared memory objects
- device controller I/O addresses, interrupt vectors, and interrupt levels
- miscellaneous addresses and constants

The BSP-specific Configuration Header File: config.h

The BSP-specific header file, **config.h**, is located in the directory *installDir*/**target/config/***bspname*. This file contains definitions that apply only to the specific target, and can also redefine default definitions in **configAll.h** that are inappropriate for the particular target. For example, if a target cannot access a device controller at the default I/O address defined in **configAll.h** because of addressing limitations, the address can be redefined in **config.h**.

The **config.h** header file includes definitions for the following parameters:

- default boot parameter string for boot ROMs
- interrupt vectors for system clock and parity errors
- device controller I/O addresses, interrupt vectors, and interrupt levels
- shared memory network parameters
- miscellaneous memory addresses and constants

⚠ **CAUTION:** If any options from **configAll.h** need to be changed for this one BSP, then any previous definition of that option should be undefined and redefined as necessary in **config.h**. Unless options are to apply to all BSPs at your site, do not change them in *installDir*/**target/config/all/configAll.h**.

Selection of Optional Features

VxWorks ships with optional features and device drivers that can be included in, or omitted from, the target system. These are controlled by macros in the project facility or the configuration header files that cause conditional compilation in the *installDir*/**target/config/all/usrConfig.c** module.

The distributed versions of the configuration header files **configAll.h** and **config.h** include all the available software options and several network device drivers. If you are not using the project facility (see *4. Projects*), you define a macro by moving it from the **EXCLUDED FACILITIES** section of the header file to the **INCLUDED SOFTWARE FACILITIES** section.[1] For example, to include the ANSI C **assert** library, make sure the macro **INCLUDE_ANSI_ASSERT** is defined; to include the Network File System (NFS) facility, make sure **INCLUDE_NFS** is defined. Modification or exclusion of particular facilities is discussed in detail in *5.3.5 Alternative VxWorks Configurations*, p.193.

5.3.4 The Configuration Module: usrConfig.c

Use of the VxWorks configuration header files to configure your VxWorks system should meet all of your development requirements. Users should not resort to changing the Wind River-supplied **usrConfig.c**, or any other module in the directory *installDir*/**target/config/all**. If, however, an extreme situation requires such a change, we recommend you copy all the files in *installDir*/**target/config/all** to another directory, and add a **CONFIG_ALL** macro to your makefile to point the make system to the location of the modified files. For example, add the following to your makefile after the first group of include statements:

```
# ../myAll contains a copy of all the ../all files
    CONFIG_ALL = ../myAll
```

1. To see the available macros with their descriptions, see *installDir*/**target/config/all/configAll.h** (for macros applicable to all BSPs) and *installDir*/**target/config/***bspname*/**config.h** (for macros applicable to a specific BSP).

5.3.5 Alternative VxWorks Configurations

The discussion of the **usrConfig** module in *5.3.4 The Configuration Module: usrConfig.c*, p.192 outlined the default configuration for a development environment. In this configuration, the VxWorks system image contains all of the VxWorks modules that are necessary to allow you to interact with the system through the Tornado host tools.

However, as you approach a final production version of your application, you may want to change the VxWorks configuration in one or more of the following ways:

- Change the configuration of the target agent.
- Decrease the size of VxWorks.
- Run VxWorks from ROM.

The following sections discuss the latter two alternatives to the typical development configuration. For a discussion on reconfiguring the target agent, see *4. Projects*.

Scaling Down VxWorks

In a production configuration, it is often desirable to remove some of the VxWorks facilities to reduce the memory requirements of the system, to reduce boot time, or for security purposes.

Optional VxWorks facilities can be omitted by commenting out or using **#undef** to undefine their corresponding control constants in the header files **configAll.h** or **config.h**. For example, logging facilities can be omitted by undefining **INCLUDE_LOGGING**, and signalling facilities can be omitted by undefining **INCLUDE_SIGNALS**.

VxWorks is designed to make it easy to exclude facilities you do not need. However, not every BSP is organized in this way. If you wish to minimize the size of your system, be sure to examine your BSP code and eliminate references to facilities you do not need. Even though you exclude them, if your code refers to them, your exclusion will be overridden.

Excluding Kernel Facilities

The definition of the following constants in **configAll.h** is optional, because referencing any of the corresponding kernel facilities from the application automatically includes the kernel service:

– **INCLUDE_SEM_BINARY**

- **INCLUDE_SEM_MUTEX**
- **INCLUDE_SEM_COUNTING**
- **INCLUDE_MSG_Q**
- **INCLUDE_WATCHDOGS**

These configuration constants appear in the default VxWorks configuration to ensure that all kernel facilities are configured into the system, even if not referenced by the application. However, if your goal is to achieve the smallest possible system, exclude these constants; this ensures that the kernel does not include facilities you are not actually using.

There are two other configuration constants that control optional kernel facilities: **INCLUDE_TASK_HOOKS** and **INCLUDE_CONSTANT_RDY_Q**. Define these constants in **configAll.h** if the application requires either kernel callouts (use of task hook routines) or a constant-insertion-time, priority-based ready queue. A ready queue with constant insert time allows the kernel to operate context switches with a fixed overhead regardless of the number of tasks in the system. Otherwise, the worst-case performance degrades linearly with the number of ready tasks in the system. Note that the constant-insert-time ready queue uses 2 KB for the data structure; some systems do not have sufficient memory for this. In those cases, the definition of **INCLUDE_CONSTANT_RDY_Q** may be omitted, thus enabling use of a smaller (but less deterministic) ready queue mechanism.

Excluding Network Facilities

In some applications it may be appropriate to eliminate the VxWorks network facilities. For example, in the ROM-based systems or standalone configurations described in *5.6 Creating Bootable Applications, p.210*, there may be no need for network facilities.

To exclude the network facilities, be sure the following constants are not defined:

- **INCLUDE_NETWORK**
- **INCLUDE_NET_INIT**
- **INCLUDE_NET_SYM_TBL**
- **INCLUDE_NFS**
- **INCLUDE_RPC**

Option Dependencies

Option dependencies are coded in the file *installDir*/**target/src/config/usrDepend.c**, so that when a particular option is chosen, everything required is included. This assures you of a working system with minimum effort. Although you can exclude the features that you do not need

by undefining them in **config.h** and **configAll.h**, you should be aware that in some cases they may not be excluded because of dependencies.

For example, you cannot use **telnet** without running the network. Therefore, if in your **configAll.h** file, the option **INCLUDE_TELNET** is selected but the option **INCLUDE_NET_INIT** is not, **usrDepend.c** defines **INCLUDE_NET_INIT** for you. Because the network initialization requires the network software, the **userDepend.c** file also defines **INCLUDE_NETWORK**.

Because most of the dependencies are taken care of in **usrDepend.c**, that file is currently included in **usrConfig.c**. This simplifies the build process and the selection of options. However, you can change or add dependencies if you choose.

Executing VxWorks from ROM

You can put VxWorks or a VxWorks-based application into ROM; this is discussed in *5.6.3 Creating a VxWorks System in ROM*, p.212. For an example of a ROM-based VxWorks application, see the VxWorks boot ROM program. The file *installDir*/**target/config/all/bootConfig.c** is the configuration module for the boot ROM, replacing the file **usrConfig.c** provided for the default VxWorks development system.

In such ROM configurations, the *text* and *data* segments of the boot or VxWorks image are first copied into the system RAM, then the boot procedure or VxWorks executes in RAM. On some systems where memory is a scarce resource, it is possible to save space by copying only the data segment to RAM. The text segment remains in ROM and executes from that address space, and thus is termed *ROM resident*. The memory that was to be occupied by the text segment in RAM is now available for an application (up to 300 KB for a standalone VxWorks system). Note that ROM-resident VxWorks is not supported on all boards; see the reference entry for your target if you are not sure that your board supports this configuration.

The drawback of a ROM-resident text segment is the limited data widths and lower memory access time of the EPROM, which causes ROM-resident text to execute more slowly than if it was in RAM. This can sometimes be alleviated by using faster EPROM devices or by reconfiguring the standalone system to exclude unnecessary system features.

Aside from program text not being copied to RAM, the ROM-resident versions of the VxWorks boot ROMs and the standalone VxWorks system are identical to the conventional versions. A ROM-resident image is built with an uncompressed

version of either the boot ROM or standalone VxWorks system image. VxWorks target makefiles include entries for building these images; see Table 5-4.

Table 5-4 **Makefile ROM-Resident Images**

Architecture	Image FIle*	Description
MIPS and PowerPC	**bootrom_res_high**	ROM-resident boot ROM image. The data segment is copied from ROM to RAM at address **RAM_HIGH_ADRS**.
	vxWorks.res_rom_res_low	ROM-resident standalone system image without compression. The data segment is copied from ROM to RAM at address **RAM_LOW_ADRS**.
	vxWorks.res_rom_nosym_res_low	ROM-resident standalone system image without compression or symbol table. Data segment is copied from ROM to RAM at address **RAM_LOW_ADRS**.
All Other Targets	**bootrom_res**	ROM-resident boot ROM image.
	vxWorks.res_rom	ROM-resident standalone system image without compression.
	vxWorks.res_rom_nosym	ROM-resident system image without compression or symbol table. Ideal for the Tornado environment.

* All images have a corresponding file in Motorola S-record or Intel Hex format with the same file name plus the extension **.hex** and one in binary format with the extension **.bin**.

Because of the size of the system image, 512 KB of EPROM is recommended for the ROM-resident version of the standalone VxWorks system. More space is probably required if applications are linked with the standalone VxWorks system. For a ROM-resident version of the boot ROM, 256 KB of EPROM is recommended. If you use ROMs of a size other than the default, modify the value of **ROM_SIZE** in the target makefile and **config.h**.

A separate make target, **vxWorks.res_rom_nosym**, has been created to provide a ROM-resident image without the symbol table. This is intended to be a standard ROM image for use with the Tornado environment where the symbol table resides on the host system. Being ROM-resident, the debug agent and VxWorks are ready almost immediately after power-up or restart.

The data segment of a ROM-resident standalone VxWorks system is loaded at **RAM_LOW_ADRS** (defined in the makefile) to minimize fragmentation. The data segment of ROM-resident boot ROMs is loaded at **RAM_HIGH_ADRS**, so that loading VxWorks does not overwrite the resident boot ROMs. For a CPU board with limited memory (under 1 MB of RAM), make sure that **RAM_HIGH_ADRS** is less than **LOCAL_MEM_SIZE** by a margin sufficient to accommodate the data segment. Note that **RAM_HIGH_ADRS** is defined in both the makefile and **config.h**. These definitions *must* agree.

Figure 5-1 shows the memory layout for ROM-resident boot and VxWorks images. The lower portion of the diagram shows the layout for ROM; the upper portion shows the layout for RAM. **LOCAL_MEM_LOCAL_ADRS** is the starting address of RAM. For the boot image, the data segment gets copied into RAM above **RAM_HIGH_ADRS** (after space for *bss* is reserved). For the VxWorks image, the data segment gets copied into RAM above **RAM_LOW_ADRS** (after space for *bss* is reserved). Note that for both images the text segment remains in ROM.

5.4 Building a VxWorks System Image

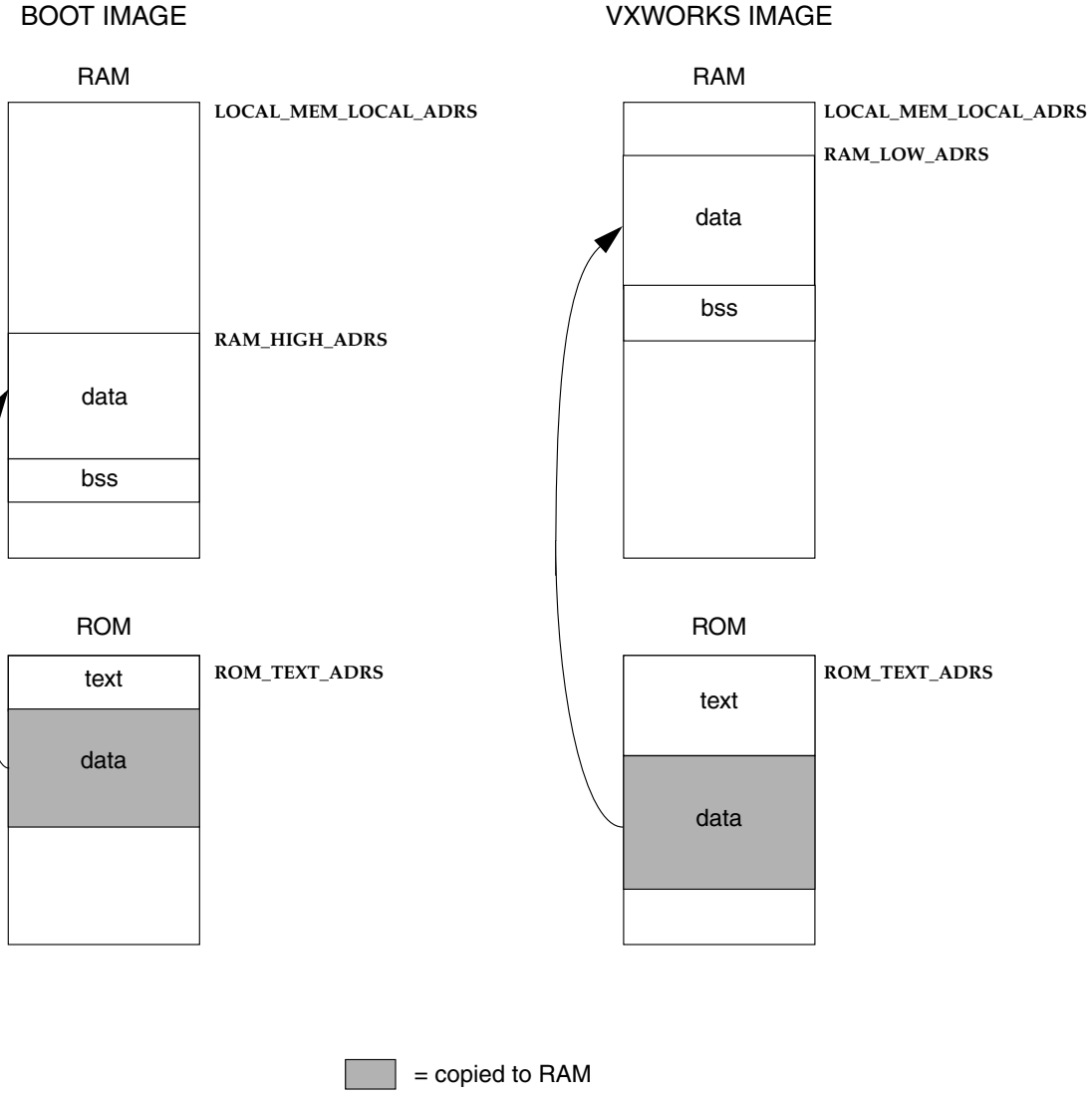
You can redefine the VxWorks configuration in two ways: interactively, as described in *4. Projects*, or by editing VxWorks configuration files as described in *5.3 Configuring VxWorks*, p.186. In either case, after you alter the configuration, VxWorks must be rebuilt to incorporate the changes. This includes recompiling certain modules and re-linking the system image. This section explains the procedures for rebuilding the VxWorks system image using manual techniques.

5.4.1 Available VxWorks Images

There are three types of VxWorks images:

- Boot application images
- Downloadable VxWorks images
- ROM-based VxWorks images

Figure 5-1 **ROM-Resident Memory Layout**



Boot ROM images come in three flavors: compressed, uncompressed, and ROM-resident.

bootrom	normal compressed boot ROM
bootrom_uncmp	uncompressed boot ROM
bootrom_res	ROM-resident boot ROM

Downloaded VxWorks images come in two basic varieties, Tornado and standalone. (Here "Tornado" is a Vxworks image that uses the host-based tools and symbol table, while "standalone" is an image that uses the target tools and symbol table.)

vxWorks	basic Tornado	uses host shell and symbol table
vxWorks.st	standalone image	has target shell and symbol table

ROMmed VxWorks images include:

vxWorks_rom	Tornado in ROM (uncompressed)
vxWorks.st_rom	**vxWorks.st** in ROM (compressed)
vxWorks.res_rom	**vxWorks.st** ROM-resident
vxWorks.res_rom_nosym	Tornado, ROM-resident

5.4.2 Rebuilding VxWorks with make

VxWorks uses the GNU **make** facility to recompile and relink modules. A file called **Makefile** in each VxWorks target directory contains the directives for rebuilding VxWorks for that target. See the *GNU Make User's Guide* for a detailed description of GNU **make** and of how to write makefiles.

With a UNIX host, you must use the GNU version of **make** included with Tornado; makefiles distributed by Wind River may, and often do, make use of features supported only by GNU **make**. Ensure that your **PATH** variable has *installDir***/host/***hostType***/bin** ahead of the directory that contains your native OS version of **make**.

To rebuild VxWorks, first change to the VxWorks target directory for the desired target, and invoke **make** as follows:

```
% cd ${WIND_BASE}/target/config/bspname
% make
```

make compiles and links modules as necessary, based on the directives in the target directory's makefile.

To rebuild VxWorks when only header files change, use one of the following methods:

```
% make clean VxWorks
```

Or:

```
% make clean
% make
```

Either method removes all existing **.o** files, and then recreates the new **.o** files required by VxWorks.

5.4.3 Including Customized VxWorks Code

The directory *installDir*/**target/src/usr** contains the source code for certain portions of VxWorks that you may wish to customize. For example, **usrLib.c** is a popular place to add target-resident routines that provide application-specific development aids. For a summary of other files in this directory, see *A. Directories and Files*.

If you modify one of these files, an extra step is necessary before rebuilding your VxWorks image: you must replace the modified object code in the appropriate VxWorks archive. The makefile in *installDir*/**target/src/usr** automates the details; however, because this directory is not specific to a single architecture, you must specify the value of the **CPU** variable on the **make** command line:

```
% make CPU=cputype TOOL=tool
```

This step recompiles all modified files in the directory, and replaces the corresponding object code in the appropriate architecture-dependent directory. After that, the next time you rebuild VxWorks, the resulting system image includes your modified code.

The following example illustrates replacing **usrLib** with a modified version, rebuilding the archives, and then rebuilding the VxWorks system image. For the sake of conciseness, the **make** output is not shown. The example assumes the **epc4** (**I80386**) BSP; replace the BSP directory name and **CPU** value as appropriate for your environment.

```
% cd ${WIND_BASE}/target/src/usr
% cp usrLib.c usrLib.c.orig
% cp develDir/usrLib.c usrLib.c
% make CPU=PPC860
...
```

```
% cd ${WIND_BASE}/target/config/epc4
% make
...
```

5.4.4 Linking the System Modules

The commands used to link a VxWorks system image are somewhat complicated. Fortunately, it is not necessary to understand those commands in detail because they are included in the makefile in each VxWorks target directory. However, for completeness, this section gives an explanation of the flags and parameters used to link VxWorks modules.

VxWorks operating system modules are distributed in the form of archive libraries. One set of archives is provided for each target architecture. These archives are located under *installDir*/**target/lib**. For more details about the archive directory structure, see the *Tornado Migration Guide: Binary Compatibility*.

These modules are combined with the configuration module **usrConfig.o** by the **cc***arch* command on the host. (The file **usrConfig.c** is described in *5.3.4 The Configuration Module: usrConfig.c*, p.192.) The following are example commands for building and linking a VxWorks system using the GNU compiler for PowerPC:

For a partial image (partially linked):

```
ccppc -r -nostdlib -Wl,-X -o vxWorks.tmp sysALib.o sysLib.o \
miiLib.obj  usrConfig.o version.o -Wl,--start-group \
-L/vobs/wpwr/target/lib/ppc/PPC604/gnu \
-L/vobs/wpwr/target/lib/ppc/PPC604/common -lcplus -lgnucplus \
-lvxcom -lvxdcom -larch -lcommoncc -ldcc -ldrv -lgcc -lnet -los \
-lrpc -lsecurity -ltffs -lusb -lvxfusion -lvxmp -lvxvmi -lwdb  \
-lwind -lwindview /vobs/wpwr/target/lib/libPPC604gnuvx.a -Wl,--end-group
```

For the final image (fully linked):

```
ldppc -X -N -e _sysInit -Ttext 00100000 -o vxWorks dataSegPad.o \
vxWorks.tmp ctdt.o -T /vobs/wpwr/target/h/tool/gnu/ldscripts/link.RAM
```

The meanings of the flags in this command are as follows:

-r Generate relocatable output.

-nostdlib
Do not use the standard system libraries.

-wl,*option*
Pass *option* as an option to the linker.

-X Eliminate some compiler-generated symbols from the symbol table.

-o vxWorks
Name the output object module **vxWorks**.

--start-group *archives* **--end-group**
The specified *archives* are searched repeatedly until no new undefined references are created. *archives* should be a list of archive files. They may be either explicit file names, or **-l** options.

-l*arch*
List of all the archive files added to the list of files to link. **ld** searches its path-list for occurrences of **lib***arch***.a** for every archive specified.

-L*searchdir*
List of all the paths that **ld** will search for archive libraries. The directories are searched in the order in which they are specified on the command line. All **-L** options apply to all **-l** options, regardless of the order in which the options appear.

-N Do not configure the output object module for a virtual-memory system.

-Ttext 1000
Specify the relocation address as a hexadecimal constant; in this example, 1000 hexadecimal. This is the address where the system must be loaded in the target, and is also the address where execution starts. Some target systems have limitations on where this relocation address can be.

-e _sysInit
Define the entry point to **vxWorks**. **sysInit( )** is the first routine in **sysALib.o**, which is the first module loaded by **ld***arch*.

sysALib.o and **sysLib.o**
Modules that contain CPU-dependent initialization and support routines. The module **sysALib.o** must be the first module specified in the **ld***arch* command.

usrConfig.o
The configuration module (described in detail in *5.3.4 The Configuration Module: usrConfig.c*, p.192). If you have several different system configurations, you may maintain several different configuration modules, either in *installDir***/target** or in your own directory.

version.o
A module that defines the creation date and version number of this **vxWorks** object module. It is created by compiling the output of **makeVersion**, an auxiliary tool in the *installDir***/host/***host***-***os***/bin** directory.

installDir/**target/lib/lib***cpu***gnuvx.a**
A VxWorks 5.4.x archive, included for backward compatibility. VxWorks is completely specified by the libraries indicated by **-l** and **-L**. This library might be used by optional or third-party products.

Additional object modules
You can link additional object modules (with **.o** suffix) into the run-time VxWorks system by naming them on the **ld***arch* command line. An easy way to do this is to use the variable **MACH_EXTRA** in the BSP makefiles. Define this variable and list the object modules to be linked with VxWorks. Note that during development, application object modules are generally not linked with the system (unless they are needed by the **usrConfig** module), because it is more convenient to load them incrementally from the host after booting VxWorks. See *5.6 Creating Bootable Applications, p.210* for more detail on linking application modules in a bootable system.

5.4.5 Creating the System Symbol Table Module

The Tornado target server uses the VxWorks symbol table on the host system, both for dynamic linking and for symbolic debugging. The symbol table file is created by the supplied tool **xsym**. Processing an object module with **xsym** creates a new object module that contains all the symbols of the original file, but with no code or data. The line in the makefile that creates this file executes the command:

```
xsym < vxWorks > vxWorks.sym
```

The file **vxWorks.sym** is downloaded to the target to build the target symbol table when **INCLUDE_NET_SYM_TBL** is included.

5.5 Makefiles for BSPs and Applications

Makefiles for VxWorks applications are easy to create by exploiting the makefiles and **make** include files shipped with VxWorks BSPs. This section discusses how the VxWorks BSP makefiles are structured. For more information, see . An example of how to utilize this structure for application makefiles is in *5.5.2 Using Makefile Include Files for Application Modules*, p.209.

A set of supporting files in *installDir*/**target/h/make** makes it possible for each BSP or application makefile to be terse, specifying only the essential parameters that are unique to the object being built.

Example 5-1 shows the makefile from the *installDir*/**target/config/mbx860** directory; the makefile for any other BSP is similar. Two variables are defined at the start of the makefile: **CPU**, to specify the target architecture; and **TOOL** to identify what compilation tools to use. Based on the values of these variables and on the environment variables defined as part of your Tornado configuration, the makefile selects the appropriate set of definitions from *installDir*/**target/h/make**. After the standard definitions, several variables define properties specific to this BSP. Finally, the standard rules for building a BSP on your host are included.

Example 5-1 **Makefile for mbx860**

```
# Makefile - makefile for target/config/mbx860
#
# Copyright 1984-2001 Wind River Systems, Inc.
# Copyright 1997,1998 Motorola, Inc., All Rights Reserved
#
# DESCRIPTION
# This file contains rules for building VxWorks for the
# MBX Board with a PowerPC 860 or PowerPC 821 processor.
#
# INCLUDES
#     makeTarget
#*/

CPU              = PPC860
TOOL             = gnu

TGT_DIR = $(WIND_BASE)/target

include $(TGT_DIR)/h/make/defs.bsp
#include $(TGT_DIR)/h/make/make.$(CPU)$(TOOL)
#include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)

## Only redefine make definitions below this point, or your definitions will
## be overwritten by the makefile stubs above.

TARGET_DIR       = mbx860
VENDOR           = Motorola
BOARD            = MBX860

## The constants ROM_TEXT_ADRS, ROM_SIZE, and RAM_HIGH_ADRS are defined
## in config.h and Makefile. All definitions for these constants must be
## identical.

ROM_TEXT_ADRS    = FE000100 # ROM entry address
ROM_SIZE         = 00080000 # number of bytes of ROM space

RAM_LOW_ADRS     = 00010000 # RAM text/data address
```

```
RAM_HIGH_ADRS    = 00200000 # RAM text/data address

USR_ENTRY        = usrInit

BOOT_EXTRA       = mbxI2c.o mbxALib.o

MACH_EXTRA       = mbxALib.o

RELEASE          += bootrom.bin

## Only redefine make definitions above this point, or the expansion of
## makefile target dependencies may be incorrect.

include $(TGT_DIR)/h/make/rules.bsp
```

The following **make** include files define variables. These files are useful for application-module makefiles, as well as for BSP makefiles.

defs.bsp
Standard variable definitions for a VxWorks run-time system.

The following include files define **make** targets, and the rules to build them. These files are usually not required for building application modules in separate directories, because most of the rules they define are specific to the VxWorks run-time system and boot programs.

rules.bsp
Rules defining all the standard targets for building a VxWorks run-time system (described in *5.4 Building a VxWorks System Image*, p.197 and *5.6 Creating Bootable Applications, p.210*). The rules for building object code from C, C++, or assembly language are also spelled out here.

5.5.1 Make Variables

The variables defined in the **make** include files provide convenient defaults for most situations, and allow individual makefiles to specify only the definitions that are unique to each. This section describes the **make** variables most often used to specify properties of BSPs or applications. The following lists are not intended to be comprehensive; see the **make** include files for the complete set.

CAUTION: Certain **make** variables are intended specifically for customization; see *Variables for Customizing the Run-Time*, p.208. Be very cautious about overriding other variables in BSP makefiles. They are described in the following sections primarily for expository purposes.

Variables for Compilation Options

The variables grouped in this section are useful for either BSP makefiles or application-module makefiles. They specify aspects of how to invoke the compiler.

CFLAGS
The complete set of option flags for any invocation of the C compiler. This variable gathers the options specified in **CC_COMPILER**, **CC_WARNINGS**, **CC_OPTIM**, **CC_INCLUDE**, **CC_DEFINES**, and **ADDED_CFLAGS**.

C++FLAGS
The complete set of option flags for any invocation of the C++ compiler. This variable gathers together the options specified in **C++_COMPILER**, **C++_WARNINGS**, **CC_OPTIM**, **CC_INCLUDE**, **CC_DEFINES**, and **ADDED_C++FLAGS**.

CC_COMPILER
Option flags specific to compiling the C language. Default: **-ansi -nostdinc**.

C++_COMPILER
Option flags specific to compiling the C++ language. Default: **-ansi -nostdinc**.

CC_WARNINGS
Option flags to select the level of warning messages from the compiler, when compiling C programs. Two predefined sets of warnings are available: **CC_WARNINGS_ALL** (the compiler's most comprehensive collection of warnings) and **CC_WARNINGS_NONE** (no warning flags). Default: **CC_WARNINGS_ALL**.

C++_WARNINGS
Option flags to select the level of warning messages from the compiler, when compiling C++ programs. The same two sets of flags are available as for C programs. Default: **CC_WARNINGS_NONE**.

CC_OPTIM
Optimization flags. Three sets of flags are predefined for each architecture: **CC_OPTIM_DRIVER** (optimization level appropriate to a device driver), **CC_OPTIM_TARGET** (optimization level for BSPs), and **CC_OPTIM_NORMAL** (optimization level for application modules). Default: **CC_OPTIM_TARGET**.

CC_INCLUDE
Standard set of header-file directories. To add application-specific header-file paths, specify them in **EXTRA_INCLUDE**.

CC_DEFINES
Definitions of preprocessor constants. This variable is predefined to propagate the makefile variable **CPU** to the preprocessor, to include any constants required for particular target architectures, and to include the value of the makefile variable **EXTRA_DEFINE**. To add application-specific constants, specify them in **EXTRA_DEFINE**.

Variables for BSP Parameters

The variables included in this section specify properties of a particular BSP, and are thus recorded in each BSP makefile. They are not normally used in application-module makefiles.

TARGET_DIR
Name of the BSP (used for dependency lists and name of documentation reference entry). The value matches the *bspname* directory name.

ROM_TEXT_ADRS
Address of the ROM entry point. Also defined in **config.h**; the two definitions must match.

ROM_SIZE
Number of bytes available in the ROM. Also defined in **config.h**; the two definitions must match.

RAM_HIGH_ADRS
RAM address where the boot ROM data segment is loaded. Must be a high enough value to ensure loading VxWorks does not overwrite part of the ROM program. Also defined in **config.h**; the two definitions must match. See *5.6 Creating Bootable Applications, p.210* for more discussion.

RAM_LOW_ADRS
Beginning address to use for the VxWorks run-time in RAM.

HEX_FLAGS
GNU **objcopy** flags. These vary by architecture; for more information, see the *GNU ToolKit User's Guide*.

LDFLAGS
Linker options for the static link of VxWorks and boot ROMs.

ROM_LDFLAGS
Additional static-link option flags specific to boot ROM images.

Variables for Customizing the Run-Time

The variables listed in this section make it easy to control what facilities are statically linked into your run-time system. You can specify values for these variables either from the **make** command line, or from your own makefiles (when you take advantage of the predefined VxWorks **make** include files).

CONFIG_ALL
Location of a directory containing the architecture-independent BSP configuration files. Set this variable if you maintain several versions of these files for different purposes. Default: *installDir*/**target/config/all**.

LIB_EXTRA
Linker options to include additional archive libraries (you must specify the complete option, including the **-L** for each library). These libraries appear in the link command before the standard VxWorks libraries.

MACH_EXTRA
Names of application modules to include in the static link to produce a VxWorks run-time. See *5.6 Creating Bootable Applications*, p.210.

BOOT_EXTRA
Names of application modules to include in the static link to produce a VxWorks boot image but not in a normal VxWorks image.

ADDED_MODULES
Do not define a value for this variable in makefiles. This variable is reserved for adding modules to a static link from the **make** command line. Its value is used in the same way as **MACH_EXTRA**, to include additional modules in the link. Reserving a separate variable for use from the command line avoids the danger of overriding any object modules that are already listed in **MACH_EXTRA**.

EXTRA_INCLUDE
Preprocessor options to define any additional header-file directories required for your application (you must specify the complete option, including the **-I**).

EXTRA_DEFINE
Definitions for application-specific preprocessor constants (you must specify the complete option, including the **-D**).

ADDED_CFLAGS
Application-specific compiler options for C programs.

ADDED_C++FLAGS
Application-specific compiler options for C++ programs.

5.5.2 Using Makefile Include Files for Application Modules

You can exploit the VxWorks makefile structure to put together your own application makefiles quickly and tersely. If you build your application directly in a BSP directory (or in a copy of one), you can use the makefile in that BSP, by specifying variable definitions (*Variables for Customizing the Run-Time*, p.208) that include the components of your application.

You can also take advantage of the Tornado makefile structure if you develop application modules in separate directories. Example 5-2 illustrates the general scheme. Include the makefile headers that specify variables, and list the object modules you want built as dependencies of a target. This simple scheme is usually sufficient, because the Tornado makefile variables are carefully designed to fit into the default rules that **make** knows about.[2]

NOTE: The target name **exe** is the Tornado convention for a default make target. You may either use that target name (as in Example 5-2), or define a different **default** rule in your makefiles. However, there must always be an **exe** target in makefiles based on the Tornado makefile headers (even if the associated rules do nothing).

Example 5-2 **Skeleton Makefile for Application Modules**

```
# Makefile - makefile for ...
#
# Copyright ...
#
# DESCRIPTION
# This file specifies how to build ...
#

## It is often convenient to override the following with "make CPU=..."
CPU             = cputype
TOOL            = gnu

include $(WIND_BASE)/target/h/make/defs.bsp

## Only redefine make definitions below this point, or your definitions
## will be overwritten by the makefile stubs above.

exe : myApp.o
```

2. However, if you are working with C++, it may be also convenient to copy the **.cpp.out** rule from *installDir*/**target/h/make/rules.bsp** into your application's makefile.

5.6 Creating Bootable Applications

As you approach a final version of your application, you will probably want to add modules to the bootable system image, and include startup of your application with the system initialization routines. In this way, you can create a *bootable application*, which is completely initialized and functional after booting, without requiring any interaction with the host-resident development tools.

5.6.1 Linking Bootable Applications

Linking the application with VxWorks is a two-step process. You must include the application initialization code in **config.h**, and you must modify the makefile to link the application statically with VxWorks.

To include the application code in **config.h**, you must:

- **#define INCLUDE_USER_APPL** (change **#undef** to **#define**)
- Modify the code fragment that defines **USER_APPL_INIT**. A template is provided; modify it as necessary to start your application:

```
#define USER_APPL_INIT \
    { \
    IMPORT int myAppInit(); \
    taskSpawn ("myApp", 30, 0, 5120, \
                mpAppInit, 0x1, 0x2, 0x3, 0,0,0,0,0,0,0); \
    }
```

To include your application modules in the bootable system image, add the names of the application object modules (with the **.o** suffix) to **MACH_EXTRA** in the makefile. For example, to link the module **myMod.o**, add a line like the following:

```
MACH_EXTRA = myMod.o
```

Building the system image with the application linked in is the final part of this step. In the target directory, execute the following command:

```
% make vxWorks
```

Application size is usually an important consideration in bootable applications. Generally, VxWorks boot ROM code is copied to a start address in RAM above the constant **RAM_HIGH_ADRS**, and the ROM in turn copies the downloaded system image starting at **RAM_LOW_ADRS**. The values of these constants are architecture dependent, but in any case the system image must not exceed the space between

the two. Otherwise the system image overwrites the boot ROM code while downloading, thus killing the booting process.

To help avoid this, the last command executed when you make a new VxWorks image is **vxsize**, which shows the size of the new executable image and how much space (if any) is left in the area below the space used for ROM code:

```
vxsize 386 -v 00100000  00020000  vxWorks
vxWorks: 612328(t) + 69456(d) + 34736(b) = 716520 (235720 bytes left)
```


If your new image is too large, **vxsize** issues a warning. In this case, you can reprogram the boot ROMs to copy the ROM code to a sufficiently high memory address by increasing the value of **RAM_HIGH_ADRS** in **config.h** and in your BSP's makefile (both values must agree). Then rebuild the boot ROMs by executing the following command:

```
% make bootrom.hex
```

The binary image size of typical boot ROM code is 128 KB or less. This small size is achieved through compression; see *Boot ROM Compression*, p.213. The compressed boot image begins execution with a single uncompressed routine, which uncompresses the remaining boot code to RAM. To avoid uncompressing and thus initialize the system a bit faster, you can build a larger, uncompressed boot ROM image by specifying the **make** target **bootrom_uncmp.hex**.

5.6.2 Creating a Standalone VxWorks System with a Built-in Symbol Table

It is sometimes useful to create a VxWorks system that includes a copy of its own symbol table. The procedure for building such a system is somewhat different from the procedure described in *5.6.1 Linking Bootable Applications*, p.210. No change is necessary to **usrConfig.c**. A different **make** target, **vxWorks.st**, specifies the standalone form of VxWorks:

```
% make vxWorks.st
```

The rules for building **vxWorks.st** create a module **usrConfig_st.o**, which is the **usrConfig.c** module compiled with the **STANDALONE** flag defined. The **STANDALONE** flag causes the **usrConfig.c** module to be compiled with the built-in system symbol table, the target-resident shell, and associated interactive routines.

The **STANDALONE** flag also suppresses the initialization of the network. If you want to include network initialization, define **STANDALONE_NET** in either of the

header files *installDir*/**target/config/***bspname*/**config.h** or *installDir*/**target/config/all/configAll.h**.[3]

VxWorks is linked as described previously, except that the first pass through the loader does not specify the final load address; thus the output from this stage is still relocatable. The **makeSymTbl** tool is invoked on the loader output; it constructs a data structure containing all the symbols in VxWorks. This structure is then compiled and linked with VxWorks itself to produce the final bootable VxWorks object module.

To include your own application in the system image, add the object modules to the definition of **MACH_EXTRA** and follow the procedures discussed in *5.6.1 Linking Bootable Applications*, p.210.

Because **vxWorks.st** has a built-in symbol table, there are some minor differences in how it treats VxWorks symbols, in contrast with the host symbol table used by the Tornado tools through the target server. First, VxWorks symbol table entries cannot be deleted from the **vxWorks.st** symbol table. Second, no local (**static**) VxWorks symbols are present in **vxWorks.st**.

5.6.3 Creating a VxWorks System in ROM

To put VxWorks or a VxWorks-based application into ROM, you must enter the object files on the loader command line in an order that lists the module **romInit.o** before **sysALib.o**. Also specify the entry point option **-e _romInit**. The **romInit( )** routine initializes the stack pointer to point directly below the text segment. It then calls **bootInit( )**, which clears memory and copies the **vxWorks** text and data segments to the proper location in RAM. Control is then passed to **usrInit( )**.

A good example of a ROM-based VxWorks application is the VxWorks boot ROM program itself. The file *installDir*/**target/config/all/bootConfig.c** is the configuration module for the boot ROM, replacing the file **usrConfig.c** provided for the default VxWorks development system. The makefiles in the target-specific directories contain directives for building the boot ROMs, including conversion to a file format suitable for downloading to a PROM programmer. Thus, you can generate the ROM image with the following **make** command:

```
% make bootrom.hex
```

3. **vxWorks.st** suppresses network initialization, but it includes the network. The **STANDALONE** option defines **INCLUDE_STANDALONE_SYM_TBL** and **INCLUDE_NETWORK**, and undefines **INCLUDE_NET_SYM_TBL** and **INCLUDE_NET_INIT**. The alternative option **STANDALONE_NET** includes **INCLUDE_NET_INIT**.

Tornado makefiles also define a ROMable VxWorks run-time system suitable for use with Tornado tools as the target **vxWorks.res_rom_nosym**. To generate this image in a form suitable for writing ROMs, run the following command:

```
% make vxWorks.res_rom_nosym.hex
```

VxWorks target makefiles also include the entry **vxWorks.st_rom** for creating a ROMable version of the standalone system described in *5.6.2 Creating a Standalone VxWorks System with a Built-in Symbol Table*, p.211. The image **vxWorks.st_rom** differs from **vxWorks.st** in two respects: (1) **romInit** code is loaded as discussed above, and (2) the portion of the system image that is not essential for booting is compressed by approximately 40 percent using the VxWorks **compress** tool (see *Boot ROM Compression*, p.213).

To build the form of this target that is suitable for writing into a ROM (most often, this form uses the Motorola S-record format), enter:

```
% make vxWorks.st_rom.hex
```

When adding application modules to a ROMable system, size is again an important consideration. Keep in mind that by using the **compress** tool, a configuration that normally requires a 256-KB ROM may well fit into a 128-KB ROM. Be sure that **ROM_SIZE** (in both **config.h** and the makefile) reflects the capacity of the ROMs used.

Boot ROM Compression

VxWorks boot ROMs are compressed to about 40 percent of their actual size using a binary compression algorithm, which is supplied as the tool **compress**. When control is passed to the ROMs on system reset or reboot, a small (8 KB) uncompression routine, which is *not* itself compressed, is executed. It then uncompresses the remainder of the ROM into RAM and jumps to the start of the uncompressed image in RAM. There is a short delay during the uncompression before the VxWorks prompt appears. The uncompression time depends on CPU speed and code size; it takes about 4 seconds on an MC68030 at 25 MHz.

This mechanism is also available to compress a ROMable VxWorks application. The entry for **vxWorks.st_rom** in the architecture-independent portion of the makefile, *installDir*/**target/h/make/rules.bsp**, demonstrates how this can be accomplished. For more information, see also the reference manual entries for **bootInit** and **compress**.

5.7 Building Projects From a BSP

In some cases, you may wish to change and customize your BSP using the techniques described in this chapter, and then build the VxWorks image for use by application developers using the project facility. This is the one case where you can "mix" the two methods of configuration. Using one of these **make** targets creates a project based on the BSP you have created with all your customizations. This project can serve as a base for further development.

WARNING: If you make additional changes to the configuration files of your BSP after you have begun using it with the project facility, these changes will not be available for subsequent project facility use because the project files override **config.h** and other configuration files.

The following **make** targets are available:

make prj_default
Builds a single project using the default toolchain and building all four default build specifications. These are **default**, **default_rom**, **default_romCompress**, and **default_romResident**.

make prj_gnu
Builds a single project using the GNU toolchain and building all four default build specifications.

make prj_diab
Builds a single project using the Diab toolchain and building all four default build specifications.

make prj_diab_def
Builds a single project using the Diab toolchain and building a single, default build specification.

make prj_gnu_def
Builds a single project using the GNU toolchain and building a single, default build specification.

make bsp2prj
The same as **prj_default_one**, which is **prj_default** with only the single, default build specification.

```
c:\> make [CPU=XXXX TOOL=YYYY] bsp2prj
```

A Tcl script is also available for **bsp2prj**; you can run it from the UNIX command line (order *is* important):

```
c:\> wtxtcl /vobs/wpwr/host/src/hutils/bsp2prj.tcl BSP [CPU TOOL]
```

6

VxSim

Integrated Simulator and Full Simulator (Optional)

6.1 Introduction

VxSim, the VxWorks simulator, is a port of VxWorks to the various host architectures. It provides a simulated target for use as a prototyping and test-bed environment. In most regards, its capabilities are identical to a true VxWorks system running on target hardware. Users link in applications and rebuild the VxWorks image exactly as they do in any VxWorks cross-development environment using a standard BSP.

The difference between VxSim and the VxWorks target environment is that in VxSim the image is executed on the host machine itself as a host process. There is no emulation of instructions, because the code is for the host's own architecture. A communication mechanism is provided to allow VxSim to obtain an Internet IP address and communicate with the Tornado tools on the host (or with other nodes on the network) using the VxWorks networking tools.

Because target hardware interaction is not possible, device driver development may not be suitable for simulation. However, the VxWorks scheduler is implemented in the host process, maintaining true tasking interaction with respect to priorities and preemption. This means that any application that is written in a portable style and with minimal hardware interaction should be portable between VxSim and VxWorks.

The basic functionality of VxSim is included with the Tornado tools and is preconfigured to allow immediate access to the simulated target. The integrated simulator does not include networking and provides only single instance usage. The VxSim full simulator is an optional product providing for networking and multiple instance usage.

The key differences between VxSim and other BSPs are summarized below. For a detailed discussion of subtle implementation differences which may affect application development, see *6.4 Architecture Considerations*, p.223.

Integrated Simulator

VxSim has only a few differences from VxWorks:

- **Drivers.** Because device drivers require direct hardware interaction, most VxWorks device drivers are not available with VxSim.
- **File System.** VxSim defaults to using a pass-through file system (passFs) to access files directly on the workstation. (See the online reference for **passFsLib** under VxWorks Reference Manual> Libraries.) Most VxWorks targets default to using **netDrv** to access files on the host.
- **Networking.** Networking is not available in the base product.

Full Simulator

The VxSim full simulator provides full network capability for your simulator. The optional product also allows you to run more than one instance of VxSim on your host.

In order to simulate the network IP connectivity of a VxWorks target, the VxSim full simulator includes special drivers which operate using IP addresses. The PPP network interface is available for UNIX hosts.

All interfaces provide an I/O-based interface for IP networking that allows VxSim processes to be addressed at the IP level. When multiple programs are run, they can send packets to each other directly. This is because the host hands the packets back and forth; that is, the host OS effectively becomes a router with multiple interfaces.

For more information on PPP, see the *VxWorks Network Programmer's Guide*.

6.2 Integrated Simulator

All the functionality of the integrated simulator is available with the optional full simulator. All the information in this section applies to both versions of VxSim. For information specific to the full simulator product, see *6.5 Configuring the VxSim Full Simulator*, p.227.

Installation and Configuration

Tornado 2.2 comes configured with basic VxSim on all hosts. Installing and starting Tornado as described in the *Tornado Getting Started Guide* installs the integrated VxSim.

Starting VxSim

You can start VxSim from the VxSim icon on the launcher or from the command line using the command **vxWorks**. Available options are:

-p[rocessorNumber]
set the processor number [0-15] (default is 0)

-r[am bytes]
set the memory size (default is 3Mbytes: 0x00300000)

WARNING: On real targets, you can use **bootChange( )** to boot another VxWorks core file on the next reboot. On simulators, changing the core file using **bootChange( )** has no effect; in other words, on the next reboot, the simulator will not start with the core file set in the boot line.

Changing the Simulator Boot Line

Because the hardware environment is different from the simulator environment, **bootChange( )** does not behave the same way on simulators as it does on real targets.

Table 6-1 **Simulator Boot Parameters**

Parameters	Comments
boot device	Only change for Shared Memory Network settings.*
processor number	Do not change.
host name	Do not change.
file name	Ignored by the simulator.
inet on ethernet (e)	Only change for Shared Memory Network settings.*
inet on backplane (b)	Only change for Shared Memory Network settings.*

Table 6-1 **Simulator Boot Parameters** *(Continued)*

Parameters	Comments
host inet (h)	Do not change.
gateway inet (g)	Only change for Shared Memory Network settings.*
user (u)	Do not change.
ftp password (pw)	Refer to description of boot parameters.†
flags (f)	Refer to description of boot parameters.†
target name (tn)	Refer to description of boot parameters.†
startup script (s)	Refer to description of boot parameters.†
other (o)	Refer to description of boot parameters.†

* See *Setting up the Shared Memory Network*, p.239.
† See *2.6.4 Description of Boot Parameters*, p.50.

NOTE: Like a real target with nonvolatile RAM (NVRAM), all values you enter in the boot parameters are saved in a file simulating NVRAM. This file is created in the same directory as VxWorks executable and is named **vxWorks.nvram***processorNumber*.

Rebooting VxSim

As with other targets, you can reboot VxSim by typing **CTRL+X** in the VxSim window.

Exiting VxSim

Type **CTRL+** in the VxSim window.

Back End

The integrated simulator uses the pipe back end (**INCLUDE_WDB_COMM_PIPE**), which is configured by default, to communicate with the target session.

System-Mode Debugging

System-mode debugging allows developers to suspend the entire VxWorks operating system.[1] One notable application of system mode is to debug ISRs, which—because they run outside any task context—are not visible to debugging tools in the default task mode. For more discussion of system mode, see the chapters *7. Shell* and *9. Debugger*.

The Solaris integrated simulator is configured by default for system mode debugging.

File Systems

VxSim can use any VxWorks file system.

The default file system is the pass-through file system, passFs, which is unique to VxSim. passFs allows direct access to any files on the host. Essentially, the VxWorks functions **open( )**, **read( )**, **write( )**, and **close( )** eventually call the host equivalents in the host library **libc.a**. With passFs, you can open any file available on the host, including NFS-mounted files. By default, the **INCLUDE_PASSFS** macro is enabled to cause this file system to be mounted on startup.

For more information on passFs, see the library entry for **passFsLib** in the *VxWorks API Reference* or HTML help. For more information on other VxWorks file systems, see the *VxWorks Programmer's Guide: Local File Systems*.

Symbols

Particular care must be taken when using absolute symbols from loaded object modules in the simulators. The VxWorks simulators execute within the memory space of a host operating system. Their actual execution address space is more constrained than is the case for the real VxWorks operating system. Absolute references to addresses must be carefully chosen in order to point to memory areas actually existing and allocated to the simulator. The values of absolute symbols defined within object modules are not modified by the loader so these values (in other words, these addresses) must be set correctly by the code developers.

---

1. System mode is sometimes also called *external mode*, reflecting that the target agent operates externally to the VxWorks system in this mode.

6.3 Building Applications

The following sections describe how to use the VxSim compilers. The recommended way to build VxSim modules is to use the project tool. For complete information on this tool, see *4. Projects*. If you are using manual methods in your project, the information required for manual builds and loading is summarized below.

This information applies to using manual methods on both the built-in version of VxSim and the optional networking product.

Defining the CPU Type

Setting the preprocessor variable **CPU** ensures that VxWorks and your applications build with the appropriate features enabled. Define this variable to **SIMSPARCSOLARIS** for your Solaris host.

The Toolkit Environment

All VxWorks simulators use the GNU C/C++ compiler.

Compiling C and C++ Modules

Only the GNU compiler is supported for **SIMSPARCSOLARIS**; the Diab compiler is not supported. If you compile using the IDE build facilities, default build settings are already in place. If you wish to modify the defaults, or if you wish to build from the command line, the following information may be helpful.

The following is an example of a compiler command line for VxSim development. The file to be compiled in this example has a base name of **applic**.

```
% ccsimso -g -ansi -DRW_MULTI_THREAD -D_REENTRANT -fvolatile
-fno_builtin -I. -I installDir/target/h -DCPU=SIMSPARCSOLARIS
-DTOOL_FAMILY=gnu -DTOOL=gnu -c applic.c
```

Option Definitions

The options shown in the example and other compiler options are detailed in the online version of the *GNU ToolKit User's Guide*. Wind River supports compiler options used in building Wind River software; see the *Guide* for a list. Other options are not supported, although they are available with the tools as shipped.

Linking an Application to VxSim

Linking and loading for VxSim are identical to other BSPs. See *5. Command-Line Configuration and Build*.

6.4 Architecture Considerations

The information in this section highlights differences between VxSim (both the integrated and full versions) and other VxWorks BSPs. These differences should be taken into consideration as you develop applications on VxSim that will eventually be ported to another target architecture.

VxSim uses the VxWorks scheduler, which behaves the same way as for any other VxWorks architecture (see *VxWorks Programmer's Guide: Basic OS*). The BSP is extensible; for example, pseudo-drivers can be written for additional timers, serial drivers, and so forth.

The rest of this section discusses some details of the VxSim implementation. Differences between VxSim and other VxWorks environments are noted where appropriate.

Supported Configurations

Most of the optional features and device drivers for VxWorks are supported by VxSim. The few that are not are hardware devices (SCSI, Ethernet), ROM configurations, and so on. The BSP makefile builds only the images **vxWorks** and **vxWorks.st** (standalone VxWorks).

Endianess

The Solaris simulator uses a big-endian environment.

Simulator Timeout

Occasionally a simulator session loses its target server connection due to the many things competing for CPU time on the host. If you find that your application is frequently losing its target server connection, adjust the back end timeout (**-Bt**) and back end retry (**-Br**) parameters when starting the target server with the launcher. To do this from the launcher, add the new values using the Backend timeout and Backend resend fields of the Create Target Server window. For example, you may want to increase the back end timeout from 1 to 3 and the retry parameter from 3 to 4:

```
-Bt 3 -Br 4
```

You can also add this string to the **tgtsvr** command when you start the target server from the command line.

The BSP Directory

Aside from the following exceptions, the VxSim BSP is the same as a VxWorks BSP:

- The **sysLib.c** module contains the same essential functions: **sysModel( )**, **sysHwInit( )**, and **sysClkConnect( )** through **sysNvRamSet( )**. Because there is no bus, **sysBusToLocalAdrs( )** and related functions have no effect.
- The file **unixSio.c** ultimately calls the host operating system **read( )** and **write( )** routines on the process's standard input and output. Nevertheless, it supports all the functionality provided by **tyLib.c**.
- The configuration header **config.h** is minimal:
  - It does not reference a *bspname***.h** file.
  - Most network devices are excluded.
  - The boot line has no fixed memory location. Instead, it is stored in the variable **sysBootLine** in **sysLib.c**.
- The **Makefile** is the standard version for VxWorks BSPs. It does not build boot ROM images (although the makefile rules remain intact); it can only build **vxWorks** and **vxWorks.st** (standalone) images. The final linking does not arrange for the TEXT segment to be loaded at a fixed area in RAM, but follows the usual loading model. The makefile macro **MACH_EXTRA** is provided so that users can easily link their application modules into the VxWorks image if they are using manual build methods.

- The **solarisDrv.a** file is the library for solaris BSP drivers.

The BSP file **sysLib.c** can be extended to emulate the eventual target hardware more completely.

Interrupts

Host signals are used to simulate hardware interrupts. For example, VxSim uses the **SIGALRM** signal to simulate system clock interrupts, the **SIGPROF** signal for the auxiliary clock, and the **SIGVTALRM** signal for virtual timer interrupts. Furthermore, all host file descriptors (such as standard input) are put in asynchronous mode, so that the **SIGIO** signal is sent to VxSim when data becomes ready. The signal handlers are the VxSim equivalent to Interrupt Service Routines (ISRs) on other VxWorks targets.

You can install ISRs in VxSim to handle these "interrupts." Not all VxWorks functions can be called from ISRs; see the *VxWorks Programmer's Guide*: *Basic OS*.

To run ISR code during a future system clock interrupt, use the watchdog timer facilities. To run ISR code during auxiliary clock interrupts, use the **sysAuxClk***xxx***( )** functions.

Table 6-2 shows how the interrupt vector table is set up.

Table 6-2 **Interrupt Assignments**

Interrupts	Assigned To
1–32	host signals
33–64	host file descriptors 1-32 (**SIGIO**)

Pseudo-drivers can be created to use these interrupts. Interrupt code must be connected with the standard VxWorks **intConnect( )** mechanism.

For example, to install an ISR that logs a message whenever host signal **SIGUSR2** arrives, execute the following:

```
-> intConnect (17, logMsg, "Help!\n")
```

Then send signal 17 to VxSim from a host task, for example using the host **kill** command. Every time the signal is received, the ISR (**logMsg( )** in this case) runs.

⚠ **CAUTION:** Do not use the preprocessor constants **SIGUSR1** or **SIGUSR2** for this purpose in VxWorks applications, since those constants evaluate to the VxWorks definitions for these signals. You need to specify your host's signal numbers instead.

⚠ **CAUTION:** Only **SIGUSR1** (16 on Solaris 2 hosts) and **SIGUSR2** (17 on Solaris 2 hosts) can be used to represent user-defined interrupts.

If a VxSim task reads from a host device, the task would normally require a blocking read; however, this would stop the VxSim process entirely until data is ready. The alternative is to put the device into asynchronous mode so that a **SIGIO** signal is sent whenever data becomes ready. In this case, an input ISR reads the data, puts it in a buffer, and unblocks some waiting task.

To install an ISR that runs whenever data is ready on some underlying host device, first open the host device (use **u_open( )**, the underlying host routine, *not* the VxSim **open( )** function). Put the file descriptor in asynchronous mode, using the VxSim-specific routine **s_fdint( )** so that the host sends a **SIGIO** signal when data is ready. Finally, connect the ISR. The following code fragment does this on one of the host serial ports:

```
...
fd = u_open ("/dev/ttyb", 2);
s_fdint (fd, 1);
intConnect (32 + fd, ISRfunc, 0);
...
```

Since VxSim uses the task stack when taking interrupts, the task stacks are artificially inflated to compensate. You may notice this if you spawn a task of a certain size and then examine the stack size.

Clock and Timing Issues

The execution times of VxSim functions are not, in general, the same as on a real target. For example, the VxWorks **intLock( )** function is normally very fast because it just writes to the processor status register. However, under VxSim, **intLock( )** is relatively slow because it makes a host system call to mask signals.

The clock facilities are provided by the host routine **setitimer( )** (**ITIMER_REAL** for the system clock; **ITIMER_PROF** for the auxiliary clock). The problem with using **ITIMER_REAL** for the system clock is that it produces inaccurate timings when VxSim is swapped out as a host process. On the other hand, the timing of VxSim is, in general, different than on an actual target, so this is not really a problem.

NOTE: Because VxSim is a host process, it shares resources with all other processes and is swapped in and out. In addition, the kernel's idle loop has been modified to suspend VxSim until a signal arrives (rather than busy waiting), thus allowing other processes to run.

The BSP system clock can be configured to use the virtual timer (**ITIMER_VIRTUAL**) in addition to **ITIMER_REAL**; see **sysLib.c**. In this way, when the process is swapped out by the host, VxSim does not count wall-clock elapsed time as part of simulated elapsed time. VxSim still uses **ITIMER_REAL** to keep track of the elapsed time during the *wind* kernel's idle loop. Although the addition of **ITIMER_VIRTUAL** results in more accurate *relative* time, the problem is that the host system becomes increasingly loaded (due to the extra signal generation) and as a result connections to the outside world (such as the network) become delayed and can fail.

The **spy( )** facility is built on top of the auxiliary clock (**ITIMER_PROF**). The task monitoring occurs during each interrupt of the auxiliary clock to see which task is executing or if the kernel is executing. Because the profiling timer includes host system time and user time, discrepancies can occur, especially if intensive host I/O occurs.

6.5 Configuring the VxSim Full Simulator

This section contains information pertaining only to the VxSim full simulator. (All information in previous sections also pertains to that product, as well as to the integrated version.) The VxSim full simulator provides networking facilities. Most of the special considerations associated with it are network considerations.

If you purchase the VxSim optional full simulator for networking, you must take additional configuration steps:

- Install the optional VxSim component using **SETUP**, either when you install Tornado 2.2 or at a later time. (For more information, see the*Tornado Getting Started Guide*.)
- Install the appropriate network driver on your host. (See *Installing VxSim Network Drivers*, p.228.)
- Configure VxWorks to use networking, rebuild it, and download it using either the project facility or manual methods. (See *Configuring VxSim for Networking*, p.234.)

WARNING: Project facility configuration and building of projects is independent of the methods used for configuring and building applications prior to Tornado 2.x (which included manually editing **config.h** and **configAll.h**). Use of the project facility is the recommended, and is much simpler. However, the manual method may still be used (see *5. Command-Line Configuration and Build* for details). Avoid using the two methods together for the same project except where specific BSP and driver macros are not available in the project facility.

- Be sure to correctly set target server options for the full simulator from the Create Target Server dialog of the launcher:
  - Select **wdbrpc** as the back end in the Backend list. (This differs from the integrated simulator which uses the **wdbpipe** backend; selecting the wrong back end generates an error message)
  - Set the IP address of the simulator in the Target name or IP address field.

Installing VxSim Network Drivers

The **SETUP** tool writes the appropriate host drivers on your disk, but they must be installed on your host operating system. PPP is the network interface provided for Solaris hosts.

WARNING: The *VxWorks Network Programmer's Guide* states that the PPP link can serve as an additional network interface apart from the existing default network interface. This is not the case with VxSim; the simulator only support one PPP interface per simulator.

CAUTION: If you have problems using the PPP driver after following the directions below, you may have to reboot your Solaris machine to reload the drivers. Hold down the STOP key (on some Sun workstations, this is the L1 key), and hit the A key to enter the boot monitor. Then reboot the machine by issuing **boot -r** from the boot monitor. The **-r** option tells the system to reconfigure for the new device(s).

Loading PPP on a Solaris 2.7 or 2.8 Host

First, use the command **pkginfo** to check whether the following packages are installed on your host:

SUNWapppr	PPP/IP Asynchronous PPP daemon configuration files
SUNWapppu	PPP/IP Asynchronous PPP daemon and PPP login service
SUNWpppk	PPP/IP and IPdialup Device Drivers
SUNWpppkx	PPP/IP and IPdialup Device Drivers (64-bit)
SUNWbnur	Networking UUCP Utilities (Root)
SUNWbnuu	Networking UUCP Utilities (Usr)

For example:

```
% pkginfo | egrep 'ppp|bnu'
system SUNWapppr    PPP/IP Asynchronous PPP daemon configuration files
system SUNWapppu    PPP/IP Asynchronous PPP daemon and PPP login service
system SUNWpppk     PPP/IP and IPdialup Device Drivers
system SUNWpppkx    PPP/IP and IPdialup Device Drivers (64-bit)
system SUNWbnur     Networking UUCP Utilities, (Root)
system SUNWbnuu     Networking UUCP Utilities, (Usr)
```

The SUNWpppkx package should only be installed on Solaris hosts running in 64-bit mode. Use the **isainfo -b** command to determine the mode in which your Solaris 2.7 or 2.8 host runs. This command returns 32 or 64. If the packages are not already installed, mount the Solaris installation disk and change your working directory to the location of these packages (for example, on a Solaris 2.7 CD-ROM, they can be found in **/cdrom/sol7_599_sparc_sun_srvr/s0/Solaris_2.7/Product**) and install them using the following commands:

- For a Solaris host running in 32 bits:

```
% isainfo -b
32
% su root
Password:
# pkgadd -d 'pwd' SUNWbnur SUNWbnuu SUNWpppk SUNWapppr SUNWapppu
```

- For a Solaris host running in 64 bits:

```
% isainfo -b
64
% su root
Password:
# pkgadd -d 'pwd' SUNWbnur SUNWbnuu SUNWpppk SUNWpppkx SUNWapppr SUNWapppu
```

Next, as root, copy *installDir*/**target/config/solaris/asppp.cf** to the **/etc** directory.

```
# cp installDir/target/config/solaris/asppp.cf /etc
```

CAUTION: If you already have **aspppd** running, stop it with **asppp stop** before proceeding.

Finally, start the PPP daemon aspppd by typing the following as root:

```
# /etc/init.d/asppp start
```

The PPP driver is now installed and running on your Solaris system, and will be restarted automatically when Solaris reboots.

The PPP configuration assigns IP addresses 192.168.255.1 through 192.168.255.16 to sixteen devices, and associates with them the peer system names **vxsim0** through **vxsim15** respectively, as configured in **asppp.cf**.

If those IP addresses are not suitable, you can update them in **asppp.cf** and modify the **VXSIM_IP_ADDR** parameter from the workspace (the default is "192.168.255.%d"). This parameter belongs to the hardware>BSP configuration variants component. You may want to directly change that parameter in the **bsp.cdf** and **config.h** files; in that case it needs to be done only once. Finally, copy the **asppp.cf** file as explained above and rebuild your project or your BSP.

You can also use the following commands to start or stop the Solaris PPP driver after the driver has been installed (you must have root privileges):

```
# /etc/init.d/asppp start
# /etc/init.d/asppp stop
```

If you get the following error message at simulator startup, you need to check access rights to the file **/tmp/.asppp.fifo**:

```
Can't open /tmp/.asppp.fifo
solaris_ppp_init failed
Attaching network interface ppp0...
ppp0: ppp 2.1.2 started by
ppp0: Connect: ppp0 <--> /tyCo/1
ppp0: timeout: could not establish link with peer.
Unable to initialize PPP connection.
```

When you attempt to view permissions on the file, you may get the following:

```
% ls -l /tmp/.asppp.fifo
prw------- 1 root other 0 Oct 18 17:04 /tmp/.asppp.fifo|
```

In this case, as root, add the following command to the **/etc/init.d/asppp** file:

```
chmod a+rw /tmp/.asppp.fifo
```

Add the preceding command line at the end of the following code:

```
% su root
password:
# vi /etc/init.d/asppp

# Start the aspppd daemon
/usr/sbin/aspppd -d 1 ||
echo "aspppd not started, see /var/adm/log/asppp.log"
```

Save the file and restart PPP.

WARNING: The above problem is likely to occur on Solaris 2.8 hosts only.

If you want to use Tornado 2.0 and Tornado 2.2 Solaris simulators on the same machine, you should assign similar IP addresses for both simulators, that is, those declared in the **asppp.cf** file. For Tornado 2.2, default addresses are of the form 192.168.255.*Processor_Number*+1, whereas, for Tornado 2.0, they are of the form 127.0.1.*Processor_Number*.

In order to use the two simulators simultaneously, you must modify either the Tornado 2.2 IP addresses or the Tornado 2.0 IP addresses. The better solution is to modify the Tornado 2.0 addresses, as follows:

- In the file **sysLib.c** (in the directory *installDir*/**target/config/solaris**), modify the values of the global variables **vxsim_gateway** and **vxsim_ip_addr**. Change from:

```
...
#elif defined(INCLUDE_PPP)
char *vxsim_bootdev = "ppp";
char *vxsim_gateway = "g=127.0.1.254";
#else
...
char *vxsim_ip_addr = "127.0.1.%d";
```

  To:

```
char *vxsim_gateway = "g=192.168.255.254";
char *vxsim_ip_addr = "192.168.255.%d";
```

- Find the following line in the **sysHwInit2( )** routine:

```
sprintf (target_ip, vxsim_ip_addr, sysProcNumGet ());
```

  Replace it with the following:

```
sprintf (target_ip, vxsim_ip_addr, (sysProcNumGet () + 1));
```

When you have made these changes, rebuild your system.

Loading PPP on a Solaris 2.9 Host

First, use the command **pkginfo** to check whether the following packages are installed on your host:

SUNWpppd	Solaris PPP Device Drivers
SUNWpppdr	Solaris PPP configuration files
SUNWpppdu	Solaris PPP daemon and utilities
SUNWpppdt	Solaris PPP Tunneling
SUNWpppdx	Solaris PPP Device Drivers (64-bit)
SUNWpppg	GNU utilities for PPP
SUNWbnur	Networking UUCP Utilities (Root)
SUNWbnuu	Networking UUCP Utilities (Usr)

For example:

```
% pkginfo | egrep 'ppp|bnu'
system      SUNWbnur       Networking UUCP Utilities, (Root)
system      SUNWbnuu       Networking UUCP Utilities, (Usr)
system      SUNWpppd       Solaris PPP Device Drivers
system      SUNWpppdr      Solaris PPP configuration files
system      SUNWpppdt      Solaris PPP Tunneling
system      SUNWpppdu      Solaris PPP daemon and utilities
system      SUNWpppdx      Solaris PPP Device Drivers (64-bit)
system      SUNWpppg       GNU utilities for PPP
```

The SUNWpppdx package should only be installed on Solaris hosts running in 64-bit mode. Use the **isainfo -b** command to determine the mode in which your Solaris 2.9 host runs. This command returns 32 or 64. If the packages are not already installed, mount the Solaris installation disk and change your working directory to the location of these packages and install them using the following commands:

- For a Solaris host running in 32 bits:

```
% isainfo -b
32
% su root
```

```
Password:
# pkgadd -d  'pwd' SUNWbnur SUNWbnuu SUNWpppd SUNWpppdr SUNWpppdt
SUNWpppdu SUNWpppg
```

For a Solaris host running in 64 bits:

```
% isainfo -b
64
% su root
Password:
# pkgadd -d  'pwd' SUNWbnur SUNWbnuu SUNWpppd SUNWpppdr SUNWpppdt
SUNWpppdu SUNWpppdx SUNWpppg
```


Next, still as root, copy *installDir*/**target/config/solaris/vxsimppp** to the **/etc/init.d** directory. If you want PPP daemons to be automatically started during Solaris startup, then you need to create a link in **/etc/rc2.d**:

```
# cp installDir/target/config/solaris/vxsimppp /etc/init.d
# ln -s /etc/init.d/vxsimppp /etc/rc2.d/S80vxsimppp
```

Finally, start the PPP daemons **pppd** by typing the following as root:

```
# /etc/rc.d/init.d/vxsimppp start
```

The following table describe the configurable parameters of that script. The default values will usually be appropriate, but make sure there is no conflict with other applications.

Table 6-3 **Configurable Parameters of vxsimppp**

Parameters	Default	Description
PPP_MASTER_PSEUDO_TTY_PATH	"/dev/ttyr%x"	PPP master pseudo-terminal path. This value must be coherent with the slave pseudo-terminal path defined in the project facility (parameter **PPP_PSEUDO_TTY_PATH** of the **INCLUDE_SOLARIS_NET_CONFIG** component). By default the master is **"/dev/ttyr%x"** and slave is **"/dev/ptyr%x"**.
ENABLE_REMOTE_ACCESS	1 (true)	To enable remote access of VxSim targets, IP forwarding must be enabled. This is done by setting the Solaris kernel parameter **ip_forwarding** to true using the **ndd** command. To disable this facility, reset **ENABLE_REMOTE_ACCESS** to 0.

Table 6-3 **Configurable Parameters of vxsimppp** *(Continued)*

Parameters	Default	Description
FIRST_VXSIM **LAST_VXSIM**	0 15	This defines the simulator IP address range so by default: vxsim0 192.168.255.1 vxsim1 192.168.255.2 ... ... vxsim15 192.168.255.16
DEBUG	0 (false)	Enables PPP connection debugging facilities. If this option is given, **pppd** will log the contents of all control packets sent or received in a readable form. The packets are logged through **syslog** with facility daemon and level debug. This information can be directed to a file by setting up **/etc/syslog.conf** appropriately (see **syslog.conf** manual).

The PPP daemons are now installed and running on your Solaris system, and will be restarted automatically when Solaris reboots.

The PPP configuration assigns IP addresses 192.168.255.1 through 192.168.255.16 to sixteen devices. You may want to match these IP addresses with host names in **/etc/hosts**; for example: vxsim vxsim0 through vxsim15. If those IP addresses are not suitable, you can modify **VXSIM_IP_ADDR** parameter from the workspace (the default is "192.168.255.%d"). This parameter belongs to the hardware>BSP configuration variants component. You may want to directly change that parameter in the **bsp.cdf** and **config.h** files, in that case, it need be done only once. Finally, rebuild your project or your BSP.

You can also use the following commands to start or stop the Solaris PPP driver after the driver has been installed (you must have root privileges):

```
# /etc/init.d/vxsimppp start
# /etc/init.d/vxsimppp stop
```

Configuring VxSim for Networking

As with any other BSP, adding components to VxWorks requires including them, rebuilding VxWorks, then downloading and restarting it. The easiest method for doing this is to use the project facility. However, if you have used manual methods in your project, you should continue to use those methods.

For a discussion of networking as it relates to VxSim, see *6.5 Configuring the VxSim Full Simulator*, p.227.

Using the Project Facility

Use the Create Project facility to create a bootable VxWorks image.

- On the VxWorks tab in the Project Workspace window, select the folder called network components. Right click and select Include 'network components' from the pop-up menu. Click OK to accept the defaults.
- Change WDB connection from WDB simulator pipe connection to WDB network connection; this change generates a project configuration error since WDB system mode is incompatible with WDB network connection. To fix this conflict, right click WDB system debugging and select Exclude.

For more information on using the configuration tool, see *4. Projects*.

Using Manual Techniques

Edit **target/config/solaris/config.h**, and replace:

```
#if TRUE
#undef INCLUDE_NETWORK
...
```

With:

```
#if FALSE
#undef INCLUDE_NETWORK
...
```

Then rebuild and download VxWorks.

You must also change your target server configuration from **wdbpipe** to **wdbrpc**. From the Create Target Server dialog of the launcher, select wdbrpc in the Backend list, and set the IP address of the simulator in the Target name or IP address field.

For additional information on configuring BSPs using manual methods, see the *VxWorks Network Programmer's Guide*.

Running Multiple Simulators

When you install the optional VxSim component, your system is automatically configured to run up to 16 simulators. When you start VxSim from the launcher, you can specify the processor number from the Launch VxSim window. The processor number must be a positive number ranging from 0 (first instance:

vxsim0) to 15 (last instance: **vxsim15**). To start VxSim from the command line, the command takes the following form (where *n* is the processor number):

vxsim0 starts. To start additional instances, use the command line. The command takes the following form (where *n* is the processor number):

```
% vxWorks -p n
```

WARNING: Killing the full simulator with **kill -9** will prevent PPP from cleanly ending its connection. If you cannot exit the simulator using **CTRL+** in the VxSim window or by using **kill** *vxWorks_processId*, then use **kill -9** *vxWorks_processId*, knowing you will have to restart PPP using the following commands:

Solaris 2.7 or 2.8 (as root):

```
# /etc/init.d/asppp stop
# /etc/init.d/asppp start
```

Solaris 2.9 (as root):

```
# /etc/init.d/vxsimppp stop
# /etc/init.d/vxsimppp start
```

System Mode Debugging

The full simulator does not support system mode debugging because of an incompatibility with the RPC back end.

IP Addressing

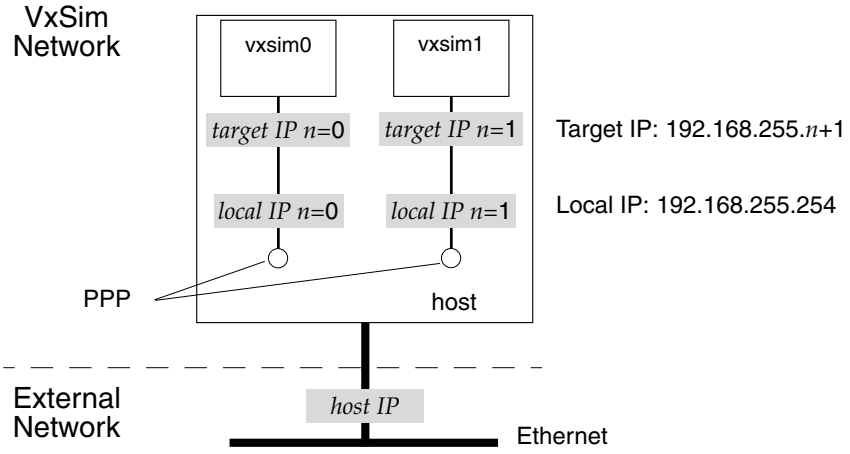
All of the networking facilities available under VxWorks—for example, sockets, RPC, NFS—are available with VxSim. For VxSim to communicate with the outside world, it must have its own target IP address as provided through a network interface.

Internet addressing is handled slightly differently among the available network interfaces. For each VxSim process, there are three associated IP addresses:

- Target IP – the address of each VxSim process, internal to your host.
- Local IP – your host's address on the VxSim network, internal to your host.
- Host IP – your host's address according to the network at your site.

The target IP address and the local IP address communicate according to the protocol of the chosen network interface. The host IP address is not directly relevant to the VxSim network.

Figure 6-1 **VxSim IP Addressing**



Addressing is according to processor number, such that when you run VxSim with processor number *n* (with the command **vxWorks -p** *n*), the network addresses packets as shown: .

Target IP	197.168.255.*n*+1
Local IP	192.168.255.254

PPP (Solaris 2.7, 2.8, and 2.9)

When you run VxSim with PPP and specify processor number *n* (with the command **vxWorks -p** *n*), VxSim creates a network connection to the IP address 192.168.255.*n*+1 by communicating through a pipe. Normally, VxWorks uses PPP over a serial device to connect to the host (see the *VxWorks Network Programmer's Guide*). The only difference with PPP is that a pipe replaces the physical serial link.

Only one process at a time can open the same PPP device. Thus, if you want multiple VxSim targets to use PPP, you *must* give each of them a distinct processor number. If another VxSim process is already running with the same processor number, the following message is displayed during the startup of VxSim:

```
Target Name: vxTarget
Attaching network interface ppp0...
ppp0: ppp 2.1.2 started by
```

```
ppp0: Connect: ppp0 <--> /tyCo/1

ppp0: timeout: could not establish link with peer.
usrPPPInit() returned errno = 0x3d0001
Bind failed
wdbConfig: error configuring WDB communication interface


                 VxWorks

Copyright 1984-2001  Wind River Systems, Inc.

          CPU: SunOS 5.7 [sun4u]
      VxWorks: VxWorks5.5
  BSP version: 1.2/1
      Created: Jan 23 2002, 17:01:44
          WDB: Agent configuration failed.
```

Setting Up Remote Access

You can add host-specific routing entries to the local host to allow remote hosts to connect to a local VxSim "target." IP addresses are set up only for the host where the network simulation software is installed. The network interface does not have to be installed remotely; the remote host uses the local host as the gateway to the VxSim target.

In the example shown in Figure 6-2, **host1** can communicate with **vxsim0** or **vxsim1** if the following steps are taken:

Issue the following commands on **host1** (as **root**):

```
% route add host 192.168.255.1 90.0.0.1 1
% route add host 192.168.255.2 90.0.0.1 1
```

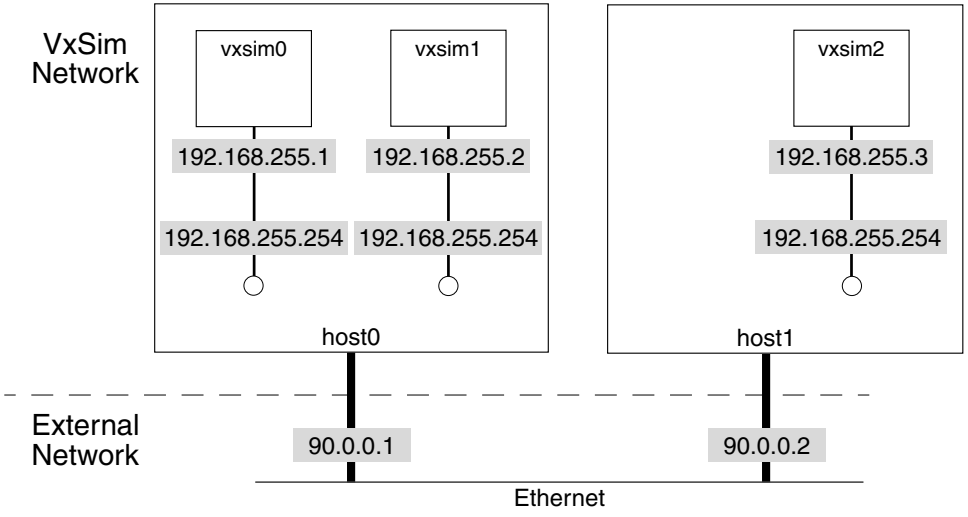
Contrast Figure 6-2 below with Figure 6-1, to see the way addresses are set up, paying particular attention to the addressing algorithm.

Verify the success of the above commands by pinging **vxsim0** from **host1**:

```
% ping 192.168.255.1
```

To allow a VxSim process on one host to communicate with a VxSim process on a different host, you must make sure that the two VxSim processes have different IP addresses. You must also make additional host-specific routes using unique addresses for each process.

Figure 6-2 **Example of VxSim IP Addressing (PPP on Solaris)**



For example, to ping **vxsim2** from **host0** above, you must add an additional route from **host0** as follows:

```
% route add host 192.168.255.3 90.0.0.2 1
```

IP Forwarding

To enable remote access to a simulator, IP forwarding must be enabled. This can be done using the following command (as root):

```
% ndd -set /dev/tcp ip_forwarding 1
```

Use the following command ti check current setting:

```
% ndd -get /dev/tcp ip_forwarding
```

NOTE: On Solaris 2.9, IP forwarding is enabled by default in the **vxsimppp** script. To disable this facility, reset the **ENABLE_REMOTE_ACCESS** parameter to 0.

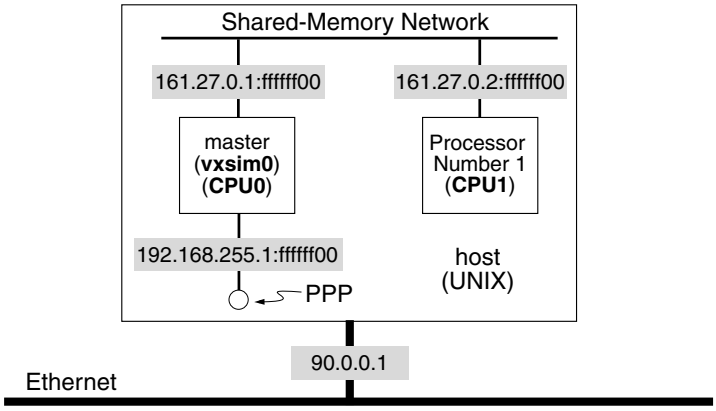
Setting up the Shared Memory Network

Many VxWorks users connect multiple CPU boards through a backplane (for example, VMEbus), which allows the boards to communicate through shared

memory. VxWorks provides a standard network driver to access this shared memory so that all the higher level network protocols are available over the backplane. In a typical configuration, one of the CPU boards (CPU 0) communicates with the host using Ethernet. The rest of the CPU boards communicate with each other and the host using the shared memory network, using CPU 0 as a gateway to the outside world. For more information on this configuration in a normal VxWorks environment, see *VxWorks Network Programmer's Guide*.

This configuration can be emulated for VxSim (the full simulator version). Multiple VxSim processes use a host shared-memory region as the basis for the shared memory network (see Figure 6-3).The full simulator uses the point-to-point interface mode to communicate and does not support broadcasting. However, inside the shared memory network, simulators use broadcast mode to communicate.

Figure 6-3 **VxSim Shared Memory Network**



Getting a shared-memory network configured for the first time can be tricky. Follow these steps to configure both master and slave simulators:

1. The following components should be set in both simulator configurations:

   INCLUDE_SM_COMMON
   INCLUDE_SM_NET
   INCLUDE_SM_NET_ADDRGET
   INCLUDE_SM_OBJ
   INCLUDE_SM_SEQ_ADDR

2. To configure the master simulator, set the **INCLUDE_SECOND_SMNET** component.
3. Add the target shell facility (required to add the route when the slave simulator starts).

To set up a shared-memory network, use a subnet mask of 0xffffff00 to create a 161.27.0.0 subnet (from the 192.168.255.1 network) for the shared-memory network. The following steps are required:

1. Use the **bootChange( )** command from the Tornado shell to change the following boot parameter on CPU 0. You must specify the subnet mask, as follows:

   ```
   inet on backplane (b): 161.27.0.1:ffffff00
   ```

   The first time you boot the simulator CPU 0, the following message is displayed at the beginning of startup. It indicates that the boot file with shared-memory information is missing.

   ```
   Attaching shared memory objects at 0xef680000... done
   Attaching network interface ppp0...
   ppp0: ppp 2.1.2 started by
   ppp0: Connect: ppp0 <--> /tyCo/1
   ppp0: local  IP address 192.168.255.1
   ppp0: remote IP address 192.168.255.254
   done.
   Attaching interface lo0...done
   Backplane IP address required if no Proxy ARP
   ```

   The **bootChange( )** command creates a boot file containing the information necessary to set up the shared memory network.

2. Restart VxSim by typing **^X**. When VxSim boots, it sets up the shared-memory network and prints the address of the shared-memory region it has created (in the VxSim console window, with the other boot messages).

   ```
   Attaching shared memory objects at 0xef680000... done
   Attaching network interface ppp0...
   ppp0: ppp 2.1.2 started by
   ppp0: Connect: ppp0 <--> /tyCo/1
   ppp0: local  IP address 192.168.255.1
   ppp0: remote IP address 192.168.255.254
   done.
   Attaching interface lo0...done
   Initializing backplane net with anchor at 0xef680000... done.
   Backplane anchor at 0xef680000... Backplane IP address = 161.27.0.1
   Attaching network interface sm0... done.
   ```

3. Start CPU 1 (**vxWorks -p 1**), and then use **bootChange( )** to set the following boot parameters on CPU 1. For the *boot device* parameter, use the address printed in step 2 (in this case, 0xef680000). Leave the *inet on ethernet* parameter blank by typing a period (.).

```
boot device           :  sm=0xef680000
inet on ethernet (e)  :  .
inet on backplane (b) :  161.27.0.2:ffffff00
gateway inet (g)      :  161.27.0.1
```

4. Quit CPU 1 and restart it. When it comes up again, it should attach to the shared-memory network.

```
Attaching shared memory objects at 0xef680000... done
Backplane anchor at 0xef680000... Attaching network interface sm0...
done.
Attaching interface lo0...done
```

5. Add the route to the slave simulator's routing table:

```
-> routeAdd ("192.168.255.0", "161.27.0.1")
```

6. To verify that everything is working correctly, ping CPU 1 (from a shell attached to CPU 0) with the following command:

```
-> ping "161.27.0.2"
```

NOTE: Any time you need to attach a VxSim process within the subnet to the target server, you must specify it by its new IP address rather than by the host name, because all VxSim processors other than 0 are no longer directly accessible to the external network. The processors use **vxsim0** as the gateway. The host names normally associated with VxSim IP addresses cannot be used, because the routing table entries point to their usual IP addresses. For example, **vxsim1** is normally associated with IP address 192.168.255.2; with the shared-memory network active, CPU 1 must be addressed through the subnet as 161.27.0.2.

7. Until you configure your UNIX routing table with information on how to reach the new subnet, you will be unable to use network communication between CPU 1 and the host over the shared-memory network. To configure the route from UNIX, use the following commands:

```
% su root
password:
# route add net 161.27.0.0 192.168.255.1 1
# exit
%
```

To attach a target session to CPU 1, increase the default time-out value if necessary:

```
% tgtsvr 161.27.0.2 -Bt 10.
```

8. Verify that you can now communicate from the host to CPU 1 over the shared-memory network by using **ping** from the host to CPU 1.

```
% ping 161.27.0.2
```

If you attempt to access CPU 1 through its normally associated IP address, it appears to be unavailable:

```
% ping 192.168.255.2
ping: no answer
```


NOTE: The master simulator should always be started first, before all the slave simulators. If you reboot the master simulator, wait for its entire initialization (shell prompt) before rebooting all the slave simulators.

NOTE: The optional product VxMP can be used with the Solaris full simulator. This product provides shared semaphores and other shared-memory objects to multiple VxWorks targets over the backplane. VxMP is part of the full simulator product. For more information on VxMP, see the *VxWorks Programmer's Guide: Shared-Memory Objects*.

7 Shell

7.1 Introduction

The Tornado shell, WindSh, allows you to download application modules, and to invoke both VxWorks and application module subroutines. This facility has many uses: interactive exploration of the VxWorks operating system, prototyping, interactive development, and testing.

WindSh can interpret most C language expressions; it can execute most C operators and resolve symbolic data references and subroutine invocations. You can also interact with the shell through a Tcl interpreter, which provides a full set of control structures and lower-level access to target facilities. For a more detailed explanation of the Tcl interface, see *7.7 Tcl: Shell Interpretation*, p.297.

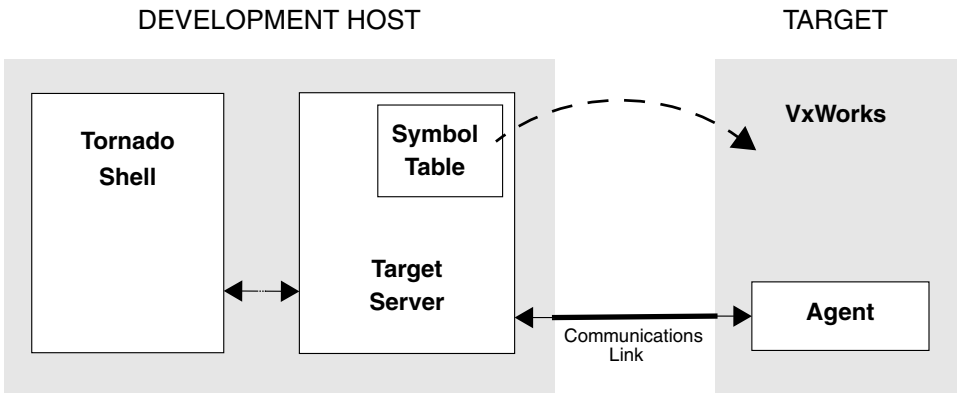
WindSh executes on the development host, not the target, but it allows you to spawn tasks, to read from or write to target devices, and to exert full control of the target.[1] Because the shell executes on the host system, you can use it with minimal intrusion on target resources. As with other Tornado tools, only the target agent is required on the target system. Thus, the shell can remain always available; you can use it to maintain a production system if appropriate as well as for experimentation and testing during development.

Shell operation involves three components of the Tornado system, as shown in Figure 7-1.

1. A target-resident version of the shell is also available; for more information, see *VxWorks Programmer's Guide: Target Shell*.

- The *shell* is where you directly exercise control; it receives your commands and executes them locally on the host, dispatching requests to the target server for any action involving the symbol table or target-resident programs or data.
- The *target server* manages the symbol table and handles all communications with the remote target, dispatching function calls and sending their results back as needed. (The symbol table itself resides entirely on the host, although the addresses it contains refer to the target system.)
- The *target agent* is the only component that runs on the target; it is a minimal monitor program that mediates access to target memory and other facilities.

Figure 7-1 **Tornado and the Shell**



The shell has a dual role:

- It acts as a command interpreter that provides access to all VxWorks facilities by allowing you to call any VxWorks routine.
- It can be used as a prototyping and debugging tool for the application developer. You can run application modules interactively by calling any application routine. The shell provides notification of any hardware exceptions. See *System Modification and Debugging*, p.260, for information about downloading application modules.

The capabilities of WindSh include the following:

- task-specific breakpoints
- task-specific single-stepping
- symbolic disassembler
- task and system information utilities

- ability to call user routines
- ability to create and examine variables symbolically
- ability to examine and modify memory
- exception trapping

7.2 Using the Shell

The shell reads lines of input from an input stream, parses and evaluates each line, and writes the result of the evaluation to an output stream. With its default C-expression interpreter, the shell accepts the same expression syntax as a C compiler with only a few variations.

The following sections explain how to start and stop the shell and provide examples illustrating some typical uses of the shell's C interpreter. In the examples, the default shell prompt for interactive input in C is "->". User input is shown in **`bold face`** and shell responses are shown in a `plain roman face`.

7.2.1 Starting and Stopping the Tornado Shell

There are two ways to start a Tornado shell:

- From the Tornado Launch window: Select the desired target and press the 

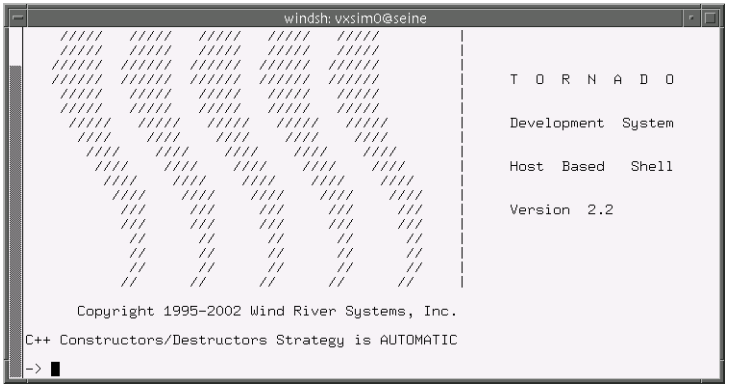
button.
- UNIX command line: Invoke **windsh**, specifying the target server name as in the following example:

```
% windsh phobos
```

In the first case, a shell window like that shown in Figure 7-2 appears, ready for your input at the -> prompt. In the second case, WindSh simply executes in the environment where you call it, using the parent shell's window.

Regardless of how you start it, you can terminate a Tornado shell session by executing the **exit( )** or the **quit( )** command or by typing your host system's end-of-file character (usually **CTRL+D**). If the shell is not accepting input (for instance, if it loses the connection to the target server), you can use the interrupt key (**CTRL+C**).

Figure 7-2 **WindSh Initial Display**



You may run as many different shells attached to the same target as you wish. All functions called from a shell have their output redirected to the WindSh window from which they received input unless you changed the shell defaults using **shConfig** (see *WindSh Environment Variables*, p.252).

You can also redirect **windsh** input and output to other UNIX commands, as in the following test (using the shell built-in command **i( )** to report on what tasks are running on the target) to look for a target-system task called **tPortmapd**:

```
% echo "i" | windsh -q ev386ex@yunque | grep tPortmapd
tPortmapd   _portmapd   3a4280 100 PEND    2afc4  3a3f28  16 0
```

Because you can start as many Tornado shell sessions as you like, such combinations of the Tornado shell and the UNIX shell do not interfere with interactive **windsh** sessions.

7.2.2 Downloading From the Shell

One of the most useful shell features for interactive development is the dynamic linker. With the shell command **ld( )**, you can download and link new portions of the application.

```
-> ld < /home/moduleDir/module.o
```

Because the linking is dynamic, you only have to rebuild the particular piece you are working on, not the entire application. Download can be cancelled with **CTRL+C** or by clicking Cancel in the load progress indicator window. The dynamic linker is discussed further in *5.4.4 Linking the System Modules*, p.201.

The WTX error **(0x10197) EXCHANGE_TIMEOUT** may occur when a WTX request keeps the target server busy longer than 30 seconds (default timeout). This may happen when loading a large object module. A Tcl procedure is available to change the default timeout. From WindSh use **wtxTimeout** *sec* where *sec* is the number of seconds before timeout:

```
-> ?wtxTimeout 120
```

7.2.3 Shell Features

The shell provides many features which simplify your development and testing activities. These include command name and path completion, command and function synopsis printing, automatic data conversion, calculation with most C operators and variables, and help on all shell and VxWorks functions.

Target Symbol and Path Completion

Start to type any target symbol name or any existing directory name and then type **CTRL+D**. The shell automatically completes the command or directory name for you. If there are multiple options, it prints them for you and then reprints your entry. For example, entering an ambiguous request generates the following result:

```
-> /usr/Tor [CTRL+D]
Tornado/ TorClass/
-> /usr/Tor
```

You can add one or more letters and then type **CTRL+D** again until the path or symbol is complete.

Synopsis Printing

Once you have typed the complete function name, typing **CTRL+D** again prints the function synopsis and then reprints the function name ready for your input:

```
-> _taskIdDefault [CTRL+D]
taskIdDefault() - set the default task ID (WindSh)

int taskIdDefault
    (
    int tid     /* user-supplied task ID; if 0, return default */
    )

-> _taskIdDefault
```

If the routine exists on both host and target, the WindSh synopsis is printed. To print the target synopsis of a function add the meta character @ before the function name.

You can extend the synopsis printing function to include your own routines. To do this, follow these steps:

1. Create the files that include the new routines following Wind River Coding Conventions. (See *VxWorks Programmer's Guide: Coding Conventions*.)
2. Include these files in your project. (See *Creating, Adding, and Removing Application Files*, p.117.)
3. Add the file names to the **DOC_FILES** macro in your makefile.
4. Go to the top of your project tree and run "make synopsis":

```
-> cd installDir/target/src/projectX
-> make synopsis
```

   This adds a file **projectX** to the **host/resource/synopsis** directory.

HTML Help

Typing any function name, a space, and **CTRL+W** opens a browser and displays the HTML reference page for the function. Be sure to leave a space after the function name.

```
-> i [CTRL+W]
```

or

```
-> @i [CTRL+W]
```

Typing **CTRL+W** without any function name launches the HTML help tool. If a browser is already running, the reference page is displayed in that browser; otherwise a new browser is started.

Typing **CTRL+W** without a typing a space after the function name launches the HTML help tool if the function name is unique. If not, **CTRL+W** acts as **CTRL+D** and returns a list of functions whose names begin with the string you entered.

Data Conversion

The shell prints all integers and characters in both decimal and hexadecimal, and if possible, as a character constant or a symbolic address and offset.

```
-> 68
value = 68 = 0x44 = 'D'
```

```
-> 0xf5de
value = 62942 = 0xf5de = _init + 0x52

-> 's'
value = 115 = 0x73 = 's'
```

Data Calculation

Almost all C operators can be used for data calculation. Use "**(**" and "**)**" to force order of precedence.

```
-> (14 * 9) / 3
value = 42 = 0x2a = '*'

-> (0x1355 << 3) & 0x0f0f
value = 2568 = 0xa08

-> 4.3 * 5
value = 21.5
```

Calculations With Variables

```
-> (j + k) * 3
value = ...

-> *(j + 8 * k)
(…address (j + 8 * k)…): value = …

-> x = (val1 - val2) / val3
new symbol "x" added to symbol table
address = …
value = …

-> f = 1.41 * 2
new symbol "f" added to symbol table
f = (…address of f…): value = 2.82
```

Variable **f** gets an 8-byte floating point value.

```
-> ddd=5.2
new symbol "ddd" added to symbol table.
ddd = 0xba0e2c: value = 5.2

-> eee=10.5
new symbol "eee" added to symbol table.
eee = 0xba0e24: value = 10.5

-> fff=(double)ddd+(double)eee
new symbol "fff" added to symbol table.
fff = 0xba0e1c: value = 15.7
->
```

WindSh Environment Variables

WindSh allows you to change the behavior of a particular shell session by setting several environment variables. The Tcl procedure **shConfig** allows you to display and set how I/O redirection, C++ constructors and destructors, loading, and the load path are defined and handled by the shell.

Table 7-1 **WindSh Environment Variables**

Variable	Result
SH_GET_TASK_IO	Sets the I/O redirection mode for called functions. The default is "on", which redirects input and output of called functions to WindSh. To have input and output of called functions appear in the target console, set **SH_GET_TASK_IO** to "off."
LD_CALL_XTORS	Sets the C++ strategy related to constructors and destructors. The default is "target", which causes WindSh to use the value set on the target using **cplusXtorSet( )**. If **LD_CALL_XTORS** is set to "on", the C++ strategy is set to automatic (for the current WindSh only). "Off" sets the C++ strategy to manual for the current shell.
LD_SEND_MODULES	Sets the load mode. The default "on" causes modules to be transferred to the target server. This means that any module WindSh can see can be loaded. If **LD_SEND_MODULES** if "off", the target server must be able to see the module to load it.
LD_PATH	Sets the search path for modules using the separator ";". When a **ld( )** command is issued, WindSh first searches the current directory and loads the module if it finds it. If not, WindSh searches the directory path for the module.
LD_COMMON_MATCH_ALL	Sets the loader behavior for common symbols. If it is set to **on**, the loader tries to match a common symbol with an existing one. If a symbol with the same name is already defined, the loader take its address. Otherwise, the loader creates a new entry. If set to **off**, the loader does not try to find an existing symbol. It creates an entry for each common symbol.

Table 7-1 **WindSh Environment Variables** *(Continued)*

Variable	Result
DSM_HEX_MOD	Sets the disassembling "symbolic + offset" mode. When set to "off" the "symbolic + offset" address representation is turned on and addresses inside the disassembled instructions are given in terms of "symbol name + offset." When set to "on" these addresses are given in hexadecimal.

Because **shConfig** is a Tcl procedure, use the ? to move from the C interpreter to the Tcl interpreter. (See *7.7.2 Tcl: Calling Under C Control*, p.299.)

Example 7-1 **Using shConfig to Modify WindSh Behavior**

```
-> ?shConfig
SH_GET_TASK_IO = on
LD_CALL_XTORS = target
LD_SEND_MODULES = on
LD_PATH = C:/ProjectX/lib/objR4650gnutest/;C:/ProjectY/lib/objR4560gnuvx
-> ?shConfig LD_CALL_XTORS on
-> ?shConfig LD_CALL_XTORS
LD_CALL_XTORS = on
```

7.2.4 Invoking Built-In Shell Routines

Some of the commands (or routines) that you can execute from the shell are built into the host shell, rather than running as function calls on the target. These facilities parallel interactive utilities that can be linked into VxWorks itself. By using the host commands, you minimize the impact on both target memory and performance.

The following sections give summaries of the Tornado WindSh commands. For more detailed reference information, see the **windsh** reference entry (either online, or in the online *Tornado API Reference*).

WARNING: Most of the shell commands correspond to similar routines that can be linked into VxWorks for use with the target-resident version of the shell (*VxWorks Programmer's Guide: Target Shell*). However, the target-resident routines differ in some details.For reference information on a shell command, be sure to consult the **windsh** entry in the online *Tornado API Reference* or use the HTML help for the command. Although there is usually an entry with the same name in the *VxWorks API Reference*, it describes a related target routine, not the shell command.

Task Management

Table 7-2 summarizes the WindSh commands that manage VxWorks tasks.

Table 7-2 **WindSh Commands for Task Management**

Call	Description
sp( )	Spawn a task with default parameters.
sps( )	Spawn a task, but leave it suspended.
tr( )	Resume a suspended task.
ts( )	Suspend a task.
td( )	Delete a task.
period( )	Spawn a task to call a function periodically.
repeat( )	Spawn a task to call a function repeatedly.
taskIdDefault( )	Set or report the default (current) task ID (see *7.3.5 The "Current" Task and Address*, p.279 for a discussion of how the current task is established and used).

The **repeat( )** and **period( )** commands spawn tasks whose entry points are **_repeatHost** and **_periodHost**. The shell downloads these support routines when you call **repeat( )** or **period( )**. (With remote target servers, that download sometimes fails; for a discussion of when this is possible, and what you can do about it, see *7.6 Object Module Load Path*, p.295.) These tasks may be controlled like any other tasks on the target; for example, you can suspend or delete them with **ts( )** or **td( )** respectively.

Task Information

Table 7-3 summarizes the WindSh commands that report task information.

The **i( )** command is commonly used to get a quick report on target activity. (To see this information periodically, use the Tornado browser; see *8. Browser*). If nothing seems to be happening, **i( )** is often a good place to start investigating.

Table 7-3 **WindSh Commands for Task Information**

Call	Description
checkStack( )	Show a stack usage summary for a task, or for all tasks if no task is specified. The summary includes the total stack size (SIZE), the current number of stack bytes (CUR), the maximum number of stack bytes used (HIGH), and the number of bytes never used at the top of the stack (MARGIN = SIZE - HIGH). Use this routine to determine how much stack space to allocate, and to detect stack overflow. **checkStack( )** does not work for tasks that use the **VX_NO_STACK_FILL** option.
i( )	Display system information. This command gives a snapshot of what tasks are in the system, and some information about each of them, such as state, PC, SP, and TCB address. To save memory, this command queries the target repeatedly; thus, it may occasionally give an inconsistent snapshot.
iStrict( )	Display the same information as **i( )**, but query target system information only once. At the expense of consuming more intermediate memory, this guarantees an accurate snapshot.
ti( )	Display task information. This command gives all the information contained in a task's TCB. This includes everything shown for that task by an **i( )** command, plus all the task's registers, and the links in the TCB chain. If *task* is 0 (or the argument is omitted), the current task is reported on.
w( )	Print a summary of each task's pending information, task by task. This routine calls **taskWaitShow( )** in quiet mode on all tasks in the system, or a specified task if the argument is given.
tw( )	Print information about the object the given task is pending on. This routine calls **taskWaitShow( )** on the given task in verbose mode.
tt( )	Display a stack trace.
taskIdFigure( )	Report a task ID, given its name.

To display summary information about all running tasks:

```
-> i
  NAME        ENTRY       TID    PRI   STATUS    PC       SP      ERRNO  DELAY
--------- ----------- -------- --- -------- -------- -------- ------- -----
tExcTask  _excTask       3ad290   0 PEND       4df10   3ad0c0       0     0
tLogTask  _logTask       3aa918   0 PEND       4df10   3aa748       0     0
tWdbTask  0x41288        3870f0   3 READY      23ff4   386d78  3d0004     0
tNetTask  _netTask       3a59c0  50 READY      24200   3a5730       0     0
tFtpdTask _ftpdTask      3a2c18  55 PEND       23b28   3a2938       0     0
value = 0 = 0x0
```

The **w( )** and **tw( )** commands allow you to see what object a VxWorks task is pending on. **w( )** displays summary information for all tasks, while **tw( )** displays object information for a specific task. Note that the **OBJ_NAME** field is used only for objects that have a symbolic name associated with the address of their structure.

```
-> w

NAME        ENTRY      TID        STATUS    DELAY OBJ_TYPE   OBJ_ID   OBJ_NAME
---------- ---------- -------- --------- ----- ---------- -------- --------
tExcTask   _excTask   3d9e3c   PEND          0 MSG_Q(R)   3d9ff4   N/A
tLogTask   _logTask   3d7510   PEND          0 MSG_Q(R)   3d76c8   N/A
tWdbTask   _wdbCmdLoo 36dde4   READY         0                0
tNetTask   _netTask   3a43d0   READY         0                0
u0         _smtask1   36cc2c   PEND          0 MSG_Q_S(S) 370b61   N/A
u1         _smtask3   367c54   PEND          0 MSG_Q_S(S) 370b61   N/A
u3         _taskB     362c7c   PEND          0 SEM_B       8d378   _mySem2
u6         _smtask1   35dca4   PEND          0 MSG_Q_S(S) 370ae1   N/A
u9         _task3B    358ccc   PEND          0 MSG_Q(S)    8cf1c   _myMsgQ
value = 0 = 0x0
->
-> tw u1

NAME        ENTRY      TID        STATUS    DELAY OBJ_TYPE   OBJ_ID   OBJ_NAME
---------- ---------- -------- --------- ----- ---------- -------- --------
u1         _smtask3   367c54   PEND          0 MSG_Q_S(S) 370b61   N/A

Message Queue Id   : 0x370b61
Task Queueing      : SHARED_FIFO
Message Byte Len   : 100
Messages Max       : 0
Messages Queued    : 0
Senders Blocked    : 2
Send Timeouts      : 0
Receive Timeouts   : 0

Senders Blocked:
TID        CPU Number Shared TCB
---------- ---------- ----------
0x36cc2c       0      0x36e464
0x367c54       0      0x36e47c
```

```
value = 0 = 0x0
->
```

System Information

Table 7-4 shows the WindSh commands that display information from the symbol table, from the target system, and from the shell itself.

Table 7-4 **WindSh Commands for System Information**

Call	Description
devs( )	List all devices known on the target system.
lkup( )	List symbols from symbol table.
lkAddr( )	List symbols whose values are near a specified value.
d( )	Display target memory. You can specify a starting address, size of memory units, and number of units to display.
l( )	Disassemble and display a specified number of instructions.
printErrno( )	Describe the most recent error status value.
version( )	Print VxWorks version information.
cd( )	Change the host working directory (no effect on target).
ls( )	List files in host working directory.
pwd( )	Display the current host working directory.
help( )	Display a summary of selected shell commands.
h( )	Display up to 20 lines of command history.
shellHistory( )	Set or display shell history.
shellPromptSet( )	Change the C-interpreter shell prompt.
printLogo( )	Display the Tornado shell logo.

The **lkup( )** command takes a regular expression as its argument, and looks up all symbols containing strings that match. In the simplest case, you can specify a substring to see any symbols containing that string. For example, to display a list

containing routines and declared variables with names containing the string *dsm*, do the following:

```
-> lkup "dsm"
_dsmData                  0x00049d08 text     (vxWorks)
_dsmNbytes                0x00049d76 text     (vxWorks)
_dsmInst                  0x00049d28 text     (vxWorks)
mydsm                    0x003c6510 bss     (vxWorks)
```

Case is significant, but position is not (**mydsm** is shown, but **myDsm** would not be). To explicitly write a search that would match either **mydsm** or **myDsm**, you could write the following:

```
-> lkup "[dD]sm"
```

Regular-expression searches of the symbol table can be as simple or elaborate as required. For example, the following simple regular expression displays the names of three internal VxWorks semaphore functions:

```
-> lkup "sem.Take"
_semBTake                 0x0002aeec text     (vxWorks)
_semCTake                 0x0002b268 text     (vxWorks)
_semMTake                 0x0002bc48 text     (vxWorks)
value = 0 = 0x0
```

Another information command is a symbolic disassembler, **l( )**. The command syntax is:

l [*adr*[**,** *n*]]

This command lists *n* disassembled instructions, starting at *adr*. If *n* is 0 or not given, the *n* from a previous **l( )** or the default value (10) is used. If *adr* is 0, **l( )** starts from where the previous **l( )** stopped, or from where an exception occurred (if there was an exception trap or a breakpoint since the last **l( )** command).

The disassembler uses any symbols that are in the symbol table. If an instruction whose address corresponds to a symbol is disassembled (the beginning of a routine, for instance), the symbol is shown as a label in the address field. Symbols are also used in the operand field. The following is an example of disassembled code for an MC680*x*0 target:

```
-> l printf
_printf

    00033bce  4856                    PEA         (A6)
    00033bd0  2c4f                    MOVEA .L    A7,A6
    00033bd2  4878 0001               PEA         0x1
    00033bd6  4879 0003 460e          PEA         _fioFormatV + 0x780
    00033bdc  486e 000c               PEA         (0xc,A6)
    00033be0  2f2e 0008               MOVE  .L    (0x8,A6),-(A7)
```

```
00033be4  6100 02a8                BSR         _fioFormatV
00033be8  4e5e                     UNLK         A6
00033bea  4e75                   RTS
```

This example shows the **printf( )** routine. The routine does a **LINK**, then pushes the value of **std_out** onto the stack and calls the routine **fioFormatV( )**. Notice that symbols defined in C (routine and variable names) are prefixed with an underbar (_) by the compiler.

Perhaps the most frequently used system information command is **d( )**, which displays a block of memory starting at the address which is passed to it as a parameter. As with any other routine that requires an address, the starting address can be a number, the name of a variable or routine, or the result of an expression.

Several examples of variations on **d( )** appear below.

Display starting at address 1000 decimal:

```
-> d (1000)
```

Display starting at 1000 hex:

```
-> d 0x1000
```

Display starting at the address contained in the variable **dog**:

```
-> d dog
```

The above is different from a display starting at the address of **dog**. For example, if **dog** is a variable at location 0x1234, and that memory location contains the value 10000, **d( )** displays starting at 10000 in the previous example and at 0x1234 in the following:

```
-> d &dog
```

Display starting at an offset from the value of **dog**:

```
-> d dog + 100
```

Display starting at the result of a function call:

```
-> d func (dog)
```

Display the code of **func( )** as a simple hex memory dump:

```
-> d func
```

When you use **cd( )** in the host shell, you are changing the working directory on the host. It does not change the directory on the target. WindSh has no knowledge

of the target file system. Thus if you mount a drive on the target from the host shell and try to **cd( )** to it, you see the following:

```
-> cd "/ata0/"
couldn't change working directory to "\ata0": no such file or directory
value = -1 = 0xffffffff
```

However, the result is different if you execute **cd( )** and **ls( )** on the target by prefixing the commands with **@**:

```
-> @cd "/ata0/"
value = 0 = 0x0

-> @ls
IO.SYS
MSDOS.SYS
DRVSPACE.BIN
```

@cd and **@ls** only work if you have the component **INCLUDE_DISK_UTIL** included in your target image.

The above also applies if you wish to use the target server file system (TSFS) from WindSh: **cd "/tgtsvr"** does not work but **@cd "/tgtsvr"** does. To use TSFS you must have the TSFS component, **INCLUDE_WDB_TSFS**, installed in VxWorks and start the target server with the **"-R** *dirName***"** or **"-R** *dirName* **-RW"** option.

System Modification and Debugging

Developers often need to change the state of the target, whether to run a new version of some software module, to patch memory, or simply to single-step a program. Table 7-5 summarizes the WindSh commands of this type.

Table 7-5 **WindSh Commands for System Modification and Debugging**

Call	Description
ld( )	Load an object module into target memory and link it dynamically into the run-time.
unld( )	Remove a dynamically-linked object module from target memory, and free the storage it occupied.
m( )	Modify memory in *width* (byte, short, or long) starting at *adr*. The **m( )** command displays successive words in memory on the terminal; you can change each word by typing a new hex value, leave the word unchanged and continue by typing **ENTER**, or return to the shell by typing a dot (**.**).

Table 7-5 **WindSh Commands for System Modification and Debugging** *(Continued)*

Call	Description
mRegs( )	Modify register values for a particular task.
b( )	Set or display breakpoints, in a specified task or in all tasks.
bh( )	Set a hardware breakpoint.
s( )	Step a program to the next instruction.
so( )	Single-step, but step over a subroutine.
c( )	Continue from a breakpoint.
cret( )	Continue until the current subroutine returns.
bdall( )	Delete all breakpoints.
bd( )	Delete a breakpoint.
reboot( )	Return target control to the target boot ROMs, then reset the target server and reattach the shell.
bootChange( )	Modify the saved values of boot parameters (see *2.6.4 Description of Boot Parameters*, p.50).
sysSuspend( )	If supported by the target-agent configuration, enter system mode. See *7.2.7 Using the Shell for System Mode Debugging*, p.267.
sysResume( )	If supported by the target agent (and if system mode is in effect), return to task mode from system mode.
agentModeShow( )	Show the agent mode (*system* or *task*).
sysStatusShow( )	Show the system context status (*suspended* or *running).*
quit( ) or **exit( )**	Dismiss the shell.

One of the most useful shell features for interactive development is the dynamic linker. With the shell command **ld( )**, you can download and link new portions of the application. Because the linking is dynamic, you only have to rebuild the particular piece you are working on, not the entire application. Download can be cancelled with **CTRL+C** or by clicking Cancel in the load progress indicator window. The dynamic linker is discussed further in *5.4.4 Linking the System Modules*, p.201.

The **m( )** command provides an interactive way of manipulating target memory.

The remaining commands in this group are for breakpoints and single-stepping. You can set a breakpoint at any instruction. When that instruction is executed by an eligible task (as specified with the **b( )** command), the task that was executing on the target suspends, and a message appears at the shell. At this point, you can examine the task's registers, do a task trace, and so on. The task can then be deleted, continued, or single-stepped.

If a routine called from the shell encounters a breakpoint, it suspends just as any other routine would, but in order to allow you to regain control of the shell, such suspended routines are treated in the shell as though they had returned 0. The suspended routine is nevertheless available for your inspection.

When you use **s( )** to single-step a task, the task executes one machine instruction, then suspends again. The shell display shows all the task registers and the *next* instruction to be executed by the task.

You can use the **bh( )** command to set hardware breakpoints at any instruction or data element. Instruction hardware breakpoints can be useful to debug code running in ROM or Flash EPROM. Data hardware breakpoints (watchpoints) are useful if you want to stop when your program accesses a specific address. Hardware breakpoints are available on some BSPs; see your BSP documentation to determine if they are supported for your BSP. The arguments of the **bh( )** command are architecture specific. For more information, run the **help( )** command. The number of hardware breakpoints you can set is limited by the hardware; if you exceed the maximum number, you will receive an error.

C++ Development

Certain WindSh commands are intended specifically for work with C++ applications. Table 7-6 summarizes these commands. For more discussion of these shell commands, see *VxWorks Programmer's Guide: C++ Development*.

Table 7-6 **WindSh Commands for C++ Development**

Call	Description
cplusCtors( )	Call static constructors manually.
cplusDtors( )	Call static destructors manually.
cplusStratShow( )	Report on whether current constructor/destructor strategy is manual or automatic.
cplusXtorSet( )	Set constructor/destructor strategy.

In addition, you can use the Tcl routine **shConfig** to set the environment variable **LD_CALL_XTORS** within a particular shell. This allows you to use a different C++ strategy in a shell than is used on the target. For more information on **shConfig**, see *WindSh Environment Variables*, p.252.

Object Display

Table 7-7 summarizes the WindSh commands that display VxWorks objects. The browser provides displays that are analogous to the output of many of these routines, except that browser windows can update their contents periodically; see *8. Browser*.

Table 7-7 **WindSh Commands for Object Display**

Call	Description
show( )	Print information on a specified object in the shell window.
browse( )	Display a specified object in the Tornado browser.
classShow( )	Show information about a class of VxWorks kernel objects. List available classes with: `-> lkup "ClassId"`
taskShow( )	Display information from a task's TCB.
taskCreateHookShow( )	Show the list of task create routines.
taskDeleteHookShow( )	Show the list of task delete routines.
taskRegsShow( )	Display the contents of a task's registers.
taskSwitchHookShow( )	Show the list of task switch routines.
taskWaitShow( )	Show information about the object a task is pended on. Note that **taskWaitShow( )** can not give object IDs for POSIX semaphores or message queues.
semShow( )	Show information about a semaphore.
semPxShow( )	Show information about a POSIX semaphore.
wdShow( )	Show information about a watchdog timer.
msgQShow( )	Show information about a message queue.

Table 7-7 **WindSh Commands for Object Display** *(Continued)*

Call	Description
mqPxShow( )	Show information about a POSIX message queue.
iosDrvShow( )	Display a list of system drivers.
iosDevShow( )	Display the list of devices in the system.
iosFdShow( )	Display a list of file descriptor names in the system.
memPartShow( )	Show partition blocks and statistics.
memShow( )	Display the total amount of free and allocated space in the system partition, the number of free and allocated fragments, the average free and allocated fragment sizes, and the maximum free fragment size. Show current as well as cumulative values. With an argument of 1, also display the free list of the system partition.
smMemShow( )	Display the amount of free space and statistics on memory-block allocation for the shared-memory system partition.
smMemPartShow( )	Display the amount of free space and statistics on memory-block allocation for a specified shared-memory partition.
moduleShow( )	Show the current status for all the loaded modules.
moduleIdFigure( )	Report a loaded module's module ID, given its name.
intVecShow( )	Display the interrupt vector table. This routine displays information about the given vector or the whole interrupt vector table if *vector* is equal to -1. Note that **intVecShow( )** is not supported on architectures such as ARM and PowerPC that do not use interrupt vectors.

Network Status Display

Table 7-8 summarizes the WindSh commands that display information about the VxWorks network.

In order for a protocol-specific command to work, the appropriate protocol must be included in your VxWorks configuration.

Table 7-8 **WindSh Commands for Network Status Display**

Call	Description
hostShow( )	Display the host table.
icmpstatShow( )	Display statistics for ICMP (Internet Control Message Protocol).
ifShow( )	Display the attached network interfaces.
inetstatShow( )	Display all active connections for Internet protocol sockets.
ipstatShow( )	Display IP statistics.
routestatShow( )	Display routing statistics.
tcpstatShow( )	Display all statistics for the TCP protocol.
tftpInfoShow( )	Get TFTP status information.
udpstatShow( )	Display statistics for the UDP protocol.

Resolving Name Conflicts between Host and Target

If you invoke a name that stands for a host shell command, the shell always invokes that command, even if there is also a target routine with the same name. Thus, for example, **i( )** always runs on the host, regardless of whether you have the VxWorks routine of the same name linked into your target.

However, you may occasionally need to call a target routine that has the same name as a host shell command. The shell supports a convention allowing you to make this choice: use the single-character prefix **@** to identify the target version of any routine. For example, to run a target routine named **i( )**, invoke it with the name **@i( )**.

7.2.5 Running Target Routines from the Shell

All target routines are available from WindSh. This includes both VxWorks routines and your application routines. Thus the shell provides a powerful tool for testing and debugging your applications using all the host resources while having minimal impact on how the target performs and how the application behaves.

Invocations of VxWorks Subroutines

```
-> taskSpawn ("tmyTask", 10, 0, 1000, myTask, fd1, 300)
value = ...

-> fd = open ("file", 0, 0)
new symbol "fd" added to symbol table
fd = (...address of fd...): value = ...
```

Invocations of Application Subroutines

```
-> testFunc (123)
value = ...

-> myValue = myFunc (1, &val, testFunc (123))
myValue = (...address of myValue...): value = ...

-> myDouble = (double ()) myFuncWhichReturnsADouble (x)
myDouble = (...address of myDouble...): value = ...
```

For situations where the result of a routine is something other than a 4-byte integer, see *Function Calls*, p.277.

7.2.6 Rebooting from the Shell

In an interactive real-time development session, it is sometimes convenient to restart everything to make sure the target is in a known state. WindSh provides the **reboot( )** command or **CTRL+SHIFT+X** to make this easy.

When you execute **reboot( )** or type **CTRL+SHIFT+X**, the following reboot sequence occurs:

1. The shell displays a message to confirm rebooting has begun:

   ```
   -> reboot
   Rebooting...
   ```

2. The target reboots.

3. The original target server on the host detects the target reboot and restarts itself, with the same configuration as previously. The target-server configuration options **-Bt** (timeout) and **-Br** (retries) govern how long the new server waits for the target to reboot, and how many times the new server attempts to reconnect; see the **tgtsvr** reference entry in the online *Tornado API Reference*, or in the HTML help.

4. The shell detects the target-server restart and begins an automatic-restart sequence (initiated any time it loses contact with the target server for any reason), indicated with the following messages:

```
Target connection has been lost.  Restarting shell...
Waiting to attach to target server......
```

5. When WindSh establishes contact with the new target server, it displays the Tornado shell logo and awaits your input.

NOTE: If the target server timeout (**-Bt**) and retry count (**-Br**) are too low for your target and your connection method, the new target server may abandon execution before the target finishes rebooting. The default timeout is one second, and the default retry count is three; thus, by default the target server waits three seconds for the target to reboot. If the shell does not restart in a reasonably short time after a **reboot( )**, try starting a new target server manually.

7.2.7 Using the Shell for System Mode Debugging

The bulk of this chapter discusses the shell in its most frequent style of use: attached to a normally running VxWorks system, through a target agent running in task mode. You can also use the shell with a system-mode agent. Entering system mode stops the entire target system: all tasks, the kernel, and all ISRs. Similarly, breakpoints affect all tasks.

CAUTION: When you use system mode debugging, you cannot execute expressions that call target-resident routines. You must use **sp( )** to spawn a task with the target-resident routine as the entry point argument. A newly-spawned task will not execute until you allow the kernel to run long enough to schedule that task.

Depending on how the target agent is configured, you may be able to switch between system mode and task mode; see *4.7 Configuring the Target-Host Communication Interface*, p.156. When the agent supports mode switching, the following WindSh commands control system mode:

sysSuspend( )
Enter system mode and stop the target system.

sysResume( )
Return to task mode and resume execution of the target system.

The following commands are to determine the state of the system and the agent:

agentModeShow( )
Show the agent mode (*system* or *task*).

sysStatusShow( )
Show the system context status (*suspended* or *running*).

The following shell commands behave differently in system mode:

b( )
Set a system-wide breakpoint; the system stops when this breakpoint is encountered by any task, or the kernel, or an ISR.

c( )
Resume execution of the entire system (but remain in system mode).

WARNING: If you are running either CrossWind or Look! you must not use **c( )** from the shell to continue; instead continue from the debugger itself. Using **c( )** from the shell when the debugger is running will confuse the debugger.

i( )
Display the state of the system context and the mode of the agent.

s( )
Single-step the entire system.

sp( )
Add a task to the execution queue. The task does not begin to execute until you continue the kernel or step through the task scheduler.

The following example shows how to use system mode debugging to debug a system interrupt.

Example 7-2 **System-Mode Debugging**

In this case, **usrClock( )** is attached to the system clock interrupt handler which is called at each system clock tick when VxWorks is running. First suspend the system and confirm that it is suspended using either **i( )** or **sysStatusShow( )**.

```
-> sysSuspend
value = 0 = 0x0
->
-> i
NAME      ENTRY      TID      PRI   STATUS  PC      SP      ERRNO  DELAY
--------- ---------- -------- ----- ------- ------- ------- -----  -----
tExcTask  _excTask   3e8f98   0     PEND    47982   3e8ef4  0      0
tLogTask  _logTask   3e6670   0     PEND    47982   3e65c8  0      0
tWdbTask  0x3f024    398e04   3     PEND    405ac   398d50  30067  0
```

```
tNetTask  _netTask   3b39e0   50    PEND    405ac   3b3988  0       0

Agent mode     : Extern
System context : Suspended
value = 0 = 0x0
->
-> sysStatusShow
System context is suspended
value = 0 = 0x0
```

Next, set the system mode breakpoint on the entry point of the interrupt handler you want to debug. Since the target agent is running in system mode, the breakpoint will automatically be a system mode breakpoint, which you can confirm with the **b( )** command. Resume the system using **c( )** and wait for it to enter the interrupt handler and hit the breakpoint.


```
-> b usrClock
value = 0 = 0x0
-> b
0x00022d9a: _usrClock          Task:     SYSTEM Count:  0
value = 0 = 0x0
-> c
value = 0 = 0x0
->
Break at 0x00022d9a: _usrClock               Task: SYSTEM
```

You can now debug the interrupt handler. For example, you can determine which task was running when system mode was entered using **taskIdCurrent( )** and **i( )**.

```
-> taskIdCurrent
_taskIdCurrent = 0x838d0: value = 3880092 = 0x3b349c
-> i
NAME      ENTRY      TID      PRI   STATUS  PC      SP      ERRNO  DELAY
--------- ---------- -------- ----- ------- ------- ------- -----  -----
tExcTask  _excTask   3e8a54   0     PEND    4eb8c   3e89b4  0      0
tLogTask  _logTask   3e612c   0     PEND    4eb8c   3e6088  0      0
tWdbTask  0x44d54    389774   3     PEND    46cb6   3896c0  0      0
tNetTask  _netTask   3b349c   50    READY   46cb6   3b3444  0      0

Agent mode     : Extern
System context : Suspended
value = 0 = 0x0
```

You can trace all the tasks except the one that was running when you placed the system in system mode and you can step through the interrupt handler.

```
-> tt tLogTask
4da78   _vxTaskEntry   +10 : _logTask (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
3f2bc   _logTask       +18 : _msgQReceive (3e62e4, 3e60dc, 20, ffffffff)
27e64   _msgQReceive   +1ba: _qJobGet ([3e62e8, ffffffff, 0, 0, 0, 0])
value = 0 = 0x0
-> l
_usrClock
```

```
00022d9a  4856                    PEA         (A6)
00022d9c  2c4f                    MOVEA .L    A7,A6
00022d9e  61ff 0002 3d8c          BSR         _tickAnnounce
00022da4  4e5e                    UNLK        A6
00022da6  4e75                    RTS
00022da8  352e 3400               MOVE  .W    (0x3400,A6),-(A2)
00022dac  4a75 6c20               TST   .W    (0x20,A5,D6.L*4)
00022db0  3234 2031               MOVE  .W    (0x31,A4,D2.W*1),D1
00022db4  3939 382c 2031          MOVE  .W    0x382c2031,-(A4)
00022dba  343a 3337               MOVE  .W    (0x3337,PC),D2
value = 0 = 0x0
-> s
d0   =      3e   d1   =     3700   d2     =     3000   d3     =   3b09dc
d4   =       0   d5   =        0   d6     =        0   d7     =        0
a0   =   230b8   a1   =   3b3318   a2     =   3b3324   a3     =    7e094
a4   =  38a7c0   a5   =        0   a6/fp  =    bcb90   a7/sp  =    bcb84
sr   =    2604   pc   =    230ba
    000230ba  2c4f               MOVEA .L    A7,A6
value = 0 = 0x0
```

Return to task mode and confirm that return by calling **i( )**:

```
-> sysResume
value = 0 = 0x0

-> i
NAME      ENTRY      TID      PRI   STATUS  PC      SP      ERRNO  DELAY
--------- ---------- -------- ----- ------- ------- ------- -----  -----
tExcTask  _excTask   3e8f98   0     PEND    47982   3e8ef4  0      0
tLogTask  _logTask   3e6670   0     PEND    47982   3e65c8  0      0
tWdbTask  0x3f024    398e04   3     READY   405ac   398d50  30067  0
tNetTask  _netTask   3b39e0   50    PEND    405ac   3b3988  0      0
value = 0 = 0x0
```

If you want to debug an application you have loaded dynamically, set an appropriate breakpoint and spawn a task which runs when you continue the system:

```
-> sysSuspend
value = 0 = 0x0
-> ld < test.o
Loading /view/didier.temp/vobs/wpwr/target/lib/objMC68040gnutest//test.o /
value = 400496 = 0x61c70 = _rn_addroute + 0x1d4
-> b address
value = 0 = 0x0
-> sp test
value = 0 = 0x0
-> c
```

The application breaks on *address* when the instruction at *address* is executed.

7.2.8 Interrupting a Shell Command

Occasionally it is desirable to abort the shell's evaluation of a statement. For example, an invoked routine may loop excessively, suspend, or wait on a semaphore. This may happen as the result of errors in arguments specified in the invocation, errors in the implementation of the routine itself, or simply oversight as to the consequences of calling the routine.

To regain control of the shell in such cases, press the interrupt character on the keyboard, usually **CTRL+C**.[2] This makes the shell stop waiting for a result and allows input of a new statement. Any remaining portions of the statement are discarded and the task that ran the function call is deleted.

CAUTION: CTRL+C does not interrupt non-blocking functions. If the task transmitting the break request is of lower priority than the task to be interrupted, the request is not conveyed until the original task completes.

Pressing **CTRL+C** is also necessary to regain control of the shell after calling a routine on the target that ends with **exit( )** rather than **return**.

Occasionally a subroutine invoked from the shell may incur a fatal error, such as a bus/address error or a privilege violation. When this happens, the failing routine is suspended. If the fatal error involved a hardware exception, the shell automatically notifies you of the exception. For example:

```
-> taskSpawn -4
Exception number 11: Task: 0x264ed8 (tCallTask)
```

In cases like this, you do not need to type **CTRL+C** to recover control of the shell; it automatically returns to the prompt, just as if you had interrupted. Whether you interrupt or the shell does it for you, you can proceed to investigate the cause of the suspension. For example, in the case above you could run the Tornado browser on **tCallTask**.

An interrupted routine may have left things in a state which was not cleared when you interrupted it. For instance, a routine may have taken a semaphore, which cannot be given automatically. Be sure to perform manual cleanup if you are going to continue the application from this point.

2. The interrupt character matches whatever you normally use in UNIX shells as an interrupt; you can set it with the UNIX command **stty intr**. See your host system documentation for details.

7.3 The Shell C-Expression Interpreter

The C-expression interpreter is the most common command interface to the Tornado shell. This interpreter can evaluate almost any C expression interactively in the context of the attached target. This includes the ability to use variables and functions whose names are defined in the symbol table. Any command you type is interpreted as a C expression. The shell evaluates that expression and, if the expression so specifies, assigns the result to a variable.

7.3.1 I/O Redirection

Developers often call routines that display data on standard output or accept data from standard input. By default the standard output and input streams are directed to the same window as the Tornado shell. For example, in a default configuration of Tornado, invoking **printf( )** from the shell window gives the following display:

```
-> printf("Hello World\n")
Hello World!
value = 13 = 0xd
->
```

This behavior can be dynamically modified using the Tcl procedure **shConfig** as follows:

```
-> ?shConfig SH_GET_TASK_IO off
->
-> printf("Hello World!\n")
value = 13 = 0xd
->
```

The shell reports the **printf( )** result, indicating that 13 characters have been printed. The output, however, goes to the target's standard output, not to the shell.

To determine the current configuration, use **shConfig**. If you issue the command without an argument, all parameters are listed. Use an argument to list only one parameter.

```
-> ?shConfig SH_GET_TASK_IO
SH_GET_TASK_IO = off
```

For more information, see on **shConfig**, see *WindSh Environment Variables*, p.252.

The standard input and output are only redirected for the function called from WindSh. If this function spawns other tasks, the input and output of the spawned

tasks are not redirected to WindSh. To have all I/O redirected to WindSh, you can start the target server with the options **-C -redirectShell**.

7.3.2 Data Types

The most significant difference between the shell C-expression interpreter and a C compiler lies in the way that they handle data types. The shell does not accept any C declaration statements, and no data-type information is available in the symbol table. Instead, an expression's type is determined by the types of its terms.

Unless you use explicit type-casting, the shell makes the following assumptions about data types:

- In an assignment statement, the type of the left hand side is determined by the type of the right hand side.
- If floating-point numbers and integers both appear in an arithmetic expression, the resulting type is a floating-point number.
- Data types are assigned to various elements as shown in Table 7-9.

Table 7-9 **Shell C Interpreter Data-Type Assumptions**

Element	Data Type
variable	**int**
variable used as floating-point	**double**
return value of subroutine	**int**
constant with no decimal point	**int/long**
constant with decimal point	**double**

A constant or variable can be treated as a different type than what the shell assumes by explicitly specifying the type with the syntax of C type-casting. Functions that return values other than integers require a slightly different type-casting; see *Function Calls*, p.277. Table 7-10 shows the various data types available in the shell C interpreter, with examples of how they can be set and referenced.

Table 7-10 **Data Types in the Shell C Interpreter**

Type	Bytes	Set Variable	Display Variable
int	4	`x = 99`	`x` `(int) x`
long	4	`x = 33` `x = (long)33`	`x` `(long) x`
short	2	`x = (short)20`	`(short) x`

char	1	`x = 'A'`
		`x = (char)65`
		`x = (char)0x41`
			`(char) x`

double	8	`x = 11.2` `x = (double)11.2`	`(double) x`
float	4	`x = (float)5.42`	`(float) x`

Strings, or character arrays, are not treated as separate types in the shell C interpreter. To declare a string, set a variable to a string value.[3] For example:

```
-> ss = "shoe bee doo"
```

The variable **ss** is a pointer to the string *shoe bee doo*. To display **ss**, enter:

```
-> d ss
```

The **d( )** command displays memory where **ss** is pointing.[4] You can also use **printf( )** to display strings.

The shell places no type restrictions on the application of operators. For example, the shell expression:

```
*(70000 + 3 * 16)
```

evaluates to the 4-byte integer value at memory location 70048.

3. Memory allocated for string constants is never freed by the shell. See *7.3.8 Strings*, p.281 for more information.
4. **d( )** is one of the WindSh commands, implemented in Tcl and executing on the host.

7.3.3 Lines and Statements

The shell parses and evaluates its input one line at a time. A line may consist of a single shell statement or several shell statements separated by semicolons. A semicolon is not required on a line containing only a single statement. A statement cannot continue on multiple lines.

Shell statements are either C expressions or assignment statements. Either kind of shell statement may call WindSh commands or target routines.

7.3.4 Expressions

Shell expressions consist of literals, symbolic data references, function calls, and the usual C operators.

Literals

The shell interprets the literals in Table 7-11 in the same way as the C compiler, with one addition: the shell also allows hex numbers to be preceded by **$** instead of **0x**.

Table 7-11 **Literals in the Shell C Interpreter**

Literal	Example
decimal numbers	**143967**
octal numbers	**017734**
hex numbers	**0xf3ba** or **$f3ba**
floating point numbers	**666.666**
character constants	**'x'** and **'$'**
string constants	**"hello world!!!"**

Variable References

Shell expressions may contain references to variables whose names have been entered in the system symbol table. Unless a particular type is specified with a variable reference, the variable's value in an expression is the 4-byte value at the memory address obtained from the symbol table. It is an error if an identifier in an expression is not found in the symbol table, except in the case of assignment statements discussed below.

Some C compilers prefix user-defined identifiers with an underbar, so that **myVar** is actually in the symbol table as **_myVar**. In this case, the identifier can be entered either way to the shell—the shell searches the symbol table for a match either with or without a prefixed underbar.

You can also access data in memory that does not have a symbolic name in the symbol table, as long as you know its address. To do this, apply the C indirection operator "*****" to a constant. For example, ***0x10000** refers to the 4-byte integer value at memory address 10000 hex.

Operators

The shell interprets the operators in Table 7-12 in the same way as the C compiler.

Table 7-12 **Operators in the Shell C Interpreter**

Operator Type	Operators					
arithmetic	+	-	*	/	unary -	
relational	==	!=	<	>	<=	>=
shift	<<	>>				
logical	||	&&	!			
bitwise	|	&	~	^		
address and indirection	&	*				

The shell assigns the same precedence to the operators as the C compiler. However, unlike the C compiler, the shell always evaluates both sub-expressions of the logical binary operators || and &&.

Function Calls

Shell expressions may contain calls to C functions (or C-compatible functions) whose names have been entered in the system symbol table; they may also contain function calls to WindSh commands that execute on the host.

The shell executes such function calls in tasks spawned for the purpose, with the specified arguments and default task parameters; if the task parameters make a difference, you can call **taskSpawn( )** instead of calling functions from the shell directly. The value of a function call is the 4-byte integer value returned by the function. The shell assumes that all functions return integers. If a function returns a value other than an integer, the shell must know the data type being returned before the function is invoked. This requires a slightly unusual syntax because you must cast the function, not its return value. For example:

```
-> floatVar = ( float ()) funcThatReturnsAFloat (x,y)
```

The shell can pass up to ten arguments to a function. In fact, the shell always passes exactly ten arguments to every function called, passing values of zero for any arguments not specified. This is harmless because the C function-call protocol handles passing of variable numbers of arguments. However, it allows you to omit trailing arguments of value zero from function calls in shell expressions.

Function calls can be nested. That is, a function call can be an argument to another function call. In the following example, **myFunc( )** takes two arguments: the return value from **yourFunc( )** and **myVal**. The shell displays the value of the overall expression, which in this case is the value returned from **myFunc( )**.

```
myFunc (yourFunc (yourVal), myVal);
```

Shell expressions can also contain references to function addresses instead of function invocations. As in C, this is indicated by the absence of parentheses after the function name. Thus the following expression evaluates to the result returned by the function **myFunc2( )** plus 4:

```
4 + myFunc2 ( )
```

However, the following expression evaluates to the address of **myFunc2( )** plus 4:

```
4 + myFunc2
```

An important exception to this occurs when the function name is the very first item encountered in a statement. This is discussed in *Arguments to Commands*, p.278.

Shell expressions can also contain calls to functions that do not have a symbolic name in the symbol table, but whose addresses are known to you. To do this, simply supply the address in place of the function name. Thus the following expression calls a parameterless function whose entry point is at address 10000 hex:

```
0x10000 ()
```

You can assign the address of a function to a variable and then dereference the variable to call the function as in the following example:

```
-> aaa=printf
-> (* aaa)("The clock speed is %d\n" ,sysClckRateGet())
```

Subroutines as Commands

Both VxWorks and the Tornado shell itself provide routines that are meant to be called from the shell interactively. You can think of these routines as *commands*, rather than as *subroutines*, even though they can also be called with the same syntax as C subroutines (and those that run on the target are in fact subroutines). All the commands discussed in this chapter fall in this category. When you see the word *command*, you can read *subroutine*, or vice versa, since their meaning here is identical.

Arguments to Commands

In practice, most statements input to the shell are function calls, often to invoke VxWorks facilities. To simplify this use of the shell, an important exception is allowed to the standard expression syntax required by C. When a function name is the very first item encountered in a shell statement, the parentheses surrounding the function's arguments may be omitted. Thus the following shell statements are synonymous:

```
-> rename ("oldname", "newname")
-> rename "oldname", "newname"
```

The following statements are also synonymous:

```
-> evtBufferAddress ( )
-> evtBufferAddress
```

However, note that if you wish to assign the result to a variable, the function call cannot be the first item in the shell statement—thus, the syntactic exception above does not apply. The following captures the address, not the return value, of **evtBufferAddress( )**:

```
-> value = evtBufferAddress
```


Task References

Most VxWorks routines that take an argument representing a task require a task ID. However, when invoking routines interactively, specifying a task ID can be cumbersome since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, shell expressions can reference a task by either task ID or task name. The shell attempts to resolve a task argument to a task ID as follows: if no match is found in the symbol table for a task argument, the shell searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

By convention, task names are prefixed with a *u* for tasks started from the Tornado shell, and with a *t* for VxWorks tasks started from the target itself. In addition, tasks started from a shell are prefixed by *s1*, *s2*, and so on to indicate which shell they were started from. This avoids name conflicts with entries in the symbol table. The names of system tasks and the default task names assigned when tasks are spawned use this convention. For example, tasks spawned with the shell command **sp( )** in the first shell opened are given names such as **s1u0** and **s1u1**. Tasks spawned with the second shell opened have names such as **s2u0** and **s2u1**. You are urged to adopt a similar convention for tasks named in your applications.

7.3.5 The "Current" Task and Address

A number of commands, such as **c( )**, **s( )**, and **ti( )**, take a task parameter that may be omitted. If omitted, the *current task* is used. The **l( )** and **d( )** commands use the *current address* if no address is specified.

The current task and address are set when:

- A task hits a breakpoint or an exception trap. The current address is the address of the instruction that caused the break or exception.
- A task is single-stepped. The current address is the address of the *next* instruction to be executed.
- Any of the commands that use the current task or address are executed with a specific task parameter. The current address will be the address of the byte *following* the last byte that was displayed or disassembled.

7.3.6 Assignments

The shell C interpreter accepts assignment statements in the following form:

addressExpression **=** *expression*

The left side of an expression must evaluate to an addressable entity; that is, a legal C value.

Typing and Assignment

The data type of the left side is determined by the type of the right side. If the right side does not contain any floating-point constants or noninteger type-casts, then the type of the left side will be an integer. The value of the right side of the assignment is put at the address provided by the left side. For example, the following assignment sets the 4-byte integer variable **x** to 0x1000:

```
-> x = 0x1000
```

The following assignment sets the 4-byte integer value at memory address 0x1000 to the current value of **x**:

```
-> *0x1000 = x
```

The following compound assignment adds 300 to the 4-byte integer variable **x**:

```
-> x += 300
```

The following adds 300 to the 4-byte integer at address 0x1000:

```
-> *0x1000 += 300
```

The compound assignment operator **-=**, as well as the increment and decrement operators **++** and **--**, are also available.

Automatic Creation of New Variables

New variables can be created automatically by assigning a value to an undefined identifier (one not already in the symbol table) with an assignment statement.

When the shell encounters such an assignment, it allocates space for the variable and enters the new identifier in the symbol table along with the address of the newly allocated variable. The new variable is set to the value and type of the right-side expression of the assignment statement. The shell prints a message indicating that a new variable has been allocated and assigned the specified value.

For example, if the identifier **fd** is not currently in the symbol table, the following statement creates a new variable named **fd** and assigns to it the result of the function call:

```
-> fd = open ("file", 0)
```

7.3.7 Comments

The shell allows two kinds of comments. First, comments of the form **/*** … ***/** can be included anywhere on a shell input line. These comments are simply discarded, and the rest of the input line evaluated as usual. Second, any line whose first nonblank character is **#** is ignored completely. Comments are particularly useful for Tornado shell scripts. See *Scripts: Redirecting Shell I/O*, p.288.

7.3.8 Strings

When the shell encounters a string literal ("…") in an expression, it allocates space for the string including the null-byte string terminator. The value of the literal is the address of the string in the newly allocated storage.

For instance, the following expression allocates 12 bytes from the target-agent memory pool, enters the string in those 12 bytes (including the null terminator), and assigns the address of the string to **x**:

```
-> x = "hello there"
```

Even when a string literal is not assigned to a symbol, memory is still permanently allocated for it. For example, the following uses 12 bytes of memory that are never freed:

```
-> printf ("hello there")
```

If strings were only temporarily allocated, and a string literal were passed to a routine being spawned as a task, then by the time the task executed and attempted to access the string, the shell would have already released—possibly even reused—the temporary storage where the string was held.

This memory, like other memory used by the Tornado tools, comes from the target-agent memory pool; it does not reduce the amount of memory available for application execution (the VxWorks memory pool). The amount of target memory allocated for each of the two memory pools is defined at configuration time; see *Scaling the Target Agent*, p.161.

After extended development sessions in Tornado shells, the cumulative memory used for strings may be noticeable. If this is a problem, restart your target server.

7.3.9 Ambiguity of Arrays and Pointers

In a C expression, a nonsubscripted reference to an array has a special meaning, namely the address of the first element of the array. The shell, to be compatible, should use the address obtained from the symbol table as the value of such a reference, rather than the contents of memory at that address. Unfortunately, the information that the identifier is an array, like all data type information, is not available after compilation. For example, if a module contains the following:

```
char string [ ] = "hello";
```

you might be tempted to enter a shell expression like sample 1:

```
-> printf (string)
```

While this would be correct in C, the shell will pass the first 4 bytes of the string itself to **printf( )**, instead of the address of the string. To correct this, the shell expression must explicitly take the address of the identifier as shown in sample 2:

```
-> printf (&string)
```

To make matters worse, in C if the identifier had been declared a character pointer instead of a character array:

```
char *string = "hello";
```

then to a compiler, sample 1 would be correct and sample 2 would be wrong! This is especially confusing since C allows pointers to be subscripted exactly like arrays, so that the value of **string[0]** would be "h" in either of the above declarations.

The moral of the story is that array references and pointer references in shell expressions are different from their C counterparts. In particular, array references require an explicit application of the address operator **&**.

7.3.10 Pointer Arithmetic

While the C language treats pointer arithmetic specially, the shell C interpreter does not, because it treats all non-type-cast variables as 4-byte integers.

In the shell, pointer arithmetic is no different than integer arithmetic. Pointer arithmetic is valid, but it does not take into account the size of the data pointed to. Consider the following example:

```
-> *(myPtr + 4) = 5
```

Assume that the value of **myPtr** is 0x1000. In C, if **myPtr** is a pointer to a type **char**, this would put the value 5 in the byte at address at 0x1004. If **myPtr** is a pointer to a 4-byte integer, the 4-byte value 0x00000005 would go into bytes 0x1010–0x1013. The shell, on the other hand, treats variables as integers, and therefore would put the 4-byte value 0x00000005 in bytes 0x1004–0x1007.

7.3.11 C Interpreter Limitations

Powerful though it is, the C interpreter in the shell is not a complete interpreter for the C language. The following C features are *not* present in the Tornado shell:

- **Control Structures**

  The shell interprets only C *expressions* (and comments). The shell does not support C control structures such as **if**, **goto**, and **switch** statements, or **do**, **while**, and **for** loops. Control structures are rarely needed during shell interaction. If you do come across a situation that requires a control structure, you can use the Tcl interface to the shell instead of using its C interpreter directly; see *7.7 Tcl: Shell Interpretation*, p.297.

- **Compound or Derived Types**

  No compound types (**struct** or **union** types) or derived types (**typedef**) are recognized in the shell C interpreter. You can use CrossWind instead of the shell for interactive debugging, when you need to examine compound or derived types.

- **Macros**

  No C preprocessor macros (or any other preprocessor facilities) are available in the shell. CrossWind does not support preprocessor macros either, but indirect workarounds are available using either the shell or CrossWind. For constant macros, you can define variables in the shell with similar names to the macros. To avoid intrusion into the application symbol table, you can use CrossWind instead; in this case, use CrossWind convenience variables with names corresponding to the desired macros. In either case, you can automate the effort of defining any variables you need repeatedly, by using an initialization script.

For the first two problems (control structures, or display and manipulation of types that are not supported in the shell), you might also consider writing auxiliary subroutines to provide these services during development; you can call such subroutines at will from the shell, and later omit them from your final application.

7.3.12 C-Interpreter Primitives

Table 7-13 lists all the primitives (commands) built into WindSh. Because the shell tries to find a primitive first before attempting to call a target subroutine, it is best to avoid these names in the target code. If you do have a name conflict, however, you can force the shell to call a target routine instead of an identically-named primitive by prefacing the subroutine call with the character @. See *Resolving Name Conflicts between Host and Target*, p.265.

Table 7-13 **List of WindSh Commands**

agentModeShow( )	ipstatShow( )	smMemPartShow( )
b( )	iStrict( )	smMemShow( )
bd( )	l( )	so( )
bdall( )	ld( )	sp( )
bh( )	lkAddr( )	sps( )
bootChange( )	lkup( )	sysResume( )
browse( )	ls( )	sysStatusShow( )
c( )	m( )	sysSuspend( )
cd( )	memPartShow( )	taskCreateHookShow( )
checkStack( )	memShow( )	taskDeleteHookShow( )
classShow( )	moduleIdFigure( )	taskIdDefault( )
cplusCtors( )	moduleShow( )	taskIdFigure( )
cplusDtors( )	mqPxShow( )	taskRegsShow( )
cplusStratShow( )	mRegs( )	taskShow( )
cplusXtorSet( )	msgQShow( )	taskSwitchHookShow( )
cret( )	period( )	taskWaitShow( )
d( )	printErrno( )	tcpstatShow( )
devs( )	printLogo( )	td( )
h( )	pwd( )	tftpInfoShow( )
help( )	quit( )	ti( )
hostShow( )	reboot( )	tr( )
i( )	repeat( )	ts( )
icmpstatShow( )	routestatShow( )	tw( )
ifShow( )	s( )	udpstatShow( )
inetstatShow( )	semPxShow( )	unld( )
intVecShow( )	semShow( )	version( )
iosDevShow( )	shellHistory( )	w( )
iosDrvShow( )	shellPromptSet( )	wdShow( )
iosFdShow( )	show( )	tt( )

7.3.13 Terminal Control Characters

When you start a shell from the launcher, the launcher creates a new **xterm** window for the shell. The terminal control characters available in that window match whatever you are used to in your UNIX shells. You can specify all but one of these control characters (as shown in Table 7-15); see your host documentation for the UNIX command **stty**.

When you start the shell from the UNIX command line, it inherits standard input and output (and the associated **stty** settings) from the parent UNIX shell.

Table 7-15 lists special terminal characters frequently used for shell control. For more information on the use of these characters, see *7.5 Shell Line Editing*, p.292 and *7.2.8 Interrupting a Shell Command*, p.271.

Table 7-14 **Shell Special Characters**

stty Setting	Common Value	Description
eof	**CTRL+D**	End shell session.
erase	**CTRL+H**	Delete a character (backspace).
kill	**CTRL+U**	Delete an entire line.
intr	**CTRL+C**	Interrupt a function call.
quit	**CTRL+X**	Reboot the target, restart server, reattach shell.
stop	**CTRL+S**	Temporarily suspend output.
start	**CTRL+Q**	Resume output.
susp	**CTRL+Z**	Suspend the Tornado shell.
N/A	**ESC**	Toggle between input mode and edit mode. This character is fixed; you cannot change it with **stty**.

7.3.14 Redirection in the C Interpreter

The shell provides a *redirection* mechanism for momentarily reassigning the standard input and standard output file descriptors just for the duration of the parse and evaluation of an input line. The redirection is indicated by the **<** and **>** symbols followed by file names, at the very end of an input line. No other syntactic

elements may follow the redirection specifications. The redirections are in effect for all subroutine calls on the line.

For example, the following input line sets standard input to the file named **input** and standard output to the file named **output** during the execution of **copy( )**:

```
-> copy < input > output
```

If the file to which standard output is redirected does not exist, it is created.

Ambiguity Between Redirection and C Operators

There is an ambiguity between redirection specifications and the relational operators *less than* and *greater than*. The shell always assumes that an ambiguous use of **<** or **>** specifies a redirection rather than a relational operation. Thus the ambiguous input line:

```
-> x > y
```

writes the value of the variable **x** to the stream named **y**, rather than comparing the value of variable **x** to the value of variable **y**. However, you can use a semicolon to remove the ambiguity explicitly, because the shell requires that the redirection specification be the last element on a line. Thus the following input lines are unambiguous:

```
-> x; > y

-> x > y;
```

The first line prints the value of the variable **x** to the output stream **y**. The second line prints on standard output the value of the expression "**x** greater than **y**."

The Nature of Redirection

The redirection mechanism of the Tornado shell is fundamentally different from that of the Windows command shell, although the syntax and terminology are similar. In the Tornado shell, redirecting input or output affects only a command executed from the shell. In particular, this redirection is not inherited by any tasks started while output is redirected.

For example, you might be tempted to specify redirection streams when spawning a routine as a task, intending to send the output of **printf( )** calls in the new task to

an output stream, while leaving the shell's I/O directed at the virtual console. This stratagem does not work. For example, the shell input line:

```
-> taskSpawn (...myFunc...) > output
```

momentarily redirects the shell standard output during the brief execution of the spawn routine, but does not affect the I/O of the resulting task. To redirect the input or output streams of a particular task, call **ioTaskStdSet( )** once the task exists.

Scripts: Redirecting Shell I/O

A special case of I/O redirection concerns the I/O of the shell itself; that is, redirection of the streams the shell's input is read from, and its output is written to. The syntax for this is simply the usual redirection specification, on a line that contains no other expressions.

The typical use of this mechanism is to have the shell read and execute lines from a file. For example, the input lines:

```
-> <startup
-> < /usr/fred/startup
```

cause the shell to read and execute the commands in the file **startup**, either on the current working directory as in the first line or explicitly on the complete path name in the second line. If your working directory is **/usr/fred**, the two commands are equivalent.

Such command files are called *scripts*. Scripts are processed exactly like input from an interactive terminal. After reaching the end of the script file, the shell returns to processing I/O from the original streams.

During execution of a script, the shell displays each command as well as any output from that command. You can change this by invoking the shell with the **-q** option; see the **windsh** reference entry (in the online *Tornado API Reference*).

An easy way to create a shell script is from a list of commands you have just executed in the shell. The history command **h( )** prints a list of the last 20 shell commands. The following creates a file **/tmp/script** with the current shell history:

```
-> h > /tmp/script
```

The command numbers must be deleted from this file before using it a shell script.

Scripts can also be nested. That is, scripts can contain shell input redirections that cause the shell to process other scripts.

⚠ **CAUTION:** Input and output redirection must refer to files on a host file system. If you have a local file system on your target, files that reside there are available to target-resident subroutines, but not to the shell or to other Tornado tools (unless you export them from the target using NFS, and mount them on your host).

⚠ **CAUTION:** You should set the WindSh environment variable **SH_GET_TASK_IO** to off before you use redirection of input from scripts or before you copy and paste blocks of commands to the shell command line. Otherwise commands might be taken as input for a command that preceeds them, and lost.

C-Interpreter Startup Scripts

Tornado shell scripts can be especially useful for setting up your working environment. You can run a startup script through the shell C interpreter[5] by specifying its name with the **-s** command-line option to **windsh**. For example:

```
% windsh -s /usr/fred/startup
```

Like the **.login** or **.profile** files of the UNIX shells, startup scripts can be used for setting system parameters to personal preferences: defining variables, specifying the target's working directory, and so forth. They can also be useful for tailoring the configuration of your system without having to rebuild VxWorks. For example:

- creating additional devices
- loading and starting up application modules
- adding a complete set of network host names and routes
- setting NFS parameters and mounting NFS partitions

For additional information on initialization scripts, see *7.7 Tcl: Shell Interpretation*, p.297.

5. You can also use the **-e** option to run a Tcl expression at startup, or place Tcl initialization in **.wind/windsh.tcl** under your home directory. See *7.7.3 Tcl: Tornado Shell lnitialization*, p.300.

7.4 C++ Interpretation

Tornado supports both C and C++ as development languages; see *VxWorks Programmer's Guide: C++ Development* for information about C++ development. Because C and C++ expressions are so similar, the WindSh C-expression interpreter supports many C++ expressions. The facilities explained in *7.3 The Shell C-Expression Interpreter*, p.272 are all available regardless of whether your source language is C or C++. In addition, there are a few special facilities for C++ extensions. This section describes those extensions.

However, WindSh is not a complete interpreter for C++ expressions. In particular, the shell has no information about user-defined types; there is no support for the **::** operator; constructors, destructors, and operator functions cannot be called directly from the shell; and member functions cannot be called with the **.** or **->** operators.

To exercise C++ facilities that are missing from the C-expression interpreter, you can compile and download routines that encapsulate the special C++ syntax. Fortunately, the Tornado dynamic linker makes this relatively painless.

7.4.1 Overloaded Function Names

If you have several C++ functions with the same name, distinguished by their argument lists, call any of them as usual with the name they share. When the shell detects the fact that several functions exist with the specified name, it lists them in an interactive dialogue, printing the matching functions' signatures so that you can recall the different versions and make a choice among them.

You make your choice by entering the number of the desired function. If you make an invalid choice, the list is repeated and you are prompted to choose again. If you enter 0 (zero), the shell stops evaluating the current command and prints a message like the following, with *xxx* replaced by the function name you entered:

```
undefined symbol: xxx
```

This can be useful, for example, if you misspelled the function name and you want to abandon the interactive dialogue. However, because WindSh is an interpreter, portions of the expression may already have executed (perhaps with side effects) before you abandon execution in this way.

The following example shows how the support for overloaded names works. In this example, there are four versions of a function called **xmin( )**. Each version of

xmin( ) returns at least two arguments, but each version takes arguments of different types.

```
-> l xmin
"xmin" is overloaded - Please select:
    1: _xmin(double,double)
    2: _xmin(long,long)
    3: _xmin(int,int)
    4: _xmin(float,float)
Enter <number> to select, anything else to stop: 1
                _xmin(double,double):
3fe710 4e56 0000        LINK    .W      A6,#0
3fe714 f22e 5400 0008   FMOVE   .D      (0x8,A6),F0
3fe71a f22e 5438 0010   FCMP    .D      (0x10,A6),F0
3fe720 f295 0008        FB      .W      #0x8f22e
3fe724 f22e 5400 0010   FMOVE   .D      (0x10,A6),F0
3fe72a f227 7400        FMOVE   .D      F0,-(A7)
3fe72e 201f             MOVE    .L      (A7)+,D0
3fe730 221f             MOVE    .L      (A7)+,D1
3fe732 6000 0002        BRA             0x003fe736
3fe736 4e5e             UNLK            A6
value = 4187960 = 0x3fe738 = _xmin(double,double) + 0x28

-> l xmin
"xmin" is overloaded - Please select:
    1: _xmin(double,double)
    2: _xmin(long,long)
    3: _xmin(int,int)
    4: _xmin(float,float)
Enter <number> to select, anything else to stop: 3
                _xmin(int,int):
3fe73a 4e56 0000        LINK    .W      A6,#0
3fe73e 202e 0008        MOVE    .L      (0x8,A6),D0
3fe742 b0ae 000c        CMP     .L      (0xc,A6),D0
3fe746 6f04             BLE             0x003fe74c
3fe748 202e 000c        MOVE    .L      (0xc,A6),D0
3fe74c 6000 0002        BRA             0x003fe750
3fe750 4e5e             UNLK            A6
3fe752 4e75             RTS
                _xmin(long,long):
3fe7544e560000          LINK    .W      A6,#0
3fe758202e0008          MOVE    .L      (0x8,A6),D0
value = 4187996 = 0x3fe75c = _xmin(long,long) + 0x8
```

In this example, the disassembler is called to list the instructions for **xmin( )**, then the version that computes the minimum of two **double** values is selected. Next, the disassembler is invoked again, this time selecting the version that computes the minimum of two **int** values. Note that a different routine is disassembled in each case.

7.4.2 Automatic Name Demangling

Many shell debugging and system information functions display addresses symbolically (for example, the **l( )** routine). This might be confusing for C++, because compilers encode a function's class membership (if any) and the type and number of the function's arguments in the function's linkage name. The encoding is meant to be efficient for development tools, but not necessarily convenient for human comprehension. This technique is commonly known as *name mangling* and can be a source of frustration when the mangled names are exposed to the developer.

To avoid this confusion, the debugging and system information routines in WindSh print C++ function names in a demangled representation. Whenever the shell prints an address symbolically, it checks whether the name has been mangled. If it has, the name is demangled (complete with the function's class name, if any, and the type of each of the function's arguments) and printed.

The following example shows the demangled output when **lkup( )** displays the addresses of the **xmin( )** functions mentioned in *7.4.1 Overloaded Function Names*, p.290.

```
-> lkup "xmin"
_xmin(double,double)    0x003fe710 text         (templex.out)
_xmin(long,long)        0x003fe754 text         (templex.out)
_xmin(int,int)          0x003fe73a text         (templex.out)
_xmin(float,float)      0x003fe6ee text         (templex.out)
value = 0 = 0x0
```

7.5 Shell Line Editing

The WindSh front end provides a history mechanism similar to the UNIX Korn-shell history facility, including a built-in **vi**-like line editor that allows you to scroll, search, and edit previously typed commands. Line editing is available regardless of which interpreter you are using (C or Tcl[6]), and the command history spans both interpreters—you can switch from one to the other and back, and scroll through the history of both modes.

You can control what characters to use for certain editing commands. The input keystrokes shown in Table 7-15 (*7.3.13 Terminal Control Characters*, p.286) are set by

6. The WindSh Tcl-interpreter interface is described in *7.7 Tcl: Shell Interpretation*, p.297.

the host **stty** command (which you can call from the Tcl interpreter; see *7.7 Tcl: Shell Interpretation*, p.297). They must be single characters, usually control characters; Table 7-15 includes these characters, but shows only common default values.

The **ESC** key switches the shell from normal input mode to *edit mode*. The history and editing commands in Table 7-15 are available in edit mode.

Some line-editing commands switch the line editor to insert mode until an **ESC** is typed (as in **vi**) or until an **ENTER** gives the line to one of the shell interpreters. **ENTER** always gives the line as input to the current shell interpreter, from either input or edit mode.

In input mode, the shell history command **h( )** (described in *System Information*, p.257) displays up to 20 of the most recent commands typed to the shell; older commands are lost as new ones are entered. You can change the number of commands kept in history by running **h( )** with a numeric argument. To locate a line entered previously, press **ESC** followed by one of the search commands listed in Table 7-15; you can then edit and execute the line with one of the commands from Table 7-15.

NOTE: Not all the editing commands that take counts in **vi** do so in the shell's line editor. For example, *n***i** does not repeat the inserted text *n* times.

Table 7-15 **Shell Line-Editing Commands**

Basic Control	
h [*size*]	Display shell history if no argument; otherwise set history buffer to *size*.
ESC	Switch to line-editing mode from regular input mode.
ENTER	Give line to shell and leave edit mode.
CTRL+D	Complete symbol or path name (edit mode), display synopsis of current symbol (symbol must be complete, followed by a space), or end shell session (if the command line is empty).
[tab]	Complete symbol or path name (edit mode).
CTRL+H	Delete a character (backspace).
CTRL+U	Delete entire line (edit mode).
CTRL+L	Redraw line (works in edit mode).

Table 7-15 **Shell Line-Editing Commands** *(Continued)*

CTRL+S and **CTRL+Q**		Suspend output, and resume output.
CTRL+W		Display HTML reference page for a routine.
Movement and Search Commands		
*n***G**		Go to command number *n*.*
/*s* or **?***s*		Search for string *s* backward in history, or forward.
n		Repeat last search.
*n***k** or *n***-**		Get *n*th previous shell command.*
*n***j** or *n***+**		Get *n*th next shell command.*
*n***h**		Go left *n* characters (also **CTRL+H**).*
*n***l** or **SPACE**		Go right *n* characters.*
*n***w** or *n***W**		Go *n* words forward, or *n* large words. *†
*n***e** or *n***E**		Go to end of the *n*th next word, or *n*th next large word. *†
*n***b** or *n***B**		Go back *n* words, or *n* large words.*†
$		Go to end of line.
0 or **^**		Go to beginning of line, or first nonblank character.
f*c* or **F***c*		Find character *c*, searching forward, or backward.
Insert and Change Commands		
a or **A**	...**ESC**	Append, or append at end of line (**ESC** ends input).
i or **I**	...**ESC**	Insert, or insert at beginning of line (**ESC** ends input).
*n***s**	...**ESC**	Change *n* characters (**ESC** ends input).*
*n***c SPACE**	...**ESC**	Change *n* characters (**ESC** ends input).*
cw	...**ESC**	Change word (**ESC** ends input).
cc or **S**	...**ESC**	Change entire line (**ESC** ends input).
c$ or **C**	...**ESC**	Change from cursor to end of line (**ESC** ends input).

Table 7-15 **Shell Line-Editing Commands** *(Continued)*

c0	...**ESC**	Change from cursor to beginning of line (**ESC** ends input).
R	...**ESC**	Type over characters (**ESC** ends input).
*n***r***c*		Replace the following *n* characters with *c*.*
~		Toggle case, lower to upper or vice versa.
Delete Commands		
*n***x**		Delete *n* characters starting at cursor.*
*n***X**		Delete *n* character to left of cursor.*
dw		Delete word.
dd		Delete entire line (also **CTRL+U**).
d$ or **D**		Delete from cursor to end of line.
d0		Delete from cursor to beginning of line.
Put and Undo Commands		
p or **P**		Put last deletion after cursor, or in front of cursor.
u		Undo last command.

* The default value for *n* is 1.
† *words* are separated by blanks or punctuation; *large words* are separated by blanks only.

7.6 Object Module Load Path

In order to download an object module dynamically to the target, both WindSh and the target server must be able to locate the file. If path naming conventions are different between WindSh and the target server, the two systems may both have access to the file, but mounted with different path names. This situation arises often in environments where UNIX and Windows systems are networked together, because the path naming convention is different: the UNIX **/usr/fred/applic.o** may well correspond to the Windows **n:\fred\applic.o**. If you encounter this problem,

check to be sure the **LD_SEND_MODULES** variable of **shConfig** is set to "on" or use the **LD_PATH** facility to tell the target server about the path known to the shell.

Example 7-3 **Loading a Module: Alternate Path Names**

Your target server is running on a UNIX host. You start a WindSh on a Windows host with **LD_SEND_MODULES** set to "off" (the default is "on"). You want to download a file that resides on the Windows host called **c:/tmp/test.o**.

```
-> ld < c:/tmp/test.o
Loading c:/tmp/test.o
WTX Error 0x2 (no such file or directory)
value = -1 = 0xffffffff
```

This behavior is normal because the UNIX target server does not have access to this path. To correct the problem, reset **LD_SEND_MODULES** to "on" (the default).

```
-> ?shConfig LD_SEND_MODULES on
-> ld < c:/tmp/test.o
Loading C:/tmp/test.o
value = 17427840 = 0x109ed80
```

For more information on using **LD_SEND_MODULES**, **LD_PATH**, and other **shConfig** facilities, see *WindSh Environment Variables*, p.252.

Certain WindSh commands and browser utilities imply dynamic downloads of auxiliary target-resident code. These subroutines fail in situations where the shell and target-server view of the file system is incompatible. To get around this problem, download the required routines explicitly from the host where the target server is running (or configure the routines statically into the VxWorks image). Once the supporting routines are on the target, any host can use the corresponding shell and browser utilities. Table 7-16 lists the affected utilities. The object modules are in **wind/target/lib/obj***cputype***gnuvx**.

Table 7-16 **Shell and Browser Utilities with Target-Resident Components**

Utility	Supporting Module
repeat( )	**repeatHost.o**
period( )	**periodHost.o**
tt( )	**trcLib.o**, **ttHostLib.o**
Browser spy panel	**spyLib.o**

7.7 Tcl: Shell Interpretation

The shell has a Tcl interpreter interface as well as the C interpreter interface. This section illustrates some uses of the shell Tcl interpreter. If you are not familiar with Tcl, we suggest you skip this section and return to it after you have gotten acquainted with Tcl. (For an outline of Tcl, see *C. Tcl*.) In the interim, you can do a great deal of development work with the shell C interpreter alone.

To toggle between the Tcl interpreter and the C interpreter in the shell, type the single character **?**. The shell prompt changes to remind you of the interpreter state: the prompt **->** indicates the C interpreter is listening, and the prompt **tcl>** indicates the Tcl interpreter is listening.[7] For example, in the following interaction we use the C interpreter to define a variable in the symbol table, then switch into the Tcl interpreter to define a similar Tcl variable in the shell itself, and finally switch back to the C interpreter:

```
-> hello="hi there"
new symbol "hello" added to symbol table.
hello = 0x3616e8: value = 3544824 = 0x3616f8 = hello + 0x10
-> ?
tcl> set hello {hi there}
hi there
tcl> ?
->
```

If you start **windsh** from the Windows command line, you can also use the option **-Tclmode** (or **-T**) to start with the Tcl interpreter rather than the C interpreter.

Using the shell's Tcl interface allows you to extend the shell with your own procedures, and also provides a set of control structures which you can use interactively. The Tcl interpreter also acts as a host shell, giving you access to UNIX command-line utilities on your development host.

For example, you can call **stty** from the Tcl interpreter to change the special characters in use—in the following, to specify the quit character as **CTRL+B** and verify the new setting (the quit character is normally **CTRL+X**; when you type it in the shell, it reboots the target, restarts the target server, and resets all attached tools):

```
tcl> stty quit \002
tcl> stty
speed 9600 baud; line = 0;
quit = ^B; eof = ^A; status = <undef>; min = 1; time = 0;
-icanon -echo
```

7. The examples in this book assume you are using the default shell prompts, but you can change the C interpreter prompt to whatever string you like using **shellPromptSet( )**.

7.7.1 Tcl: Controlling the Target

In the Tcl interpreter, you can create custom commands, or use Tcl control structures for repetitive tasks, while using the building blocks that allow the C interpreter and the WindSh commands to control the target remotely. These building blocks as a whole are called the **wtxtcl** procedures.

For example, **wtxMemRead** returns the contents of a block of target memory (given its starting address and length). That command in turn uses a special memory-block datatype designed to permit memory transfers without unnecessary Tcl data conversions. The following example uses **wtxMemRead**, together with the memory-block routine **memBlockWriteFile**, to write a Tcl procedure that dumps target memory to a host file. Because almost all the work is done on the host, this procedure works whether or not the target run-time environment contains I/O libraries or any networked access to the host file system.

```
# tgtMemDump - copy target memory to host file
#
# SYNOPSIS:
#  tgtMemDump hostfile start nbytes

proc tgtMemDump {fname start nbytes} {
    set memHandle [wtxMemRead $start $nbytes]
    memBlockWriteFile $memHandle $fname
}
```

For reference information on the **wtxtcl** routines available in the Tornado shell, see the online *Tornado API Reference*.

All of the commands defined for the C interpreter (*7.2.4 Invoking Built-In Shell Routines*, p.253) are also available, with a double-underscore prefix, from the Tcl level; for example, to call **i( )** from the Tcl interpreter, run the Tcl procedure **__i**. However, in many cases, it is more convenient to call a **wtxtcl** routine instead, because the WindSh commands are designed to operate in the C-interpreter context. For example, you can call the dynamic linker using **ld** from the Tcl interpreter, but the argument that names the object module may not seem intuitive: it is the address of a string stored on the target. It is more convenient to call the underlying **wtxtcl** command. In the case of the dynamic linker, the underlying **wtxtcl** command is **wtxObjModuleLoad**, which takes an ordinary Tcl string as its argument, as described in the online *Tornado API Reference: WTX Tcl API*.

Tcl: Calling Target Routines

The **shParse** utility allows you to embed calls to the C interpreter in Tcl expressions; the most frequent application is to call a single target routine, with the arguments specified (and perhaps capture the result). For example, the following sends a logging message to your VxWorks target console:

```
tcl> shParse {logMsg("Greetings from Tcl!\n")}
32
```

You can also use **shParse** to call WindSh commands more conveniently from the Tcl interpreter, rather than using their **wtxtcl** building blocks. For example, the following is a convenient way to spawn a task from Tcl, using the C-interpreter command **sp( )**, if you do not remember the underlying **wtxtcl** command:

```
tcl> shParse {sp appTaskBegin}
task spawned: id = 25e388, name = u1
0
```

Tcl: Passing Values to Target Routines

Because **shParse** accepts a single, ordinary Tcl string as its argument, you can pass values from the Tcl interpreter to C subroutine calls simply by using Tcl facilities to concatenate the appropriate values into a C expression.

For example, a more realistic way of calling **logMsg( )** from the Tcl interpreter would be to pass as its argument the value of a Tcl variable rather than a literal string. The following example evaluates a Tcl variable **tclLog** and inserts its value (with a newline appended) as the **logMsg( )** argument:

```
tcl> shParse "logMsg(\"$tclLog\\n\")"
32
```

7.7.2 Tcl: Calling Under C Control

To dip quickly into Tcl and return immediately to the C interpreter, you can type a single line of Tcl prefixed with the **?** character (rather than using **?** by itself to toggle into Tcl mode). For example:

```
-> ?set test wonder; puts "This is a $test."
This is a wonder.
->
```

Notice that the **->** prompt indicates we are still in the C interpreter, even though we just executed a line of Tcl.

CAUTION: You may not embed Tcl evaluation inside a C expression; the **?** prefix works only as the first nonblank character on a line, and passes the entire line following it to the Tcl interpreter.

For example, you may occasionally want to use Tcl control structures to supplement the facilities of the C interpreter. Suppose you have an application under development that involves several collaborating tasks; in an interactive development session, you may need to restart the whole group of tasks repeatedly. You can define a Tcl variable with a list of all the task entry points, as follows:

```
-> ? set appTasks {appFrobStart appGetStart appPutStart …}
appFrobStart appGetStart appPutStart …
```

Then whenever you need to restart the whole list of tasks, you can use something like the following:

```
-> ? foreach it $appTasks {shParse "sp($it)"}
task spawned: id = 25e388, name = u0
task spawned: id = 259368, name = u1
task spawned: id = 254348, name = u2
task spawned: id = 24f328, name = u3
```

7.7.3 Tcl: Tornado Shell Initialization

When you execute an instance of the Tornado shell, it begins by looking for a file called **.wind/windsh.tcl** in the directory specified by the **HOME** environment variable (if that environment variable is defined). In each of these directories, if the file exists, the shell reads and executes its contents as Tcl expressions before beginning to interact. You can use this file to automate any initialization steps you perform repeatedly.

You can also specify a Tcl expression to execute initially on the **windsh** command line, with the option **-e** *tclExpr*. For example, you can test an initialization file before saving it as **.wind/windsh.tcl** using this option, as follows:

```
% windsh phobos -e "source ./tcltest" &
```

Example 7-4 **Shell Initialization File**

This file causes I/O for target routines called in WindSh to be directed to the target's standard I/O rather than to WindSh. It changes the default C++ strategy

to automatic for this shell, sets a path for locating load modules, and causes modules not to be copied to the target server.

```
# Redirect Task I/O to WindSh
shConfig SH_GET_TASK_IO off
# Set C++ strategy
shConfig LD_CALL_XTORS on
# Set Load Path
shConfig LD_PATH "/folk/jmichel/project/app;/folk/jmichel/project/test"
# Let the Target Server directly access the module
shConfig LD_SEND_MODULES off
```


7.8 The Shell Architecture

7.8.1 Controlling the Target from the Host

Tornado integrates host and target resources so well that it creates the illusion of executing entirely on the target itself. In reality, however, most interactions with any Tornado tool exploit the resources of both host and target. For example, Table 7-17 shows how the shell distributes the interpretation and execution of the following simple expression:

```
-> dir = opendir ("/myDev/myFile")
```

Parsing the expression is the activity that controls overall execution, and dispatches the other execution activities. This takes place on the host, in the shell's C interpreter, and continues until the entire expression is evaluated and the shell displays its result.

To avoid repetitive clutter, Table 7-17 omits the following important steps, which must be carried out to link the activities in the three contexts (and two systems) shown in each column of the table:

– After every C-interpreter step, the shell program sends a request to the target server representing the next activity required.

– The target server receives each such request, and determines whether to execute it in its own context on the host. If not, it passes an equivalent request on to the target agent to execute on the target.

Table 7-17 **Interpreting: dir = opendir ("/myDev/myFile")**

Tornado Shell (on host)	Target Server & Symbol Table (on host)	Agent (on target)
Parse the string **"/myDev/myFile"**.		
	Allocate memory for the string; return address **A**.	
		Write **"/myDev/myFile"**; return address **A**.
Parse the name **opendir**.		
	Look up **opendir**; return address **B**.	
Parse the function call **B(A)**; wait for the result.		
		Spawn a task to run **opendir( )** and signal result **C** when done.
	Receive **C** from target agent and pass it to host shell.	
Parse the symbol **dir**.		
	Look up **dir** (fails).	
Request a new symbol table entry **dir**.		
	Define **dir**; return symbol **D**.	
Parse the assignment **D=C**.		
	Allocate agent-pool memory for the value of **dir**.	
		Write the value of **dir**.

The first access to server and agent is to allocate storage for the string **"/myDev/myFile"** on the target and store it there, so that VxWorks subroutines (notably **opendir( )** in this case) have access to it. There is a pool of target memory

reserved for host interactions. Because this pool is reserved, it can be managed from the host system. The server allocates the required memory, and informs the shell of its location; the shell then issues the requests to actually copy the string to that memory. This request reaches the agent on the target, and it writes the 14 bytes (including the terminating null) there.

The shell's C-expression interpreter must now determine what the name **opendir** represents. Because **opendir( )** is not one of the shell's own commands, the shell looks up the symbol (through the target server) in the symbol table.

The C interpreter now needs to evaluate the function call to **opendir( )** with the particular argument specified, now represented by a memory location on the target. It instructs the agent (through the server) to spawn a task on the target for that purpose, and awaits the result.

As before, the C interpreter looks up a symbol name (**dir**) through the target server; when the name turns out to be undefined, it instructs the target server to allocate storage for a new **int** and to make an entry pointing to it with the name **dir** in the symbol table. Again these symbol-table manipulations take place entirely on the host.

The interpreter now has an address (in target memory) corresponding to **dir**, on the left of the assignment statement; and it has the value returned by **opendir( )**, on the right of the assignment statement. It instructs the agent (again, through the server) to record the result at the **dir** address, and evaluation of the statement is complete.

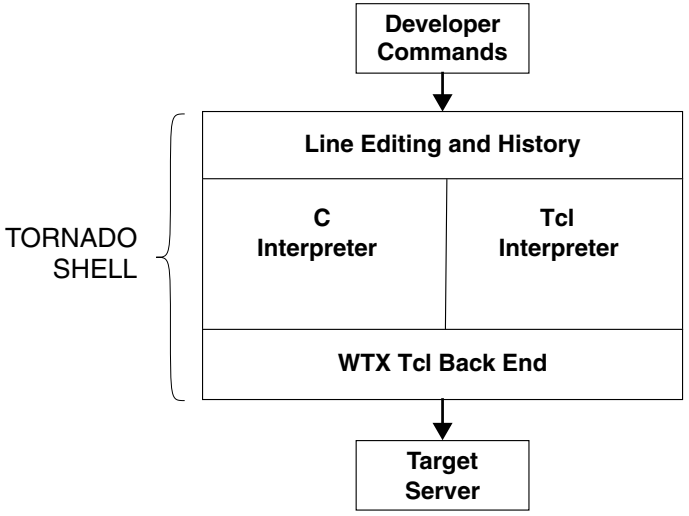
7.8.2 Shell Components

The Tornado shell includes two interpreters, a common front end for command entry and history, and a back end that connects the shell to the global Tornado environment to communicate with the target server. Figure 7-3 illustrates these components:

Line Editing
: The line-editing and command history facilities are designed to be unobtrusive, and support your access to the interpreters. *7.5 Shell Line Editing*, p.292 describes the **vi**-like editing and history front end.

C-Expression Interpreter
: The most visible component is the C-expression interpreter, because it is the interface that most closely resembles the application programming environment. The bulk of this chapter describes that interpreter.

Figure 7-3 **Components of the Tornado Shell**



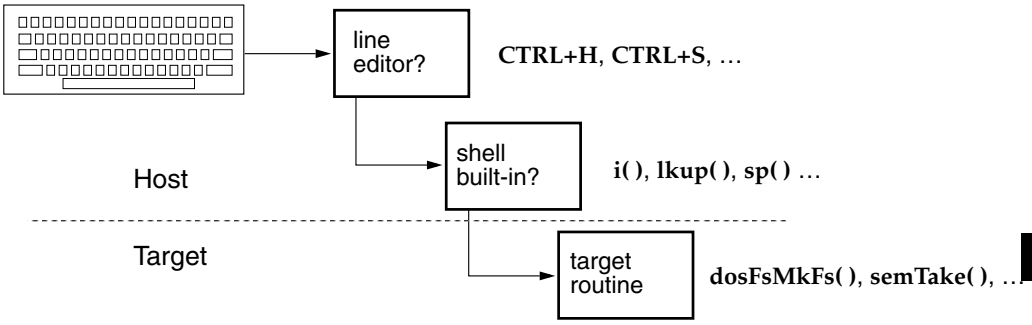
Tcl Interpreter
: An interface for extending the shell or automating shell interactions, described in *7.7 Tcl: Shell Interpretation*, p.297.

WTX Tcl
: The back-end mechanism that ties together all of Tornado; the Wind River Tool Exchange protocol, implemented as a set of Tcl extensions.

7.8.3 Layers of Interpretation

In daily use, the shell seems to be a seamless environment; but in fact, the characters you type in WindSh go through several layers of interpretation, as illustrated by Figure 7-4. First, input is examined for special editing keystrokes (described in *7.5 Shell Line Editing*, p.292). Then as much interpretation as possible is done in WindSh itself. In particular, execution of any subroutine is first attempted in the shell itself; if a shell built-in (also called a *primitive*) with that name exists, the built-in runs without any further checking. Only when a subroutine call does not match any shell built-ins does WindSh call a target routine. See *7.2.4 Invoking Built-In Shell Routines*, p.253 for more information. For a list of all WindSh primitives, see Table 7-13.

Figure 7-4 **Layers of Interpretation in the Shell**



8

Browser

8.1 A System-Object Browser

The Tornado browser conveniently monitors the state of your target. The main browser window summarizes active tasks (classified as system tasks or application tasks), memory consumption, and a summary of the current target memory map. Using the browser, you can also examine:

- detailed task information
- semaphores
- message queues
- memory partitions
- watchdog timers
- stack usage by all tasks on the target
- target CPU usage by task
- object-module structure and symbols
- interrupt vectors

These displays are snapshots. They can be updated interactively, or the browser can be configured to automatically update its displays at a specified interval. When any displayed information changes, in any browser display, the browser highlights the affected line. (The style of highlighting depends on the capabilities of your X Window System display server, but always includes boldface display for the changed data.)

8.2 Starting the Browser

There are two ways to start a Tornado browser:

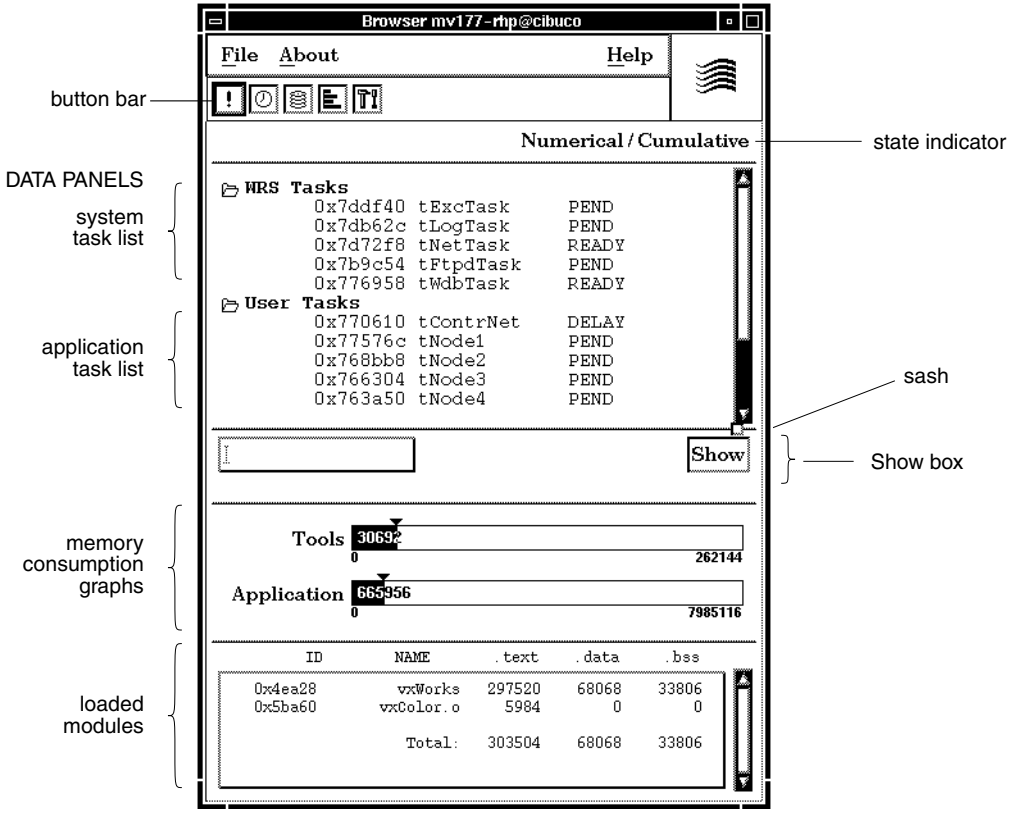
- From the launcher: select the desired target and press the 

button.
- From the UNIX command line: run **browser**, specifying the target server's name as the argument as in the following example:

```
% browser mv177-rhp@cibuco &
```

In either case, the first display is the main target-browser window, shown in Figure 8-1.

Figure 8-1 **Target Browser Window**



8.3 Anatomy of the Target Browser

The main browser window, shown in Figure 8-1, provides an overview of the attached target, and also allows you to control other browser functionality.

Data Panels
: The panels labeled along the left hand side of Figure 8-1 (system task list, application task list, memory-consumption graphs, and loaded modules) provide overall information about your target system. See *8.5 Data Panels*, p.312 for a more detailed description.

Button Bar
: Buttons to give you fresh snapshots of your target, to request specialized displays containing overall target information, and to adjust browser parameters. *8.4 Browser Menus and Buttons*, p.310 describes each button.

State indicator
: Using the controls in the browser's button bar, you can change how the browser behaves. The *state indicator* bar summarizes the current state of toggles that affect the browser. The states listed in below may appear in the state indicator.

Alphabetical	Sort all symbols alphabetically by name. Non-default state. Converse: Numerical.
Numerical	Sort all symbols by numerical value. Default state. Converse: Alphabetical.
Cumulative	Show total CPU usage in the spy window. See *8.9 The Spy Window*, p.324. Default state. Converse: Differential.
Differential	Show CPU usage within the sampling interval in the spy window. See *8.9 The Spy Window*, p.324. Non-default state. Converse: Cumulative.
Update	Sample target state and update displays periodically, rather than on demand. Non-default state. Converse: blank.

Sash
: The sash allows you to allocate space between the panels of the main target-browser window. To reapportion the space above and below the sash, drag the small square up or down with your mouse pointer.

Show box
: A text entry box where you can request display of system objects. See *8.6 Object Browsers*, p.313.

8.4 Browser Menus and Buttons

The browser's menu bar offers the standard Tornado menu entries: you can abandon the browser session by selecting File>Quit, query the Tornado version from the About menu, and peruse the *Tornado Online Manuals* from the Help menu.

The row of buttons immediately below the menu bar provides browser-specific controls, with the following meanings:

	Immediate-update	Use this button to update all browser displays immediately. This button causes an immediate update even if a periodic update is running.
	Periodic-update	This button is a toggle: press it to request or cancel regular updates of all browser displays. When periodic updates are on, the browser reflects this by displaying the word Update in its state indicator, below the button bar.
	Stack-check	This button produces a stack-usage histogram for all tasks in the target system (see *8.10 The Stack-Check Window*, p.325).
	Interrupt Vectors	This button produces an interrupt vector table for all possible interrupt vectors. It appears only for those targets which support the interrupt vector table.
	Spy	This button is a toggle: press it to bring up a histogram displaying CPU utilization by all running tasks (see *8.9 The Spy Window*, p.324). Press the button a second time to stop data sampling for the histogram.
	Parameter adjustment	Press this button to adjust the parameters that govern the browser's behavior. Figure 8-2 shows the browser config form displayed when you press this button.

You can use the browser config form produced by the  button (Figure 8-2) to change the following browser parameters:

Symbol sort
: This toggle switches between numeric or alphabetic sorting order for symbols displayed by the browser, and updates the state indicator to match.

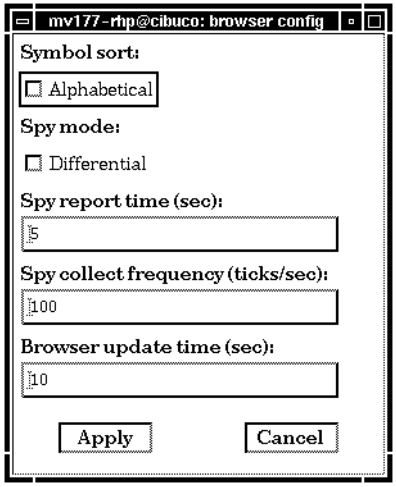
Spy mode
: This toggle switches the spy window between cumulative and differential modes (see *8.9 The Spy Window*, p.324).

Spy report time
: This text box specifies how many seconds elapse between browser updates while spy mode is on.

Spy collect frequency
: This text box specifies how many times per second to gather data for the spy window.

Browser update time
: This text box specifies how often browser windows are updated if spy mode is not on, but periodic updates are running.

Figure 8-2 **Form: Browser Parameters**



8.5 Data Panels

The main browser window includes several information panels (labeled along the left of Figure 8-1) to provide an overview of the state of the target system.

System Task List
: Summary information on all operating-system tasks currently running on the target. To hide this task list (leaving more space for the application-task summary), click on the folder labeled WRS Tasks. To bring up the list again, click again on the same folder.

Application Task List
: Summary information on all application tasks currently running on the target. To hide this task list, click on the folder labeled User Tasks. To bring up the list again, click again on the same folder.

The task-summary display (for either system or application tasks) includes the task ID, the task name (if known), and the task state.

You can display detailed information on any of these tasks by clicking on the summary line for that task; see *8.6.1 The Task Browser*, p.314.

Memory-Consumption Graphs
: The two bar graphs in this panel show what proportions of target memory are currently in use.

  The upper bar shows the state of the memory pool managed by the target agent.[1] This represents target memory consumed by Tornado tools, for example with dynamically-linked modules or for variables defined from the shell.

  The lower bar shows the memory consumed by all tasks in the target system, including both application (user) tasks and system tasks.

  The agent-memory pool is not part of VxWorks' memory. If the target server wants to allocate more memory than available in the agent-memory pool, it will allocate memory from the VxWorks memory pool and add it to the agent-memory pool.

  Clicking on the lower bar produces a more detailed display of system memory (the memory display described in *8.6.4 The Memory-Partition Browser*, p.318, applied to the system-memory partition; this display is shown in Figure 8-12).

1. To set the size of this memory pool, see *Scaling the Target Agent*, p.161.

In both bars, the shaded portion and the numeric label inside the bar measure the memory currently in use; the small triangle above the bar is a "high-water mark," indicating the largest amount of memory that has been in use so far in the current session; and the numbers below the bar indicate the total memory size available in each pool. All memory-size numbers are byte counts in decimal.

Loaded-Module List
The bottom panel in the main target browser lists each binary object file currently loaded into your target. This includes the VxWorks image (including any statically linked application code) and all dynamically-loaded object modules.

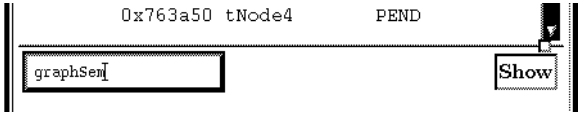
8.6 Object Browsers

The Show box (in the middle of the main browser window) gives you access to the browser's specialized object displays. Type either the name or the ID of a system object in the text-entry field to the left of this panel. Then press the Show button (or simply press the **ENTER** key) to bring up a browser for that particular object.

Another way to bring up the specialized browser displays is to click on the name of an object in the module browser (*8.7 The Module Browser*, p.320). If the object is a recognized system object, the browser for it is displayed just as if you had copied the name to the Show box.

For example, Figure 8-3 shows the Show box filled in with a request to display a browser for an object called **graphSem**:

Figure 8-3 **Filling in the** Show **Box**

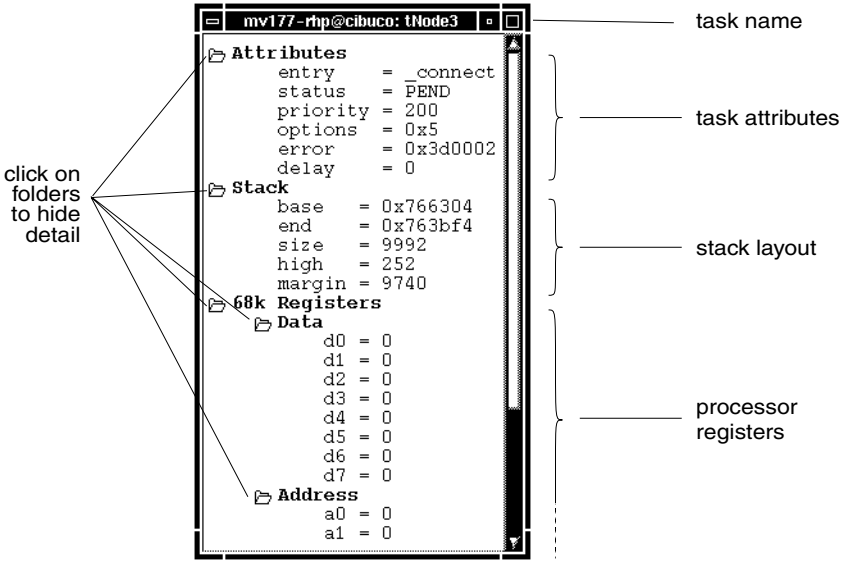


To dismiss specialized object browsers, use the window manager's controls.

8.6.1 The Task Browser

To see more detailed information about a particular task, click on any browser window displaying the task name or task ID. For example, you can click on any task's summary line in the main target browser. The browser displays a window for that task, similar to Figure 8-4.

Figure 8-4 **Task Browser (Initial Display)**

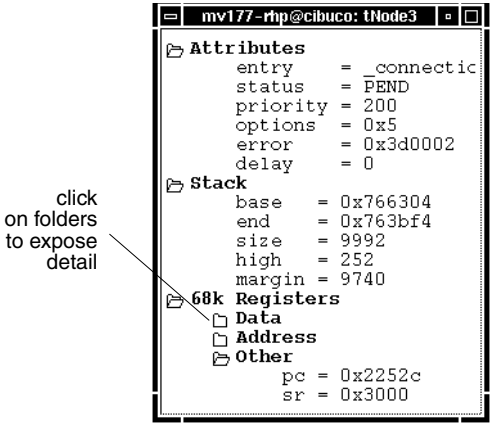


The task name appears on the title bar, to help you observe multiple tasks side by side. At the top of the task browser you can see global task attributes, and information about stack allocation and usage. The last major region shows the hardware registers for this task; their precise organization and contents depends on your target architecture. As usual, a scrollbar is displayed if more room is needed.

Notice the folder icons; the lines they mark categorize the task information. You can hide any information that is not of interest to you by clicking on any open folder, or expose such hidden information by clicking on any closed folder. Figure 8-5 shows another task browser running on the same target architecture, but with most of the hardware registers hidden.

Task-browser windows close automatically when the corresponding tasks are deleted.

Figure 8-5 **Task Browser (Hiding Registers)**



8.6.2 The Semaphore Browser

To inspect a semaphore, enter either its name or its semaphore ID in the main target browser's Show box. A specialized semaphore browser appears, similar to the one shown in Figure 8-6. The semaphore browser displays both information about the semaphore itself (under the heading Attributes), and the complete queue of tasks blocked on that semaphore, under the heading Blocked Tasks. The title bar shows the semaphore ID, to help you distinguish browser displays for multiple semaphores.

Figure 8-6 **Semaphore Browser**

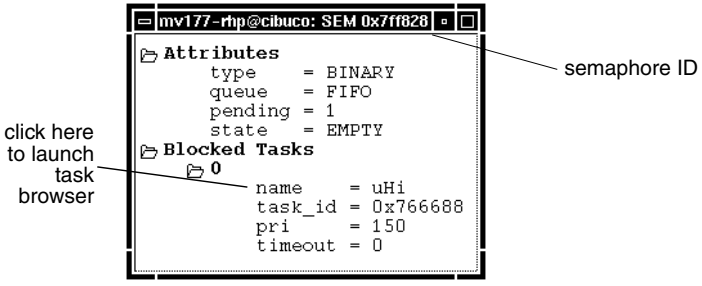


Figure 8-6 shows a binary semaphore with one blocked task in its queue. As in other browser windows, you can click on the folders to control detail. To start a

browser for any queued task, click on the task name or ID; both are displayed for each task.

POSIX semaphores have a somewhat different collection of attributes, and the browser display for a POSIX semaphore reflects those differences. Figure 8-7 shows an example of a browser display for a POSIX semaphore. Similarly, the semaphore browser adapts to shared-memory semaphores; Figure 8-8 exhibits that semaphore display.

Figure 8-7 **POSIX Semaphore**

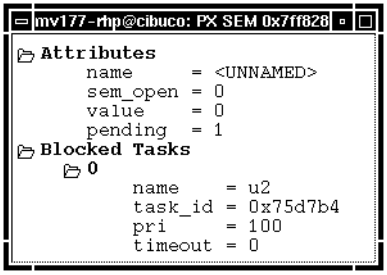
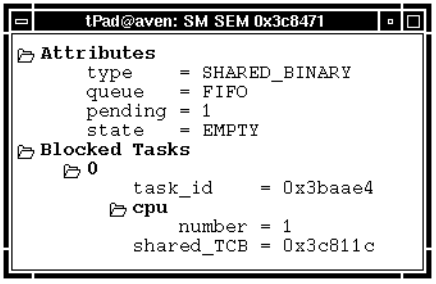


Figure 8-8 **Shared-Memory Semaphore**



Semaphore-browser windows are closed automatically when the corresponding semaphore is deleted.

8.6.3 The Message-Queue Browser

To inspect a message queue, enter its name or message-queue ID in the main target browser's Show box. A message-queue browser like the one in Figure 8-9 is displayed.

Figure 8-9 **Message Queue**

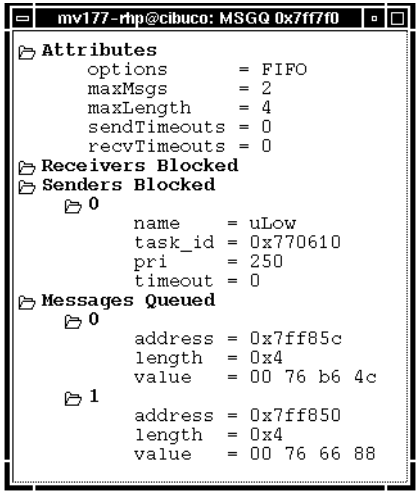
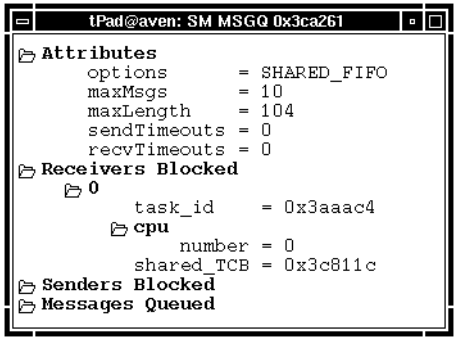


Figure 8-10 **Shared-Memory Message Queue**

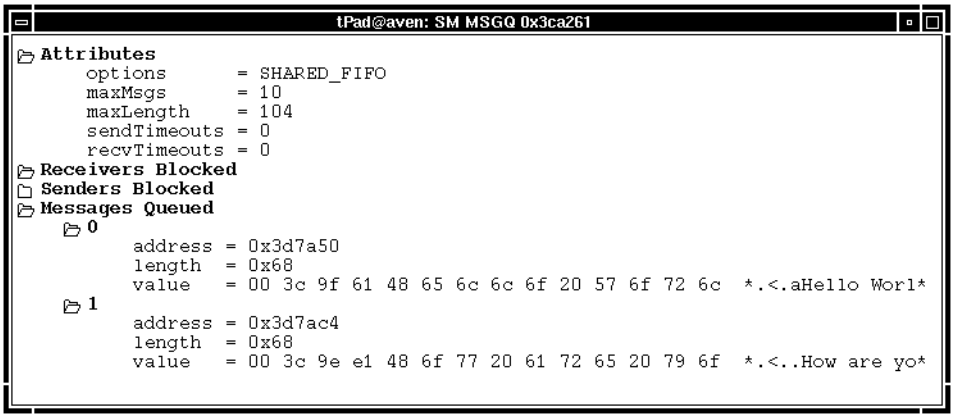


As well as displaying the attributes of the message queue, the message-queue browser shows three queues. Receivers Blocked shows all tasks waiting for messages from the message queue. Senders Blocked shows all tasks waiting for space to become available to place a message on the message queue. Messages Queued shows the address and length of each message currently on the message queue. As shown in Figure 8-10, shared-memory message queues have a very similar display format (differing only in the title bar).

Just as for semaphores, the message-queue browser also has a POSIX-attribute version (not shown).

If a message queue contains longer messages, you can resize the browser window to exhibit as much of the message as is convenient. Figure 8-11 shows a shared-memory message queue in a display widened for this purpose.

Figure 8-11 **Message Queue Browser: Wider Display**



Message-queue browser windows are closed automatically when the corresponding message queue is deleted.

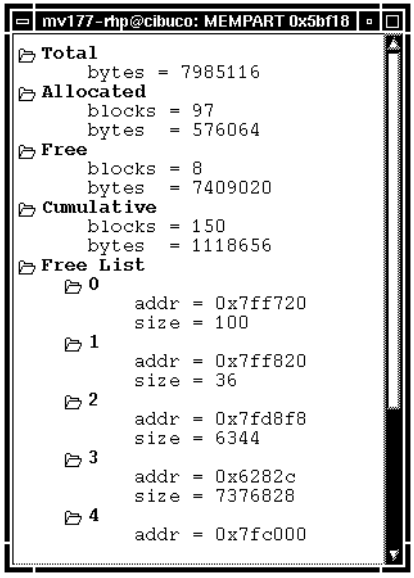
8.6.4 The Memory-Partition Browser

Just as is the case for all other specialized browser windows, the memory-partition browser comes up when the browser recognizes a memory partition ID (or a variable name containing one) entered in the Show box. Figure 8-12 shows **memSysPartId**, the VxWorks system memory partition.

By default the memory-partition browser displays the following:

- The total size of the partition.
- The number of blocks currently allocated, and their total size in bytes.
- The number of blocks currently free, and their total size in bytes.
- The total of all blocks and all bytes allocated since booting the target system (headed Cumulative).
- For each block currently on the free list, its size and address.

Figure 8-12 **Memory-Partition Browser**

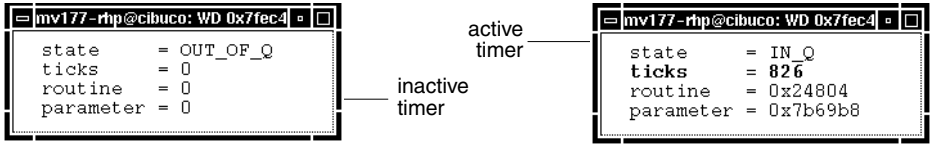


As for other object browsers, you can control the level of detail visible by clicking on the folder icons beside each heading.

8.6.5 The Watchdog Browser

When the Tornado browser recognizes a watchdog-timer ID (or a variable containing one) in the Show box, it displays a window like those shown in Figure 8-13.

Figure 8-13 **Watchdog Browser**

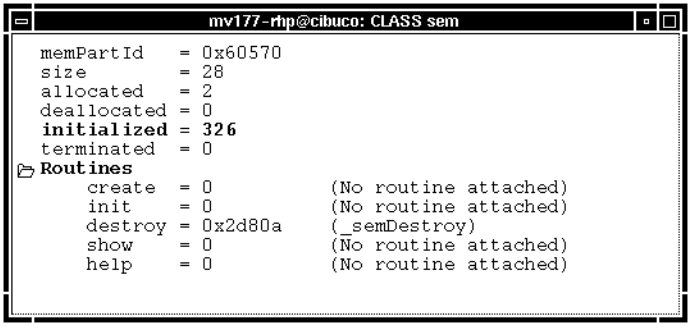


Before you start a timer, the display resembles the one on the left of Figure 8-13; only the state field is particularly meaningful. After the timer starts counting, however, you can see the number of ticks remaining, the address of the routine to be executed when the timer expires, and the address of its parameter.

8.6.6 The Class Browser

VxWorks kernel objects are implemented as related *classes*: collections of objects with similar properties. Each class has an identifier in the run-time; the symbol names for these identifiers end with the string *ClassId*, making them easy to recognize. When you enter a class identifier in the Show box, the browser displays a window with overall information about the objects in that class. For example, Figure 8-14 shows the display for **semClassId** (the semaphore class).

Figure 8-14 **Class Browser (Semaphore Class)**



You can get a list of the class identifiers in your run-time by executing the following in a shell window:

```
-> lkup "ClassId"
```

8.7 The Module Browser

To inspect the memory map of any currently loaded module, click on the line that lists the module in the loaded-module list (the bottom panel in the main browser window).

The browser opens a specialized object-module browser resembling Figure 8-16 for the selected module.

Figure 8-15 **Loaded-Module List in Main Browser Window**

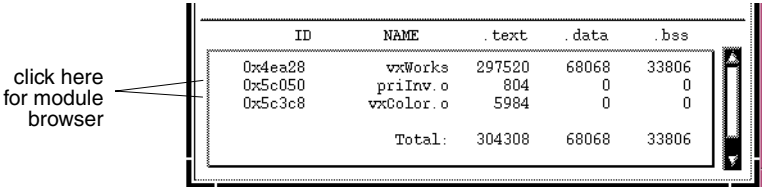
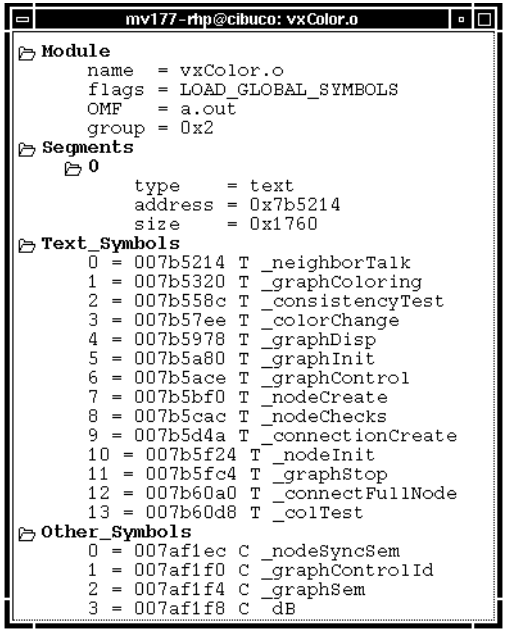


Figure 8-16 **Object-Module Browser**



The object-module browser displays information in the following categories:

Module

Overall characteristics of the object module: its name, the loader flags used when the module was downloaded to the target, the object-module format (OMF), and the group number. (The group number is a sequence number recorded in the symbol table to identify all of the symbols belonging to a single module.)

Segments
: For each segment (section) of the object module: the segment type (text, bss, or data), starting address, and size in bytes.

Symbols
: The bulk of the object-module browser display is occupied by a listing of symbols and their addresses. Symbols are displayed in either alphabetical or numeric order, depending on what browser state is in effect when you request a module browser.

Each symbol's display occupies one line. The symbol display includes the symbol's address in hexadecimal, a letter representing the symbol type (Table 8-1), and the symbol name (in its internal representation—C++ symbols are displayed "mangled", and all compiled-language symbols begin with an underbar).

Table 8-1 **Key Letters in Symbol Displays**

Symbol Key		Symbol Type
Global	**Local**	
A	a	absolute
B	b	bss segment
C		common (uninitialized global symbol)
D	d	data segment
T	t	text segment
?	?	unrecognized symbol

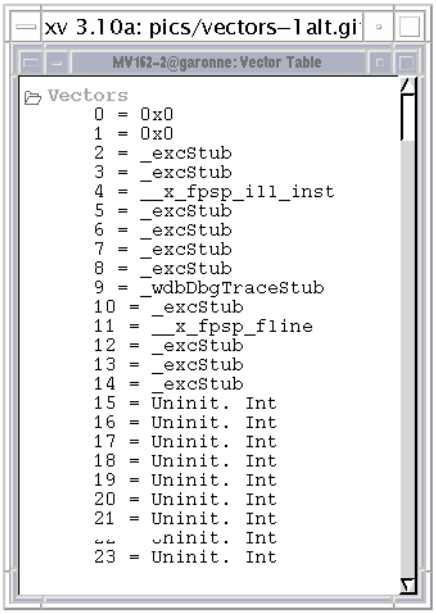
For symbols that represent system object, clicking on the symbol name brings up the specialized object browser; see *8.6 Object Browsers*, p.313.

Symbol displays are grouped by category. There is one category for the symbols in each section, plus a category headed Other_Symbols that contains uninitialized globals and unrecognized symbols.

8.8 The Vector Table Window

To inspect the interrupt/exception vector table, click Vector Table in the browser window selector. (This facility is available for all target architectures except PowerPC, and ARM.) The display is similar to Figure 8-17.

Figure 8-17 **Vector Table Window**



Vectors are numbered from 0 to X (X = number of interrupt/exception vectors). The connected routines or addresses are displayed, or if no routine is connected the following key words are be displayed:

Std Excep. Handler
: standard exception handler

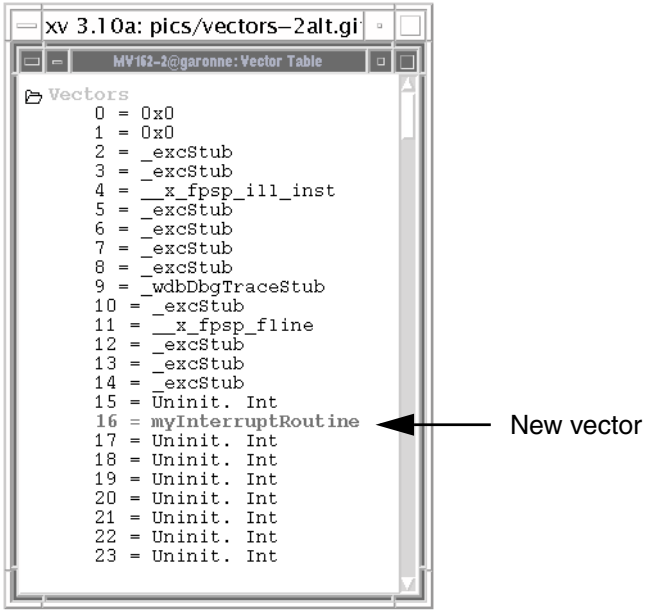
Default Trap
: default trap (Sparc)

Uninit. Int
: uninitialized interrupt vector

Corrupted Int
: corrupted interrupt vector

If you set a new vector from WindSh and then update the browser, the new vector is highlighted as shown in Figure 8-18.

Figure 8-18 **New Interrupt Vector**

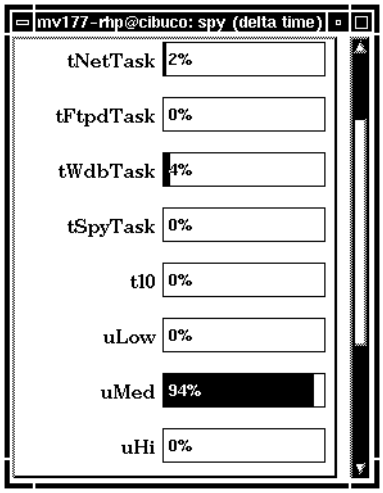


8.9 The Spy Window

Pressing the spy button produces a window similar to Figure 8-19. This spy window reports on CPU utilization for each task on your target, as a percentage of CPU cycles. Besides tasks, the spy window always includes the following additional categories for CPU-cycle usage: time spent in the kernel, time spent in interrupt handlers, and idle time. These additional categories appear below all task data; you may need to use the scrollbar to see them.

Spy data is reported in one of two modes (selected with the browser config form shown in Figure 8-2). Reports in Cumulative mode (noted on the state indicator line) show total CPU usage since you first display the spy window. Reports in Differential

Figure 8-19 **Spy Window**



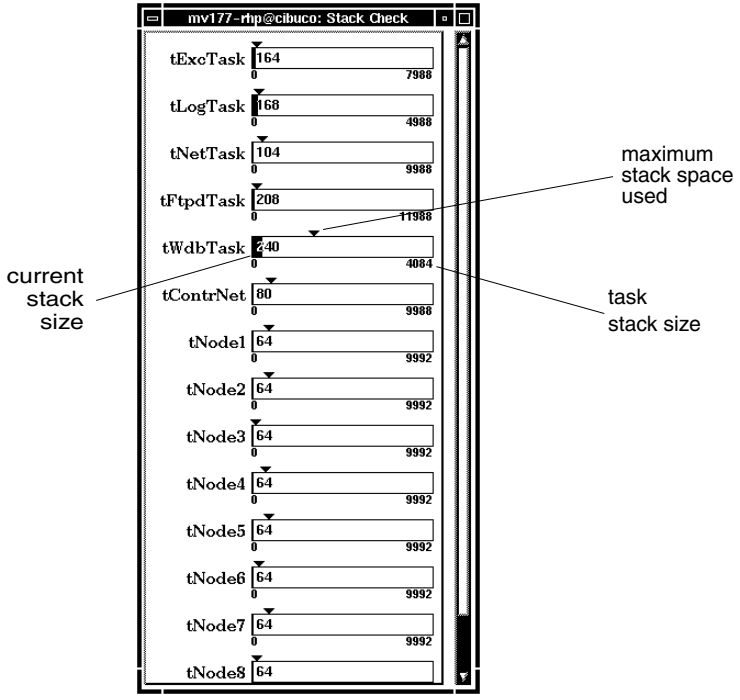
mode reflect only the CPU usage since the last update. The spy mode for the window is also noted in the title line: in cumulative mode, the title bar reads spy (total time), while in differential mode it reads spy (delta time).

The spy window uses the facilities of the VxWorks target software in **spyLib** (which is automatically downloaded to the target when you request a spy window, if it is not already present there). For related information, see the reference entries for **spyLib**.

8.10 The Stack-Check Window

When you press the stack-check button, the browser displays a stack-check window similar to Figure 8-20. The stack-check window summarizes the current and maximum stack usage for each task currently running.

Figure 8-20 **Stack-Check Window**



The display for each task presents three values:

– The stack size allocated for each task, shown as a number of bytes beneath the bar representing that task.

– The maximum stack space used so far by each task is indicated graphically by the small triangle above the task's bar.

– The portion of the stack currently in use, shown in two different ways: as a number of bytes, displayed within the bar graph for each task, and as a proportion of that task's stack space, indicated graphically by the shaded portion of each task's bar.

NOTE: It is possible for a task to write beyond the end of its stack while not writing to the last part of its stack. This will not be detected by **checkStack( )**, the underlying routine for the stack-check window.

8.11 Browser Displays and Target Link Speed

If your communications link to the target is slow (a serial line, for example), use the browser judiciously. The traffic back and forth to the target grows with the number of objects displayed, and with the update frequency. This traffic may seriously slow down overall Tornado performance, on slow links. If you experience this problem, try displaying fewer objects, updating browser displays on request instead of periodically, or setting updates to a longer interval.

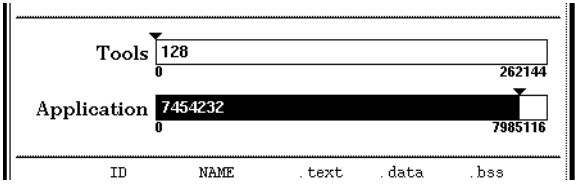
8.12 Troubleshooting with the Browser

Many problem conditions in target applications become much clearer with the browser's visual feedback on the state of tasks and critical objects in the target. The examples in this section illustrate some of the possibilities.

8.12.1 Memory Leaks

The browser makes memory leaks easy to notice, through the memory-consumption bar graphs in the main browser window: if the allocated portion of memory grows continually, you have a problem. The memory-consumption graph in Figure 8-21 corresponds to a memory leak in an application that has run long enough to almost completely run out of memory.

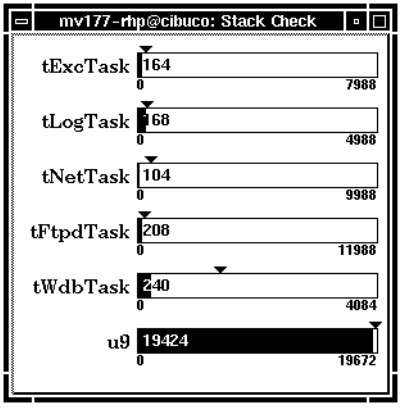
Figure 8-21 **A Memory Leak as Seen in the Browser**



8.12.2 Stack Overflow

When a task exceeds its stack size, the resulting problem is often hard to trace, because the initial symptom may be in some other task altogether. The browser's stack-check window is useful when faced with behavior that is hard to explain: if the problem is a stack overflow, you can spot it immediately. The affected task's stack display shows a high-water mark at the right edge, as in the example in Figure 8-22.

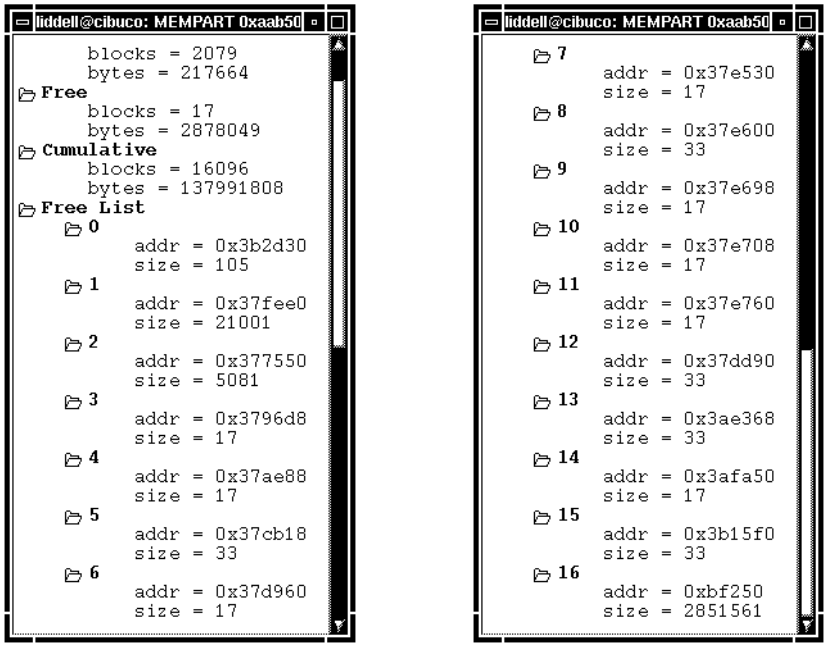
Figure 8-22 **Stack Overflow on Task u9**



8.12.3 Memory Fragmentation

A more subtle memory-management problem occurs when small blocks of memory that are not freed for long periods are allocated interleaved with moderate-sized blocks of memory that are freed more frequently: memory can become fragmented, because the calls to **free( )** for the large blocks cannot coalesce the free memory back into a single large available-memory pool. This problem is easily observed by examining the affected memory partition (in simple applications this is the VxWorks system memory partition, **memSysPartId**) with the browser. Figure 8-23 shows an example of a growing free-list with many small blocks, characteristic of memory fragmentation.

Figure 8-23 **Fragmented Memory as Seen in the Browser**



8.12.4 Priority Inversion

The browser's displays are most useful when they complement each other. For example, suppose you notice in the main browser window (as in Figure 8-24) that a task expected to be high priority is blocked while two other tasks are ready to run.

An immediate thing to check is whether the three tasks really have the expected priority relationship (in this example, the names are chosen to suggest the intended priorities: **uHi** is supposed to have highest priority, **uMed** medium priority, and **uLow** the lowest). You can check this immediately by clicking on each task's summary line, thus bringing up the windows shown in Figure 8-25.

Unfortunately, that turns out not to be the explanation; the priorities (shown for each task under Attributes) are indeed as expected. Examining the CPU allocations with the spy window (Figure 8-26) reveals that the observed situation is ongoing; **uMed** is monopolizing the target CPU. It should certainly execute by preference to the low-priority **uLow**, but why is **uHi** not getting to run?

Figure 8-24 **Browser: uHi Pended**

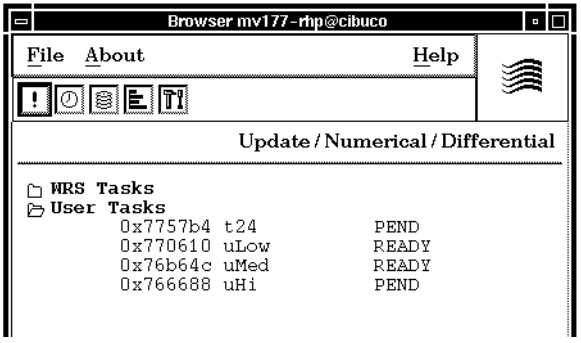


Figure 8-25 **Task Browsers for uHi, uMed, uLow**

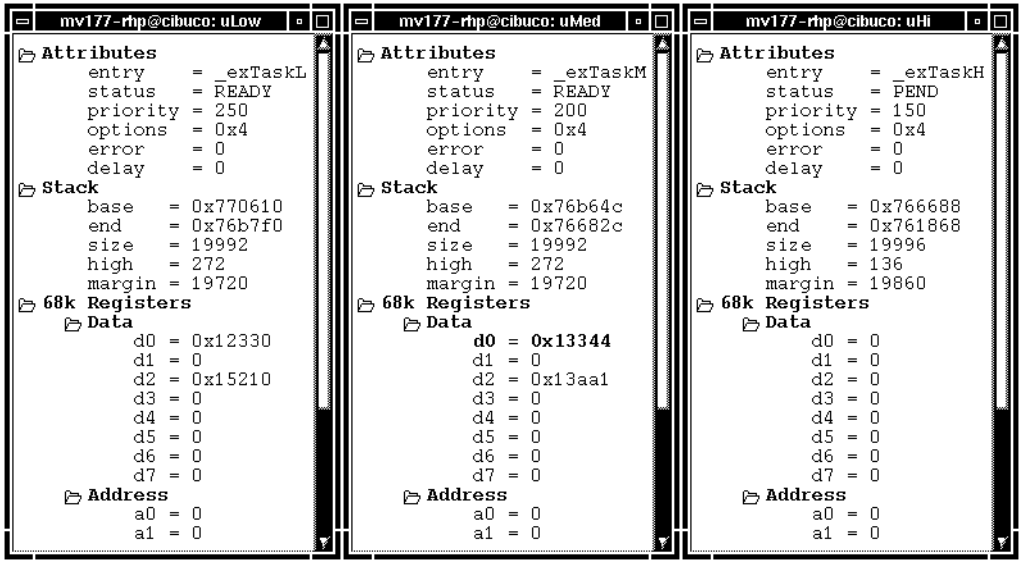
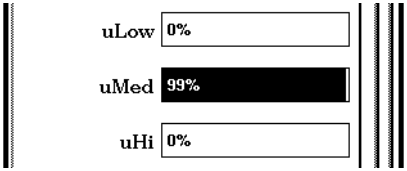


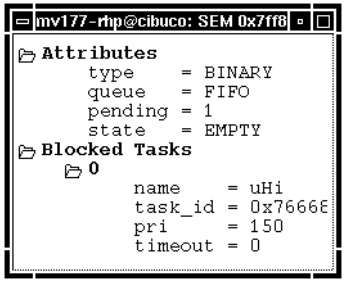
Figure 8-26 **uMed Monopolizing CPU (Spy Window Excerpt)**



At this point examining the code (not shown) may seem worthwhile. Doing so, you notice that **uMed** uses no shared resources, but **uHi** and **uLow** synchronize their work with a semaphore.

Examining the semaphore with the browser (Figure 8-27) confirms the dawning suspicion: **uHi** is indeed blocking on the semaphore, which **uLow** cannot release because **uMed** has preempted it.

Figure 8-27 **uHi Blocked on Semaphore**



Having diagnosed the problem as a classic priority inversion, the fix is straightforward. As described in *VxWorks Programmer's Guide: Basic OS*, you can revise the application to synchronize **uLow** and **uHi** with a mutual-exclusion semaphore created with the **SEM_INVERSION_SAFE** option.

8.13 Tcl: the Browser Initialization File

When the browser begins executing, it first checks for a file called **.wind/browser.tcl** in your home directory. If this file exists, its contents are sourced as Tcl code.

For example, it may be convenient to download object modules from the browser. The following **browser.tcl** code defines a button and a procedure to implement this.

Example 8-1 **Browser Extension: a Download Button**

```
# BUTTON: "Ld" -- Download an object module under browser control

toolBarItemCreate Ld button {loadDialog}

set currentWdir [pwd]              ;# default working dir for loadDialog

##############################################################################
#
#
# loadDialog - load an object module from the browser
#
# This routine supports a "load" button added to the browser's button bar.
# It prompts for a file name, and calls the WTX download facility to load it.
#
# SYNOPSIS: loadDialog
#
# RETURNS: N/A
#
# ERRORS: N/A

proc loadDialog {} {
    global currentWdir

    cd [string trim $currentWdir "\n"]
    set result [noticePost fileselect Download Load "*.o"]
    if {$result != ""} {
        set currentWdir [file dirname $result]
        wtxObjModuleLoad $result
        update       ;# Show new object module in browser
    }
}
```

9

Debugger

9.1 Introduction

The design of the Tornado debugger, CrossWind, combines the best features of graphical and command-line debugging interfaces. The most common debugging activities, such as setting breakpoints and controlling program execution, are available through convenient point-and-click interfaces. Similarly, program listings and data-inspection windows provide an immediate visual context for the crucial portions of your application.

For more complex or unpredictable debugging needs, a command-line interface gives you full access to a wealth of specialized debugging commands. You can extend or automate command-line debugger interactions in the following complementary ways:

– A Tcl scripting interface allows you to develop custom debugger commands.

– You can extend the point-and-click interface, defining new buttons that attach to whatever debugging commands (including your own debugger scripts) you use most frequently.

The underlying debugging engine is an enhanced version of GDB, the portable symbolic debugger from the Free Software Foundation (FSF). For full documentation of the GDB commands, see *GDB User's Guide*.

9.2 Starting CrossWind

There are two ways to start a debugging session:

- From the launcher: Select the desired target and press the 

button.
- From the UNIX command line: Invoke **crosswind**, specifying the target architecture with the **-t** option as in the following example.

```
% crosswind -t ppc &
```

  If you start the debugger from the command line, you must still select a target. You can either use the Targets menu (see *CrossWind Menus*, p.338 for details) or the **target wtx** command (see *Managing Targets*, p.358).

9.3 A Sketch of CrossWind

Figure 9-1 illustrates the layout of the main debugger window. This section discusses each part of the window briefly. The following sections provide more detail.

The menu bar ❶ provides access to overall control facilities: the File menu lets you load application modules, or exit; the Targets menu provides a quick way to switch tasks or targets; the Source menu lets you choose among source-code display formats; the Tcl menu re-initializes the graphical front end after any customization to its Tcl definitions; and the Windows menu controls the display of auxiliary debugger windows. For detailed descriptions of these menus, see *CrossWind Menus*, p.338. As usual, the About menu leads to Tornado version information, and the Help menu leads to the Tornado online manuals.

The buttons ❷ are the quick path to common debugger commands. Most of them are grouped into related pairs. Table 9-1 shows a summary of each button's purpose. For more detailed descriptions of these buttons, see *CrossWind Buttons*, p.344.

The *program-display panel* ❸ is empty when the debugger begins executing. The debugger automatically displays the current context here, whenever a command or an event sets the context. In the case of Figure 9-1, the display is the effect of the **list** command, often a useful way to start. Once there is a display in the program-display panel, you can select symbols or lines inside that display to serve as arguments to the *button bar* in area ❷.

Figure 9-1 **CrossWind Display**

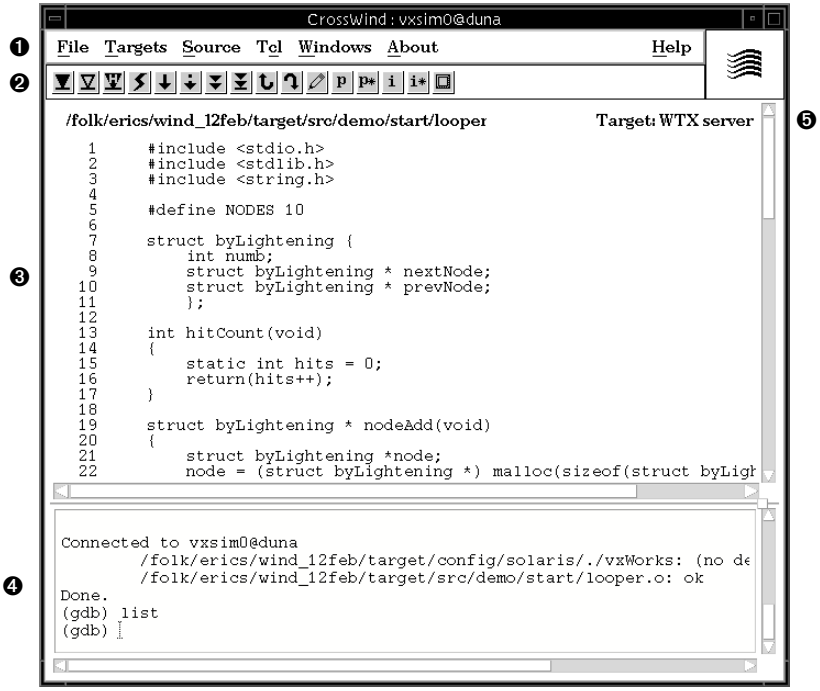


Table 9-1 **Summary of CrossWind Buttons**

Button	Description	Button	Description
	Breakpoint.		Move up the subroutine stack.
	Temporary breakpoint.		Move down the stack.
	Hardware breakpoint.		Call up editor.
	Interrupt.		Print selected symbol.
	Step to the next line of code.		Dereference pointer.
	Step over a function call.		Monitor symbol value.
	Continue program execution.		Monitor value at pointer.
	Finish the current subroutine.		Define a new button.

The *command panel* ❹ allows you to interact directly with the debugger, issuing commands such as the **list** command. Type the **add-symbols** command here if needed to load symbol information for any modules the debugger cannot find on its own; see *What Modules to Debug*, p.354.

The *state indicator* line ❺ reports on the state of the debugger connection. At the left of this line, the debugger shows the name of the source file (if any) for the code being debugged. At the right of the line, the debugger indicates what it is connected to (if anything) by showing one of the messages shown in Table 9-2.

Table 9-2 **Messages in CrossWind State Indicator Line**

Message	Status
No Target	No target currently selected.
Target: WTX server	Connected to a target in task mode, but not to any particular task.
Target: WTX Task/Stopped	Connected to a task, which is stopped.
Target: WTX Task/Exception	Connected to a task, which is stopped due to an exception.
Target: WTX Task/Running	Connected to a task, which is running.
Target: WTX System	Connected to a target in system mode.

9.4 CrossWind in Detail

This section describes the debugger commands and controls in detail. *9.4.1 Graphical Controls*, p.336 is a complete discussion of all graphical debugger controls. *9.4.2 Debugger Command Panel: GDB*, p.353 discusses when to use the command panel rather than graphical controls, and what commands are particularly useful because of their effects on the graphical context.

9.4.1 Graphical Controls

The debugger provides three kinds of graphical controls: menus, buttons, and mouse-based manipulation of other display elements.

Display Manipulation

scrollbars
: Whenever the amount of text in either main panel exceeds the available space, the debugger displays scrollbars to allow you to view whatever portion of either display you are interested in. As shown in Figure 9-1, you can scroll the command panel either vertically or horizontally, and you can scroll the program-display panel vertically. (Make the window wider if you need to see wider lines in your source program.)

sash
: If you refer again to Figure 9-1, you can see that the two major portions of the debugger display are the program-display panel ❸ and the command panel ❹. As in other Tornado tools, the separator line between these two panels includes a *sash* ( ). The sash allows you to allocate space between the two panels. To change the amount of space each panel takes up in the overall display, drag the small square up or down with your mouse.

left click
: Clicking the left mouse button selects the entire word under the pointer in the program-display panel (but not in the command panel). This is often useful for selecting a symbol as an argument to one of the debugger buttons.

right click
: Clicking the right mouse button anywhere in the program-display panel sets a breakpoint on the line under the pointer.

  A right click on any line with an existing breakpoint (marked in the margin of the program-display panel) removes that breakpoint.

left drag
: As with other X applications, you can drag the left mouse button over any displayed text to select it (whether as an argument to another control in the debugger, or to copy into another window).

middle drag
: Use the middle mouse button to drag certain controls and symbols between the button bar and the program-display panel. For example, drag the pencil (using the middle button) to a particular source line, to edit that line; or drag the program-counter symbol to another source line to continue executing until that line is reached.

context ▶
: The debugger displays this icon in the program-display panel, at the left of the next line to be executed, each time your program stops. Drag the context

pointer (using the middle mouse button) to another line to allow execution to continue until the program reaches the line you indicate. The shading of the context pointer becomes gray if the program is running, or if the stack level displayed is not the current stack level. (For another way of doing this, see the discussion of the continue button in *CrossWind Buttons*, p.344.)

CrossWind Menus

The menu bar at the top of the debugger display provides commands for overall control of the debugger display or debugging session (Figure 9-2).

Figure 9-2 **CrossWind Menu Bar**

The following paragraphs describe the effect of each debugger-specific menu command.

File Menu

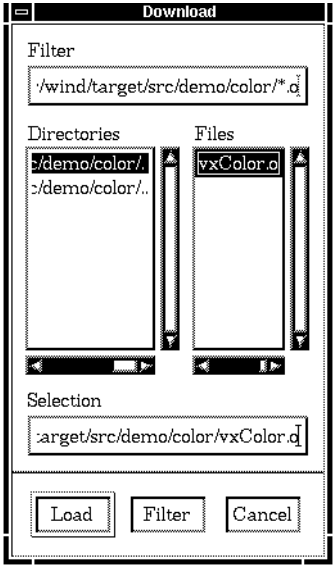
File>Quit
: Ends the debugging session. If a target is attached, Quit also kills any suspended routines from the debugging session.

  (If you want to leave the target undisturbed when you quit, first use the Detach Task command under Targets, or type the GDB command **detach** in the command panel.)

File>Download
: Load an object module into target memory, link it dynamically into the run-time, and load its symbols into the debugger. This command displays a file selector, as shown in Figure 9-3, to choose the object module.

  In the file selector, choose files or directories from the two scrolling lists by clicking on a name. You can type directly in the Filter text box to change the selection criteria. The Filter button redisplays the scrolling lists for a new directory or a new file-name expansion constraint; click the Load button when the file you want to download is selected. Double-clicking on a directory is equivalent to selecting the directory and then pressing Filter; double-clicking on a file is equivalent to Load.

Figure 9-3 **Download File Selector**



⚠ **CAUTION:** Because the download is controlled by the target server, a download can fail when the server and CrossWind have different views of the file system. See *Extended Debugger Variables*, p.360.

In the command panel, you can use a form of the **load** command to get around this problem. See *What Modules to Debug*, p.354.

Targets Menu

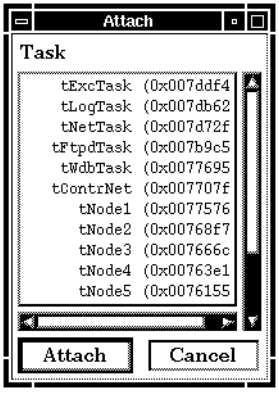
The commands in the Targets menu allow you to select or change debugging targets.

⚠ **CAUTION:** If you select a command from the Targets menu while the debugger is attached to a running task, the command does not take effect until the next time the task stops. You can force the task to stop by pressing the interrupt button.

Targets>Attach Task

Attach the debugging session to a task that is already running. This command displays a scrolling list of the tasks that are running on the target (Figure 9-4). When you select one, the debugger stops the task.

Figure 9-4 **Attach Task Selector**



Usually, a newly-attached task stops in a system routine; thus, the debugger displays an assembly listing in its program-display panel. Use the up-stack button to view a stack level where source code is visible, or use the finish button to allow the system routine to return to its caller.

Targets>Attach System
: Switches the target connection into system mode (if supported by the target agent) and stops the entire target system.

Targets>Detach Task/System
: If the debugger is currently attached to a task, it releases the current task from debugger control. This allows exiting the debugger, or switching to system mode, without killing the task that was being debugged. If the debugger is currently attached to the target system, it sets the agent to tasking mode (if supported) and the target system resumes operation.

Targets>Kill Task
: Delete the current task from the target system without exiting the debugger.

Targets>Connect Target Servers
: Connect the debugger to a target. This command displays a scrolling list of all targets available through the Tornado registry (Figure 9-4). If the debugger is already connected to a target, selecting a new target releases the current target from debugger control.

Figure 9-5 **Connect**

Source Menu

The commands in the Source menu control how your program is displayed.

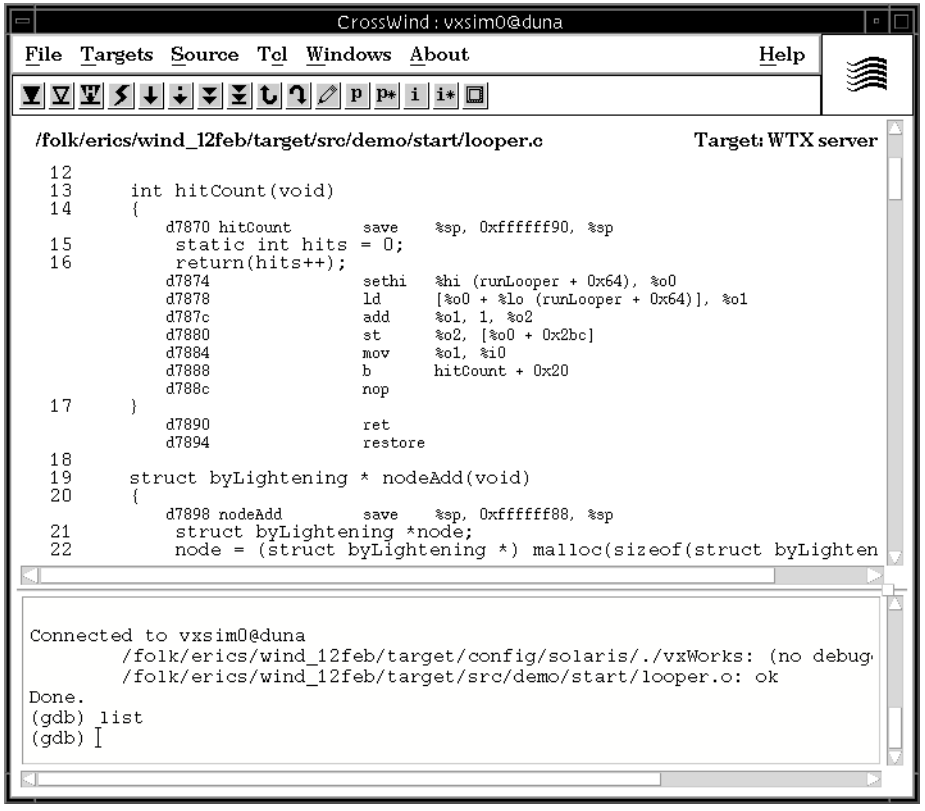
Source>C/C++
: Displays the original high-level language source code (usually C or C++). This style of display is only available for modules compiled with debugging information. When this display is available, it is also the default style of program display.

Source>Assembly
: Displays only assembly-level code (a symbolic disassembly of your program's object code). This style of display is the default for routines compiled without debugging information (such as VxWorks system routines supplied as object code only).

Source>Mixed
: Displays both high-level source and a symbolic disassembly, with the assembly-level code shown as close as possible to the source code that generates the corresponding object code. This display style is only available for modules compiled with debugging information.

  Figure 9-6 shows the debugger using mixed-mode code display. (Notice also that the sash was dragged all the way down for this figure, thus devoting the maximum available area to the program-display panel.)

Figure 9-6 **Mixed-Mode Code Display**



NOTE: For some source lines, compilers can generate code that is not contiguous, because it is sometimes more efficient to interleave the object code from separate lines of source.

In this situation, the mixed-mode display rearranges the assembly listing to group all object code below the line that generates it. The debugger indicates any rearranged chunks of the assembly with an asterisk at the start of each non-contiguous segment in the mixed-mode display.

Tcl Menu

The Tcl menu provides a way to re-initialize the debugger. Because the debugger can be customized on the fly (see *9.6 Tcl: CrossWind Customization*, p.374), this

provides a way to restore the environment after experiments with custom modifications.

Tcl>Reread Home
: Re-initializes the definitions from *homeDir*/**.wind/crosswind.tcl** in your home directory (see *9.6.1 Tcl: Debugger Initialization Files*, p.375).

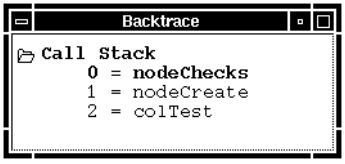
Tcl>Reread All
: Re-initializes the complete graphical environment defined in Tcl resource files, including both the basic CrossWind definitions from your home directory and the definitions from *installDir*/**host/resource/tcl/CrossWind.tcl**.

Windows Menu

The Windows menu controls auxiliary debugger displays. All such displays are automatically updated whenever the control of execution passes to the debugger—for example, at each breakpoint, or after single-stepping.

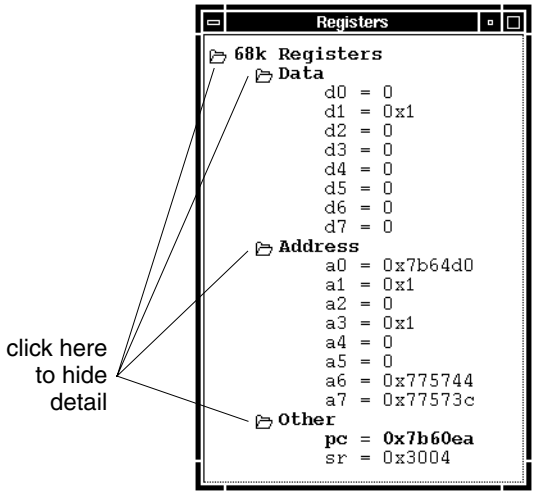
Windows>Backtrace
: Displays an auxiliary window with the current stack trace, like the one in Figure 9-7.

Figure 9-7 **Stack Display**



Windows>Registers
: Displays an auxiliary window that shows the machine registers for the task you are debugging. Because registers are different for each architecture, the precise contents of this window differ depending on your target. Figure 9-8 shows a register-display window for a 68K target. As with the register displays in task browsers (see *8.6.1 The Task Browser*, p.314), registers are grouped by category, and you can control the level of detail by clicking on the folder icons that head each category.

Figure 9-8 **Register Display**



Help Menu

The Help menu has the standard entries On CrossWind (the debugger's reference entry), Manuals Index (the online search tool), and Manuals Contents (the start of the Tornado online manuals). It also has one additional entry. The GDB Online command brings up an auxiliary viewer for the command-language usage summaries built into GDB.

CrossWind Buttons

Just below the menu bar is a row of buttons called the *button bar*. These buttons provide quick access to the most important debugger functionality. The following paragraphs describe each button:

Sets a breakpoint on the current line, or on a selected symbol. For example, if you have just single-stepped through a portion of your program, press this button to stop execution the next time your program executes this line. Alternately, to stop at the beginning of a routine, click on the routine name (either where it is defined, or anywhere that the subroutine is called), then press this button.

The debugger uses the same symbol that appears on the breakpoint button to indicate the breakpoint location in the program-display panel's left margin.

You can also set a breakpoint on any particular line by right-clicking on that line, or by dragging the breakpoint symbol from the button-bar (with the middle mouse button) down to the line where you want to break.

To delete a breakpoint, click with the right mouse button anywhere on a line that is already marked with the breakpoint icon, or drag the breakpoint icon back to the break button. You can also use the debugger **delete** command with the breakpoint number (as originally shown in the command panel, or as displayed with **info break**). The **delete** command with no arguments deletes all breakpoints.

To disable a breakpoint, drag the breakpoint icon to the hollow breakpoint symbol in the button bar (the temporary-breakpoint button), or use the **disable** command with the breakpoint number.

NOTE: The breakpoints you set in this way will affect only the task to which the debugger is attached. If you want your breakpoint to stop all tasks when the attached task hits the breakpoint, set it using **gbreak** from the command line.

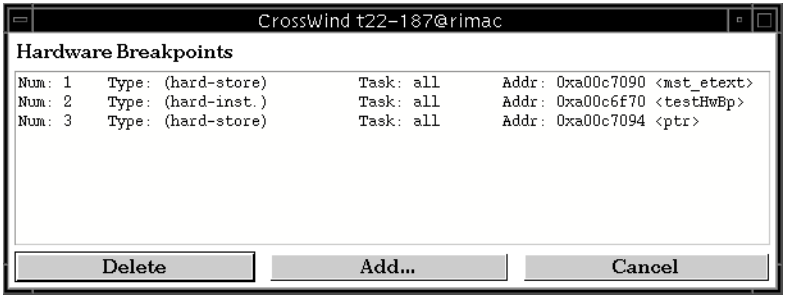
Sets a temporary breakpoint. This button works almost the same way as the breakpoint button above; the difference is that a temporary breakpoint stops the program only once. The debugger deletes it automatically as soon as the program stops there. The hollow breakpoint symbol on the button marks temporary breakpoints in the program-display panel, so that you can readily distinguish the two kinds of breakpoints there.

You can delete or disable temporary breakpoints in the same ways as other breakpoints—delete by right-clicking on a line displaying a breakpoint symbol, by dragging the breakpoint symbol up to the solid breakpoint symbol in the button bar, or by using the **delete** command; disable by dragging to the hollow breakpoint symbol, or by invoking the **disable** command.

Launches a hardware breakpoint window that allows you to set and delete hardware breakpoints, if they are supported by the target (Figure 9-9).

The hardware breakpoint window lists all hardware breakpoints currently set on the target. Hardware breakpoints set during the current CrossWind session are marked with an asterisk (*).[1]

Figure 9-9 **Hardware Breakpoints Window**



To add a hardware breakpoint, click on the Add button to display the Set Hardware Breakpoint window (Figure 9-10). If you had previously selected a symbol in the source window, the Address or Symbol Name field would be filled automatically with the symbol. Otherwise, you can enter information about where the breakpoint should be set using standard GDB syntax. For example, enter "value" to set the breakpoint on the symbol **value**; enter "'testHwBp.c'::value" to set the breakpoint on symbol **value** in file **testHwBp.c**; enter "*ptr" to set the breakpoint on the address pointed to by **ptr**; enter "0x10000" to set the breakpoint at address 0x10000; and so on.

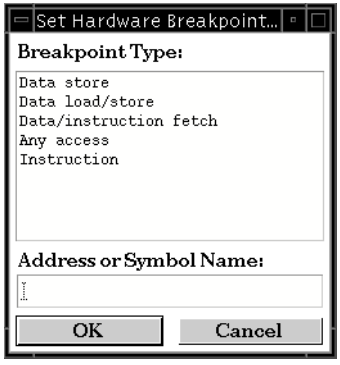
Then select the breakpoint type from the Breakpoint Type list, and click the OK button.

To delete a breakpoint, click on its name in the Hardware Breakpoints window, and then the Delete button. You can only delete hardware breakpoints set during the current CrossWind session.

If the target agent is running in task mode, a hardware breakpoint is set on all the system tasks. If the agent is running in system mode, a hardware breakpoint is set on the system context.

1. The hardware breakpoint list is refreshed at a regular interval (five seconds by default). It allows you to see the hardware breakpoints set or removed by other tools (such as the shell, WindSh). To change the polling interval or simply suppress polling, edit your *installDir*/**.wind/crosswind.tcl** file (set **hwBpPollInterval** to 0 to suppress polling). If polling is suppressed, the hardware breakpoint list is only updated when the Add or Delete buttons are used.

Figure 9-10 **Set Hardware Breakpoint Dialog Box**



NOTE: CrossWind does not manage hardware breakpoints in the same manner as standard GDB breakpoints. The hardware breakpoint interface is provided as a simple means of setting hardware breakpoints on the target (which is why it is only possible to set hardware breakpoints on all the tasks or on the system context, and not only on the task to which the debugger is attached).

When a data access hardware breakpoint stops the program, the context icon indicates the line of code that has been executed. However, on some processors a data access exception is generated only after the data has been accessed and the program counter has been incremented. For those processors, CrossWind marks the line after the one that was executed when the data hardware breakpoint was hit.

NOTE: GDB software watchpoints are very intrusive. They should only be used with real-time programs if the overhead is acceptable.

Interrupt. Sends an interrupt to the task that the debugger is controlling. For example, if your program keeps running instead of following an expected path to a breakpoint, press this button to regain control and investigate. Pressing this button is equivalent to keying the interrupt character (usually **CTRL+C**).

Steps to the next line of code. The precise effect depends on the style of program display you have selected. If the program-display area shows high-level source code only (the default), this button advances execution to the next line of source, like the **step** command. On the other hand, if the program-display panel shows assembler instructions (when either Assembly or Mixed selected from the Source menu, or execution is in a routine compiled with no debugging symbols), this button advances execution to the next instruction—the equivalent of the **stepi** or **si** command.

Steps over a function call. This is a variant of the  button: instead of stepping to the very next statement executed (which, in the case of a function call, is typically not the next statement displayed), this button steps to the next line on the screen. If there is no intervening function call, this is the same thing as the  button. But if there is a function call, the  button executes that function in its entirety, then stops at the line after the function call.

The display style has the same effect as with the  button: thus, the  button steps to either the next machine instruction or the next source statement, if necessary completing a subroutine call first.

Continues program execution. Click this button to return control to the task you are debugging, rather than operating it manually from the debugger after a suspension. If there are no remaining breakpoints, interrupts, or signals, the task runs to completion.

To continue only until the program reaches a particular line in your program, drag this icon (using the middle mouse button) from the button bar to the line in the display panel where the program is to suspend once more. This has the same effect as dragging the context pointer, but is more convenient when you scroll the program-display window away from the current point of suspension.

This button issues the **continue** command.

Finishes the current subroutine. While stepping through a program, you may conclude that the problem you are interested in lies in the current subroutine's caller, rather than at the stack level where your task is suspended. Use this button in that situation: execution will continue until the current subroutine completes, then return control to the debugger in the calling statement.

This button issues the **finish** command.

Moves one level up the subroutine stack. The debugger usually has the same point of view as the executing program: in particular, what variable definitions are visible depends on the current subroutine. This button changes the context to the current subroutine's caller; press it again to get to that subroutine's caller, and so on.

This button does not change the location of the program counter; it only affects what data and symbols are visible to you. If you continue or step the program, it still takes up where it left off, regardless of whether you have used this button.

This button issues the **up** command, and has the same effect on a following **finish** or **until** command as **up**. The location of the temporary breakpoint that is set for **finish** or **until** depends on the *selected frame*, which is changed by **up**.

Moves one level down the stack. This is the converse of the  button, and like it, affects the data you can see but not the state of your program.

This button issues the **down** command, and has the same effect on a following **finish** or **until** command as **down**. The location of the temporary breakpoint that is set for **finish** or **until** depends on the *selected frame*, which is changed by **down**.

Calls up an editor (specified by your **EDITOR** environment variable—or **vi** if **EDITOR** is not defined) on the current source file. To specify the starting context, drag the editor button to a line in the region you wish to edit, using the middle button on your mouse.

Prints a symbol's value in the command panel. Begin by left-clicking on the symbol of interest, in the program-display panel; the debugger highlights the symbol. Then press this button to display its value.

This button issues the **print** command, and echoes the command and its output—the symbol value—to the command panel.

Prints the value at a pointer location. This button has a similar effect to the print button above, except that it de-references the selected symbol. Use this button to inspect data when you have a pointer to the data, rather than the data itself.

This button issues the **print *** command, and echoes the command and its output—the value at the selected pointer location—to the command panel.

Launches an inspection window that monitors a symbol's current value. This auxiliary display is automatically updated each time control returns to the debugger.

Several different kinds of data-inspection windows are available, depending on data structure; the debugger chooses the right one automatically.

Figure 9-11 shows the two simplest displays: for ordinary numeric data, and for a pointer. In both cases, the numeric value of the variable is displayed in a small independent window. The name of the variable being displayed appears next to the numeric value. The window's title bar also shows the name of the variable displayed, preceded by a parenthesized display number.[2]

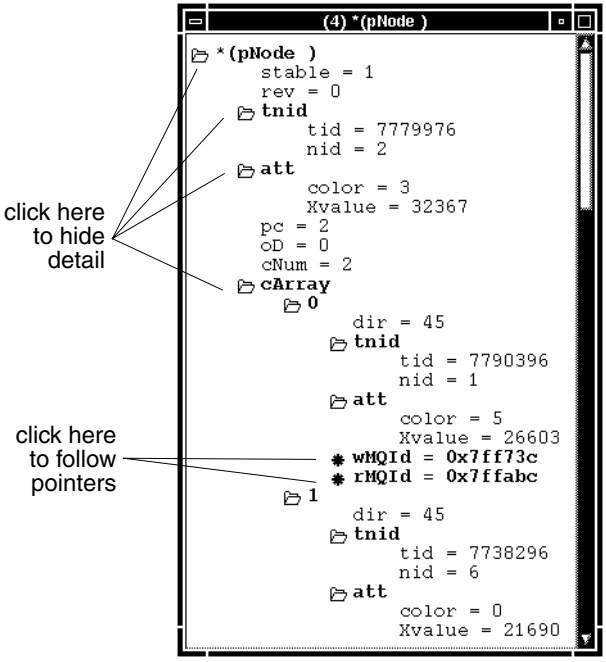
Figure 9-11 **Display Windows: Numeric and Pointer Data**

The debugger indicates whether the displayed variable is a pointer, by placing an asterisk to its left (as with **pNode** in Figure 9-11). To follow a pointer variable, click on its name in the display window; a new display pops up with the selected value.

If the displayed variable is a C **struct** (or a C++ **class** object), the debugger uses a special *structure browser* that exhibits the data structure graphically, using a folder icon to group nested structures. Figure 9-12 shows an example of a structure browser.

2. Display numbers are useful with the GDB commands **delete display**, **disable display**, and **info display**, which you can execute in the command panel. See *GDB User's Guide* for details.

Figure 9-12 **Display Window: Structure Browser**



You can click on any folder in a structure browser to hide data that is not of current interest (or to reveal it, once a folder is closed). You can also click on pointers (highlighted in bold type) to follow them; this provides a convenient way of exploring list values interactively.

Launches an inspection window on the value at a pointer location. When you want to see the contents of a pointer immediately, rather than going through a display of the pointer address, click this button rather than the previous one. the debugger displays one of the same set of windows described above.

Both this button and the previous one issue the **display /W** command. (The **/W** display format is a CrossWind enhancement. See *Graphically Enhanced Commands*, p.357 for more information.)

Defines a new button, or delete an existing one. You can add your own buttons (with text labels) to the button bar by clicking this button. The debugger displays a form where you can specify the name of the button,

as well as one or more debugger commands to execute when the button is pressed.

Figure 9-13 **New-Button Form**

To delete a button (whether a standard one, or a user-defined one), drag it (using the middle mouse button, as usual) to the button icon. Standard buttons come back the next time you start the debugger; user-defined buttons are similarly persistent if their definitions are recorded in your *homeDir***/.wind/crosswind.tcl** initialization file.

Figure 9-13 shows the new-button form. In this example, a button labeled Home is defined to execute the GDB **frame** command. Because the **frame** command controls context in the program-display panel, a button with this definition is a convenient way to get the display panel back to the location where your program is suspended, after scrolling elsewhere. After completing the form, press the OK box at the bottom; your new button appears at the right of the menu bar.

CrossWind automatically saves your button definition by writing Tcl code at the end of your *homeDir***/.wind/crosswind.tcl** initialization file. For example, the button definition above writes the following there:

```
toolBarItemCreate Home button {ttySend "frame\n"}
```

For buttons with more elaborate effects, consider first defining a new debugger command as described in *9.5.4 Tcl: Debugger Automation*, p.367; then you can hook up the new command to a new button. You can also attach buttons to commands resulting from the GDB **define** command (see *GDB User's Guide: Canned Sequences of Commands*).

For examples of how to record your own button definitions in a CrossWind initialization file, see *Tcl: "This" Buttons for C++*, p.377.

9.4.2 Debugger Command Panel: GDB

CrossWind is designed to provide graphical access to those debugger actions that are best controlled graphically, but also to exploit the command-line GDB interface when that is the best way to perform some particular action. For example, the housekeeping of getting subroutines started necessarily involves typing subroutine names and argument lists. So that you do not have to switch back and forth between menus, buttons, and dialogs or forms for commands of this sort, the debugger exploits the command panel, which is inherently best suited to commands with typed arguments. The command panel provides full access to the GDB command language described in the GDB manual, *GDB User's Guide*.

NOTE: As a convenience, the GDB command interpreter repeats the previous command when you press **ENTER** (or **RETURN**) on an empty line, except for a few commands where it would be dangerous or pointless. See *GDB User's Guide: GDB Commands* for more information. Press **ENTER** in the debugger only when you want to execute or repeat a command.

The following sections summarize some particularly useful commands, and describe commands added by Wind River that are not part of other versions of GDB.

GDB Initialization Files

One use of the command panel is to experiment with text-based commands for debugger actions that you might want to perform automatically.

When the debugger first executes GDB,[3] it looks for a file named **.gdbinit**: first in your home directory, then in your current working directory. If it finds the file in either directory, the debugger commands in it are executed; if it finds the file in both directories, the commands in both are executed.

A related initialization file under your home directory, called *homeDir*/**.wind/gdb.tcl**, is specifically intended for Tcl code to customize GDB with your own extensions written in Tcl. The Tcl code in this file executes before

3. The graphical interface to the debugger has a separate initialization file *homeDir*/**.wind/crosswind.tcl**, which runs after **.gdbinit**.

.gdbinit. See *9.5.4 Tcl: Debugger Automation*, p.367 for a discussion of extending GDB through Tcl. See also *9.6.1 Tcl: Debugger Initialization Files*, p.375 for a discussion of how the various CrossWind initialization files interact.

What Modules to Debug

You can use the following commands to establish the debugging context:

add-symbol-file *filename*
Specifies an object file on the host for the debugging session.

When the module you want to debug is already on the target (whether linked there statically, or downloaded by another Tornado tool), the debugger attempts to locate the corresponding object code on the host by querying the target server for the original file name and location. However, many factors (such as differing mount points on host and target, symbolic links, virtual file systems, or simply moving a file after downloading it) often make it necessary to specify the object file explicitly; you can do so with the **add-symbol-file** command.

The debugger recognizes the abbreviation **add** for this command.

load *filename*
This command is equivalent to the Download command in the File menu. You may sometimes find it preferable to invoke the command from the command panel—for example, when you can use your window manager to cut and paste a complex pathname instead of iteratively applying a file selector.

load *filename serverFilename*
Use this version of the load command when the target server you are using is on a host with a different view of the file system from your CrossWind session. For example, in some complex networks different hosts may mount the same file at different points: you may want to download a file **/usr/fred/applic.o** which your target server on another host sees as **/fredshost/fred/applic.o**.[4]

Use the *serverFilename* argument to specify what file to download from the server's point of view. (You must also specify the *filename* argument from the local point of view for the benefit of the debugger itself.)

See *7.6 Object Module Load Path*, p.295 for a more extended discussion of the same problem in the context of the shell.

4. See also the description of **wtx-load-path-qualify** in *Extended Debugger Variables*, p.360 for another way of managing how the debugger reports **load** pathnames to the target server.

unload *filename*
Undo the effect of **load**: remove a dynamically linked file from the target, and delete its symbols from the debugger session.

The **load** and **unload** commands both request confirmation if the debugger is attached to an active task. You can disable this confirmation request, as well as all other debugger confirmation requests, with **set confirm**. See *GDB User's Guide: Controlling GDB*.

What Code to Display

After a debugging session is underway, the program-display panel keeps pace with execution: when the program hits a breakpoint, the corresponding source is centered on the display panel, and each time you step or continue program execution, the display scrolls accordingly.

When you begin your debugging session by attaching to an existing task, the display panel is filled immediately as a side effect of stopping the task. In other situations, it may be convenient to use one of the commands in this section for an initial display.

list *linespec*
Displays source code immediately in the program-display panel, with the display centered around the line you specify with the *linespec* argument. The most common forms for *linespec* are a routine name (which identifies the place where that subroutine begins executing) or a source-file line number in the form *filename***:***num* (the source file name, a colon, and a line number). For a full description of *linespec* formats, see *GDB User's Guide: Printing Source Lines*.

search *regexp*
Displays code centered around the first line (in the current source file) that contains the regular expression *regexp*, instead of specifying what line to display. The command **rev** is similar, but searches backwards from the current context. See *GDB User's Guide: Searching Source Files*.

break *fn*
Sets a breakpoint at *fn*. Instead of first displaying source code, then setting breakpoints using the graphical interface, you can set a breakpoint directly (if you know where to go!). The argument *fn* can be a function name or a line number. See *GDB User's Guide: Setting Breakpoints*.

The **break** command does not produce a display directly, but sets things up so that there is at least one place where your program suspends. You can use **run**

to start the program (except in system mode); when the program suspends at the breakpoint, the display panel shows the context.

Executing Your Program

Just as with the Tornado shell, you can execute any subroutine in your application from the debugger. Use the following commands:

run *routine args*
: This is the principal command used to begin execution under debugger control. Execution begins at *routine*; you can specify an argument list after the routine name, with the arguments separated by spaces. The argument list may not contain floating-point or double-precision values. (This command is not available in system mode; use the shell to get tasks started in that mode. See *7.2.7 Using the Shell for System Mode Debugging*, p.267.)

call *expr*
: If a task is already running (and suspended, so that the debugger has control), you can evaluate any source-language expression (including subroutine calls) with the **call** command. This provides a way of exploring the effects of other subroutines without abandoning the suspended call. Subroutine arguments in the expression *expr* may be of any type, including floating point or double precision.

When you run a routine from the debugger using one of these commands, the routine runs until it encounters a breakpoint or a signal, or until it completes execution. The normal practice is to set one or more breakpoints in contexts of interest before starting a routine. However, you can interrupt the running task by clicking on the interrupt button  or by keying the interrupt character (as set on your host with **stty intr**; usually **CTRL+C**) from the debugger window.

Application I/O

By default, any tasks you start with the **run** command use the standard I/O channels assigned globally in VxWorks. However, the debugger has the following mechanisms to specify input and output channels:

- **Redirection with < and >**

  Each time you use the **run** command, you can redirect I/O explicitly for that particular task by using **<** to redirect input and **>** to redirect output. For both input and output, ordinary pathnames refer to files or devices on the host

where the debugger is running, and pathnames preceded by an @ character refer to files or devices on the target. For example, the following command starts the routine **appMain( )** in a task that gets input from target device **/tyCo/0** and writes output to host device **/dev/ttyp2**:

```
(gdb) run appMain > /dev/ttyp2 < @/tyCo/0
```

- **New Default I/O with tty Command**

  The debugger command **tty** sets a new default input and output device for all future run commands in the debugging session. The same conventions used with explicit redirection on the **run** line allow you to specify host or target devices for I/O. For example, the following command sets the default input and output channels to host device **/dev/ttyp2**:

  ```
  (gdb) tty /dev/ttyp2
  ```

- **Tcl: Redirection of Global or Task-Specific I/O**

  Tcl extensions are available within the debugger's Tcl interpreter to redirect either all target I/O, or the I/O channels of any running task. See *9.5.7 Tcl: Invoking GDB Facilities*, p.370 for details.

Graphically Enhanced Commands

Certain GDB commands, even though typed in the command panel, are especially useful due to the CrossWind graphical presentation. Among these are **list**, **search**, and **rev**, discussed already in *What Code to Display*, p.355. The following commands are also especially useful because of CrossWind graphical extensions:

display /W *expr*

The i and i* buttons provide a convenient way to generate active displays of symbol and pointer values, allowing you to monitor important data as your application executes under debugger control. However, sometimes the most useful data to monitor is the result of an expression—something that does not appear in your program, and hence cannot be selected before clicking a button.

In this situation, you can use the CrossWind **/W** format with the GDB **display** command to request an inspection window from the command panel. Because you type the expression argument directly, you can use any source-language expression to specify the value to monitor. An inspection window appears, which behaves just like those generated with buttons (*CrossWind Buttons*, p.344).

frame *n*
Displays a summary of a stack frame, in the command panel. But it also has a useful side effect: it re-displays the code in the program-display panel, centered around the line of code corresponding to that stack frame.

Used without any arguments, this command provides a quick way of restoring the program-display panel context for the current stack frame, after you scroll to inspect some other region of code. Used with an argument *n* (a stack-frame number, or a stack-frame address), this command provides a quick way of looking at the source-code context elsewhere in the calling stack. For more information about stack frames in GDB and about the GDB **frame** command, see *GDB User's Guide: Examining the Stack*.

Managing Targets

Instead of using the Targets menu (*CrossWind Menus*, p.338), you can select a target from the command panel with the **target wtx** command. The two methods of selecting a target are interchangeable; however, it may sometimes be more convenient to use the GDB command language—for example, you might specify a target this way in your **.gdbinit** initialization file or in other debugger scripts.

target wtx *servername*
Connects to a target managed by the target server registered as *servername* in the Tornado registry, using the WTX protocol. Use this command regardless of whether your target is attached through a serial line or through an Ethernet connection; the target server subsumes such communication details. (See *2.7 Connecting a Tornado Target Server*, p.56.) There is no need to specify the full registered name as *servername*; any unique abbreviation (or any regular expression that uniquely specifies a server name) will do.

Command-Interaction Facilities

The following GDB facilities are designed to streamline command-line interaction:

– Command-line editing, using either **Emacs**-like (the default) or **vi**-like keystrokes, as described in *GDB User's Guide: Command Line Editing*. By default, command-line editing in the debugger is **Emacs**-like.[5] To make it more

5. There is one exception: the Meta key is not available, because it is reserved for keyboard shortcuts that select items from menus. Instead, use the **ESC** key as a Meta prefix, as usual in **Emacs** and related programs when no Meta key is available.

consistent with the WindSh **vi**-like editing facilities, write the following line in a file named **.inputrc** in your home directory:

```
set editing-mode vi
```

– Command history and history expansion in the style of the UNIX C shell, as described in *GDB User's Guide: Using History Interactively*.

– **TAB**-key completion of commands, program symbols, and file names, depending on context, as described in *GDB User's Guide: Command Completion*. (This is most useful with C++ symbols, where completion is supplemented by interactive menus to choose among overloaded symbol definitions.)

– Specialized commands to give information about the state of your program (**info** and its sub-commands), the state of the debugger (**show** and its sub-commands), and brief descriptions of available commands and their syntax (**help**; the same summaries are also available through GDB Online in the Help menu).

– **CTRL+L** clears the input and output displayed in the command panel.

Extended Debugger Commands

The command area also provides two kinds of extended commands:

- **Shell Commands**

  You can run any of the WindSh primitive facilities described in *7.2.4 Invoking Built-In Shell Routines*, p.253 in the command panel, by inserting the prefix "wind-" before the shell command name. For example, to run the shell **td( )** command from the debugger, enter **wind-td** in the command panel.

  Because of GDB naming conventions, mixed-case command names cannot be used; if the shell command you need has upper case characters, use lower case and insert a hyphen before the capital letter. For example, to run the **semShow( )** command, enter **wind-sem-show**.

CAUTION: The debugger does not include the shell's C interpreter; thus, the "wind-" commands are interfaces only to the underlying Tcl implementations of the shell primitives. For shell primitives that take no arguments, this makes no difference; but for shell primitives that require an argument, you must use the shell Tcl command **shSymAddr** to translate C symbols to the Tcl form. For example, to call the shell built-in **show( )** on a semaphore ID **mySemID**, use the following:

```
(gdb) wind-show [shSymAddr mySemId]
```

- **Server Protocol Requests**

  The Tornado tools use a protocol called WTX to communicate with the target server. You can send WTX protocol requests directly from the GDB command area as well, by using a family of commands beginning with the prefix "wtx-". See *Tornado API Programmer's Guide: WTX Protocol* for descriptions of WTX protocol requests. Convert protocol message names to lower case, and use hyphens in place of underbars; for example, issue the message **WTX_CONSOLE_CREATE** as the command **wtx-console-create**.

Extended Debugger Variables

You can change many details of the debugger's behavior by using the **set** command to establish alternative values for internal parameters. (The **show** command displays these values; you can list the full set interactively with **help set**.)

The following additional **set**/**show** parameters are available in CrossWind beyond those described in *GDB User's Guide*:

inhibit-gdbinit
Do not read the GDB-language initialization files *homeDir*/**.gdbinit** and **${PWD}/.gdbinit**, discussed in *9.6.1 Tcl: Debugger Initialization Files*, p.375. Default: **no** (that is, read initialization files).

wtx-ignore-exit-status
Whether or not to report the explicit exit status of a routine that exits under debugger control. When this parameter is **on** (the default), the debugger always reports completion of a routine with the message "Program terminated normally." If your application's routines use the exit status to convey information, set this parameter to **off** to see the explicit exit status as part of the termination message.

wtx-load-flags
Specifies the option flags for the dynamic loader (Download in the File menu, or **load** in the command panel). These flags are described in the discussion of **ld( )** in *VxWorks Programmer's Guide: Configuration and Build*. Default: **LOAD_GLOBAL_SYMBOLS** (4).

wtx-load-path-qualify
Controls whether the debugger translates a relative path specified in the **load** argument to an absolute path when instructing the target server to download a file. By default, this value is set to **yes**: this instructs the debugger to perform this translation, so that the target server can locate the file even if the server and the debugger have different working directories.

However, in some networks where the debugger and target server have different views of the file system, a relative pathname can be interpreted correctly by both programs even though the absolute pathname is different for the two. In this case, you may wish to set **wtx-load-path-qualify** to **no**.

wtx-load-timeout
As a safeguard against losing contact with the target during a download, the debugger uses a timeout controlled by this parameter. If a download does not complete in less time than is specified here (in seconds), the debugger reports an error. Default: 30 seconds. To reset this parameter to 120 seconds, use:

```
(gdb) set wtx-load-timeout 120
```

wtx-task-priority
Priority for transient VxWorks tasks spawned by the **run** command. Default: 100.

wtx-task-stack-size
Stack size for transient tasks spawned by the **run** command. Default: 20,000.

wtx-tool-name
The name supplied for the debugger session to the target server. This is the name reported in the launcher's list of tools attached to a target. Default: **crosswind**. If you often run multiple debugger sessions, you can use this parameter to give each session a distinct name.

9.5 System-Mode Debugging

By default, in CrossWind you debug only one task at a time. The task is selected either by using the **run** command to create a new task, or by using **attach** to debug an existing task. When the debugger is attached to a task, debugger commands affect only that particular task. For example, when a breakpoint is set it applies only to that task. When the task reaches a breakpoint, only that task stops, not the entire system. This form of debugging is called *task mode* debugging. (All the material in *9.4 CrossWind in Detail*, p.336 applies to task mode debugging).

Tornado also supports an alternate form of debugging, where you can switch among multiple tasks (also called *threads*) and even examine execution in system routines such as ISRs. This alternative mode is called *system mode* debugging; it is also sometimes called *external mode*.

Most of the debugger features described elsewhere in this manual, and the debugging commands described in *GDB User's Guide*, are available regardless of which debugging mode you select. However, certain debugging commands (discussed below in *9.5.2 Thread Facilities in System Mode*, p.363) are useful only in system mode.

NOTE: The **run** command is not available in system mode, because its use of a new subordinate task is more intrusive in that mode. In system mode, use the shell to start new tasks as discussed in *7.2.7 Using the Shell for System Mode Debugging*, p.267, then attach to them with the **thread** command.

9.5.1 Entering System Mode

To debug in system mode, first make sure your debugger session is not attached to any task (type the command **detach**, or select Detach from the Targets menu).

Then issue the following command:

attach system
Switches the target connection into system mode (if supported by the target agent) and stops the entire target system.

Or, select Target>Attach System from the CrossWind menu.

The response to a successful **attach system** is output similar to the following:

```
(gdb) attach system
Attaching to system.
0x5b58 in wdbSuspendSystemHere ()
```

Once in system mode, the entire target system stops. In the example above, the system stopped in **wdbSuspendSystemHere( )**, the normal suspension point after **attach system**.

CAUTION: Not all targets support system mode, because the BSP must include a special driver for that purpose (see *2.5 Host-Target Communication Configuration*, p.31). If your target does not support system mode, attempting to use **attach system** produces an error.

9.5.2 Thread Facilities in System Mode

In system mode, the GDB thread-debugging facilities become useful. A *thread* is the general term for processes with some independent context, but a shared address space. In VxWorks, each task is a thread; the system context (including ISRs and drivers) is also a thread. GDB identifies each thread with a *thread ID*, a single arbitrary number internal to the debugger.

You can use the following GDB commands to manage thread context.

info threads
Displays summary information (including thread ID) for every thread in the target system.

thread *idNo*
Selects the specified thread as the current thread.

break *linespec* **thread** *idNo*
Sets a breakpoint affecting only the specified thread.

For a general description of these commands, see *GDB User's Guide: Debugging Programs with Multiple Threads*. The sections below discuss the thread commands in the context of debugging a VxWorks target in system mode.

Displaying Summary Thread Information

The command **info threads** shows what thread ID corresponds to which VxWorks task. For example, immediately after executing **attach system** to stop a VxWorks target, the **info threads** display resembles the following:

```
(gdb) info threads
  4 task 0x4fc868   tExcTask    0x444f58 in ?? ()
  3 task 0x4f9f40   tLogTask    0x444f58 in ?? ()
```

```
  2 task 0x4c7380 + tNetTask    0x4151e0 in ?? ()
  1 task 0x4b0a24   tWdbTask    0x4184fe in ?? ()
(gdb)
```

In the **info threads** output, the left-most number on each line is the debugger's thread ID. The single asterisk at the left margin indicates which thread is the *current thread*. The current thread identifies the "most local" perspective: debugger commands that report task-specific information, such as **bt** and **info regs** (as well as the corresponding displays) apply only to the current thread.

The next two columns in the thread list show the VxWorks task ID and the task name; if the system context is shown, the single word **system** replaces both of these columns. The thread (either a task, or the system context) currently scheduled by the kernel is marked with a **+** to the right of the task identification.

CAUTION: The thread ID of the system thread is not constant. To identify the system thread at each suspension, you must use **info threads** whenever the debugger regains control, in order to see whether the system thread is present and, if so, what its ID is currently.

The remainder of each line in the **info threads** output shows a summary of each thread's current stack frame: the program counter value, and the corresponding function name.

The thread ID is required to specify a particular thread with commands such as **break** and **thread**.

Switching Threads Explicitly

To switch to a different thread (making that thread the current one for debugging, but without affecting kernel task scheduling), use the **thread** command. For example:

```
(gdb) thread 2
[Switching to task 0x3a4bd8   tShell    ]
#0  0x66454 in semBTake ()
(gdb) bt
#0  0x66454 in semBTake ()
#1  0x66980 in semTake ()
#2  0x63a50 in tyRead ()
#3  0x5b07c in iosRead ()
#4  0x5a050 in read ()
#5  0x997a8 in ledRead ()
#6  0x4a144 in execShell ()
#7  0x49fe4 in shell ()
(gdb) thread 3
```

```
[Switching to task 0x3aa9d8   tFtpdTask ]
#0  0x66454 in semBTake ()
(gdb) print/x $i0
$3 = 0x3bdb50
```

As in the display shown above, each time you switch threads the debugger exhibits the newly current thread's VxWorks task ID and task name.

Thread-Specific Breakpoints

In system mode, unqualified breakpoints (set with graphical controls on the program-display window, or in the command panel with the **break** command and a single argument) apply globally: any thread stops when it reaches such a breakpoint. You can also set thread-specific breakpoints, so that only one thread stops there.

To set a thread-specific breakpoint, append the word **thread** followed by a thread ID to the **break** command. For example:

```
(gdb) break printf thread 2
Breakpoint 1 at 0x568b8
(gdb) cont
Continuing.
[Switching to task 0x3a4bd8 + tShell    ]

Breakpoint 1, 0x568b8 in printf ()

(gdb) i th
  8 task 0x3b8ef0   tExcTask    0x9bfd0 in qJobGet ()
  7 task 0x3b6580   tLogTask    0x9bfd0 in qJobGet ()
  6 task 0x3b15b8   tNetTask    0x66454 in semBTake ()
  5 task 0x3ade80   tRlogind    0x66454 in semBTake ()
  4 task 0x3abf60   tTelnetd    0x66454 in semBTake ()
  3 task 0x3aa9d8   tFtpdTask   0x66454 in semBTake ()
* 2 task 0x3a4bd8 + tShell      0x568b8 in printf ()
  1 task 0x398688   tWdbTask    0x66454 in semBTake ()
(gdb) bt
#0  0x568b8 in printf ()
#1  0x4a108 in execShell ()
#2  0x49fe4 in shell ()
```

Internally, the debugger still gets control every time any thread encounters the breakpoint; but if the thread ID is not the one you specified with the **break** command, the debugger silently continues program execution without prompting you.

CAUTION: Because the thread ID for the system context is not constant, it is not possible to set a breakpoint specific to system context. The only way to stop when a breakpoint is encountered in system context is to use a non-task-specific breakpoint.

Switching Threads Implicitly

Your program may not always suspend in the thread you expect. If any breakpoint or other event (such as an exception) occurs while in system mode, in any thread, the debugger gets control. Whenever the target system is stopped, the debugger switches to the thread that was executing. If the new current thread is different from the previous value, a message beginning "Switching to" shows what thread suspended:

```
(gdb) thread 2
(gdb) cont
Continuing.
Interrupt...
Program received signal SIGINT, Interrupt.
[Switching to system +]

0x5b58 in wdbSuspendSystemHere ()
```

Whenever the debugger does not have control, you can interrupt the target system by clicking on the interrupt button  or by keying the interrupt character (usually **CTRL+C**). This usually suspends the target in the system thread rather than in any task.

When you step program execution (with any of the commands **step**, **stepi, next**, or **nexti**, or the equivalent buttons  or ), the target resumes execution where it left off, in the thread marked with **+** in the **info threads** display. However, in the course of stepping that thread, other threads may begin executing. Hence, the debugger may stop in another thread before the stepping command completes, due to an event in that other thread.

9.5.3 Configuring VxWorks for System Mode Debugging

In order for system mode debugging to work properly, the items in Table 9-3 must be set correctly.

Table 9-3 **Definitions for System Mode Debugging**

# Define	Required Value
#define WDB_COMM_TYPE	**WDB_SERIAL**
#define WDB_MODE	**WDB_MODE_DUAL**
#define INCLUDE_PC_CONSOLE	
#define PC_CONSOLE	
#define CONSOLE_TTY	
#define NUM_TTY	
#define WDB_TTY_CHANNEL	
#define WDB_TTY_DEV_NAME	
#define WDB_TTY_BAUD	As appropriate.

9.5.4 Tcl: Debugger Automation

CrossWind exploits Tcl at two levels: like other Tornado tools, it uses Tcl to build the graphical interface, but it also includes a Tcl interpreter at the GDB command level. This section discusses using the Tcl interpreter inside the CrossWind enhanced GDB, at the command level.

NOTE: For information about using Tcl to customize the CrossWind GUI, see *9.6 Tcl: CrossWind Customization*, p.374. The discussion in this section is mainly of interest when you need complex debugger macros; you might want to skip this section on first reading.

Tcl has two major advantages over the other GDB macro facility (the **define** command). First, Tcl provides control and looping (such as **for**, **foreach**, **while**, and **case**). Second, Tcl procedures can take parameters. Tcl, building on the command interface, extends the scripting facility of GDB to allow you to create new commands.

9.5.5 Tcl: A Simple Debugger Example

To submit commands to the Tcl interpreter within GDB from the command panel, use the **tcl** command. For example:

```
(gdb) tcl info tclversion
```

This command reports which version of Tcl is integrated with GDB. All the text passed as arguments to the **tcl** command (in this example, **info tclversion**) is provided to the Tcl interpreter exactly as typed. Convenience variables (described in *GDB User's Guide: Convenience Variables*) are not expanded by GDB. However, Tcl scripts can force GDB to evaluate their arguments; see *9.5.7 Tcl: Invoking GDB Facilities*, p.370.

You can also define Tcl procedures from the GDB command line. The following example procedure, **mld**, calls the **load** command for each file in a list:

```
(gdb) tcl proc mload args {foreach obj $args {gdb load $obj}}
```

You can run the new procedure from the GDB command line; for example:

```
(gdb) tcl mload vxColor.o priTst.o
```

To avoid typing **tcl** every time, use the **tclproc** command to assign a new GDB command name to the Tcl procedure. For example:

```
(gdb) tclproc mld mload
```

This command creates a new GDB command, **mld**. Now, instead of typing **tcl mload**, you can run **mld** as follows:

```
(gdb) mld vxColor.o priTst.o
```

You can collect Tcl procedures in a file, and load them into the GDB Tcl interpreter with this command:

```
(gdb) tcl source tclFile
```

If you develop a collection of Tcl procedures that you want to make available automatically in all your debugging sessions, write them in the file *homeDir*/**.wind/gdb.tcl** under your home directory. The GDB Tcl interpreter reads this file when it begins executing. (See *9.6.1 Tcl: Debugger Initialization Files*, p.375 for a discussion of how all the CrossWind and GDB initialization files interact.)

9.5.6 Tcl: Specialized GDB Commands

The CrossWind version of GDB includes four commands to help you use Tcl. The first two were discussed in the previous section. The commands are:

tcl *command*
Passes the remainder of the command line to the Tcl interpreter, without attempting to evaluate any of the text as a GDB command.

tclproc *gdbName TclName*
Creates a GDB command *gdbName* that corresponds to a Tcl procedure name *TclName*. GDB does not evaluate the arguments when *gdbName* is invoked; it passes them to the named Tcl procedure just as they were entered.

NOTE: To execute **tclproc** commands automatically when GDB begins executing, you can place them in **.gdbinit** directly (see *GDB Initialization Files*, p.353), because **tclproc** is a GDB command rather than a Tcl command. However, if you want to keep the **tclproc** definition together with supporting Tcl code, you can exploit the **gdb** Tcl extension described in *9.5.7 Tcl: Invoking GDB Facilities*, p.370 to call **gdb tclproc** in *homeDir***/.wind/gdb.tcl**.

tcldebug
Toggles Tcl debugging mode. Helps debug Tcl scripts that use GDB facilities. When Tcl debugging is ON, all GDB commands or other GDB queries made by the Tcl interpreter are printed.

tclerror
Toggles Tcl verbose error printing, to help debug Tcl scripts. When verbose error mode is ON, the entire stack of error information maintained by the Tcl interpreter appears when a Tcl error occurs that is not caught. Otherwise, when verbose error mode is OFF, only the innermost error message is printed. For example:

```
(gdb) tcl puts stdout [expr $x+2]
can't read "x": no such variable

(gdb) tclerror
TCL verbose error reporting is ON.

(gdb) tcl puts stdout [expr $x+2]
can't read "x": no such variable
    while executing
"expr $x..."
    invoked from within
"puts stdout [expr $x..."
```

Tcl also stores the error stack in a global variable, **errorInfo**. To see the error stack when Tcl verbose error mode is OFF, examine this variable as follows:

```
(gdb) tcl $errorInfo
```

For more information about error handling in Tcl, see *C.2.9 Tcl Error Handling*, p.448.

9.5.7 Tcl: Invoking GDB Facilities

You can access GDB facilities from Tcl scripts with the following Tcl extensions:

gdb *arguments*
Executes a GDB command (the converse of the GDB **tcl** command). Tcl evaluates the arguments, performing all applicable substitutions, then combines them (separated by spaces) into one string, which is passed to GDB's internal command interpreter for execution.

If the GDB command produces output, it is shown in the command panel.

If Tcl debugging is enabled (with **tcldebug**), the following message is printed:

```
execute: command
```

If the GDB command causes an error, the Tcl procedure **gdb** signals a Tcl error, which causes unwinding if not caught (for information about unwinding, see *C.2.9 Tcl Error Handling*, p.448).

gdbEvalScalar *exprlist*
Evaluates a list of expressions *exprlist* and returns a list of single integer values (in hexadecimal), one for each element of *exprlist*.[6] If an expression represents a scalar value (such as **int**, **long**, or **char**), that value is returned. If an expression represents a **float** or **double**, the fractional part is truncated. If an expression represents an aggregate type, such as a structure or array, the address of the indicated object is returned. Standard rules for Tcl argument evaluation apply.

6. A more restricted form of this command, called **gdbEvalAddress**, can only evaluate a single expression (constructed by concatenating all its arguments). **gdbEvalAddress** is only supported to provide compatibility with Tcl debugger extensions written for an older debugger, VxGDB. Use the more general **gdbEvalScalar** in new Tcl extensions.

If Tcl debugging is enabled, the following message is printed for each expression:

```
evaluate: expression
```

If an expression does not evaluate to an object that can be cast to pointer type, an error message is printed, and **gdbEvalScalar** signals a Tcl error, which unwinds the Tcl stack if not caught (see *C.2.9 Tcl Error Handling*, p.448 for information about unwinding).

gdbFileAddrInfo *fileName*
Returns a Tcl list with four elements: the first source line number of *fileName* that corresponds to generated object code, the last such line number, the lowest object-code address from *fileName* in the target, and the highest object-code address from *fileName* in the target. The argument *fileName* must be the source file (**.c**, not **.o**) corresponding to code loaded in the target and in the debugger.

For example:

```
(gdb) tcl gdbFileAddrInfo vxColor.c
{239 1058 0x39e2d0 0x39fbfc}
```

gdbFileLineInfo *fileName*
Returns a Tcl list with as many elements as there are source lines of *fileName* that correspond to generated object code. Each element of the list is itself a list with three elements: the source-file line number, the beginning address of object code for that line, and the ending address of object code for that line. The argument *fileName* must be the source file (**.c**, not **.o**) of a file corresponding to code loaded in the target and in the debugger.

For example:

```
(gdb) tcl gdbFileLineInfo vxColor.c
{239 0x39e2d0 0x39e2d4} {244 0x39e2d4 0x39e2ec} ...
```

gdbIORedirect *inFile outFile* [*taskId*]
Redirect target input to the file or device *inFile*, and target output and error streams to the file or device *outFile*. If *taskId* is specified, redirect input and output only for that task; otherwise, redirect global input and output. To leave either input or output unchanged, specify the corresponding argument as a dash (**-**). Ordinary pathnames indicate host files or devices; arguments with an **@** prefix indicate target files or devices. For target files, you may specify either a path name or a numeric file descriptor.

For example, the following command redirects all target output (including **stderr**) to host device **/dev/ttyp2**:

```
(gdb) tcl gdbIORedirect - /dev/ttyp2
```

The following command redirects input from task 0x3b7c7c to host device **/dev/ttyp2**, and output from the same task to target file descriptor 13:

```
(gdb) tcl gdbIORedirect /dev/ttyp2 @13 0x3b7c7c
```

gdbIOClose
Close all file descriptors opened on the host by the most recent **gdbIoRedirect** call.

gdbLocalsTags
Returns a list of names of local symbols for the current stack frame.

gdbStackFrameTags
Returns a list of names of the routines currently on the stack.

gdbSymbol *integer*
Translates *integer*, interpreted as a target address, into an offset from the nearest target symbol. The display has the following format:

symbolName **[** ± *Offset* **]**

Offset is a decimal integer. If *Offset* is zero, it is not printed. For example:

```
(gdb) tcl puts stdout [gdbSymbol 0x20000]
floatInit+2276
```

If Tcl debugging is on, **gdbSymbol** prints the following message:

```
symbol: value
```

gdbSymbolExists *symbolName*
Returns 1 if the specified symbol exists in any loaded symbol table, or 0 if not. You can use this command to test for the presence of a symbol without generating error messages from GDB if the symbol does not exist. This procedure cannot signal a Tcl error.

When Tcl debugging is on, **gdbSymbolExists** prints a message like the following:

```
symbol exists: symbolName
```

9.5.8 Tcl: A Linked-List Traversal Macro

This section shows a Tcl procedure to traverse a linked list, printing information about each node.[7] The example is tailored to a list where each node has the following structure:

```
struct node
    {
    int data;
    struct node *next;
    }
```

A common method of list traversal in C is a **for** loop like the following:

```
for (pNode = pHead; pNode; pNode = pNode->next)
...
```


We imitate this code in Tcl, with the important difference that all Tcl data is in strings, not pointers.

The argument to the Tcl procedure will be an expression (called **head** in our procedure) representing the first node of the list.

Use **gdbEvalScalar** to convert the GDB expression for a pointer into a Tcl string:

```
set pNode [gdbEvalScalar "$head"]
```

To get the pointer to the next element in the list:

```
set pNode [gdbEvalScalar "( (struct node *) $pNode)->next"]
```

Putting these lines together into a Tcl **for** loop, the procedure (in a form suitable for a Tcl script file) looks like the following:

```
proc traverse head {
    for {set pNode [gdbEvalScalar "$head"] }    \
        {$pNode}   \
        {set pNode [gdbEvalScalar "( (struct node *)$pNode)->next"]} \
            {puts stdout $pNode}
}
```

In the body of the loop, the Tcl command **puts** prints the address of the node.

To type the procedure directly into the command panel would require prefacing the text above with the **tcl** command, and would require additional backslashes (one at the end of every line).

---

7. Remember, though, that for interactive exploration of a list the structure browser (Figure 9-12) described in *CrossWind Buttons*, p.344 is probably more convenient.

If **pList** is a variable of type **(struct *node)**, you can execute:

```
(gdb) tcl traverse pList
```

The procedure displays the address of each node in the list. For a list with two elements, the output would look something like the following:

```
0xffeb00
0xffea2c
```

It might be more useful to redefine the procedure body to print out the integer member **data**, instead. For example, replace the last line with the following:

```
{puts stdout [format "data = %d" \
    [gdbEvalScalar "((struct node *) $pNode)->data"]]}
```

You can bind a new GDB command to this Tcl procedure by using **tclproc** (typically, in the same Tcl script file as the procedure definition):

```
tclproc traverse traverse
```

The **traverse** command can be abbreviated, like any GDB command. With these definitions, you can type the following command:

```
(gdb) trav pList
```

The output now exhibits the contents of each node in the list:

```
data = 1
data = 2
```

9.6 Tcl: CrossWind Customization

Like every other Tornado tool, the CrossWind graphical user interface is "soft" (amenable to customization) because it is written in Tcl, which is an interpreted language. The online *Tornado API Reference* describes the graphical building blocks available; you can also study the Tcl implementation of CrossWind itself. You can find the source in **host/resource/tcl/CrossWind.tcl**.

9.6.1 Tcl: Debugger Initialization Files

You can write Tcl code to customize the debugger's graphical presentation in a file called *homeDir*/**.wind/crosswind.tcl**. Use this file to collect your custom modifications, or to incorporate shared customizations from a central repository of Tcl extensions at your site.

Recall that the debugger uses two separate Tcl interpreters. Previous sections described the **.gdbinit** and *homeDir*/**.wind/gdb.tcl** initialization files that initialize the debugger command language (see *9.5.4 Tcl: Debugger Automation*, p.367).

The following outline summarizes the role of all the CrossWind customization files. The files are listed in the order in which they execute.

installDir/**.wind/gdb.tcl**
: Use this file to customize the Tcl interpreter built into GDB itself (for example, to define Tcl procedures for new GDB commands). This file is unique to the CrossWind version of GDB. When issuing commands intended for GDB, you must prepend them with **gdb**.

homeDir/**.gdbinit**
: Use this file for any initialization you want to perform in GDB's command language rather than in Tcl. This file is not unique to CrossWind; it is shared by any other GDB configuration you may install.

${PWD}/.gdbinit
: Akin to the **.gdbinit** in your home directory, this file also contains commands in GDB's command language, and is not unique to the CrossWind configuration of GDB. However, this file is specific to a particular working directory; thus it may be an appropriate place to record application-specific debugger settings.

homeDir/**.wind/crosswind.tcl**
: Use this file to customize the debugger's graphical presentation, using Tcl: for example, to define new buttons or menu commands. This file is unique to the CrossWind version of GDB.

You can prevent CrossWind from looking for the two **.gdbinit** files, if you choose, by setting the internal GDB parameter **inhibit-gdbinit** to **yes**. Because the initialization files execute in the order they are listed above, you have the opportunity to set this parameter before the debugger reads either **.gdbinit** file. To do this, insert the following line in your *homeDir*/**.wind/gdb.tcl**:

```
gdb set inhibit-gdbinit yes
```

9.6.2 Tcl: Passing Control between the Two CrossWind Interpreters

You can use the following specialized Tcl commands to pass control between the two CrossWind Tcl interpreters.

uptcl
: From the Tcl interpreter integrated with the GDB command parser, **uptcl** executes the remainder of the line in the CrossWind graphical-interface Tcl interpreter. **uptcl** does not return a result.

downtcl
: From the graphical-interface layer, **downtcl** executes the remainder of the line in the Tcl interpreter integrated with GDB. The result of **downtcl** is whatever GDB output the command generates. Use **downtcl** rather than **ttySend** if your goal is to capture the result for presentation in the graphical layer.

ttySend
: From the graphical-interface layer, **ttySend** passes its string argument to GDB, exactly as if you had typed the argument in the command panel. A newline is not assumed; if you are writing a command and want it to be executed, include the newline character (**\n**) at the end of the string. Use **ttySend** rather than **downtcl** if your goal is to make information appear in the command panel (this can be useful for providing information to other GDB prompts besides the command prompt).

The major use of **uptcl** is to experiment with customizing or extending the graphical interface. For example, if you have a file **myXWind** containing experimental Tcl code for extending the interface, you can try it out by entering the following in the command panel:

```
(gdb) tcl uptcl source myXWind
```

By contrast, **downtcl** and **ttySend** are likely to be embedded in Tcl procedures, because (in conjunction with the commands discussed in *9.5.7 Tcl: Invoking GDB Facilities*, p.370) they are the path to debugger functionality from the graphical front end.

Most of the examples in *9.6.3 Tcl: Experimenting with CrossWind Extensions*, p.377, below, center around calls to **downtcl**.

9.6.3 Tcl: Experimenting with CrossWind Extensions

The examples in this section use the Tcl extensions summarized in Table 9-4. For detailed descriptions of these and other Tornado graphical building blocks in Tcl, see the online *Tornado API Reference*.

Table 9-4 **Tornado UI Tcl Extensions Used in Example 9-2.**

Tcl Extension	Description
dialogCreate	Define the layout of a form (dialog box). Includes a list of all graphical controls (such as buttons, text boxes, lists). The description of each control ends with the name of a Tcl callback used when the control is acted on.
dialogPost	Display or update a named form (dialog).
dialogUnpost	Remove a form (dialog) from the screen.
dialogGetValue	Report the current value of a dialog graphical element (the contents of a text box, or the current selection in a list).
noticePost	Display a popup notice or a file selector.
menuButtonCreate	Add a command to an existing menu.
toolBarItemCreate … button	Add a new button (and associated command string) to the button bar.
toolBarItemCreate … space	Add space before new buttons in the button bar.

Tcl: "This" Buttons for C++

In C++ programs, one particular named value has great special interest: **this**, which is a pointer to the object where the currently executing function is a member.

Example 9-1 defines two buttons related to **this**:

– A [t] button, akin to the [p] button, to display the address of **this** in the command panel.

– A [t*] button, akin to the [p*] button, to launch a dedicated window that monitors the value where **this** points.

The Tcl primitive **catch** is used in the second button definition in order to avoid propagating error conditions (for instance, if the buttons are pressed with no code

loaded) from GDB back to the controlling CrossWind session. This does not prevent GDB from issuing the appropriate error messages to the command panel.

Example 9-1 **Buttons for C++ this Pointer**

```
# Make a nice gap before new buttons

toolBarItemCreate " " space

# BUTTON: "t"     Print C++ "this" value.

toolBarItemCreate " t " button {
    ttySend "print this\n"
}

# BUTTON: "t*"    Launch "inspect" window on current C++ class (*this)

toolBarItemCreate " t*" button {
    catch {downtcl gdb display/W *this}
}
```

Tcl: A List Command for the File Menu

Example 9-2 illustrates how to add extensions to the CrossWind graphical interface with a simple enhancement: adding a menu command to list the displayed program source centered on a particular line.

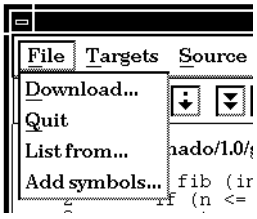
In Example 9-2, the procedure **xwindList** uses **downtcl** to run the GDB **list** command. To tie this into the graphical interface, the example adds a new command List from... to the File menu. The new command displays a form (described in the **dialogCreate** call) to collect input specifying an argument to the **list** command. When input is complete, the form in turn runs **xwindList**, through a call-back attached to its OK button. Figure 9-14 shows the new menu command and form defined here (and the Example 9-3 menu command).

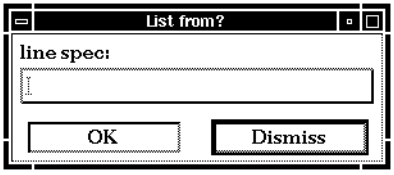
Example 9-2 **List Command**

```
# FORM: a form to prompt for list argument
# (part of "List from..." command addition to "File" menu)

dialogCreate "List from?" -size 290 100 {
    {text "line spec:" -hspan}
    {button "OK" -left 2 -right 48 -bottom .+5 xwindList}
    {button "Dismiss" -left 52 -right 98 -bottom .+5
    {dialogUnpost "List from?"}}
}
```

```
# MENU COMMAND: "List", additional entry under "File"

menuButtonCreate File "List from..." S {
    dialogPost "List from?"
}

##############################################################################
#
#
# xwindList - procedure for "List" command in CrossWind "File" menu
#
# This procedure sends a "list" command to GDB.  It is intended to be
# called from the "List from?" dialog (posted by the "Display" command
# in the CrossWind "File" menu).  Do not call it from other contexts;
# it interacts with the dialog.
#
# SYNOPSIS:
#   xwindList
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc xwindList {} {
    set lspec [dialogGetValue "List from?" "line spec:"]
    catch {downtcl gdb list $lspec}

    # gdb does not send back error indication,
    # but it does display any errors in command panel

    dialogUnpost "List from?"
}
```

Figure 9-14 **List Menu Command and Form Defined in Example 9-2**





Tcl: An Add-Symbols Command for the File Menu

As explained in *What Modules to Debug*, p.354, you sometimes need to tell the debugger explicitly to load symbols for modules that were downloaded to the target using other programs (such as the shell).

Example 9-3 illustrates a File menu command Add Symbols to handle this through the graphical user interface, instead of typing the **add-symbol-file** command.

Example 9-3 **Add-Symbols Command**

```
# MENU COMMAND: "Add Symbols", additional entry under "File"

menuButtonCreate File "Add Symbols..." S { xwindAddSyms }

##############################################################################
#
#
# xwindAddSyms - called from File menu to add symbols from chosen object file
#
# This routine implements the "Add Symbols" command in the File menu.
# It prompts the user for a filename; if the user selects one, it tells
# GDB to load symbols from that file.
#
# SYNOPSIS:
#   xwindAddSyms
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc xwindAddSyms {} {
    set result [noticePost fileselect "Symbols from file" Add "*.\[o|out\]"]
    if {$result != ""} {

        # we violate good taste here by not capturing or testing the result
        # of catch, because GDB poats an error message in the command panel
        # when the file cannot be loaded.

        catch {downtcl gdb add-symbol-file $result}
    }
}
```

10

Building VxDCOM Applications

10.1 Introduction

This chapter describes the step-by-step process of creating and building VxDCOM applications using Tornado. To more easily create VxDCOM applications, the basic VxDCOM support includes the following tools:

an application wizard
: Lets you easily generate skeleton code for a basic VxDCOM application, without having to define CoClasses and interfaces in IDL (the Interface Definition Language).

a C++ template class library
: Facilitates writing client and server implementation code.

an IDL compiler
: Compiles IDL file, generating the necessary proxy/stub and header files required by VxDCOM.

For a detailed description of the VxDCOM technology, see the *VxWorks Programmer's Guide: VxDCOM Applications*.

10.2 The VxDCOM Development Process

VxDCOM clients and servers for VxWorks can be created either as bootable or downloadable applications. The following step-by-step overview summarizes the VxDCOM development process:

Step 1: Create a Bootable Image with VxDCOM Support

Build a VxWorks bootable image with VxDCOM support components. You will need this image whether you are creating a bootable or a downloadable application. This step is covered in *Configuring a VxDCOM Bootable Image* on p. 383.

Step 2: Configure Any DCOM Component Parameters

Optionally configure the parameters for the DCOM components. This step is covered in *Configuring the DCOM Parameters* on p. 384.

Step 3: Generate Skeleton Project Files with the VxDCOM Wizard

Run the VxDCOM wizard from the command line. Use the wizard to define the CoClass and interfaces, to choose the server model and client skeleton program. This generates skeleton files for header prototypes, coclass definitions, interface and library definitions, and so on. These steps are described in *Using the VxDCOM Wizard* on p. 385.

Step 4: Implement the Server and Client

Complete the implementation of the server by editing the CoClass files to implement the interface methods. These files also contains (auto-generated) code to auto-register the server. This step is covered in the *Implementing the Server and Client on p. 396*.

Step 5: Add the Files to Project and Build It

Choose a bootable or downloadable application model. You can use either the project facility or the **makefile**, generated by the wizard, to build your application. Build and link the application. This step is covered in *Building and Linking the Application* on p. 397.

(a) Build the application.
(b) Ensure that it is correctly linked with proxy/stub code.
(c) Build the client program, if your project includes one.

Step 6: Register and Deploy Your VxDCOM Application

Deploy your application by registering the type library and setting the proper configurations for server authentication. These steps, listed below, are described in *Registering, Deploying, and Running Your Application* on p. 398.

(a) Register any necessary proxy DLLs on Windows.
(b) Register the type library.
(c) Register the server.
(d) Authenticate the server.
(e) Run the application and activate the server.

10.3 Configuring a VxDCOM Bootable Image

Whether you choose to create a bootable or downloadable client or server, you need a VxWorks bootable image that contains a kernel and VxDCOM support. You need such an image for all VxDCOM applications.

If you are creating a bootable application, add your VxDCOM application files to the bootable system image. If you are creating a downloadable application, add your VxDCOM files to this downloadable module. Then download that module to the bootable system containing the VxDCOM support.

10.3.1 Adding VxDCOM Component Support

After you have created the kernel, add the appropriate VxDCOM support components. Some components are required for all VxDCOM applications, some are required only for DCOM, and some are optional, providing additional functionality. This section describes VxDCOM support and when to add it to your kernel.

COM Core Component

This component is called **COM_CORE** and provides support for C COM projects as well as C++ COM and DCOM projects.

COM Support Component

The **COM** support component is required for all C++ COM and DCOM applications.

DCOM Support Components

The **DCOM** and **DCOM_PROXY** support components are required for DCOM applications. The **DCOM** component provides support for distributed COM. The **DCOM_PROXY** component provides support for proxy/stub code. Both of these components require the basic COM support, described above.

OPC Program Support

The **DCOM_OPC** support component is required if you are writing your own OPC server. If you are using the **VxOPC** product it is not required. For more information, see the *VxWorks Programmer's Guide: VxDCOM Applications*.

ComShow Routines Support

The **COM_SHOW** support component is required only if you want to use the COM show routines. Including this component adds diagnostic routines that may be used to interrogate the registry held within the VxWorks run-time.

DCOMShow Routines Support

The **DCOM_SHOW** component is required only if you want to debug the VxDCOM wire protocol. Including this component adds a significant overhead; therefore, it should only be used at the request of Wind River Customer Support to provide debug information. It should not be shipped as part of a production system.

10.3.2 Configuring the DCOM Parameters

If your application includes DCOM, you can optionally configure parameters for this component. Change the parameter values from within the project facility by selecting Properties from the popup menu on the appropriate binary component.

The DCOM parameter descriptions, their default values, and the value ranges are described in *VxWorks Programmer's Guide: VxDCOM Applications*.

10.4 Using the VxDCOM Wizard

The VxDCOM Wizard lets you generate a completely new VxDCOM project, or import an existing COM or DCOM server. To run the wizard, type at the command line:

```
% installDir/host/hostType/bin/torVars.bat
% comwizard projDir
```


where the *projDir* is the directory for your project. The files generated by the wizard are generated in this directory. The directory name you type in is the default name for your CoClass for new projects. You have an option to modify this name in the wizard.

10.4.1 Choosing the Project Type

The first page of the VxDCOM wizard, shown in Figure 10-1,lets you choose the project type.

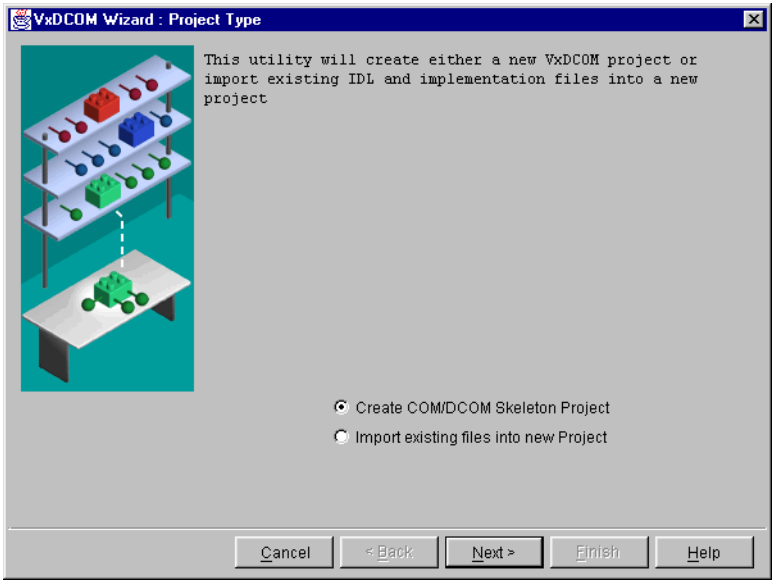
The options are:

Create COM/DCOM Skeleton Project
: Create skeleton files for a new COM or DCOM component. For details, see *Creating a COM/DCOM Skeleton Project* on p. 386.

Import existing files into new Project
: Create a project using existing COM application code. For details, see *Importing Existing Files into a New Project* on p. 392.

Choose the type of project and click Next.

Figure 10-1 **Choosing the VxDCOM Project Type**



10.4.2 Creating a COM/DCOM Skeleton Project

If you are creating a new project, you simply specify your server type and implementation language, and define your CoClass and interfaces using the wizard GUI. You do not have to write then in IDL (Interface Definition Language); nor do you have to write proxy/stub code.

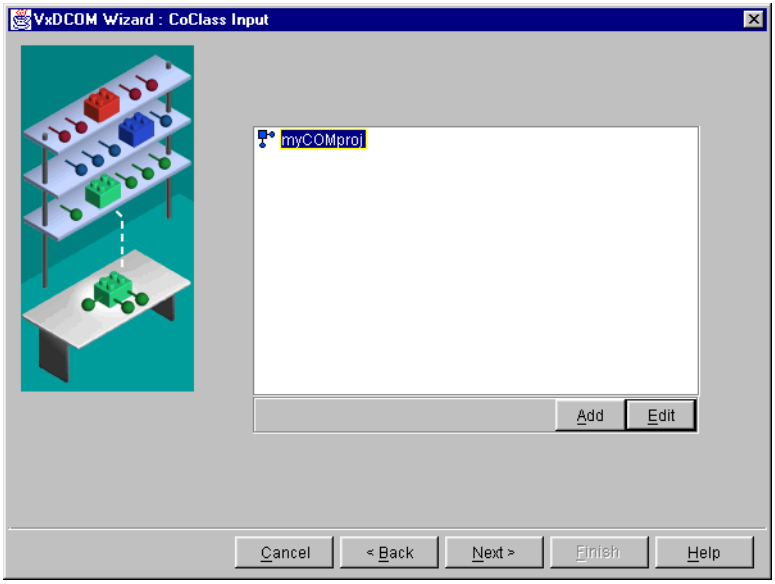
From the information you enter, the wizard generates the primary IDL definition file and the additional files appropriate to the server model and client program selected. These output files are described in *The Generated Output* on p. 393.

Defining the CoClass

Figure 10-2 shows the CoClass Input dialog of the wizard. From this dialog you define the CoClass by adding interface methods and parameters. This dialog defaults to a CoClass named for the argument you passed to **comwizard**.

You can modify the name of the CoClass (or any items), by highlighting the item and choosing Edit.

Figure 10-2 **Defining the CoClass**



To define the CoClass, you add interfaces and interface methods, and you specify parameter types for those methods. You typically do this in steps:

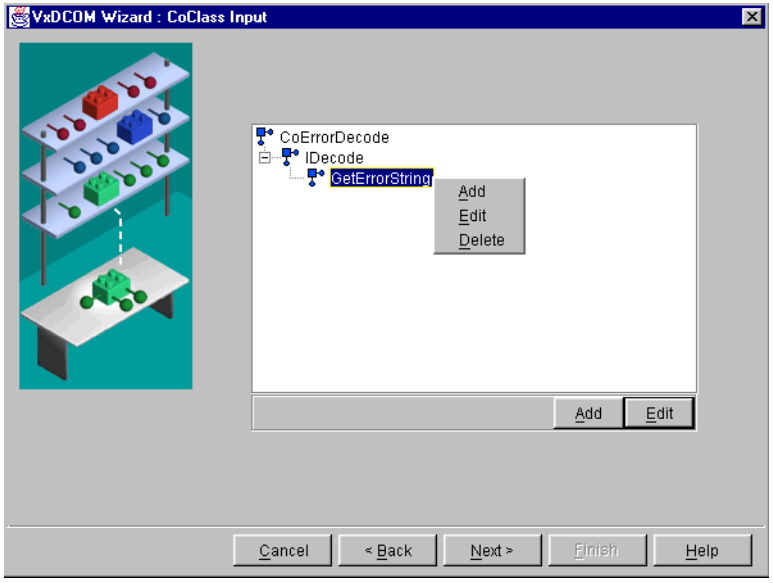
- Add one or more interfaces to the CoClass.
- For each interface, add one or more methods.
- For each method, add one or more parameters, specifying the attribute and type of each.

When your CoClass definition is complete, click Next.

Adding Items

To add an item, highlight the item and choose Add, as shown in Figure 10-3. This opens the appropriate Add dialog. These dialogs let you enter names for new interfaces, interface methods, or interface method parameters.

Figure 10-3 **Adding Interfaces**



Adding Method Parameters

When adding interface method parameters, you must also specify the attribute and type of each parameter. As shown in Figure 10-4, edit boxes for the parameter attribute and type appear with defaults. To change these, select the correct option from the dropdown listboxes. Once you click Apply, your selections will appear in the CoClass definition.

- **Method Parameter Attributes.** When selecting interface method parameters, all **[out]** parameters and **[out]** parameter combinations must be pointers. The attribute options for these parameters are part of the IDL language and are documented in the IDL reference section of the *VxWorks Programmer's Guide: VxDCOM Applications*.
- **Method Parameter Types.** When selecting the data type of the interface method parameter, the dropdown listbox displays a list of automation data types, as shown in Figure 10-5. Using automation data types provides built-in marshaling support under DCOM.

It is also possible to use non-automation types, however these are not available from the VxDCOM wizard dialog and may require additional linking. If you need to use non-automation data types, see the *Adding Non-Automation Types*, p.393.

Figure 10-4 **Adding Method Parameters**

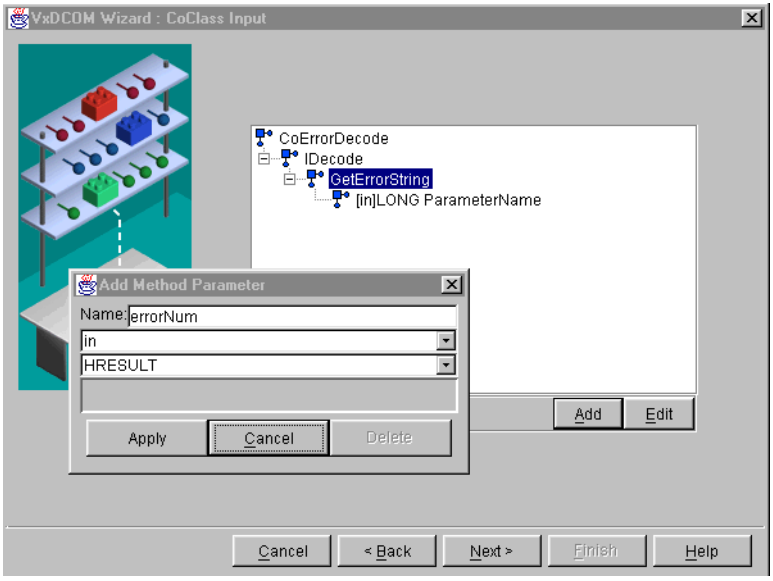
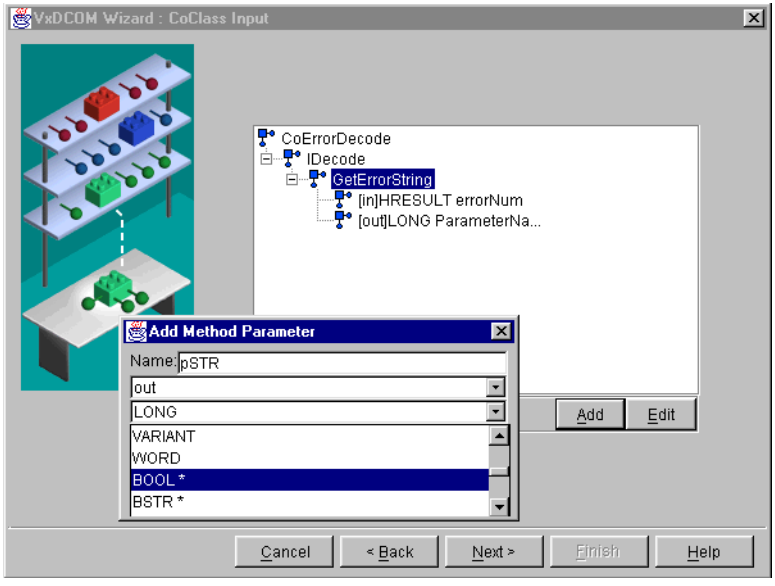


Figure 10-5 **Method Parameter Data Types**

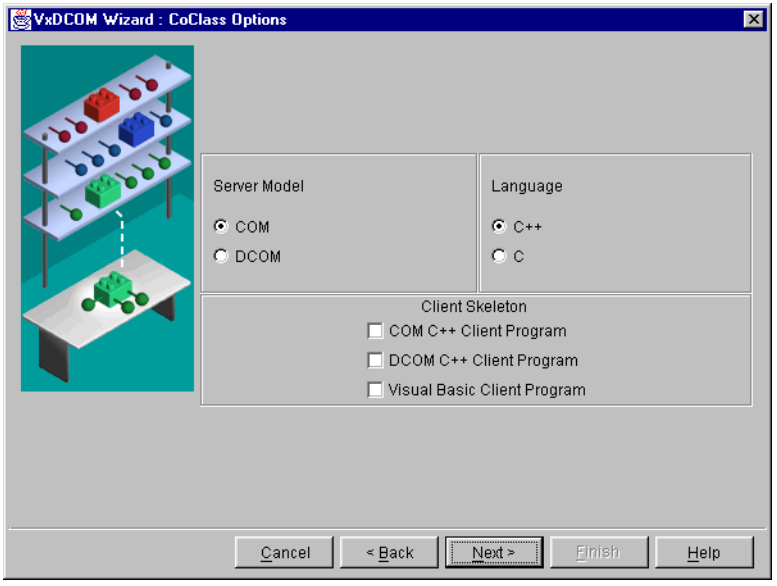


The **widl** tool compiles both automation data types and non-automation data types. The types compiled by **widl** are listed in the *VxWorks Programmer's Guide: VxDCOM Applications*.

Choosing CoClass Options

Once your CoClass definition is complete, clicking Next displays the CoClass Options dialog of the wizard, shown in Figure 10-6. From this dialog you specify the CoClass server model, implementation language, and any optional client programs. Then click Next.

Figure 10-6 **Choosing CoClass Options**



Server Models

Choose the Server Model for which you want to generate project files.

COM

A COM server model uses the COM technology entirely within the VxWorks system. A COM server is limited to communication with a COM C or C++ client on the same VxWorks target. You can use COM components to design object-oriented code based on the internal use of COM interfaces.

DCOM
A DCOM server model uses a distributed, component-based system (Distributed COM). DCOM extends the basic COM technology across process and machine boundaries by using a client-server application-level protocol for remote procedure calls. You can use a VxDCOM server, for example, to connect the desktop PC with distributed objects over a network.

Language

Choose the language used to implement the server CoClass.

C++
This language can be used for either COM or DCOM applications.

C
This language can be used only for COM, not DCOM, applications.

Client Skeleton Programs

Depending upon the server model, you can select from among several client application types. If you have a COM server, you can have only one C++ COM client project. If you have a DCOM server, you can choose more than one client program and clients of any type. Choosing a client program is optional.

C++ COM Client Program
The client must exist on the same VxWorks communication server as the VxWorks COM server.

C++ DCOM Client Program
The DCOM client can reside on either a VxWorks target or on a PCs running Windows NT. The communication server must be a VxWorks DCOM target server. When a client is on a PC, or another target, the DCE network layer is used. When the client is on a target, even though DCOM calls are made, it detects that the client is on the same host, and uses COM instead.

Visual Basic DCOM Client Program
A Visual Basic client program (such as Excel Basic, Word Basic, Access Basic, and so on) must reside on a PC running Windows NT. The client is written in Visual Basic, and the communication server is a DCOM VxWorks target server.

Generating the Skeleton Files

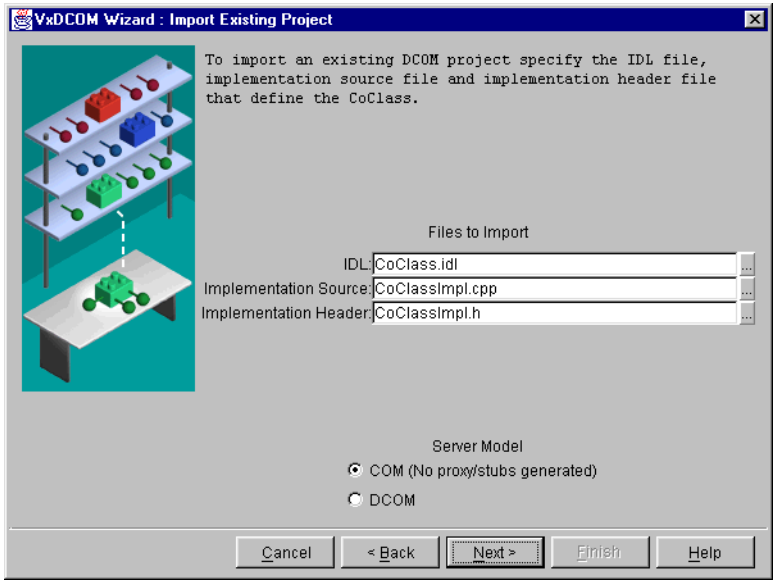
The last dialog of the wizard, Project Creation, appears. Simply click Finish to generate your project.

10.4.3 Importing Existing Files into a New Project

Porting Existing Applications

If you have an existing COM application, choose the option to Import existing files into new Project and choose Next. This brings up the Import Existing Project dialog shown in Figure 10-7.

Figure 10-7 **Adding Existing VxDCOM Files**



The files to add are the **.idl** file, the server implementation file, and the CoClass header file.

Editing IDL Files

If you wish to edit your **.idl** file by hand, you can refer to the *VxWorks Programmer's Guide* for details on the IDL file structure and the correct syntax for defining elements in that file. Remember to specify an **HRESULT** as the return type for all interface methods and to use automation data-types and directional attributes appropriately for your interface parameters.

If you do not use automation data types, refer to *Adding Non-Automation Types*, p.393 below. As a guide for writing in these files you can use the wizard GUI dialogs and wizard generated IDL files, and the **CoMathDemo.idl** file.

If for any reason you need to add additional interface definitions manually to the auto-generated **.idl** file, for each additional interface you must:

- Generate a new GUID (you can use the **UUIDGEN** utility to do this).
- Specify this value as the **[uuid]** attribute of your interface.

Adding Non-Automation Types

If you are using non-automation data types, the simplest way to add them is to generate all of the simple automation data types for the interface first using the wizard, generate the skeleton code. Then edit the **.idl** file and server implementation file by hand, adding the interface methods that could not be added with the wizard because they use non-automation types.

If you are defining interface methods that use non-automation types, you would not use the **[oleautomation]** attribute in your interface definition.

For more information on IDL, see the Microsoft documentation. Be aware that some of that information applies only to RPC interfaces and not to COM interfaces.

10.5 The Generated Output

The wizard generates output files in several subdirectories of your project name directory referred to here as *basename*. The content of those directories differs depending upon the options you selected in the wizard.

Output Directories

When the wizard finishes running it generates output files and creates 2 directories, which are:

- *basename*
- *basename*/**Client**

Subsequent sections describe the additional files that appear in these two directories. Depending upon the type of server and client you selected in the wizard, the content of these directories differs.

Project Work Files

These are the primary files to identify for working with your project. You can use these files to write code, compile, edit, or browse. Some are used for compiling, linking, building, and deployment of your application.

Makefile
The makefile used for building the project from the command line. Using this file is the default method for building the downloadable module. For details, see *10.7 Building and Linking the Application*, p.397.

basename.**idl**
This is the IDL file, identified by the **.idl** extension. This file is automatically compiled by **widl** when you build your Tornado project.

*basename***Impl.cpp**
This is the CoClass implementation file for your server, identified by the **Impl.cpp** extension. This is the file in which you implement the interface methods of your server CoClass.

*basename***Impl.h**
This is the CoClass header file, identified by the **Impl.h** extension. This file contains definitions and prototypes generated from the items you defined in the wizard.

Client/*basename***Client.cpp**
This is the skeleton implementation file for your client if you choose COM as your server and client model. It is identified by the **Client.cpp** extension. You edit this file to add client code to activate the COM server.

Client/*basename***DCOMClient.cpp**
This is the skeleton implementation file for your DCOM client application. It is identified by the **DCOMClient.cpp** extension. You edit this file to add client code to activate the DCOM server.

Client/*basename*.**rgs** (DCOM -only)
This is the registry-script file, identified by the **.rgs** extension. This file is used by the VxDCOM build tools to register the CoClass after it has been built. It is generated only when the DCOM server is selected. You only need to edit this

file if you are writing a DCOM application, in which case you modify the 'vxworks.target' entry.

Client/nmakefile (C++ Client only)
This is the host-side (client) **makefile**, always generated as **nmakefile**. It is used (with **nmake**) to build the MFC client application for Win32 or for VxWorks.

Server Output Files

The COM and DCOM server files differ only in the proxy stub code file used for marshaling.

COM Server Output

These are the files output to the *basename* directory for a COM server application.

- *basename*.**h**
- *basename*.**idl**
- *basename***Impl.cpp**
- *basename***Impl.h**
- *basename*_**i.c**

DCOM Server Output

These are the files output to the *basename* directory for a COM server application.

- *basename*.**h**
- *basename*.**idl**
- *basename***Impl.cpp**
- *basename***Impl.h**
- *basename*_**i.c**
- *basename*_**ps.cpp**

This file is output to the *basename*/**Client** directory, whether or not a client type is selected from the wizard:

- *basename*.**rgs** file

Client Output Files

If you selected more than one client program with your server choice, then all files necessary for each client are output into the *basename***/Client** directory. Remember that, for DCOM server projects, the *basename***/Client** directory also contains the *basename***.rgs** file. However, this file is only used by DCOM clients to register the type library.

COM Client Output

For a COM client, the following files are generated in the **Client** directory:

- *basename***Client.cpp**

DCOM C++ Client Output

For a DCOM C++ client, the following files are generated in the **Client** directory:

- *basename***DCOMClient.cpp**
- **nmakefile**

DCOM Visual Basic Client Output

For a DCOM Visual Basic client, the following files are generated in the **Client** directory:

- *basename***Client.bas**

10.6 Implementing the Server and Client

Once your system is created, you are ready to implement your CoClass methods for the server component and write the code for any client programs.

- COM server code in the implementation file *basename***Impl.cpp**, located in your project directory.
- COM client code in the implementation files, which are either *basename***Client.cpp** for a C++ COM client, or *basename***DCOMClient.cpp** for a C++ DCOM client. The client implementation files are located in the **/Client** subdirectory of your project directory.

The *VxWorks Programmer's Guide: VxDCOM Applications* describes this process. That chapter includes a reference for WOTL, the C++ template library used to write VxDCOM applications. It also covers the topics of using real-time extensions, adding OPC interfaces, working with **HRESULT** return values, and details of the **SAFEARRAY** types supported under VxDCOM.

10.7 Building and Linking the Application

Add the appropriate files, generated by the wizard, to your project. These files include the implementation code for the server and client. If you are creating a bootable application, add the files to the bootable system image. If you are creating a downloadable application, add the files to a downloadable module, ensuring that the module is downloaded to a bootable system containing the required VxDCOM support for you application.

Then, build the project. Although you can use the project facility from the IDE to build, the recommended method is to use the wizard-generated **Makefile** from the command line. The generated **Makefile** automatically runs **widl**. If you are building from the project facility, you must run **widl** manually. Running **widl** generates the proxy/stub code, the identification GUIDs, and the interface header prototypes. The build process also links the proxy/stub code.

Linking Proxy/Stub Code

The proxy/stub components generated by **widl** are required for the marshaling method used to remotely invoke interface methods across task boundaries.[1] In order for your application to use marshaling, server and client code must be linked with proxy/stub code. The proxy/stub code source file is *basename*_**ps.cpp** and the derived object file is *basename*_**ps.o**. You will find these in the workspace under each client and server component. The proxy/stub code is automatically linked to your client and server components when the system is built in the IDE.

On an NT machine, proxy/stub code is not required for automation data types.

1. Win32 uses type library based marshaling. VxDCOM generates custom marshaling code that is linked into the object module.

Building the Win32 Client

To build the client application for Win32, be sure that you have Visual C++ installed with the path to **nmake** configured in the command shell. Then, from the command line, run:

```
nmake -f nmakefile
```

10.8 Registering, Deploying, and Running Your Application

This section describes registering proxy DLLs, the type library, and the server, authenticating and activating the server.

10.8.1 Registering Proxy DLLs on Windows

If your project uses non-automation types, you must register the proxy DLLs on Windows.

Typically, when you define your interface you use automation data types. These are the types you can select from the VxDCOM wizard for your interface method parameters. When you use these types, the parameters are defined by the **[oleautomation]** attribute, to indicate that they are automation types. This signals to the Win32 Automation Marshaler that no extra Win32 proxy/stub DLLs are required because the marshaling of these types is handled automatically.

However, if your project requires non-automation types then you cannot specify the **[oleautomation]** attribute nor automate the marshaling of the parameter types. For interfaces that use non-automation types, you must generate and install your own Win32 proxy/stub DLLs. The DLL containing the interface proxy must be distributed with all client applications that use that new interface.

The specific non-automation types that **widl** can compile are described in the *VxWorks Programmer's Guide: VxDCOM Applications*.

The details of generating Win32 proxy/stub DLLs is outside of the scope of this document. Please refer to the Win32SDK MIDL documentation for details.

10.8.2 Register the Type Library

The type library is a binary file with a **.tlb** extension, that stores information about a DCOM object's properties and methods in a form that is accessible to other applications at run time. Windows' client applications use a type library to determine which interfaces an object supports and how to invoke the interface methods. For this reason, the type library must be registered. To register the type library, run *installDir***host\x86-win32\bin\vxreg32.exe**. This command adds the type library to the Windows Registry at its current location, so that the object can be accessed from the Windows host. If you add interfaces to the IDL file, the type library must also be re-registered so that the interface becomes known to the Windows Automation Marshaler.

The registry will contain an entry called **'vxworks.target'** that needs to be modified to point to the actual target, so that it is available from, for example, Visual Basic. There are two ways to change the target machine entry and register the type library:

- Edit the *basename*.**rgs** file (generated by the wizard) by changing the **'vxworks.target'** entry to the target IP address, and then run **vxreg32** as described above.
- Run **vxreg32** first, and then change the Windows registry entry for the target by opening up **DCOMCNFG** and using that tool to change the location.

10.8.3 Registering the Server

You must register your DCOM server classes in the VxWorks COM registry, so that client applications can locate them. There is one server-class per object-module. The object-module contains the server-class code, the proxy/stub code, and a registration object. This registration object registers the server-class with the VxWorks Registry when its constructor runs, guaranteeing that modules are automatically registered, regardless of whether they are pre-linked or downloaded. This auto-registration process does not rely on constructor ordering and can, thus, be safely performed at any point during system initialization.

To automatically register a DCOM server at load time or system startup, include the **AUTOREGISTER_COCLASS** macro in the CoClass implementation file. This will create an entry for the class in the VxWorks COM registry. **AUTOREGISTER_COCLASS** associates the CoClass with its CLSID, and registers the module name in the VxRegistry against its CLSID.

This macro is automatically included when the skeleton implementation file is generated. Thus, you do not need to add the macro code yourself (unless you work with files that were not auto-generated.)

The **AUTOREGISTER_COCLASS** macro takes three arguments: the server implementation CoClass, priority-scheme, and the default priority. The priority parameters are used as part of the real-time extension priority schemes, and are discussed in the *VxWorks AE Programmer's Guide: Writing COM and DCOM Clients and Servers*.

10.8.4 Authenticating the Server

The DCOM registry under VxWorks is not the complex, multi-purpose registry that Win32 supports, but is designed specifically for that purpose of:

- Allowing COM server classes to register their CLSIDs (class-IDs).
- Providing a link to an instantiation procedure, given that CLSID.
- Determining the correct proxy/stub configuration for any given interface, given its IID (unique identifier).

Therefore, the VxWorks registry works on a simple associative lookup method, keyed by the interface and class identifiers. The value that can be stored in the registry-entry is simply a string, which is used by various internal functions to look up proxy/stub classes, system-provided objects (such as the standard marshaler), and other objects identified by these values.

VxWorks also exposes non-Win32-compliant API calls to provide access to this registry. However, user/application code does not need to use these APIs, since **widl** automatically generates the code for registration of server-classes.

VxDCOM can participate in the basic NTLM authentication procedures when acting as a server, but not as a client. It can recognize incoming authentication requests from NT and correctly take part in the challenge/response transaction, but by default will not take any action based upon those transactions. Future versions of VxDCOM may more fully support the NTLM security system depending upon NT domain security, network trusts, and so on. However, for now the safest and most predictable approach is to disable client-side security by using either the registry/**DCOMCNFG** tool or by calling **CoInitializeSecurity( )**.

10.8.5 Activating the Server

After you have completed these steps you can run your program. See the *VxWorks Programmer's Guide: VxDCOM Applications* for descriptions of the **MathDemo** program client code, which activates the demo server component.

All VxDCOM threads must be created with **VX_FP_TASK**.

11 Customization

11.1 Introduction

Tornado allows you to customize certain tools in the Project window and to add menu entries for other tools you may wish to use. Clicking Tools>Options in the Project window displays a command to customize download options and version control tool usage. The Tools>Customize opens a dialog box for adding menu items.

11.2 Setting Download Options

The Download page provides options for handling symbols when objects are downloaded to the target (Figure 11-1).

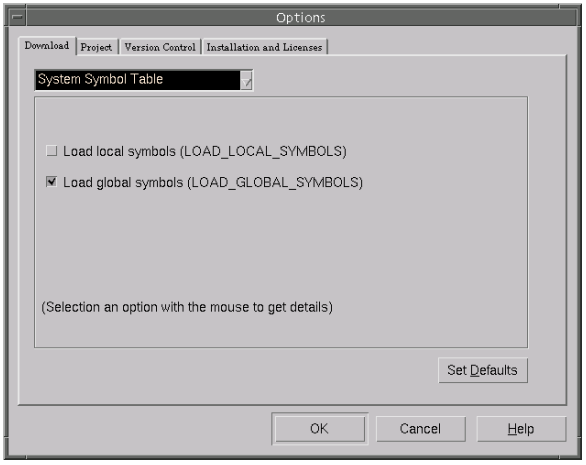
The options are as follows:

System Symbol Table
: The system symbol table offers two check box options:

  LOAD_LOCAL_SYMBOLS
  : Only local symbols are added to the system symbol table

  LOAD_GLOBAL_SYMBOLS (default)
  : Only external symbols are added to the system symbol table

Figure 11-1 **Download Page**



In order to obtain the other symbol options, you can:

- Check nothing, the equivalent of **LOAD_NO_SYMBOLS**, which adds no symbols to the system symbol table.
- Check both boxes, the equivalent of **LOAD_ALL_SYMBOLS**, which adds all symbols to the system symbol table.

Common Symbol Resolution

The common symbol resolution options are mutually exclusive.

LOAD_COMMON_MATCH_NONE
Allocate common symbols, but do not search for any matching symbols (the default)

LOAD_COMMON_MATCH_USER
Allocate common symbols, but search for matching symbols in user-loaded modules

LOAD_COMMON_MATCH_ALL (default)
Allocate common symbols, but search for matching symbols in user-loaded modules and the target-system core file

C++ Constructors

The C++ constructors options are mutually exclusive:

LOAD_CPLUS_XTOR_AUTO (default)
C++ ctors/dtors management is explicitly turned on

LOAD_CPLUS_XTOR_MANUAL
C++ ctors/dtors management is explicitly turned off

Miscellaneous

LOAD_HIDDEN_MODULE (default is not set)
Load the module but make it invisible to **WTX_MSG_MODULE_LIST**

The Set Defaults buttons resets the options to their defaults.

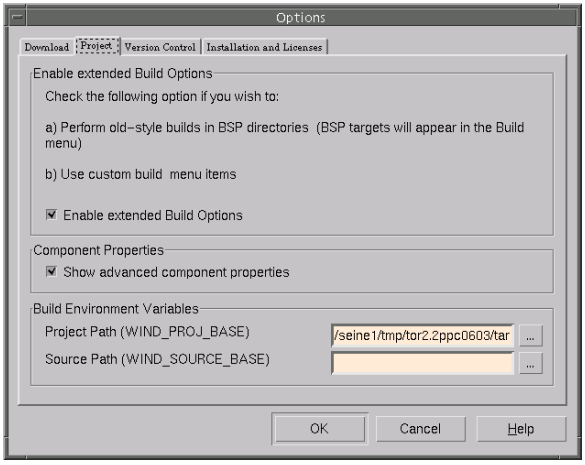
WARNING: The first time you start Tornado, you must open the Tools>Options>Download tab and set the defaults. Otherwise you may receive unexpected results.

See Help>Manuals Contents>Tornado API Reference>WTX Protocol> WTX>WTX_OBJ_MODULE_LOAD for more information about these options.

11.3 Setting Project Options

Select Options in the Tools menu, then click Projects to specify certain project attributes. The Projects page is shown in Figure 11-2.

Figure 11-2 **Projects Page**



The following choices are available on the Projects page:

Enable extended Build Options
: Checking the Enable extended Build Options box causes Standard BSP Builds to be added to the Build tab. For more information on command-line builds, see *5. Command-Line Configuration and Build*.

Component Properties
: Checking the Show advanced component properties box adds the Definition tab to the component property window. The Definition page shows the internal schema and attributes for the component. This may be helpful for authoring or debugging components.

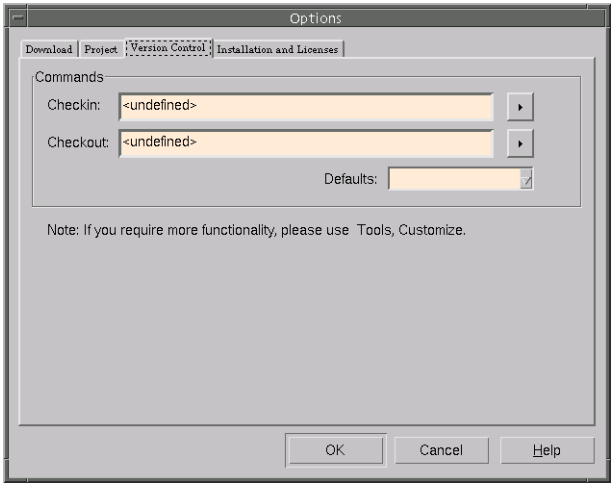
Build Environment Variables
: The Build Environment Variables section displays two environment variables, **WIND_SOURCE_BASE** and **WIND_PROJ_BASE**, that provide flexibility in locating and sharing project files. In most cases these variables are *not needed*. For information on when you might need them and how to use them, see *Sub-Projects*, p.106.

For example, using Tornado 1.0.1-style Build menu customizations, you can add a command that compiles the default **make** target in the same directory as the file currently open or selected in the Project tool. Use the **$filepath** macro in Working Directory and leaving Build Target blank in the Customize Builds dialog box.

11.4 Setting Version Control Options

If your organization uses a source-control (sometimes called configuration management) system to manage changes to source code, you probably need to execute a command to "check out" a file before you can make changes to it. Select Options in the Tools menu, then click Version Control to create commands to automatically check out or check in an open file using your version control system (Figure 11-3).

Figure 11-3 **Version Control Page**



The following choices are available on the Version Control page:

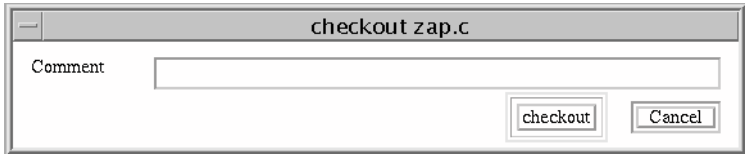
Checkin
: The Checkin text box allows you to enter the command that checks a file in to your version control system. The button at the end of the box opens a pop-up menu which has a Browse item to help you locate the command and macros to provide the Tornado context (see Table 11-1).

Checkout
: The Checkout text box allows you to enter the command that checks a file out of your version control system. The button at the end of the box opens a pop-up menu which has a Browse item to help you locate the command and macros to provide the Tornado context (see Table 11-1).

Table 11-1 **Macros for Version Control**

Menu Entry	Macro	Description	Example
File name	**$filename**	Name of the active file, without path information.	**zap.c**
Comment	**$comment**	Prompt for a checkin or checkout comment; any parameter needed by the tool can also be entered.	See Figure 11-4.

Figure 11-4 **Modify Comment Window**



Defaults
: Selecting an item from the Defaults drop-down list box automatically fills in the appropriate commands for the selected version control system.

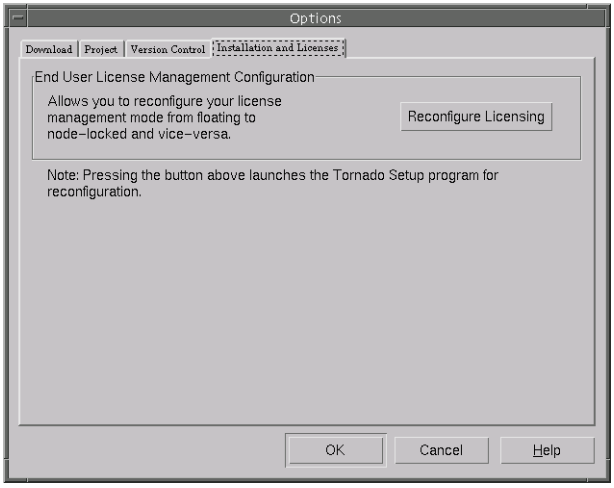
Commands created with the Checkin and Checkout text boxes appear on the pop-up menu accessed by right-clicking on the source file in the Tornado Editor window or the Project window.

NOTE: If you use Visual SourceSafe as your version control system, you must also have "Assume project based on working folder" checked in SourceSafe. This is not the default, but it affects only command-line SourceSafe use. Go to SourceSafe Explorer, Tools->Options->Command Line Options, check this option, and exit SourceSafe Explorer: the change does not take effect until you exit SourceSafe Explorer.

11.5 Installation and Licenses

Press the Reconfigure Licensing button to launch the Tornado **SETUP** program. You can change the license settings that you specified from SETUP when you installed Tornado.

Figure 11-5 **Installation and Licenses Page**



11.6 Customizing the Tools Menu

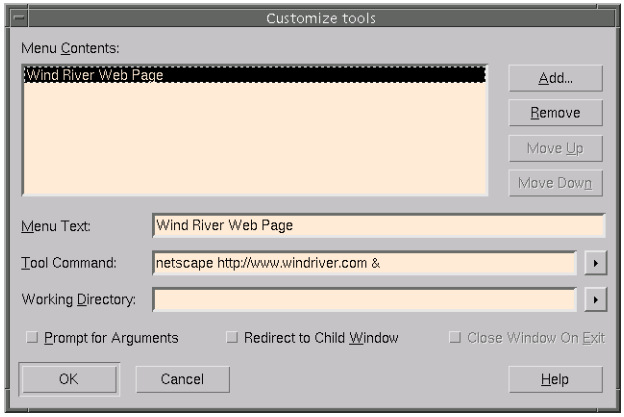
You can add entries to the Tools menu to allow easy access to additional tools. Before you add any commands in this part of the menu, Tornado displays the placeholder No Custom Tools as a disabled menu entry. The Customize command in the Tools menu allows you to add (or remove) entries at the end of the Tools menu.

11.6.1 The Customize Tools Dialog Box

Click Tools>Customize to open the Customize Tools dialog box (Figure 11-6).

The Menu Contents list box in the Customize Tools dialog box shows all custom commands currently in the Tools menu. When you select any item in this list, you can edit its attributes in the three text boxes near the bottom of the dialog box.

Figure 11-6 **Customize Tools Dialog Box**



The Customize Tools dialog box has the following buttons:

Add
: Activates the list and check boxes at the bottom of the Customize Tools dialog box and enters New Tool in the Menu Text list box. Replace New Tool with the command description; when you click OK, the new menu item appears in the Tools menu.

Remove
: Deletes the selected menu item from the Tools menu.

Move Up
: Moves the selected menu item up one line in the Menu Contents list box and changes the displayed order on the Tools menu.

Move Down
: Moves the selected menu item down one line in the Menu Contents list box and changes the displayed order on the Tools menu.

OK
: Applies your changes to the Tools menu.

Cancel
: Discards your changes without modifying the Tools menu.

The three text boxes near the bottom of the Customize Tools dialog box allow you to specify or change the attributes of a custom command.

Menu Text
: Contains the name of the custom command, as it appears in the Tools menu.

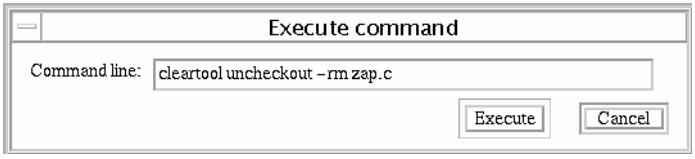
Tool Command
: Specifies the instructions to execute your command. You can place anything here that you could execute at the command prompt or in a batch file. Click the button at the right of the box to see a pop-up menu including a browse option and a list of macros which allow you to capture Tornado context in your commands. See *Macros for Customized Menu Commands*, p.412 for explanations of these macros.

Working Directory
: Use this field to specify where (in what directory) to run the custom command. You can edit the directory name in place, or click the button at the right of this field to bring up a menu similar to the Tool Command menu. It includes a directory browser where you can search for the right directory and the same list of macros. To use the Tornado working directory, leave this field blank.

At the bottom of the Customize Tools dialog box are the following check boxes:

Prompt for Arguments
: When this box is checked, Tornado prompts for command arguments using a dialog box, when you click the new command. The command line is displayed in a window where you can add additional information. (See Figure 11-7.)

Figure 11-7 **Command Line Arguments Dialog Box**



Redirect to Child Window
: When this box is checked, Tornado redirects the output of your command to a child window—a window contained within the Tornado application window. Otherwise, the command runs independently, either as a console application or a Windows application.

Close Window On Exit
: When this box is checked, Tornado closes the window associated with your tool when the command is done. This only applies when you also check the Redirect to Child Window box to redirect command output to a child window.

Macros for Customized Menu Commands

The pop-up menu opened by the buttons to the right of the text boxes provides several macros for your use in custom menu commands. These macros allow you to write custom commands that are sensitive to the context in the editor, or to the global Tornado context. For example, there are macros for the full path of the file in the active editor window, and for useful fragments of that file's name. Table 11-2 lists macros for editor context; in this table, the phrase *active file* refers to the file that is currently selected in the project facility.

Table 11-2 **Menu-Customization Macros for Editor Context**

Menu Entry	Macro	Description	Example
File path	**$filepath**	Full path to the active file.	**/usr/xeno/zap.c**
Dir name	**$filedir**	Directory containing the active file.	**/usr/xeno**
File name	**$filename**	Name of the active file, without path information.	**zap.c**
Base name	**$basename**	Name of the active file, without the file extension.	**zap**

Table 11-3 lists macros for the project facility context.

Table 11-3 **Menu-Customization Macros for Project Context**

Menu Entry	Macro	Description	Example
Project dir	**$projdir**	The name of the directory of the current project.	**/usr/xeno/proj/widget**
Project build dir	**$builddir**	The name of the directory for the current build of the current project.	**/usr/xeno/proj/widget/default**

Table 11-3 **Menu-Customization Macros for Project Context** *(Continued)*

Menu Entry	Macro	Description	Example
Derived object	**$derivedobj**	The name of the derived object of the currently selected source file in the current project.	**/usr/xeno/proj/widget/default/zap.o**

Table 11-4 lists macros for the global Tornado context.

Table 11-4 **Menu-Customization Macros for Global Context**

Menu Entry	Macro	Description	Example
Target name	**$targetName**	The full name of the target server selected in the Tornado Launch toolbar.	**vxsim@ontario**
Target	**$target**	The name of the target selected in the Tornado Launch toolbar.	**vxsim**
Tornado inst. dir	**$wind_base**	Installation directory recorded in the environment variable **WIND_BASE**.	**/usr/wind**
Tornado host type	**$wind_host_type**	Host type recorded in the environment variable **WIND_HOST_TYPE**.	sun4-solaris2
Tornado registry	**$wind_registry**	Registry recorded in the environment variable **WIND_REGISTRY**	**mars**

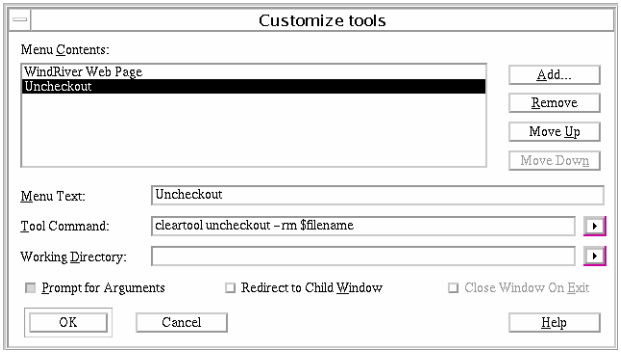
11.6.2 Examples of Tools Menu Customization

When creating a menu command be sure to check Redirect to Child Window for all applications that do not automatically open their own shell or window. Otherwise the command will attempt to run in the window that launched Tornado, if that window is still open. Thus an editor or version control command should be redirected, but a call to a browser need not be.

Version Control

This example illustrates how to use the Customize Tools dialog box to add an Uncheckout command to the Tools menu: the command cancels the checkout of whatever file is currently open in Tornado (that is, the file visible in the current editor window). Figure 11-8 illustrates the specification for a ClearCase command to uncheckout a module.

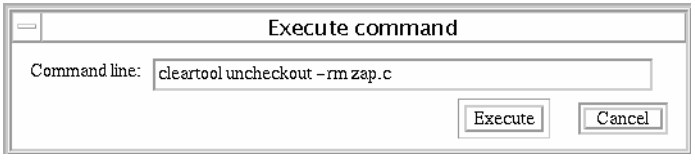
Figure 11-8 **Uncheckout Command for Tools Menu**



The Menu Text entry indicates that the command unchecks out a file, but is not specific to any particular file. The Tool Command field uses the **$filepath** macro (*Macros for Customized Menu Commands*, p.412) to expand the current file to its full path name.

In this example, the Prompt for Arguments and Redirect to Child Window boxes are checked. When the new Uncheckout command in the Tools menu is executed, the predefined argument list appears as a default in a dialog box (shown in Figure 11-9), to permit specifying other arguments if necessary.

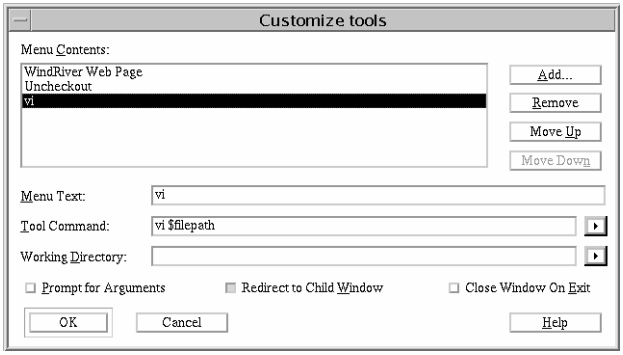
Figure 11-9 **Prompt for Arguments Dialog Box**



Alternate Editor

Figure 11-10 illustrates the specification for a command to run the UNIX vi editor on the file that is currently open in Tornado. The Menu Text contains a useful name, while the Tool Command field uses the actual execution command and **$filepath** to identify the current file. In this case, Prompt for Arguments is not checked; thus the editor runs immediately. Again, Redirect to Child Window is checked so that the editor will open in a new window.

Figure 11-10 **Custom Editor Command for Tools Menu**

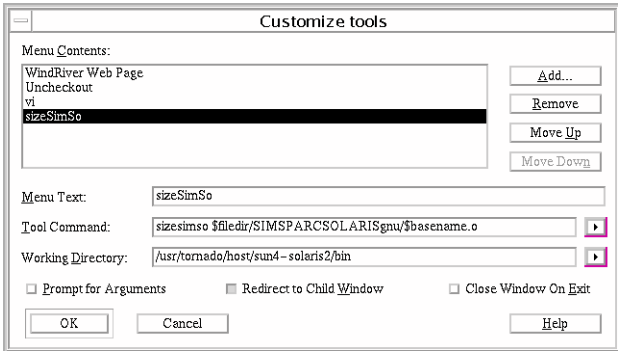


Binary Utilities

Tornado includes a suite of software-development utilities described in the *GNU ToolKit User's Guide: The GNU Binary Utilities*. If you execute any of these utilities frequently, it may be convenient to define commands in the Tools menu for that purpose.

Figure 11-11 illustrates the specification for a command to run the **size***arch* utility, which lists the size of each section of an object module for target architecture *arch*. In this example, the Tool Command field constructs the path and name of the object file generated from the current source file using **$filedir/SIMSPARCSOLARISgnu/$basename**. The Working Directory field is filled in using the browse option to locate the appropriate version of **size***arch* in the correct directory.

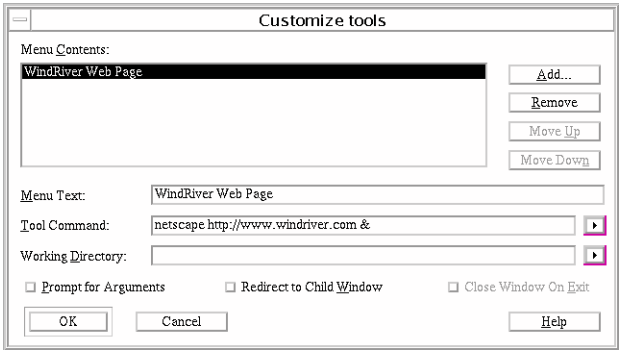
Figure 11-11 **Object-Module Size Command for Tools Menu**



World Wide Web

You can add a Tools command to link your Web browser directly to announcements from Wind River (and to related Internet resources). Figure 11-12 shows the specification for a Wind River Web Page command. (For a description of the information available on the Wind River home page, see *1.6 Customer Services*, p.12.

Figure 11-12 **Web Browser Command for Tools Menu**



Tornado itself does not include a Web browser. If you do not have a Web browser, or your system does not have direct Internet access, ignore this example; it provides convenient access to information, but no essential functionality.

11.7 Alternate Default Editor

In order to use an editor other than the default you must set the environment variable **EDITOR** to the command name. In order to use some other window besides xterm in which to execute a build, set the **WIND_BUILDTOOL** environment variable to that command.

Example 11-1 **Configuring emacs For Editing and Building**

To use emacs as your default editor and build tool requires two steps:

(1) Set the Tornado environment variables as follows:

```
EDITOR='gnuclient -q'
WIND_BUILDTOOL='gnudoit -q'
```

(2) Add these lines to your **.emacs** file if they are not already present:

```
;set up server mode
(setq gnuserv-frame (selected-frame))
(gnuserve-start)
```

You can also add emacs to the Tools menu as described in *Alternate Editor*, p.415. You will still need to perform Step (2) in order to be able to call emacs from the menu.

11.8 Tcl Customization Files

When Tornado begins executing, it checks for initialization files of the form **.wind/***filename***.tcl** in two places: first under *installDir* (that is, in the directory specified by the **WIND_BASE** environment variable), and then in the directory specified by the **HOME** environment variable (if that environment variable is defined). If any files are found, their contents are sourced as Tcl code when Tornado starts.

Tornado Initialization

The **Tornado.tcl** file allows you to customize the Tools menu and tool bar, as well as other elements of the Tornado window. For example, you can have your own dialog box appear based on a menu item you add to any menu. For more

information about the Tcl customization facilities available, see the *Tornado API Programmer's Guide* or the online *Tornado API Reference*.

HTML Help Initialization

The **windHelp.tcl** file allows you to specify a different HTML browser. The default is NetScape Communicator. To change the default, create **windHelp.tcl** with the following contents:

```
set htmlBrowser "newbrowser -install"
```

Appendices

A

Directories and Files

A.1 Introduction

All Wind River products are designed to install in a single coordinated directory tree. The directory root is shown as *installDir* in our documentation. The overall layout of the tree segregates files meant for the development host (such as the compilers and debugging tools), files for the target system (such as VxWorks, BSPs, and configuration files), and files that perform other functions (Table A-1).

Table A-1 **Contents of the Installation Directory**

Directory/File	Description
.wind	Directory for customization files and files that capture Tornado application state. Described in.*A.4 Initialization and State-Information Files*, p.432
LICENSE.TXT	Text of Wind River License agreement. This is a plain-text file that you can examine with any editor.
README.TXT	Last-minute information about the current Tornado release. This is a plain-text file that you can examine with any editor.
SETUP	Directory of SETUP program, including the DOCS subdirectory where PDF versions of documentation are located.
docs	Directory of online documentation in HTML format.
host	Directory of host-resident tools and associated files. Described in more detail in *A.2 Host Directories and Files*, p.422.
lmEnvVar.txt	Records license manager information from installation.

Table A-1 **Contents of the Installation Directory** *(Continued)*

Directory/File	Description
man	Directory of man page-style reference entries for VxWorks.
share	Directory of protocol definitions shared by both host and target software.
target	Directory of VxWorks target-resident software and associated files. Described in more detail in *A.3 Target Directories and Files*, p.424.
setup.log	Log file for SETUP program. This is a plain-text file that you can examine with any editor.

A.2 Host Directories and Files

Table A-2 is a summary and description of the Tornado directories and files below the top-level **host** directory.

Table A-2 **installDir/host**

Directory/File	Description
include	Directory containing header files for the Tornado tools.
host-os	Host-specific directory to permit Tornado installations for multiple hosts to be installed in a single tree, and share files in other directories.
host-os/**bin**	Directory containing executables for the Tornado tools (both interactive tools and the GNU ToolKit binaries) on a particular host. This directory must be on your execution path to use Tornado conveniently.
host-os/**include**	Directory containing OS-specific header files.
host-os/**jre118**	Directory containing Java Runtime Environment-related files.

Table A-2 **installDir/host** *(Continued)*

Directory/File	Description
host-*os*/**lib**	Directory containing both static libraries (**lib***name*.**a**) and shared libraries (*name*.**sl**) for the interactive Tornado tools. Subdirectories include **backend** (implementing the communications back ends available to the target server) and **gcc-lib** (containing the separate programs called by the GNU compiler driver).
host-*os*/*host-type*	Routines and scripts required by the host machine.
host-*os*/*x*-**wrs-vxworks**	Directory containing component programs and libraries for the GNU ToolKit configured for architecture *x*.
java	Directory containing various **.jar** files.
resource	Directory containing host-independent supporting files for the Tornado interactive tools.
resource/X11	A directory subtree containing data files and directories (such as font definitions, printer descriptions, and color definitions) related to the X Window System.
resource/app-defaults	Directory containing default X resource definitions for the Tornado interactive tools.
resource/bitmaps	Directory containing icons, button definitions, and busy-animation sequences for the Tornado tools.
resource/doctools	Directory containing tools for creating reference entries from source code.
resource/help	Directory containing online help.
resource/loader	Directory containing object-module format information for the Tornado dynamic linker.
resource/synopsis	Directory containing entries that support WindSh command completion and look-up.
resource/target	Directory containing the target information database.
resource/tcl	Directory containing Tcl source code for the Tornado tools, including sub-directories with dialog descriptions and other definitions for each tool.
resource/test	Directory containing tests for the WTX protocol.

Table A-2 **installDir/host** *(Continued)*

Directory/File	Description
resource/userlock	Global authorization file for Tornado users.
resource/vxdcom	Directory containing VxDCOM images and template.
resource/wdb	Mappings for WDB protocol extensions.
resource/wind_host_type	Returns the host type of the host to facilitate setting the environment variable **WIND_HOST_TYPE**.
src	Directory containing source for host utilities and examples, including **demo** (VxWorks sample programs) and **windview** (WindView and triggering sample programs).
tcl	Directory containing the standard Tcl and Tk distribution.

A.3 Target Directories and Files

Table A-3 is a summary and description of the Tornado directories and files below the top-level **target** directory.

Table A-3 **installDir/target**

Directory	File	Description
config		Directory containing files used to configure and build particular VxWorks systems. It includes system-dependent modules and some user-alterable modules. These files are organized into several subdirectories: the subdirectory **all**, which contains modules common to all implementations of VxWorks (system-*independent* modules), and a subdirectory for each port of VxWorks to specific target hardware (system-*dependent* modules).

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
config/all		Subdirectory containing system configuration modules. *Note that this method of configuration can be replaced by the project facility* (see *4. Projects*).
	bootInit.c	System-independent boot ROM facilities.
	configAll.h	Generic header file used to define configuration parameters common to all targets.
	usrConfig.c, **bootConfig.c**	Source of the configuration module for a VxWorks development system (**usrConfig.c**), and a configuration module for the VxWorks boot ROM (**bootConfig.c**).
config/*bspname*		Directory containing system-dependent modules for each port of VxWorks to a particular target CPU. Each of these directories includes the files listed below. In addition, drivers specific to each BSP are located here. **See** *4.4.4 Selecting the VxWorks Image Type*, p.143**.**
	00bsp.cdf	Project facility configuration file that overrides the generic BSP components in **comps/vxWorks/00bsp.cdf** with BSP-specific versions of components and parameters.
	00html.cdf	Project facility configuration file for HTML.
	Makefile	Makefile for creating boot ROMs and the VxWorks system image for a particular target.
	bootrom, **bootrom.hex**	VxWorks boot ROM code, as object module (**bootrom**) and as an ASCII file (**bootrom.hex**) in Motorola S-record format or Intel hex format (i960 targets), suitable for downloading over a serial connection to a PROM programmer. For more information see *4.8 Configuring and Building a VxWorks Boot Program*, p.164.
	*bspname***.h**	Header file for the target board.
	config.h	Header file of hardware configuration parameters.

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
	romInit.s	Assembly language source for initialization code that is the entry point for the VxWorks boot ROMs and ROM-based versions of VxWorks.
	sysLib.c, **sysALib.s**	Two source modules of system-dependent routines.
	target.nr	The source for the BSP-specific reference entry.
	vxWorks, **vxWorks.sym**	Complete, linked VxWorks system binary (**vxWorks**), and its symbol table (**vxWorks.sym**) created with the supplied configuration files.
config/comps		Directory containing source and configuration files.
config/*hostname*		Directory containing host-related configuration files.
h		Directory containing all the header (include) files supplied by VxWorks. Your application modules must include several of them in order to access VxWorks facilities.
h/arch		Directory containing architecture-dependent header files.
h/arpa		Directory containing a header file for use with **inetLib**.
h/dhcp		Directory containing header files for use with dhcp.
h/drv		Directory containing hardware-specific headers (primarily for drivers). Not all of the subdirectories shown are present for all BSPs.
h/make		Directory containing files that describe the rules for the makefiles for each CPU and toolset.
h/net		Directory containing all the internal header (include) files used by the VxWorks network. Network drivers will need to include several of these headers, but no application modules should need them.

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
h/netinet		Directory containing Internet-specific header files.
h/private		Directory containing header files for code private to VxWorks.
h/resolv		Directory containing header files for use with name service.
h/rip		Directory containing header files for use with rip.
h/rpc		Directory containing header files that must be included by applications using the Remote Procedure Call library (RPC).
h/sys		Directory containing header files specified by POSIX.
h/tffs		Directory containing header files for use with TrueFFS.
h/tool		Directory containing header files for Diab and GNU.
h/types		Directory containing header files used for defining types.
h/wdb		Directory containing header files for use with WDB.
idl		Directory containing COM-related files.
lib		Directory containing the machine-independent object libraries and modules provided by VxWorks.
lib/lib*cputool***vx.a**		An archive (**ar**) format library containing the object modules that make up VxWorks.
lib/obj*cputool***test**		Directory containing **vxColor.o**, a test program, for the specified target.
lib/*archfamily*		A directory of sub-directories containing BSP-specific libraries.

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
proj		Directory containing the default VxWorks images and the default location for projects to be created.
src		Directory containing all source files for VxWorks.
src/arch		Directory containing makefiles to build source.
src/config		Directory containing files used to force inclusion of specific VxWorks modules.
	ansi_5_0.c	Used to include all 5.0 ANSI C library routines.
	assertInit.c	Used to include the *assert* ANSI C library routine.
	cplusdiabComplex.c, **cplusdiabComplexIo.c**, **cplusdiabIos.c**, **cplusdiabIosLang.c**, **cplusdiabStl.c**, **cplusdiabString.c**, **cplusdiabStringIo.c**, **cplusgnuComplex.c**, **cplusgnuComplexIo.c**, **cplusgnuIos.c**, **cplusgnuIosLang.c**, **cplusgnuStl.c**, **cplusgnuString.c**, **cplusgnuStringIo.c**	Used to include the various groups of C++ routines.
	ctypeInit.c	Used to include the *ctype* ANSI C library routines.
	intrinsics.c	Used to include support for toolchain-dependent intrinsics.
	localeInit.c	Used to include the *locale* ANSI C library routines.
	mathInit.c	Used to include the *math* ANSI C library routines.

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
	stdioInit.c	Used to include the *stdio* ANSI C library routines.
	stdlibInit.c	Used to include the *stdlib* ANSI C library routines.
	stringInit.c	Used to include the *string* ANSI C library routines.
	timeInit.c	Used to include the time ANSI C library routines.
	usrAta.c	Used to include the ATA intialization routines.
	usrBreakpoint.c	Used to include the breakpoint management routines.
	usrDepend.c	Used to check module dependences for constants defined in **configAll.h** and **config.h**.
	usrDsp.c	Used to activate *dsp* support.
	usrExtra.c	Used to include extra modules that are needed by VxWorks but not referenced in the VxWorks code.
	usrFd.c	Used to mount a dosFs file system on a boot diskette (i386/i486 targets only).
	usrIde.c	Used to mount a dosFs file system on a boot IDE hard disk drive (i386/i486 targets only).
	usrKernel.c	Used to configure and initialize the *wind* kernel.
	usrLoadSym.c	Used to load the VxWorks symbol table.
	usrMmuInit.c	Used to initialize the memory management unit.
	usrNetwork.c	Used to configure and initialize networking support.
	usrPcmcia.c	Used to configure and initialize PCMCIA support.
	usrScript.c	Used to execute a startup script when VxWorks first boots.

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
	usrScsi.c	Used to configure and initialize SCSI support.
	usrTffs.c	Used to configure and initialize TrueFFS support.
	usrWdb.c	Used to configure and initialize the Tornado target agent.
	usrWindview.c	Used to configure and initialize WindView.
src/demo		Directory containing sample application modules for demonstration purposes, including both the source and the compiled object modules ready to be loaded into VxWorks.
src/demo/1		Directory containing a simple introductory demo program as well as a server/client socket demonstration.
src/demo/cplusplus		Directory containing the factory example application.
src/demo/color		Directory containing the VxColor example application.
src/demo/dg		Directory containing a simple datagram facility, useful for demonstrating and testing datagrams on VxWorks and/or other TCP/IP systems.
src/demo/start		Directory containing the program used with the *Tornado Getting Started Guide* tutorial.
src/demo/wind		Directory containing the windDemo example application.
src/drv		Directory containing source code for supported board device drivers.
src/usr		Directory containing user-modifiable code. Not every file in the directory is listed here.
	Makefile	Contains the makefile rules for building the vx library.

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
	devSplit.c	Provides a routine to split the device name from a full path name.
	memDrv.c	Pseudo-device driver that allows memory to be accessed as a VxWorks character (non-block) device.
	ramDiskCbio.c	RAM-disk driver with a CBIO interface which can be utilized directly by **dosFsLib** without using **dcacheCbio**.
	ramDrv.c	Block device driver that allows memory to be used as a device with VxWorks local file systems.
	usrLib.c	Library of routines designed for interactive invocation, which can be modified or extended if desired.
src\wdb		Directory containing target agent communication support.
	wdbUdpLib.c	Implements communication methods for the target agent using a lightweight UDP/IP stack.
	wdbUdpSockLib.c	Initializes UDP socket routines for the target agent.
The following directories are included only with a VxWorks source license.		
src/arch		Directory containing VxWorks source code for architecture-specific modules.
src/cplus		Directory containing source code for the Wind C++ Foundation Classes.
src/libc		Directory containing the source files for the ANSI C library.
src/math		Directory containing the source files for various math routines (non-ANSI).
src/netwrs		Directory containing the source files for the VxWorks network subsystem modules.

Table A-3 **installDir/target** *(Continued)*

Directory	File	Description
src/netinet		Directory containing the source files for Internet network protocols.
src/os		Directory containing the source code for VxWorks kernel extensions (for example: I/O, file systems).
src/ostool		Directory containing the source code for VxWorks tools.
src/rpc		Directory containing the source code for RPC that has been modified to run under VxWorks.
src/util		Directory containing source code for the VxWorks utilities.
src/wind		Directory containing source code for the VxWorks kernel.

A.4 Initialization and State-Information Files

You can define initialization files to customize each of the Tornado tools. These files, if they are present, are collected in a directory called **.wind** in your home directory. Some Tornado tools also use this directory to store state information, and some demos and optional products store both initialization and state information here.

Table A-4 **.wind Initialization Files**

Directory/Files	Description
browser.tcl	Optional Tcl initialization code for the browser. .
crosswind.tcl	Optional Tcl initialization code for the debugger front end.
gdb.tcl	Optional Tcl initialization code for the debugging engine itself.
launch.tcl	Optional Tcl initialization code for the launcher.

Table A-4 **.wind Initialization Files** *(Continued)*

Directory/Files	Description
windsh.tcl	Optional Tcl initialization code for the shell.
wtxtcl.tcl	Optional Tcl initialization code for **wtxtcl**, the Tcl interpreter with WTX-protocol extensions. See the *Tornado API Programmer's Guide: Extending Tornado Tools*.

Table A-5 **.wind State-Information Files**

Directory/Files	Description
launchLog.*server*	Log file for target-server verbose output, if requested from the launcher.
profile	A file of identification information used for your Tornado support requests. This information is collected and updated through the launcher's Support menu.
tgtsvr	A directory collecting your saved target-server configurations. Target-server configurations are defined and viewed through the launcher's Target menu.
tsr	A directory recording the history of your Tornado support requests. This information is managed through the launcher's Support menu; see *1.6 Customer Services*, p.12.

B

Makefile Details

B.1 Introduction

Each BSP has a makefile for building VxWorks. This makefile, called **Makefile**, is abbreviated to declare only the basic information needed to build VxWorks with the BSP. The makefile includes other files to provide target and VxWorks specific rules and dependencies. In particular, a file of the form **depend.***bspname* is included. The first time that **make** is run in the BSP directory, it creates the **depend.***bspname* file.

The **Makefile** in the BSP directory is used only when building from the traditional command line. It is not used when building projects from the project tool. Each build option for a project has its own makefile that the tool uses to build the project modules.

When projects are created from a BSP, the BSP makefile is scanned once and the make parameters are captured into the project. Any changes made to the BSP makefile after a project has been created do not affect that project. Only projects built from the BSP after the change is made are affected.

B.2 Customizing the VxWorks Makefile

The BSP makefile provides several mechanisms for configuring the VxWorks build. Although VxWorks configuration is more commonly controlled at compile-time by macros in **configAll.h** and *bspname***/config.h**.

NOTE: See the *BSP Developer's Guide* for important information on BSP distribution standards. Acceptable makefile customization is limited by the guidelines described in that section.

Most of the makefile macros fall into two categories: macros intended for use by the BSP developer, and macros intended for use by the end user. When building a VxWorks image, the needs of these two audiences differ considerably. Maintaining two separate compile-time macro sets lets the **make** separate the BSP-specific requirements from user-specific requirements.

Macros containing **EXTRA** in their name are intended for use by the BSP developer to specify additional object modules that must be compiled and linked with all VxWorks images.

Macros containing **ADDED** in their name are intended for use by the end-user on the **make** command line. This allows for easy compile time options to be specified by the user, without having to repeat any BSP-specific options in the same macro declaration.

B.3 Commonly Used Makefile Macros

Of the 135 or so makefile macros, this document discusses only the most commonly used.

MACH_EXTRA

You can add an object module to VxWorks by adding its name to the skeletal makefile. To include **fooLib.o**, for example, add it to the **MACH_EXTRA** definition in **Makefile**. This macro causes the linker to link it into the output object.

Finally, regenerate VxWorks with **make**. The module will now be included in all future VxWorks builds. If necessary, the module will be made from **fooLib.c** or **fooLib.s** using the **.c.o** or **.s.o** makefile rule.

MACH_EXTRA can be used for drivers that are not included in the VxWorks driver library. BSPs do not usually include source code for Ethernet and SCSI device drivers; thus, when preparing your system for distribution, omit the driver source file and change the object file's name from **.o** to **.obj** (update the makefiles, too). Now the end user can build VxWorks without the driver source, and **rm *.o** will not inadvertently remove the driver's object file.

LIB_EXTRA

The **LIB_EXTRA** makefile variable makes it possible to include library archives in the VxWorks build without altering the standard VxWorks archive or the driver library archive. Define **LIB_EXTRA** in **Makefile** to indicate the location of the extra libraries.

The libraries specified by **LIB_EXTRA** are provided to the link editor when building any VxWorks or boot ROM images. This is useful for third-party developers who want to supply end users with network or SCSI drivers, or other modules in object form, and find that the **MACH_EXTRA** mechanism described earlier in this chapter does not suit their needs.

The extra libraries are searched first, before Wind River libraries, and any references to VxWorks symbols are resolved properly.

EXTRA_INCLUDE

The makefile variable **EXTRA_INCLUDE** is available for specifying additional header directory locations. This is useful when the user or BSP provider has a separate directory of header files to be used in addition to the normal directory locations.


```
EXTRA_INCLUDE = -I../myHdrs
```

The normal order of directory searching for **#include** directives is:

```
$(INCLUDE_CC) (reserved for compiler specific uses)

$(EXTRA_INCLUDE)

.

$(CONFIG_ALL)

$(TGT_DIR)/h

$(TGT_DIR)/src/config

$(TGT_DIR)/src/drv
```

EXTRA_DEFINE

The makefile variable **EXTRA_DEFINE** is available for specifying compile time macros required for building a specific BSP. In the following example the macro **BRD_TYPE** is given the value **MB934**. This macro is defined on the command line for all compiler operations.

```
EXTRA_DEFINE = -DBRD_TYPE=MB934
```

By default a minimum set of macro names are defined on the compiler command line. This is primarily used to pass the same memory addresses used in both the compiling and linking operations.

These default macro definitions include:

```
-DCPU=$(CPU)
```

ADDED_CFLAGS

Sometimes it is inconvenient to modify **config.h** to control VxWorks configuration. **ADDED_CFLAGS** is useful for defining macros without modifying any source code.

Consider the hypothetical Acme XYZ-75 BSP that supports two hardware configurations. The XYZ-75 has a daughter board interface, and in this interface either a Galaxy-A or a Galaxy-B module is installed. The drivers for the modules are found in the directory **src/drv/multi**.

The macro **GALAXY_C_FILE** determines which driver to include at compile-time. The file named by **GALAXY_C_FILE** is **#include**d by **sysLib.c**.

The default configuration (Galaxy-A module installed) is established in **config.h**:

```
#ifndef GALAXY_C_FILE

#define GALAXY_C_FILE "multi/acmeGalaxyA.c"

#endif /* GALAXY_C_FILE */
```

When **make** is called normally, VxWorks supports the XYZ-75 with the Galaxy-A module installed. To override the default and build VxWorks for the XYZ-75/Galaxy-B configuration, do the following:

```
% make ADDED_CFLAGS='-DGALAXY_C_FILE=\"multi\/acmeGalaxy02.c\"' vxWorks
```

For ease of use, you can encapsulate the lengthy command line within a shell script or independent makefile.

To ensure that a module is incorporated in **vxWorks**, remove the module's object file and **vxWorks** before running **make**.

ADDED_MODULES

The **ADDED_MODULES** makefile variable makes it possible to add modules to VxWorks without modifying any source code.

While **MACH_EXTRA** requires the makefile to be modified, **ADDED_MODULES** allows one or more extra modules to be specified on the **make** command line. For

example, to build VxWorks with the BSP VTS support library included, copy **pkLib.c** to the target directory and enter the following:

```
% make ADDED_MODULES=pkLib.o vxWorks
```

One disadvantage of using **ADDED_MODULES** is that makefile dependencies are not generated for the module(s). In the above example, if **pkLib.c**, **pkLib.o**, and **vxWorks** already exist, you must remove **pkLib.o** and **vxWorks** before running **make** to force the latest **pkLib.c** to be incorporated into **vxWorks**.

CONFIG_ALL

Under extreme circumstances, the files in the **config/all** directory might not be flexible enough to support a complex BSP. In this case, copy all the **config/all** files to the BSP directory (**config/***bspname*) and edit the files as necessary. Then redefine the **CONFIG_ALL** makefile variable in **Makefile** to direct the build to the altered files. To do this, define **CONFIG_ALL** to equal the absolute path to the BSP's **config/***bspname* directory as shown in the following example:

```
CONFIG_ALL    =    $(TGT_DIR)/config/bspname/
```


The procedure described above works well if you must modify all or nearly all the files in **config/all**. However, if you know that only one or two files from **config/all** need modification, you can copy just those files to the BSP's **config/***bspname* directory. Then, instead of changing the **CONFIG_ALL** makefile macro, change one or more of the following (which ever are appropriate).

USRCONFIG
The path to an alternate **config/all/usrConfig.c** file.

BOOTCONFIG
The path to an alternate **config/all/bootConfig.c** file.

BOOTINIT
The path to an alternate **config/all/bootInit.c** file.

DATASEGPAD
The path to an alternate **config/all/dataSegPad.s** file.

CONFIG_ALL_H
The path to an alternate **config/all/configAll.h** file.

TGT_DIR
The path to the target directory tree, normally **$(WIND_BASE)/target**.

COMPRESS
The path to the host's compression program. This is the program that compresses an executable image. The binary image is input through **stdin**, and the output is placed on the **stdout** device. This macro can contain command-line flags for the program if necessary.

BINHEX
The path to the host's object-format-to-hex program. This program is called using **HEX_FLAGS** as command line flags. See **target/h/make/rules.bsp** for actual calling sequence.

HEX_FLAGS
Command line flags for the **$(BINHEX)** program.

BOOT_EXTRAA
Additional modules to be linked with compressed ROM images. These modules are not linked with uncompressed or ROM-resident images, just compressed images.

EXTRA_DOC_FLAGS
Additional preprocessor flags for making man pages. The default documentation flags are **-DDOC -DINCLUDE_SCSI**. If **EXTRA_DOC_FLAGS** is defined, these flags are passed to the man page generation routines in addition to the default flags.

C Tcl

C.1 Why Tcl?

Tcl is a scripting language which is designed to be embedded in applications. It can be embedded in applications that present command-line interfaces (the Tornado shell, for example) as well as in those that do not (such as the browser). Almost any program can benefit from the inclusion of such a language, because it provides a way for users to combine the program's features in new and unforeseen ways to meet their own needs. Many programs implement a command-line interface that is unique to the particular application. However, application-specific command line interfaces often have weak languages. Tcl holds some promise of unifying application command languages. This has an additional benefit: the more programs use a common language, the easier it is for everyone to learn to use each additional program that incorporates the language.

To encourage widespread adoption, John Ousterhout (the creator of Tcl) has placed the language and its implementation in the public domain.

Tk is often mentioned in conjunction with Tcl. Tk is a graphics library that extends Tcl with graphical-interface facilities. Tornado does not currently use Tk, but you may find Tk useful for your own Tcl applications.

C.2 Introduction to Tcl

Tcl represents all data as ordinary text strings. As you might expect, the string-handling features of Tcl are particularly strong. However, Tcl also provides a full complement of C-like arithmetic operators to manipulate strings that represent numbers.

The examples in the following sections exhibit some of the fundamental mechanisms of the Tcl language, in order to provide some of the flavor of working in Tcl. However, this is only an introduction.

For documentation on all Tcl interfaces in Tornado (as well as on C interfaces), see the *Tornado API Programmer's Guide* from Wind River.

For the Tcl language itself, the following generally available books are helpful:

- Ousterhout, John K.: *Tcl and the Tk Toolkit* (Addison-Wesley, 1994) – The definitive book on Tcl, written by its creator.
- Welch, Brent: *Practical Programming in Tcl and Tk* (Prentice Hall, 1995) – Useful both as a quick Tcl reference and as a tutorial.

C.2.1 Tcl Variables

The Tcl **set** command defines variables. Its result is the current value of the variable, as shown in the following examples:

Table C-1 **Setting Tcl Variables**

Tcl Expression	Result
`set num 6`	`6`
`set y hello`	`hello`
`set z "hello world"`	`hello world`
`set t $z`	`hello world`
`set u "$z $y"`	`hello world hello`
`set v {$z $y}`	`$z $y`

The expressions above also illustrate the use of some special characters in Tcl:

SPACE
Spaces normally separate single words, or tokens, each of which is a syntactic unit in Tcl expressions.

" … "
A pair of double quotes groups the enclosed string, including spaces, into a single token.

$*vname*
The **$** character normally introduces a variable reference. A token **$***vname* (either not surrounded by quotes, or inside double quotes) substitutes the value of the variable named *vname*.

{ … }
Curly braces are a stronger form of quoting. They group the enclosed string into a single token, and also prevent any substitutions in that string. For example, you can get the character **$** into a string by enclosing it in curly braces.

With a single argument, **set** gives the current value of a variable:

Table C-2 **Evaluating Tcl Variables**

Tcl Expression	Result
`set num`	`6`
`set z`	`hello world`

C.2.2 Lists in Tcl

Tcl provides special facilities for manipulating lists. In Tcl, a *list* is just a string, with the list elements delimited by spaces, as shown in the following examples:

Table C-3 **Using Tcl Lists**

Tcl Expression	Result	Description
`llength $v`	`2`	Length of list **v**.
`lindex $u 1`	`world`	Second element of list **u**.
`set long "a b c d e f g"`	`a b c d e f g`	Define a longer list.

Table C-3 **Using Tcl Lists** *(Continued)*

Tcl Expression	Result	Description
`lrange $long 2 4`	`c d e`	Select elements 2 through 4 of list **long**.
`lreplace $long 2 4 C D E`	`a b C D E f g`	Replace elements 2 through 4 of list **long**.
`set V "{c d e} f {h {i j} k}"`		Define a list of lists.
`lindex $V 1`	`f`	Some elements of **V** are singletons.
`lindex $V 0`	`c d e`	Some elements of **V** are lists.

The last examples use curly braces to delimit list items, yielding "lists of lists." This powerful technique, especially combined with recursive command substitution (see *C.2.4 Command Substitution*, p.445), can provide a little of the flavor of Lisp in Tcl programs.

C.2.3 Associative Arrays

Tcl arrays are all associative arrays, using a parenthesized key to select or define a particular element of an array: *arrayName***(***keyString***)**. The *keyString* may in fact represent a number, giving the effect of ordinary indexed arrays. The following are some examples of expressions involving Tcl arrays:

Table C-4 **Using Tcl Arrays**

Tcl Expression	Result	Description
`set taskId(tNetTask)`	`0x4f300`	Get element **tNetTask** of array **taskId**.
`set cpuFamily(5) m68k`	`m68k`	Define array **cpuFamily** and an element keyed **5**.
`set cpuFamily(10) sparc`	`sparc`	Define element keyed **10** of array **cpuFamily**.
`set cpuId 10`	`10`	Define **cpuId**, and use it as a key to **cpuFamily**.
`set cpuFamily($cpuId)`	`sparc`	

C.2.4 Command Substitution

In Tcl, you can capture the result of the command as text by enclosing the command in square brackets **[** … **]**. The Tcl interpreter substitutes the command result in the same process that is already running, which makes this an efficient operation.

Table C-5 **Examples of Tcl Command Substitution**

Tcl Expression	Result
`set m [lrange $long 2 4]`	`c d e`
`set n [lindex $m 1]`	`d`
`set o [lindex [lrange $long 2 4] 1]`	`d`
`set x [lindex [lindex $V 2] 1]`	`i j`

The last example selects from a list of lists (defined among the examples in *C.2.2 Lists in Tcl*, p.443). This and the previous example show that you can nest Tcl command substitutions readily. The Tcl interpreter substitutes the most deeply nested command, then continues substituting recursively until it can evaluate the outermost command.

C.2.5 Arithmetic

Tcl has an **expr** command to evaluate arithmetic expressions. The **expr** command understands numbers in decimal and hexadecimal, as in the following examples:

Table C-6 **Arithmetic in Tcl**

Tcl Expression	Result
`expr (2 << 2) + 3`	`11`
`expr 0xff00 & 0xf00`	`3840`

C.2.6 I/O, Files, and Formatting

Tcl includes many commands for working with files and for formatted I/O. Tcl also has many facilities for interrogating file directories and attributes. The examples in Table C-7 illustrate some of the possibilities.

Table C-7 **Files and Formatting in Tcl**

Tcl Expression	Description
`set myfile [open myfile.out w]`	Open a file for writing.
`puts $myfile [format "%s %d\n" \ "you are number" [expr 3+3]]`	Format a string and write it to file.
`close $myfile`	Close the file.
`file exists myfile.out`	`1`
`file writable myfile.out`	`1`
`file executable myfile.out`	`0`
`glob *.o`	`testCall.o foo.o bar.o`

C.2.7 Procedures

Procedure definition in Tcl is straightforward, and resembles many other languages. The command **proc** builds a procedure from its arguments, which give the procedure name, a list of its arguments, and a sequence of statements for the procedure body. In the body, the **return** command specifies the result of the procedure. For example, the following defines a procedure to compute the square of a number:

```
proc square {i} {
    return [expr $i * $i]
}
```

If a procedure's argument list ends with the word **args**, the result is a procedure that can be called with any number of arguments. All trailing arguments are captured in a list **$args**. For example, the following procedure calculates the sum of all its arguments:

```
proc sum {args} {
    set accum 0
    foreach item $args {
        incr accum $item
    }
    return $accum
}
```

Defined Tcl procedures are called by name, and can be used just like any other Tcl command. The following examples illustrate some possibilities:

Table C-8 **Calling a Tcl Procedure**

Tcl Expression	Result
`square 4`	16
`square [sum 1 2 3]`	36
`set x "squ"`	squ
`set y "are"`	are
`$x$y 4`	16

The technique illustrated by the last example—constructing a procedure name "on the fly"—is used extensively by Tornado tools to group a set of related procedures. The effect is similar to what can be achieved with function pointers in C.

For example, in Tornado tools, events are represented in Tcl as structured strings. The first element of the string is the name of the event. Tcl scripts that handle events can search for the appropriate procedure to handle a particular event by mapping the event name to a procedure name, and calling that procedure if it exists. The following Tcl script demonstrates this approach:

```
proc shEventDispatch {event} {
        set handlerProc "[lindex $event 0]_Handler"

        if {[info procs $handlerProc] != ""} {
            $handlerProc $event
        } {
            #event has no handler--do nothing.
        }
}
```

C.2.8 Control Structures

Tcl provides all the popular control structures: conditionals (**if**), loops (**while**, **for**, and **foreach**), case statements (**switch**), and explicit variable-scope control (**global**, **upvar**, and **uplevel** variable declarations). By using these facilities, you can even define your own control structures. While there is nothing mysterious about these facilities, more detailed descriptions are beyond the scope of this summary. For detailed information, see the books cited at the beginning of *C.2 Introduction to Tcl*, p.442.

C.2.9 Tcl Error Handling

Every Tcl procedure, whether built-in or script, normally returns a string. Tcl procedures may signal an error instead: in a defined procedure, this is done with the **error** command. This starts a process called *unwinding*. When a procedure signals an error, it passes to its caller a string containing information about the error. Control is passed to the calling procedure. If that procedure did not provide for this possibility by using the Tcl **catch** command, control is passed to its caller in turn. This recursive unwinding continues until the top level, the Tcl interpreter, is reached.

As control is passed along, any procedure can catch the error and take one of two actions: signal another error and provide error information, or work around the error and return as usual, ending the unwinding process.

At each unwinding step, the Tcl interpreter adds a description of the current execution context to the Tcl variable **errorInfo**. After unwinding ends, you can display **errorInfo** to trace error information. Another variable, **errorCode**, may contain diagnostic information, such as an operating system dependent error code returned by a system call.

C.2.10 Integrating Tcl and C Applications

Tcl is designed to integrate with C applications. The Tcl interpreter itself is distributed as a library, ready to link with other applications. The core of the Tcl integration strategy is to allow each application to add its own commands to the Tcl language. This is accomplished primarily through the subroutine **Tcl_CreateCommand( )** in the Tcl interpreter library, which associates a new Tcl command name and a pointer to an application-specific routine. For more details, consult the Tcl books cited at the beginning of *C.2 Introduction to Tcl*, p.442.

D

Coding Conventions

D.1 Introduction

This document defines the Wind River standard for all C code and for the accompanying documentation included in source code. The conventions are intended, in part, to encourage higher quality code; every source module is required to have certain essential documentation, and the code and documentation is required to be in a format that has been found to be readable and accessible.

The conventions are also intended to provide a level of uniformity in the code produced by different programmers. Uniformity allows programmers to work on code written by others with less overhead in adjusting to stylistic differences. Also it allows automated processing of the source; tools can be written to generate reference entries, module summaries, change reports, and so on.

The conventions described here are grouped as follows:

- **File Headings.** Regardless of the programming language, a single convention specifies a heading at the top of every source file.
- **C Coding Conventions.**
- **TCL Coding Conventions.**

D.2 File Heading

Every file containing C code—whether it is a header file, a resource file, or a file that implements a host tool, a library of routines, or an application—must contain a *standard file heading*. The conventions in this section define the standard for the heading that must come at the beginning of every source file.

The file heading consists of the blocks described below. The blocks are separated by one or more empty lines and contain no empty lines within the block. This facilitates automated processing of the heading.

- **Title:** The title consists of a one-line comment containing the tool, library, or applications name followed by a short description. The name must be the same as the file name. This line will become the title of automatically generated reference entries and indexes.
- **Copyright:** The copyright consists of a single-line comment containing the appropriate copyright information.
- **Modification History:** The modification history consists of a comment block: in C, a multi-line comment. Each entry in the modification history consists of the version number, date of modification, initials of the programmer who made the change, and a complete description of the change. If the modification fixes an SPR, then the modification history must include the SPR number.

  The version number is a two-digit number and a letter (for example, 03c). The letter is incremented for internal changes, and the number is incremented for large changes, especially those that materially affect the module's external interface.

The following example shows a standard file heading from a C source file:

Example D-1 **Standard File Heading (C Version)**

```
/* fooLib.c - foo subroutine library */

/* Copyright 1984-1995 Wind River Systems, Inc. */

/*
modification history
--------------------
02a,15sep92,nfs  added defines MAX_FOOS and MIN_FATS.
01b,15feb86,dnw  added routines fooGet() and fooPut();
                 added check for invalid index in fooFind().
01a,10feb86,dnw  written.
*/
```

D.3 C Coding Conventions

These conventions are divided into the following categories:

- Module Layout
- Subroutine Layout
- Code Layout
- Naming Conventions
- Style
- Header File Layout
- Documentation Generation

D.3.1 C Module Layout

A *module* is any unit of code that resides in a single source file. The conventions in this section define the standard module heading that must come at the beginning of every source module following the standard file heading. The module heading consists of the blocks described below; the blocks should be separated by one or more blank lines.

After the modification history and before the first function or executable code of the module, the following sections are included in the following order, if appropriate:

- **General Module Documentation:** The module documentation is a C comment consisting of a complete description of the overall module purpose and function, especially the external interface. The description includes the heading *INCLUDE FILES:* followed by a list of relevant header files.
- **Includes:** The include block consists of a one-line C comment containing the word *includes* followed by one or more C pre-processor **#include** directives. This block groups all header files included in the module in one place.
- **Defines:** The defines block consists of a one-line C comment containing the word *defines* followed by one or more C pre-processor **#define** directives. This block groups all definitions made in the module in one place.
- **Typedefs:** The typedefs block consists of a one-line C comment containing the word *typedefs* followed by one or more C **typedef** statements, one per line. This block groups all type definitions made in the module in one place.
- **Globals:** The globals block consists of a one-line C comment containing the word *globals* followed by one or more C declarations, one per line. This block

groups together all declarations in the module that are intended to be visible outside the module.

- **Locals:** The locals block consists of a one-line C comment containing the word *locals* followed by one or more C declarations, one per line. This block groups together all declarations in the module that are intended not to be visible outside the module.
- **Forward Declarations:** The forward declarations block consists of a one-line C comment containing the words *forward declarations* followed by one or more ANSI C function prototypes, one per line. This block groups together all the function prototype definitions required in the module. Forward declarations must only apply to local functions; other types of functions belong in a header file.

The format of these blocks is shown in the following example (which also includes the file heading specified earlier).

Example D-2 **C File and Module Headings**

```
/* fooLib.c - foo subroutine library */

/* Copyright 1984-1995 Wind River Systems, Inc. */

/*
modification history
--------------------
02a,15sep92,nfs  added defines MAX_FOOS and MIN_FATS.
01b,15feb86,dnw  added routines fooGet() and fooPut();
                 added check for invalid index in fooFind().
01a,10feb86,dnw  written.
*/

/*
DESCRIPTION
This module is an example of the Wind River Systems C coding conventions.
...
INCLUDE FILES: fooLib.h
*/

/* includes */

#include "vxWorks.h"
#include "fooLib.h"

/* defines */

#define MAX_FOOS        112  /* max # of foo entries */
#define MIN_FATS        2     * min # of FAT copies */

/* typedefs */
```

```
typedef struct fooMsg       /* FOO_MSG */
    {
    VOIDFUNCPTR func;        /* pointer to function to invoke */
    int arg [FOO_MAX_ARGS];  /* args for function */
    } FOO_MSG;

/* globals */

char *    pGlobalFoo;        /* global foo table */

/* locals */

LOCAL int numFoosLost;       /* count of foos lost */

/* forward declarations */

LOCAL int     fooMat (list * aList, int fooBar, BOOL doFoo);
FOO_MSG       fooNext (void);
STATUS        fooPut (FOO_MSG inPar);
```

D.3.2 C Subroutine Layout

The following conventions define the standard layout for every subroutine.

Each subroutine is preceded by a C comment heading consisting of documentation that includes the following blocks. There should be no blank lines in the heading, but each block should be separated with a line containing a single asterisk (*) in the first column.

- **Banner:** This is the start of a C comment and consists of a slash character (/) followed by 75 asterisks (*) across the page.
- **Title:** One line containing the routine name followed by a short, one-line description. The routine name in the title must match the declared routine name. This line becomes the title of automatically generated reference entries and indexes.
- **Description:** A full description of what the routine does and how to use it.
- **Returns:** The word *RETURNS:* followed by a description of the possible result values of the subroutine. If there is no return value (as in the case of routines declared **void**), enter:

  ```
  RETURNS: N/A
  ```

  Mention only true returns in this section—not values copied to a buffer given as an argument.

- **Error Number:** The word *ERRNO:* followed by all possible **errno** values returned by the function. No description of the **errno** value is given, only the **errno** value and only in the form of a defined constant.[1]

The subroutine documentation heading is terminated by the C end-of-comment character (*/), which must appear on a single line, starting in column one.

The subroutine declaration immediately follows the subroutine heading.[2] The format of the subroutine and parameter declarations is shown in *D.3.3 C Declaration Formats*, p.454.

Example D-3 **Standard C Subroutine Layout:**

```
/*********************************************************************
*
* fooGet - get an element from a foo
*
* This routine finds the element of a specified index in a specified
* foo.  The value of the element found is copied to <pValue>.
*
* RETURNS: OK, or ERROR if the element is not found.
*
* ERRNO:
*  S_fooLib_BLAH
*  S_fooLib_GRONK
*/

STATUS fooGet
    (
    FOO      foo,          /* foo in which to find element */
    int      index,        /* element to be found in foo */
    int *    pValue        /* where to put value */
    )
    {
    ...
    }
```

D.3.3 C Declaration Formats

Include only one declaration per line. Declarations are indented in accordance with *Indentation*, p.458, and are typed at the current indentation level.

The rest of this section describes the declaration formats for variables and subroutines.

1. A list containing the definitions of each **errno** is maintained and documented separately.
2. The declaration is used in the automatic generation of reference entries.

Variables

- For basic type variables, the type appears first on the line and is separated from the identifier by a tab. Complete the declaration with a meaningful one-line comment. For example:

```
unsigned    rootMemNBytes;   /* memory for TCB and root stack */
int         rootTaskId;      /* root task ID */
BOOL        roundRobinOn;    /* boolean for round-robin mode */
```

- The * and ** pointer declarators *belong* with the type. For example:

```
FOO_NODE *  pFooNode;        /* foo node pointer */
FOO_NODE ** ppFooNode;       /* pointer to the foo node pointer */
```

- Structures are formatted as follows: the keyword **struct** appears on the first line with the structure tag. The opening brace appears on the next line, followed by the elements of the structure. Each structure element is placed on a separate line with the appropriate indentation and comment. If necessary, the comments can extend over more than one line; see *Comments*, p.460, for details. The declaration is concluded by a line containing the closing brace, the type name, and the ending semicolon. Always define structures (and unions) with a **typedef** declaration, and always include the structure tag as well as the type name. Never use a structure (or union) definition to declare a variable directly. The following is an example of acceptable style:

```
typedef struct symtab   /* SYMTAB - symbol table */
    {
    OBJ_CORE    objCore;        /* object maintanance */
    HASH_ID     nameHashId;     /* hash table for names */
    SEMAPHORE   symMutex;       /* symbol table mutual exclusion sem */
    PART_ID     symPartId;      /* memory partition id for symbols */
    BOOL        sameNameOk;     /* symbol table name clash policy */
    int         nSymbols;       /* current number of symbols in table */
    } SYMTAB;
```

  This format is used for other composite type declarations such as **union** and **enum**.

  The exception to never using a structure definition to declare a variable directly is structure definitions that contain pointers to structures, which effectively declare another **typedef**. This exception allows structures to store pointers to related structures without requiring the inclusion of a header that defines the type.

For example, the following compiles without including the header that defines **struct fooInfo** (so long as the surrounding code never delves inside this structure):

CORRECT:

```
typedef struct tcbInfo
    {
    struct fooInfo *  pfooInfo;
    ...
    }  TCB_INFO;
```

By contrast, the following cannot compile without including a header file to define the type **FOO_INFO**:

INCORRECT:

```
typedef struct tcbInfo
    {
    FOO_INFO *  pfooInfo;
    ...
    }  TCB_INFO;
```

Subroutines

There are two formats for subroutine declarations, depending on whether the subroutine takes arguments.

- For subroutines that take arguments, the subroutine return type and name appear on the first line, the opening parenthesis on the next, followed by the arguments to the routine, each on a separate line. The declaration is concluded by a line containing the closing parenthesis. For example:

```
int lstFind
    (
    LIST *   pList,   /* list in which to search */
    NODE *   pNode    /* pointer to node to search for */
    )
```

- For subroutines that take no parameters, the word *void* in parentheses is required and appears on the same line as the subroutine return type and name. For example:

```
STATUS fppProbe (void)
```

D.3.4 C Code Layout

The maximum length for any line of code is 80 characters.

The rest of this section describes the conventions for the graphic layout of C code, and covers the following elements:

- vertical spacing
- horizontal spacing
- indentation
- comments

Vertical Spacing

- Use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines.
- Do not put more than one declaration on a line. Each variable and function argument must be declared on a separate line. Do not use comma-separated lists to declare multiple identifiers.
- Do not put more than one statement on a line. The only exceptions are the **for** statement, where the initial, conditional, and loop statements can go on a single line:

  ```
  for (i = 0; i < count; i++)
  ```

  or the **switch** statement if the actions are short and nearly identical (see the **switch** statement format in *Indentation*, p.476).

  The **if** statement is not an exception: the executed statement always goes on a separate line from the conditional expression:

  ```
  if (i > count)
      i = count;
  ```

- Braces ({ and }) and **case** labels always have their own line.

Horizontal Spacing

- Put spaces around binary operators, after commas, and before an open parenthesis. Do not put spaces around structure members and pointer operators. Put spaces before open brackets of array subscripts; however, if a

subscript is only one or two characters long, the space can be omitted. For example:

```
status = fooGet (foo, i + 3, &value);
foo.index
pFoo->index
fooArray [(max + min) / 2]
string[0]
```

- Line up continuation lines with the part of the preceding line they continue:

```
a = (b + c) *
    (d + e);

status = fooList (foo, a, b, c,
                  d, e);

if ((a == b) &&
    (c == d))
    ...
```

Indentation

- Indentation levels are every four characters (columns 1, 5, 9, 13, …).
- The module and subroutine headings and the subroutine declarations start in column one.
- Indent one indentation level after:
  - subroutine declarations
  - conditionals (see below)
  - looping constructs
  - switch statements
  - case labels
  - structure definitions in a **typedef**
- The **else** of a conditional has the same indentation as the corresponding **if**. Thus the form of the conditional is as follows:

```
if ( condition )
    {
    statements
    }
else
    {
    statements
    }
```

The form of the conditional statement with an **else if** is:

```
if ( condition )
    {
    statements
    }
else if ( condition )
    {
    statements
    }
else
    {
    statements
    }
```

- The general form of the **switch** statement is:

```
switch ( input )
    {
    case 'a':
        ...
        break;
    case 'b':
        ...
        break;
    default:
        ...
        break;
    }
```

If the actions are very short and nearly identical in all cases, an alternate form of the switch statement is acceptable:

```
switch ( input )
    {
    case 'a': x = aVar; break;
    case 'b': x = bVar; break;
    case 'c': x = cVar; break;
    default: x = defaultVar; break;
    }
```

- Comments have the same indentation level as the section of code to which they refer (see *Comments*, p.478).
- Section braces ({ and }) have the same indentation as the code they enclose.

Comments

- Place comments within code so that they precede the section of code to which they refer and have the same level of indentation. Separate such comments from the code by a single blank line.
  - Begin single-line comments with the open-comment and end with the close-comment, as in the following:

```
/* This is the correct format for a single-line comment */

foo = MAX_FOO;
```

  - Begin and end multi-line comments with the open-comment and close-comment on separate lines, and precede each line of the comment with an asterisk (*), as in the following:

```
/*
 * This is the correct format for a multiline comment
 * in a section of code.
 */

foo = MIN_FOO;
```

- Compose multi-line comments in declarations and at the end of code statements with one or more one-line comments, opened and closed on the same line. For example:

```
int foo
    (
    int     a,      /* this is the correct format for a */
                    /* multiline comment in a declaration */
    BOOL    b       /* standard comment at the end of a line */
    )

    {
    day = night;    /* when necessary, a comment about a line */
                    /* of code can be done this way */
    }
```

D.3.5 C Naming Conventions

The following conventions define the standards for naming modules, routines, variables, constants, macros, types, and structure and union members. The purpose of these conventions is uniformity and readability of code.

- When creating names, remember that the code is written only once, but read many times. Assign names that are meaningful and readable; avoid obscure abbreviations.
- Names of routines, variables, and structure and union members are composed of upper- and lowercase characters and no underbars. Capitalize each "word" except the first:

  aVariableName

- Names of defined types (defined with **typedef**), and constants and macros (defined with **#define**), are all uppercase with underbars separating the words in the name:

  A_CONSTANT_VALUE

- Every module has a short prefix (two to five characters). The prefix is attached to the module name and all externally available routines, variables, constants, macros, and **typedefs**. (Names not available externally do not follow this convention.)

fooLib.c	module name
fooObjFind	subroutine name
fooCount	variable name
FOO_MAX_COUNT	constant
FOO_NODE	type

- Names of routines follow the *module-noun-verb* rule. Start the routine name with the module prefix, followed by the noun or object that the routine manipulates. Conclude the name with the verb or action the routine performs:

fooObjFind	foo - object - find
sysNvRamGet	system - NVRAM - get
taskSwitchHookAdd	task - switch hook - add

- Every header file defines a preprocessor symbol that prevents the file from being included more than once. This symbol is formed from the header file name by prefixing **__INC** and removing the dot (**.**). For example, if the header file is called **fooLib.h**, the *multiple inclusion guard symbol* is:

  __INCfooLibh

- Pointer variable names have the prefix *p* for each level of indirection. For example:

```
FOO_NODE *      pFooNode;
FOO_NODE **     ppFooNode;
FOO_NODE ***    pppFooNode;
```

D.3.6 C Style

The following conventions define additional standards of programming style:

- **Comments:** Insufficiently commented code is unacceptable.
- **Numeric Constants:** Use **#define** to define meaningful names for constants. Do not use numeric constants in code or declarations (except for obvious uses of small constants like 0 and 1).
- **Boolean Tests:** Do not test non-booleans as you test a boolean. For example, where **x** is an integer:

  CORRECT:

  ```
  if (x == 0)
  ```

  INCORRECT:

  ```
  if (! x)
  ```

  Similarly, do not test booleans as non-booleans. For example, where **libInstalled** is declared as **BOOL**:

  CORRECT:

  ```
  if (libInstalled)
  ```

  INCORRECT:

  ```
  if (libInstalled == TRUE)
  ```

- **Private Interfaces:** Private interfaces are functions and data that are internal to an application or library and do not form part of the intended external user interface. Place private interfaces in a header file residing in a directory named **private**. End the name of the header file with an uppercase *P* (for *private*). For example, the private function prototypes and data for the commonly used internal functions in the library **blahLib** would be placed in the file **private/blahLibP.h**.
- **Passing and Returning Structures:** Always pass and return pointers to structures. Never pass or return structures directly.
- **Return Status Values:** Routines that return status values should return either OK or ERROR (defined in **vxWorks.h**). The specific type of error is identified by setting **errno**. Routines that do not return any values should return **void**.

- **Use Defined Names:** Use the names defined in **vxWorks.h** wherever possible. In particular, note the following definitions:
  - Use **TRUE** and **FALSE** for boolean assignment.
  - Use **EOS** for end-of-string tests.
  - Use **NULL** for zero pointer tests.
  - Use **IMPORT** for **extern** variables.
  - Use **LOCAL** for **static** variables.
  - Use **FUNCPTR** or **VOIDFUNCPTR** for pointer-to-function types.

D.3.7 C Header File Layout

Header files, denoted by a **.h** extension, contain definitions of status codes, type definitions, function prototypes, and other declarations that are to be used (through **#include**) by one or more modules. In common with other files, header files must have a *standard file heading* at the top. The conventions in this section define the header file contents that follow the standard file heading.

Structural

The following structural conventions ensure that generic header files can be used in as wide a range of circumstances as possible, without running into problems associated with multiple inclusion or differences between ANSI C and C++.

- To ensure that a header file is not included more than once, the following must bracket all code in the header file. This follows the standard file heading, with the **#endif** appearing on the last line in the file.

```
#ifndef __INCfooLibh
#define __INCfooLibh
    ...
#endif /* __INCfooLibh */
```

  See *D.3.5 C Naming Conventions*, p.460, for the convention for naming preprocessor symbols used to prevent multiple inclusion.

- To ensure C++ compatibility, header files that are compiled in both a C and C++ environment must use the following code as a nested bracket structure, subordinate to the statements defined above:

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
    ...
```

```
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

Order of Declaration

The following order is recommended for declarations within a header file:

(1) Statements that include other header files.

(2) Simple defines of such items as error status codes and macro definitions.

(3) Type definitions.

(4) Function prototype declarations.

Example D-4 **Sample C Header File**

The following header file demonstrates the conventions described above:

```
/* bootLib.h - boot support subroutine library */

/* Copyright 1984-1993 Wind River Systems, Inc. */

/*
modification history
--------------------
01g,22sep92,rrr  added support for c++.
01f,04jul92,jcf  cleaned up.
01e,26may92,rrr  the tree shuffle.
01d,04oct91,rrr  passed through the ansification filter,
                 -changed VOID to void
                 -changed copyright notice
01c,05oct90,shl  added ANSI function prototypes;
                 added copyright notice.
01b,10aug90,dnw  added declaration of bootParamsErrorPrint().
01a,18jul90,dnw  written.
*/

#ifndef __INCbootLibh
#define __INCbootLibh
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/*
 * BOOT_PARAMS is a structure containing all the fields of the
 * VxWorks boot line. The routines in bootLib convert this structure
 * to and from the boot line ASCII string.
 */

/* defines */
```

```
#define BOOT_DEV_LEN            20      /* max chars in device name */
#define BOOT_HOST_LEN           20      /* max chars in host name */
#define BOOT_ADDR_LEN           30      /* max chars in net addr */
#define BOOT_FILE_LEN           80      /* max chars in file name */
#define BOOT_USR_LEN            20      /* max chars in user name */
#define BOOT_PASSWORD_LEN       20      /* max chars in password */
#define BOOT_OTHER_LEN          80      /* max chars in "other" field */
#define BOOT_FIELD_LEN          80      /* max chars in boot field */

/* typedefs */

typedef struct bootParams               /* BOOT_PARAMS */
    {
    char bootDev [BOOT_DEV_LEN];        /* boot device code */
    char hostName [BOOT_HOST_LEN];      /* name of host */
    char targetName [BOOT_HOST_LEN];    /* name of target */
    char ead [BOOT_ADDR_LEN];           /* ethernet internet addr */
    char bad [BOOT_ADDR_LEN];           /* backplane internet addr */
    char had [BOOT_ADDR_LEN];           /* host internet addr */
    char gad [BOOT_ADDR_LEN];           /* gateway internet addr */
    char bootFile [BOOT_FILE_LEN];      /* name of boot file */
    char startupScript [BOOT_FILE_LEN]; /* name of startup script */
    char usr [BOOT_USR_LEN];            /* user name */
    char passwd [BOOT_PASSWORD_LEN];    /* password */
    char other [BOOT_OTHER_LEN];        /* avail to application */
    int  procNum;                       /* processor number */
    int  flags;                         /* configuration flags */
    } BOOT_PARAMS;

/* function declarations */

extern STATUS bootBpAnchorExtract (char * string, char ** pAnchorAdrs);
extern STATUS bootNetmaskExtract (char * string, int * pNetmask);
extern STATUS bootScanNum (char ** ppString, int * pValue, BOOL hex);
extern STATUS bootStructToString (char * paramString, BOOT_PARAMS *
                                  pBootParams);
extern char * bootStringToStruct (char * bootString, BOOT_PARAMS *
                                  pBootParams);
extern void   bootParamsErrorPrint (char * bootString, char * pError);
extern void   bootParamsPrompt (char * string);
extern void   bootParamsShow (char * paramString);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __INCbootLibh */
```

D.3.8 Documentation Format Conventions for C

This section specifies the text-formatting conventions for source-code derived documentation. The Wind River tool **refgen** is used to generate reference entries (in HTML format) for every module automatically. All modules must be able to generate valid reference entries. This section is a summary of basic documentation format issues; for a more detailed discussion, see the *Tornado BSP Developer's Kit User's Guide: Documentation Guidelines*.

Layout

To work with **refgen**, the documentation in source modules must be laid out following a few simple principles. The file **sample.c** in *installDir*/**target/unsupported/tools/mangen** provides an example and more information.

Lines of text should fill out the full line length (assume about 75 characters); do not start every sentence on a new line.

Format Commands

Documentation in source modules can be formatted with UNIX **nroff**/**troff** formatting commands, including the standard **man** macros and several Wind River extensions to the **man** macros. Some examples are described in the sections below. Such commands should be used sparingly.

Any macro (or "dot command") must appear on a line by itself, and the dot (**.**) must be the first character on the logical line (in the case of subroutines, this is column 3, because subroutine comment sections begin each line with an asterisk plus a space character).

Special Elements

- **Parameters:** When referring to a parameter in text, surround the name with the angle brackets, **<** and **>**. For example, if the routine **getName( )** had the following declaration:

```
void getName
    (
    int     tid,    /* task ID */
    char *  pTname  /* task name */
    )
```

You might write something like the following:

```
This routine gets the name associated with a specified task ID and copies
it to <pTname>.
```

- **Subroutines:** Include parentheses with all subroutine names, even those generally construed as shell commands. Do not put a space between the parentheses or after the name (unlike the Wind River convention for code):

CORRECT: `taskSpawn()`

INCORRECT: `taskSpawn (), taskSpawn( ), taskSpawn`

Note that there is one major exception to this rule. In the subroutine title, do not include the parentheses in the name of the subroutine being defined:

CORRECT:

```
/***********************************************
*
* xxxFunc - do such and such
```

INCORRECT:

```
/***********************************************
*
* xxxFunc() - do such and such
```

Avoid using a library, driver, or routine name as the first word in a sentence, but if you must, do not capitalize it.

- **Terminal Keys:** Enter the names of terminal keys in all uppercase, as in **TAB** or **ESC**. Prefix the names of control characters with **CTRL+**; for example, **CTRL+C**.
- **References to Publications:** References to chapters of publications should take the form *Title*: *Chapter*. For example, you might say:

  For more information, see the *VxWorks Programmer's Guide: I/O System*.

  References to documentation volumes should be set off in italics. For general cases, use the **.I** macro. However, in SEE ALSO sections, use the **.pG** and **.tG** macros for the *VxWorks Programmer's Guide* and *Tornado User's Guide*, respectively.

- **Section-Number Cross-References:** Do not use the UNIX parentheses-plus-number scheme to cross-reference the documentation sections for libraries and routines:

  CORRECT:

  ```
  sysLib, vxTas()
  ```

  INCORRECT:

  ```
  sysLib(1), vxTas(2)
  ```

Table D-1 **Format of Special Elements**

Component	Input	Output (mangen + troff)
library in title	`sysLib.c`	sysLib
library in text	`sysLib`	(same)
subroutine in title	`sysMemTop`	sysMemTop()
subroutine in text	`sysMemTop()`	(same)
subroutine parameter	`<ptid>`	(same)
terminal key	`TAB, ESC, CTRL+C`	(same)
publication	`.I "Tornado User's Guide"`	*Tornado User's Guide*
VxWorks Programmer's Guide in **SEE ALSO**	`.pG "Configuration"`	*VxWorks Programmer's Guide: Configuration*
Tornado User's Guide in **SEE ALSO**	`.tG "Cross-Development"`	*Tornado User's Guide: Shell*
emphasis	`\f2must\fP`	*must*

Formatting Displays

- **Code:** Use the **.CS** and **.CE** macros for displays of code or terminal input/output. Introduce the display with the **.CS** macro; end the display with **.CE**. Indent such displays by four spaces from the left margin. For example:

```
* .CS
*     struct stat statStruct;
*     fd = open ("file", READ);
*     status = ioctl (fd, FIOFSTATGET, &statStruct);
* .CE
```

- **Board Diagrams:** Use the **.bS** and **.bE** macros to display board diagrams under the BOARD LAYOUT heading in the **target.nr** module for a BSP. Introduce the display with the **.bS** macro; end the display with **.bE**.

- **Tables:** Tables built with **tbl** are easy as long as you stick to basics, which suffice in almost all cases. Tables always start with the **.TS** macro and end with a **.TE**. The **.TS** should be followed immediately by a line of options terminated by a semicolon (**;**); then by one or more lines of column specification commands followed by a dot (**.**). For more details on table commands, refer to any UNIX documentation on **tbl**. The following is a basic example:

```
.TS
center; tab(|);
lf3 lf3
l l.
Command          | Op Code
_
INQUIRY          | (0x12)
REQUEST SENSE    | (0x03)
TEST UNIT READY | (0x00)
.TE
```

  General stylistic considerations are as follows:

  – Redefine the tab character using the **tab** option; keyboard tabs cannot be used by **tbl** tables. Typically the pipe character (**|**) is used.
  – Center small tables on the page.
  – Expand wide tables to the current line length.
  – Make column headings bold.
  – Separate column headings from the table body with a single line.
  – Align columns visually.

  Do not use **.CS**/**.CE** to build tables. This markup is reserved for code examples.

- **Lists:** List items are easily created using the standard **man** macro **.IP**. Do not use the **.CS**/**.CE** macros to build lists. The following is a basic example:

```
.IP "FIODISKFORMAT"
Formats the entire disk with appropriate hardware track and
sector marks.
.IP "FIODISKINIT"
Initializes a DOS file system on the disk volume.
```

D.4 Tcl Coding Conventions

These conventions are divided into the following categories:

- Module Layout
- Procedure Layout
- Code outside of procedure
- Code Layout
- Naming Conventions
- Style

D.4.1 Tcl Module Layout

A *module* is any unit of code that resides in a single Tcl file. The conventions in this section define the standard module heading that must come at the beginning of every Tcl module following the standard file heading. The module heading consists of the blocks described below; the blocks are separated by one or more blank lines.

After the modification history and before the first function or executable code of the module, the following sections are included in the following order, if appropriate:

- **General Module Documentation:** The module documentation is a block of single-line Tcl comments beginning by the keyword *DESCRIPTION* and consisting of a complete description of the overall module purpose and function, especially the external interface. The description includes the heading *RESOURCE FILES* followed by a list of relevant Tcl files sourced inside the file.
- **Globals:** The globals block consists of a one-line Tcl comment containing the word *globals* followed by one or more Tcl declarations, one per line. This block groups together all declarations in the module that are intended to be visible outside the module.

The format of these blocks is shown in the following example (which also includes the Tcl version of the file heading):

Example D-5 **Tcl File and Module Headings**

```
# Browser.tcl - Browser Tcl implementation file
#
# Copyright 1994-1995 Wind River Systems, Inc.
#
```

```
# modification history
# --------------------
# 02b,30oct95,jco  added About menu and source browser.tcl in .wind.
# 02a,02sep95,pad  fixed communications loss with license daemon (SPR #1234).
# 01c,05mar95,jcf  upgraded spy dialog
# 01b,08feb95,p_m  take care of loadFlags in wtxObjModuleInfoGet.
# 01a,06dec94,c_s  written.
#
# DESCRIPTION
# This module is the Tcl code for the browser. It creates the main window and
# initializes the objects in it, such as the task list and memory charts.
#
# RESOURCE FILES
# wpwr/host/resource/tcl/shelbrws.tcl
# wpwr/host/resource/tcl/app-config/Browser/*.tcl
# ...
#*/

# globals

set browserUpdate         0         ;# no auto update by default
```

D.4.2 Tcl Procedure Layout

The following conventions define the standard layout for every procedure in a module.

Each procedure is preceded by the procedure documentation, a series of Tcl comments that includes the following blocks. The documentation contains no blank lines, but each block is delimited with a line containing a single pound symbol (#) in the first column.

- **Banner:** A Tcl comment that consists of 78 pound symbols (#) across the page.
- **Title:** One line containing the routine name followed by a short, one-line description. The routine name in the title must match the declared routine name. This line becomes the title of automatically generated reference entries and indexes.
- **Description:** A full description of what the routine does and how to use it.
- **Synopsis:** The word *SYNOPSIS:* followed by a the synopsis of the procedure—its name and parameter list between **.tS** and **.tE** macros. Optional parameters are shown in square brackets. A variable list of arguments is represented by three dots (**...**).
- **Parameters:** For each parameter, the **.IP** macro followed by the parameter name on one line, followed by its complete description on the next line.

Include the default value and domain of definition in each parameter description.

- **Returns:** The word *RETURNS:* followed by a description of the possible explicit result values of the subroutine (that is, values returned with the Tcl **return** command).

```
RETURNS:
A list of 11 items: vxTicks taskId status priority pc sp errno
timeout entry priNormal name
```

If the return value is meaningless enter N/A:

```
RETURNS: N/A
```

- **Errors:** The word *ERRORS:* followed by all the error messages or error code (or both, if necessary) raised in the procedure by the Tcl **error** command.

```
ERRORS:
"Cannot find symbol in symbol table"
```

If no **error** statement is invoked in the procedure, enter N/A.

```
ERRORS: N/A
```

The procedure documentation ends with an empty Tcl comment starting in column one.

The procedure declaration follows the procedure heading and is separated from the documentation block by a single blank line. The format of the procedure and parameter declarations is shown in *D.3.3 C Declaration Formats*, p.454.

The following is an example of a standard procedure layout.

Example D-6 **Standard Tcl Procedure Layout**

```
##########################################################################
#
# browse - browse an object, given its ID
#
# This routine is bound to the "Show" button, and is invoked when
# that button is clicked.  If the argument (the contents of...
#
# SYNOPSIS
# .tS
# browse [objAddr | symbol | &symbol]
# .tE
#
```

```
# PARAMETERS
# .IP <objAddr>
# the address of an object to browse
# .IP <symbol>
# a symbolic address whose contents is the address of
# an object to browse
# .IP <&symbol>
# a symbolic address that is the address of an object to browse
#
# RETURNS: N/A
#
# ERRORS: N/A
#

proc browse {args} {
    ...
}
```

D.4.3 Tcl Code Outside Procedures

Tcl allows code that is not in a procedure. This code is interpreted immediately when the file is read by the Tcl interpreter. Aside from the global-variable initialization done in the globals block near the top of the file, collect all such material at the bottom of the file.

However, it improves clarity—when possible—to collect any initialization code in an initialization procedure, leaving only a single call to that procedure at the bottom of the file. This is especially true for dialog creation and initialization, and more generally for all commands related to graphic objects.

Tcl code outside procedures must also have a documentation heading, including the following blocks:

- **Banner:** A Tcl comment that consists of 78 pound symbols (#) across the page.
- **Title:** One line containing the file name followed by a short, one-line description. The file name in the title must match the file name in the file heading.
- **Description:** A description of the out-of-procedure code.

The following is a sample heading for Tcl code outside all procedures.

Example D-7 **Heading for Out-of-Procedure Tcl Code**

```
########################################################################
# 01Spy.tcl - Initialization code
#
# This code is executed when the file is sourced. It executes the module
# entry routine which does all the necessary initialization to get a
# runnable spy utility.
#

# Call the entry point for the module

spyInit
```

D.4.4 Declaration Formats

Include only one declaration per line. Declarations are indented in accordance with *Indentation*, p.476, and begin at the current indentation level. The remainder of this section describes the declaration formats for variables and procedures.

Variables

For global variables, the Tcl **set** command appears first on the line, separated from the identifier by a tab character. Complete the declaration with a meaningful comment at the end of the same line. Variables, values, and comments should be aligned, as in the following example:

```
set     rootMemNBytes     0  ;# memory for TCB and root stack
set     rootTaskId        0  ;# root task ID
set     symSortByName     1  ;# boolean for alphabetical sort
```

Procedures

The procedure name and list of parameters appear on the first line, followed by the opening curly brace. The declarations of global variables used inside the procedure begin on the next line, one on each separate line. The rest of the procedure code begins after a blank line.

For example:

```
proc lstFind {list node} {
    global firstNode
    global lastNode

    ...
}
```

D.4.5 Code Layout

The maximum length for any line of code is 80 characters. If more than 80 characters are required, use the backslash character to continue on the next line.

The rest of this section describes conventions for the graphic layout of Tcl code, covering the following elements:

- vertical spacing
- horizontal spacing
- indentation
- comments

Vertical Spacing

- Use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines.
- Do not put more than one declaration on a line. Each variable and function argument must be declared on a separate line.
- Do not put more than one statement on a line. The only exceptions are:
  - A **for** statement where the initial, conditional, and loop statements can be written on a single line:

    ```
    for {set i 0} {$i < 10} {incr i 3} {
    ```

  - A **switch** statement whose actions are short and nearly identical (see the **switch** statement format in *Indentation*, p.476).

  The **if** statement is not an exception. The conditionally executed statement always goes on a separate line from the conditional expression:

  ```
  if {$i > $count} {
      set i $count
  }
  ```

- Opening braces ({), defining a command body, are always on the same line as the command itself.
- Closing braces (}) and **switch** patterns always have their own line.

Horizontal Spacing

- Put spaces around binary operators. Put spaces before an open parenthesis, open brace and open square bracket if it follows a command or assignment statement. For example:

```
set status [fooGet $foo [expr $i + 3] $value]
if {&value & &mask} {
```

- Line up continuation lines with the part of the preceding line they continue:

```
set a [expr ($b + $c) * \
            ($d + $e)]
set status [fooList $foo $a $b $c \
                   $d $e]
if {($a == $b) && \
    ($c == $d)} {
    ...
}
```

Indentation

- Indentation levels are every four characters (columns 1, 5, 9, 13, …).
- The default tab width must be eight characters (assumed to be at columns 1, 9, 17, ...). The intermediate indentations are achieved with spaces.
- The module and procedure headings and the procedure declarations start in column one.
- The closing brace of a command body is always aligned on the same column as the command it is related to:

```
while { condition }{
    statements
}

foreach i $elem {
    statements
}
```

- Add one more indentation level after any of the following:
  - procedure declarations
  - conditionals (see below)
  - looping constructs
  - switch statements
  - switch patterns
- The **else** of a conditional is on the same line as the closing brace of the first command body. It is followed by the opening brace of the second command body. Thus the form of the conditional is:

```
if { condition } {
    statements
} else {
    statements
}
```

The form of the conditional statement with an **elseif** is:

```
if { condition } {
    statements
} elseif { condition } {
    statements
} else {
    statements
}
```

- The general form of the **switch** statement is:

```
switch [flags] value {
    a   {
        statements
        }
    b   {
        statements
        }
    default {
        statements
        }
}
```

If the actions are very short and nearly identical in all cases, an alternate form of the switch statement is acceptable:

```
switch [flags] value {
    a   {set x $aVar}
    b   {set x $bVar}
    c   {set x $cVar}
}
```

- Comments have the same indentation level as the section of code to which they refer (see *Comments*, p.478).
- Opening body braces ({) have no specific indentation; they follow the command on the same line.

Comments

- Place comments within code so that they precede the section of code to which they refer and have the same level of indentation. Separate such comments from the code by a single blank line.
  - Begin single-line comments with the pound symbol as in the following:

```
# This is the correct format for a single-line comment

set foo 0
```

  - Multi-line comments have each line beginning with the pound symbol as in the example below. Do not use a backslash to continue a comment across lines.

```
# This is the CORRECT format for a multiline comment
# in a section of code.

set foo 0

# This is the INCORRECT format for a multiline comment \
in a section of code.

set foo 0
```

- Comments on global variables appear on the same line as the variable declaration, using the semicolon (;) character:

```
set day    night    ;# This is a global variable
```

D.4.6 Naming Conventions

The following conventions define the standards for naming modules, routines and variables. The purpose of these conventions is uniformity and readability of code.

- When creating names, remember that code is written once but read many times. Make names meaningful and readable. Avoid obscure abbreviations.

- Names of routines and variables are composed of upper- and lowercase characters and no underbars. Capitalize each "word" except the first:

  aVariableName

- Every module has a short prefix (two to five characters). The prefix is attached to the module name and to all externally available procedures and variables. (Names that are not available externally need not follow this convention.)

fooLib.tcl	module name
fooObjFind	procedure name
fooCount	variable name

- Names of procedures follow the *module-noun-verb* rule. Start the procedure name with the module prefix, followed by the noun or object that the procedure manipulates. Conclude the name with the verb or action that the procedure performs:

fooObjFind	foo - object - find
sysNvRamGet	system - non volatile RAM - get
taskInfoGet	task - info - get

D.4.7 Tcl Style

The following conventions define additional standards of programming style:

- **Comments:** Insufficiently commented code is unacceptable.
- **Procedure Length:** Procedures should have a reasonably small number of lines, less than 50 if possible.
- **Case Statement:** Do not use the **case** keyword. Use **switch** instead.
- **expr and Control Flow Commands:** Do not use **expr** in commands such as **if**, **for** or **while** except to convert a variable from one format to another:

  CORRECT:

  ```
  if {$id != 0} {
  ```

  CORRECT:

  ```
  if {[expr $id] != 0} {
  ```

  INCORRECT:

  ```
  if {[expr $id != 0]} {
  ```

- **expr and incr:** Do not use **expr** to increment or decrement the value of a variable. Use **incr** instead.

  CORRECT:

  ```
  incr index
  ```

  CORRECT:

  ```
  incr index -4
  ```

  INCORRECT:

  ```
  set index [expr $index + 1]
  ```

- **wtxPath and wtxHostType**: Use these routines when developing tools for Tornado. With no arguments, **wtxPath** returns the value of the environment variable **WIND_BASE** with a "/" appended. With an argument list, the result of **wtxPath** is an absolute path rooted in **WIND_BASE** with each argument as a directory segment. Use this command in Tornado tools to read resource files. The **wtxHostType** call returns the host-type string for the current process (the environment variable **WIND_HOST_TYPE**. if properly set, has the same value). For example:

  ```
  source [wtxPath host resource tcl]wtxcore.tcl
  set backenddir [wtxPath host [wtxHostType] lib backend]*
  ```

- **catch Command:** The **catch** command is very useful to intercept errors raised by underlying procedures so that a script does not abort prematurely. However, use the **catch** command with caution. It can obscure the real source of a problem, thus causing errors that are particularly hard to diagnose. In particular, do not use **catch** to capture the return value of a command without testing it. Note also that if the intercepted error cannot be handled, the error must be resubmitted exactly as it was received (or translated to one of the defined errors in the current procedure):

  CORRECT:

  ```
  if [catch "dataFetch $list" result] {
          if {$result == "known problem"} {
              specialCaseHandle
          } else {
              error $result
          }
  ```

  INCORRECT:

  ```
  catch "dataFetch $list" result
  ```

- **if then else Statement:** In an **if** command, you may omit the keyword **then** before the first command body; but do not omit **else** if there is a second command body.

  CORRECT:

```
if {$id != 0} {
        ...
    } else {
        ...
    }
```

  INCORRECT:

```
if {$id !=0} then {
        ...
    } {
        ...
    }
```

- **Return Values:** Tcl procedures only return strings; whatever meaning the string has (a list for instance) is up to the application. Therefore each constant value that a procedure can return must be described in the procedure documentation, in the *RETURNS:* block. If a complex element is returned, provide a complete description of the element layout. Do not use the **return** statement to indicate that an abnormal situation has occurred; use the **error** statement in that situation.

  The following illustrates a complex return value consisting of a description:

```
# Return a list of 11 items: vxTicks taskId status priority pc
# sp errno timeout entry priNormal name

return [concat [lrange $tiList 0 1] [lrange $tiList 3 end]]
```

  The following illustrates and simple return value:

```
# This code checks whether the VxMP component is installed:

if [catch "wtxSymFind -name smObjPoolMinusOne" result] {
    if {[wtxErrorName $result] == "SYMTBL_SYMBOL_NOT_FOUND"} {
        return -1       # VxMP is not installed
    } else {
        error $result
    }
} else {
    return 0             # VxMP is installed
}
```

- **Error Conditions:** The Tcl **error** command raises an error condition that can be trapped by the **catch** command. If not caught, an error condition terminates script execution. For example:

```
if {$defaultTaskId == 0} {
    error "No default task has been established."
}
```

Because every error message and error code must be described in the procedure header in the *ERRORS:* block, it is sometimes useful to call **error** in order to replace an underlying error message with an error expressed in terms directly relevant to the current procedure. For example:

```
if [catch "wtxObjModuleLoad $periodModule" status] {
    error "Cannot add period support module to Target ($status)"
}
```

E

X Resources

E.1 Predefined X Resource Collections

The following X resource settings are described in *2.3.4 X Resource Settings*, p.23:

```
Browser*customization
CrossWind*customization
Dialog*customization
Launch*customization
```

When you set these properties to **-color** or **-grayscale** (or leave them unset, for a monochrome display), the Tornado tools start up with X resource definitions tailored by the Tornado designers for each of those three kinds of display.

E.2 Resource Definition Files

If you wish to exercise more detailed control over the X resources (colors, fonts, bitmaps, and so on) used by Tornado, refer to the X resource files in the Tornado distribution (listed in this section) to select the properties you wish to override.

The documentation for X resources used by Tornado consists of comments in the resource files themselves.

WARNING: The names and values of all X resource strings other than the ***customization** properties are subject to change from one Tornado release to the next.

If you choose to override Tornado X resource values, we recommend that you do not edit these files in place; instead, use them as references, and override the resource settings as you wish in the **.Xdefaults** or **.Xresources** file in your home directory.

The following files contain the Tornado X resource definitions:

*installDir***/host/resource/app-defaults/Tornado**
Comprehensive definitions for all resources common to all the Tornado tools.

*installDir***/host/resource/app-defaults/Tornado-grayscale**
Grayscale-monitor overrides for resources common to all the Tornado tools.

*installDir***/host/resource/app-defaults/***toolName*
Comprehensive definitions for all resources specific to the Tornado tool *toolName* (where *toolName* is one of **Browser**, **CrossWind**, **Dialog**, or **Launch**).

*installDir***/host/resource/app-defaults/***toolName***-color**
Color-monitor overrides specific to a particular Tornado tool.

*installDir***/host/resource/app-defaults/***toolName***-grayscale**
Grayscale-monitor overrides specific to a particular Tornado tool.

F

VxWorks Initialization Timeline

F.1 Introduction

This section covers the initialization sequence for VxWorks in a typical development configuration. The steps are described in sequence of execution. This is not the only way VxWorks can be bootstrapped on a particular processor. There are often more efficient or robust techniques unique to a particular processor or hardware; consult your hardware's documentation.

For final production, the sequence can be revisited to include diagnostics or to remove some of the generic operations that are required for booting a development environment, but that are unnecessary for production. This description can provide only an approximate guide to the processor initialization sequence and does not document every exception to this time-line.

The early steps of the initialization sequence are slightly different for ROM-based versions of VxWorks; for information, see *F.9 Initialization Sequence for ROM-Based VxWorks*, p.499.

For a summary of the initialization timeline, see Table F-1. The following sections describe the initialization in detail by routine name. For clarity, the sequence is divided into a number of main steps or function calls. The key routines are listed in the headings and are described in chronological order.

F.2 The VxWorks Entry Point: sysInit( )

The first step in starting a VxWorks system is to load a system image into main memory. This usually occurs as a download from the development host, under the control of the VxWorks boot ROM. Next, the boot ROM transfers control to the VxWorks startup entry point, **sysInit( )**. This entry point is configured by **RAM_LOW_ADRS** in the makefile and in **config.h**. The VxWorks memory layout is different for each architecture; for details, see the appendix that describes your architecture.

The entry point, **sysInit( )**, is in the system-dependent assembly language module, **sysALib.s**. It locks out all interrupts, invalidates caches if applicable, and initializes processor registers (including the C stack pointer) to default values. It also disables tracing, clears all pending interrupts, and invokes **usrInit( )**, a C subroutine in the **usrConfig.c** module. For some targets, **sysInit( )** also performs some minimal system-dependent hardware initialization, enough to execute the remaining initialization in **usrInit( )**. The initial stack pointer, which is used only by **usrInit( )**, is set to occupy an area below the system image but above the vector table (if any).

F.3 The Initial Routine: usrInit( )

The **usrInit( )** routine (in **usrConfig.c**) saves information about the boot type, handles all the initialization that must be performed before the kernel is actually started, and then starts the kernel execution. It is the first C code to run in VxWorks. It is invoked in supervisor mode with all hardware interrupts locked out.

Many VxWorks facilities cannot be invoked from this routine. Because there is no task context as yet (no TCB and no task stack), facilities that require a task context cannot be invoked. This includes any facility that can cause the caller to be preempted, such as semaphores, or any facility that uses such facilities, such as **printf( )**. Instead, the **usrInit( )** routine does only what is necessary to create an initial task, **usrRoot( )**. This task then completes the startup.

The initialization in **usrInit( )** includes the following:

Cache Initialization

The code at the beginning of **usrInit( )** initializes the caches, sets the mode of the caches and puts the caches in a safe state. At the end of **usrInit( )**, the instruction and data caches are enabled by default.

Zeroing Out the System bss Segment

The C and C++ languages specify that all uninitialized variables must have initial values of 0. These uninitialized variables are put together in a segment called *bss*. This segment is not actually loaded during the bootstrap, because it is known to be zeroed out. Because **usrInit( )** is the first C code to execute, it clears the section of memory containing *bss* as its very first action. While the VxWorks boot ROMs clear all memory, VxWorks does not assume that the boot ROMs are used.

Initializing Interrupt Vectors

The exception vectors must be set up before enabling interrupts and starting the kernel. First, **intVecBaseSet( )** is called to establish the vector table base address.

NOTE: There are exceptions to this in some architectures; see the appendix that describes your architecture for details.

After **intVecBaseSet( )** is called, the routine **excVecInit( )** initializes all exception vectors to default handlers that safely trap and report exceptions caused by program errors or unexpected hardware interrupts.

Initializing System Hardware to a Quiescent State

System hardware is initialized by calling the system-dependent routine **sysHwInit( )**. This mainly consists of resetting and disabling hardware devices that can cause interrupts after interrupts are enabled (when the kernel is started). This is important because the VxWorks ISRs (for I/O devices, system clocks, and so on), are not connected to their interrupt vectors until the system initialization is completed in the **usrRoot( )** task. However, do not attempt to connect an interrupt

handler to an interrupt during the **sysHwInit( )** call, because the memory pool is not yet initialized.

F.4 Initializing the Kernel

The **usrInit( )** routine ends with calls to two kernel initialization routines:

usrKernelInit( ) (defined in **usrKernel.c**)
calls the appropriate initialization routines for each of the specified optional kernel facilities (see Table F-1 for a list).

kernelInit( ) (part of **kernelLib.c**)
initiates the multitasking environment and never returns. It takes the following parameters:

- The application to be spawned as the "root" task, typically **usrRoot( )**.
- The stack size.
- The start of usable memory; that is, the memory after the main text, data, and *bss* of the VxWorks image. All memory after this area is added to the system memory pool, which is managed by **memPartLib**. Allocation for dynamic module loading, task control blocks, stacks, and so on, all come out of this region. See *F.5 Initializing the Memory Pool*, p.489.
- The top of memory as indicated by **sysMemTop( )**. If a contiguous block of memory is to be preserved from normal memory allocation, pass **sysMemTop( )** less the reserved memory.
- The interrupt stack size. The interrupt stack corresponds to the largest amount of stack space any interrupt-level routine uses, plus a safe margin for the nesting of interrupts.
- The interrupt lock-out level. For architectures that have a *level* concept, it is the maximum level. For architectures that do not have a level concept, it is the mask to disable interrupts. See the appendix that describes your architecture for details.

kernelInit( ) calls **intLockLevelSet( )**, disables round-robin mode, and creates an interrupt stack if supported by the architecture. It then creates a root stack and TCB from the top of the memory pool, spawns the root task, **usrRoot( )**, and terminates

the **usrInit( )** thread of execution. At this time, interrupts are enabled; it is critical that all interrupt sources are disabled and pending interrupts cleared.

F.5 Initializing the Memory Pool

VxWorks includes a memory allocation facility, in the module **memPartLib**, that manages a pool of available memory. The **malloc( )** routine allows callers to obtain variable-size blocks of memory from the pool. Internally, VxWorks uses **malloc( )** for dynamic allocation of memory. In particular, many VxWorks facilities allocate data structures during initialization. Therefore, the memory pool must be initialized before any other VxWorks facilities are initialized.

Note that the Tornado target server manages a portion of target memory to support downloading of object modules and other development functions. VxWorks makes heavy use of **malloc( )**, including allocation of space for loaded modules, allocation of stacks for spawned tasks, and allocation of data structures on initialization. You are also encouraged to use **malloc( )** to allocate any memory your application requires. Therefore, it is recommended that you assign to the VxWorks memory pool all unused memory, unless you must reserve some fixed absolute memory area for a particular application use.

The memory pool is initialized by **kernelInit( )**. The parameters to **kernelInit( )** specify the start and end address of the initial memory pool. In the default **usrInit( )** distributed with VxWorks, the pool is set to start immediately following the end of the booted system, and to contain all the rest of available memory.

The extent of available memory is determined by **sysMemTop( )**, which is a system-dependent routine that determines the size of available memory. If your system has other noncontiguous memory areas, you can make them available in the general memory pool by later calling **memAddToPool( )** in the **usrRoot( )** task.

F.6 The Initial Task: usrRoot( )

When the multitasking kernel starts executing, all VxWorks multitasking facilities are available. Control is transferred to the **usrRoot( )** task and the initialization of the system can be completed. For example, **usrRoot( )** performs the following:

- initialization of the system clock
- initialization of the I/O system and drivers
- creation of the console devices
- setting of standard in and standard out
- installation of exception handling and logging
- initialization of the pipe driver
- initialization of standard I/O
- creation of file system devices and installation of disk drivers
- initialization of floating-point support
- initialization of performance monitoring facilities
- initialization of the network
- initialization of optional facilities
- initialization of WindView (see the *WindView User's Guide*)
- initialization of target agent
- execution of a user-supplied startup script

To review the complete initialization sequence within **usrRoot( )**, see *installDir*/**target/config/all/ usrConfig.c**.

Modify these initializations to suit your configuration. The meaning of each step and the significance of the various parameters are explained in the following sections.

Initialization of the System Clock

The first action in the **usrRoot( )** task is to initialize the VxWorks clock. The system clock interrupt vector is connected to the routine **usrClock( )** (described in *F.7 The System Clock Routine: usrClock( )*, p.496) by calling **sysClkConnect( )**. Then, the system clock rate (usually 60Hz) is set by **sysClkRateSet( )**. Most boards allow clock rates as low as 30Hz (some even as low as 1Hz), and as high as several thousand Hz. High clock rates (>1000Hz) are not desirable, because they can cause system *thrashing*.[1]

1. *Thrashing* occurs when clock interrupts are so frequent that the processor spends too much time servicing the interrupts, and no application code can run.

The timer drivers supplied by Wind River include a call to **sysHwInit2( )** as part of the **sysClkConnect( )** routine. Wind River BSPs use **sysHwInit2( )** to perform further board initialization that is not completed in **sysHwInit( )**. For example, an **intConnect( )** of ISRs can take place here, because memory can be allocated now that the system is multitasking.

Initialization of the I/O System

If **INCLUDE_IO_SYSTEM** is defined in **configAll.h**, the VxWorks I/O system is initialized by calling the routine **iosInit( )**. The arguments specify the maximum number of drivers that can be subsequently installed, the maximum number of files that can be open in the system simultaneously, and the desired name of the "null" device that is included in the VxWorks I/O system. This null device is a "bit-bucket" on output and always returns end-of-file for input.

The inclusion or exclusion of **INCLUDE_IO_SYSTEM** also affects whether the console devices are created, and whether standard in, standard out, and standard error are set; see the next two sections for more information.

Creation of the Console Devices

If the driver for the on-board serial ports is included (**INCLUDE_TTY_DEV**), it is installed in the I/O system by calling the driver's initialization routine, typically **ttyDrv( )**. The actual devices are then created and named by calling the driver's device-creation routine, typically **ttyDevCreate( )**. The arguments to this routine includes the device name, a serial I/O channel descriptor (from the BSP), and input and output buffer sizes.

The macro **NUM_TTY** specifies the number of *tty* ports (default is 2), **CONSOLE_TTY** specifies which port is the console (default is 0), and **CONSOLE_BAUD_RATE** specifies the bps rate (default is 9600). These macros are specified in **configAll.h**, but can be overridden in **config.h** for boards with a nonstandard number of ports.

PCs can use an alternative console with keyboard input and VGA output; see your PC workstation documentation for details.

Setting of Standard In, Standard Out, and Standard Error

The system-wide standard in, standard out, and standard error assignments are established by opening the console device and calling **ioGlobalStdSet( )**. These assignments are used throughout VxWorks as the default devices for communicating with the application developer. To make the console device an interactive terminal, call **ioctl( )** to set the device options to **OPT_TERMINAL**.

Installation of Exception Handling and Logging

Initialization of the VxWorks exception handling facilities (supplied by the module **excLib**) and logging facilities (supplied by **logLib**) takes place early in the execution of the root task. This facilitates detection of program errors in the root task itself or in the initialization of the various facilities.

The exception handling facilities are initialized by calling **excInit( )** when **INCLUDE_EXC_HANDLING** and **INCLUDE_EXC_TASK** are defined. The **excInit( )** routine spawns the exception support task, **excTask( )**. Following this initialization, program errors causing hardware exceptions are safely trapped and reported, and hardware interrupts to uninitialized vectors are reported and dismissed. The VxWorks signal facility, used for task-specific exception handling, is initialized by calling **sigInit( )** when **INCLUDE_SIGNALS** is defined.

The logging facilities are initialized by calling **logInit( )** when **INCLUDE_LOGGING** is defined. The arguments specify the file descriptor of the device to which logging messages are to be written, and the number of log message buffers to allocate. The logging initialization also includes spawning the logging task, **logTask( )**.

Initialization of the Pipe Driver

If named pipes are desired, define **INCLUDE_PIPE** in **configAll.h** so that **pipeDrv( )** is called automatically to initialize the pipe driver. Tasks can then use pipes to communicate with each other through the standard I/O interface. Pipes must be created with **pipeDevCreate( )**.

Initialization of Standard I/O

VxWorks includes an optional *standard I/O* package when **INCLUDE_STDIO** is defined.

Creation of File System Devices and Initialization of Device Drivers

Many VxWorks configurations include at least one disk device or RAM disk with a dosFs, rt11Fs, or rawFs file system. First, a disk driver is installed by calling the driver's initialization routine. Next, the driver's device-creation routine defines a device. This call returns a pointer to a **BLK_DEV** structure that describes the device.

The new device can then be initialized and named by calling the file system's device-initialization routine—**dosFsDevInit( )**, **rt11FsDevInit( )**, or **rawFsDevInit( )**—when the respective constants **INCLUDE_DOSFS**, **INCLUDE_RT11FS**, and **INCLUDE_RAWFS** are defined. (Before a device can be initialized, the file system module must already be initialized with **dosFsInit( )**, **rt11FsInit( )**, or **rawFsInit( )**.) The arguments to the file system device-initialization routines depend on the particular file system, but typically include the device name, a pointer to the **BLK_DEV** structure created by the driver's device-creation routine, and possibly some file-system-specific configuration parameters.

Initialization of Floating-Point Support

Support for floating-point *I/O* is initialized by calling the routine **floatInit( )** when **INCLUDE_FLOATING_POINT** is defined in **configAll.h**. Support for floating-point *coprocessors* is initialized by calling **mathHardInit( )** when **INCLUDE_HW_FP** is defined. Support for software floating-point *emulation* is initialized by calling **mathSoftInit( )** when **INCLUDE_SW_FP** is defined. See the appropriate architecture appendix for details on your processor's floating-point support.

Inclusion of Performance Monitoring Tools

VxWorks has two built-in performance monitoring tools. A task activity summary is provided by **spyLib**, and a subroutine execution timer is provided by **timexLib**. These facilities are included by defining the macros **INCLUDE_SPY** and **INCLUDE_TIMEX**, respectively, in **configAll.h**.

Initialization of the Network

Before the network can be used, it must be initialized with the routine **usrNetInit( )**, which is called by **usrRoot( )** when the constant **INCLUDE_NET_INIT** is defined in one of the configuration header files. (The source for **usrNetInit( )** is in *installDir*/**target/src/config/usrNetwork.c**.) The routine **usrNetInit( )** takes a configuration string as an argument. This configuration string is usually the "boot line" that is specified to the VxWorks boot ROMs to boot the system (see *Tornado Getting Started*). Based on this string, **usrNetInit( )** performs the following:

- Initializes network subsystem by calling the routine **netLibInit( )**.
- Attaches and configures appropriate network drivers.
- Adds gateway routes.
- Initializes the remote file access driver **netDrv**, and adds a remote file access device.
- Initializes the remote login facilities.
- Optionally initializes the Remote Procedure Calls (RPC) facility.
- Optionally initializes the Network File System (NFS) facility.

As noted previously, the inclusion of some of these network facilities is controlled by definitions in **configAll.h**. The network initialization steps are described in the *VxWorks Network Programmer's Guide*.

Initialization of Optional Products and Other Facilities

Shared memory objects are provided with the optional product VxMP. Before shared memory objects can be used, they must be initialized with the routine **usrSmObjInit( )** (in *installDir*/**target/src/config/usrSmObj.c**), which is called from **usrRoot( )** if **INCLUDE_SM_OBJ** is defined.

CAUTION: The shared memory objects library requires information from fields in the VxWorks boot line. The functions are contained in the **usrNetwork.c** file. If no network services are included, **usrNetwork.c** is not included and the shared memory initialization fails. The project facility calculates all dependencies but if you are using manual configuration, either add **INCLUDE_NETWORK** to **configAll.h** or extract the bootline cracking routines from **usrNetwork.c** and include them elsewhere.

Basic MMU support is provided if **INCLUDE_MMU_BASIC** is defined. Text protection, vector table protection, and a virtual memory interface are provided with the optional product VxVMI, if **INCLUDE_MMU_FULL** is defined. The MMU is initialized by the routine **usrMmuInit( )**, located in *installDir*/**target/src/config/usrMmuInit.c**. If the macros **INCLUDE_PROTECT_TEXT** and **INCLUDE_PROTECT_VEC_TABLE** are also defined, text protection and vector table protection are initialized.

Wind River compilers support the C++ language. Run-time C++ support is enabled by defining **INCLUDE_CPLUS**. Additional C++ facilities can also be included by defining the appropriate **INCLUDE_CPLUS**_*XXX* macros. For more details see the *VxWorks Programmer's Guide: C++ Development*.

Initialization of WindView

Kernel instrumentation is provided with the optional product WindView. It is initialized in **usrRoot( )** when **INCLUDE_WINDVIEW** is defined in **configAll.h**. Other WindView configuration constants control particular initialization steps; see the *WindView User's Guide: Configuring WindView.*

Initialization of the Target Agent

If **INCLUDE_WDB** is defined, **wdbConfig( )** in *installDir*/**target/src/config/usrWdb.c** is called. This routine initializes the agent's

communication interface, then starts the agent. For information on configuring the agent, see *5.3 Configuring VxWorks*, p.186.

Execution of a Startup Script

The **usrRoot( )** routine executes a user-supplied startup script if the target-resident shell is configured into VxWorks, **INCLUDE_STARTUP_SCRIPT** is defined, and the script's file name is specified at boot time with the startup script parameter (see *Tornado Getting Started*). If the parameter is missing, no startup script is executed.

F.7 The System Clock Routine: usrClock( )

Finally, the system clock ISR **usrClock( )** is attached to the system clock timer interrupt by the **usrRoot( )** task described *F.6 The Initial Task: usrRoot( )*, p.490. The **usrClock( )** routine calls the kernel clock tick routine **tickAnnounce( )**, which performs OS bookkeeping. You can add application-specific processing to this routine.

F.8 Initialization Summary

Table F-1 shows a summary of the entire VxWorks initialization sequence for typical configurations. For a similar summary applicable to ROM-based VxWorks systems, see *F.9 Initialization Sequence for ROM-Based VxWorks*, p.499.

Table F-1 **VxWorks Run-time System Initialization Sequence**

Routine	Activity	File
sysInit( )	(a) lock out interrupts	**sysALib.s**
	(b) invalidate caches, if any	
	(c) initialize system interrupt tables with default stubs (i960 only)	

Table F-1 **VxWorks Run-time System Initialization Sequence** *(Continued)*

Routine	Activity	File
	(d) initialize system fault tables with default stubs (i960 only)	
	(e) initialize processor registers to known default values	
	(f) disable tracing	
	(g) clear all pending interrupts	
	(h) invoke **usrInit( )** specifying boot type	
usrInit( )	(a) invoke optional **sysHwInit0( )**	**usrConfig.c**
	(b) invoke **cacheInit( )**	
	(c) zero *bss* (uninitialized data)	
	(d) save **bootType** in **sysStartType**	
	(e) invoke **excVecInit( )** to initialize all system and default interrupt vectors	
	(f) invoke **sysHwInit( )**	
	(g) invoke **usrKernelInit( )**	
	(h) enable caches	
	(i) invoke **kernelInit( )**	
usrKernelInit( )	The following routines are invoked if their configuration constants are defined.	**usrKernel.c**
	(a) **classLibInit( )**	
	(b) **taskLibInit( )**	
	(c) **taskHookInit( )**	
	(d) **semBLibInit( )**	
	(e) **semMLibInit( )**	
	(f) **semCLibInit( )**	
	(g) **semOLibInit( )**	

Table F-1 **VxWorks Run-time System Initialization Sequence** *(Continued)*

Routine	Activity	File
	(h) **wdLibInit( )**	
	(i) **msgQLibInit( )**	
	(j) **qInit( )** for all system queues	
	(k) **workQInit( )**	
kernelInit( )	Initialize and start the kernel.	**kernelLib.c**
	(a) invoke **intLockLevelSet( )**	
	(b) create root stack and TCB from top of memory pool	
	(c) invoke **taskInit( )** for **usrRoot( )**	
	(d) invoke **taskActivate( )** for **usrRoot( )**	
	(e) **usrRoot( )**	
usrRoot( )	Initialize I/O system, install drivers, and create devices as specified in **configAll.h** and **config.h**. See **usrConfig.c** for a complete list of optional kernel facilities initialized.	**usrConfig.c**
	(a) Initialize memory partitions and MMU	
	(b) **sysClkConnect( )**	
	(c) **sysClkRateSet( )**	
	(d) **selectInit( )**	
	(e) **iosInit( )**	
	(f) if (**INCLUDE_TTY_DEV** and **NUM_TTY**) **ttyDrv( )**, then establish console port, **STD_IN**, **STD_OUT**, **STD_ERR**	
	(g) initialize exception handling with **excInit( )**, **logInit( )**, **sigInit( )**	
	(h) initialize the pipe driver with **pipeDrv( )**	
	(i) **stdioInit( )**	

Table F-1 **VxWorks Run-time System Initialization Sequence** *(Continued)*

Routine	Activity	File
	(j) **mathSoftInit( )** or **mathHardInit( )**	
	(k) **wdbConfig( )**: configure and initialize target agent	
	(l) run startup script if target-resident shell is configured	

F.9 Initialization Sequence for ROM-Based VxWorks

The early steps of system initialization are somewhat different for the ROM-based versions of VxWorks: on most target architectures, the two routines **romInit( )** and **romStart( )** execute instead of the usual VxWorks entry point, **sysInit( )**.

ROM Entry Point: romInit( )

At power-up the processor begins executing at **romInit( )** (defined in *installDir*/**target/config/***bspname*/**romInit.s**). The **romInit( )** routine disables interrupts, puts the boot type (cold/warm) on the stack, performs hardware-dependent initialization (such as clearing caches or enabling DRAM), and branches to **romStart( )**. The stack pointer is initialized to reside below the data section in the case of ROM-resident versions of VxWorks (in RAM versions, the stack pointer instead resides below the text section).

Copying the VxWorks Image: romStart( )

Next, the **romStart( )** routine (in *installDir*/**target/config/all/bootInit.c**) loads the VxWorks system image into RAM. If the ROM-resident version of VxWorks is selected, the data segment is copied from ROM to RAM and memory is cleared. If VxWorks is not ROM resident, all of the text and code segment is copied and decompressed from ROM to RAM, to the location defined by **RAM_HIGH_ADRS** in **Makefile**. If VxWorks is neither ROM resident nor compressed, the entire text and data segment is copied without decompression straight to RAM, to the location defined by **RAM_LOW_ADRS** in **Makefile**.

Overall Initialization for ROM-Based VxWorks

Beyond **romStart( )**, the initialization sequence for ROM-based VxWorks resembles the normal sequence, continuing with the **usrInit( )** call.

Table F-2 summarizes the complete initialization sequence. For details on the steps after **romInit( )** and **romStart( )**, see *F.8 Initialization Summary*, p.496.

Table F-2 **ROM-Based VxWorks Initialization Sequence**

Routine	Activity	File
1. **romInit( )**	(a) disable interrupts	**romInit.s**
	(b) save boot type (cold/warm)	
	(c) hardware-dependent initialization	
	(d) branch to **romStart( )**	
2. **romStart( )**	(a) copy data segment from ROM to RAM; clear memory	**bootInit.c**
	(b) copy code segment from ROM to RAM, decompressing if necessary	
	(c) invoke **usrInit( )** with boot type	
3. **usrInit( )**	Initial routine.	**usrConfig.c**
4. **usrKernelInit( )**	Routines invoked if the corresponding configuration constants are defined.	**usrKernel.c**
5. **kernelInit( )**	Initialize and start the kernel.	**kernelLib.c**
6. **usrRoot( )**	Initialize I/O system, install drivers, and create devices as configured in **configAll.h** and **config.h**.	**usrConfig.c**
Application routine	Application code.	Application source file

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X