Tornado™

User's Guide

(UNIX Version)

2.0

Edition 1

WindRiver® SYSTEMS
An ISO 9001 Registered Company

Corporate Headquarters
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
USA

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

Europe
Wind River Systems, S.A.R.L.
19, Avenue de Norvège
Immeuble B4, Bâtiment 3
Z.A. de Courtaboeuf 1
91953 Les Ulis Cédex
FRANCE

telephone: 33-1-60-92-63-00
facsimile: 33-1-60-92-63-15

Japan
Wind River Systems K.K.
Ebisu Prime Square Tower 5th Fl.
1-1-39 Hiroo
Shibuya-ku
Tokyo 150-0012
JAPAN

telephone: 81-3-5778-6001
facsimile: 81-3-5778-6002

CUSTOMER SUPPORT

	Telephone	E-mail	Fax
Corporate:	800/872-4977 toll free, U.S. & Canada 510/748-4100 direct	support@wrs.com	510/749-2164
Europe:	33-1-69-07-78-78	support@wrsec.fr	33-1-69-07-08-26
Japan:	011-81-3-5467-5900	support@kk.wrs.com	011-81-3-5467-5877

If you purchased your Wind River Systems product from a distributor, please contact your distributor to determine how to reach your technical support organization.

Please provide your license number when contacting Customer Support.

Tornado User's Guide, 2.0 (UNIX Version)
Edition 1
6 May 99
Part #: DOC-12772-ZD-01

	Documentation Guide	xix
1	**Overview**	1
2	**Setup and Startup**	17
3	**Launcher**	65
4	**Projects**	91
5	**Shell**	149
6	**Browser**	207
7	**Debugger**	233
8	**Customization**	279
9	**Customer Service**	293
	Appendices	303
A	**Directories and Files**	305
B	**Tcl**	317
C	**Tornado Tools Reference**	339
D	**Utilities Reference**	437
E	**X Resources**	471
	Index	473

Contents

Documentation Guide xix

Overview xix
Release Information xx
Tornado Host-Tool Documentation xxi
VxWorks Target-OS Documentation xxii
Online Documentation xxiii
Free Software Foundation Documentation xxiii
Documentation Conventions xxiv

1 Overview 1

1.1 Introduction 1

1.2 Cross-Development with Tornado 3

1.3 VxWorks Target Environment 4

1.4 Tornado Host Tools 5

Launcher 5
Project Management 6
Compiler 6
WindSh Command Shell 6
CrossWind Debugger 7
Browser 8
WindView Software Logic Analyzer 8
VxWorks Target Simulator 9

1.5 Host-Target Interface 9
Target Agent 10
Tornado Target Server 11
Tornado Registry 11
Virtual I/O 12

1.6 Online Documentation 12

1.7 Extending Tornado 13
Published Application Program Interfaces 13
Optional Products 13
Third-Party Products 14

1.8 Customer Services 14
Customer Support 14
Training 14

2 Setup and Startup 17

2.1 Introduction 17

2.2 Setting up the Tornado Registry 18

2.3 The Tornado Host Environment 20
2.3.1 Environment Variables for Tornado Components 20
2.3.2 Environment Variable 22
2.3.3 Environment Variables for Convenience 22
2.3.4 X Resource Settings 23

2.4 Target Setup 24
2.4.1 Example Target Configurations 24
2.4.2 Networking the Host and Target 26
Initializing the Host Network Software 26
Establishing the VxWorks System Name and Address 27
Giving VxWorks Access to the Host 27
2.4.3 Configuring the Target Hardware 28

Boot ROMs and Other Boot Media 28
Setting Board Jumpers 29
Board Installation and Power 29
Connecting the Cables 30

2.5 Host-Target Communication Configuration 31

2.5.1 Network Connections 33

Configuring the Target Agent for Network Connection 34

2.5.2 END Connections 34

Configuring the Target Agent for END Connection 34

2.5.3 Serial-Line Connections 34

Configuring the Target Agent for Serial Connection 35
Configuring the Boot Program for Serial Connection 35
Testing the Connection 35
Starting the Target Server 36

2.5.4 The NetROM ROM-Emulator Connection 37

Configuring the Target Agent for NetROM 38
Configuring the NetROM 38
Starting the Target Server 42
Troubleshooting the NetROM ROM-Emulator Connection 42

2.6 Booting VxWorks 45

2.6.1 Default Boot Process 45

2.6.2 Entering New Boot Parameters 46

2.6.3 Boot Program Commands 48

2.6.4 Description of Boot Parameters 48

2.6.5 Booting With New Parameters 51

2.6.6 Alternate Booting Procedures 52

2.6.7 Booting a Target Without a Network 53

2.6.8 Rebooting 54

2.7 Connecting a Tornado Target Server 54

2.8 Launching Tornado 55

2.9 Tornado Interface Conventions 56

2.10 Troubleshooting 58

2.10.1 Things to Check 58

Hardware Configuration 58
Booting Problems 59
Target-Server Problems 62

2.10.2 Technical Support 63

3 Launcher 65

3.1 Introduction 65

3.2 The Tornado Launcher 65

3.3 Anatomy of the Launcher Window 66

3.4 Tools and Targets 68

3.4.1 Selecting a Target Server 69

3.4.2 Launching a Tool 71

3.5 Managing Target Servers 72

3.5.1 Configuring a Target Server 73

Simple Server Configuration for Networked Targets 75
Simple Server Configuration for WDB Serial Targets 75
Saved Configurations 75
Target-Server Action Buttons 76
Target-Server Configuration Options 76

3.5.2 Sharing and Reserving Target Servers 82

3.6 Tornado Central Services 83

3.6.1 Support and Information 83

3.6.2 Administrative Activities 83

3.7 Tcl: Customizing the Launcher 84

3.7.1 Tcl: Launcher Initialization File 85

3.7.2 Tcl: Launcher Customization Examples 85
Re-Reading Tcl Initialization 86
Quit Launcher Without Prompting 86
An Open Command for the File Menu 87

4 Projects 91

4.1 Introduction 91

4.2 Creating a Downloadable Application 95
4.2.1 Creating a Project for a Downloadable Application 97
4.2.2 Project Files for a Downloadable Application 100
4.2.3 Working With Application Files 101
Creating, Adding, and Removing Application Files 102
Displaying and Modifying File Properties 103
Opening, Saving, and Closing Files 103
4.2.4 Building a Downloadable Application 104
Calculating Makefile Dependencies 105
Build Specifications 106
Building an Application 108
4.2.5 Downloading and Running an Application 109
4.2.6 Adding and Removing Projects 110

4.3 Creating a Custom VxWorks Image 110
4.3.1 Creating a Project for VxWorks 111
4.3.2 Project Files for VxWorks 113
4.3.3 Configuring VxWorks Components 117
Finding VxWorks Components and Configuration Macros 118
Displaying Descriptions and Online Help for Components 118
Including and Excluding Components 118
Component Conflicts 122
Changing Component Parameters 124
Estimating Total Component Size 124
4.3.4 Selecting the VxWorks Image Type 124

4.3.5 Building VxWorks ... 126
4.3.6 Booting VxWorks ... 127

4.4 Creating a Bootable Application ... 127
4.4.1 Using Automated Scaling of VxWorks ... 127
4.4.2 Adding Application Initialization Routines ... 128

4.5 Working With Build Specifications ... 128
4.5.1 Changing a Build Specification ... 128
Custom Makefile Rules ... 130
Makefile Macros ... 131
Compiler Options ... 132
Assembler Options ... 133
Link Order Options ... 133
Linker Options ... 134
4.5.2 Creating New Build Specifications ... 135
4.5.3 Selecting a Specification for the Current Build ... 136

4.6 Configuring the Target-Host Communication Interface ... 137
Configuration for an END Driver Connection ... 138
Configuration for Integrated Target Simulators ... 138
Configuration for NetROM Connection ... 138
Configuration for Network Connection ... 140
Configuration for Serial Connection ... 141
Configuration for tyCoDrv Connection ... 141
Scaling the Target Agent ... 141
Configuring the Target Agent for Exception Hooks ... 142
Starting the Agent Before the Kernel ... 142

4.7 Configuring and Building a VxWorks Boot Program ... 144

5 Shell ... 149

5.1 Introduction ... 149

5.2 Using the Shell ... 151
5.2.1 Starting and Stopping the Tornado Shell ... 151

5.2.2 Shell Features 152
5.2.3 Invoking Built-In Shell Routines 156
Task Management 157
Task Information 158
System Information 160
System Modification and Debugging 163
C++ Development 165
Object Display 165
Network Status Display 167
Resolving Name Conflicts between Host and Target 168
5.2.4 Running Target Routines from the Shell 168
5.2.5 Rebooting from the Shell 169
5.2.6 Using the Shell for System Mode Debugging 170
5.2.7 Interrupting a Shell Command 173
5.3 The Shell C-Expression Interpreter 174
5.3.1 I/O Redirection 174
5.3.2 Data Types 175
5.3.3 Lines and Statements 177
5.3.4 Expressions 177
Literals 177
Variable References 178
Operators 178
Function Calls 179
Subroutines as Commands 180
Arguments to Commands 180
Task References 181
5.3.5 The “Current” Task and Address 181
5.3.6 Assignments 182
Typing and Assignment 182
Automatic Creation of New Variables 182
5.3.7 Comments 183
5.3.8 Strings 183

5.3.9 Ambiguity of Arrays and Pointers 184
5.3.10 Pointer Arithmetic 184
5.3.11 C Interpreter Limitations 185
5.3.12 C-Interpreter Primitives 186
5.3.13 Terminal Control Characters 186
5.3.14 Redirection in the C Interpreter 186
Ambiguity Between Redirection and C Operators 188
The Nature of Redirection 188
Scripts: Redirecting Shell I/O 189
C-Interpreter Startup Scripts 190
5.4 C++ Interpretation 191
5.4.1 Overloaded Function Names 191
5.4.2 Automatic Name Demangling 193
5.5 Shell Line Editing 193
5.6 Object Module Load Path 196
5.7 Tcl: Shell Interpretation 198
5.7.1 Tcl: Controlling the Target 199
Tcl: Calling Target Routines 200
Tcl: Passing Values to Target Routines 200
5.7.2 Tcl: Calling Under C Control 200
5.7.3 Tcl: Tornado Shell lnitialization 201
5.8 The Shell Architecture 202
5.8.1 Controlling the Target from the Host 202
5.8.2 Shell Components 204
5.8.3 Layers of Interpretation 205
6 Browser 207
6.1 A System-Object Browser 207

6.2 Starting the Browser 208
6.3 Anatomy of the Target Browser 209
6.4 Browser Menus and Buttons 210
6.5 Data Panels 212
6.6 Object Browsers 213
6.6.1 The Task Browser 214
6.6.2 The Semaphore Browser 215
6.6.3 The Message-Queue Browser 217
6.6.4 The Memory-Partition Browser 218
6.6.5 The Watchdog Browser 219
6.6.6 The Class Browser 220
6.7 The Module Browser 221
6.8 The Vector Table Window 223
6.9 The Spy Window 224
6.10 The Stack-Check Window 226
6.11 Browser Displays and Target Link Speed 227
6.12 Troubleshooting with the Browser 227
6.12.1 Memory Leaks 227
6.12.2 Stack Overflow 228
6.12.3 Memory Fragmentation 228
6.12.4 Priority Inversion 229
6.13 Tcl: the Browser Initialization File 231

7 Debugger 233

7.1 Introduction 233

7.2 Starting CrossWind 234

7.3 A Sketch of CrossWind 234

7.4 CrossWind in Detail 236

7.4.1 Graphical Controls 236

Display Manipulation 237
CrossWind Menus 238
CrossWind Buttons 244

7.4.2 Debugger Command Panel: GDB 252

GDB Initialization Files 253
What Modules to Debug 253
What Code to Display 254
Executing Your Program 255
Application I/O 256
Graphically Enhanced Commands 256
Managing Targets 257
Command-Interaction Facilities 258
Extended Debugger Commands 258
Extended Debugger Variables 259

7.5 System-Mode Debugging 261

7.5.1 Entering System Mode 261

7.5.2 Thread Facilities in System Mode 262

Displaying Summary Thread Information 263
Switching Threads Explicitly 263
Thread-Specific Breakpoints 264
Switching Threads Implicitly 265

7.6 Tcl: Debugger Automation 266

7.6.1 Tcl: A Simple Debugger Example 266

7.6.2 Tcl: Specialized GDB Commands 267

7.6.3 Tcl: Invoking GDB Facilities 268

7.6.4 Tcl: A Linked-List Traversal Macro 271

7.7 Tcl: CrossWind Customization 273
7.7.1 Tcl: Debugger Initialization Files 273
7.7.2 Tcl: Passing Control between the Two CrossWind Interpreters .. 274
7.7.3 Tcl: Experimenting with CrossWind Extensions 275
Tcl: "This" Buttons for C++ 275
Tcl: A List Command for the File Menu 276
Tcl: An Add-Symbols Command for the File Menu 278

8 Customization 279

8.1 Introduction 279

8.2 Setting Download Options 279

8.3 Setting Version Control Options 281

8.4 Customizing the Tools Menu 282
8.4.1 The Customize Tools Dialog Box 283
Macros for Customized Menu Commands 285
8.4.2 Examples of Tools Menu Customization 286
Version Control 287
Alternate Editor 288
Binary Utilities 288
World Wide Web 289

8.5 Alternate Default Editor 290

8.6 Tcl Customization Files 290
Tornado Initialization 290
HTML Help Initialization 291

9 Customer Service 293

9.1 Introduction 293

9.2 WRS Support Services 294

9.2.1 The Customer Information Form 295
9.2.2 Sending a Technical Support Request (TSR) 296
9.3 WRS Broadcasts on the World Wide Web 301
Appendices 303
A Directories and Files 305
A.1 Introduction 305
A.2 Host Directories and Files 306
A.3 Target Directories and Files 308
A.4 Initialization and State-Information Files 314
B Tcl 317
B.1 Why Tcl? 317
B.2 A Taste of Tcl 318
B.2.1 Tcl Variables 319
B.2.2 Lists in Tcl 320
B.2.3 Associative Arrays 321
B.2.4 Command Substitution 321
B.2.5 Arithmetic 322
B.2.6 I/O, Files, and Formatting 322
B.2.7 Procedures 323
B.2.8 Control Structures 324
B.2.9 Tcl Error Handling 324
B.2.10 Integrating Tcl and C Applications 325
B.3 Tcl Coding Conventions 325
B.3.1 Tcl Module Layout 326

B.3.2 Tcl Procedure Layout 327
B.3.3 Tcl Code Outside Procedures 329
B.3.4 Declaration Formats 329
Variables 330
Procedures 330
B.3.5 Code Layout 330
Vertical Spacing 330
Horizontal Spacing 331
Indentation 332
Comments 333
B.3.6 Naming Conventions 334
B.3.7 Tcl Style 334

C Tornado Tools Reference 339

D Utilities Reference 437

E X Resources 471

E.1 Predefined X Resource Collections 471

E.2 Resource Definition Files 471

Index 473

Documentation Guide

Overview

Documentation for Tornado falls into the following general categories:

- **Release Information.** The *Tornado Release Notes* describe changes in the current release, and supported platforms. The files **README.TXT** and **README.HTML** at the root of the CD-ROM, contain last-minute information about the current Tornado release. In addition, the *Release Bulletin* at the following URL contains fixed and known problems lists and other pertinent information:

  http://www.wrs.com/corporate/support/prodbullet/T2.0

  PROBLEMS.TXT lists known problems for this release and **FIXED.TXT** lists problems fixed since the previous release.

- **Tornado Host-Tool Documentation.** These manuals focus principally on the interactive cross-development tools in Tornado. The *Tornado Getting Started Guide* provides instructions on installing Tornado, and a tutorial introduction. The *Tornado User's Guide* (this manual) describes the Tornado development tools and how they are used with the target system. The *Tornado API Programmer's Guide* and the online Tornado API reference describe the interfaces available to extend this development environment.

- **VxWorks Target-OS Documentation.** These manuals describe the VxWorks operating system and associated run-time facilities. The *VxWorks Programmer's Guide* is a comprehensive introduction to VxWorks, with the exception of networking, which is covered separately in the *VxWorks Network Programmer's Guide*. The *VxWorks Reference Manual* contains reference entries for all VxWorks modules and subroutines.

- **Online Documentation.** Most Tornado documentation is available online in HTML format, including the setup information for VxWorks Board Support Packages (BSP)[1].
- **Free Software Foundation Documentation.** These supporting manuals are redistributed by Wind River Systems to provide auxiliary reference information on the compiler, debugger, and associated utilities supplied with Tornado. The *GNU ToolKit User's Guide* contains detailed information about the compiler, and its supporting tools. *GNU Make User's Guide* describes makefiles and **make** usage. *GDB User's Guide* describes the command-line interface to the Tornado debugger.

Release Information

Tornado Release Notes

The *Tornado Release Notes* contains the latest list of supported hosts and targets, as well as information on compatibility with older releases, an outline of new features, and any caveats concerning the current release.

Readme File

The **README.TXT** file (and its online counterpart, **README.HTML**), found at the root of the CD-ROM, contains information about product issues encountered after the *Tornado Release Notes* were printed.

Problem Lists

The URL **http://www.wrs.com/corporate/support/prodbullet/T2.0** contains two files of problem lists:

FIXED.TXT
For each WRS product on the CD-ROM, describes the problems fixed since that product's previous release.

PROBLEMS.TXT
Describes all known problems with WRS products on the CD-ROM at the time of release.

1. A BSP is a VxWorks component that is specific to a given target.

Tornado Host-Tool Documentation

Tornado Getting Started Guide

The *Tornado Getting Started Guide* provides instructions on installing Tornado (and other products), as well as a tutorial introduction to Tornado.

Tornado User's Guide

This manual, the *Tornado User's Guide*, is the central document for the Tornado IDE. It includes:

- A global overview of Tornado and its capabilities.
- Instructions on how to configure your environment and set up communications with a target system.
- Chapters on the Tornado development environment and its major interactive features—the launcher, project facility, shell, browser, and the debugger.
- Appendices on Tornado directories and files, the use of Tcl (Tool Command Language) in Tornado, and reference information for host tools.
- Tcl-customization sections. Many chapters in the *Tornado User's Guide* contain suggestions about how to customize the Tornado tools using Tcl. These discussions are segregated into separate sections so that readers unfamiliar with Tcl can skip them. Titles for these sections always begin with "Tcl:" to indicate the special focus.

Tornado API Programmer's Guide

The *Tornado API Programmer's Guide* is for developers who wish to extend the Tornado development environment. It discusses the Tornado architecture from the perspective of software application program interfaces (APIs) and protocols, and describes how to extend and modify the Tornado tools and integrate them with custom software. It contains descriptive information about the run-time target agent; on its host-system counterpart, the target server; and on the WTX protocol used by the Tornado tools to communicate with the target server.

Tornado API Reference

The *Tornado API Reference* is the reference companion to the *Tornado API Programmer's Guide.* It is available online only (in HTML format).

VxWorks Target-OS Documentation

VxWorks Programmer's Guide

The *VxWorks Programmer's Guide* describes the VxWorks operating system and associated run-time facilities. The *Programmer's Guide* is the best starting point to learn about VxWorks from a problem-solving perspective, because it is organized by the function of VxWorks components. It includes the following topics:

- Basic OS: the fundamentals of the VxWorks kernel and run-time environment.
- I/O System: the VxWorks I/O system and the device drivers that underlie it.
- Local File Systems: VxWorks file systems, including a DOS-compatible file system, the "raw" file system, and the RT-11 file system.
- Optional Products: VxWorks optional products VxMP (shared memory objects), VxVMI (virtual memory interface), and the Wind Foundation Classes (C++ libraries).
- Configuration and Build: how to configure VxWorks for your application by editing configuration files and how to build it from the command line. (The *Tornado User's Guide: Projects* discusses the automated features provided by the project facility GUI.
- Architecture Appendices: special considerations for each supported target-CPU architecture.

VxWorks Network Programmer's Guide

The *VxWorks Network Programmer's Guide* describes the networking facilities available with VxWorks. It includes the following topics:

- Configuring the network stack.
- Booting over the network.
- Using the MUX interface.
- Upgrading 4.3 BSD drivers.
- Appendices: Libraries and subroutine references.

VxWorks Reference Manual

The *VxWorks Reference Manual* consists of reference entries divided into the following sections:

- **Libraries.** Reference descriptions of all VxWorks libraries that apply to all targets. Each entry lists the routines found in a given library, including a one-line synopsis of each, along with a general description of their use. This section also contains entries for the serial, Ethernet, and SCSI drivers available with VxWorks Board Support Packages (BSPs).
- **Subroutines.** Reference descriptions for each of the subroutines found in the libraries in *Libraries*.

Online Documentation

Online Manuals

The Tornado software suite includes a hypertext collection of all Tornado and VxWorks manuals (in HTML format). You can open the online manuals from the Help>Manuals contents menu. An index of reference page names is available from the Help>Manuals index menu option.

Board Support Package (BSP) Documentation

The online manuals contain reference entries for the libraries and subroutines specific to each BSP. These entries include a target information entry, which covers such topics as: what drivers the board uses; how the board should be jumpered for use with VxWorks; the board layout, indicating the location of board jumpers (if applicable) and ROM sockets; and any other board-specific considerations. See Help>Manuals contents>BSP Reference (**wind/docs/BSP_Reference.html**).

Check with your sales representative for a current list of supported BSPs.

Context-Sensitive Help

Help buttons in every Tornado dialog box (and the Help menu in the menu bar) provide information on the Tornado component you are currently executing.

Free Software Foundation Documentation

The following manuals contain Free Software Foundation (FSF) documentation that has been edited to remove information not relevant to Wind River Systems products, and assembled into three volumes.

GNU ToolKit User's Guide

The *Gnu ToolKit User's Guide* is a convenient collection of the manuals for the GNU C and C++ compiler and its supporting tools: the C preprocessor, assembler, static linker and binary utilities.

GNU Make User's Guide

The Free Software Foundation's manual for the **make** utility.

GDB User's Guide

The *GDB User's Guide* is the FSF manual for the command-line interface to the GNU debugger GDB, which is the foundation of the Tornado graphical debugger, CrossWind.

Documentation Conventions

Cross-References

In the Tornado guides, cross-references to a *reference page* or to a *manual entry* for a specified tool or module refer to an entry in the *VxWorks Reference Manual* (for target libraries or subroutines), to the reference appendix in the *Tornado User's Guide* (for host tools), or to their corresponding online versions.

Other cross-references between books take the form *Book Title: Chapter Name*.

Path Names

In general, this manual refers to Tornado directories and files with relative path names starting at the directory **wind**. However, nothing in Tornado assumes or requires this particular path name. Use the path name chosen on your system for Tornado installation.

Screen Displays

The screen displays in this book are for illustrative purposes. They may not correspond exactly to the Tornado environment you see on your computer, because both Tornado and the UNIX environment in which it runs can be customized. Tornado is also designed to permit easy integration with added tools.

Tcl

The Tornado tools make extensive use of Tcl, which allows a great degree of customization. However, it is not necessary to know Tcl in order to use the tools.

Section titles in this manual that begin with *Tcl:* are of interest only to readers who may want to use Tcl to change some aspect of the tool's behavior, and can safely be skipped by other readers. See *B. Tcl*.

Typographical Conventions

Tornado manuals use the font conventions in the following table for special elements. Subroutine names always include parentheses, as in ***printf*( )**. Combinations of keys that must be pressed simultaneously are shown with a + linking the keys. For example, **CTRL+C** means to press the key labeled **CTRL** and the key labeled **C**.

Term	Example
files, path names	**/etc/hosts**
libraries, drivers	**memLib.c**
UNIX shell tools	**xsym**
Tcl procedures	**wtxMemRead**
subroutines	***semTake*( )**
VxWorks boot commands	**p**
code display	`main ();`
keyboard input display	**`wtxregd -V`**
output display	`value = 0`
user-supplied values	*name*
constants	**INCLUDE_NFS**
keywords	**struct**
named key on keyboard	**RETURN**
key combinations	**CTRL+C**
lower-case acronyms	*fd*
GUI titles and commands	Help
GUI menu path	Tools>Target Server>Configure

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 comprises 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.

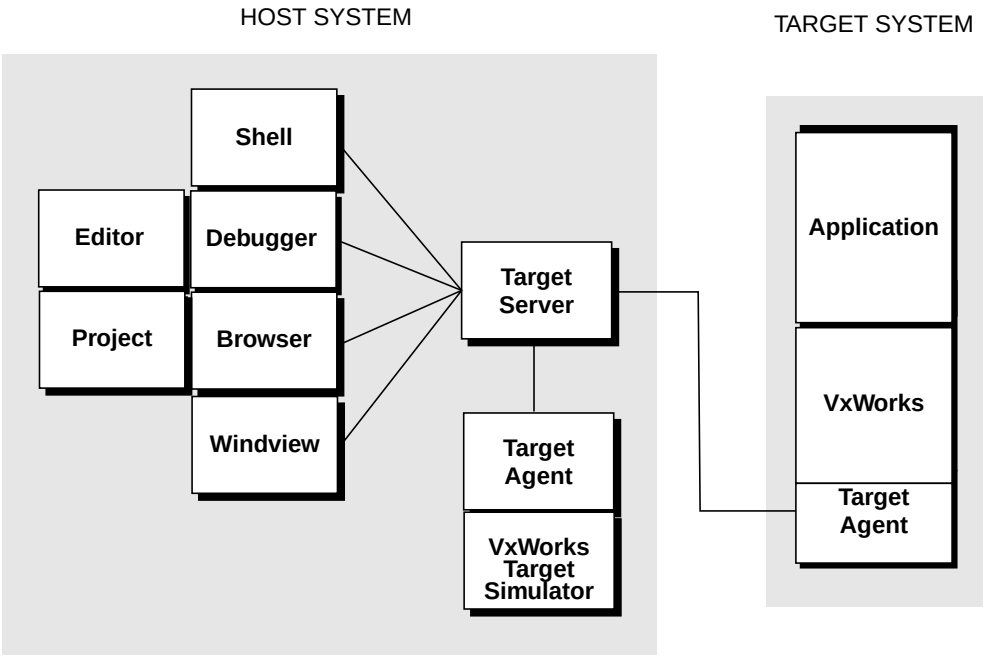
The Tornado interactive development tools include:

- 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).

Figure 1-1 **Tornado Development Environment**



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

A unified macro language called Tcl provides a consistent environment for customization across the entire range of graphical Tornado development tools and the shell.

1.2 Cross-Development with Tornado

Tornado provides the optimal cross-development environment by ensuring the smallest possible difference between the target system during development and after deployment. This is accomplished by segregating the vast majority of the development facilities on the host system, while providing precise yet minimally intrusive access to the target. For all practical purposes, the facilities of the run-time and the development environment are independent of each other, regardless of the scale of the target application.

With Tornado, you can use the cross-development host to manage project files, to edit, compile, link, and store real-time code, to configure the VxWorks operating system, as well as to run and debug real-time code on the target while under host-system control. To understand this environment more clearly, it is useful to outline the typical development process.

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 serial link. If hardware or hardware-specific code is not initially available, the integrated VxWorks target simulator can be used to begin application development.

A typical host development system is fully 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. The target may include no more than a CPU with on-chip RAM and a serial I/O channel (although connections with higher throughput are generally desirable).

Application modules in C or C++ are compiled with the cross-compiler provided as part of Tornado. These application modules can draw on the VxWorks run-time libraries to accelerate application development. 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. Instead, Tornado can load the relocatable object modules directly, using the symbol tables in each object module to resolve external symbol references dynamically. In Tornado, this symbol table resolution is done by the target server (which executes on the host).

With Tornado, object-module sizes are considerably smaller during development, in contrast with other environments. This is because there is no requirement to link the application fully. This advantage provides important leverage in a cross-development environment: the less data to be downloaded, the shorter the

development cycle. Dynamic linking means that it makes sense for even partially completed modules to be downloaded for incremental testing and debugging. The host-resident Tornado shell and debugger can then 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 part of 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 component VxMP. VxMP provides shared semaphores for synchronization of tasks on different CPUs, as well as shared message queues and shared memory management.

For detailed information on VxWorks and on its optional components, see the *VxWorks Programmer's Guide* and the *VxWorks Network Programmer's Guide*.

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 provides more detail on 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 Systems publications on the Internet.
- Help you prepare and transmit support requests to the customer support group at Wind River Systems.

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 leading-edge GNU tools originally developed by the Free Software Foundation (FSF). Users of the GNU tools benefit from the innovative FSF development environment as well as from testing and support by Wind River Systems.

Among other features, 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*.

WindSh Command Shell

WindSh is a host-resident command shell that provides interactive access from the host to all run-time facilities. The shell provides a simple but powerful capability: it can interpret and execute almost all C-language expressions. It also supports C++, including "demangling" to allow developers to refer to symbols in the same form as used by the original C++ source code.

Thus 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.

For even more versatile shell scripting and target control, the Tornado shell includes a complete Tcl interpreter as well as the C interpreter. 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.

As a convenience, there is some overlap between WindSh and CrossWind, the Tornado debugger. (Conversely, the CrossWind debugger provides access to all shell built-in commands.) From the shell, you can perform the following debugging activities:

- Display system and task status.
- Generate a symbolic disassembly of any loaded module.
- Set breakpoints and single-step specific tasks, even in shared code.
- Set breakpoints and single-step the system as a whole, even in interrupt service routines.

As with all Tornado tools, these facilities provide symbolic references wherever possible, using the symbol table managed by the target server.

The shell is described in *5. 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 *7. Debugger*. Also see *Debugging with GDB*.

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 *6. 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 VxWorks target simulator. WindView is available as an optional product for all supported target architectures.

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

VxWorks Target Simulator

The VxWorks 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 a limited version of the target simulator that runs as a single instance per user, without networking support. Optional products such as STREAMS, SNMP, and Wind Foundation Classes are not available for this version.

The full-scale version of the simulator, VxSim, is available as an optional product. It supports multiple-instance use, networking, and all 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. This greatly simplifies debugging of any aspect of an embedded application, whether it be a task, an interrupt service routine, or the kernel itself.

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, because it avoids 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.

This 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. Given the incremental loading capabilities of Tornado, it is quite common to configure the target with the agent linked into the run-time and stored in ROM. When started, the target server automatically initializes the symbol table from the host-resident image of the target run-time system. 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 or from the UNIX command line (or scripts). See *2.7 Connecting a Tornado Target Server*, p.54 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 *D. Tornado Tools Reference*.

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.

See *2.2 Setting up the Tornado Registry*, p.18.

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 Online Documentation

Tornado online documentation includes online versions of all manuals, as well as a context-sensitive reference entry for each tool, and a search facility.

The online manuals include all standard Tornado and VxWorks manuals, as well as the GNU manuals, in HTML format. To view the online manuals, click Help>Manuals Contents in the Tornado launcher, browser, and debugger. Reference

manual information can also be accessed directly from the shell and project facility workspace.

1.7 Extending Tornado

Tornado can be extended and customized through standard APIs, as well as with optional products available from Wind River Systems and third-party vendors.

Published Application Program Interfaces

A central feature of Tornado is the rich set of Application Program Interfaces (APIs) that allow access to every level of the technology. Much of Tornado is implemented in Tcl; source code is included automatically, because Tcl is an interpreted language. By virtue of their Tcl implementation, Tornado facilities for target inspection and manipulation are available for customization, extension, or simply for their educational value. Tornado goes further yet: every aspect of the user interface is also under user control. From forms to buttons and menu items, the Tornado environment can be customized. (For a brief summary of Tcl, see *B. Tcl*.)

At the target-server layer, there are C and Tcl language bindings to the underlying protocol. APIs are available for new back ends supporting additional host-to-target connection methods. These bindings use libraries; this makes it unnecessary to build, manage, or maintain alternative configurations of the target server. The target agent also has stable, published run-time and driver interfaces.

Each of these APIs is discussed in detail in the *Tornado API Guide* and the online *Tornado API Reference*.

Optional Products

Contact your sales representative for information about optional products from Wind River Systems.

Third-Party Products

You can further extend the capabilities of your Tornado development system with products designed for Tornado by other companies. Ask your Wind River sales representative for a list of available third-party products.

1.8 Customer Services

A full range of support services is available from Wind River Systems to ensure that you have the opportunity to make optimal use of the extensive features of Tornado.

This section summarizes the major services available. For more detailed information, see *9. Customer Service* and the *Customer Support User's Guide*.

Customer Support

Direct contact with a staff of software engineers experienced in Tornado is available through the Wind River Systems Customer Support program. For information on how to contact WRS Customer Support, see *9. Customer Service*.

Training

In the United States, Wind River Systems holds regularly scheduled classes on Tornado. Customers can also arrange to have Tornado classes held at their facility. The easiest way to learn about WRS training services, schedules, and prices is through the World Wide Web. Connect to the Wind River Systems home page (URL: **http://www.wrs.com/**) and select the link to Training.

You can contact the Training Department at:

Phone:	510/749-2148
	800/545–WIND
FAX:	510/749–2378
E-mail:	training@wrs.com

Outside of the United States, call your nearest Wind River Systems office or your local distributor for training information. See the back cover of this manual for a list of Wind River Systems offices.

2 Setup and Startup

2.1 Introduction

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.45.

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 to your target.

4. 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 relative path names starting at the directory **wind**. Use the path name chosen on your system for Tornado installation.

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]

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.

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:

```
% /usr/wind/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
    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 *D. Tornado Tools 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.20 discusses all Tornado environment variables.

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

WIND_HOST_TYPE
name of host type; see Table 2-1

WIND_REGISTRY
registry host; see *2.2 Setting up the Tornado Registry*, p.18

PATH
shell search path; add **$WIND_BASE/***host-os***/bin** directory

LD_LIBRARY_PATH
dynamic library search path for Solaris;
add Tornado **$WIND_BASE/host/sun4-solaris2/lib** directory

SHLIB_PATH
dynamic library search path for HP-UX;
add **$WIND_BASE/host/parisc-hpux10/lib** directory

The Tornado installation includes a **wind/host/***host-os***/bin** directory of tools that run on your development host. The values for *host-os* match those listed in Table 2-1 for **WIND_HOST_TYPE**. Add the **wind/host/***host-os***/bin** directory for your host to the UNIX shell search path (**PATH**).

Table 2-1 **Values for WIND_HOST_TYPE and *host*/bin**

Host	Host OS	WIND_HOST_TYPE
HP9000 series 700	HP-UX 10.2x	**parisc-hpux10**
Sun-4	Solaris 2.x	**sun4-solaris2**

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 HP9000 host in a network whose Tornado registry is on host **venus**:

```
WIND_BASE=/usr/wind; export WIND_BASE
WIND_HOST_TYPE=parisc-hpux10; export WIND_HOST_TYPE
WIND_REGISTRY=venus; export WIND_REGISTRY
PATH=$WIND_BASE/host/parisc-hpux10/bin:$PATH; export PATH
SHLIB_PATH=$WIND_BASE/host/parisc-hpux10/bin:$SHLIB_PATH; export SHLIB_PATH
```

2.3.2 Environment Variable

Solaris 2 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 **$WIND_BASE/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
```

HP-UX Hosts

If your development host runs HP-UX, you must also modify the value of **SHLIB_PATH** to include **$WIND_BASE/host/parisc-hpux10/lib**.

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 Target Setup

This section covers bringing up VxWorks on a target with a relatively simple configuration. 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 Example Target Configurations

VxWorks is a flexible system that has been ported to many different hardware platforms. Two common examples are illustrated in this section.

Figure 2-1 **A Minimal Tornado Configuration**

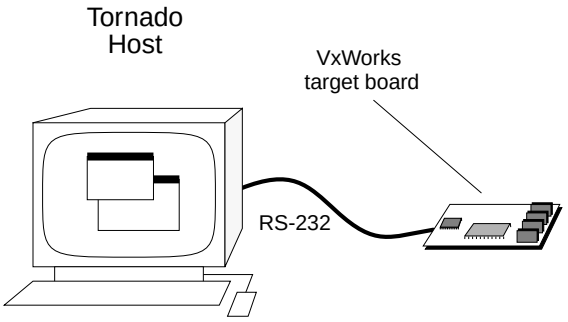


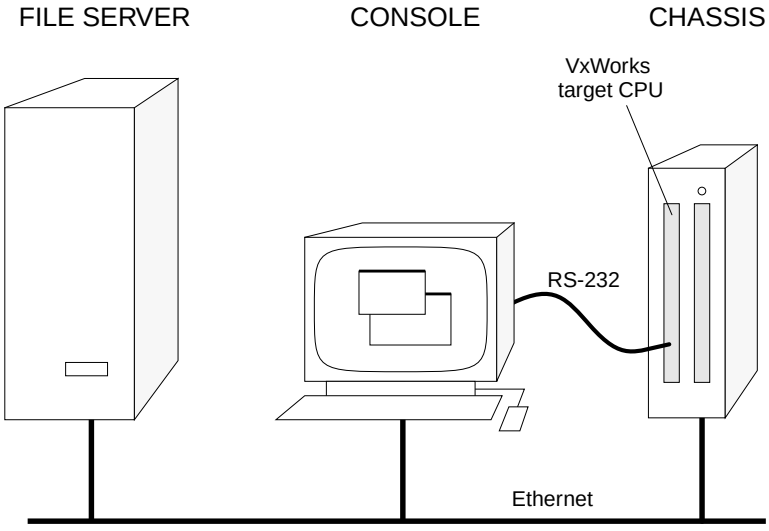
Figure 2-1 illustrates a minimal cross-development configuration: the target is a bare board, connected to the host development system by a single serial line. When you connect the host and target exclusively over serial lines, you must:

- 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.

See *2.5.3 Serial-Line Connections*, p.34.

Another target system configuration, representing a more resource-rich development environment, is shown in Figure 2-2. The default boot program and VxWorks image support booting the target over a network.

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 (the file **wind/docs/BSP_Reference.html**).

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 to get started with 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 2-2 **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 (the file **wind/docs/BSP_Reference.html**).

Boot ROMs and Other Boot Media

Tornado includes one of the following boot media as part of each VxWorks BSP package:

- **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.
- **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 (the file **wind/docs/BSP_Reference.html**).

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

2. NetROM is a trademark of Applied Microsystems Corporation.

about how to use NetROM on your target, refer to *2.5.4 The NetROM ROM-Emulator Connection*, p.37.

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 *4.7 Configuring and Building a VxWorks Boot Program*, p.144 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 (**wind/docs/BSP_Reference.html**).

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, first install the CPU board in the first slot of the backplane. See Figure 2-3.

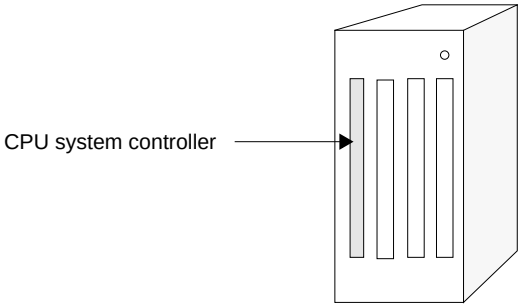
On a VMEbus backplane, there are several issues to consider:

- **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-

Figure 2-3 **Assembling VME Targets**



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), at least for the initial configuration of the boot parameters and getting started with VxWorks. Subsequently, VxWorks can be configured 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

Tornado host tools such as the shell and 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 2-4 for a detailed diagram of host-target communications.

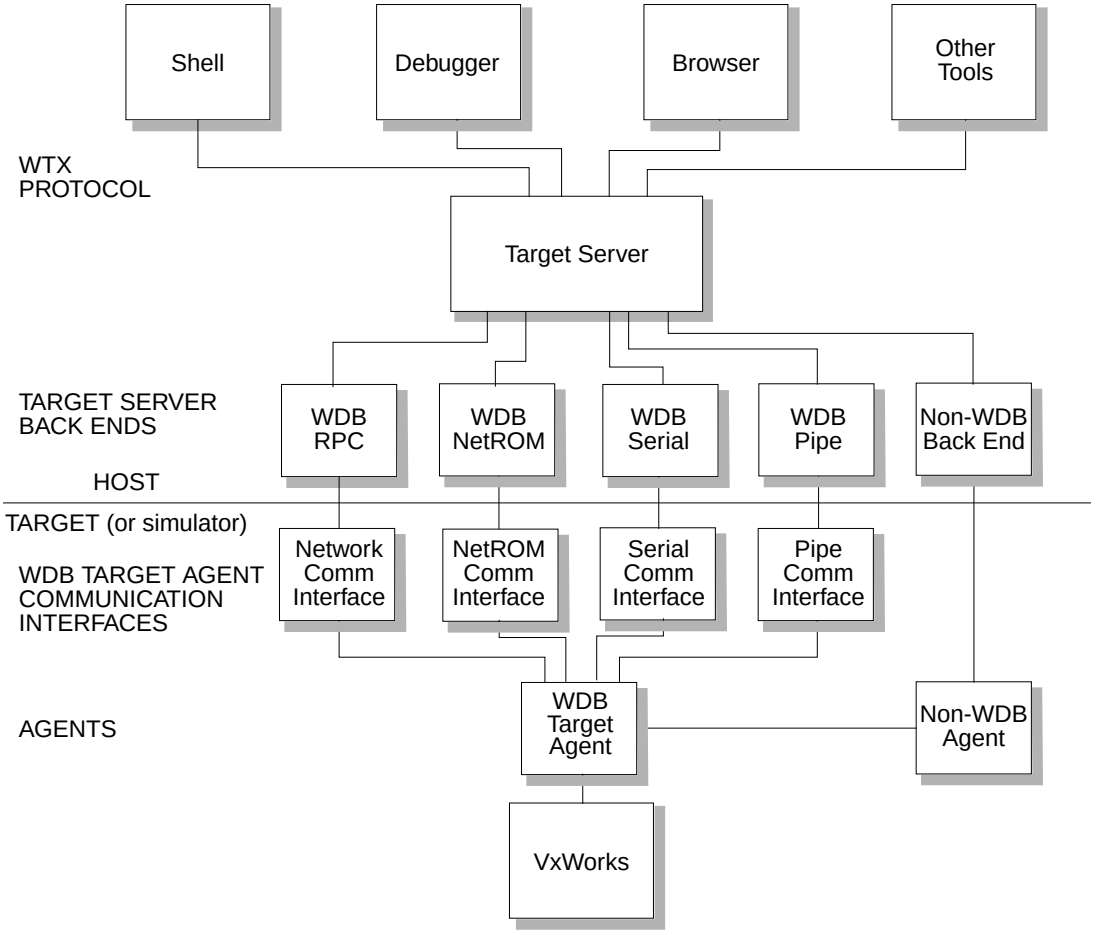
The default configurations for both the VxWorks target agent and Tornado target servers are for a network connection. If you are using a network connection, you can proceed with booting your target (*2.6 Booting VxWorks*, p.45).

WARNING: If you are using a connection other than network (the default), you must rebuild VxWorks with the appropriate target agent communication interface, and configure a target server with the corresponding back end, before you can use Tornado tools with your target. See *4.6 Configuring the Target-Host Communication Interface*, p.137 and *2.5 Host-Target Communication Configuration*, p.31.

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*.

- 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).

Figure 2-4 **Tornado Host-Target Communication**



- 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 *7.5 System-Mode Debugging*, p.261) or the shell (*5.2.6 Using the Shell for System Mode Debugging*, p.170). 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.

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. VxWorks network interface drivers that do not support polled operations (older versions) 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.53. 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 (or the file **wind/docs/BSP_Reference.html**).

Connecting the target server to the target requires a little work on both the host and target. The next few subsections describe the details for the standard target-server back end connections.

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 *5.2 Configuring and Starting a Target Server*, p.128).

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 *4.6 Configuring the Target-Host Communication Interface*, p.137 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.138 for information about configuring the VxWorks target agent for an END connection.

2.5.3 Serial-Line Connections

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.141).

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 *4.7 Configuring and Building a VxWorks Boot Program*, p.144). 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.53.

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 that the serial connection to the target is good. 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.

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

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 **wind/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.

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:

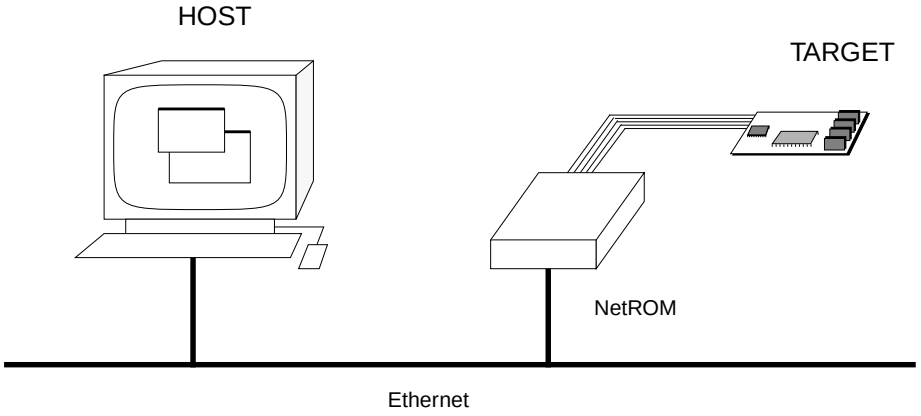
```
% tgtsvr -V targetname -B wdbserial -bps 38400 &
```

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 Systems 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.53.

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.138.

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.

Step 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 2*. 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.

Step 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**).

Step 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.

Step 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.

Step 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.

Step 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.140.

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 WRS technical support for more troubleshooting assistance.

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 **wind/docs/BSP_Reference.html**).

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

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

4. Commonly available terminal emulators are **tip**, **cu**, and **kermit**; consult your host reference documentation.
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 *4.7 Configuring and Building a VxWorks Boot Program*, p.144). In the course of your developing an application, you will also build bootable applications (see *4.4 Creating a Bootable Application*, p.127).

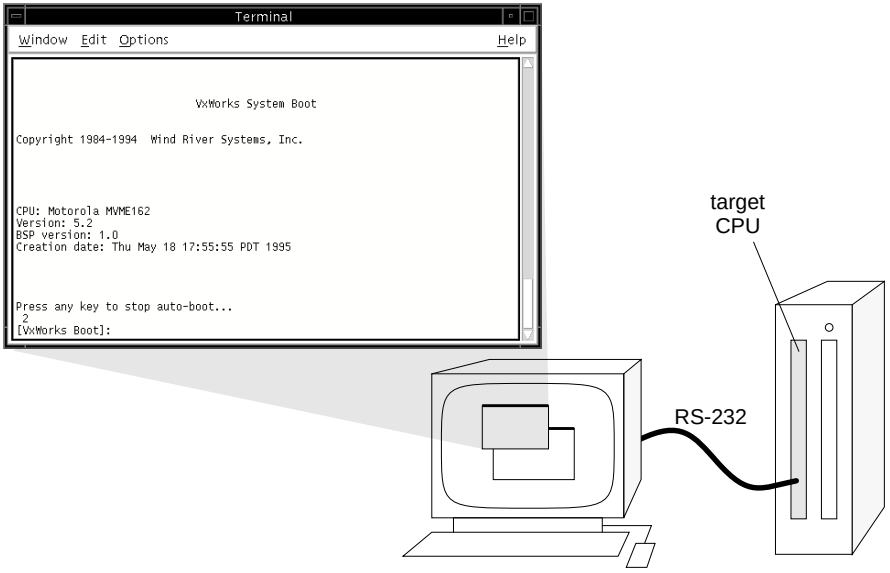
program first displays a banner page, and then starts a seven-second countdown, visible on the screen as shown in Figure 2-6. Unless you press any key on the keyboard within that seven-second period, the boot loader will attempt to proceed with a default configuration, and will not be able to boot the target with VxWorks.

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]:
```

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



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

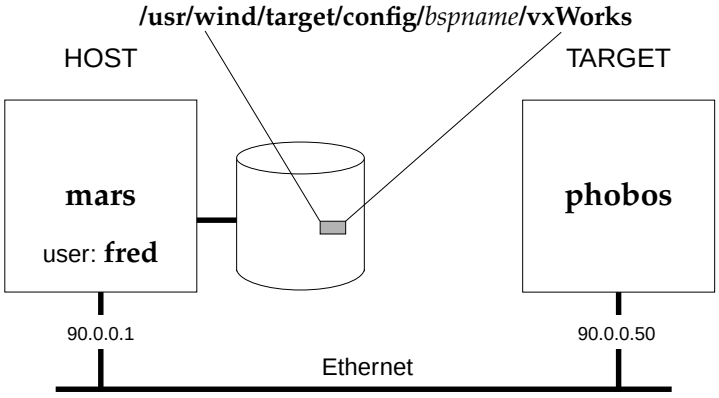
```
[VxWorks Boot]: p
```

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               : /usr/wind/target/config/bspname/vxWorks
inet on ethernet (e)    : 90.0.0.50
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.27.

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 the help command (**h** or **?**) followed by **RETURN**:

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

Table 2-3 lists and describes each of the VxWorks boot commands and their arguments.

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.46. 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.52.

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.46.)

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]

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.

Table 2-3 **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.
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*.

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.

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:

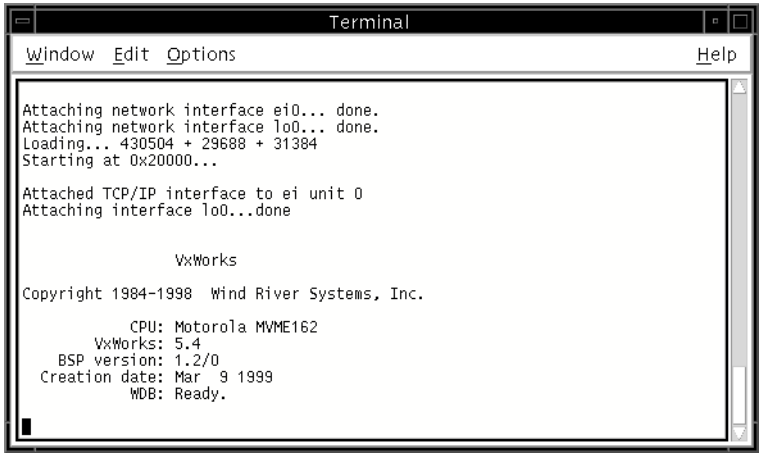
- The boot program first initializes its network interfaces ❶.
- While VxWorks is booting, you can see the size of each VxWorks section (text, data, and bss) as it is loaded ❷.
- After the system is completely loaded, the boot program displays the entry address and transfers control to the loaded VxWorks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

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.54.

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.58.

❶ 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.

Figure 2-8 **VxWorks Booting Display**



❷ 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).

❸ 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.

❹ 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 *4.7 Configuring and Building a VxWorks Boot Program*, p.144 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 *4.7 Configuring and Building a VxWorks Boot Program*, p.144.

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.48). 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

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 *5.7 Tcl: Shell Interpretation*, p.198.)
- 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.27) 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 *D. Tornado Tools Reference*). The easiest way to select and manage these options is with the Tornado launcher.

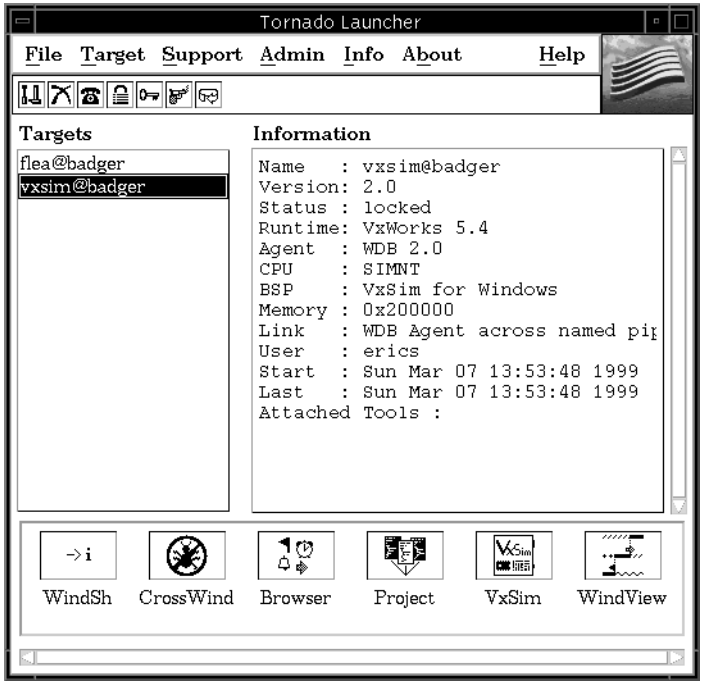
2.8 Launching Tornado

Are you ready to try Tornado? The launcher is a good place to start; it provides access to all other Tornado facilities. 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, and 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).

But it's safe to play around. You may want to take a break from reading, and experiment with Tornado now!

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.xxiii.

- **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.58 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** (*D. Tornado Tools 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 Systems technical support organization. Your comments and suggestions are welcome as well.

The Tornado launcher has facilities to help you submit trouble reports; you can use them even if you were unable to hook up a target. See *9. Customer Service*.

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 Systems, 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

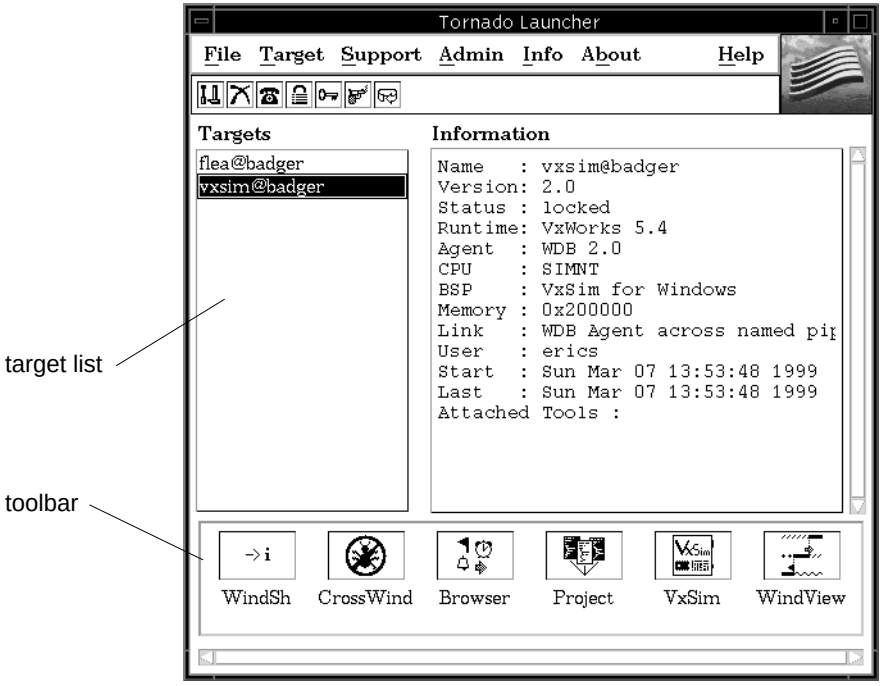
Most of the main launcher window (Figure 3-1) reflects the two main kinds of objects it links together—tools and target servers:

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 **$WIND_BASE/.wind**. If that directory is not writable, the database is created in **$HOME/.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 *5. Shell*.

CrossWind
: The Tornado graphical debugger, a powerful source-level debugger that provides both graphical and command-driven access to target programs. *7. Debugger* provides full documentation.

Browser
: A viewer to explore and monitor target system objects, described in *6. 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 ❶ is connected to a target on the local Ethernet subnet.

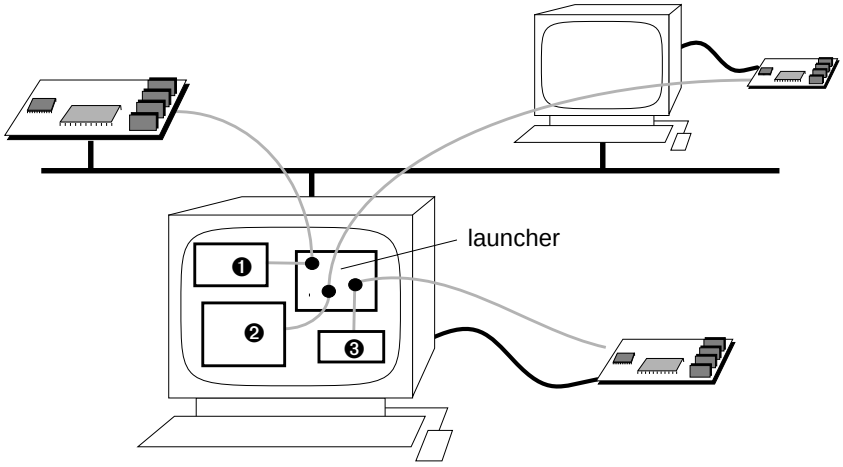
– Tool ❷ is connected over the local Ethernet to a target that is physically attached to a remote host.

– Tool ❸ 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 Target Setup*, 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 STREAMS, SNMP, and Wind Foundation Classes 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 *D. Tornado Tools Reference* or online (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.

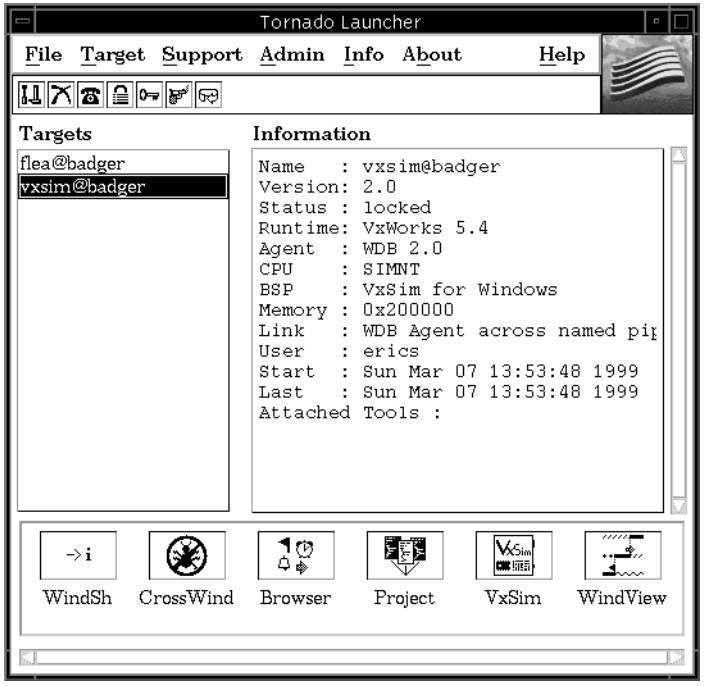
If you make a mistake, or if you wish to select another target, simply click on another target-server name. Any tools that you have already launched remain connected to the previous target (the plugboard analogy does not extend that far).

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

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



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.

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.

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.

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

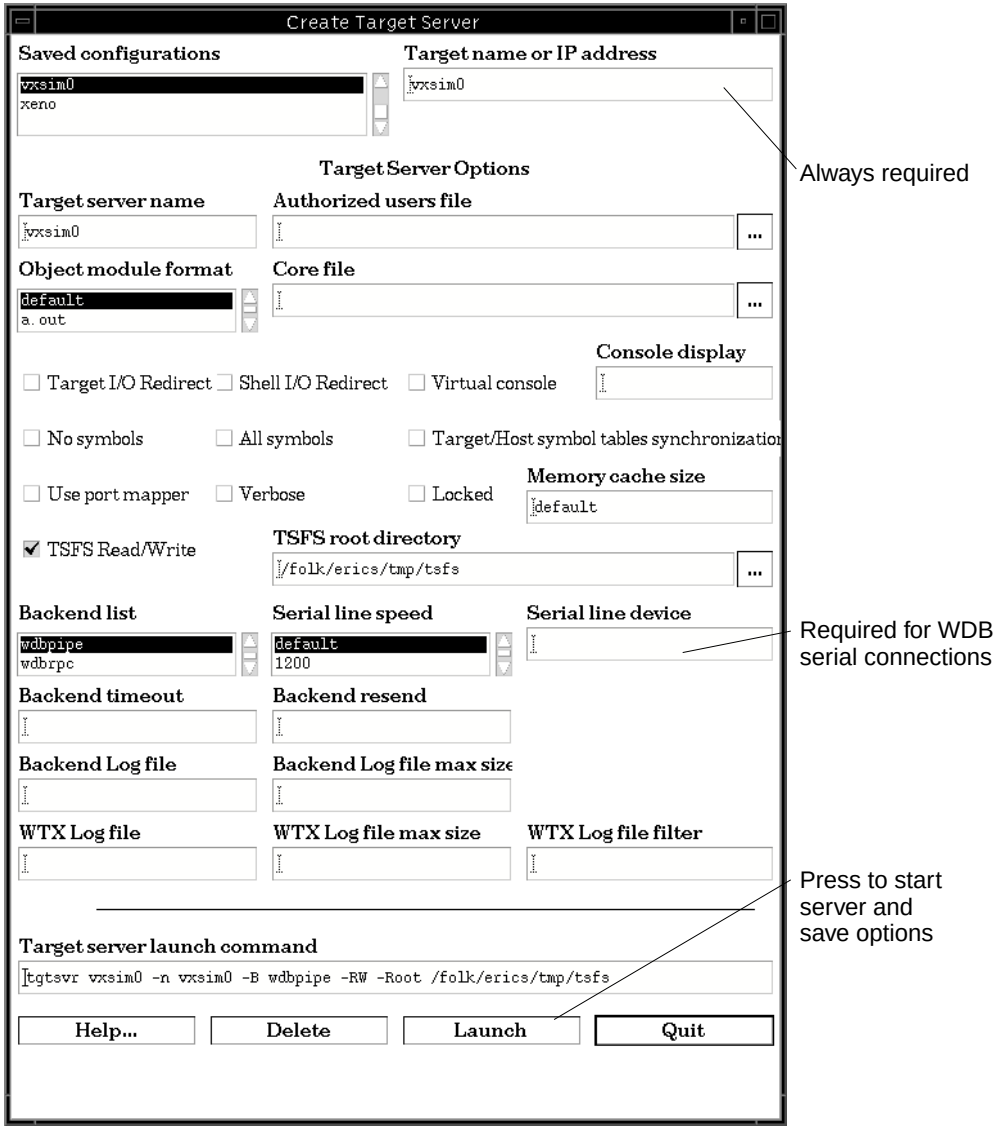
To use a new target, you must first ensure the host and target are connected properly. The details are unique to each target, but *2.4.2 Networking the Host and Target*, p.26 discusses some of the issues that are frequently involved. Your BSP also contains a target-information reference that describes 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**.) 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 (either online, or in *D. Tornado Tools Reference*).

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



- 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 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.141.)

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.

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

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

from target programs assume the target server is running with this option set. See *Virtual I/O*, p.12 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 *4.3.3 Configuring VxWorks Components*, p.117 and the reference entry for **symSyncLib**.

Use portmapper
: Turn on this toggle to register a target server with the RPC portmapper. While the portmapper is not needed for Tornado 2.0, this option is included for development environments in which both Tornado 2.0 and Tornado 1.0.1 are in use. When both releases are in use, the portmapper must be used on:

  - Any host running a Tornado 2.0 registry that will be accessed by any host running Tornado 1.0.1.
  - Any host running a Tornado 2.0 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 *Tornado User's Guide: 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.

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.141 for more information about the memory pool managed by the server on the target.

TSFS Read/Write

Click the Read/Write box to change the option from the default read only. Make this change only when you plan to run WindView; at other times the TSFS option for your target server should be read only.

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.53).

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
```

If you use the TSFS for booting a target, it is recommended that you use the base Tornado installation directory (**$WIND_BASE**) 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).

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

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 *4.6 Configuring the Target-Host Communication Interface*, p.137.

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.
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.141).

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]

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

9. Strictly speaking, there is another layer of authorization defining who is meant by "any user". The file **$WIND_BASE/.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.

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 WRS
- 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 Systems' support, training, and sales services. See *9. Customer Service* 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 Systems maintains a small archive of auxiliary software and useful information available over the Internet by FTP. Click on this command to connect to the WRS 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 **${WIND_BASE}/.wind/userlock**. This file controls overall access to Tornado host tools at your site. This file employs the same simple conventions described in *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 *B. 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 (all file names below are relative to **WIND_BASE**):

host/resource/tcl/Launch.tcl
: The main launcher implementation file.

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

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

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

host/resource/tcl/app-config/all/${WIND_HOST_TYPE}.tcl
Host-specific overrides for global settings.

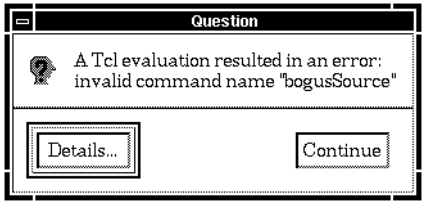
3.7.1 Tcl: Launcher Initialization File

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.

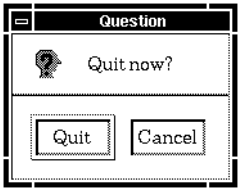
Example 3-1 **Tcl Reinitialization**

```
# "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 **${WIND_BASE}/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. 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:

- Organizing the files that make up a project.
- Grouping related projects into a workspace.
- Customizing and scaling VxWorks.
- Adding application initialization routines to 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 2.0 (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, although the manual method may still be used (see *VxWorks Programmer's Guide: Configuration and Build*).

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 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.118). 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.

Terminology

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

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.

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, a custom VxWorks image, 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 that are not linked to application code 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.

Toolchain

A toolchain is a set of cross-development tools used to build applications for a specific target processor. The toolchains provided with Tornado are based on the GNU preprocessor, compiler, assembler, and linker (see the *GNU Toolkit User's Guide*). However, 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. BSPs also include project files that facilitate creation of projects for bootable applications and custom VxWorks images.[2]

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.

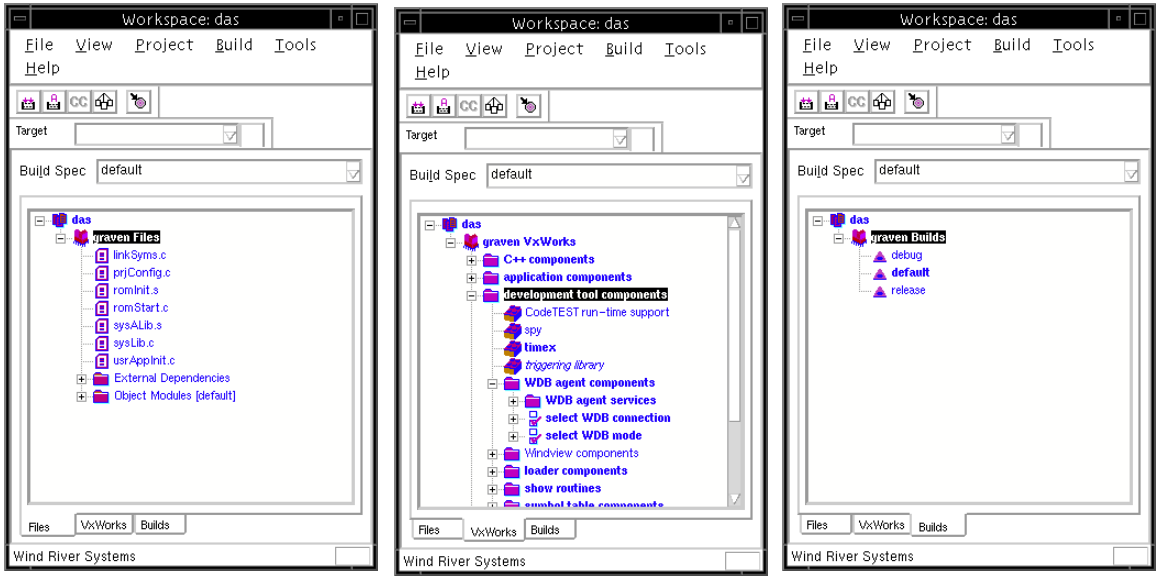
As its name implies, the Workspace window provides the framework for the project facility. The window displays information about projects files, VxWorks components (if any), and build options in three tabbed views: Files, VxWorks, and Builds (Figure 4-1).

The workspace allows you to:

- 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.
- Scale and customize VxWorks by adding and deleting components, as well as display component dependencies and view object sizes.
- 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.

2. Beginning with the 2.0 release of Tornado.

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



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).

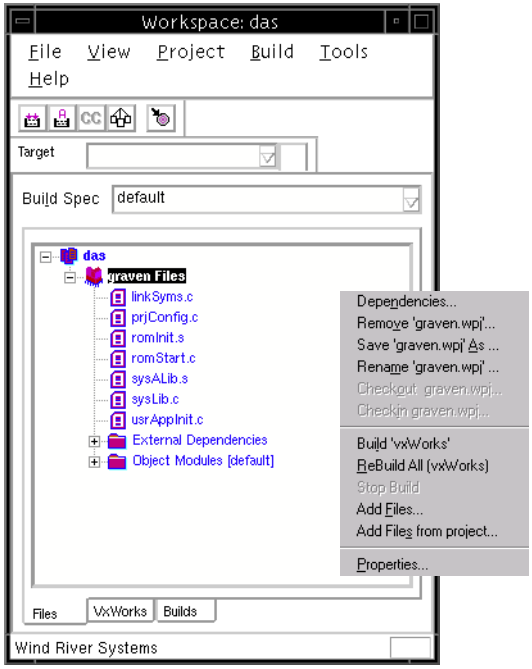
Many of the context menu options are also available under the File, Project, and Build menus.

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 *8. Customization*.

4.2 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

Figure 4-2 **Workspace Window Context Menu**



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.

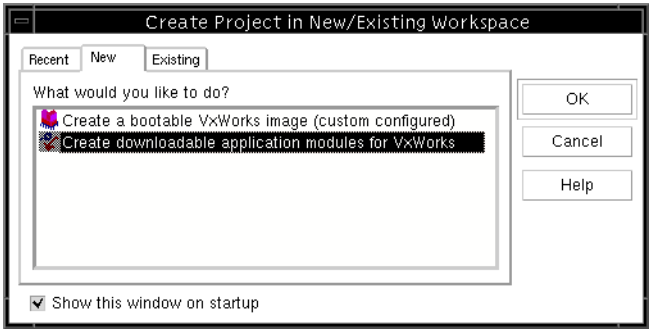
You can then download the object module(s) to the target system and run the application.

4.2.1 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).

Figure 4-3 **Create Downloadable Application**



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

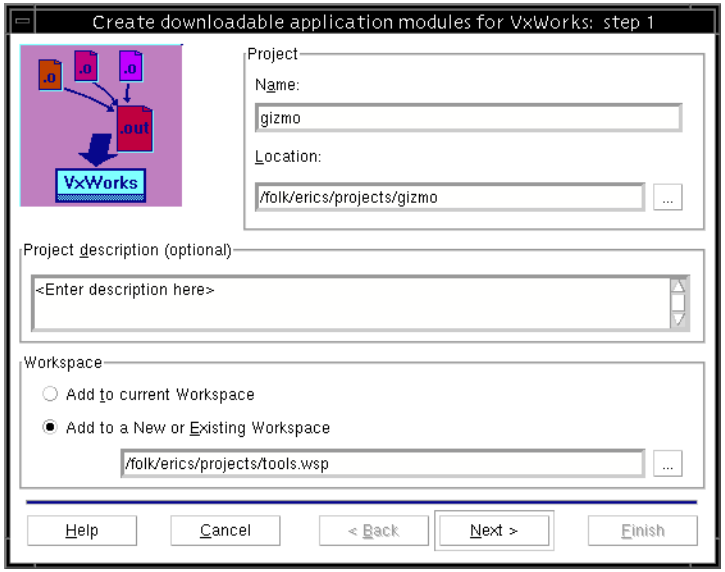
First, enter the full directory path and name of the directory you want to use for the project (only one project is allowed in a directory), and enter the project name. It is usually most convenient to use the same name for the directory and project, but it is not required.

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.

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

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



Then you 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.

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).

4. The default toolchain names for target simulators take the form SIM*HOST_OS*gnu (for example, SIMSPARCSOLARISgnu and SIMHPPAgnu).

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

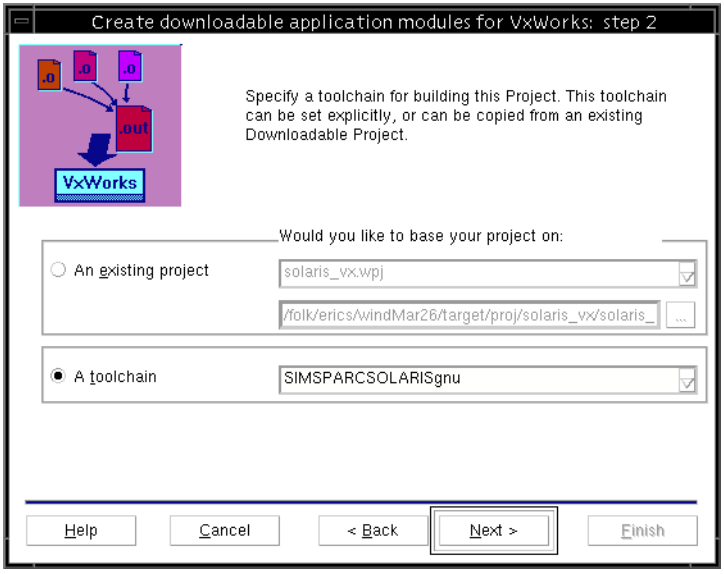


Figure 4-6 **Application Wizard: Step Three for Downloadable Application**

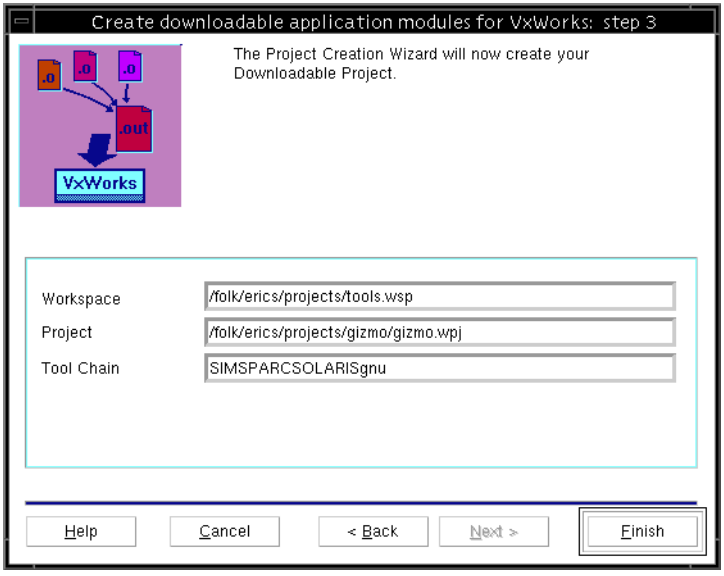
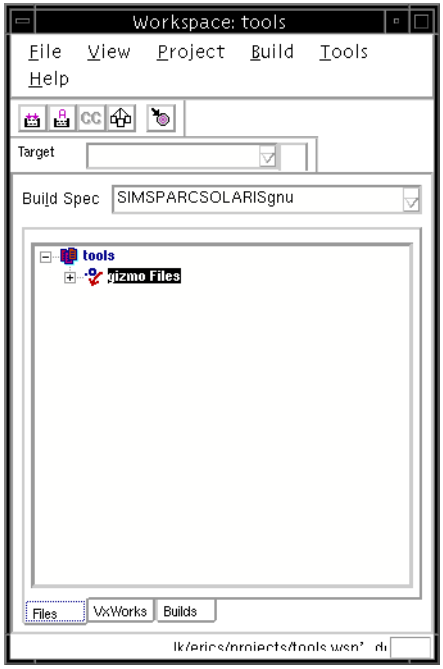


Figure 4-7 **Initial Workspace Window for a Downloadable Application**



NOTE: Context 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).

4.2.2 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.

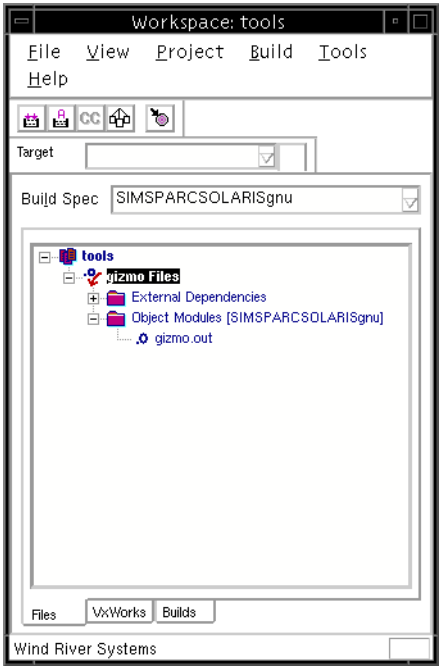
Both of these files contain information that changes as you modify your project, and add projects to, or delete projects from, the workspace.

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.

4.2.3 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).

Figure 4-8 **Workspace Files View**



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.

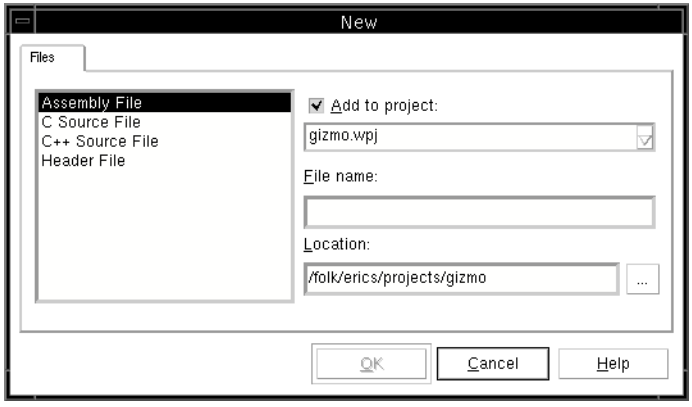
Initially, there are only the default folders for Object Modules and External Dependencies, and the *projectName***.out** file. The file *projectName***.out** is created as a single, partially-linked module when the project is built. It comprises all of the individual object modules in a project for 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 context menu, and then using the associated dialog box to locate and select the file(s).

To link object files with your project, use the Linker page of the build specification property sheet (see *Linker Options*, p.134). 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.131).

Remove files from the project by right-clicking on the file name and selecting Remove from the context 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 context 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).

See *Calculating Makefile Dependencies*, p.105, for information about how and when to calculate makefile dependencies. See *Compiler Options*, p.132 for information about overriding default compiler options for individual files.

Opening, Saving, and Closing Files

The File menu and context menu provides options for opening, saving, and closing files. You can also use standard Windows shortcuts (such as double clicking on a file name to open the file in the editor).

Figure 4-10 **Source File Property Sheet**

4.2.4 Building a Downloadable Application

The project facility uses the GNU **make** utility to automate compiling and linking an application.[5] It creates a makefile automatically 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.

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 *Linker Options*, p.134.

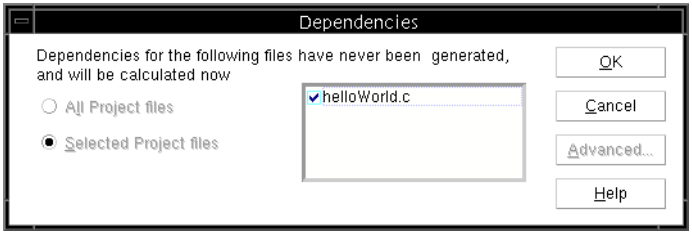
NOTE: The project facility allows you to create specifications for different types of builds, to modify the options for any one build, and to easily select the build specification you want to use at any given time. See *4.5 Working With Build Specifications*, p.128.

5. See the *GNU Make User's Guide* for more information about **make**.

Calculating Makefile Dependencies

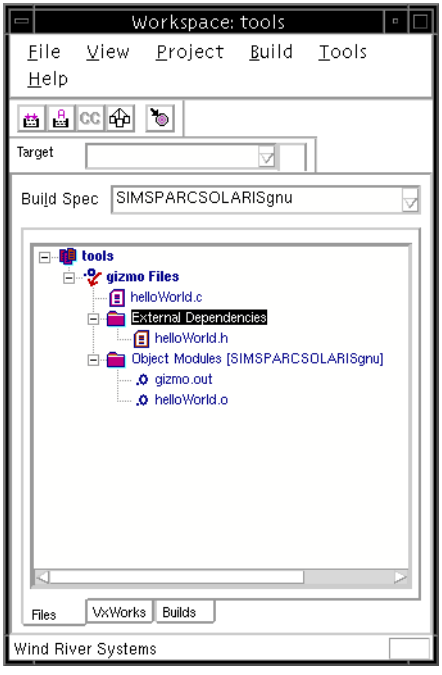
To calculate makefile dependencies select Dependencies from the workspace context menu. The Dependencies dialog box appears (Figure 4-11). Click OK.

Figure 4-11 **Dependencies Dialog Box**



After dependencies have been calculated, the files are listed in the External Dependencies folder (Figure 4-12).

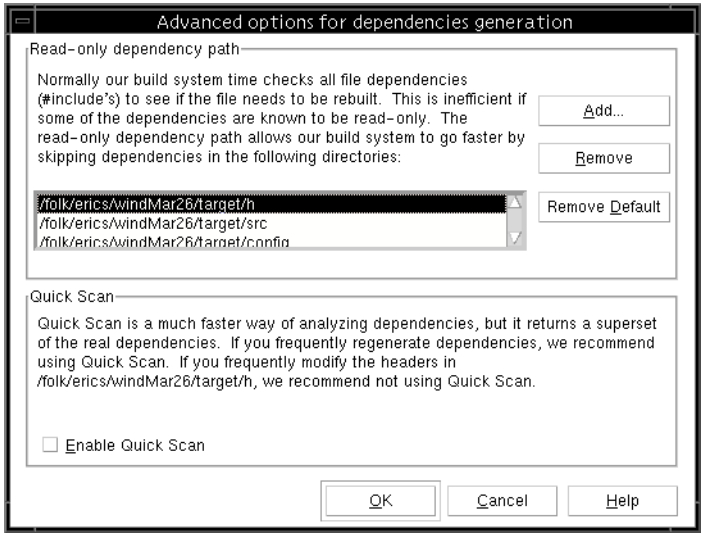
Figure 4-12 **External Dependencies**



If you do not calculate dependencies before you start a build, Tornado prompts you to do so for any project files for which dependencies were not previously calculated.

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**



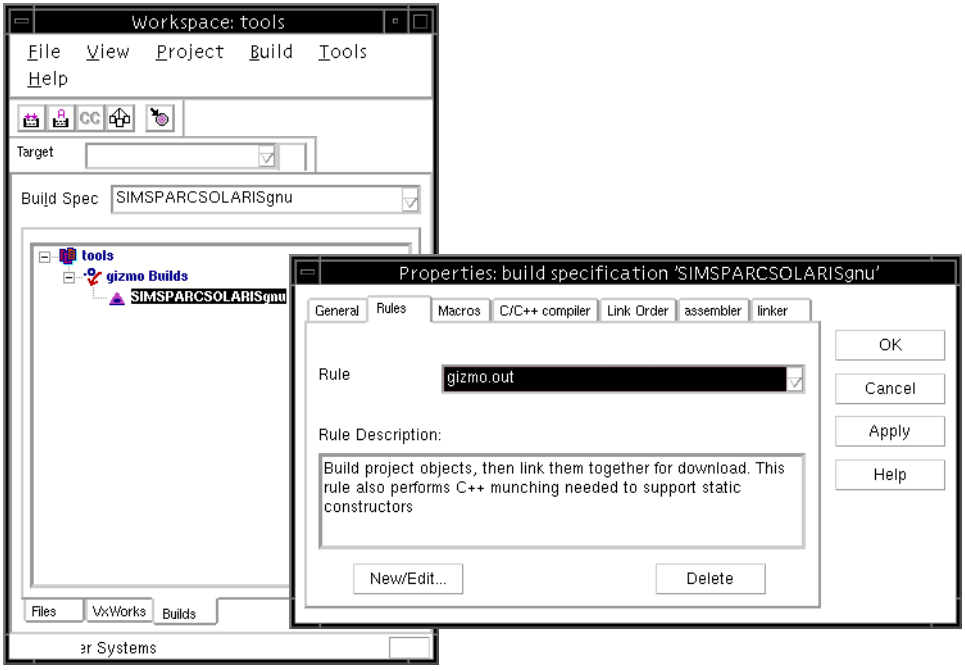
Build Specifications

Each build 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 information about it, double-click on the build name in the Builds view of the workspace to display the property sheet for the build. 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.

Figure 4-14 **Build Property Sheet**



projectName.out
: A single, partially-linked and munched object that comprises all of the individual object modules in a project.

You can use the project facility to change the options for a given build, 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 *4.5 Working With Build Specifications*, p.128.

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.

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 context 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**

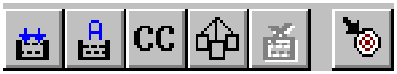
WARNING: The default compiler options include **-g** for debugging information. Using **-g** with the optimization option **-O** set to anything but zero may produce unexpected results. See *4.5 Working With Build Specifications*, p.128 for information about modifying builds and creating new build configurations.

To force a rebuild of all project objects, select Rebuild All from the context menu (which performs a **make clean** before the build).

Build Toolbar

The Build toolbar provides quick access to build commands. Display of the toolbar is controlled with the View>Build Toolbar menu option (Figure 4-16).

Figure 4-16 **Build Toolbar**



The Build toolbar commands (Table 4-1) are also available from the main menus and the Workspace context menu.

Table 4-1 **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).

4.2.5 Downloading and Running 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. See *2. Setup and Startup* and *3. Launcher* for more information.

You can download an entire project from the project workspace by selecting Download '*projectName*.out' from the context 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 and the Download '*filename*.o' option from the context 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 should be downloaded as *projectName*.out because this file is produced from application files and munched for proper static constructor initialization.

To unload a project from the target, use the Unload '*projectName*.out' option on the context menu.

4.2.6 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 context 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.

4.3 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 *4.4 Creating a Bootable Application*, p.127).

4.3.1 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.

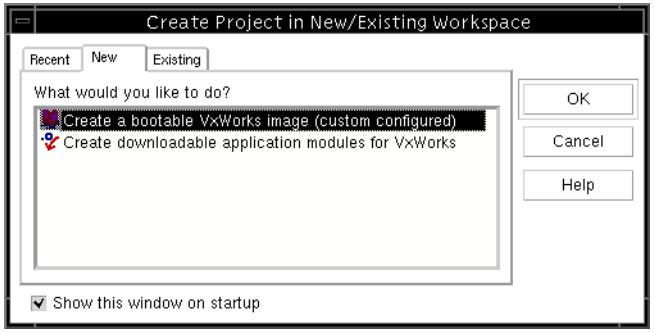
Open a project workspace by clicking the Project button in the Tornado Launch window. If the Create Project or Open Workspace window is open (the default when you first open the Tornado Project window[6]), click the New tab. Otherwise, click File>New Project. Then choose the selection for a bootable 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.

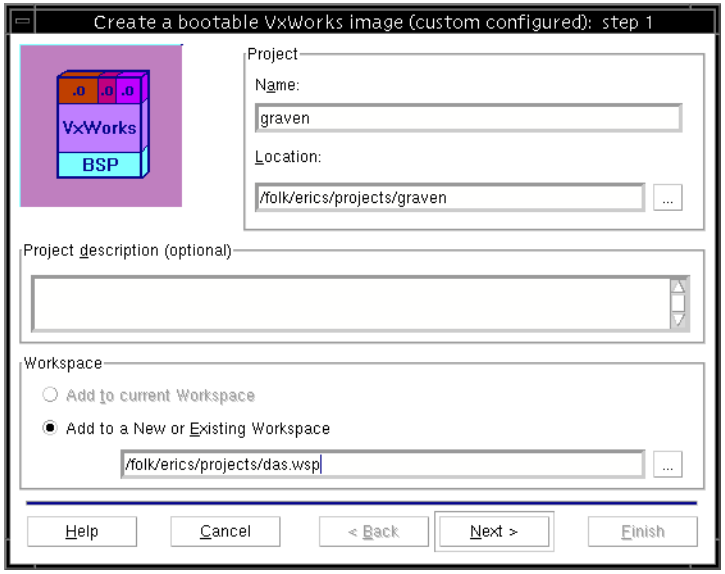
Figure 4-17 **Create Bootable Application**



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 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 project files for this purpose. Options for BSP projects are available in the drop-down list for existing projects. For example, the mv162 BSP project file is

wind/target/proj/mv162_vx.wpj.

Projects can only be created from BSPs installed in the Tornado 2.x directory tree. If you want to use a BSP from an earlier version of Tornado (or any third-party BSP that is compliant with Wind River Systems coding conventions for BSPs), you must install it in the current tree before creating your project.

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 486 PC target, select An existing project and then select pc486_vx.wpj from the drop-down list (Figure 4-19). Click Next.

The wizard confirms your selections (Figure 4-20). Click Finish.

The Workspace window appears.

4.3.2 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).

Figure 4-19 **Application Wizard: Step Two for Bootable Application**

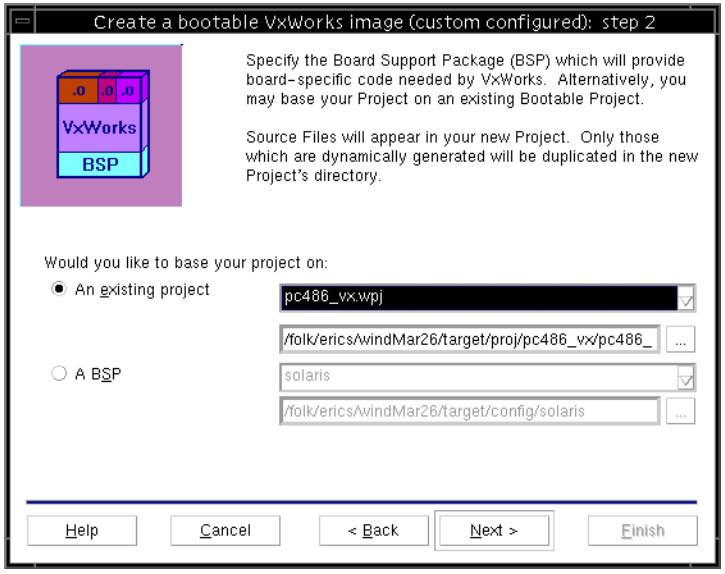
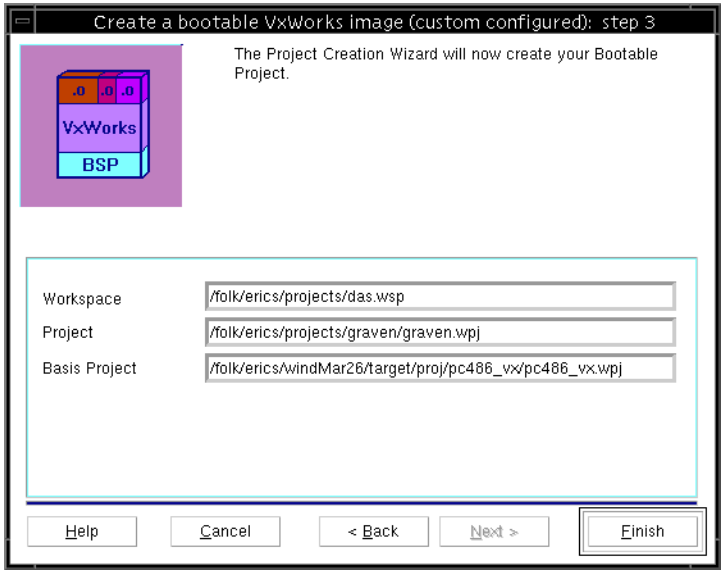
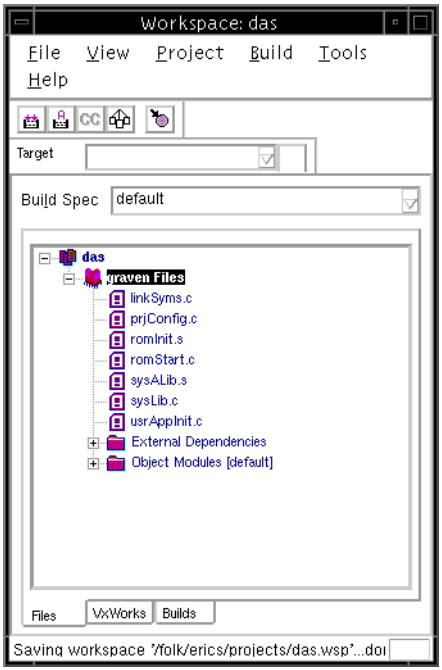


Figure 4-20 **Application Wizard: Step Three for Bootable Application**



NOTE: Context 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 *4.4 Creating a Bootable Application*, p.127).

The VxWorks project files serve the following purposes:

linkSyms.c

A dynamically generated configuration file that includes code from the VxWorks archive by creating references to the appropriate symbols.

prjConfig.c
A dynamically generated configuration file 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 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
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
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.

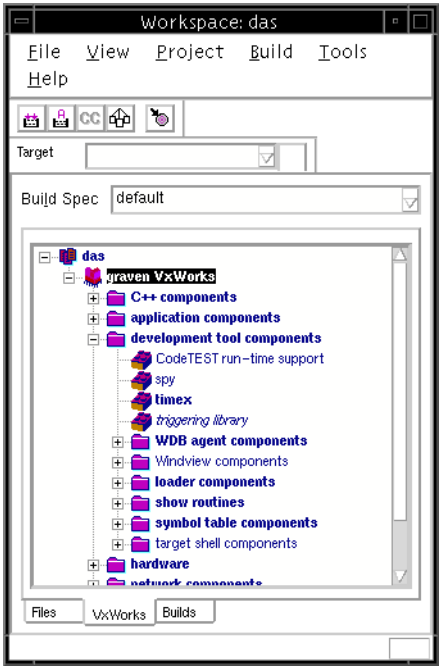
When you build the project, a makefile is dynamically generated in the main project directory, and a subdirectory is created containing the objects produced by the build. The subdirectory is named after the selected build specification. If other build specifications are created and used for other builds, parallel directories are created for their objects.

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.

4.3.3 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 on *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**

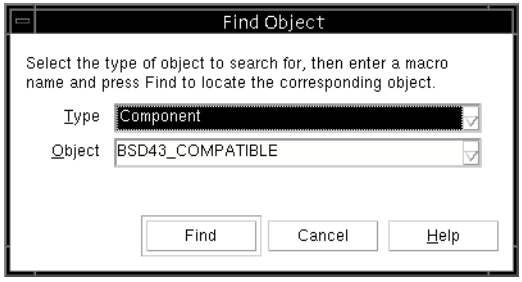


NOTE: See the *VxWorks Programmer's Guide* for detailed information about the use of VxWorks facilities, target-resident tools, and optional components.

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. The dialog box can be accessed with the context 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

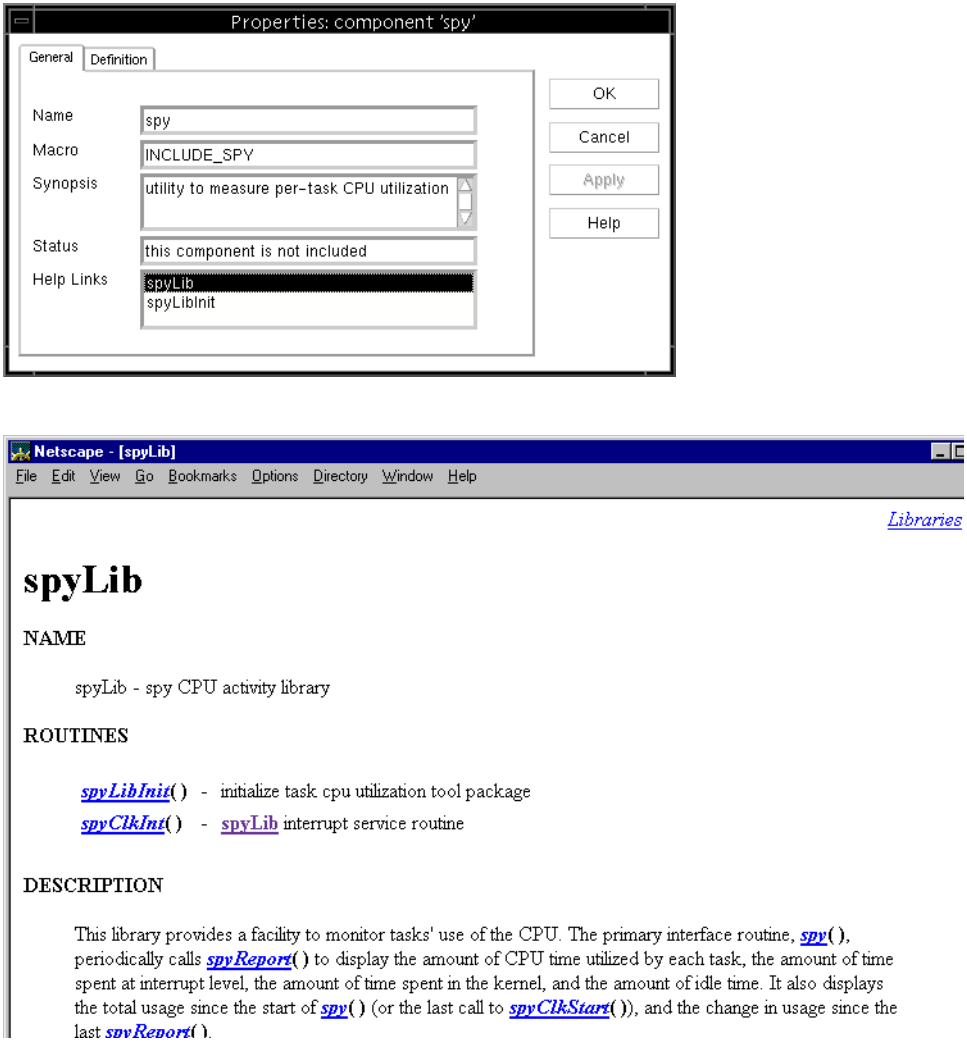
The component tree in the VxWorks view provides descriptive names for components. You can display a component description property sheet, which includes the name of the pre-processor macro for the component, by double-clicking on the component name (Figure 4-24).

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).

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 context menu

Figure 4-24 **VxWorks Component Properties and HTML Reference**

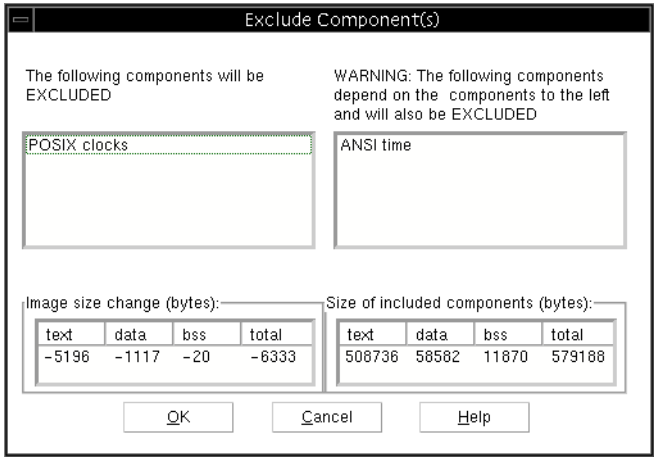


provides Include and Exclude options for components you select in the VxWorks view. You can also use the **DELETE** key to exclude options.

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 dependent components 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 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, and 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).

Figure 4-26 **Include VxWorks Component**

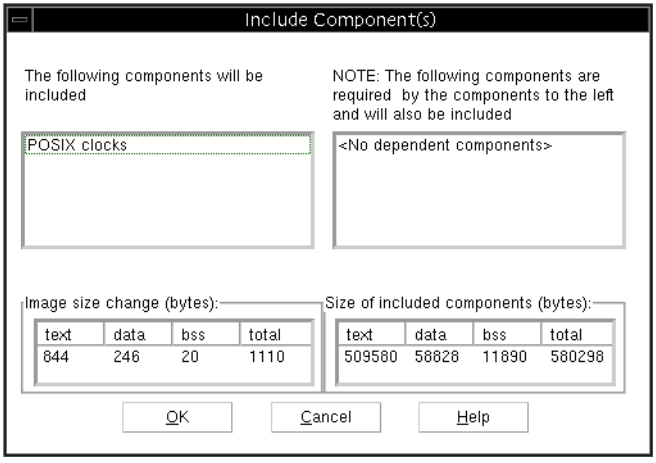
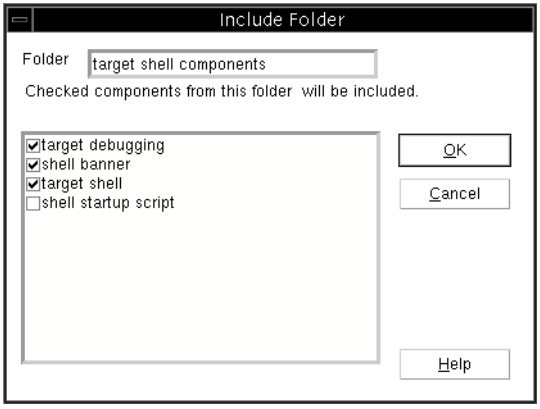


Figure 4-27 **Including a Component Folder**

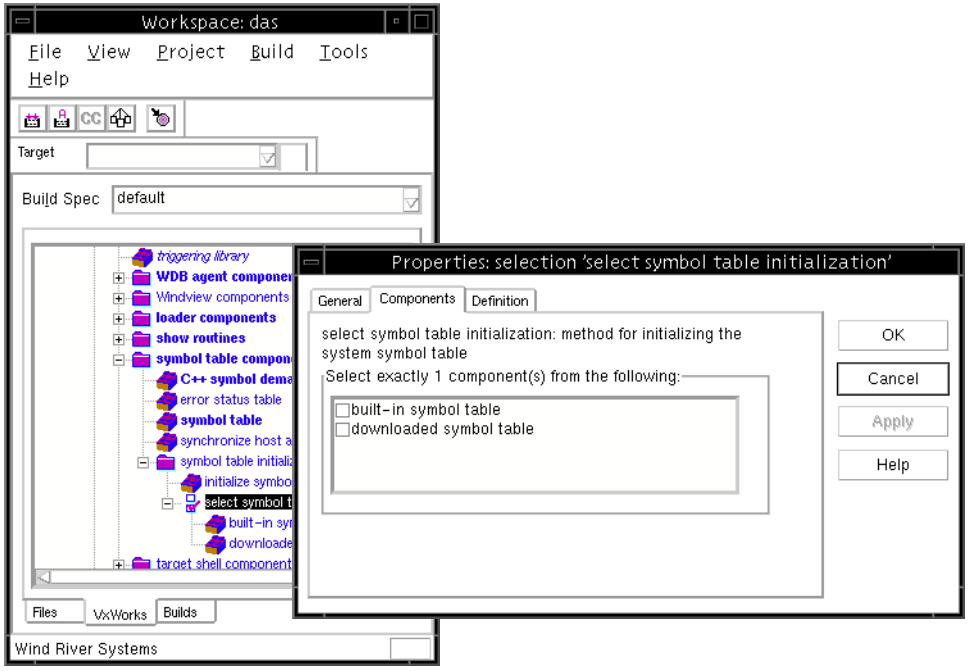


Tornado automatically calculates an estimate of the change in the size of the image resulting from the inclusion or exclusion, as well as the new image size. The Include and Exclude dialog boxes display this information. (Also see *Estimating Total Component Size*, p.124).

Some folders contain component options that are explicitly combinative or mutually exclusive (in the sense of being potentially conflictual). The name of these folders 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).

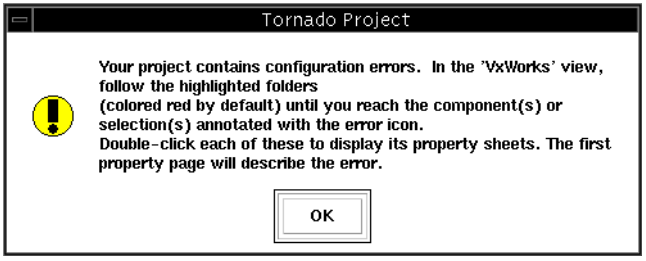
Figure 4-28 **Including Conflicting Components**

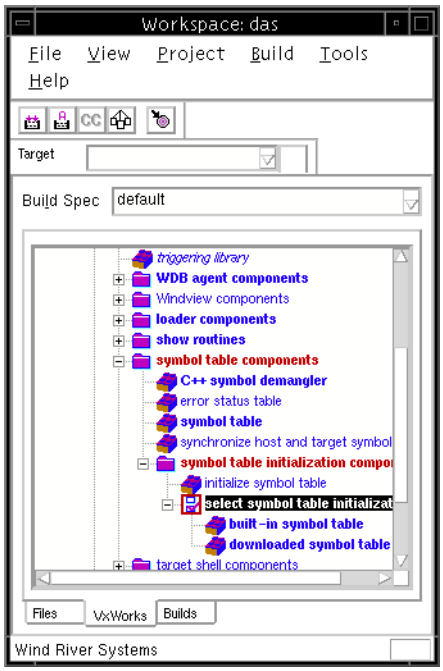


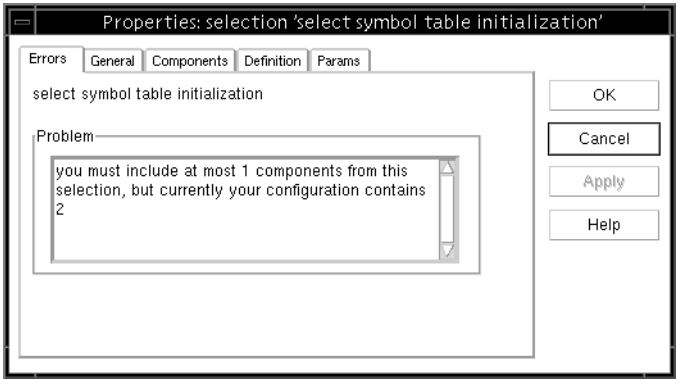
Component Conflicts

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**





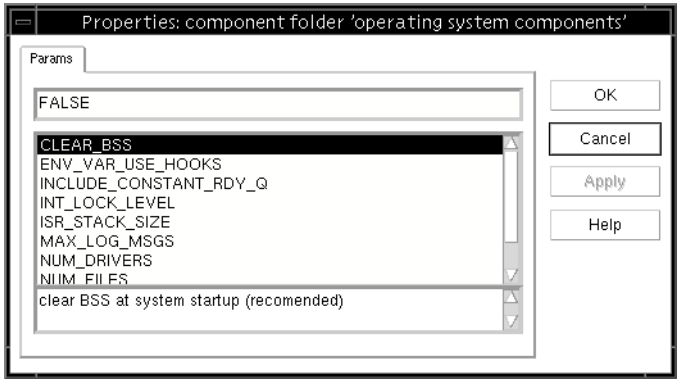


WARNING: You can build VxWorks even if there are conflicts between the components you have selected, but you may have linker errors or the run-time results may be unpredictable.

Changing Component Parameters

In the VxWorks view, the context menu provides access to component parameters (preprocessor macros). For example, selecting the operating system components folder, then Params for 'operating system components' from the context 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.

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 context 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.

4.3.4 Selecting the VxWorks Image Type

The default VxWorks is a RAM-based image. If you want to create something other than the default, double click on the build name in the Builds view to display the property sheet for that build. Then select the Rules tab, use the drop-down list to select the type of VxWorks image that you want to build, and click OK (Figure 4-32).

Figure 4-31 **Total Component Size**

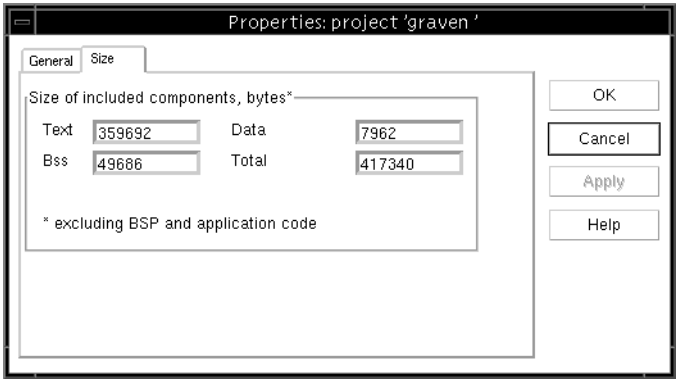
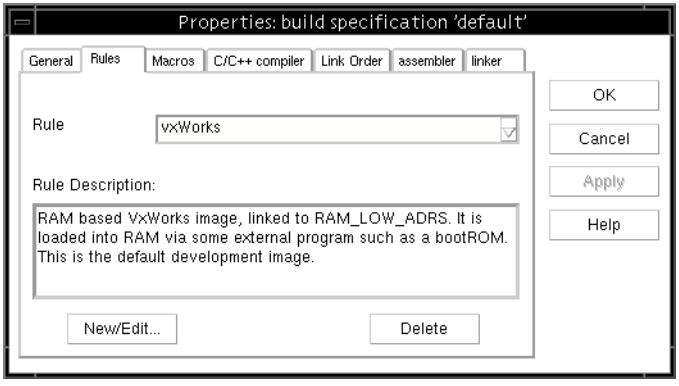


Figure 4-32 **Build Rules for VxWork Images**



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.

vxWorks_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.

vxWorks_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.

vxWorks_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.

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.132.

4.3.5 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 context 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.

See *4.2.4 Building a Downloadable Application*, p.104 for more information about a generic build, and *4.5 Working With Build Specifications*, p.128 for information about modifying builds and creating new build configurations.

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.134.

WARNING: The default compiler options include **-g** for debugging information. Using **-g** with the optimization option **-O** set to anything but zero may produce unexpected results. See *4.5 Working With Build Specifications*, p.128 for information about modifying builds and creating new build configurations.

4.3.6 Booting VxWorks

For information about booting VxWorks (and bootable applications) see *2.6 Booting VxWorks*, p.45. VxWorks images for the target simulator can be downloaded and booted with the context-menu Start command.

4.4 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.

Once you have created and tested a downloadable application and a customized version of VxWorks with which your application is designed to run, creating a bootable application is straightforward. To do so, you need to add application modules to a VxWorks project, and include application startup calls in the VxWorks system initialization routines. There are various ways to go about this, but if you have already created one or more projects for application code and a project for a custom VxWorks, you could simply:

- Add the application project(s) to the VxWorks workspace (or vice versa).
- Edit the VxWorks initialization file **usrAppInit.c**, adding calls to the application's initialization and startup routines.
- Use the project facility to help scale VxWorks.
- Build the bootable application.

For information about developing applications with the project facility, see *4.2 Creating a Downloadable Application*, p.95. For information about configuring and building VxWorks, see *4.3 Creating a Custom VxWorks Image*, p.110. For information about additional build options, see *4.5 Working With Build Specifications*, p.128.

4.4.1 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 context 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.

4.4.2 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 4-33, for example, illustrates the addition of a call to ***runItAll*( )**, the main routine in the application file **helloWorld.c**.

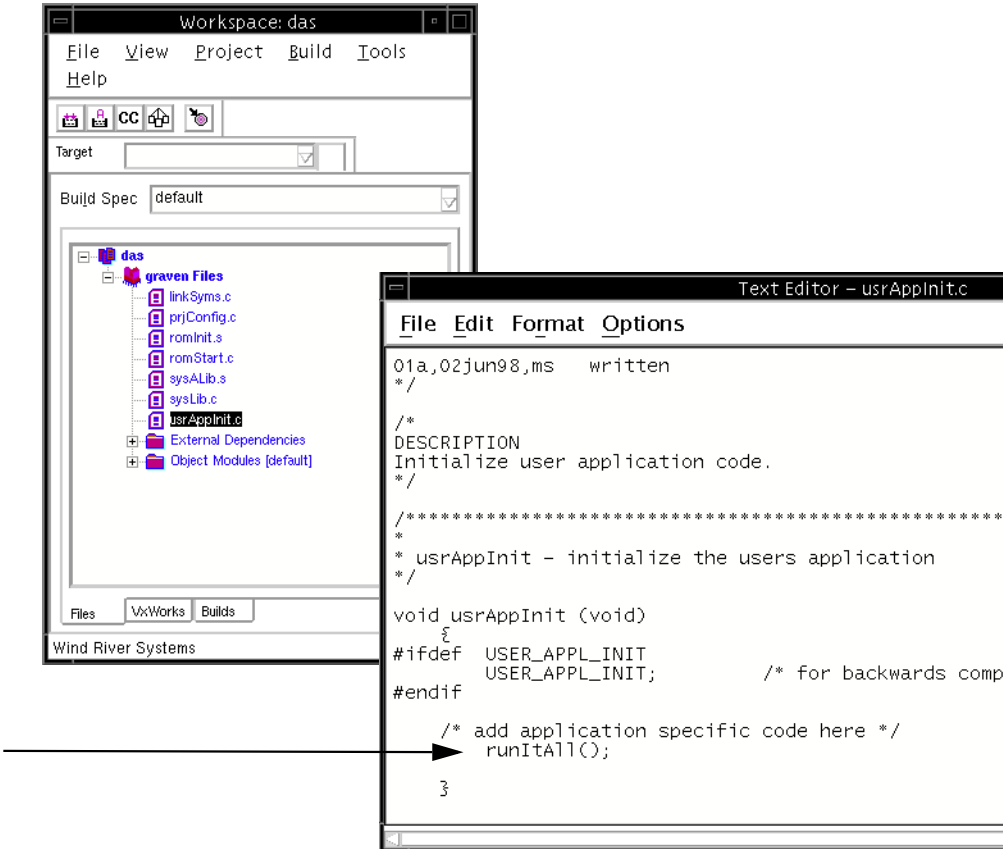
4.5 Working With Build Specifications

The project facility allows you to create, modify, and select specifications for any number of builds. A default build specification is 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 don't, and so on.

4.5.1 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.

Figure 4-33 **Adding Application Initialization Calls to VxWorks**



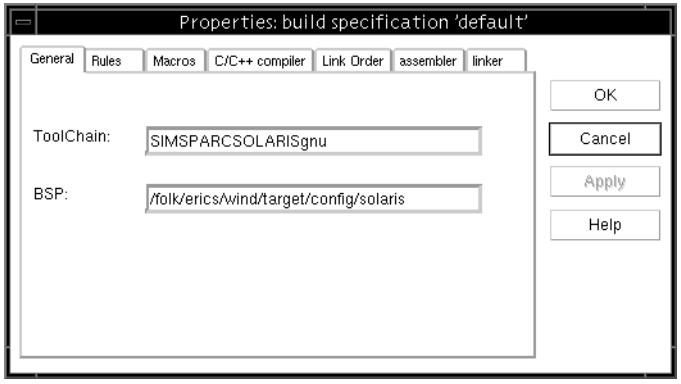
NOTE: For detailed information about compiler, assembler, and linker options, see the *GNU Toolkit User's Guide*.

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 4-34),

You can use the property sheet to modify:

- build targets
- makefile rules
- makefile macros for the compiler and linker

Figure 4-34 **Build Property Sheet**



- compiler options
- assembler options
- linker options

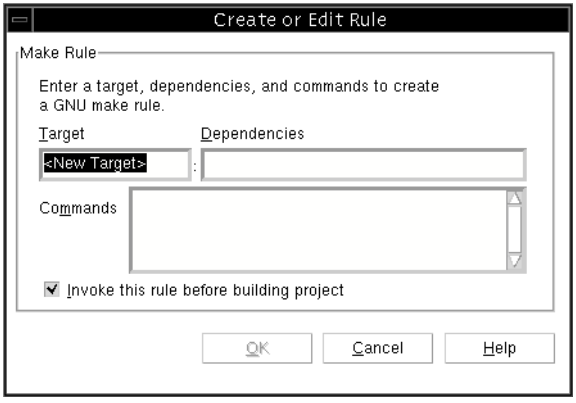
For information about build targets for downloadable applications, see *Build Specifications*, p.106. For information about build targets for bootable applications, see *4.3.4 Selecting the VxWorks Image Type*, p.124. Other features of the build property sheet are covered in the following sections.

WARNING: As of this release the project facility does not note changes in build options when you attempt to rebuild a project, and it will report that the build is up to date. If the only changes you have made are to the build options, you must select Rebuild All from the context menu to start a build with the new options.

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 4-35). 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). New rules are added to the *projectName***.wpj** file and written to the makefile prior to a build.

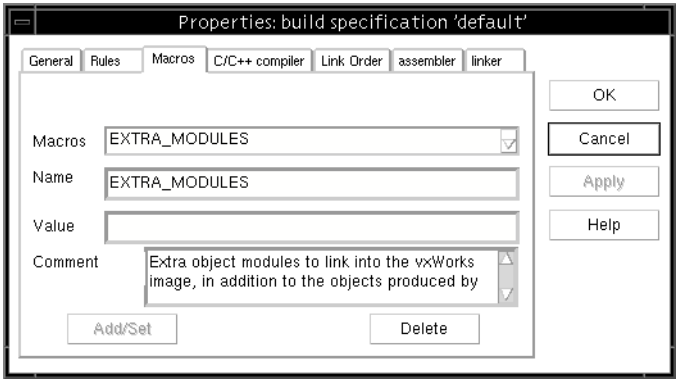
Figure 4-35 **Makefile Rule**



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 4-36).

Figure 4-36 **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 add a macro, change the name and

value of an existing macro, and click the Add/Set button. To change an existing macro, modify the value and click the Add/Set button.

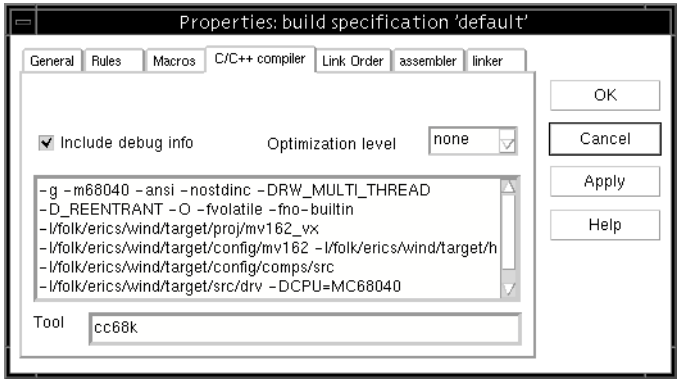
The recommended way to link library (archive) files to your project is to add the libraries to the list defined by the **LIBS** macro.

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 4-37).

WARNING: The default compiler options include **-g** for debugging information. Using **-g** with the optimization option **-O** 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.

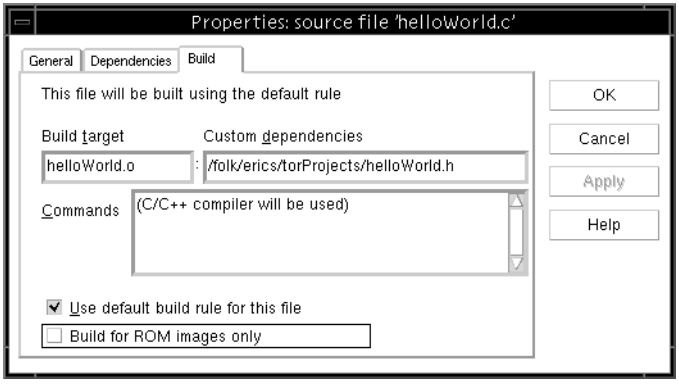
Figure 4-37 **Compiler Options**



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 context 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 4-38).

If the file should be used only when building a ROM-based image, check the Build for ROM images only box. See *4.3.4 Selecting the VxWorks Image Type*, p.124.

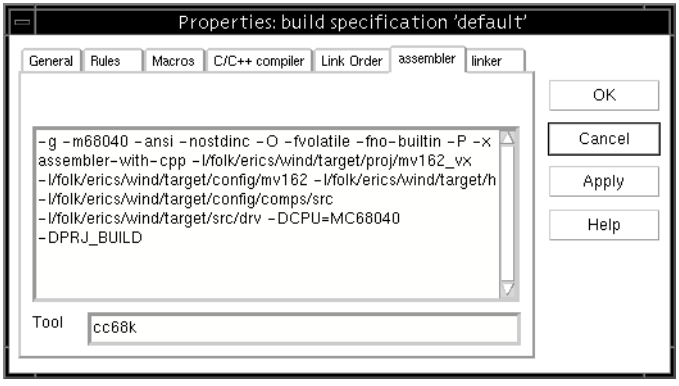
Figure 4-38 **Compiler Options for Individual Files**



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 4-39).

Figure 4-39 **Assembler Options**

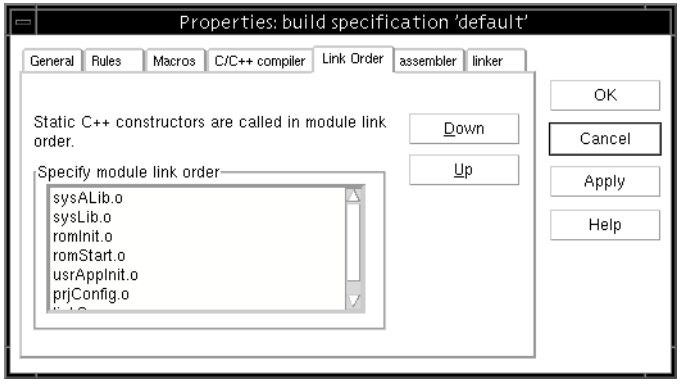


Link Order Options

Select the Link Order tab of the build specification property sheet to view module link order (Figure 4-40). 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.

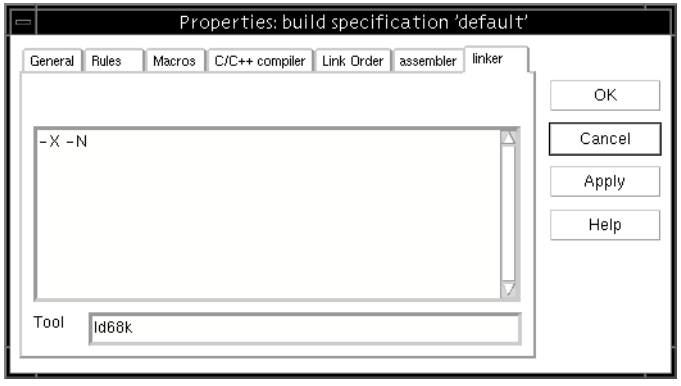
Figure 4-40 **Link Order Options**



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 4-41).

Figure 4-41 **Linker Options**



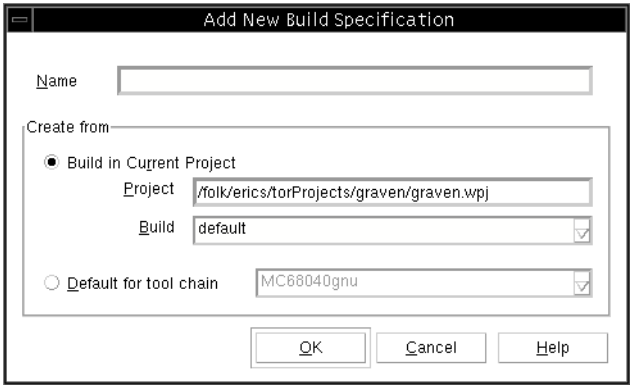
To link an object file with a project, list the full path to the file. The recommended way to link library (archive) files is to add the libraries to the list defined by the **LIBS** macro (see *Makefile Macros*, p.131).

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.106), and then link it with the current project.

4.5.2 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 context 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 4-42).

Figure 4-42 **New Build Specification**



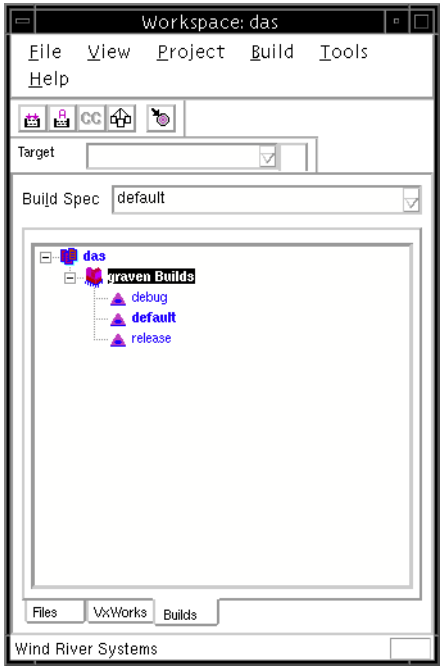
Once you have created a new build specification, use the build specification property sheet to define it (see *4.5.1 Changing a Build Specification*, p.128).

NOTE: For downloadable applications, it is often useful to create a build specification for the target simulator, and another for the real target hardware. For bootable applications and custom VxWorks images, you are usually restricted to the toolchains that support the CPU required by the BSP. But you can still create different build specifications (for example, with different optimization levels or rules).

4.5.3 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 4-43).

Figure 4-43 **Build Specification Selection**



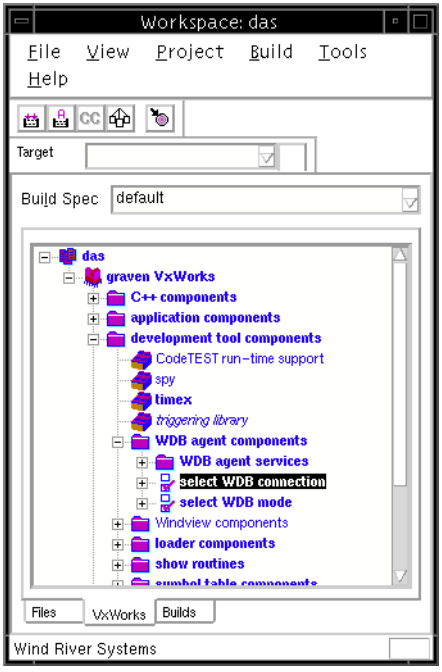
Binaries produced by a build are created in the *buildName* subdirectory of your project directory.

4.6 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, an 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.26; 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 4-44).

Figure 4-44 **Target Agent Connection Options**



To select an interface, select it from the list and select the Include '*component name*' option from the context 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.)

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 4-45). The options for the target agent communication interface are described below.

NOTE: Also see *Scaling the Target Agent*, p.141 and *Starting the Agent Before the Kernel*, p.142.

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 target simulator, add the WDB simulator pipe connection component.

Configuration for NetROM Connection

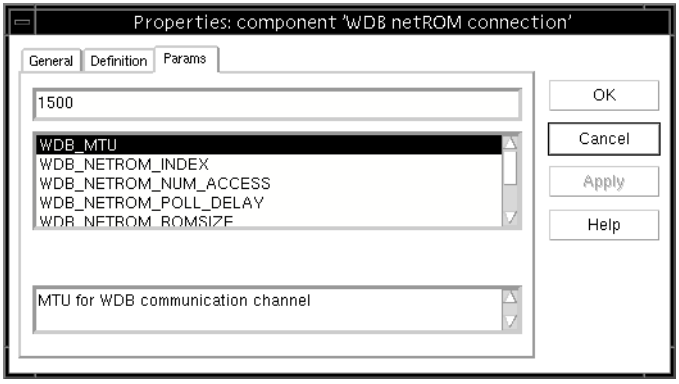
To configure the target agent to use a NetROM communication path, add the WDB netROM connection component. (See *2.5.4 The NetROM ROM-Emulator Connection*, p.37).

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 4-45).

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.

Figure 4-45 **NetROM Connection Macros**



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.131.

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 **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.

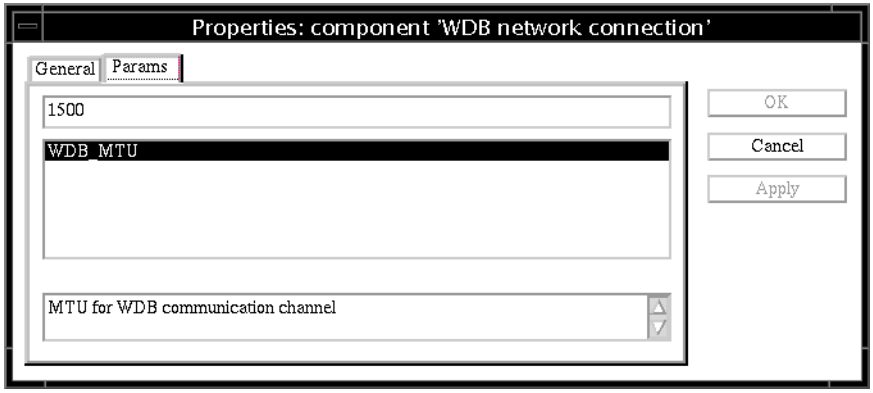
WARNING: On Intel i960 processors, the IMI (Initial Memory Image) file is located at the top of the ROM, and this memory space cannot be used for communication between the target server and the target agent (netrom dual-port RAM). **WDB_NETROM_ROMSIZE** must be assigned the value **WDB_NETROM_ROM_SIZE ROM_SIZE - 0x1000**. The dualport RAM base address in the NetROM must also be modified to reflect this change. For example, for a 27020 EPROM, *dprbase* should be set to 0x3e800 (rather than the default 0x3f800).

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.33).

The default MTU is 1500 octets. To change it, display the component property sheet, select the Params tab, select WDB_MTU item and change the value associated with it (Figure 4-46).

Figure 4-46 **Network Connection Macro**

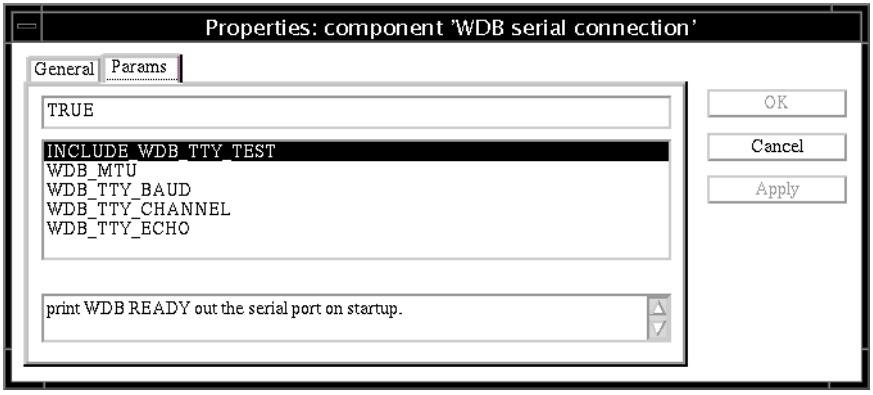


Configuration for Serial Connection

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.34).

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.

Figure 4-47 **Serial Connection Macros**



Configuration for tyCoDrv Connection

To configure a version 1.0 BSP 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:

7. VxWorks serial channels are numbered starting at zero. Channel 1 corresponds to the port labeled COM2 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 banner (**INCLUDE_WDB_BANNER**)
- VIO driver (**INCLUDE_WDB_VIO**)
- WDB task creation (**INCLUDE_WDB_START_NOTIFY**)
- 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. And if you are using a serial connection, you can 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 agent or only a system-mode 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 *VxWorks Programmer's Guide: Configuration and Build* 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 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 *5.2.6 Using the Shell for System Mode Debugging*, p.170 for information on using system mode from the shell, and *7.5 System-Mode Debugging*, p.261 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's

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.

***sysHwInit*( )** routine. Once ***sysHwInit*( )** is working properly, you no longer need to start the agent before the kernel.

CAUTION: If you are using the NetROM connection, and the agent is started before the kernel, then the agent cannot spawn the NetROM polling task to check periodically for incoming packets from the host. As a result there is no way for the host to get the agent's attention until a breakpoint occurs. On the other hand, there is no good reason to do so: you can set breakpoints in ***usrRoot*( )** to verify that VxWorks can get through this routine. Once this routine is working, you can start the agent after the kernel (that is, within the ***usrRoot*( )** routine), 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.

4.7 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.
- Do not have a target with NVRAM, and do not want to enter boot parameters at the target console each time it boots.
- 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. However, any changes you make to **config.h** may be absorbed by your projects unless they are masked by project facility selections (see page 92).

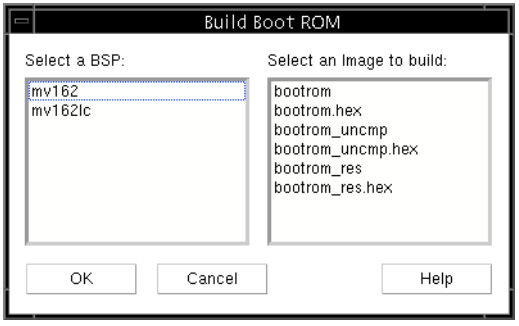
Configuring Boot Parameters

To customize a boot program for your development environment, you must edit **$WIND_BASE/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.48 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 4-48). Then click OK.

Figure 4-48 **Build Boot ROM**



The three main options for a boot images are:

bootrom
: A compressed boot image.

bootrom_uncmp
: An uncompressed boot image.

bootrom_res
: A ROM-resident boot image.

The .hex options are variants of the main options, with Motorola S-Record output.

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 *4.6 Configuring the Target-Host Communication Interface*, p.137.

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.145 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 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 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 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.145).

See the *VxWorks Programmer's Guide: Local File Systems* for more information about the TSFS.

5 Shell

5.1 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 *5.7 Tcl: Shell Interpretation*, p.198.

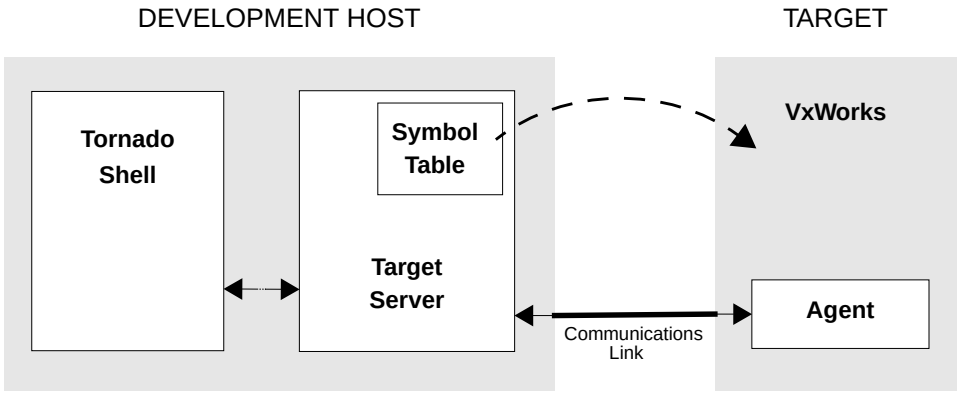
WindSh executes on the development host, not the target, but it allows you to spawn tasks, to read from or write to target devices, and to exert full control of the target.[1] Because the shell executes on the host system, you can use it with minimal intrusion on target resources. As with other Tornado 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 5-1.

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 5-1 **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.163, 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

5.2 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`.

5.2.1 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 

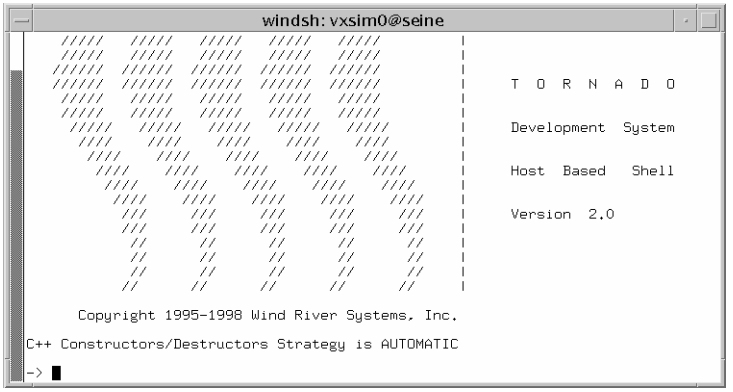
button.
- UNIX command line: Invoke **windsh**, specifying the target server name as in the following example:

```
% windsh phobos
```

In the first case, a shell window like that shown in Figure 5-2 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 5-2 **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.155).

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.

5.2.2 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 WRS Coding Conventions. (See *VxWorks Programmer's Guide: Coding Conventions*.)
2. Include these files in your project. (See *Creating, Adding, and Removing Application Files*, p.102.)
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 $WIND_BASE/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.

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.

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 5-1 **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.

Table 5-1 **WindSh Environment Variables** *(Continued)*

Variable	Result
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.
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 *5.7.2 Tcl: Calling Under C Control*, p.200.)

Example 5-1 **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
```

5.2.3 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 *D. Tornado Tools 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 *D. Tornado Tools Reference* or use the HTML help for the command. Although there is usually an entry with the same name in the *VxWorks Reference Manual*, it describes a related target routine, not the shell command.

Task Management

Table 5-2 summarizes the WindSh commands that manage VxWorks tasks.

Table 5-2 **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 *5.3.5 The "Current" Task and Address*, p.181 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 *5.6 Object Module Load Path*, p.196.) 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 5-3 summarizes the WindSh commands that report task information.

Table 5-3 **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.

The ***i*( )** command is commonly used to get a quick report on target activity. (To see this information periodically, use the Tornado browser; see *6. Browser*). If nothing seems to be happening, ***i*( )** is often a good place to start investigating. 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 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 5-4 shows the WindSh commands that display information from the symbol table, from the target system, and from the shell itself.

Table 5-4 **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
```

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 5-5 summarizes the WindSh commands of this type.

Table 5-5 **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 (**.**).
***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.48).
***sysSuspend*( )**	If supported by the target-agent configuration, enter system mode. See *5.2.6 Using the Shell for System Mode Debugging*, p.170.

Table 5-5 **WindSh Commands for System Modification and Debugging** *(Continued)*

Call	Description
***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 *VxWorks Programmer's Guide: Configuration and Build*.

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 Intel x86, Intel I960(CX/JX/HX), MIPS R4650, and some PPC processors (PPC860, PPC603, PPC604, PPC403). 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 5-6 summarizes these commands. For more discussion of these shell commands, see *VxWorks Programmer's Guide: C++ Development*.

Table 5-6 **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.155.

Object Display

Table 5-7 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 *6. Browser*.

Table 5-7 **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.

Table 5-7 **WindSh Commands for Object Display** *(Continued)*

Call	Description
***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.
***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.

Table 5-7 **WindSh Commands for Object Display** *(Continued)*

Call	Description
***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 5-8 summarizes the WindSh commands that display information about the VxWorks network.

Table 5-8 **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.

In order for a protocol-specific command to work, the appropriate protocol must be included in your VxWorks configuration.

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*( )**.

5.2.4 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.179.

5.2.5 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 *D. Tornado Tools 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.

5.2.6 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. One major shell feature is not available in system mode: you cannot execute expressions that call target-resident routines. You can still spawn tasks, but bear in mind that, because the entire system is stopped, a newly-spawned task can only execute when 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 *4.6 Configuring the Target-Host Communication Interface*, p.137. 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).

***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 5-2 **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.

5.2.7 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.

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:

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.

```
-> 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.

5.3 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.

5.3.1 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.155.

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**.

5.3.2 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 5-9.

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.179. Table 5-10 shows the various data types available in the shell C interpreter, with examples of how they can be set and referenced.

Table 5-9 **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**

Table 5-10 **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.

3. Memory allocated for string constants is never freed by the shell. See *5.3.8 Strings*, p.183 for more information.
4. ***d*( )** is one of the WindSh commands, implemented in Tcl and executing on the host.

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.

5.3.3 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.

5.3.4 Expressions

Shell expressions consist of literals, symbolic data references, function calls, and the usual C operators.

Literals

The shell interprets the literals in Table 5-11 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 5-11 **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.

C compilers usually prefix all user-defined identifiers with an underbar, so that **myVar** is actually in the symbol table as **_myVar**. 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 5-12 in the same way as the C compiler.

Table 5-12 **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 ***myFunc*2( )** plus 4:

```
4 + myFunc2 ( )
```

However, the following expression evaluates to the address of ***myFunc*2( )** 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.180.

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 ()
```

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"
```

as are:

```
-> 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.

5.3.5 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.

5.3.6 Assignments

The shell C interpreter accepts assignment statements in the 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)
```


5.3.7 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 the section *Scripts: Redirecting Shell I/O*, p.189 below.

5.3.8 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.141.

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.

5.3.9 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:

```
❶ -> 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:

```
❷ -> 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 ❶ would be correct and ❷ 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 **&**.

5.3.10 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.

5.3.11 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 *5.7 Tcl: Shell Interpretation*, p.198.

- **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.

5.3.12 C-Interpreter Primitives

Table 5-13 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.168.

5.3.13 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 5-14); 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 5-14 lists special terminal characters frequently used for shell control. For more information on the use of these characters, see *5.5 Shell Line Editing*, p.193 and *5.2.7 Interrupting a Shell Command*, p.173.

5.3.14 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*( )**:

Table 5-13 **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*( )	*tt*( )
ifShow( )	*s*( )	*tw*( )
inetstatShow( )	*semPxShow*( )	*udpstatShow*( )
intVecShow( )	*semShow*( )	*unld*( )
iosDevShow( )	*shellHistory*( )	*version*( )
iosDrvShow( )	*shellPromptSet*( )	*w*( )
iosFdShow( )	*show*( )	*wdShow*( )

Table 5-14 **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.

Table 5-14 **Shell Special Characters** *(Continued)*

stty Setting	Common Value	Description
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**.

```
-> 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:

```
❶  -> <startup
❷  -> < /usr/fred/startup
```

cause the shell to read and execute the commands in the file **startup**, either on the current working directory as in ❶ or explicitly on the complete path name in ❷. If your working directory is **/usr/fred**, commands ❶ and ❷ 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 (online or in *D. Tornado Tools 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 *5.7 Tcl: Shell Interpretation*, p.198.

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 *5.7.3 Tcl: Tornado Shell lnitialization*, p.201.

5.4 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 *5.3 The Shell C-Expression Interpreter*, p.174 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.

5.4.1 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.

5.4.2 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 *5.4.1 Overloaded Function Names*, p.191.

```
-> 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
```

5.5 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 5-14 (*5.3.13 Terminal Control Characters*, p.186) are set by

6. The WindSh Tcl-interpreter interface is described in *5.7 Tcl: Shell Interpretation*, p.198.

the host **stty** command (which you can call from the Tcl interpreter; see *5.7 Tcl: Shell Interpretation*, p.198). They must be single characters, usually control characters; Table 5-15 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 5-15 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.160) 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 5-15; you can then edit and execute the line with one of the commands from Table 5-15.

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 5-15 **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 5-15 **Shell Line-Editing Commands**

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 5-15 **Shell Line-Editing Commands**

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.

5.6 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 5-3 **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" (changed from the default). 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.155.

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 5-16 lists the affected utilities. The object modules are in **wind/target/lib/obj***cputype***gnuvx**.

Table 5-16 **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**

5.7 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 *B. 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*( )**.

5.7.1 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 *Tornado API Guide* (or the *Tornado API* entry in the *Tornado Online Manuals*).

All of the commands defined for the C interpreter (*5.2.3 Invoking Built-In Shell Routines*, p.156) 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 *Tornado API Guide: 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
```

5.7.2 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
```

5.7.3 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 two places: first under **wind**, and then 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 5-4 **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
```

5.8 The Shell Architecture

5.8.1 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 5-17 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 5-17 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.

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

Table 5-17 **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**.

(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.

5.8.2 Shell Components

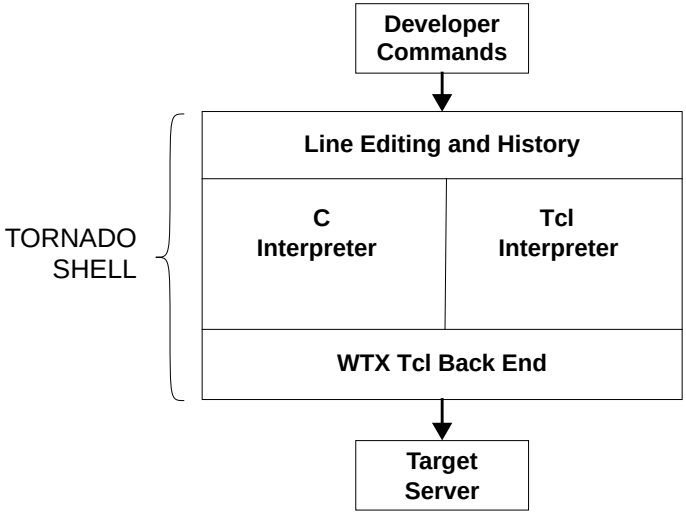
The Tornado shell includes two interpreters, a common front end for command entry and history, and a back end that connects the shell to the global Tornado environment to communicate with the target server. Figure 5-3 illustrates these components:

Line Editing
: The line-editing and command history facilities are designed to be unobtrusive, and support your access to the interpreters. *5.5 Shell Line Editing*, p.193 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.

Tcl Interpreter
: An interface for extending the shell or automating shell interactions, described in *5.7 Tcl: Shell Interpretation*, p.198.

Figure 5-3 **Components of the Tornado Shell**

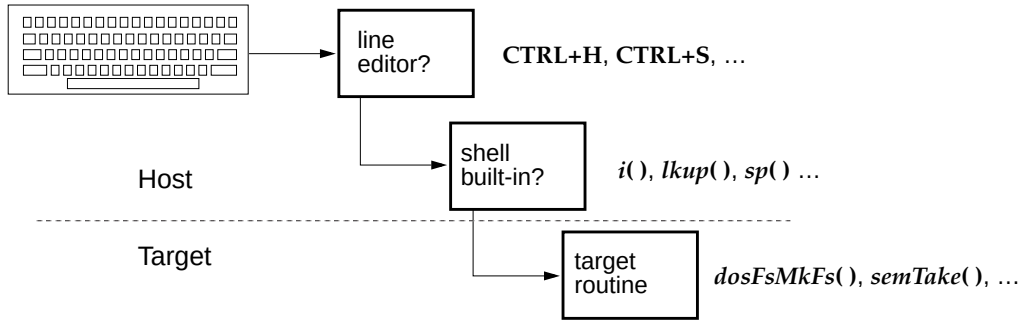


WTX Tcl
: The back-end mechanism that ties together all of Tornado; the Wind River Systems Tool Exchange protocol, implemented as a set of Tcl extensions.

5.8.3 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 5-4. First, input is examined for special editing keystrokes (described in *5.5 Shell Line Editing*, p.193). 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 *5.2.3 Invoking Built-In Shell Routines*, p.156 for more information. For a list of all WindSh primitives, see Table 5-13. .

Figure 5-4 **Layers of Interpretation in the Shell**



6

Browser

6.1 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.)

6.2 Starting the Browser

There are two ways to start a Tornado browser:

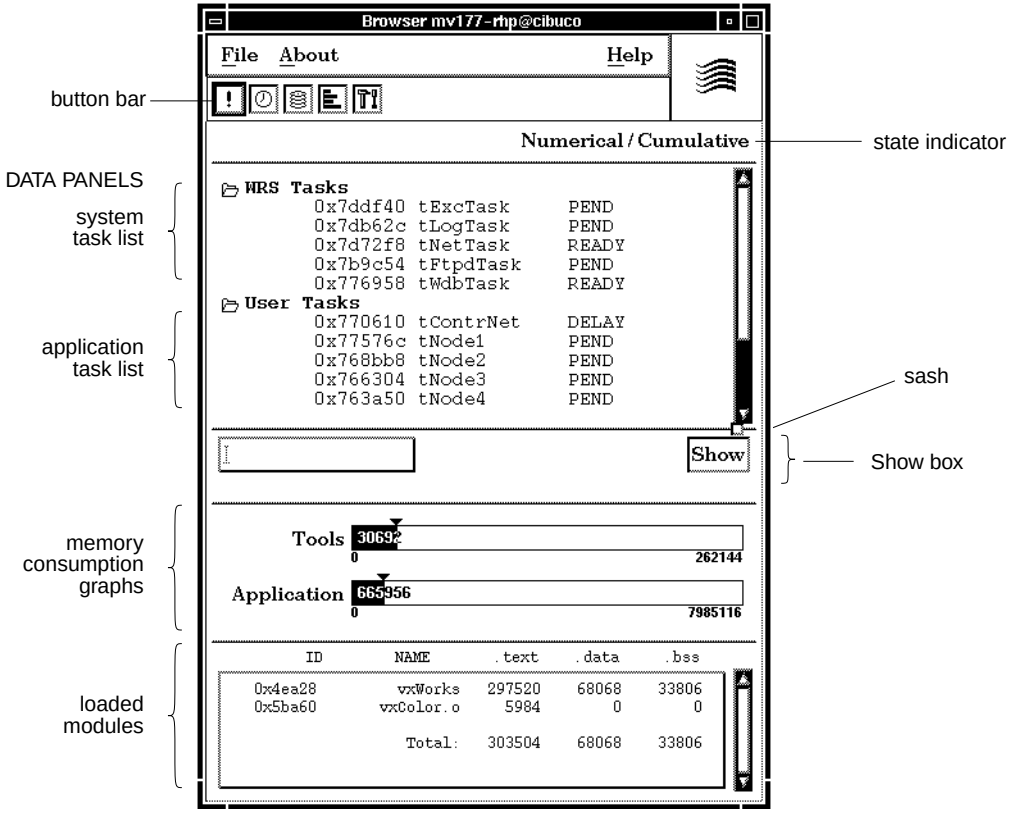
- From the launcher: select the desired target and press the 

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 6-1.

Figure 6-1 **Target Browser Window**



6.3 Anatomy of the Target Browser

The main browser window, shown in Figure 6-1, 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 6-1 (system task list, application task list, memory-consumption graphs, and loaded modules) provide overall information about your target system. See *6.5 Data Panels*, p.212 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. *6.4 Browser Menus and Buttons*, p.210 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 *6.9 The Spy Window*, p.224. Default state. Converse: Differential.
Differential	Show CPU usage within the sampling interval in the spy window. See *6.9 The Spy Window*, p.224. 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 *6.6 Object Browsers*, p.213.

6.4 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.

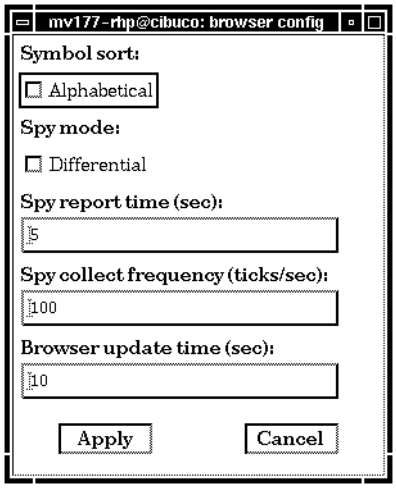
The row of buttons immediately below the menu bar provides browser-specific controls, with the following meanings:

 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 *6.10 The Stack-Check Window*, p.226).
 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 *6.9 The Spy Window*, p.224). 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 6-2 shows the browser config form displayed when you press this button.

You can use the browser config form produced by the button (Figure 6-2) to change the following browser parameters:

Figure 6-2 **Form: 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 *6.9 The Spy Window*, p.224).

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.

6.5 Data Panels

The main browser window includes several information panels (labeled along the left of Figure 6-1) 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 *6.6.1 The Task Browser*, p.214.

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.

1. To set the size of this memory pool, see *Scaling the Target Agent*, p.141.

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 *6.6.4 The Memory-Partition Browser*, p.218, applied to the system-memory partition; this display is shown in Figure 6-12).

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.

6.6 Object Browsers

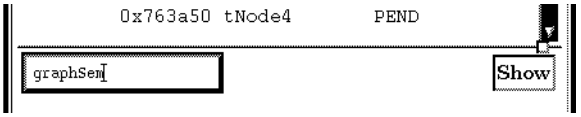
The Show box (in the middle of the main browser window) gives you access to the browser's specialized object displays. Type either the name or the ID of a system object in the text-entry field to the left of this panel. Then press the Show button (or simply press the **ENTER** key) to bring up a browser 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 (*6.7 The Module Browser*, p.221). 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 6-3 shows the Show box filled in with a request to display a browser for an object called **graphSem**:

To dismiss specialized object browsers, use the window manager's controls.

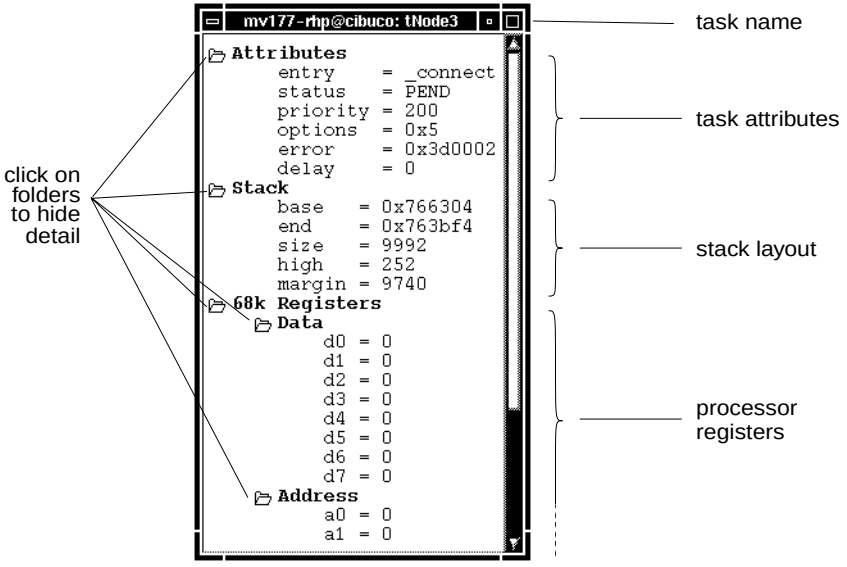
Figure 6-3 **Filling in the** Show **Box**



6.6.1 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 6-4.

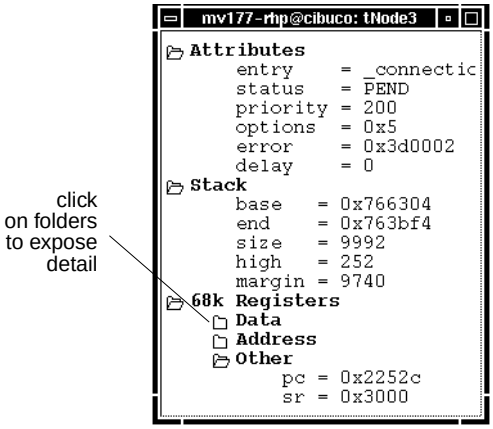
Figure 6-4 **Task Browser (Initial Display)**



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 6-5 shows another task browser running on the same target architecture, but with most of the hardware registers hidden.

Figure 6-5 **Task Browser (Hiding Registers)**



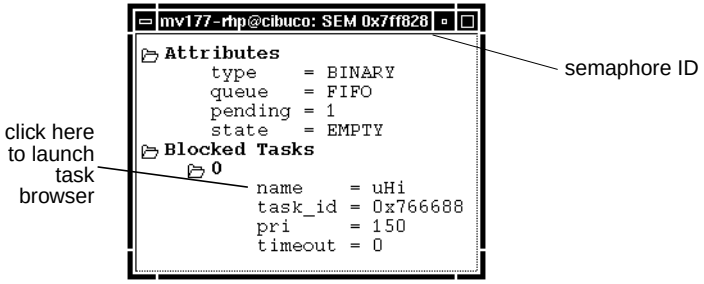
Task-browser windows close automatically when the corresponding tasks are deleted.

6.6.2 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 6-6. 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 6-6 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.

Figure 6-6 **Semaphore Browser**



POSIX semaphores have a somewhat different collection of attributes, and the browser display for a POSIX semaphore reflects those differences. Figure 6-7 shows an example of a browser display for a POSIX semaphore. Similarly, the semaphore browser adapts to shared-memory semaphores; Figure 6-8 exhibits that semaphore display.

Figure 6-7 **POSIX Semaphore**

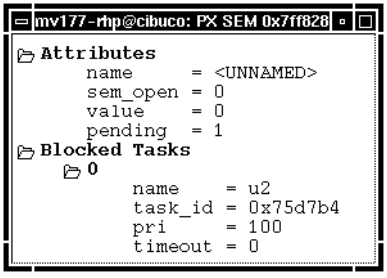
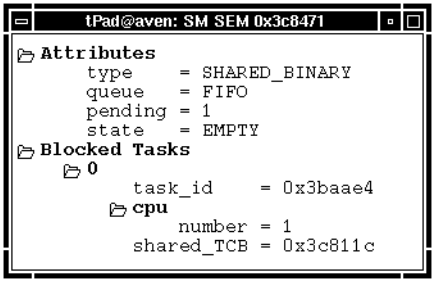


Figure 6-8 **Shared-Memory Semaphore**



Semaphore-browser windows are closed automatically when the corresponding semaphore is deleted.

6.6.3 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 6-9 is displayed.

Figure 6-9 **Message Queue**

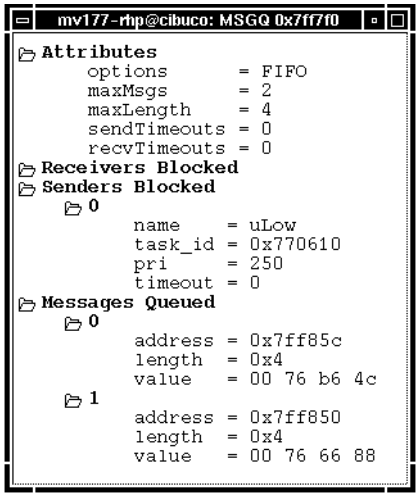
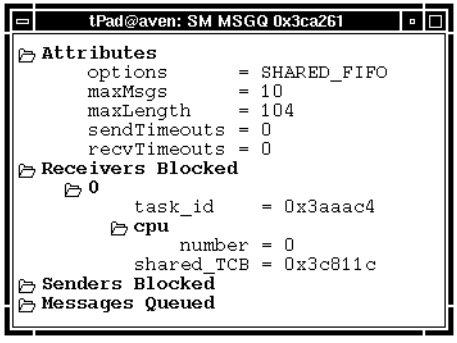


Figure 6-10 **Shared-Memory Message Queue**

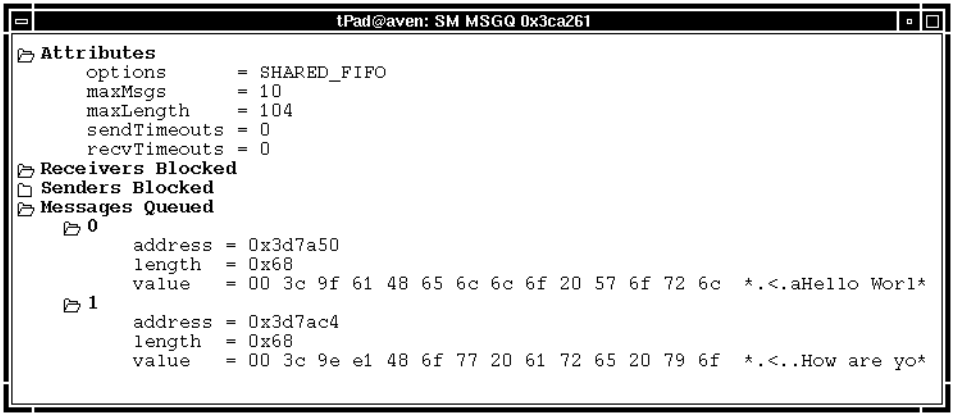


As well as displaying the attributes of the message queue, the message-queue browser shows three queues. Receivers Blocked shows all tasks waiting for messages from the message queue. Senders Blocked shows all tasks waiting for space to become available to place a message on the message queue. Messages Queued shows the address and length of each message currently on the message queue. As shown in Figure 6-10, 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 6-11 shows a shared-memory message queue in a display widened for this purpose.

Figure 6-11 **Message Queue Browser: Wider Display**



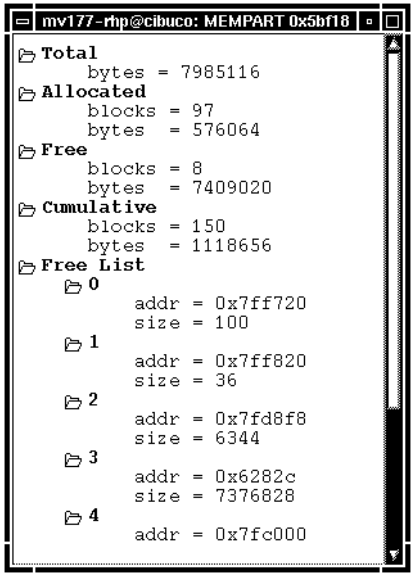
Message-queue browser windows are closed automatically when the corresponding message queue is deleted.

6.6.4 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 6-12 shows **memSysPartId**, the VxWorks system memory partition.

By default the memory-partition browser displays the following:

Figure 6-12 **Memory-Partition Browser**



- 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.

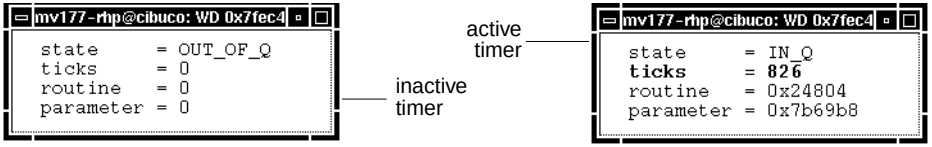
As for other object browsers, you can control the level of detail visible by clicking on the folder icons beside each heading.

6.6.5 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 6-13.

Before you start a timer, the display resembles the one on the left of Figure 6-13; only the state field is particularly meaningful. After the timer starts counting,

Figure 6-13 **Watchdog Browser**

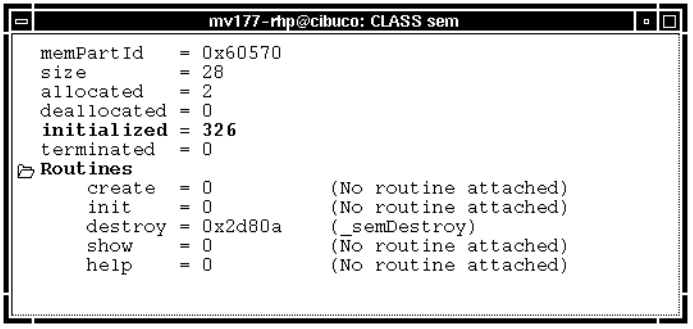


however, you can see the number of ticks remaining, the address of the routine to be executed when the timer expires, and the address of its parameter.

6.6.6 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 6-14 shows the display for **semClassId** (the semaphore class).

Figure 6-14 **Class Browser (Semaphore Class)**



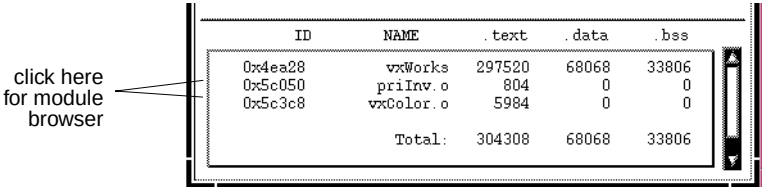
You can get a list of the class identifiers in your run-time by executing the following in a shell window:

```
-> lkup "ClassId"
```

6.7 The Module Browser

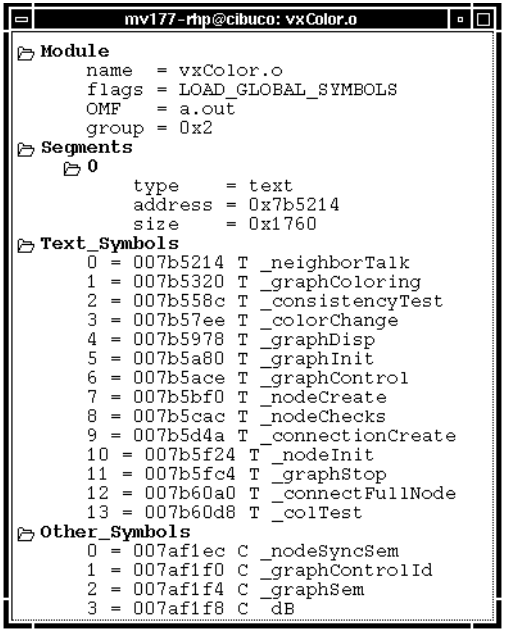
To inspect the memory map of any currently loaded module, click on the line that lists the module in the loaded-module list (the bottom panel in the main browser window).

Figure 6-15 **Loaded-Module List in Main Browser Window**



The browser opens a specialized object-module browser resembling Figure 6-16 for the selected module.

Figure 6-16 **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 6-1), and the symbol name (in its internal representation—C++ symbols are displayed "mangled", and all compiled-language symbols begin with an underbar).

Table 6-1 **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

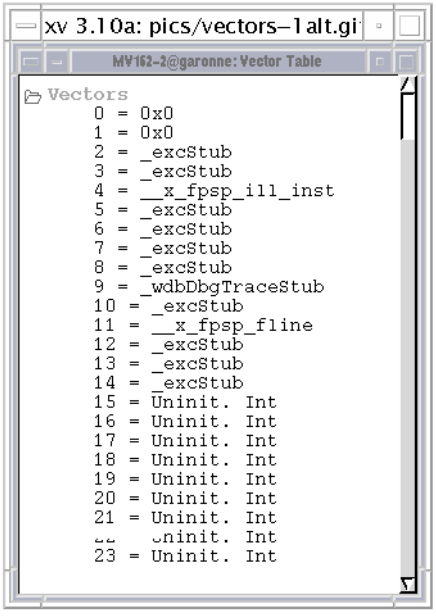
For symbols that represent system object, clicking on the symbol name brings up the specialized object browser; see *6.6 Object Browsers*, p.213.

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.

6.8 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 the HP-UX simulator, PowerPC, and ARM.) The display is similar to Figure 6-17.

Figure 6-17 **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

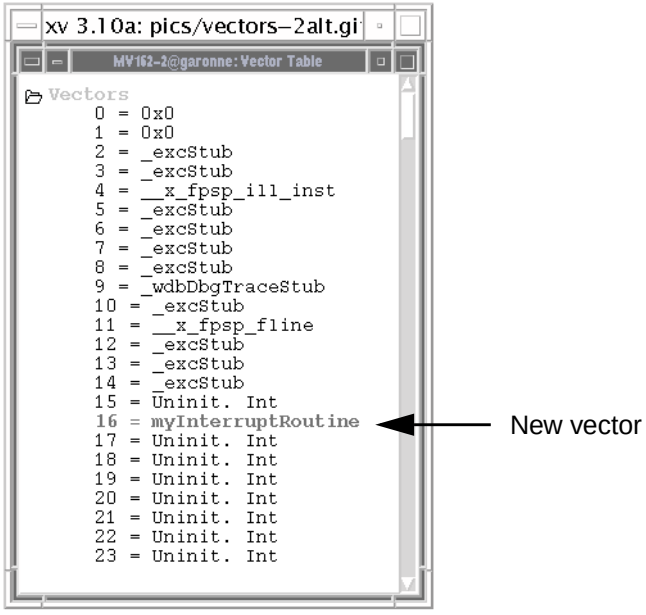
Default Trap
: default trap (Sparc)

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 6-18.

Figure 6-18 **New Interrupt Vector**

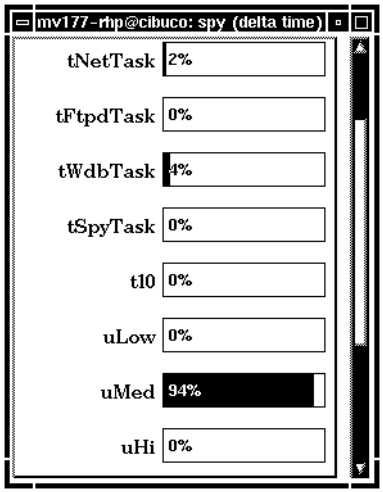


6.9 The Spy Window

Pressing the spy button produces a window similar to Figure 6-19. 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.

Figure 6-19 **Spy Window**



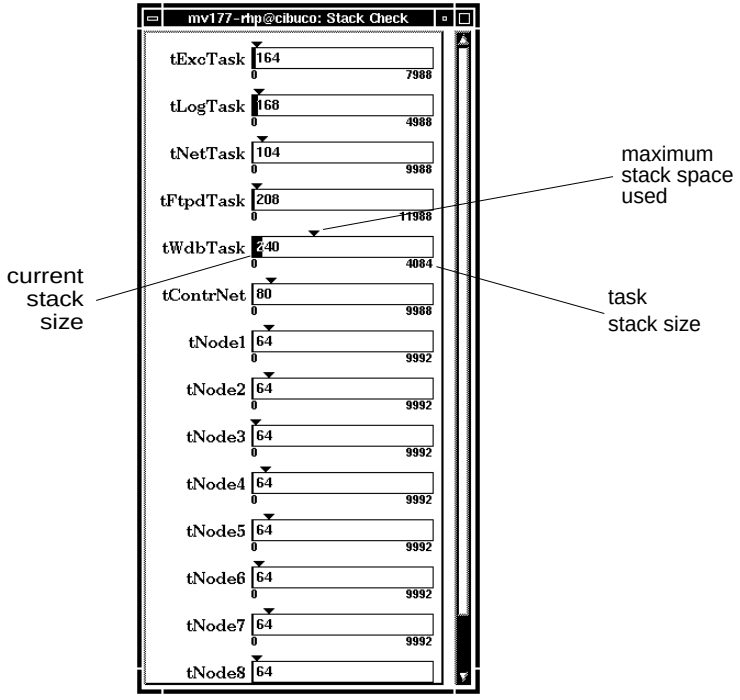
Spy data is reported in one of two modes (selected with the browser config form shown in Figure 6-2). Reports in Cumulative mode (noted on the state indicator line) show total CPU usage since you first display the spy window. Reports in Differential mode reflect only the CPU usage since the last update. 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**.

6.10 The Stack-Check Window

When you press the stack-check button , the browser displays a stack-check window similar to Figure 6-20. The stack-check window summarizes the current and maximum stack usage for each task currently running.

Figure 6-20 **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.

6.11 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.

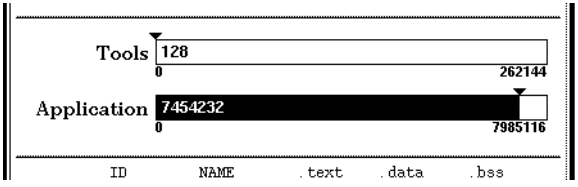
6.12 Troubleshooting with the Browser

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.

6.12.1 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 6-21 corresponds to a memory leak in an application that has run long enough to almost completely run out of memory.

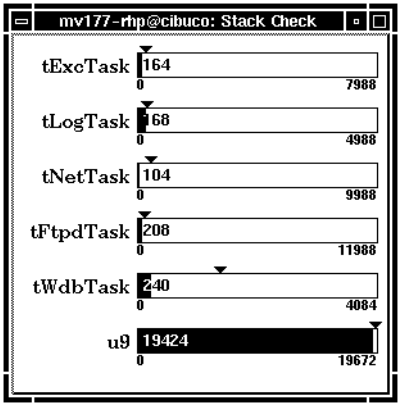
Figure 6-21 **A Memory Leak as Seen in the Browser**



6.12.2 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 6-22.

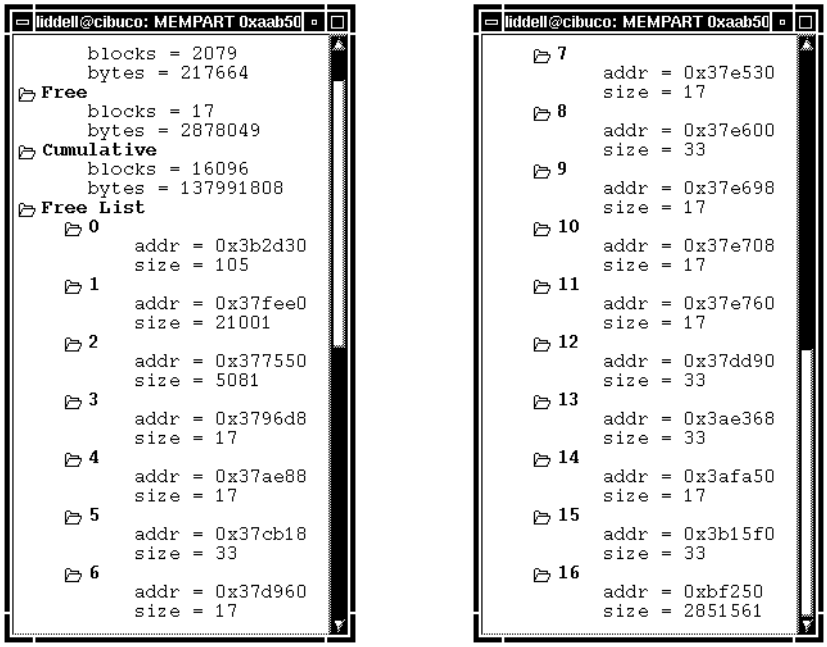
Figure 6-22 **Stack Overflow on Task u9**



6.12.3 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 6-23 shows an example of a growing free-list with many small blocks, characteristic of memory fragmentation.

Figure 6-23 **Fragmented Memory as Seen in the Browser**



6.12.4 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 6-24) 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 6-25.

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 6-26) 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 6-24 **Browser: uHi Pended**

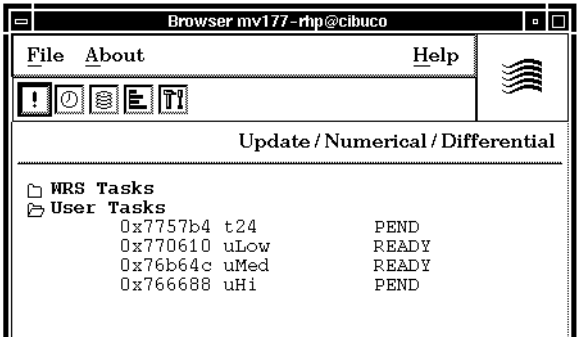


Figure 6-25 **Task Browsers for uHi, uMed, uLow**

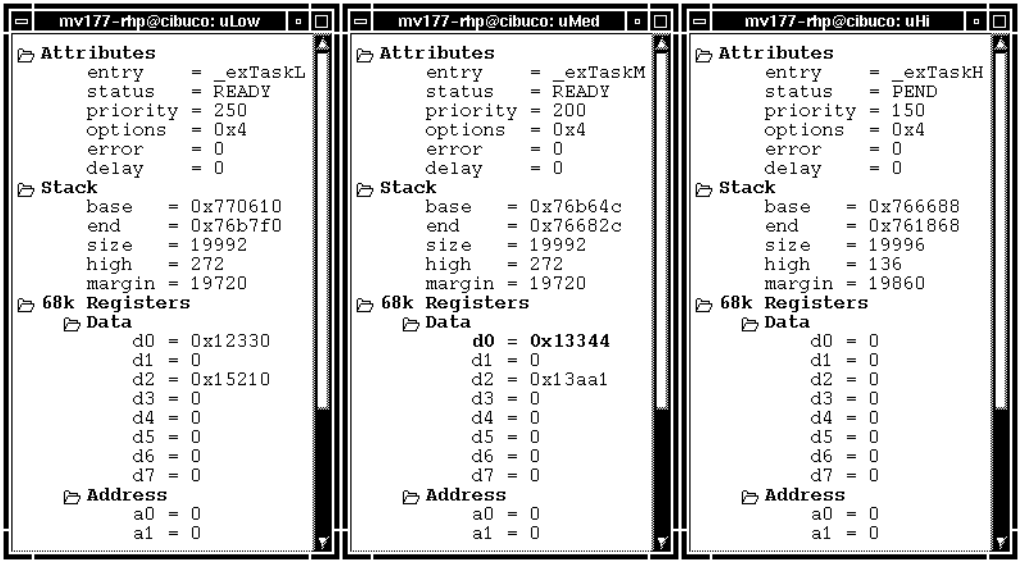
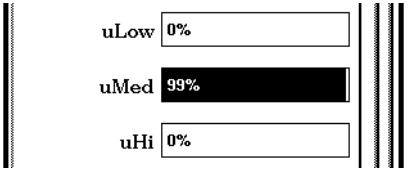


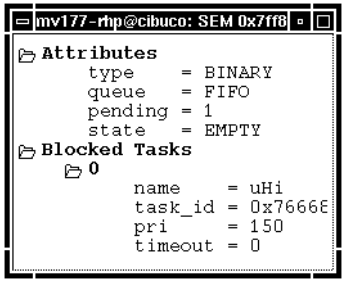
Figure 6-26 **uMed Monopolizing CPU (Spy Window Excerpt)**



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 6-27) confirms the dawning suspicion: **uHi** is indeed blocking on the semaphore, which **uLow** cannot release because **uMed** has preempted it.

Figure 6-27 **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.

6.13 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 6-1 **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
    }
}
```

7

Debugger

7.1 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*.

7.2 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 sparc &
  ```

  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.238 for details) or the **target wtx** command (see *Managing Targets*, p.257).

7.3 A Sketch of CrossWind

Figure 7-1 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.238. 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 7-1 shows a summary of each button's purpose. For more detailed descriptions of these buttons, see *CrossWind Buttons*, p.244.

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 7-1, 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 7-1 **CrossWind Display**

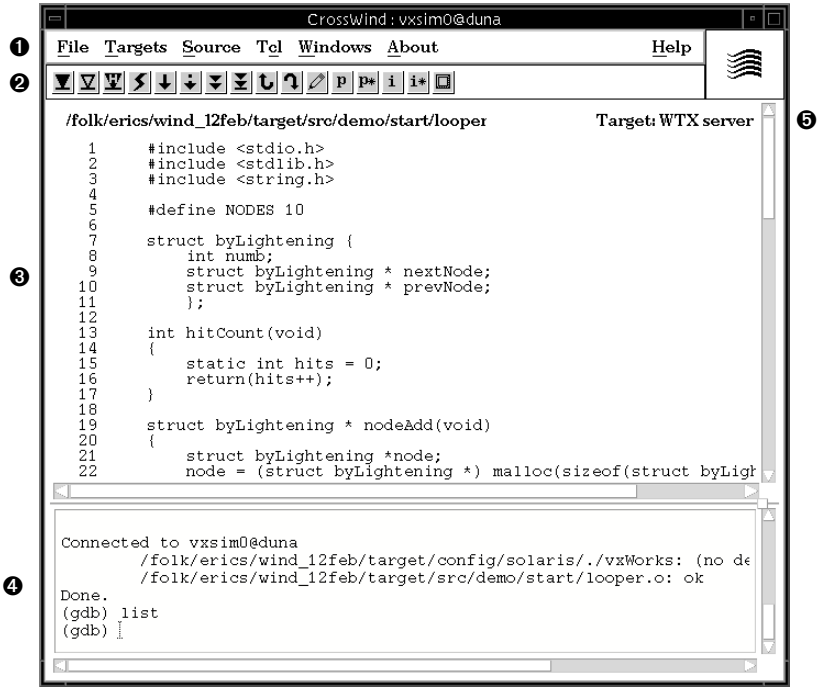


Table 7-1 **Summary of CrossWind Buttons**

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.253.

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 7-2.

Table 7-2 **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.

7.4 CrossWind in Detail

This section describes the debugger commands and controls in detail. *7.4.1 Graphical Controls*, p.236 is a complete discussion of all graphical debugger controls. *7.4.2 Debugger Command Panel: GDB*, p.252 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.

7.4.1 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 7-1, 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 7-1, 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.244.)

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 7-2).

Figure 7-2 **CrossWind Menu Bar**

The following paragraphs describe the effect of each debugger-specific menu command.

File Menu

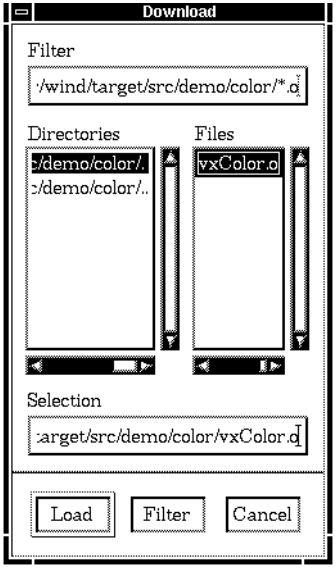
File>Quit
: Ends the debugging session. If a target is attached, Quit also kills any suspended routines from the debugging session.

  (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 7-3, 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 7-3 **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.259.

In the command panel, you can use a form of the **load** command to get around this problem. See *What Modules to Debug*, p.253.

Targets Menu

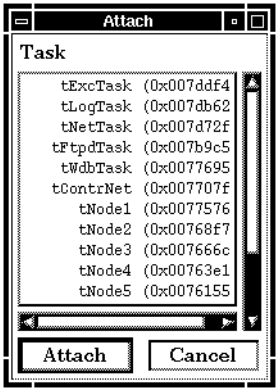
The commands in the Targets menu allow you to select or change debugging targets.

⚠ **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 7-4). When you select one, the debugger stops the task.

Figure 7-4 **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 7-4). If the debugger is already connected to a target, selecting a new target releases the current target from debugger control.

Figure 7-5 **Connect**

Source Menu

The commands in the Source menu control how your program is displayed.

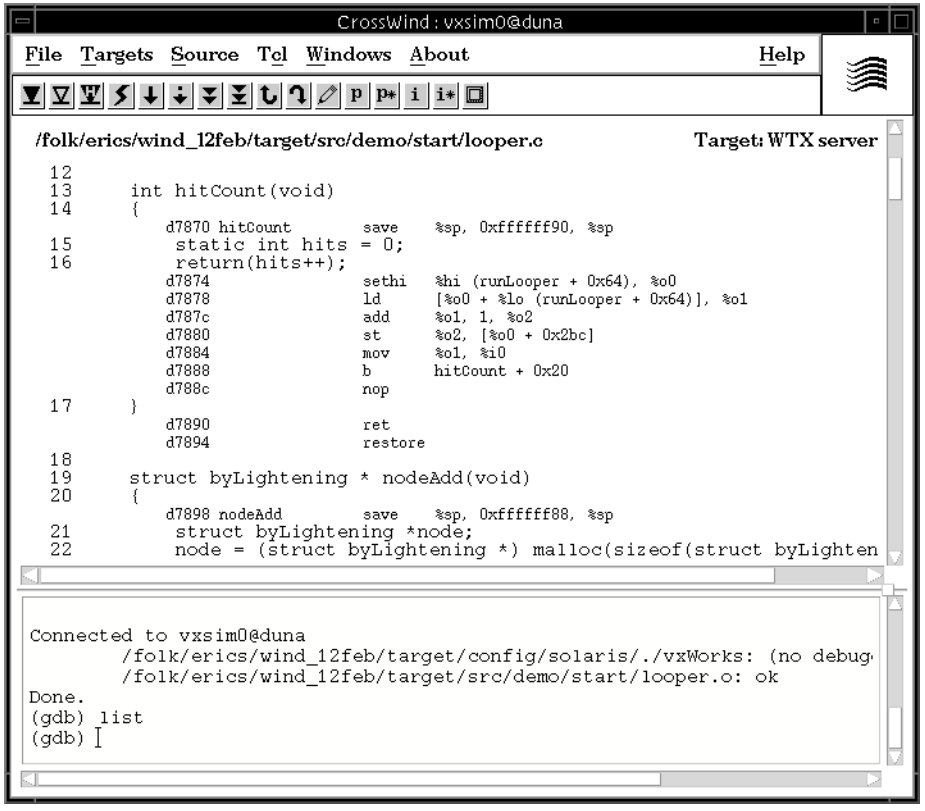
Source>C/C++
: Displays the original high-level language source code (usually C or C++). This style of display is only available for modules compiled with debugging information (the **-g** option, for the GNU compiler). When this display is available, it is also the default style of program display.

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 7-6 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 7-6 **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 *7.7 Tcl: CrossWind Customization*, p.273), this provides a way to restore the environment after experiments with custom modifications.

Tcl>Reread Home
: Re-initializes the definitions from **.wind/crosswind.tcl** in your home directory (see *7.7.1 Tcl: Debugger Initialization Files*, p.273).

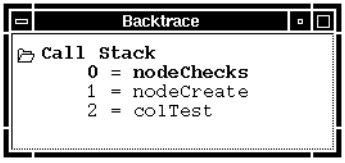
Tcl>Reread All
: Re-initializes the complete graphical environment defined in Tcl resource files, including both the basic CrossWind definitions and the definitions from **~/.wind/crosswind.tcl**.

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.

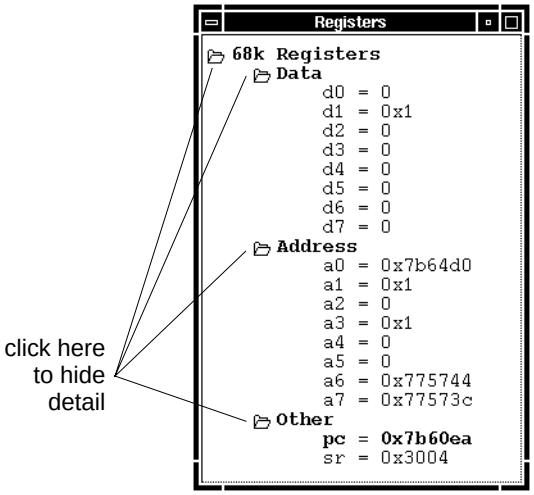
Windows>Backtrace
: Displays an auxiliary window with the current stack trace, like the one in Figure 7-7.

Figure 7-7 **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 7-8 shows a register-display window for a SPARC target. As with the register displays in task browsers (see *6.6.1 The Task Browser*, p.214), registers are grouped by category, and you can control the level of detail by clicking on the folder icons that head each category.

Figure 7-8 **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.

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.

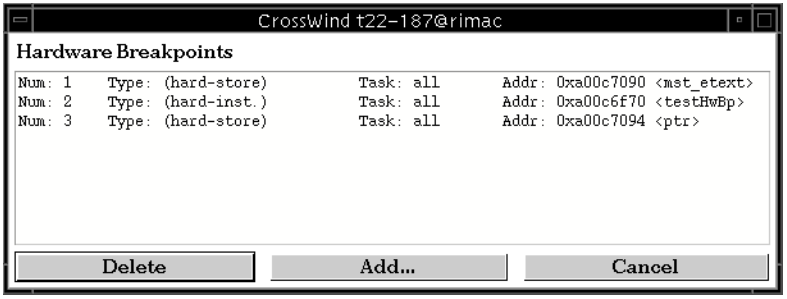
You can delete or disable temporary breakpoints in the same ways as other breakpoints—delete by right-clicking on a line displaying a breakpoint symbol, by dragging the breakpoint symbol up to the solid breakpoint symbol in the button bar, or by using the **delete** command; disable by dragging to the 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 7-9).

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]

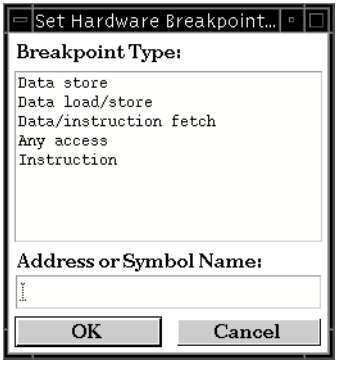
To add a hardware breakpoint, click on the Add button to display the Set Hardware Breakpoint window (Figure 7-10). If you had previously selected a symbol in the source window, the Address or Symbol Name field would be filled automatically with the symbol. Otherwise, you can enter information about where the breakpoint should be set using standard

Figure 7-9 **Hardware Breakpoints Window**



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.

Figure 7-10 **Set Hardware Breakpoint Dialog Box**



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 **~/.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.

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.

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 (such as the I960), 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 has been executed when data hardware breakpoint was hit.

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 7-11 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]

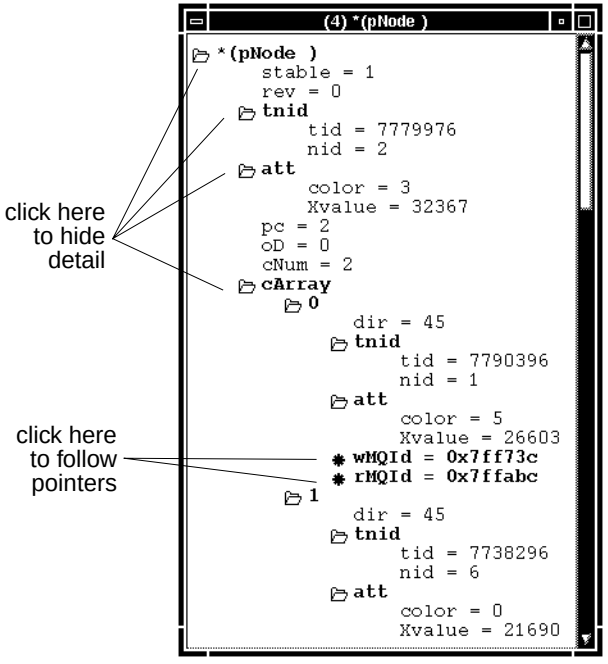
The debugger indicates whether the displayed variable is a pointer, by placing an asterisk to its left (as with **pNode** in Figure 7-11). To follow a

Figure 7-11 **Display Windows: Numeric and Pointer Data**

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 7-12 shows an example of a structure browser.

Figure 7-12 **Display Window: 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.

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.256 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 7-13 **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 **~/.wind/crosswind.tcl** initialization file.

Figure 7-13 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 **~/.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 *7.6 Tcl: Debugger Automation*, p.266; 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.275.

7.4.2 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 Systems 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 **.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 **.gdbinit**. See *7.6 Tcl: Debugger Automation*, p.266 for a discussion of extending GDB through Tcl. See also *7.7.1 Tcl: Debugger Initialization Files*, p.273 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.

---

3. The graphical interface to the debugger has a separate initialization file **.wind/crosswind.tcl**, which runs after **.gdbinit**.

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 *5.6 Object Module Load Path*, p.196 for a more extended discussion of the same problem in the context of the shell.

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

4. See also the description of **wtx-load-path-qualify** in *Extended Debugger Variables*, p.259 for another way of managing how the debugger reports **load** pathnames to the target server.

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 *5.2.6 Using the Shell for System Mode Debugging*, p.170.)

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 *7.6.3 Tcl: Invoking GDB Facilities*, p.268 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.254. 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.244).

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.238), 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.54.) 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 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:

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.

- **Shell Commands**

  You can run any of the WindSh primitive facilities described in *5.2.3 Invoking Built-In Shell Routines*, p.156 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 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 **~/.gdbinit** and **${PWD}/.gdbinit**, discussed in *7.7.1 Tcl: Debugger Initialization Files*, p.273. 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 no complete in less time than is specified here (in seconds), the debugger reports an error. Default: 30 seconds.

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.

7.5 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*.

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 *7.5.2 Thread Facilities in System Mode*, p.262) 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 *5.2.6 Using the Shell for System Mode Debugging*, p.170, then attach to them with the **thread** command.

7.5.1 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.

7.5.2 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.

7.6 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 *7.7 Tcl: CrossWind Customization*, p.273. 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.

7.6.1 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 *7.6.3 Tcl: Invoking GDB Facilities*, p.268.

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 **.wind/gdb.tcl** under your home directory. The GDB Tcl interpreter reads this file when it begins executing. (See *7.7.1 Tcl: Debugger Initialization Files*, p.273 for a discussion of how all the CrossWind and GDB initialization files interact.)

7.6.2 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.253), 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 *7.6.3 Tcl: Invoking GDB Facilities*, p.268 to call **gdb tclproc** in **~/.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 *B.2.9 Tcl Error Handling*, p.324.

7.6.3 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 *B.2.9 Tcl Error Handling*, p.324).

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.

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 *B.2.9 Tcl Error Handling*, p.324 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.

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.

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
```


7.6.4 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:

---

7. Remember, though, that for interactive exploration of a list the structure browser (Figure 7-12) described in *CrossWind Buttons*, p.244 is probably more convenient.

```
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).

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
```

7.7 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. *Tornado API Guide: UI Tcl* 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**.

7.7.1 Tcl: Debugger Initialization Files

You can write Tcl code to customize the debugger's graphical presentation in a file called **.wind/crosswind.tcl** under your home directory. 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 **~/.wind/gdb.tcl** initialization files that initialize the debugger command language (see *7.6 Tcl: Debugger Automation*, p.266).

The following outline summarizes the role of all the CrossWind customization files. The files are listed in the order in which they execute.

~/.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.

~/.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.

~/.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 **~/.wind/gdb.tcl**:

```
gdb set inhibit-gdbinit yes
```

7.7.2 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 *7.6.3 Tcl: Invoking GDB Facilities*, p.268) they are the path to debugger functionality from the graphical front end.

Most of the examples in *7.7.3 Tcl: Experimenting with CrossWind Extensions*, p.275, below, center around calls to **downtcl**.

7.7.3 Tcl: Experimenting with CrossWind Extensions

The examples in this section use the Tcl extensions summarized in Table 7-3. For detailed descriptions of these and other Tornado graphical building blocks in Tcl, see *Tornado API Guide: UI Tcl*.

Table 7-3 **Tornado UI Tcl Extensions Used in Example 7-2.**

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 7-1 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 7-1 **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 7-2 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.

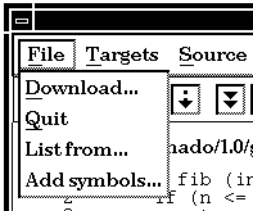
In Example 7-2, the procedure **xwindList** uses **downtcl** to run the GDB **list** command. To tie this into the graphical interface, the example adds a new command List from... to the File menu. The new command displays a form (described in the **dialogCreate** call) to collect input specifying an 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 7-14 shows the new menu command and form defined here (and the Example 7-3 menu command).

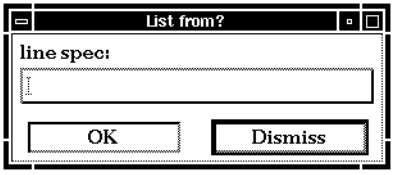
Example 7-2 **List Command**

```
# 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 7-14 **List Menu Command and Form Defined in Example 7-2**





Tcl: An Add-Symbols Command for the File Menu

As explained in *What Modules to Debug*, p.253, 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 7-3 illustrates a File menu command Add Symbols to handle this through the graphical user interface, instead of typing the **add-symbol-file** command.

Example 7-3 **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}
    }
}
```

8 Customization

8.1 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.

8.2 Setting Download Options

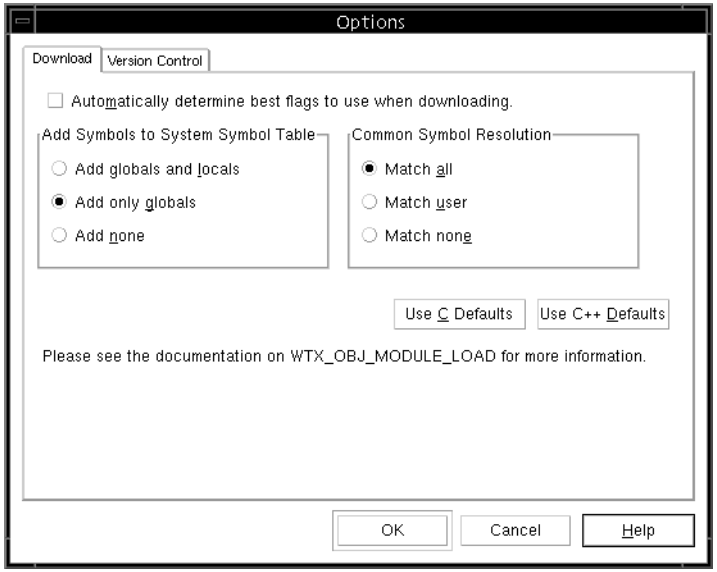
The Download page provides options for handling symbols when objects are downloaded to the target(Figure 8-1). The options are as follows:

Automatically determines best flags ...
: C and C++ object modules require different download flags. When downloading an object module to target memory, Tornado determines whether it was created from a C or C++ file, and downloads it using the appropriate flags

Add Symbols to System Symbol Table
: The symbols defined in the module being loaded may be added to the system symbol table. Choose one of the following:

  - Add globals and locals to add global and local symbols to the system symbol table.

Figure 8-1 **Download Page**



- Add only globals to add global symbols to the system symbol table.
- Add none to add no symbols to the system symbol table.

Common Symbol Resolution

Common symbols can be resolved in a variety of ways:

- Match all to allocate common symbols, but search for matching symbols in user-loaded modules and the target-system core file.
- Match user to allocate common symbols, but search for matching symbols in user-loaded modules.
- Match none to allocate common symbols, but not search for any matching symbols.

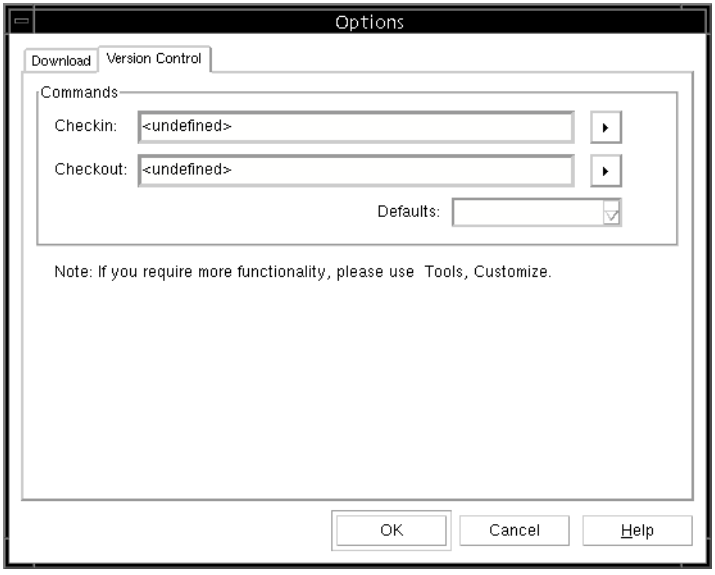
The Use Defaults buttons reset the options to their C or C++ defaults.

See Help>Manuals Contents>Tornado API Reference>WTX Protocol> WTX>WTX_OBJ_MODULE_LOAD for more information about these options.

8.3 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 8-2).

Figure 8-2 **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 8-1).

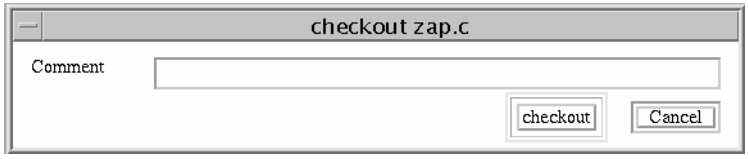
Checkout
: The Checkout text box allows you to enter the command that checks a file out of your version control system. The button at the end of the box opens a pop-

up menu which has a Browse item to help you locate the command and macros to provide the Tornado context (see Table 8-1).

Table 8-1 **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 8-3.

Figure 8-3 **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.

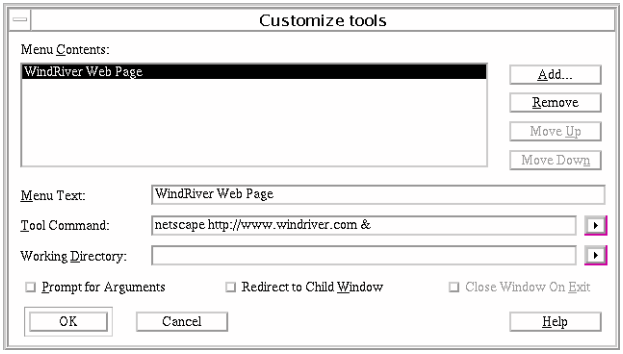
8.4 Customizing the Tools Menu

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.

8.4.1 The Customize Tools Dialog Box

Click Tools>Customize to open the Customize Tools dialog box (Figure 8-4).

Figure 8-4 **Customize Tools Dialog Box**



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.

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.

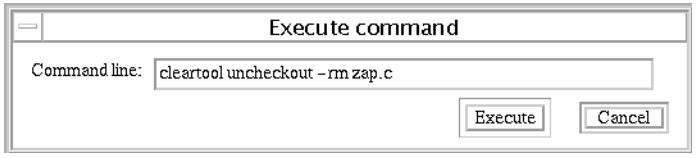
Tool Command
: Specifies the instructions to execute your command. You can place anything here that you could execute at the command prompt or in a batch file. Click the button at the right of the box to see a pop-up menu including a browse option and a list of macros which allow you to capture Tornado context in your commands. See *Macros for Customized Menu Commands*, p.285 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 8-5.)

Figure 8-5 **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 8-2 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 8-2 **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 8-3 lists macros for the project facility context.

Table 8-3 **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 8-3 **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 8-4 lists macros for the global Tornado context.

Table 8-4 **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**

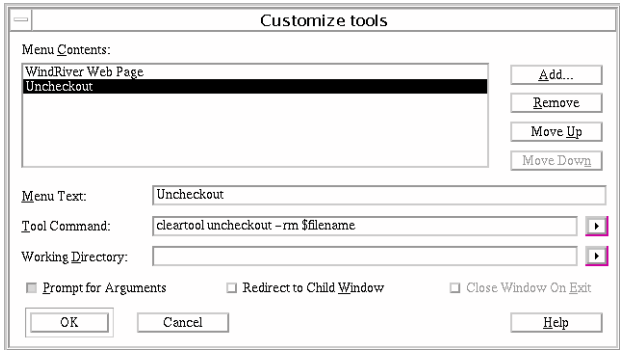
8.4.2 Examples of Tools Menu Customization

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 8-6 illustrates the specification for a ClearCase command to uncheckout a module.

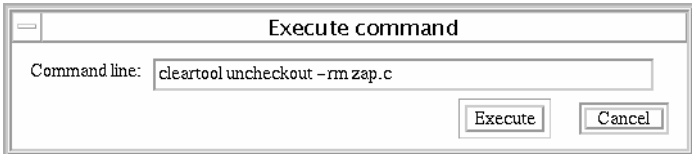
Figure 8-6 **Uncheckout Command for Tools Menu**



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.285) 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 8-7), to permit specifying other arguments if necessary.

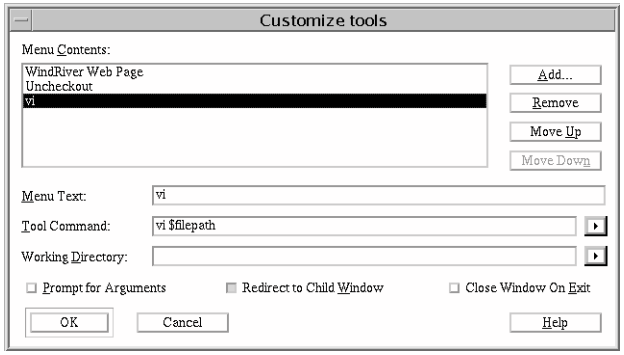
Figure 8-7 **Prompt for Arguments Dialog Box**



Alternate Editor

Figure 8-8 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 8-8 **Custom Editor Command for Tools Menu**

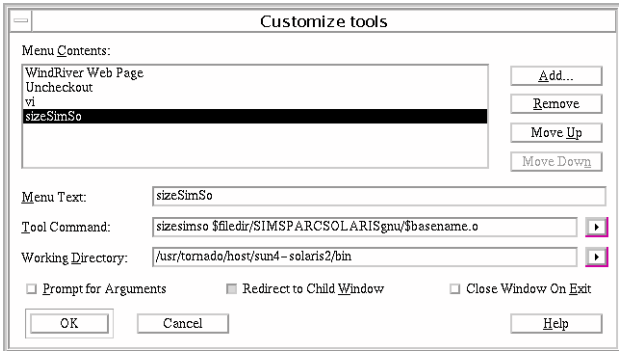


Binary Utilities

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 8-9 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.

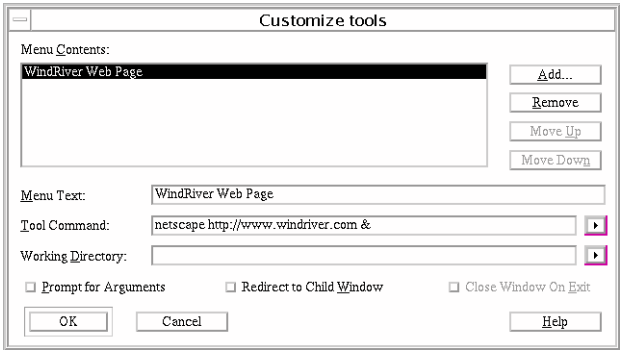
Figure 8-9 **Object-Module Size Command for Tools Menu**



World Wide Web

You can add a Tools command to link your Web browser directly to announcements from Wind River Systems (and to related Internet resources). Figure 8-10 shows the specification for a Wind River Web Page command. (For a description of the information available on the WRS home page, see *9.3 WRS Broadcasts on the World Wide Web*, p.301.

Figure 8-10 **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.

8.5 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 8-1 **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.288. You will still need to perform Step (2) in order to be able to call emacs from the menu.

8.6 Tcl Customization Files

When Tornado begins executing, it checks for initialization files of the form **.wind/***filename***.tcl** in two places: first under **usr/wind** (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 Guide* or the *Tornado Online Manuals: Tornado API*.

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"
```

9 Customer Service

9.1 Introduction

Wind River Systems is committed to meeting the needs of our customers. As part of that commitment, we provide a variety of ways of contacting our Customer Support group:

- **The Tornado Launcher**

  The launcher's Support menu provides a structured online system for requesting help with any WRS product. For more information, see *9.2 WRS Support Services*, p.294.

- **support@wrs.com**

  You can use this e-mail address directly to contact customer support if you prefer, although using the Support menu is usually helpful.

- **The World Wide Web**

  The most current information about Wind River products and services, including training courses and schedules, is always available under **http://www.wrs.com**. The launcher provides quick access to several starting points in our Web publications; see *9.3 WRS Broadcasts on the World Wide Web*, p.301.

- **1–800–872–4977 (1–800–USA–4WRS)**

  Within the North America, you can contact us with a toll-free voice telephone call. In other countries, please contact your distributor (see the back cover).

- **FAX: 1–510–814–2164**

  You can print out the structured information from the launcher's Support menu to use FAX instead of e-mail; see *9.2 WRS Support Services*, p.294.

The remainder of this chapter describes the support and information facilities built into Tornado. These facilities are helpful, but are not required: use whatever contact method you find most convenient.

NOTE: The standard Wind River Systems support contract specifies particular contact people at your organization. Please channel your support requests through them.

9.2 WRS Support Services

The Tornado launcher Support menu can be used to send support requests to Wind River Systems. All of the commands on the Support menu display a form for entering support requests. The form is the same in all cases, but it is pre-loaded with information for the Tornado component you select from the menu.

Use the menu entries for support requests as follows:

Support>WindSurf
: Access the WindSurf customer support Web page.

Support>Tornado
: Support requests on all Tornado host-resident tools other than the CrossWind debugger or the GNU tools.

Support>VxWorks
: Support requests on target-resident software.

Support>CrossWind
: Support requests on the CrossWind debugger or on the underlying command-driven debugger GDB.

Support>GNU
: Support requests on the GNU ToolKit.

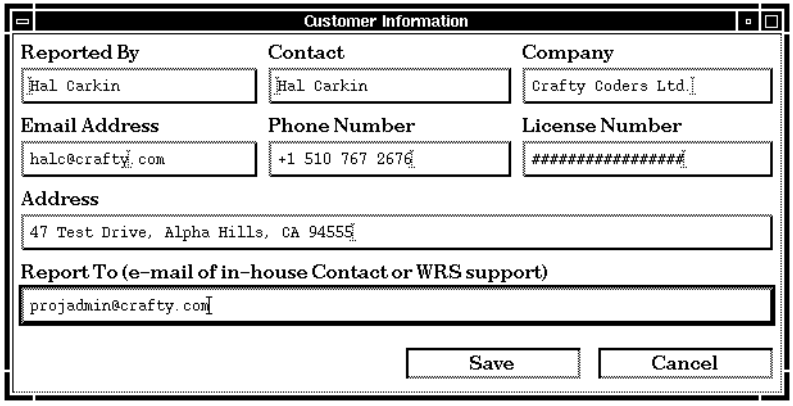
Support>Project
: Support requests on the project facility.

Support>WindView
: Support requests on the WindView software logic analyzer.

9.2.1 The Customer Information Form

The first time you select any command from the Support menu, a special auxiliary form is displayed: the Customer Information form. You must enter this information once before sending your first Technical Support Request (TSR). Thereafter, the same information is used automatically for all future TSRs you transmit.[1] You can also get to this auxiliary form by pressing the Self Info button on a Support Request form, if you need to revise it. A completed customer information form is shown in Figure 9-1 below.

Figure 9-1 **Form: Customer Information**



The following fields are available in the Customer Information form:

Reported By
: Your name. This field is filled in by default from your UNIX login information.

Contact
: The person who should receive replies to your support requests. Usually this is also your name; filled in by default from your UNIX login information.

Company
: Your company's name.

1. This information is stored in the file **.wind/profile** under your home directory.

Email Address
The electronic mail address, if available, to receive replies to support requests.

Phone Number
A telephone number where support personnel can contact you.

License Number
Your WRS license number.

Address
Your mailing address.

Report To
The default destination when you send support requests by e-mail.

NOTE: The standard Wind River Systems support contract specifies particular contact people at your organization. Support requests should go through them. If you are one of these designated contacts, leave the Report To field at the default setting; otherwise, fill in the electronic mail address of your local contact person. Taking care of this early will expedite your support requests later on.

9.2.2 Sending a Technical Support Request (TSR)

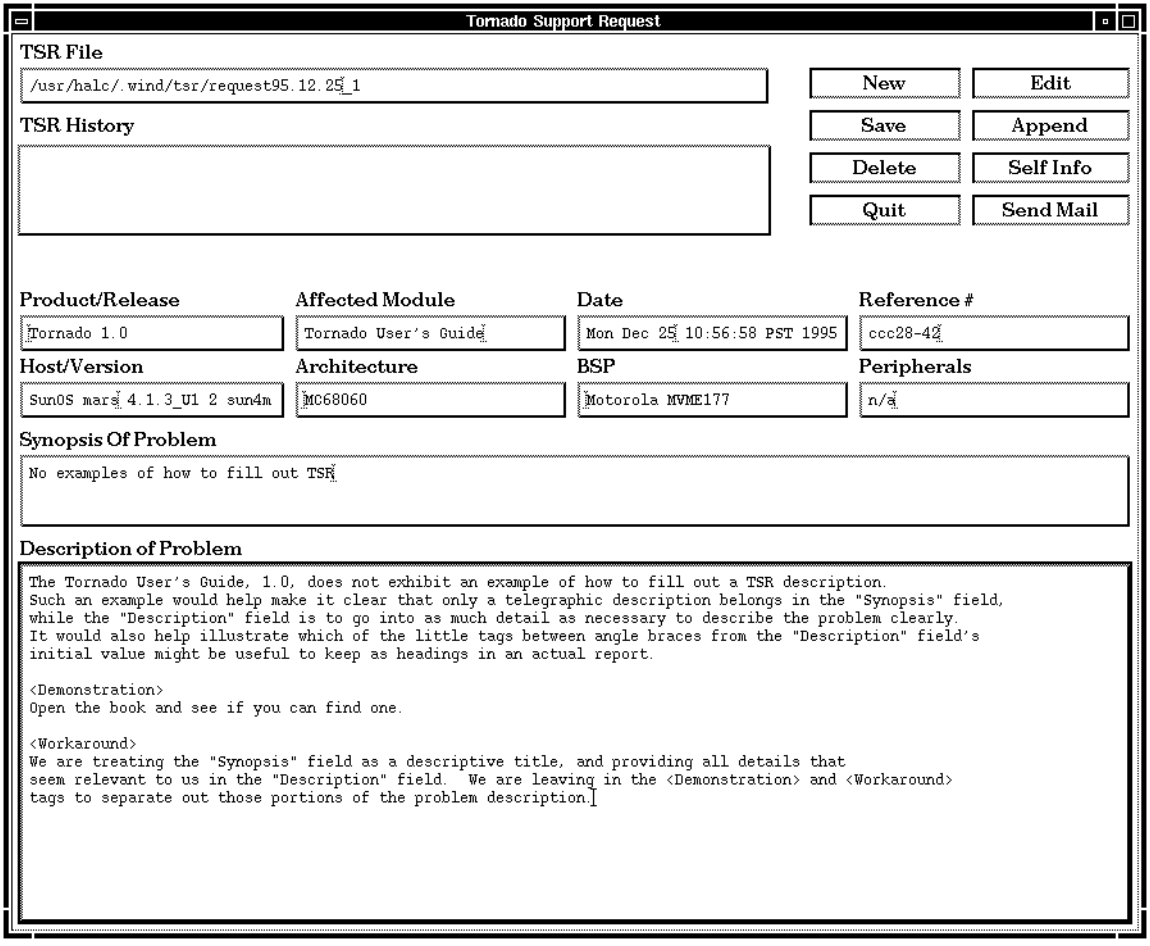
See Figure 9-2 for an example of the TSR form. You can fill out a TSR form in part, save it to investigate further, and return to it whenever you wish. Once a TSR is complete, you can send the request directly from the Support Request form by electronic mail, or print it out ready for faxing. In either case, the Tornado launcher keeps a history of all support requests you prepare; you can also use the Support Request form to review the history of your TSRs.

The Support Request form has three main components (besides the separate Customer Information form). These two components are the fields to describe the problem, in the main form shown in Figure 9-2, and the buttons to act on the problem, in the top right of the same form.

Editable Fields on the Support Request Form

The following fields appear in the TSR form. Many fields are filled out by default, but you can always overwrite any of the default field values. In all the text fields, you can type as much text as you need; use the arrow keys on your keyboard to scroll within the space available.

Figure 9-2 **Form: Support Request**



TSR File
: The file where the launcher saves this TSR for your reference. By default the launcher stores these files under **.wind/tsr/** in your home directory, and constructs the file name for each TSR from the date and a sequence number.

TSR History
: A scrollable list (the scrollbar appears when the list is longer than the space available) of previously saved TSRs. (This field is editable only indirectly, with the action buttons.) To view the information from any TSR in this list, click on

its file name: the contents of the Support Request form are filled in with the saved TSR data in response.

Product/Release
: The name and release level of the product which is the subject of this TSR. Filled in by default with the identification of the software you selected, for the host in use and to the currently selected target. If you use more than one kind of host or target, please make sure this information accurately reflects the environment where the problem occurred.

Affected Module
: If you can identify the failing product component (for example, if the bug appears only when you include one particular run-time feature), specify it here.

Date
: Automatically filled in with the date and time when you first opened this TSR.

Reference #
: A place for any reference number your own organization may find useful to associate with this TSR.

Host/Version
: The host operating system and architecture where the launcher is running. This is filled in by default to the values for the current session. If you use Tornado on more than one host, please make sure before sending the support request that the value in this field really reflects the environment where you saw the problem.

Architecture
: Target architecture. The name of the selected target's architecture is filled in by default. If you use more than one target architecture, please make sure this information is accurate for your problem.

BSP
: The name and version number of the Board Support Package linked into your target run-time. Filled in by default with data for the selected target.

Peripherals
: List any other hardware that might be involved in your problem.

Synopsis of Problem
: A terse line to identify the problem. It is to your advantage to make this line as informative as possible, because mail about this problem from the WRS support staff uses this line as its subject. Use keywords that allow you to recognize the problem.

Description of Problem

Describe your problem here, in as much detail as possible. The more information you supply, the faster a Technical Support Engineer at Wind River Systems can solve the problem. If you can exhibit specific steps that cause the problem, please include them after the tag <Demonstration>. If you have developed a work-around to the problem, that may also be a useful clue; please describe it after the tag <Workaround>.

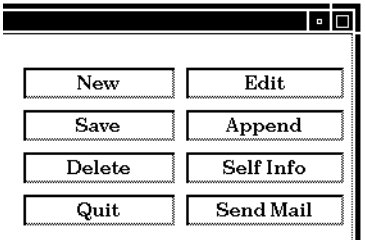
You can type directly into this field, or use the Edit button (top right on the Support Request form) to bring up the editor specified by your **EDITOR** environment variable to fill out the field. If you bring up an auxiliary editor, the Support Request form becomes inactive until you exit the editor.

The Append button (immediately below Edit) allows you to include any saved information you may have already saved that is relevant to this problem. Append brings up a dialog box that allows you to specify the name of a file, and adds the contents of that file to the end of your problem description.

Taking Action on a TSR

The buttons at the top right of the main Support Request form allow you to take a range of actions with the information you enter on the form. Figure 9-3 shows the buttons available.

Figure 9-3 **Buttons on the Support Request Form**



The following paragraphs describe the effect of each button:

New

Begin a new TSR; clear all editable fields to their default values. You might use this button to start over, or to begin entering a TSR after reviewing saved ones.

Save

Save the information currently specified (without transmitting a TSR).

Delete
: Delete the currently displayed TSR from your saved-TSR history.

Quit
: Abandon this form immediately; discard any changes entered since the last save.

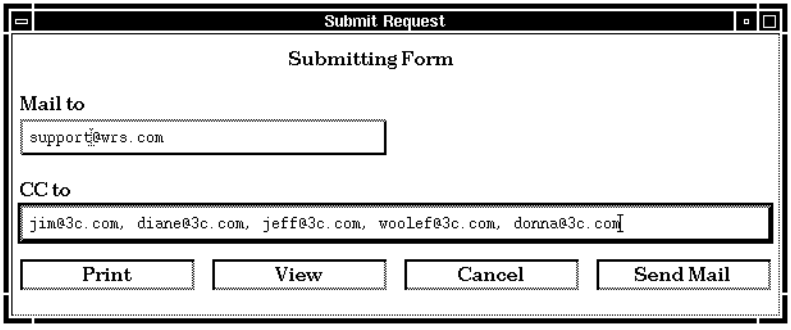
Edit
: Bring up the editor specified by your **EDITOR** environment variable on the contents of the Description of Problem field.

Append
: Add the contents of another file to the end of the Description of Problem field. (This button brings up an auxiliary dialog box to specify the file name.)

Self Info
: Bring up the Customer Information form shown in Figure 9-1.

Send Mail
: Save the current support request, then send the support request through e-mail or print the trouble report. This button brings up the dialog shown in Figure 9-4.

Figure 9-4 **TSR Submission Dialog**



If you have access to Internet mail, press the Send Mail button on this dialog box to transmit the support request. Fill in the CC to field as desired; the Mail to field comes up pre-loaded with the address specified in your saved customer information. The Print button sends a printout of the TSR, formatted in the same way as for electronic mail, to your default printer,[2] and abandons the

2. You can change your default printer by defining the **PRINTER** environment variable.

dialog. Use this button to produce a printout for faxing, if you cannot use Internet mail. The View button displays the TSR in an auxiliary window, formatted in the same way as for electronic mail. The Cancel button abandons this dialog without transmitting the TSR.

9.3 WRS Broadcasts on the World Wide Web

The Tornado launcher Info menu links your system Web browser directly to several alternative starting points in the Wind River Systems Web publications and related Internet resources.

Tornado does not itself include a Web browser (any more than it includes an editor). If your site does not have a Web browser, ignore this menu—it only provides convenient access to information, not any essential functionality.

Info>Products
: Provides information about ordering or inquiring about Wind River Systems products.

Info>Training
: Describes WRS training services, schedules, and prices.

Info>Announcements
: Announcements from WRS about current and upcoming events.

Info>User's Group
: Information about the Wind River Users Group, including links to the public software archive, instructions for joining the mailing list "exploder," and information about annual meetings.

Info>www.wrs.com
: The Wind River Systems home page.

Info>comp.os.vxworks
: The newsgroup devoted to VxWorks and its development tools. Although this is not a Wind River Systems publication, it is a valuable communications mechanism for VxWorks developers.

Appendices

A Directories and Files

A.1 Introduction

All Wind River Systems products are designed to install in a single coordinated directory tree. The directory root is shown as **$WIND_BASE** 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 **$WIND_BASE**

Directory/File	Description
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*.
SETUP	Directory of SETUP program.
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*.
.wind	Directory for customization files and files that capture Tornado application state. Described in *.A.4 Initialization and State-Information Files*

Table A-1 **$WIND_BASE** *(Continued)*

Directory/File	Description
LICENSE.TXT	Text of Wind River Systems 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.
README.HTML	Last-minute information about the current Tornado release. This is an HTML file that you can examine with any Web browser.
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 **$WIND_BASE/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*/**lib**	Directory containing both static libraries (**lib***nam***e.a**) and shared libraries (*name***.sl**) for the interactive Tornado tools.
host-*os*/**lib/backend**	Directory containing shared libraries implementing the communications back ends available to the Tornado target server.

Table A-2 **$WIND_BASE/host** *(Continued)*

Directory/File	Description
host-os/**lib/gcc-lib**	Directory containing the separate programs called for the passes of the GNU compiler.
host-os/*x*-**wrs-vxworks**	Directory containing component programs and libraries for the GNU ToolKit configured for architecture *x*.
host-os/**man**	Directory containing reference entries for the programs in the GNU ToolKit.
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/loader	Directory containing object-module format information for the Tornado dynamic linker.
resource/tcl	Directory containing Tcl source code for the Tornado tools.
resource/tcl/app-config	Directory of tool-specific directories containing dialog (form) descriptions and other subsidiary definitions for each Tornado tool. All Tcl files present in a tool-specific subdirectory are read when the corresponding tool begins executing.
resource/test	Directory containing tests for the WTX protocol.
resource/userlock	Global authorization file for Tornado users.
resource/wdb	Mappings for WDB protocol extensions.
src/demo	Directory containing source code for the host-resident portion of the VxColor demonstration program.
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 **$WIND_BASE/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).
config/all		Subdirectory containing system configuration modules. *Note that this method of configuration has been replaced by the project facility* (see *4. Projects*). The directory includes the following files:
	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*		The other subdirectories of **config** contain system-dependent modules for each port of VxWorks to a particular target CPU. Each of these directories includes the following files:
	Makefile	Makefile for creating boot ROMs and the VxWorks system image for a particular target.
	sysLib.c, **sysALib.s**	Two source modules of system-dependent routines.
	tyCoDrv.c	A driver for on-board serial ports.

Table A-3 **$WIND_BASE/target** *(Continued)*

Directory	File	Description
	config.h	Header file of hardware configuration parameters.
	bspname.**h**	Header file for the target board.
	romInit.s	Assembly language source for initialization code that is the entry point for the VxWorks boot ROMs and ROM-based versions of VxWorks.
	vxWorks, **vxWorks.sym**	Complete, linked VxWorks system binary (**vxWorks**), and its symbol table (**vxWorks.sym**) created with the supplied configuration files.
	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.
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/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.
h/netinet		Directory containing Internet-specific header files.

Table A-3 **$WIND_BASE/target** *(Continued)*

Directory	File	Description
h/private		Directory containing header files for code private to VxWorks.
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/types		Directory containing header files used for defining types.
lib		Directory containing the machine-independent object libraries and modules provided by VxWorks.
lib/obj*cputool***vx**		A directory containing VxWorks object modules as individual files (suitable for loading dynamically to the target).
lib/lib*cputool***vx.a**		An archive (**ar**) format library containing the object modules that make up VxWorks.
src		Directory containing all source files for VxWorks.
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.
	ctypeInit.c	Used to include the *ctype* ANSI C library routines.
	localeInit.c	Used to include the *locale* ANSI C library routines.
	mathInit.c	Used to include the *math* ANSI C library routines.
	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.

Table A-3 **$WIND_BASE/target** *(Continued)*

Directory	File	Description
	usrDepend.c	Used to check module dependences for constants defined in **configAll.h** and **config.h**.
	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).
	usrKernel.c	Used to configure and initialize the *wind* kernel.
	usrIde.c	Used to mount a dosFs file system on a boot IDE hard disk drive (i386/i486 targets only).
	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.
	usrScript.c	Used to execute a startup script when VxWorks first boots.
	usrScsi.c	Used to configure and initialize SCSI support.
	usrSmObj.c	Used to configure and initialize shared memory object support.
	usrWdb.c	Used to configure and initialize the Tornado target agent.
	cplusHeap.c, **cplusIos.c**, **cplusTools.c**	Used to configure Rogue Wave C++ utility classes (from optional product: Wind Foundation Classes).
	cplusVxw.c	Used to configure VxWorks wrapper classes (from optional product: Wind Foundation Classes).
	usrCplusTools.c	Used to initialize Rogue Wave C++ utility classes (from optional product: Wind Foundation Classes).
	usrCplusVxw.c	Used to initialize VxWorks wrapper classes (from optional product: Wind Foundation Classes).

Table A-3 **$WIND_BASE/target** *(Continued)*

Directory	File	Description
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/dg		Directory containing a simple datagram facility, useful for demonstrating and testing datagrams on VxWorks and/or other TCP/IP systems.
src/demo/color		Directory containing the VxColor example application used in *2. A Whirlwind Tour*.
src/demo/start		Directory containing the program used with the *Tornado Getting Started Guide* tutorial.
src/drv		Directory containing source code for supported board device drivers. Not all of the subdirectories shown are present for all BSPs.
src/usr		Directory containing user-modifiable code.
usrLib.c		Library of routines designed for interactive invocation, which can be modified or extended if desired.
statTbl.c		Source of the error status table. It contains a symbol table of the names and values of all error status codes in VxWorks. This table is used by the routine ***printErrno*( )** for translating error status codes into meaningful messages.
memDrv.c		Pseudo-device driver that allows memory to be accessed as a VxWorks character (non-block) device.
ramDrv.c		Block device driver that allows memory to be used as a device with VxWorks local file systems.

The following directories are included only with a VxWorks source license.

Table A-3 **$WIND_BASE/target** *(Continued)*

Directory	File	Description
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.
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/wdb		Directory containing source code for the Tornado target agent.
src/wind		Directory containing source code for the VxWorks kernel.
unsupported		Directory containing miscellaneous unsupported code such as public domain software, examples, host tools, obsolete BSPs, and BOOTP server.

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. See *6.12 Tcl: the Browser Initialization File*.
crosswind.tcl	Optional Tcl initialization code for the debugger front end. See *7.8 Tcl: CrossWind Customization*.
gdb.tcl	Optional Tcl initialization code for the debugging engine itself. See *7.8 Tcl: CrossWind Customization*.
launch.tcl	Optional Tcl initialization code for the launcher. See *4.7 Tcl: Customizing the Launcher*.
windsh.tcl	Optional Tcl initialization code for the shell. See *5.11.3 Tcl: Tornado Shell Initialization*.
wtxtcl.tcl	Optional Tcl initialization code for **wtxtcl**, the Tcl interpreter with WTX-protocol extensions. See the *Tornado API 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. See *4.5 Managing Target Servers*.
profile	A file of identification information used for your Tornado support requests. This information is collected and updated through the launcher's **Support** menu; see *10.2.1 The Customer Information Form*.
tgtsvr	A directory collecting your saved target-server configurations. Target-server configurations are defined and viewed through the launcher's **Target** menu; see *4.5 Managing Target Servers*. The file **.lastTgtSvr** in this directory records the name of the last target server you launched.

Table A-5 **.wind State-Information Files** *(Continued)*

Directory/Files	Description
tsr	A directory recording the history of your Tornado support requests. This information is managed through the launcher's **Support** menu; see *10.2 WRS Support Services*.

B Tcl

B.1 Why Tcl?

Much of the Tornado implementation is written in Tcl (the *tool command language* designed by John Ousterhout). Readers who are already familiar with Tcl applications are not likely to find this surprising.

However, if Tcl is new to you, you may be wondering why. Choosing Tcl as the implementation vehicle has the following benefits:

- **Customization:** Tornado can be customized to an unprecedented degree. All tools can be conditioned with Tcl scripts. At a deeper level, the Tcl code for the tool itself is available for your inspection. This allows you to more easily write your own Tcl code to modify any features you wish to change.

CAUTION: When you customize Tornado tools, write your changes as separate files that override the original tools. That way, Tornado WRS technical support can still help you, if the need arises; and it will be easier to preserve your enhancements over new releases of Tornado.

- **Development speed and robustness:** Because development in Tcl is interactive, graphical-tool design can include much more experimentation in the development cycle. This means we at WRS can build products faster, and we can build them better, checking our results as we go. Third-party developers experience exactly the same benefits. All of this means that you, the VxWorks application developer, have more and better tools available to choose from.
- **Ease of maintenance:** Because Tcl code is ordinary text, optional products and third-party add-ons can integrate themselves into a Tornado installation by

including Tcl files that customize the launcher or other components. If necessary, WRS Technical Support can also send out field-installable patches in electronic mail.

- **Portability:** Implementing the graphical user-interface building blocks as Tcl extensions makes it possible for WRS to support more kinds of host platforms more quickly, because the transition between windowing systems (otherwise often difficult) is encapsulated into a series of well defined internal calls.

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.

B.2 A Taste of 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 Guide* from Wind River Systems.

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.

B.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 B-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 B-2 **Evaluating Tcl Variables**

Tcl Expression	Result
`set num`	`6`
`set z`	`hello world`

B.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 B-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.
`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 *B.2.4 Command Substitution*, p.321), can provide a little of the flavor of Lisp in Tcl programs.

B.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 B-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`	

B.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 B-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 *B.2.2 Lists in Tcl*, p.320). 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.

B.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 B-6 **Arithmetic in Tcl**

Tcl Expression	Result
`expr (2 << 2) + 3`	`11`
`expr 0xff00 & 0xf00`	`3840`

B.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 following examples illustrate some of the possibilities:

Table B-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`

B.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 B-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.
        }
}
```

B.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 in the introduction to *B.2 A Taste of Tcl*, p.318.

B.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.

B.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 in the introduction to *B.2 A Taste of Tcl*, p.318.

B.3 Tcl Coding Conventions

This section defines the Wind River Systems standard for all C and Tcl 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.

These conventions are divided into the following categories:

- Module Layout
- Procedure Layout
- Code outside of procedure
- Code Layout
- Naming Conventions
- Style

B.3.1 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 B-1 **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
```

B.3.2 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 *VxWorks Programmer's Guide: Coding Conventions*.

The following is an example of a standard procedure layout.

Example B-2 **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} {
    ...
}
```

B.3.3 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 B-3 **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
```

B.3.4 Declaration Formats

Include only one declaration per line. Declarations are indented in accordance with *Indentation*, p.332, 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

    ...
}
```

B.3.5 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.332).

  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 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.333).
- 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
```

B.3.6 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

B.3.7 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)"
}
```

C

Tornado Tools Reference

browser – the Tornado browser (UNIX) 343
crosswind – invoke the CrossWind executable, after establishing environment (UNIX) 346
evtRecv – receive WindView event data (UNIX) 348
launch – the Tornado launcher 350
tgtsvr – the target board server 355
vxColor – graph coloring demo for Tornado 364
windsh – The Tornado Shell 368
***agentModeShow*( )** – show the agent mode (*) 378
***b*( )** – set or display breakpoints 378
***bd*( )** – delete a breakpoint 379
***bdall*( )** – delete all breakpoints 379
***bh*( )** – set a hardware breakpoint 380
***bootChange*( )** – change the boot line 380
***browse*( )** – send a message to the browser asking it to browse an address (*) 381
***c*( )** – continue from a breakpoint 382
***cd*( )** – change the default directory 382
***checkStack*( )** – print a summary of each task's stack usage 383
***classShow*( )** – show information about a class of objects (*) 384
***cplusCtors*( )** – call static constructors (C++) 385
***cplusDtors*( )** – call static destructors (C++) 385
***cplusStratShow*( )** – show C++ static constructors calling strategy (*) 386
***cplusXtorSet*( )** – change C++ static constructor calling strategy (C++) 387
***cret*( )** – continue until the current subroutine returns 387
***d*( )** – display memory 388
***devs*( )** – list all system-known devices 389
***h*( )** – display or set the size of shell history 389
***help*( )** – print a synopsis of selected routines 389
***hostShow*( )** – display the host table 391
***i*( )** – print a summary of each task's TCB, task by task 391

icmpstatShow( ) – display statistics for ICMP 392
ifShow( ) – display the attached network interfaces 392
inetstatShow( ) – display all active connections for Internet protocol sockets 393
intVecShow( ) – display the interrupt vector table 393
iosDevShow( ) – display the list of devices in the system 394
iosDrvShow( ) – display a list of system drivers 394
iosFdShow( ) – display a list of file descriptor names in the system 394
ipstatShow( ) – display IP statistics 395
iStrict( ) – print a summary of all task TCBs, as an atomic snapshot (*) 395
l( ) – disassemble and display a specified number of instructions 396
ld( ) – load an object module into memory 396
lkAddr( ) – list symbols whose values are near a specified value 397
lkup( ) – list symbols 398
ls( ) – list the contents of a directory 398
m( ) – modify memory 399
memPartShow( ) – show partition blocks and statistics 400
memShow( ) – show system memory partition blocks and statistics 400
moduleIdFigure( ) – figure out module ID, given name or number (*) 401
moduleShow( ) – show the current status for all the loaded modules 402
mqPxShow( ) – show information about a POSIX message queue (*) 402
mRegs( ) – modify registers 403
msgQShow( ) – show information about a message queue 404
period( ) – spawn a task to call a function periodically 405
printErrno( ) – print the definition of a specified error status value 405
printLogo( ) – display the Tornado logo 406
pwd( ) – display the current default directory 406
quit( ) – shut down WindSh (*) 406
reboot( ) – reset network devices and transfer control to boot ROMs 407
repeat( ) – spawn a task to call a function repeatedly 407
routestatShow( ) – display routing statistics 408
s( ) – single-step a task 408
semPxShow( ) – show information about a POSIX semaphore (*) 409
semShow( ) – show information about a semaphore 410
shellHistory( ) – display or set the size of shell history 410
shellPromptSet( ) – change the shell prompt 411
show( ) – display information on a specified object 411
smMemPartShow( ) – show user shared memory system partition blocks and statistics (*) 412
smMemShow( ) – show the shared memory system partition blocks and statistics 412
so( ) – single-step, but step over a subroutine 413
sp( ) – spawn a task with default parameters 414
sps( ) – spawn a task with default parameters, and leave it suspended (*) 415
sysResume( ) – reset the agent to tasking mode (*) 415
sysStatusShow( ) – show system context status (*) 416
sysSuspend( ) – set the agent to external mode and suspend the system (*) 416
taskCreateHookShow( ) – show the list of task create routines 417

taskDeleteHookShow( ) – show the list of task delete routines 417
taskIdDefault( ) – set the default task ID 417
taskIdFigure( ) – figure out the task ID of a specified task (*) 418
taskRegsShow( ) – display the contents of a task's registers 419
taskShow( ) – display task information from TCBs 419
taskSwitchHookShow( ) – show the list of task switch routines 420
taskWaitShow( ) – show information about the object a task is pended on (*) 420
tcpstatShow( ) – display all statistics for the TCP protocol 421
td( ) – delete a task 422
tftpInfoShow( ) – get TFTP status information 422
ti( ) – display complete information from a task's TCB 423
tr( ) – resume a task 423
ts( ) – suspend a task 424
tt( ) – display a stack trace of a task 424
tw( ) – print info about the object the given task is pending on (*) 425
udpstatShow( ) – display statistics for the UDP protocol 425
unld( ) – unload an object module by specifying a file name or module ID 426
version( ) – print VxWorks version information 426
w( ) – print a summary of each task's pending information, task by task (*) 427
wdShow( ) – show information about a watchdog 427
wtxCtest – test suite for the WTX C API 428
wtxreg – report information about target servers known to a registry 428
wtxregd – the Tornado service registry 428
wtxtcl – the Tornado Tcl shell 432
r – Tcl script, wtxtcl test suite launcher 434
configUlipSolaris – User-Level IP (ULIP) start/stop script 442
elfToBsd – convert MIPS ELF object modules to BSD a.out format 444
elfXsyms – extract the symbol table from an ELF file 445
extractBsp – extract a BSP from a VxWorks installation 445
installUlipSolaris – install ULIP files on Solaris host 452
wind_host_type – return the host type in the Tornado format 465
windman – UITclSh GUI for quick help retrieving 465
xlinkHppa – fix debug info in a partially linked HP-PA SOM COFF object module 466
xsymHppa – extract the symbol table from an HP-PA SOM COFF object module 469

Typographic Conventions

The reference entries in this Appendix use the following special characters to present information concisely:

[…] – Square brackets enclose optional parts of the command line. (Used in *Synopsis* sections only.)

. – A period within a command-line option name shows how to abbreviate the option. For example, **-f.ormat** means that you can use either **-f** or **-format**, with the same effect. (*Synopsis* sections only.)

| – Alternation; two alternative options or parameters are shown with this separator when you may choose one or the other. For example, **-N | -Nosyms** indicates that **-N** and **-Nosyms** are alternative spellings of the same option. (*Options* sections only.)

browser

NAME

browser – the Tornado browser (UNIX)

SYNOPSIS

```
browser [-T.clmode] [-v.ersion] [-h.elp] serverIdentifier
        [Xt_Standard_Command_line_Parameters]
```

DESCRIPTION

The Tornado browser is a graphical tool that regularly fetches and displays information about the target to which it is connected. When started, it connects itself to the specified target server (see **tgtsvr**).

The browser can be started from the launcher (see **launch**).

Menu bar

The menu bar contains three menu buttons. The File button provides access to the Quit item, which terminates the browser. The About button provides access to the Tornado item, which launches an information window about the current version of the Tornado development environment. The Help button provides access to two menu items: On Browser and Manuals. Both of these items launch the **windman** tool. The first item displays the browser manual pages, while the second item displays the home page for the Tornado Online Manuals.

Button bar

The button bar contains five buttons:

Immediate-update button (!)

Periodic-update button (clock)
: The browser can be updated in two modes. The first is the *per-request* mode: the browser updates its windows each time the immediate-update button is pressed. The second is the *intervals* mode and is selected when the periodic-update button is pressed. Use the parameter-adjustment button (described below) to specify the update interval. Note that a second click on this button will stop the automatic update. These actions affect all the windows associated to the browser.

Stack-check button (stacked plates)
: More information about the target activity is available via the stack-check button and the spy panel. Press the stack-check button to get the spy panel. This panel summarizes the current and maximum stack usage for each task currently running.

Spy button (chart)
: Pressing the spy button starts the spy facility on the target. Results of the survey are regularly sent to a dedicated window that shows the CPU usage per task. Use the parameter-adjustment button (described below) to configure the spy facility. Note that the spy report is stopped if the spy button is pressed again.

Parameter-adjustment button (tools)
: The browser configuration panel is called when the parameter-adjustment button is pressed. This configuration panel lets you choose between various modes: the symbol

may be sorted by name or by address, and spy may be run in cumulative or differential modes. Also, spy report time and frequency, and the browser update interval may be specified here.

State indicators (under the button bar) keep track of the browser configuration.

Hierarchical Windows

The browser's windows are organized following a hierarchical model much like directories in a file system. It is possible to hide or display groups of data selectively by closing or opening the groups in which they belong. Each group is identified by a title preceded by a folder-like icon. This icon groups related information and is open by default. A mouse click on this icon closes it, thus hiding the information in groups. Because the information model is hierarchical, closing the top-level folder icon hides all the underlying groups.

Task Information

Tasks are displayed in two lists: one for system tasks (WRS tasks), the other for user tasks. These lists can be closed or opened by clicking the folder icons. A synopsis of each task is provided, including the task ID, the task name, and the task status.

Clicking on the task ID in either task list displays a pop-up window holding data about the selected task, such as its attributes, stack, and registers.

Show Facility

The browser can display detailed information about objects managed by the remote target system. To use this feature, type or copy an object's name or ID into the text window, and then press the Show button. If the object type is known, a pop-up window holding various details about the object is displayed.

Memory Utilization

The browser displays two bar graphs that represent the amount of allocated memory in the target agent's memory pool and in the target runtime's memory pool.

Clicking in the lower bar graph displays a specialized browser that gives detailed information about the target memory usage.

Loaded Modules

The lowest part of the browser window gives information about loaded modules: module ID, module name, size of text segment, size of data segment, and size of bss segment. The total size for all text, data, and bss segments is also displayed.

If more precise information about a module is required, clicking on the module name or ID will display a hierarchical window that holds data such as load options, module format, module type, module address, module segment size, and a list of the published module symbols. Use the parameter-adjustment button to specify how to sort symbols.

OPTIONS

The browser accepts the X Toolkit Intrinsics standard command-line parameters such **-display** or **-iconic** plus the following dedicated options:

-T | -Tclmode
Run a Tcl interpreter on standard input. This is useful for adding and debugging new

features.

-v | -version
Print the current version of the Tornado development environment.

-h | -help
Print the help message.

ENVIRONMENT VARIABLES

WIND_BASE
the root location of the Tornado tree.

WIND_REGISTRY
the host on which the Tornado Registry daemon runs (see **wtxregd**).

FILES

The following resource files are required by the browser:

$WIND_BASE/host/resource/tcl/Browser.tcl
Tcl implementation of the browser's interface.

$WIND_BASE/host/resource/tcl/shelbrws.tcl
Tcl implementation of the browser's routines.

$WIND_BASE/host/resource/tcl/app-config/Browser/*.tcl
Tcl implementation of the browser's additional features.

$WIND_BASE/host/resource/tcl/app-config/all/host.tcl
Tcl implementation of the host dependencies.

$WIND_BASE/host/resource/tcl/app-config/all/host_type.tcl
Tcl implementation of the host specifics.

The following resource file is applied if present:

$HOME/.wind/browser.tcl
Tcl implementation of the user's facilities for the browser.

SEE ALSO

wtxregd, **tgtsvr**, **launch**, *Tornado User's Guide*

crosswind

NAME **crosswind** – invoke the CrossWind executable, after establishing environment (UNIX)

SYNOPSIS `crosswind [-t architecture] [-l] [gdb_options] [X toolkit options]...`

DESCRIPTION CrossWind is a graphical presentation of the GDB debugger designed for use with the Tornado environment. It can be started from the Tornado launcher; when this is done, CrossWind automatically connects to the selected target and waits for debugging commands. When started at the command line, you must issue a "target wtx" command to connect the debugger to a particular Tornado target.

File menu:

Download...
Presents a file selection dialog. The selected file is downloaded to the target.

Quit
Exits the debugger.

Targets menu:

Attach Task...
Presents a dialog of tasks that can be attached to. This can be used any time after a target is attached. The selected task will be interrupted, and the stopping point will be displayed in the source window.

Detach Task:
Detaches the attached task.

Kill Task:
Kills the attached task.

The Targets menu also contains a list of the target servers that were running at the time CrossWind was stared. These can be used to conveniently select a target to attach to.

Source menu:

C Requests that code be displayed in C source form when target execution stops, if possible.

Assembly
Requests that pure assembly code be displayed when target execution stops.

Mixed
Requests that an interleaved display of C source and assembly instructions be shown when the program stops.

Tcl menu:

Reread Home
Rereads the user's home-directory Tcl configuration file (**~/.wind/crosswind.tcl**).

Reread All
Rereads both the system Tcl initialization file

(**$WIND_BASE/host/resource/tcl/CrossWind.tcl**) and the user's home directory configuration file.

Windows menu:

Registers
: Displays a window containing the contents of the processor registers. This window is updated each time execution stops and GDB is ready to execute another command.

Backtrace
: Displays a window showing the call stack. Clicking on a level in the call stack will display the source (or assembly, depending on the display mode) associated with that level of execution.

About menu:

Tornado
: Displays a window displaying Tornado version information.

Help menu:

On CrossWind
: Displays on-line hypertext help for CrossWind.

Manuals
: Displays on-line hypertext Tornado reference manuals.

GDB Online
: An interface to the GDB native help facility. A dialog appears asking for a topic; clicking on OK will present a window with GDB on line help text.

OPTIONS

CrossWind accepts the X Toolkit Intrinsics standard command-line parameters such **-display** or **-iconic** plus the following dedicated options:

-t ***arch***
: specify the architecture to debug. Valid values are **m68k**, **sparc**, **i86**, **i960**, **simso**, **hppa**, **ppc**, **mips**, **mipsle**, and **arm**.

-l run the debugger in line-oriented mode rather than with the X GUI.

gdb_options
: any GDB options (use the options **-l --help** to see a list).

-T | -Tclmode
: Run a Tcl interpreter on standard input. This is useful for adding and debugging new features.

-v | -version
: Print the current version of the Tornado development environment.

-h | -help
: Print the help message.

ENVIRONMENT VARIABLES

WIND_BASE
the root location of the Tornado tree.

WIND_REGISTRY
the host on which the Tornado Registry daemon runs (see **wtxregd**).

FILES The following resource files are required by CrossWind:

$WIND_BASE/host/resource/tcl/CrossWind.tcl
Tcl implementation of the debugger's interface.

$WIND_BASE/host/resource/tcl/app-config/CrossWind/*.tcl
Tcl implementation of the debugger's additional features.

$WIND_BASE/host/resource/tcl/app-config/all/host.tcl
Tcl implementation of the host dependencies.

$WIND_BASE/host/resource/tcl/app-config/all/host_type.tcl
Tcl implementation of the host specifics.

The following resource file is applied if present:

$HOME/.wind/crosswind.tcl
Tcl user initialization file.

SEE ALSO **wtxregd**, **tgtsvr**, **launch**, *Tornado User's Guide*

evtRecv

NAME **evtRecv** – receive WindView event data (UNIX)

SYNOPSIS

```
evtRecv [-o file] [-p port] [-f] [-q] [-r]
```

DESCRIPTION This tool receives event data from an instrumented VxWorks target system and saves the data in a file on the host. This file can then be loaded into WindView by selecting the **Analyze** item from the **File** menu of the main WindView window.

The evtRecv tool must be started on the host system prior to enabling event logging on the target system from which the event data is to be captured. When invoked, evtRecv binds to a TCP port on the host (by default port 6164, but this can be overridden via the **-p** option), and awaits a connection from the target system. The routine ***wvEvtLogEnable*( )** is then invoked on the target (either from the VxWorks shell by the user, or from within application code); this establishes a connection with the evtRecv process on the host, and initiates the transfer of event data. Under normal circumstances, the connection is closed by a subsequent invocation of ***wvEvtLogDisable*( )**. If, however, the target application

generates event data at a rate that exceeds the network bandwidth, the target system will terminate the event transfer automatically, and evtRecv will print a message warning of target event buffer overflow. In either case, unless the **-r** option has been specified (see below), the evtRecv process on the host will exit when the logging session has been terminated.

OPTIONS

-o *file*
store the event data in *file*. If *file* does not end in **.wvr**, evtRecv appends this suffix automatically. If *file* is **-**, the event data is written to standard output, and repetitive receive mode is silently disabled, even if **-r** is specified. If **-o** is not present, **eventLog.wvr** is used by default. (See also the description of the **-r** option below.)

-p *port*
listen for connections on port, instead of the default port 6164. Note that it might be necessary to set the port number on the target system as well; use the VxWorks routine ***wvHostInfoShow*( )** to display the current port number setting on the target, and ***wvHostInfoInit*( )** to change the port number.

-f overwrite the output event data file if it already exists.

-q quiet mode; do not print informational messages.

-r repetitive receive mode. When this option is specified, **evtRecv** does not exit when the event data transfer is complete; instead, it awaits a new connection from the target. Each event log received in this manner is stored in *file.logNo*.**wvr**, where *file* is the argument given to the **-o** option (**eventLog** by default), and *logNo* is the position of the log in question within the sequence of event logs received. This process continues until the user terminates the **evtRecv** process on the host by typing the interrupt key (e.g., **Ctrl-C**) or by sending **SIGTERM** to the process. Note that repetitive receive mode is always disabled when the event stream is sent to standard output, even when **-r** is specified (see the description of the **-o** option above).

NOTE: The Windows tool is similar in function. For information see the *WindView User's Guide.*

SEE ALSO

wvLib, ***wvEvtLogEnable*( )**, ***wvEvtLogDisable*( )**, **wvHostLib**, ***wvHostInfoShow*( )**, ***wvHostInfoInit*( )**

launch

NAME

launch – the Tornado launcher

SYNOPSIS

```
launch [-T.clmode] [-v.ersion] [-h.elp] [Xt_Standard_Command_line_Parameters]
```

DESCRIPTION

The launcher is a graphical control tool for the Tornado environment. It displays information about *target servers* and *target agents* (see **tgtsvr**) registered by the Tornado Registry (see **wtxregd**). It provides a convenient interface to launch any of the Tornado tools attached to a target server (see **windsh**, **browser**, and so on). It also provides access to Tornado administrative functions.

INFORMATION DISPLAY

The launcher presents a continuously updated list of the registered target servers. Clicking on one of the listed targets will display information about the target.

Information displayed by the launcher includes:

Name
: unique identifier of the target in the form *target@serverhost*.

Version
: target server version.

Status
: either of the strings **unreserved** or **reserved** depending on whether the target server is available for use.

Runtime
: identification string of the runtime system on the target.

Agent
: target agent type and version number.

CPU
: target CPU architecture.

BSP
: name of runtime board support package.

Memory
: amount of target memory configured.

Link
: name of transport layer between target and target server.

User
: user ID of the target's most recent user.

Start
target server start time and date.

Last
time and date of last tool attachment.

Tools
a list of the tools attached to the target.

TORNADO TOOLS

Some of the Tornado tools are directly available from the launcher. They are shown as icons in the toolbar at the bottom of the launcher's window. Many tools require the selection of a target server before activation.

The fundamental collection of Tornado tools is:

windsh
the Tornado shell, which incorporates both a C and a Tcl interpreter.

crosswind
the Tornado graphical source-level debugger.

browser
a graphical tool that monitors the target system.

project
The Tornado graphical project manager (further help on Project Facility is available from within the Project tool itself).

The launcher will also let you directly call VxSim, the VxWorks simulator, and WindView, the Wind River logic analyzer.

SERVICES AND MENUS

The menu bar at the top of the launcher window gives access to various services. Just under this menu bar is a collection of small icons each of which (from left to right) is a shortcut to the Target menu's entries (read top-down). Available services are provided in the following menus:

File menu:

Quit
Quit the launcher program.

Target menu:

Create
Display a specific window to record all information needed to launch a target server and to save this configuration for further use.

Unregister
Unregister the target server from the wind registry (**wtxregd**)

Reattach
Send an attach request to a target server. The target server then tries to reattach to the target agent. This is useful if the target agent stops communicating with its server

(because the target board rebooted, for example).

Reserve
: Restrict access to a target. This service is usable only if an unreserved target has been previously selected.

Unreserve
: Allow widespread access to a reserved target. This service is usable only if the selected target is owned by the user.

Restart
: Send a restart request to a target server. This causes the target server to restart. Information related to a previous session (such the list of the object modules loaded on the target) will be lost.

Kill
: Terminate the selected target server. Beware: any unreserved target server can be killed.

Reboot
: Send a reboot command to the target.

Support menu:

Tornado
: Display a form to submit a Technical Support Request for Tornado to WRS.

VxWorks
: Display a form to submit a Technical Support Request for VxWorks to WRS.

CrossWind
: Display a form to submit a Technical Support Request for CrossWind to WRS.

GNU
: Display a form to submit a Technical Support Request for the GNU toolkit to WRS.

WindView
: Display a form to submit a Technical Support Request for WindView to WRS.

Admin menu:

License
: Display the License Request form and transmit the request to WRS.

Install CD
: Run the appropriate system command to read and extract product components from a CD-ROM.

Install Tape
: Run the **installOption** tool to read and extract product components from a tape.

FTP WRS
: Start an anonymous FTP connection with the Wind River Systems FTP server.

Authorize
: Start an editor, as defined in the EDITOR environment variable, that lets you view the

contents of the file of authorized users. If no local user lock files exists, a new one is created.

Info menu:

Products
: Access the Sales Contact Information page on the WRS Web server through your system's WWW browser.

Training
: Access the training page on the WRS Web server through your system's WWW browser.

Announcements
: Access the Sales Contact Information page on the WRS Web server through your system's WWW browser.

User's Group
: Access user-group information and the VxWorks archive on the WRS Web server through your system's WWW browser.

www.wrs.com
: Access the WRS Web home page through your system's WWW browser,

comp.os.vxworks
: Access an archive of VxWorks user discussions maintained on the WRS Web server through your system's WWW browser.

About menu:

Tornado
: Display Tornado development environment version.

Help menu:

On launch
: Display the **launch** manual page.

Manuals
: Display the Tornado Online Manuals (run your favorite HTML browser).

OPTIONS

The launcher accepts the X Toolkit Intrinsics standard command-line parameters such as **-display** or **-iconic**, plus the following:

-T | -Tclmode
: Run a Tcl interpreter on standard input. This is useful for adding and debugging new features.

-v | -version
: Print the current version of the Tornado development environment.

-h | -help
: Print the help message.

ENVIRONMENT VARIABLES

WIND_BASE
The root location of the Tornado tree.

WIND_REGISTRY
The host on which the Tornado Registry daemon runs (see **wtxregd**).

FILES

The following resource files are required by the launcher:

$WIND_BASE/host/resource/tcl/Launch.tcl
Tcl implementation of the launcher interface.

$WIND_BASE/host/resource/tcl/app-config/all/host.tcl
Tcl default host commands bindings.

$WIND_BASE/host/resource/tcl/app-config/all/*host_type***.tcl**
Tcl *host_type*-specific defines that overwrite default defines.

$WIND_BASE/host/resource/tcl/app-config/Launch/01Adm.tcl
Tcl administration library.

$WIND_BASE/host/resource/tcl/app-config/Launch/01BspCfg.tcl
Tcl support for VxWorks BSP configuration.

$WIND_BASE/host/resource/tcl/app-config/Launch/01Supprt.tcl
Tcl Technical Support Request library.

$WIND_BASE/host/resource/tcl/app-config/Launch/01TgtSvr.tcl
Tcl target server managing library.

$WIND_BASE/host/resource/tcl/app-config/Launch/01VxSim.tcl
Tcl VxSim support.

$WIND_BASE/host/resource/tcl/app-config/Launch/01WView.tcl
Tcl WindView support.

The following resource file is applied if present:

$HOME/.wind/launch.tcl
Tcl implementation of the user's facilities for the launcher.

SEE ALSO

wtxregd, **tgtsvr**, **windsh**, **browser**, *Tornado User's Guide*

tgtsvr

NAME

tgtsvr – the target board server

SYNOPSIS

```
tgtsvr  [-A.llsyms] [-B.ackend backendName] [-Bd.ebug fileName]
        [-Bm.ax size] [-b.ps linespeed] [-Br.esend number]
        [-Bt.imeout timeout] [-C.onsole] [-c.ore fileName]
        [-d.evice device] [-display hostName:0]
        [-f.ormat formatName] [-h.elp] [-hfc] [-L.ock]
        [-m.emory nbytes] [-n.ame serverName] [-N.osyms]
        [-p.ort portNumber] [-R TSFS_root] [-redirectIO]
        [-redirectShell] [-RW] [-s.ynchro] [-use_portmapper]
        [-u.sers fileName] [-V.erbose] [-v.ersion]
        [-Wd.ebug fileName] [-Wf.ilter request] [-Wm.ax size]
        targetName [backend specific options]
```

DESCRIPTION

The target server is the Tornado component that allows development tools, such as the shell (see **windsh**) or a debugger, to communicate with a remote target system, such as VxWorks. The Tornado tools are autonomous programs running on a cross-development host. They are attached to a particular target server when they begin executing.

The server communicates with the target system through an interface called the *target agent*. This agent is either integrated with the target system (for instance, as a task), or independent from it. When **tgtsvr** is started, it identifies the target agent by means of the only required argument: the name of a target board running the target agent.

The name of the target board is linked with the name of the host machine where the target server runs, to form a unique identifier used throughout the working session by all tools. This name is recorded in the name database of the Tornado *Service Registry* (see **wtxregd**). The form of this identifier is *targetName*@*serverHost*. For instance, **tPad@aven** refers to the target named **tPad** as represented by a target server running on the host **aven**.

An alternative target name, however, may be specified with the **-n** option if the board name is already in use.

Tools may use truncated identifiers, if the short names match a unique name among all names registered by the Tornado Registry (see **launch** and **wtxregd**). Any unique substring in the board name is sufficient, and the "@" extension may be omitted as well.

The target server gets requests from the Tornado tools. These requests, depending on their type, may either be satisfied by the target server itself, or require that the target server in turn send requests to the target agent.

Locating the Target Executable

The target server depends on a host-resident image of the target executable. By default (if the **-c** option is not specified), the target server queries the agent running on the target for this path name. The default works well for the common situation where the runtime code is downloaded from the host. However, in some situations (for example, if the target is running a standalone version of VxWorks generated from another host), the target agent cannot supply a useful path for the executable on the host. In this situation, use the **-c** option to specify the path explicitly. Environment variables and **~** are recognized and expanded by the target server as follow:

"**~**", if given as first character of the pathname, is expanded to the user's home directory, or if another user is specified ("**~joe**"), "**~**" will expanded to joe's home directory. **WIN32 users**,"**~**" will be expanded to the environment variable "**$HOME**", or "**$HOMEDRIVE$HOMEPATH**" if "**$HOME**" is not defined. If none of these variables are defined, "**~**" will be ignored.

Environment variables can be set by using "**$VAR**," "**$(VAR)**," and "**{VAR}**" notation. They will be expanded to the value set up on the target server's environment, or will be ignored if they are not defined.

"**~/$(VXWORKS).exe**" pathname will be expanded to "**/home/joe/proj/vxWorks.exe**" if the user's home directory is "**/home/joe**," and **VXWORKS** is set to "**proj/vxWorks**."

Authentication and Access Permission

The target server allows for user access permission to be restricted. The resource file **$WIND_BASE/.wind/userlock** can be created to hold a list of authorized user IDs (a single + sign means that universal access is allowed). If this file does not exist, the target server will assume that no user restriction will apply. Alternative resource files may be specified with the **-u** option. A target server restricted in this way refuses any requests from tools started by an unauthorized user.

It is also possible to lock a target server with the **-L** option. Then only the user that starts the target server can connect tools to that server (see also the Reserve and Unreserve menu items of **launch**).

Target Server Components

The target server is made up of the following units:

– communication unit with the tools.

– communication unit, or *back end,* with the target agent.

– object module management unit (loader, unloader).

– target symbol table management unit.

– target memory management unit.

– virtual input/output management unit.

WTX Protocol

Communication between the target server and the Tornado tools is done via the RPC/XDR mechanism. Tools' requests and target servers' answers or events follow the formats defined by the Wind River Tool Exchange (WTX) messaging protocol. There is no requirement for the Tornado tools and the target server to operate from the same host machine. They may be distributed across a network.

If the **-Wd** option is specified, all WTX requests are logged in a log file. The default behavior is to append log messages at the end of the log file (if it does not exists, it will be created). If the **-Wm** option is also specified, the file size will be limited to the given value, and written as a circular file: i.e. when this value is reached, the file is rewritten from the beginning. If the file exists, it will be erased.

Note that a tool can be connected to more than one target server allowing for managing data coming from several remote target systems.

The **-Wf** option can be used to filter a particular WTX request in the log file. The default filter is set to "**WTX_EVENT_GET**" to avoid thousands of such request when a wind shell version 1 is connected to the target server.

Back Ends

The target server's back end is the intermediary for all communication with the target agent. Thus, the back end must be designed to use whatever communication protocol and transport layer the agent uses. Because not all agents can use the default protocol (WDB over RPC/XDR) and transport layer (Ethernet), alternative back ends can be specified explicitly. Custom back ends are also possible.

The following back ends are supported by Wind River Systems (see **$WIND_BASE/host/$WIND_HOST_TYPE/lib/backend**):

wdbrpc (default)
: The Tornado WDB RPC back end. It is the most frequently used, and supports either Ethernet or serial connections. This back end supports either system-level or task-level views of the target.

wdbpipe
: This back end is to be used on all simulators. It is based on named pipes on UNIX hosts and mail slots for windows hosts.

wdbserial
: A version of the WDB back end supporting only serial hardware connection. Note that in order to use this back end the serial connection should only use the "Tx", "Rx" and "Gnd" signals by default. When the **-hfc** (hardware flow control) option is used, the "RTS", "CTS" and "DTR" signals are also supported.

netrom
: A proprietary communications protocol for NetROM, a networked ROM emulator from Applied Microsystems Corporation.

loopback
: A testing back end. This back end is not useful for connecting to targets; it is used only to exercise the target server daemon during tests.

Use the **-B** option to select an alternative back end.

If the target agent is connected through the **wdbserial** back end, target server options **-d** and **-b** allow the *tty* device and the serial line speed to be specified, respectively. The **-hfc** option activates hardware flow control on the serial link.

If the target agent is connected through the **wdbrpc** backend, the **-p** option allows to specify the UDP port number.

If the communication link between the target server and the target agent is slow, it may be necessary to adjust the back end timeout value (with the **-Bt** option), as well as the back-end retry count (with the **-Br** option).

Back ends may also provide their own set of options (see *Tornado API Programmer's Guide* for details). The back end options are shown first. These options can be viewed with:

```
tgtsvr -B bkendName -h
```

The WDB requests can be logged on a file by using the **-Bd** option. The default behavior is to append log messages at the end of the log file (if it does not exists, it will be created). If the **-Bm** option is also specified, the file size will be limited to the given value, and written as a circular file: i.e. when this value is reached, the file is rewritten from the beginning. If the file exists, it will be erased.

Object Module Management

The target server may handle object modules from various format (currently, a.out COFF, ELF, SOM, and pecoff). The core file is analyzed in order to determine what object module format will be used for the working session. It is possible to bypass this determination with the **-f** option followed by a format name. Supported format names can be found in the resource file: **$WIND_BASE/host/resource/target/architecturedb**. The target server can be extended to support new Object Module Format (see the *Tornado API Programmer's Guide: Object Module Loader*).

Target Symbol Table

The target server maintains (on the host) a symbol table for the target executable. It builds this symbol table from an input file called the *core file.* The symbols and memory locations obtained from this file are used to calculate relocation information when linking other user modules. The target server normally obtains the location of the core file from the target agent (in which case it is the file originally used as the executable for the agent itself). However, because the core file may no longer be in the location where it was used to load the agent, a path name for it can also be specified explicitly with the **-c** option (see *Locating the Target Executable* above for giving an alternate path name).

It is also possible to prevent the target server from building the target symbol table from the core file with the **-N** option. If the target server is started with this option, the first file to be loaded must be a fully-linked object file (an object file with no external references). Any subsequent modules loaded may be relocatable; the server calculates relocation information by reference to that first loaded object file.

Using the **-A** option forces the target server to include all global and local symbols from the core file in the target server symbol table.

Target Memory Management

The target server manages the target agent memory pool on the remote system. This memory pool is mainly used by the loader when object files are downloaded. The target server automatically increases the size of the agent memory pool when necessary (when there is not enough room to load a file, for example). A cache is implemented so that memory-related requests from Tornado tools may be satisfied at the target server level, avoiding the transfer of data from the real target memory. This cache has a default maximum size of 1 MB.

The **-m** option allows to specify a maximum size for the cache. This may be required when the agent memory pool size becomes greater than the maximum size of the cache. In this situation, the memory-related requests that fall outside the cache are satisfied at the target level, and thus are substantially slower.

The Tornado **browser** provides a graphical view of the target agent memory pool utilization.

Virtual Input/Output Mechanism

The target server can redirect data through *virtual Input/Output channels.* For target tasks to have access to this mechanism, a *Virtual I/O Driver* must be included in the target system. When this driver is included, any task on the target may open a virtual channel to read from, or write to, that channel. On the host, any tool may open the same virtual channel to write to, or read from, that channel. Thus the target server acts as an I/O dispatcher, multiplexing whatever physical communications layer is available to allow run-time tasks and host tools to communicate easily.

When the target server is started with the **-C** option, a console window attached to virtual channel 0 is displayed. On UNIX, this window can be displayed on a specified X server (including a host other than where the target server is running) with the **-display** option. The number of buffered lines (default 88) can be changed by setting the environment variable **WTX_CONSOLE_LINES** to the number of desired buffered line. Set this variable before starting your UNIX target server.

This permits any task on the target to open virtual I/O channel 0 to send characters to, or read characters from, this window. If started with **-redirectIO**, a redirection of target standard I/O is automatically done. If **-redirectIO** is set, but **-C** flag is not set, the target I/O will be redirected to the target server, but, since no console will be present to display the informations, events will be sent to the connected WTX tools.

The target server can also be used as a virtual target shell console: The shell is running on the target, and its I/O are done from the target server virtual console. To do this, use the **-redirectShell** flag in conjunction with **-C** flag. The target server will automatically redirect the target shell I/O into the default target server console. The shell must be included in the target system. This feature is useful if target is only accessible through back end and

actions which cannot be done via windsh (like loading object from a file system local to the target) have to be performed.

CAVEAT

Redirecting target I/O into the target server's console may lead side effects:

– If you use **-redirectIO** when a target shell is running, its I/O will also get redirected in the target server's console. You will see a double echo (one echo for the target server console, and another one by the target shell itself), and the target Shell input will be lost when the target server stops. Use the **-redirectShell** in conjunction with **-redirectIO** option to avoid this.

– **-redirectIO** and **-redirectShell** flags are not exclusives. They can be jointly used. But in this case, only the target shell will get its input from the target server's console. Other tasks pending on a read on their **stdin** will pend forever (unless the target shell is destroyed).

– If you use Windsh with a target server which has **-redirectIO** and **-C** set, remember that windsh also redirects the I/O. So if you try

```
scanf ("%s", buf)
```

from the command line the input will be done by the windsh, not from the target server's console.

– If you use windsh with a target server which has not set **-redirectIO** and try a command which launches another task which wants to write messages, you won't see them: the child tasks get their I/O reset to the global I/O descriptors (if the target server was launched with **-C**, those outputs will be sent in the target server's console). Setting **-redirectIO** without **-C** will permit to see the task's child output into the windsh which launched them. But be aware that ALL TOOLS will be notified of the target's task output.

– Using **-redirectShell**, if the shell cannot be redirected to the target server console, the characters typed in the console window won't be echoed, since that the shell's job to echo each typed characters.

Target Server File System (TSFS)

Other virtual I/O channels are available for general file I/O. Target tasks can use these channels to access the host's file system just as they would access target connected file systems. This type of virtual I/O is referred to as the Target Server File System or TSFS.

The part of the host's file system visible to targets using the TSFS is specified with the **-R *root*** option. For example, if a root of "/users/john" is specified, the target will only be able to access files on the host's file system within and below /users/john.

By default, the host's files are accessible for reading only by target processes using the TSFS. To make the files accessible for reading and writing, **-RW** should be specified. When **-RW** is specified, access to the target server is automatically restricted to host processes with the same user ID as the target server, as if the **-L** option was specified.

OPTIONS

-A | -Allsyms
Include all local and global core-file symbols in the target symbol table.

-B *backendName* **| -Backend** *backendName*
Specify an alternative back end to communicate with the target agent. The default is a back end using the WDB protocol based on the RPC/XDR mechanism. Back end names can be deduced from the names of the files in **$WIND_BASE/host/$WIND_HOST_TYPE/lib/backend** (just remove the extension).

-Bd *fileName* **| -Bdebug** *fileName*
Log every WDB request sent to the target agent in the specified file. If the file already exists, log messages will be appended to it (unless the **-Bm** flag is set). Back ends that are not based on WDB RPC ignore this option.

-Bm *size* **| -Bmax** *size*
Max size in bytes for the WDB logging file. If this flag is set, the file size will be limited to the given value, and written as a circular file: i.e. when this value is reached, the file is rewritten from the beginning. If the file exists, it will be erased. So, be aware that if the target server restarts (due to a target reboot, for example), the WDB log file will be reset.

-b *linespeed* **| -bps** *linespeed*
Specify the speed of a serial link used to communicate with a target agent. The default value is 9600 bps.

-Br *number* **| -Bresend** *number*
Specify the number of times the back end should attempt to repeat a request to the target agent, if a transmission fails. Default: three retries. Some back ends may ignore this option.

-Bt *timeout* **| -Btimeout** *timeout*
Specify the timeout, in seconds, for back-end transactions with the target agent. Default: one second. Some back ends may ignore this option.

-C | -Console
Start a console window. Target tasks can perform I/O through this window using virtual channel 0.

-c *fileName* **| -core** *fileName*
By default, the target server gets the name of the core file (the executable initially running on the target) from the target agent. If the target agent does not have this information (or if its information is out of date), use this option to specify a path to the core file explicitly.

-d device | -device *device*
Specify the tty device used to communicate with a target agent. The default device is "/dev/tty" on Solaris, "/dev/tty0p0" on HP-UX and "COM2" on Windows.

-display *hostName:0*
(UNIX only) Specify the X server to contact in order to display the virtual console window.

-f *formatName* | **-format** *formatName*
Name of the alternate object module format (for example, a.out or COFF) that will be managed by the target server.

-h | **-help**
Print a help message summarizing **tgtsvr** usage and options.

-hfc
Activate hardware flow control on a serial link using RTS, CTS, and DTR signals. This option is available only with the **wdbserial** back end.

-L | **-Lock**
Restrict access to this target server to processes running under the same user ID as the server.

-m *nbytes* | **-memory** *nbytes*
Set the size of the agent memory pool cache managed by the target server (default is 1 MB).

-n *serverName* | **-name** *serverName*
Specify an alternative name for the target server (instead of the default, based on the target's name). Target server's name should be constituted by alphanumeric characters only.

-N | **-Nosyms**
Do not use a core file to build the target symbol table.

-p *portNumber* | **-port** *portNumber*
Specify the UDP port number to communicate with a target agent when the **wdbrpc** back end is used. The default port number is 0x4321.

-R *root*
Establish the root of the host's file system visible to target processes using the Target Server File System.

-redirectIO
Redirect the target global stdin, stdout, and stderr in the target server. If **-C** flag is not set, WTX events will be sent to all WTX tools when characters come from the target.

-redirectShell
Start a console window with target shell stdin, stdout and stderr redirected in it. This flag is valid only if **-C** flag is set.

-RW
Allow read and write access to host files by target processes using the target server File System. When this option is specified, access to the target server is restricted as if **-L** were also specified.

-s | **-synchro**
Synchronize target and host symbol tables. The symbol table synchronization facility must also be included in the target image; see the reference entry for **symSyncLib**.

-use_portmapper
Use the local portmapper to register the target server rpc services. This flag **MUST** be set if tornado tools version 1.x have to connect to this target server.

-u *fileName* | **-users** *fileName*
Specify a file containing a list of authorized users. Only users whose IDs appear as lines in this file will be able to connect tools to the target server, unless the file contains the character **+** on a line by itself (which authorizes all users).

-V | -Verbose
Turn on the target server's verbose mode. By default, the target server is silent. In verbose mode, it displays information, warning and error messages on the standard output.

-v | -version
Identify the version of the target server.

-Wd *fileName* | **-Wdebug** *fileName*
Log every WTX request sent to the target server in the specified file. If the file exists, log messages will be appended to it (unless the **-Wm** flag is set).

-Wf *request* | **-Wfilter** *request*
Remove WTX *request* from the WTX log file. The default WTX log behavior is to log every requests the target server is servicing. This may lead to a huge file. This flag allows to reduce the amount of information by giving a regular expression to filter out WTX requests.

-Wm *size* | **-Wmax** *size*
Max size in bytes for the WTX logging file. If this flag is set, the file size will be limited to the given value, and written as a circular file: i.e. when this value is reached, the file is rewritten from the beginning. If the file exists, it will be erased. So, be aware that if the target server restarts (due to a target reboot, for example), the WTX log file will be reset.

EXAMPLES

Start a target server on target with IP address equal to "147.108.108.1" in verbose mode and give it the name "myTargetServer".

```
tgtsvr -V -name myTargetServer 147.108.108.1
```

Display the flags handled by the wdbpipe backend.

```
tgtsvr -h -B wdbpipe
```

Start a target server named "myTargetServer" on VxWorks simulator number "0" with WDB request log going to file "/tmp/vxsim0Wdb.log"

```
tgtsvr -B wdbpipe -Bd /tmp/vxsim0Wdb.log vxsim0 -n myTargetServer
```

Specify a core file to the target server attached to a serial line on COM2 at 38400 bauds (Windows). Note that the target name flag can be anything in this case it is only used to build the target server name.

```
tgtsvr -c D:\tornado\target\config\mv1604\vxWorks -V \
       -B wdbserial -b 38400 myTarget
```

Start a target server with a target shell console on target called "arm7tBoard". VxWorks should be configured to include the target shell for this to work.

```
tgtsvr -C -redirectShell -V arm7tBoard
```

Start a target server in verbose mode on target called "arm7tBoard" with all target I/O redirected in the target server's console.

```
tgtsvr -C -redirectIO -V arm7tBoard
```

ENVIRONMENT VARIABLES

WIND_BASE
root location of the Tornado tree.

WIND_REGISTRY
host on which the Tornado Registry daemon runs (see **wtxregd**).

FILES The following resource files are required by the target server:

$WIND_BASE/.wind/userlock
default authorization file (list of authorized users).

$WIND_BASE/host/resource/target/architecturedb
supported targets resource base.

SEE ALSO **wtxregd**, **launch**, **browser**, **windsh**, *Tornado User's Guide*, *Tornado API Programmer's Guide*

vxColor

NAME **vxColor** – graph coloring demo for Tornado

SYNOPSIS

```
vxColor targetServer [-V.erbose]
```

DESCRIPTION This command launches the **vxColor** demo Graphical User Interface. The **vxColor** demo is based on a self-stabilizing algorithm for coloring a planar graph. Given a planar graph (a 2D graph with no intersecting arc) and given 6 colors, the aim of the algorithm is to assign each region of the graph a color so that no pair of connected regions (regions sharing a least one arc) can have the same color. The main tasks in the demo are the "controller" task and some number of "region" tasks.

The GUI can be invoked from the command line on the host shell. The following parameter is required and a verbose option is available:

targetServer
A living target server connected to the target to be used.

-V
Turn on verbose mode, which displays important steps of the interaction between host and target.

NOTE: The **vxColor** demo is based on WTX and a TK-capable interpreter.

LAUNCHING THE DEMO

(1) Switch on your target system (booting with VxWorks).

(2) Launch a target server for this target.

(3) Type the command line. For example, with a Motorola MVME162 board, assuming you named your target server **demo162**, enter:

```
vxColor demo162
```

After executing the above steps, your screen should display a window named 'Tornado Graph Coloring Demo'. The window displays three buttons:

(1) a toggle that switches debug mode on or off

(2) a menu button that allows you to select a predefined graph to run the demo with

(3) a quit button

Debug mode should be turned on to debug the target side of the VxColor demonstration with CrossWind. Turning on debug mode makes the stack size for the graph coloring tasks bigger in order to accommodate the extra space needed by local variable when the code is compiled with the debug option (-g) turned on. The header file of the demo (**$WIND_BASE/target/src/demo/color/vxColor.h**) contains defines for different target types of the stack sizes of the controller and region tasks, in debug or normal mode. Normal mode demands the least memory and should be used when running the demo on boards with little memory capacity. You can adjust the stack size by editing the header file. Whichever mode you select, remember that you can watch the stack usage with the Tornado browser and that a stack overflow can lead to a severe system crash.

When a graph has been selected, the main window displays this graph as a set of colored regions separated with border lines. The window displays a new set of buttons which are contextual controls.

The following gives a description of the contextual controls:

START		Starts a coloring session.
COLOR EDIT		Menu for changing region colors:
	- RANDOM	randomly initializes each region's color.

	- UNIFORM	uniformly initializes all regions' colors.
	- LOCAL	allows editing an individual region's color.
NEIGHBORHOOD		Displays each region neighbors (following cursor).
QUIT		Exits the demo.

During a coloring session, the contextual controls are:

STOP	Breaks a coloring session
PAUSE	Temporarily halts a coloring session

During a PAUSE, the only contextual control is:

CONTINUE	Resumes a coloring session after a pause.

Actually, each region in the graph has a color attribute and is controlled by a task running on the target. The existing neighborhood relationships within the graph are represented on the target as communication channels between tasks. Thus, the target is populated with a set of tasks that have exchanges between each other. The purpose of these exchanges is to honor a simple rule which says that no neighboring regions can have the same color and that only six (6) different colors are available. This scheme has been proven to be always possible. See "A Self-stabilizing Algorithm for Coloring a Planar Graph," by S. Gosh and M.H. Karaata in *Distributed Computing* (1993) 7:55-59.

When a solution is found, region colors freeze. You can then break this stabilized status by changing some colors using the COLOR EDIT menu. If you select LOCAL mode, choose a new color for any region by clicking on the desired region as many time as necessary (one out of the 6 possible colors will be selected in a ring fashion). When you are satisfied with your changes, press the START button to compute a new coloring scheme.

While coloring is in progress, you can use the STOP and PAUSE buttons. The STOP button lets you break the target's coloring activity; the PAUSE button temporarily suspends this activity so that you can observe the tasks of the demo with various Tornado tools. When PAUSE is on, the only available action is CONTINUE, which resumes the coloring activity.

RECOMPILING THE TARGET MODULE

The demo's target-side building technology uses a simple makefile that requires two parameters settings: one for **CPU** and the second for **TOOL**. The **CPU** and **TOOL** parameters must be set to a supported architecture value and to a supported toolchain value (the default value of **TOOL** is "gnu" when omitted).

Examples To recompile the demo's target module for the listed architectures, enter the following, in the directory **target/src/demo/color**:

Motorola 68040 board:

```
make CPU=MC68040
```

Solaris simulator:

```
make CPU=SIMSPARCSOLARIS
```

Intel i960CA board:

```
make CPU=I960CA
```

To produce debugging information:

(1) Recompile the demo's target module (as described here above).

(2) Edit the command line produced in (1) to change the optimization flag to **-g**.

(3) Execute this modified command line.

ENVIRONMENT VARIABLES

WIND_BASE
root location of the Tornado tree.

WIND_REGISTRY
host on which the Tornado Registry daemon runs (see **wtxregd**).

FILES

The following files are required by the vxColor demo:

$WIND_BASE/host/WIND_HOST_TYPE/bin/wtxwish
WTX protocol capable TK interpreter.

$WIND_BASE/host/WIND_HOST_TYPE/bin/vxColor
shell script invoking **wtxwish** to interpret **demoHost.tk**.

$WIND_BASE/host/src/demo/color/demoHost.tk
demo host GUI.

$WIND_BASE/host/src/demo/color/United-States
a planar graph representing the border states of the USA.

$WIND_BASE/host/src/demo/color/FranceRegions
a planar graph representing the border regions of France.

$WIND_BASE/host/src/demo/color/Wheel
a simple planar graph.

$WIND_BASE/target/lib/obj*CPUTOOL*test/vxColor.o
a CPU specific demo object module.

BUGS

If the NumLock key is on, the demo will hang.

SEE ALSO

tgtsvr, *Tornado User's Guide: Setup and Startup*

windsh

NAME

windsh – The Tornado Shell

ROUTINES

***agentModeShow*( )** – show the agent mode (*)
***b*( )** – set or display breakpoints
***bd*( )** – delete a breakpoint
***bdall*( )** – delete all breakpoints
***bh*( )** – set a hardware breakpoint
***bootChange*( )** – change the boot line
***browse*( )** – send a message to the browser asking it to browse an address (*)
***c*( )** – continue from a breakpoint
***cd*( )** – change the default directory
***checkStack*( )** – print a summary of each task's stack usage
***classShow*( )** – show information about a class of objects (*)
***cplusCtors*( )** – call static constructors (C++)
***cplusDtors*( )** – call static destructors (C++)
***cplusStratShow*( )** – show C++ static constructors calling strategy (*)
***cplusXtorSet*( )** – change C++ static constructor calling strategy (C++)
***cret*( )** – continue until the current subroutine returns
***d*( )** – display memory
***devs*( )** – list all system-known devices
***h*( )** – display or set the size of shell history
***help*( )** – print a synopsis of selected routines
***hostShow*( )** – display the host table
***i*( )** – print a summary of each task's TCB, task by task
***iStrict*( )** – print a summary of all task TCBs, as an atomic snapshot (*)
***icmpstatShow*( )** – display statistics for ICMP
***ifShow*( )** – display the attached network interfaces
***inetstatShow*( )** – display all active connections for Internet protocol sockets
***intVecShow*( )** – display the interrupt vector table
***iosDevShow*( )** – display the list of devices in the system
***iosDrvShow*( )** – display a list of system drivers
***iosFdShow*( )** – display a list of file descriptor names in the system
***ipstatShow*( )** – display IP statistics
***l*( )** – disassemble and display a specified number of instructions
***ld*( )** – load an object module into memory
***lkAddr*( )** – list symbols whose values are near a specified value
***lkup*( )** – list symbols
***ls*( )** – list the contents of a directory
***m*( )** – modify memory
***memPartShow*( )** – show partition blocks and statistics
***memShow*( )** – show system memory partition blocks and statistics
***moduleIdFigure*( )** – figure out module ID, given name or number (*)

***moduleShow*( )** – show the current status for all the loaded modules
***mqPxShow*( )** – show information about a POSIX message queue (*)
***mRegs*( )** – modify registers
***msgQShow*( )** – show information about a message queue
***period*( )** – spawn a task to call a function periodically
***printErrno*( )** – print the definition of a specified error status value
***printLogo*( )** – display the Tornado logo
***pwd*( )** – display the current default directory
***quit*( )** – shut down WindSh (*)
***reboot*( )** – reset network devices and transfer control to boot ROMs
***repeat*( )** – spawn a task to call a function repeatedly
***routestatShow*( )** – display routing statistics
***s*( )** – single-step a task
***semPxShow*( )** – show information about a POSIX semaphore (*)
***semShow*( )** – show information about a semaphore
***shellHistory*( )** – display or set the size of shell history
***shellPromptSet*( )** – change the shell prompt
***show*( )** – display information on a specified object
***smMemPartShow*( )** – show user shared memory system partition blocks and statistics (*)
***smMemShow*( )** – show the shared memory system partition blocks and statistics
***so*( )** – single-step, but step over a subroutine
***sp*( )** – spawn a task with default parameters
***sps*( )** – spawn a task with default parameters, and leave it suspended (*)
***sysResume*( )** – reset the agent to tasking mode (*)
***sysStatusShow*( )** – show system context status (*)
***sysSuspend*( )** – set the agent to external mode and suspend the system (*)
***taskCreateHookShow*( )** – show the list of task create routines
***taskDeleteHookShow*( )** – show the list of task delete routines
***taskIdDefault*( )** – set the default task ID
***taskIdFigure*( )** – figure out the task ID of a specified task (*)
***taskRegsShow*( )** – display the contents of a task's registers
***taskShow*( )** – display task information from TCBs
***taskSwitchHookShow*( )** – show the list of task switch routines
***taskWaitShow*( )** – show information about the object a task is pended on (*)
***tcpstatShow*( )** – display all statistics for the TCP protocol
***td*( )** – delete a task
***tftpInfoShow*( )** – get TFTP status information
***ti*( )** – display complete information from a task's TCB
***tr*( )** – resume a task
***ts*( )** – suspend a task
***tt*( )** – display a stack trace of a task
***tw*( )** – print info about the object the given task is pending on (*)
***udpstatShow*( )** – display statistics for the UDP protocol
***unld*( )** – unload an object module by specifying a file name or module ID
***version*( )** – print VxWorks version information

***w*( )** – print a summary of each task's pending information, task by task (*)
***wdShow*( )** – show information about a watchdog

SYNOPSIS

```
windsh  [-c.plus C++_library] [-e.xecute expression] [-h.elp]
        [-n.oinit] [-p.oll value] [-q.uiet] [-s.tartup file]
        [-T.clmode] [-v.ersion] serverIdentifier
```

DESCRIPTION

WindSh is the Tornado shell. The shell provides remote interactive access to the target run-time system through both a C interpreter and a Tcl interpreter.

The shell attaches to the target server specified by *serverIdentifier* (see **tgtsvr**).

To execute a list of commands when the shell starts, collect these commands in a file (startup script) and identify the file with the **-s** option.

The shell has **vi**-like editing capabilities and a history mechanism. The ESC key acts as a toggle between input mode and edit mode. History and editing capabilities are available regardless of which interpreter is in use.

C Interpreter

The shell's C interpreter (prompt: >) can execute almost any expression using C operators, and can invoke compiled C functions on the target. Symbols are created as needed for shell expressions, and are added incrementally to the target symbol table. Interactive sessions use the C interpreter by default.

Tcl Interpreter

The shell's Tcl interpreter (prompt: tcl>) executes Tcl functions, including both functions based on the WTX protocol and user-provided Tcl procedures. To enter the Tcl interpreter directly for an interactive session, start **windsh** with the option **-T**.

The Tcl interpreter can also call C functions in the target; however, to establish the proper C environment for such calls you must use the **shParse** Tcl command. For example:

```
tcl> shParse sysClkRateGet()
60
```

When the Tcl interpreter does not recognize a command, it passes it to the UNIX shell or the Windows command processor.

Built-in Routines

The Tornado shell includes a set of built-in routines. These routines are executed on the host, not in the context of the remote run-time. They are available from both the C interpreter and the Tcl interpreter. The most important built-in routines are ***i*( )**, ***ti*( )**, ***d*( )**, ***l*( )**, ***ts*( )**, ***tr*( )**, ***td*( )**, ***ld*( )**. In Tcl mode, type:

```
tcl> set shellProcList
```

to get the complete list of built-in functions.

Non-interactive Sessions

windsh can be used in non-interactive sessions, by simply providing input on its standard input stream. For example:

```
garonne% echo "i" | windsh vxsim1@garonne | grep tExcTask
tExcTask _excTask 3b3fc0   0 PEND 9bee8 3b3dd8   0   0
```

It is thus possible to execute a sequence of commands without entering interactive mode by redirecting input to a command file:

```
garonne% windsh vxsim1@garonne < myOwnCommandFile
```


Completion and Synopsis facilities

The shell supports target symbol completion and path completion using CTRL-D and TAB:

– To complete a symbol (or a path), begin typing the symbol and then press CTRL-D to get the list of symbols matching the word entered:

```
-> taskS CTRL-D
taskSwitchHookAdd         taskSpawn                 taskStackAllot
taskSRSet                 taskSwitchTable           taskSuspend
taskSwitchHookDelete      taskSRInit                taskSwapHookAttach
taskSwapHookAdd           taskSwapHookDetach        taskSwapReference
taskSwapTable             taskSwapHookDelete        taskSrDefault
taskSafe
-> taskS
```

– Select the symbol to complete by adding one or more characters and press CTRL-D or TAB to fully complete the symbol. Once the symbol is fully completed, the synopsis of the target function (or the WindSh command) can be printed by pressing CTRL-D. It is also possible do display the HTML help of the function by pressing CTRL-W.

```
-> taskSp[TAB]
-> taskSpawn CTRL-D
taskSpawn() - spawn a task
int taskSpawn
    (
    char     *name,    /* name of new task (stored at pStackBase) */
    int      priority, /* priority of new task                    */
                       /* task option word                        */
    int      stackSize, /* size (bytes) of stack needed plus name    */
                       /* entry point of new task                 */
    int      arg1,  /* 1st of 10 req\xd5 d task args to pass to func */
    int      arg2,
    int      arg3,
    int      arg4,
    int      arg5,
    int      arg6,
    int      arg7,
    int      arg8,
    int      arg9,
```

```
    int      arg10
    )
-> taskSpawn
```

Meta-characters

Some characters have special meanings to the shell:

? When used alone, this meta-character acts as a toggle to switch between the C and Tcl interpreters. In the C interpreter context, if the question mark is followed by text, that text is interpreted as a Tcl expression, but without entering the Tcl mode. For example:

```
-> ? wtxAgentModeGet
AGENT_MODE_TASK
```

@ This meta-character forces the C interpreter to treat the word that follows as a target symbol. This is useful when a target function has the same name as a shell built-in function. For example:

```
-> @d
```

> This meta-character redirects C-interpreter output. For example:

```
-> moduleShow >/tmp/loaded
```

< This meta-character redirects C-interpreter input.

```
-> < myOwnCommandFile
```

Standard Input and Output

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 redirected to the same window as WindSh. For example, in a default configuration of Tornado, the following is what see in the shell window from a ***printf*( )** invocation:

```
-> printf ("Hello World!\n")
Hello World!
value = 13 = 0xd
->
```

This behavior can be dynamically modified using the **shConfig** Tcl procedure:

```
-> ?shConfig SH_GET_TASK_IO off
->
-> printf ("Hello World!\n")
value = 13 = 0xd
->
```

The shell duly reports the ***printf*( )** result, indicating that 13 characters were printed. But the output itself goes elsewhere by default.

NOTE

The standard Input and Output are only redirected for the called function, if this function spawns other tasks, the Input and Output of the spawned tasks won't be redirected to WindSh. To have all IO redirected to WindSh, the following script can be used:

```
# Turn Off WindSh IO redirection
?shConfig SH_GET_TASK_IO off
# Set stdin, stdout, and stderr to /vio/0 iff not already in use
if { [shParse {tstz = open ("/vio/0",2,0)}] != -1 } {
    shParse {vf0 = tstz};
    shParse {ioGlobalStdSet (0,vf0)} ;
    shParse {ioGlobalStdSet (1,vf0)} ;
    shParse {ioGlobalStdSet (2,vf0)} ;
    shParse {logFdSet (vf0);}
    shParse {printf ("Std I/O set here!\n")}
} else {
    shParse {printf ("Std I/O unchanged.\n")}
}
```

C++ Support

The **windsh** shell integrates a C++ demangler. If you type an overloaded function name, or if you type a method name with implementations in different classes, you will get a menu of choices. In the same way, doing a ***lkup*( )** on a symbol displays the various classes in which this symbol name is defined. Examples:

Consider the symbol "talk" as implemented by a hierarchy of classes:

```
-> lkup "talk"
Animal::_talk(void)       0x00376054 text     (cptest.o)
Bear::_talk(void)         0x00376080 text     (cptest.o)
Bird::_talk(void)         0x003760a8 text     (cptest.o)
value = 0 = 0x0
```

Consider the symbol "foo" as an overloaded function with two signatures:

```
-> lkup "foo"
_foo(int)                 0x00375fd8 text     (cptest.o)
__GLOBAL_$I$foo(int)      0x003760f8 text     (cptest.o)
_foo(char *)              0x00375ffc text     (cptest.o)
value = 0 = 0x0
```

Try to call "talk" and you'll be asked which one to invoke:

```
-> talk
0: Animal::talk(void)
1: Bear::talk(void)
2: Bird::talk(void)
Choose the number of the symbol to use: 1
growl!
```

```
value = 0 = 0x0
```

It is possible to select a specific demangler style by means of the Tcl variable **shDemangleStyle**. Three styles are currently available: **gnu** (the default), **arm**, and **none** (no demangling). For instance:

```
-> ?set shDemangleStyle none
none
```

Demangling can be reached through the Tcl routine:

demangle *style symbol mode*

The *symbol* is demangled using the given *style* and according to the given *mode*, if possible. If the *symbol* is not understood by the demangler, *symbol* is returned unmodified. *mode* is **0** for no demangling, **1** for short demangling, and **2** for full demangling.

The shell always understands the **gnu** and **arm** styles. Other styles can be dynamically loaded to the shell with the **-cplus** option.

```
-> ?demangle gnu getRegString__FPCcPcT1 2
    getRegString(char const *, char *, char *)
-> ?demangle gnu getRegString__FPCcPcT1 1
    getRegString
    ->
```

WindSh Environment variables

WindSh integrates an environment variables mechanism to configure the shell's behavior. The ***shConfig*( )** command allows you to display and set the controls that change the shell's behavior. This command affects only instantiation of the shell where the command is applied. At startup WindSh begins by looking for a file called **.wind/windsh.tcl** under the home directory. If the file exists, the shell reads and executes its contents as Tcl expressions before beginning to interact. This file can be used to initialize WindSh environment variables.

The following example shell initialization file turns off WindSh IO redirection and sets the Load path:

```
# Turn Off WindSh IO redirection
shConfig SH_GET_TASK_IO off
# Set Load path
shConfig LD_PATH
"C:/ProjectX/lib/objR4650gnutest/;C:/ProjectY/lib/objR4650gnuvx/"
```

DSM_HEX_MOD [on | off]
: This configuration parameter allows you to set the disassembling "symbolic + offset" mode to "on" or "off". While set to "off", the "symbolic + offset" address representation is turned on and addresses inside the disassembled instructions are given in terms of a "symbol name + offset". If turned to "on", these addresses are given their hexadecimal value.

SH_GET_TASK_IO [on | off]
Set the I/O redirection mode for the called function. If **SH_GET_TASK_IO** is set to "on" then input and output of called functions is redirected to WindSh.

LD_CALL_XTORS [target | on | off]
Set the C++ strategy related to constructors and destructors. If **LD_CALL_XTORS** is set to "target" then WindSh uses the strategy set on the target using ***cplusXtorSet*( )**. If **LD_CALL_XTORS** is set to "on", the C++ strategy is set to automatic (only for the current instantiation of WindSh); otherwise the C++ strategy is set to manual.

LD_SEND_MODULES [on | off]
Set the load mode. If **LD_SEND_MODULES** is set to "on" then the module is transferred to the target server, otherwise the target server directly accesses the module.

LD_COMMON_MATCH_ALL [on | off]
Set the loader behavior for common symbols. If **LD_COMMON_MATCH_ALL** is set to "on", then the loader tries to match common symbols against existing symbols. If a symbol with the same name is already defined, the loader takes its address. Otherwise, the loader creates a new entry. If **LD_COMMON_MATCH_ALL** is set to "off", then the loader never tries to search for an existing symbol. It creates an entry for each common symbol.

LD_PATH [path]
Set the search path for modules, the separator is **;**. WindSh first checks whether the file exists in current directory. If it exists it loads it. Otherwise, each directory in the path is searched for the given module.

```
-> ?shConfig
SH_GET_TASK_IO = on
LD_CALL_XTORS = target
LD_SEND_MODULES = on
LD_COMMON_MATCH_ALL = on
LD_PATH = C:/ProjX/lib/objR4650gnutest/;C:/ProjY/lib/objR4650gnuvx/
-> ?shConfig LD_CALL_XTORS on
-> ?shConfig LD_CALL_XTORS
LD_CALL_XTORS = on
->
```

OPTIONS

-c *C++_library_name* | **-cplus** *C++_library_name*
Select a C++ support other than the built-in if the shell is built with dynamic library support.

-e *expression* | **-execute** *expression*
Execute a Tcl expression after initialization. To execute a Tcl startup script, specify an expression using the Tcl **source** command.

-h | **-help**
Print a help message about **windsh** usage and its options.

-n | -noinit
Do not read the normal Tcl initialization files (see FILES section) when the shell starts.

-p *value* **| -poll** *value*
Set polling interval (in msec) for WindSh internal event list. Default is 200 msec. This event list is filled by the asynchronous event notification, no request is made to the Target Server.

-q | -quiet
Turns on quiet mode. In quiet mode, commands read from a script are not printed as they are read. This also affects command printing when **windsh** is started noninteractively (e.g., used in a filter or with standard input redirected).

-s *file* **| -startup** *file*
Execute the commands listed in *file* after initialization. These commands are interpreted by the shell's C interpreter.

-T | -Tclmode
Start session in Tcl mode.

-v | -version
Print Tornado version.

ENVIRONMENT VARIABLES

WIND_BASE
root location of the Tornado tree.

WIND_REGISTRY
host on which the Tornado Registry daemon runs (see **wtxregd**).

FILES

The following resource files are required by the shell's Tcl interpreter:

$WIND_BASE/host/resource/tcl/shell.tcl
WindSh Tcl entry point.

$WIND_BASE/host/resource/tcl/shellDbgCmd.tcl
Tcl implementation of WindSh **debug** commands.

$WIND_BASE/host/resource/tcl/shellMemCmd.tcl
Tcl implementation of WindSh **memory** commands.

$WIND_BASE/host/resource/tcl/shellShowCmd.tcl
Tcl implementation of WindSh **show** commands.

$WIND_BASE/host/resource/tcl/shellTaskCmd.tcl
Tcl implementation of WindSh **task** commands.

$WIND_BASE/host/resource/tcl/shellUtilCmd.tcl
Tcl implementation of WindSh **util** commands.

$WIND_BASE/host/resource/tcl/shelcore.tcl
Tcl implementation of the shell core routines.

$WIND_BASE/host/resource/tcl/sh-arch.tcl
Target architecture-specific constants for shell Tcl implementation.

$WIND_BASE/host/resource/tcl/hostShowCore.tcl
Tcl implementation of the ***hostShow*( )** core routine.

$WIND_BASE/host/resource/tcl/icmpstatShowCore.tcl
Tcl implementation of the ***icmpstatShow*( )** core routine.

$WIND_BASE/host/resource/tcl/ifShowCore.tcl
Tcl implementation of the ***ifShow*( )** core routine.

$WIND_BASE/host/resource/tcl/inetstatShowCore.tcl
Tcl implementation of the ***inetstatShow*( )** core routine.

$WIND_BASE/host/resource/tcl/ipstatShowCore.tcl
Tcl implementation of the ***ipstatShow*( )** core routine.

$WIND_BASE/host/resource/tcl/routestatShowCore.tcl
Tcl implementation of the ***routestatShow*( )** core routine.

$WIND_BASE/host/resource/tcl/taskHookShowCore.tcl
Tcl implementation of the ***taskCreateHookShow*( )**, ***taskDeleteHookShow*( )**, ***taskSwitchHookShow*( )**, ***taskRegsShow*( )** core routines.

$WIND_BASE/host/resource/tcl/tcpstatShowCore.tcl
Tcl implementation of the ***tcpstatShow*( )** core routine.

$WIND_BASE/host/resource/tcl/tftpInfoShowCore.tcl
Tcl implementation of the ***tftpInfoShow*( )** core routine.

$WIND_BASE/host/resource/tcl/udpstatShowCore.tcl
Tcl implementation of the ***udpstatShow*( )** core routine.

SEE ALSO **tgtsvr**, **launch**, **browser**, *Tornado User's Guide*

Commands marked with (*) have no equivalents in the VxWorks shell.

agentModeShow( )

NAME ***agentModeShow*( )** – show the agent mode (*)

SYNOPSIS

```
int agentModeShow (void)
```

DESCRIPTION This command shows the mode of the target agent. There are two agent modes: *system* and *task*.

RETURNS N/A.

SEE ALSO **windsh**, ***sysResume*( )**, ***sysSuspend*( )**, *Tornado User's Guide: Shell*

b( )

NAME ***b*( )** – set or display breakpoints

SYNOPSIS

```
STATUS b
    (
    INSTR * addr, /* where to set brkpoint, or 0 = display all brkpoints */
    int     task  /* task for which to set brkpoint 0 = set all tasks */
    )
```

DESCRIPTION This command sets or displays breakpoints. To display the list of currently active breakpoints, call ***b*( )** without arguments:

```
-> b
```

The list shows the address, task, and pass count of each breakpoint.

To set a breakpoint with ***b*( )**, include the address, which can be specified numerically or symbolically with an optional offset. The task argument is optional:

```
-> b addr [,task ,count]]
```

If *task* is zero or omitted, the breakpoint will apply to all breakable tasks. If *count* is zero or omitted, the breakpoint will occur every time it is hit. If *count* is specified, the break will not occur until the *count* +1st time an eligible task hits the breakpoint (i.e., the breakpoint is ignored the first *count* times it is hit).

Individual tasks can be unbreakable, in which case breakpoints that otherwise would apply to a task are ignored. Tasks can be spawned unbreakable by specifying the task

option **VX_UNBREAKABLE**. Tasks can also be set unbreakable or breakable by resetting **VX_UNBREAKABLE** with the routine ***taskOptionsSet( )***.

When the agent is in external mode, the ***b( )*** command sets system breakpoints that will halt the kernel. The *task* argument is ignored.

RETURNS OK, or ERROR if *addr* is illegal or the breakpoint table is full.

SEE ALSO **windsh**, ***bd( )***, *Tornado User's Guide: Shell*

---

bd( )

NAME ***bd( )*** – delete a breakpoint

SYNOPSIS

```
STATUS bd
    (
    INSTR * addr, /* address of breakpoint to delete */
    int     task  /* task for which to delete brkpoint 0 = delete for all */
    )
```

DESCRIPTION This command deletes a specified breakpoint.

To execute, enter:

```
-> bd addr [,task]
```

If *task* is omitted or zero, the breakpoint will be removed for all tasks. If the breakpoint applies to all tasks, removing it for only a single task will be ineffective. It must be removed for all tasks and then set for just those tasks desired.

RETURNS OK, or ERROR if there is no breakpoint at the specified address.

SEE ALSO **windsh**, ***b( )***, ***bdall( )***, *Tornado User's Guide: Shell*

---

bdall( )

NAME ***bdall( )*** – delete all breakpoints

SYNOPSIS

```
STATUS bdall (void)
```

DESCRIPTION This routine removes all breakpoints.

To execute, enter:

```
-> bdall
```

All breakpoints for all tasks are removed.

RETURNS OK, always.

SEE ALSO **windsh**, ***bd( )***, *Tornado User's Guide: Shell*

bh( )

NAME ***bh( )*** – set a hardware breakpoint

SYNOPSIS

```
STATUS bh
    (
    INSTR * addr,  /* where to set brkpoint, or 0 = display all brkpoints */
    int     access /* access type (arch dependant) */
    )
```

DESCRIPTION This command is used to set a hardware breakpoint. If the architecture allows it, this function will add the breakpoint to the list of breakpoints and set the hardware breakpoint register(s). For more information, see the manual entry for ***b( )***.

NOTE: The types of hardware breakpoints vary with the architectures. Generally, a hardware breakpoint can be a data breakpoint or an instruction breakpoint.

RETURNS OK, or ERROR if *addr* is illegal or the breakpoint table is full or the hardware breakpoint breakpoint table is full.

SEE ALSO **windsh**, ***b( )***, ***bd( )***, *Tornado User's Guide: Shell*

bootChange( )

NAME ***bootChange( )*** – change the boot line

SYNOPSIS

```
STATUS bootChange (void)
```

DESCRIPTION This command changes the boot line used in the boot ROMs. After changing the boot parameters, you can reboot the target with the ***reboot*( )** command. When the system reboots, the shell is restarted automatically.

This command stores the new boot line in non-volatile RAM, if the target has it.

RETURNS OK, or ERROR if boot line not changed.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

browse( )

NAME ***browse*( )** – send a message to the browser asking it to browse an address (*)

SYNOPSIS

```
void browse
    (
    int objId /* system-object ID */
    )
```

DESCRIPTION Send a protocol message to the Tornado browser, requesting a display of the system object whose ID is specified by the argument *objId*. Browser displays are available for all the system objects recognized by ***show*( )**, but are more convenient because they can be updated automatically and scrolled independently.

If the browser is not executing in update mode, this WindSh primitive has no visible effect.

There is no target-resident version of ***browse*( )**.

RETURNS N/A

SEE ALSO **windsh**, ***show*( )**, **browser**, *Tornado User's Guide: Shell*

c( )

NAME ***c*( )** – continue from a breakpoint

SYNOPSIS

```
STATUS c
    (
    int task /* task that should proceed from breakpoint */
    )
```

DESCRIPTION This routine continues the execution of a task that has stopped at a breakpoint.

To execute, enter:

```
-> c [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

If the agent is in external mode, the system is resumed. In this case *task* is ignored.

RETURNS OK, or ERROR if the specified task does not exist.

SEE ALSO **windsh**, ***tr*( )**, *Tornado User's Guide: Shell*

cd( )

NAME ***cd*( )** – change the default directory

SYNOPSIS

```
STATUS cd
    (
    char * name /* new directory name */
    )
```

DESCRIPTION This command sets the default directory to *name* on the host where **windsh** is running.

To change to a different directory, specify one of the following:

– An entire path name. In Windows, the directory path must be prefixed with a drive name and colon.

– A directory name starting with any of the following; note that for Windows hosts, subdirectories can be separated with either a slash (/) or backslash (\):

```
UNIX:
~    /    ..    .
```

Windows: **\ /** **..** **.**

– A directory name to be appended to the current default directory.

EXAMPLE On a UNIX host, the following changes the directory to **~leslie/target/config**:

```
-> cd "~leslie/target/config"
```

On a Windows host, the following lines are equivalent and change the directory to **c:\leslie\target\config:**

```
-> cd "c:\\leslie\\target\\config"
-> cd "c:/leslie/target/config"
```

Notice that the rules for C strings require that backslashes be doubled.

RETURNS OK or ERROR.

SEE ALSO **windsh**, ***pwd*( )**, *Tornado User's Guide: Shell*

---

checkStack( )

NAME ***checkStack*( )** – print a summary of each task's stack usage

SYNOPSIS

```
void checkStack
    (
    int taskNameOrId /* task name or task ID; 0 = summarize all */
    )
```

DESCRIPTION This command displays a summary of stack usage for a specified task, or for all tasks if no argument is given. The summary includes the total stack size (SIZE), the current number of stack bytes used (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). For example:

```
-> checkStack t28
    NAME         ENTRY      TID    SIZE   CUR  HIGH  MARGIN
------------ ------------ -------- ----- ----- ----- ------
t28          _foo         23e1c78   9208   832  3632   5576
```

The maximum stack usage is determined by scanning down from the top of the stack for the first byte whose value is not 0xee. In VxWorks, when a task is spawned, all bytes of a task's stack are initialized to 0xee.

DEFICIENCIES It is possible for a task to write beyond the end of its stack, but not write into the last part of its stack. This will not be detected by ***checkStack*( )**.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

classShow( )

NAME ***classShow*( )** – show information about a class of objects (*)

SYNOPSIS

```
void classShow
    (
    int classID
    )
```

DESCRIPTION VxWorks kernel objects, such as semaphores, message queues, and so on, are organized into distinct classes. All objects of each class are anchored in a class ID stored as a global variable. Given any such class ID, ***classShow*( )** displays overall information about the class, including the maximum object size, and the number of objects allocated, deallocated, initialized, and terminated in that class. Because all the class ID globals are recorded using a consistent naming convention, you can obtain a list of the class IDs available at any time with the following:

```
-> lkup "ClassId"
```

There is no target-resident version of ***classShow*( )**.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

cplusCtors( )

NAME ***cplusCtors*( )** – call static constructors (C++)

SYNOPSIS

```
void cplusCtors
    (
    const char * moduleName /* name of loaded module */
    )
```

DESCRIPTION This function is used to call static constructors under the manual strategy (see ***cplusXtorSet*( )**). *moduleName* is the name of an object module that was *munched* before loading. If *moduleName* is 0, then all static constructors, in all modules loaded by the target server loader, are called.

EXAMPLES The following example shows how to initialize the static objects in modules called **applx.out** and **apply.out**.

```
-> cplusCtors "applx.out"
value = 0 = 0x0
-> cplusCtors "apply.out"
value = 0 = 0x0
```

The following example shows how to initialize all the static objects that are currently loaded, with a single invocation of ***cplusCtors*( )**:

```
-> cplusCtors
value = 0 = 0x0
```

RETURNS N/A

SEE ALSO **windsh**, ***cplusXtorSet*( )**, *Tornado User's Guide: Shell*

cplusDtors( )

NAME ***cplusDtors*( )** – call static destructors (C++)

SYNOPSIS

```
void cplusDtors
    (
    const char * moduleName /* name of loaded module */
    )
```

DESCRIPTION This function is used to call static destructors under the manual strategy (see ***cplusXtorSet( )***). *moduleName* is the name of an object module that was *munched* before loading. If *moduleName* is 0, then all static destructors, in all modules loaded by the target server loader, are called.

EXAMPLES The following example shows how to destroy the static objects in modules called **applx.out** and **apply.out**:

```
-> cplusDtors "applx.out"
value = 0 = 0x0
-> cplusDtors "apply.out"
value = 0 = 0x0
```

The following example shows how to destroy all the static objects that are currently loaded, with a single invocation of cplusDtors:

```
-> cplusDtors
value = 0 = 0x0
```

RETURNS N/A

SEE ALSO **windsh**, ***cplusXtorSet( )***, *Tornado User's Guide: Shell*

---

cplusStratShow( )

NAME ***cplusStratShow( )*** – show C++ static constructors calling strategy (*)

SYNOPSIS

```
void cplusStratShow (void)
```

DESCRIPTION This command shows the current C++ static constructor calling strategy. There are two static constructor calling strategies: *automatic* and *manual*.

```
-> cplusStratShow
C++ ctors/dtors strategy set to MANUAL
value = 0 = 0x0
```

RETURNS N/A

SEE ALSO **windsh**, ***cplusXtorsSet( )***, *Tornado User's Guide: Shell*

cplusXtorSet( )

NAME

***cplusXtorSet*( )** – change C++ static constructor calling strategy (C++)

SYNOPSIS

```
void cplusXtorSet
    (
    int strategy /* constructor calling strategy */
    )
```

DESCRIPTION

This command sets the C++ static constructor calling strategy to *strategy*. The default strategy is 0.

There are two static constructor calling strategies: *automatic* and *manual* represented by the following numeric codes:

Strategy	Code
manual	0
automatic	1

Under the manual strategy, a module's static constructors and destructors are called by ***cplusCtors*( )** and ***cplusDtors*( )**, which are themselves invoked manually.

Under the automatic strategy, a module's static constructors are called as a side-effect of loading the module using the target server loader. A module's static destructors are called as a side-effect of unloading the module.

RETURNS

N/A

SEE ALSO

windsh, ***cplusStratShow*( )**. ***cplusCtors*( )**, ***cplusDtors*( )**, *Tornado User's Guide: Shell*

cret( )

NAME

***cret*( )** – continue until the current subroutine returns

SYNOPSIS

```
STATUS cret
    (
    int task /* task to continue, 0 = default */
    )
```

DESCRIPTION

This routine places a breakpoint at the return address of the current subroutine of a specified task, then continues execution of that task.

To execute enter:

```
-> cret [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

When the breakpoint is hit, information about the task will be printed in the same format as in single stepping. The breakpoint is automatically removed when hit, or if the task hits another breakpoint first.

RETURNS OK or ERROR if there is no such task.

SEE ALSO **windsh**, ***so*( )**, *Tornado User's Guide: Shell*

---

d( )

NAME ***d*( )** – display memory

SYNOPSIS

```
void d
    (
    void * adrs,   /* address to display (if 0, display next block */
    int    nunits, /* number of units to print (if 0, use default) */
    int    width   /* width of displaying unit (1, 2, 4, 8) */
    )
```

DESCRIPTION This command displays the contents of memory, starting at *adrs*. If *adrs* is omitted or zero, ***d*( )** displays the next memory block, starting from where the last ***d*( )** command completed.

Memory is displayed in units specified by *width*. If *nunits* is omitted or zero, the number of units displayed defaults to last use. If *nunits* is non-zero, that number of units is displayed and that number then becomes the default. If *width* is omitted or zero, it defaults to the previous value. If *width* is an invalid number, it is set to 1. The valid values for *width* are 1, 2, 4, and 8. The number of units ***d*( )** displayed is rounded up to the nearest number of full lines.

RETURNS N/A

SEE ALSO **windsh**, ***m*( )**, *Tornado User's Guide: Shell*

devs( )

NAME *devs*( ) – list all system-known devices

SYNOPSIS
```
void devs (void)
```

DESCRIPTION This command displays a list of all devices known to the I/O system.

RETURNS N/A

SEE ALSO **windsh**, ***iosDevShow*( )**, *Tornado User's Guide: Shell*

h( )

NAME *h*( ) – display or set the size of shell history

SYNOPSIS
```
void h
    (
    int size /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION This command displays or sets the size of VxWorks shell history. If no argument is specified, shell history is displayed. If *size* is specified, that number of the most recent commands is saved for display. The value of *size* is initially 20.

RETURNS N/A

SEE ALSO **windsh**, ***shellHistory*( )**, *Tornado User's Guide: Shell*

help( )

NAME *help*( ) – print a synopsis of selected routines

SYNOPSIS
```
void help (void)
```

DESCRIPTION This command prints the following list of the calling sequences for commonly used routines.

```
help                        Print this list
h         [n]               Print (or set) shell history
i         [task]            Summary of tasks\xd5 TCBs
ti        task              Complete info on TCB for task
sp        adr,args...       Spawn a task, pri=100, opt=0, stk=20000
sps       adr,args...       Spawn a task, pri=100, opt=0, stk=20000
                            and leave it suspended
td        task              Delete a task
ts        task              Suspend a task
tr        task              Resume a task
d         [adr[,nunits[,width]]] Display memory
m         adr[,width]       Modify memory
mRegs     [reg[,task]]      Modify a task\xd5 s registers interactively
version                     Print VxWorks version info, and boot line
b                           Display breakpoints
b         addr[,task[,count]] Set breakpoint
bd        addr[,task]       Delete breakpoint
bdall     [task]            Delete all breakpoints
c         [task[,addr[,addr1]]] Continue from breakpoint
s         [task[,addr[,addr1]]] Single step
l         [adr[,nInst]]     List disassembled memory
tt        [task]            Do stack trace on task
bh        addr[,access[,task[,count]] Set hardware breakpoint
                              (if supported by the architecture)
devs                        List devices
cd        "path"            Set current working path
pwd                         Print working path
ls        ["path"[,long]]   List contents of directory
ld        [syms[,noAbort][,"name"]] Load stdin, or file, into memory
                                    (syms = add symbols to table:
                                    -1 = none, 0 = globals, 1 = all)
lkup      ["substr"]        List symbols in system symbol table
lkAddr    address           List symbol table entries near address
printErrno  value           Print the name of a status value
period    secs,adr,args...  Spawn task to call function periodically
repeat    n,adr,args...     Spawn task to call function n times (0=forever)
NOTE: Arguments specifying task can be either task ID or name.
```

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

hostShow( )

NAME ***hostShow*( )** – display the host table

SYNOPSIS
```
void hostShow (void)
```

DESCRIPTION This routine prints a list of remote hosts, along with their Internet addresses and aliases.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

i( )

NAME ***i*( )** – print a summary of each task's TCB, task by task

SYNOPSIS
```
void i
    (
    int taskNameOrId /* task name or task ID; 0 = summarize all */
    )
```

DESCRIPTION This command displays a synopsis of all tasks in the system, or a specified task if the argument is given. The ***ti*( )** command provides more complete information on a specific task.

Both ***i*( )** and ***ti*( )** use ***taskShow*( )**; see the documentation for ***taskShow*( )** for a description of the output format.

EXAMPLE
```
-> i
   NAME       ENTRY     TID    PRI   STATUS    PC       SP     ERRNO DELAY
---------- ---------- -------- --- ---------- ------- -------- ----- -----
tExcTask   _excTask    20fcb00   0 PEND      200c5fc  20fca6c     0     0
tLogTask   _logTask    20fb5b8   0 PEND      200c5fc  20fb520     0     0
tRlogind   _rlogind    20f3f90   2 PEND      2038614  20f3db0     0     0
tTelnetd   _telnetd    20f2124   2 PEND      2038614  20f2070     0     0
tNetTask   _netTask    20f7398  50 PEND      2038614  20f7340     0     0
value = 57 = 0x39 = \xd5 9\xd5
```

CAVEAT This command should be used only as a debugging aid, since the information is obsolete by the time it is displayed.

RETURNS N/A

SEE ALSO **windsh**, ***iStrict*( )**, ***ti*( )**, ***taskShow*( )**, *Tornado User's Guide: Shell*

icmpstatShow( )

NAME ***icmpstatShow*( )** – display statistics for ICMP

SYNOPSIS
```
void icmpstatShow (void)
```

DESCRIPTION This routine displays statistics for the ICMP (Internet Control Message Protocol) protocol.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

ifShow( )

NAME ***ifShow*( )** – display the attached network interfaces

SYNOPSIS
```
void ifShow
    (
    char * ifName /* name of the interface to show */
    )
```

DESCRIPTION This routine displays the attached network interfaces for debugging and diagnostic purposes. If *ifName* is given, only the interfaces belonging to that group are displayed. If *ifName* is omitted, all attached interfaces are displayed.

For each interface selected, the following are shown: Internet address, point-to-point peer address (if using SLIP), broadcast address, netmask, subnet mask, Ethernet address, route metric, maximum transfer unit, number of packets sent and received on this interface, number of input and output errors, and flags (such as loopback, point-to-point, broadcast, promiscuous, ARP, running, and debug).

EXAMPLE The following call displays all interfaces whose names begin with “ln”, (such as “ln0”, “ln1”, and “ln2”):

```
-> ifShow "ln"
```

The following call displays just the interface “ln0”:

```
-> ifShow "ln0"
```

RETURNS N/A

SEE ALSO **windsh**, *Tornado User’s Guide: Shell*

inetstatShow( )

NAME ***inetstatShow*( )** – display all active connections for Internet protocol sockets

SYNOPSIS

```
void inetstatShow (void)
```

DESCRIPTION This routine displays a list of all active Internet protocol sockets in a format similar to the UNIX **netstat** command.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User’s Guide: Shell*

intVecShow( )

NAME ***intVecShow*( )** – display the interrupt vector table

SYNOPSIS

```
void intVecShow
    (
    int vector /* interrupt vector number or -1 to display the whole */
               /* vector table */
    )
```

DESCRIPTION This routine displays information about the given vector or the whole interrupt vector table if vector is equal to -1.

RETURNS OK or ERROR.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

iosDevShow( )

NAME ***iosDevShow*( )** – display the list of devices in the system

SYNOPSIS
```
void iosDevShow (void)
```

DESCRIPTION This routine displays a list of all devices in the device list.

RETURNS N/A

SEE ALSO **windsh**, ***devs*( )**, *Tornado User's Guide: Shell*

iosDrvShow( )

NAME ***iosDrvShow*( )** – display a list of system drivers

SYNOPSIS
```
void iosDrvShow (void)
```

DESCRIPTION This routine displays a list of all drivers in the driver list.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

iosFdShow( )

NAME ***iosFdShow*( )** – display a list of file descriptor names in the system

SYNOPSIS
```
void iosFdShow (void)
```

DESCRIPTION This routine displays a list of all file descriptors in the system.

NOTE: Unlike the target-resident version of ***iosFdShow( )***, this routine does not mark file descriptors with the stdio devices they are attached to.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

ipstatShow( )

NAME ***ipstatShow( )*** – display IP statistics

SYNOPSIS

```
void ipstatShow
    (
    BOOL zero /* TRUE = reset statistics to 0 */
    )
```

DESCRIPTION This routine displays detailed statistics for the IP protocol.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

iStrict( )

NAME ***iStrict( )*** – print a summary of all task TCBs, as an atomic snapshot (*)

SYNOPSIS

```
void iStrict (void)
```

DESCRIPTION This command produces a display identical to that of ***i( )***, but guarantees consistency by retrieving all data in a single target transaction.

WARNING: This command should be used only as a debugging aid, since the information is obsolete by the time it is displayed.

RETURNS N/A

SEE ALSO **windsh**, ***i( )***, *Tornado User's Guide: Shell*

l( )

NAME *l*( ) – disassemble and display a specified number of instructions

SYNOPSIS

```
void l
    (
    INSTR * addr, /* address of first instruction to disassemble if 0, */
                  /* from the last instruction disassembled on the */
                  /* call to l */
    int     count /* number of instructions to disassemble */
                  /* if 0, use the same the last call to l */
    )
```

DESCRIPTION This routine disassembles a specified number of instructions and displays them on standard output. If the address of an instruction is entered in the system symbol table, the symbol will be displayed as a label for that instruction. Also, addresses in the **opcode** field of instructions will be displayed symbolically.

To execute, enter:

```
-> l [address [,count]]
```

If *address* is omitted or zero, disassembly continues from the previous address. If *count* is omitted or zero, the last specified count is used (initially 10). As with all values entered via the shell, the address may be typed symbolically.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

ld( )

NAME *ld*( ) – load an object module into memory

SYNOPSIS

```
MODULE_ID ld
    (
    int    syms, /* -1, 0, or 1 */
    BOOL   noOp, /* ignored */
    char * name  /* name of object module, NULL = standard input */
    )
```

DESCRIPTION This command loads an object module from a file or from standard input. The object module may be in any format for which the target server has an OMF reader. External references in the module are resolved during loading. The *syms* parameter determines how symbols are loaded; possible values are:

0 - Add global symbols to the system symbol table.
1 - Add global and local symbols to the system symbol table.
-1 - Add no symbols to the system symbol table.

The second parameter *noOp* is present for compatibility with the target-resident version of ***ld( )***, but is not used by this implementation.

During load operation (progress indicator moving), a CTRL+C call cancels the current load, and unloads the module.

Errors during loading (e.g., externals undefined, too many symbols, etc.) are ignored.

The normal way of using ***ld( )*** is to load all symbols (*syms* = 1) during debugging and to load only global symbols later.

EXAMPLE The following example loads **test.o** with all symbols:

```
-> ld 1,0,"/usr/someone/devt/test.o"
```

RETURNS **MODULE_ID**, or NULL if there are too many symbols, the object file format is invalid, or there is an error reading the file.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

lkAddr( )

NAME ***lkAddr( )*** – list symbols whose values are near a specified value

SYNOPSIS

```
void lkAddr
    (
    unsigned int addr /* address around which to look */
    )
```

DESCRIPTION This command lists the symbols in the system symbol table that are near a specified value. The symbols that are displayed include:

– symbols whose values are immediately less than the specified value

– symbols with the specified value

– succeeding symbols, until at least 12 symbols have been displayed

This command also displays symbols that are local, i.e., symbols found in the system symbol table only because their module was loaded by ***ld*( )**.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

lkup( )

NAME ***lkup*( )** – list symbols

SYNOPSIS

```
void lkup
    (
    char * substr /* substring to match */
    )
```

DESCRIPTION This command lists all symbols in the system symbol table whose names contain the string *substr*. If *substr* is omitted or is 0, a short summary of symbol table statistics is displayed. If *substr* is the empty string (""), all symbols in the table are listed.

This command also displays symbols that are local, i.e., symbols found in the system symbol table only because their module was loaded by ***ld*( )**.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

ls( )

NAME ***ls*( )** – list the contents of a directory

SYNOPSIS

```
STATUS ls
    (
    char * dirName, /* name of dir to list */
    BOOL   doLong   /* if TRUE, do long listing */
    )
```

DESCRIPTION This command lists the contents of a directory in one of two formats. If *doLong* is FALSE, only the names of the files (or subdirectories) in the specified directory are displayed. If *doLong* is TRUE, then the file name, size, date, and time are displayed.

The *dirName* parameter specifies which directory to list. If *dirName* is omitted or NULL, the current working directory is listed.

RETURNS OK or ERROR.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

m( )

NAME *m*( ) – modify memory

SYNOPSIS

```
void m
    (
    void * adrs, /* address to change */
    int    width /* width of unit to be modified (1, 2, 4, 8) */
    )
```

DESCRIPTION This command prompts the user for modifications to memory in byte, short word, or long word specified by *width*, starting at the specified address. It displays each address and the current contents of that address, in turn. If *adrs* or *width* is zero or absent, it defaults to the previous value. The user can respond in one of several ways:

RETURN
: Do not change this address, but continue, prompting at the next address.

number
: Set the content of this address to *number*.

. (dot)
: Do not change this address, and quit.

EOF
: Do not change this address, and quit.

All numbers entered and displayed are in hexadecimal.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

memPartShow( )

NAME ***memPartShow*( )** – show partition blocks and statistics

SYNOPSIS

```
STATUS memPartShow
    (
    PART_ID partId, /* partition ID */
    int     type    /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION This command displays statistics about the available and allocated memory in a specified memory partition. It shows the number of bytes, the number of blocks, and the average block size in both free and allocated memory, and also the maximum block size of free memory. It also shows the number of blocks currently allocated and the average allocated block size.

In addition, if *type* is 1, the command displays a list of all the blocks in the free list of the specified partition.

RETURNS OK or ERROR.

SEE ALSO **windsh**, ***memShow*( )**, ***browse*( )**, *Tornado User's Guide: Shell*

memShow( )

NAME ***memShow*( )** – show system memory partition blocks and statistics

SYNOPSIS

```
void memShow
    (
    int type /* 1 = list all blocks in the free list */
    )
```

DESCRIPTION This command displays statistics about the available and allocated memory in the system memory partition. It shows the number of bytes, the number of blocks, and the average block size in both free and allocated memory, and also the maximum block size of free memory. It also shows the number of blocks currently allocated and the average allocated block size.

In addition, if *type* is 1, the command displays a list of all the blocks in the free list of the system partition.

EXAMPLE

```
-> memShow 1
FREE LIST:
  num     addr      size
  --- ---------- ----------
    1   0x3fee18         16
    2   0x3b1434         20
    3    0x4d188    2909400
SUMMARY:
 status   bytes    blocks   avg block  max block
 ------ --------- -------- ---------- ----------
current
   free   2909436        3     969812   2909400
  alloc    969060    16102         60         -
cumulative
  alloc   1143340    16365         69         -
```

RETURNS N/A

SEE ALSO **windsh**, ***memPartShow*( )**, *Tornado User's Guide: Shell*

---

moduleIdFigure( )

NAME ***moduleIdFigure*( )** – figure out module ID, given name or number (*)

SYNOPSIS

```
int moduleIdFigure
    (
    int modNameOrId /* target module ID or name */
    )
```

DESCRIPTION The list of module IDs known to the target server is searched for the given number; if found, it is returned (thus verifying its validity). Otherwise, if the given address points to a string that matches an existing module name, that module's ID is returned. If all this fails, ERROR (-1) is returned.

There is no target-resident version of ***moduleIdFigure*( )**.

RETURNS A module ID, or ERROR.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

moduleShow( )

NAME ***moduleShow*( )** – show the current status for all the loaded modules

SYNOPSIS

```
STATUS moduleShow
    (
    char * moduleNameOrId /* name or ID of the module to show */
    )
```

DESCRIPTION This command displays a list of the currently loaded modules and some information about where the modules are loaded.

The specific information displayed depends on the format of the object modules. In the case of a.out and ECOFF object modules, ***moduleShow*( )** displays the start of the text, data, and BSS segments.

If ***moduleShow*( )** is called with no arguments, a summary list of all loaded modules is displayed. It can also be called with an argument, *moduleNameOrId*, which can be either the name of a loaded module or a module ID. If it is called with either of these, more information about the specified module will be displayed.

RETURNS OK or ERROR.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

mqPxShow( )

NAME ***mqPxShow*( )** – show information about a POSIX message queue (*)

SYNOPSIS

```
STATUS mqPxShow
    (
    mqd_t mqDesc /* POSIX message queue to display */
    )
```

DESCRIPTION This command displays the state of a POSIX message queue.

A summary of the state of the message queue is displayed as follows:

```
Message queue name          : exampleMessageQueue
No. of messages in queue    : 2
Maximum no. of messages     : 16
Maximum message size        : 16
```

RETURNS OK or ERROR.

SEE ALSO **windsh**, ***show*( )**, ***browse*( )**, *Tornado User's Guide: Shell*

mRegs( )

NAME ***mRegs*( )** – modify registers

SYNOPSIS

```
STATUS mRegs
    (
    char * regName,     /* register name, NULL for all */
    int    taskNameOrId /* task name or task ID, 0 = default task */
    )
```

DESCRIPTION This command modifies the specified register for the specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed. If the specified register is not found, it prints out the valid register list and returns ERROR. If no register is specified, it sequentially prompts the user for new values for a task's registers. It displays each register and the current contents of that register, in turn. The user can respond in one of several ways:

RETURN
: Do not change this register, but continue, prompting at the next register.

number
: Set this register to *number*.

. (dot)
: Do not change this register, and quit.

EOF
: Do not change this register, and quit.

All numbers are entered and displayed in hexadecimal, except floating-point values, which may be entered in double precision.

RETURNS OK, or ERROR if the task or register does not exist.

SEE ALSO **windsh**, ***m*( )**, *Tornado User's Guide: Shell*

msgQShow( )

NAME

msgQShow( ) – show information about a message queue

SYNOPSIS

```
STATUS msgQShow
    (
    MSG_Q_ID msgQId, /* message queue to display */
    int      level   /* 0 = summary, 1 = details */
    )
```

DESCRIPTION

This command displays the state and optionally the contents of a message queue.

A summary of the state of the message queue is displayed as follows:

```
Message Queue Id    : 0x3f8c20
Task Queuing        : FIFO
Message Byte Len    : 150
Messages Max        : 50
Messages Queued     : 0
Receivers Blocked   : 1
Send timeouts       : 0
Receive timeouts    : 0
```

If *level* is 1, then more detailed information will be displayed. If messages are queued, they will be displayed as follows:

```
Messages queued:
 # local adrs  len  value
 1 0x3d7c0c     14  00 3c a1 e1 48 65 6c 6c 6f 31 00 ee ee ee
*....Hello.....*
```

If tasks are blocked on the queue, they will be displayed as follows:

```
Receivers blocked:
   NAME      TID    PRI DELAY
---------- -------- --- -----
tExcTask     3fd678   0   21
```

RETURNS

OK or ERROR.

SEE ALSO

windsh, ***browse*( )**, *Tornado User's Guide: Shell*

period( )

NAME *period*( ) – spawn a task to call a function periodically

SYNOPSIS

```
int period
    (
    int     secs, /* period in seconds */
    FUNCPTR func, /* function to call repeatedly */
    int     arg1, /* first of eight args to pass to func */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This command spawns a task that repeatedly calls a specified target-resident function, with up to eight of its arguments, delaying the specified number of seconds between calls.

NOTE: The task is spawned using the ***sp*( )** command. See the description of ***sp*( )** for details about priority, options, stack size, and task ID.

RETURNS A task ID, or ERROR if the task cannot be spawned.

SEE ALSO **windsh**, ***sp*( )**, *Tornado User's Guide: Shell*

printErrno( )

NAME *printErrno*( ) – print the definition of a specified error status value

SYNOPSIS

```
void printErrno
    (
    int errNo /* status code whose name is to be printed */
    )
```

DESCRIPTION This command displays the error-status string, corresponding to a specified error-status value. It is only useful if the error-status symbol table has been built and included in the system. An *errNo* equal to zero will display the error number set by the last function called from WindSh.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

printLogo( )

NAME ***printLogo*( )** – display the Tornado logo

SYNOPSIS
```
void printLogo (void)
```

DESCRIPTION This command displays the Tornado banner seen when WindSh begins executing. It also displays the Tornado version number.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

pwd( )

NAME ***pwd*( )** – display the current default directory

SYNOPSIS
```
void pwd (void)
```

DESCRIPTION This command displays the current working device/directory as known to WindSh.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

quit( )

NAME ***quit*( )** – shut down WindSh (*)

SYNOPSIS
```
void quit (void)
```

DESCRIPTION Shut down the executing session of WindSh.

There is no target-resident version of ***quit*( )**.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

reboot( )

NAME ***reboot*( )** – reset network devices and transfer control to boot ROMs

SYNOPSIS

```
void reboot (void)
```

DESCRIPTION This WindSh primitive sends a restart message to the target server, which in turn passes it on to the target agent. This causes the target to reboot, and the target server to reattach. After waiting one second, WindSh also restarts.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

repeat( )

NAME ***repeat*( )** – spawn a task to call a function repeatedly

SYNOPSIS

```
int repeat
    (
    int     n,    /* no. of times to call func (0=forever) */
    FUNCPTR func, /* function to call repeatedly */
    int     arg1, /* first of eight args to pass to func */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8
    )
```

DESCRIPTION This command spawns a task that calls a specified function *n* times, with up to eight of its arguments. If *n* is 0, the command is called endlessly, or until the spawned task is deleted.

NOTE: The task is spawned using ***sp*( )**. See the description of ***sp*( )** for details about priority, options, stack size, and task ID.

RETURNS A task ID, or ERROR if the task cannot be spawned.

SEE ALSO **windsh**, ***sp*( )**, *Tornado User's Guide: Shell*

---

routestatShow( )

NAME ***routestatShow*( )** – display routing statistics

SYNOPSIS

```
void routestatShow (void)
```

DESCRIPTION This routine displays routing statistics.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

s( )

NAME ***s*( )** – single-step a task

SYNOPSIS

```
STATUS s
    (
    int     taskNameOrId, /* task to step; 0 = use default */
    INSTR * addr0,        /* lower bound of range; 0 = next instr */
    INSTR * addr1         /* ;upper bound of range; 0 = next instr */
    )
```

DESCRIPTION This command single-steps a task that is stopped at a breakpoint.

To execute, enter:

```
-> s [task [,addr0[,addr1]]]
```

If *task* is omitted or zero, the last task referenced is assumed. If *addr0* is non-zero, the task begins stepping at the next instruction at or above *addr0*; if *addr1* is non-zero, the task next suspends at the next instruction at or above *addr1*.

If the agent is in system mode, the system context will be stepped. In this event, the *task* parameter is ignored.

RETURNS OK, or ERROR if the task cannot be found or the task is not suspended.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

semPxShow( )

NAME ***semPxShow*( )** – show information about a POSIX semaphore (*)

SYNOPSIS

```
STATUS semPxShow
    (
    sem_t semDesc /* POSIX semaphore to display */
    )
```

DESCRIPTION This command displays the state of a POSIX semaphore.

A summary of the state of a named semaphore is displayed as follows:

```
Semaphore name                : namedSem
sem_open() count              : 2
Semaphore value               : 0
No. of blocked tasks          : 1
```

A summary of the state of an unnamed semaphore is displayed as follows:

```
Semaphore value               : 0
No. of blocked tasks          : 1
```

RETURNS OK or ERROR.

SEE ALSO **windsh**, ***show*( )**, ***browse*( )**, *Tornado User's Guide: Shell*

semShow( )

NAME

***semShow*( )** – show information about a semaphore

SYNOPSIS

```
STATUS semShow
    (
    SEM_ID semId, /* semaphore to display */
    int    level  /* 0 = summary, 1 = details */
    )
```

DESCRIPTION

This command displays the state and optionally the pended tasks of a semaphore.

A summary of the state of the semaphore is displayed as follows:

```
Semaphore Id        : 0x585f2
Semaphore Type      : BINARY
Task Queuing        : PRIORITY
Pended Tasks        : 1
State               : EMPTY {Count if COUNTING, Owner if MUTEX}
```

If tasks are blocked on the queue, they are displayed in the order in which they will unblock, as follows:

```
   NAME      TID    PRI DELAY
---------- -------- --- -----
tExcTask    3fd678   0    21
tLogTask    3f8ac0   0   611
```

RETURNS

OK or ERROR.

SEE ALSO

windsh, ***browse*( )**, *Tornado User's Guide: Shell*

shellHistory( )

NAME

***shellHistory*( )** – display or set the size of shell history

SYNOPSIS

```
void shellHistory
    (
    int size /* 0 = display, >0 = set history to new size */
    )
```

DESCRIPTION This command displays shell history, or resets the default number of commands displayed by shell history to *size*. By default, history size is 20 commands.

RETURNS N/A

SEE ALSO **windsh**, ***h*( )**, *Tornado User's Guide: Shell*

shellPromptSet( )

NAME ***shellPromptSet*( )** – change the shell prompt

SYNOPSIS

```
void shellPromptSet
    (
    char * newPrompt /* string to become new shell prompt */
    )
```

DESCRIPTION This command changes the shell prompt string to *newPrompt*.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

show( )

NAME ***show*( )** – display information on a specified object

SYNOPSIS

```
void show
    (
    int objId, /* object ID */
    int level  /* information level */
    )
```

DESCRIPTION This command displays information on the specified object. System objects include tasks, semaphores, message queues, shared semaphores, shared message queues, memory partitions, and watchdogs. An information level is interpreted by the object's show routine on a class-by-class basis.

RETURNS N/A

SEE ALSO **windsh**, ***memPartShow*( )**, ***msgQShow*( )**, ***semShow*( )**, ***taskShow*( )**, ***wdShow*( )**, *Tornado User's Guide: Shell*

smMemPartShow( )

NAME ***smMemPartShow*( )** – show user shared memory system partition blocks and statistics (*)

SYNOPSIS

```
STATUS smMemPartShow
    (
    SM_PART_ID partId, /* global partition id to use */
    int        type    /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION For a specified shared partition, this routine displays the total amount of free space in the partition, the number of blocks, the average block size, and the maximum block size. It also shows the number of blocks currently allocated, and the average allocated block size.

In addition, if *type* is 1, this routine displays a list of all the blocks in the free list of the specified shared partition.

RETURNS The message "VxMP component not installed" if VxMP is not present in the target system.

SEE ALSO **windsh**, ***smMemShow*( )**, *Tornado User's Guide: Shell*

smMemShow( )

NAME ***smMemShow*( )** – show the shared memory system partition blocks and statistics

SYNOPSIS

```
void smMemShow
    (
    int type /* 0 = statistics, 1 = statistics & list */
    )
```

DESCRIPTION This command displays the total amount of free space in the shared memory system partition, including the number of blocks, the average block size, and the maximum block size. It also shows the number of blocks currently allocated, and the average allocated block size.

If *type* is 1, it displays a list of all the blocks in the free list of the shared memory system partition.

EXAMPLE

```
-> smMemShow 1
FREE LIST:
  num    addr       size
  --- ---------- ----------
    1   0x4ffef0        264
    2   0x4fef18       1700
SUMMARY:
    status        bytes    blocks   ave block  max block
-------------- --------- -------- ---------- ----------
        current
           free      1964        2        982       1700
          alloc      2356        1       2356          -
     cumulative
          alloc      2620        2       1310          -
value = 0 = 0x0
```

RETURNS The message "VxMP component not installed" if VxMP is not present in the target system.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

so( )

NAME *so*( ) – single-step, but step over a subroutine

SYNOPSIS

```
STATUS so
    (
    int task /* task to step; 0 = use default */
    )
```

DESCRIPTION This routine single-steps a task that is stopped at a breakpoint. However, if the next instruction is a JSR or BSR, ***so*( )** breaks at the instruction following the subroutine call instead.

To execute, enter:

```
-> so [task]
```

If *task* is omitted or zero, the last task referenced is assumed.

RETURNS OK or ERROR if there is no such task.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

sp( )

NAME

sp( ) – spawn a task with default parameters

SYNOPSIS

```
int sp
    (
    FUNCPTR func, /* function to call */
    int     arg1, /* first of nine args to pass to spawned task */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8,
    int     arg9
    )
```

DESCRIPTION

This command spawns a specified function as a task with the following defaults:

priority:
100

stack size:
20,000 bytes

task ID:
highest not currently used

task options:
VX_FP_TASK - execute with floating-point coprocessor support.

task name:
A name of the form **sXuN** where X is the shell number and N is an integer which increments as new tasks are spawned. The shell number depends on the number of connected shells and on the order of connection. First connected shell has a 1 shell number, then 2 3 4 ... If shell 2 disconnects, and another one connects, its shell number will be 2 (first free slot number). Thus task names should look like **s1u1**, **s1u2**, **s5u3**, etc.

The task ID is displayed after the task is spawned.

This command can also be used to spawn a task when the agent is in external mode. The new task is created in the context of tExcTask, and will not start running until that task has processed the spawn request. For a task started this way, only four arguments may be given.

RETURNS A task ID, or ERROR if the task cannot be spawned.

SEE ALSO **windsh**, ***sps*( )**, *Tornado User's Guide: Shell*

sps( )

NAME ***sps*( )** – spawn a task with default parameters, and leave it suspended (*)

SYNOPSIS

```
int sps
    (
    FUNCPTR func, /* function to call */
    int     arg1, /* first of nine args to pass to spawned task */
    int     arg2,
    int     arg3,
    int     arg4,
    int     arg5,
    int     arg6,
    int     arg7,
    int     arg8,
    int     arg9
    )
```

DESCRIPTION This command has the same effect as ***sp*( )**, except that the newly spawned task is immediately suspended.

There is no target-resident version of ***sps*( )**.

RETURNS A task ID, or ERROR if the task cannot be spawned.

SEE ALSO **windsh**, ***sp*( )**, *Tornado User's Guide: Shell*

sysResume( )

NAME ***sysResume*( )** – reset the agent to tasking mode (*)

SYNOPSIS

```
int sysResume (void)
```

DESCRIPTION This command sets the agent to tasking mode and resumes the system. If the agent is already in tasking mode, ***sysResume*( )** has no effect.

RETURNS OK or ERROR.

SEE ALSO **windsh**, ***sysSuspend*( )**, *Tornado User's Guide: Shell*

sysStatusShow( )

NAME ***sysStatusShow*( )** – show system context status (*)

SYNOPSIS
```
int sysStatusShow (void)
```

DESCRIPTION This command shows the status of the system context. There are two system context states: *suspended* and *running*. This command can be completed successfully only if the agent is running in external mode.

RETURNS OK, or ERROR if agent is running in task mode

SEE ALSO **windsh**, ***sysResume*( )**, ***sysSuspend*( )**, *Tornado User's Guide: Shell*

sysSuspend( )

NAME ***sysSuspend*( )** – set the agent to external mode and suspend the system (*)

SYNOPSIS
```
int sysSuspend (void)
```

DESCRIPTION This command sets the agent to external mode if it is supported by the agent. The system is then suspended, halting all tasks. When the agent is in external mode, certain shell commands work differently: ***b*( )**, ***s*( )**, ***c*( )** work with the system context instead of particular task contexts. To return to tasking mode, use ***sysResume*( )**.

RETURNS OK, or ERROR if external mode cannot be entered.

SEE ALSO **windsh**, ***sysResume*( )**, ***b*( )**, *Tornado User's Guide: Shell*

taskCreateHookShow( )

NAME ***taskCreateHookShow*( )** – show the list of task create routines

SYNOPSIS
```
void taskCreateHookShow (void)
```

DESCRIPTION This routine shows all the task create routines installed in the task create hook table, in the order in which they were installed.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

taskDeleteHookShow( )

NAME ***taskDeleteHookShow*( )** – show the list of task delete routines

SYNOPSIS
```
void taskDeleteHookShow (void)
```

DESCRIPTION This routine shows all the delete routines installed in the task delete hook table, in the order in which they were installed. Note that the delete routines will be run in reverse of the order in which they were installed.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

taskIdDefault( )

NAME ***taskIdDefault*( )** – set the default task ID

SYNOPSIS
```
int taskIdDefault
    (
    int tid /* user-supplied task ID; if 0, return default */
    )
```

DESCRIPTION This command maintains a global default task ID. This ID is used by WindSh primitives that allow a task ID argument to take on a default value if you do not explicitly supply one.

If *tid* is not zero (i.e., you do specify a task ID), the default ID is set to that value, and that value is returned. If *tid* is zero (i.e., you did not specify a task ID), the default ID is not changed and its value is returned. Thus the value returned is always the last default task ID you specify.

RETURNS The most recent non-zero task ID.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

---

taskIdFigure( )

NAME ***taskIdFigure*( )** – figure out the task ID of a specified task (*)

SYNOPSIS

```
int taskIdFigure
    (
    char * nameOrId /* target task name or Identifier */
    )
```

DESCRIPTION This command returns the task ID of a task specified either by name or by number. If *nameOrId* is 0, it returns the task ID of the default task. Otherwise, it searches the list of active task IDs for the given number; if found, it returns the ID (thus verifying its validity). Next, it searches a task whose name is the number (in hex); if found, it returns the ID. Otherwise, if the given address points to a string that matches an existing task name, it returns the task's ID. If all this fails, the command returns ERROR.

There is no target-resident version of ***taskIdFigure*( )**.

RETURNS A task ID, or ERROR.

SEE ALSO **windsh**, ***taskIdDefault*( )**, *Tornado User's Guide: Shell*

taskRegsShow( )

NAME ***taskRegsShow*( )** – display the contents of a task's registers

SYNOPSIS

```
void taskRegsShow
    (
    int tid /* task ID */
    )
```

DESCRIPTION This routine displays the register contents of a specified task on standard output.

EXAMPLE The following example displays the register of the shell task (68000 family):

```
-> taskRegsShow (taskNameToId ("tShell"))
d0 = 0       d1 = 0       d2 = 578fe   d3 = 1
d4 = 3e84e1  d5 = 3e8568  d6 = 0       d7 = ffffffff
a0 = 0       a1 = 0       a2 = 4f06c   a3 = 578d0
a4 = 3fffc4  a5 = 0       fp = 3e844c  sp = 3e842c
sr = 3000    pc = 4f0f2
value = 0 = 0x0
```

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

taskShow( )

NAME ***taskShow*( )** – display task information from TCBs

SYNOPSIS

```
STATUS taskShow
    (
    int tid,  /* task ID */
    int level /* 0 = summary, 1 = details, 2 = all tasks */
    )
```

DESCRIPTION This command displays the contents of a task control block (TCB) for a specified task. If *level* is 1, it also displays task options and registers. If *level* is 2, it displays all tasks.

The TCB display contains the following fields:

Field	Meaning
NAME	Task name
ENTRY	Symbol name or address where task began execution
TID	Task ID
PRI	Priority
STATUS	Task status, as formatted by ***taskStatusString*( )**
PC	Program counter
SP	Stack pointer
ERRNO	Most recent error code for this task
DELAY	If task is delayed, number of clock ticks remaining in delay (0 otherwise)

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

taskSwitchHookShow( )

NAME ***taskSwitchHookShow*( )** – show the list of task switch routines

SYNOPSIS
```
void taskSwitchHookShow (void)
```

DESCRIPTION This routine shows all the switch routines installed in the task switch hook table, in the order in which they were installed.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

taskWaitShow( )

NAME ***taskWaitShow*( )** – show information about the object a task is pended on (*)

SYNOPSIS
```
void taskWaitShow
    (
    int taskId, /* task ID */
```

```
int level   /* 0 = summary, 1 = details */
)
```

DESCRIPTION This routine shows information about the object a task is pended on. This routine doesn't support POSIX semaphores and message queues. This command doesn't support pending signals.

List of object types:

semaphores:
- **SEM_B**: binary
- **SEM_M**: mutex
- **SEM_C**: counting
- **SEM_O**: old
- **SEM_SB**: shared_binary
- **SEM_SC**: shared counting

message queues:
- **MSG_Q**(R): task is pended on a **msgQReceive** command
- **MSG_Q**(S): task is pended on a **msgQSend** command

shared message queues:
- **MSG_Q_S**(R): task is pended on a msgQReceive command
- **MSG_Q_S**(S): task is pended on a msgQSend command

unknown object:
- N/A: The task is pended on an unknown object (POSIX, signal).

RETURNS N/A

SEE ALSO **windsh**, ***w*( )**, ***tw*( )**, *Tornado User's Guide: Shell*

---

tcpstatShow( )

NAME ***tcpstatShow*( )** – display all statistics for the TCP protocol

SYNOPSIS
```
void tcpstatShow (void)
```

DESCRIPTION This routine displays detailed statistics for the TCP protocol.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

td( )

NAME ***td*( )** – delete a task

SYNOPSIS

```
void td (void)
```

DESCRIPTION This command deletes a specified task. It is equivalent to the target routine ***taskDelete*( )**.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

tftpInfoShow( )

NAME ***tftpInfoShow*( )** – get TFTP status information

SYNOPSIS

```
void tftpInfoShow
    (
    TFTP_DESC * pTftpDesc /* TFTP descriptor */
    )
```

DESCRIPTION This routine prints information associated with TFTP descriptor *pTftpDesc*.

EXAMPLE A call to ***tftpInfoShow*( )** might look like:

```
-> tftpInfoShow (tftpDesc)
       Connected to yuba [69]
       Mode: netascii  Verbose: off  Tracing: off
       Rexmt-interval: 5 seconds, Max-timeout: 25 seconds
value = 0 = 0x0
->
```

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

ti( )

NAME ***ti*( )** – display complete information from a task's TCB

SYNOPSIS

```
void ti
    (
    int taskNameOrId /* task name or task ID; 0 = use default */
    )
```

DESCRIPTION This command displays the task control block (TCB) contents, including registers, for a specified task. If *taskNameOrId* is omitted or zero, the last task referenced is assumed.

The ***ti*( )** command calls ***taskShow*( )** with a second argument of 1; see the documentation for ***taskShow*( )** for a description of the output format.

RETURNS N/A

SEE ALSO **windsh**, ***taskShow*( )**, *Tornado User's Guide: Shell*

tr( )

NAME ***tr*( )** – resume a task

SYNOPSIS

```
void tr
    (
    int taskNameOrId /* task name or task ID */
    )
```

DESCRIPTION This command resumes the execution of a suspended task. It is equivalent to the target routine ***taskResume*( )**.

RETURNS N/A

SEE ALSO **windsh**, ***ts*( )**, *Tornado User's Guide: Shell*

ts( )

NAME ***ts*( )** – suspend a task

SYNOPSIS

```
void ts
    (
    int taskNameOrId /* task name or task ID */
    )
```

DESCRIPTION This command suspends the execution of a specified task. It is equivalent to the target routine ***taskSuspend*( )**.

RETURNS N/A

SEE ALSO **windsh**, ***tr*( )**, *Tornado User's Guide: Shell*

tt( )

NAME ***tt*( )** – display a stack trace of a task

SYNOPSIS

```
STATUS tt
    (
    int taskNameOrId /* task name or task ID */
    )
```

DESCRIPTION This command displays a list of the nested routine calls that the specified task is in. Each routine call and its parameters are shown.

If *taskNameOrId* is not specified or zero, the last task referenced is assumed.

EXAMPLE

```
-> tt "logTask"
 3ab92 _vxTaskEntry   +10 : _logTask (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
  ee6e _logTask       +12 : _read (5, 3f8a10, 20)
  d460 _read          +10 : _iosRead (5, 3f8a10, 20)
  e234 _iosRead       +9c : _pipeRead (3fce1c, 3f8a10, 20)
 23978 _pipeRead      +24 : _semTake (3f8b78)
value = 0 = 0x0
```

This indicates that ***logTask*( )** is currently in ***semTake*( )** (with one parameter) and was called by ***pipeRead*( )** (with three parameters), which was called by ***iosRead*( )** (with three parameters), and so on.

WARNING: In order to do the trace, some assumptions are made. In general, the trace works for all C language routines and for assembly language routines that start with a LINK instruction. Some C compilers require specific flags to generate the LINK first. Most VxWorks assembly language routines include LINK instructions for this reason. The trace facility may produce inaccurate results or fail completely if the routine is written in a language other than C, the routine's entry point is non-standard, or the task's stack is corrupted. Also, all parameters are assumed to be 32-bit quantities, so structures passed as parameters will be displayed as long integers.

RETURNS OK, or ERROR if the task does not exist.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

tw( )

NAME ***tw*( )** – print info about the object the given task is pending on (*)

SYNOPSIS

```
void tw
    (
    int taskNameOrId /* task name or task ID */
    )
```

DESCRIPTION This routine calls ***taskWaitShow*( )** on the given task in verbose mode.

RETURNS N/A

SEE ALSO **windsh**, ***taskWaitShow*( )**, ***w*( )**, *Tornado User's Guide: Shell*

udpstatShow( )

NAME ***udpstatShow*( )** – display statistics for the UDP protocol

SYNOPSIS

```
void udpstatShow (void)
```

DESCRIPTION This routine displays statistics for the UDP protocol.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

unld( )

NAME ***unld*( )** – unload an object module by specifying a file name or module ID

SYNOPSIS

```
STATUS unld
    (
    void * nameOrId /* name or ID of the object module file */
    )
```

DESCRIPTION This command unloads the specified object module from the system. The module can be specified by name or by module ID. For a.out and ECOFF format modules, unloading does the following:

(1) It frees the space allocated for text, data, and BSS segments, unless ***loadModuleAt*( )** was called with specific addresses, in which case the user is responsible for freeing the space.

(2) It removes all symbols associated with the object module from the system symbol table.

(3) It removes the module descriptor from the module list.

For other modules of other formats, unloading has similar effects.

RETURNS OK or ERROR.

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

version( )

NAME ***version*( )** – print VxWorks version information

SYNOPSIS

```
void version (void)
```

DESCRIPTION This command displays the VxWorks release number and architecture type, the *wind* kernel version number, the date this copy of VxWorks was created, and the boot parameters for a VxWorks system running on the current target.

RETURNS N/A

SEE ALSO **windsh**, *Tornado User's Guide: Shell*

w( )

NAME *w*( ) – print a summary of each task's pending information, task by task (*)

SYNOPSIS
```
void w
    (
    int taskNameOrId /* task name or task ID */
    )
```

DESCRIPTION This routine calls ***taskWaitShow*( )** in quiet mode on all tasks in the system, or a specified task if the argument is given.

RETURNS N/A

SEE ALSO **windsh**, ***taskWaitShow*( )**, ***tw*( )**, *Tornado User's Guide: Shell*

wdShow( )

NAME ***wdShow*( )** – show information about a watchdog

SYNOPSIS
```
STATUS wdShow
    (
    WDOG_ID wdId /* watchdog to display */
    )
```

DESCRIPTION This command displays the state of a watchdog.

EXAMPLE
```
    -> wdShow myWdId
    Watchdog Id         : 0x3dd46c
    State               : OUT_OF_Q
    Ticks Remaining     : 0
    Routine             : 0
    Parameter           : 0
```

RETURNS OK or ERROR.

SEE ALSO **windsh**, ***browse*( )**, *Tornado User's Guide: Shell*

wtxCtest

NAME **wtxCtest** – test suite for the WTX C API

SYNOPSIS

```
wtxCtest targetServerName
```

DESCRIPTION This program simply calls each function of the WTX C API, and verifies that the result is valid.

wtxreg

NAME **wtxreg** – report information about target servers known to a registry

SYNOPSIS

```
wtxreg [-H registryHost] [-r] [targetServerName...]
```

DESCRIPTION With no arguments, **wtxreg** prints a one-line summary about each target server known to the registry that is running on the default registry host (the host identified by the **WIND_REGISTRY** environment variable). The **-H** flag selects a different registry host. The **-r** flag provides a raw registry listing and does not contact any of the target servers for additional information. If a target server is named, the report is constrained to that target server; in this case if **-r** is not specified, the report is very detailed.

CAVEAT Idle time will be incorrect if tool and registry are in different time zones; the target server does not report the time zone in **wtxTsInfoGet**.

SEE ALSO **wtxregd**

wtxregd

NAME **wtxregd** – the Tornado service registry

SYNOPSIS

```
wtxregd [-d.irectory registryDirectory] [-h.elp] [-pd pingDelay]
        [-pn pingNumbers] [-use_portmapper] [-V.erbose] [-v.ersion]
```

DESCRIPTION This daemon is a service registry that maintains a database of target servers, boards or any other item identifiers (see **tgtsvr**) and RPC port numbers.

Target-server identifiers are unique, and they are based on the name of the target board (or an explicit alternate name supplied when the target server is launched) and the name of the host where the target server runs. The two names are linked with the character @, for example, *targetName@serverHost*.

This registry allows the Tornado tools to establish a connection with the target servers.

After accidental shutdown of the registry, recovery is done based on the data saved in wtxregd.*hostName*. The data base file default directory is **$WIND_BASE/.wind** and can be specified through the **-d[irectory]** option.

On UNIX hosts, it is advisable to include the daemon in the **rc.local** file of the host where it runs for UNIX users, so that it starts and restores database automatically after a reboot.

On Windows hosts the Tornado registry can also be used through the wtxregd daemon. Tornado installation adds it to your Automatic Startup if specified, or you can add the command line to the Start>Programs>Startup utilities.

TORNADO REGISTRY SERVICE

Windows NT users can add the **wtxregd** service to the startup services. This addition to the Windows registry is done while installing Tornado.

The Tornado registry service does not accept any command line options; it starts with the registry default options.

Note that, if you install the Tornado registry as a service (on Windows NT), you will not be able to use Tornado 1.0.1 tools with your Tornado 2.0 installation. For more information, see *Tornado Getting Started: Installing Tornado.*

DETECTING DEAD TARGET SERVERS

The registry daemon automatically detects target servers that are not responding. It dynamically removes target servers that are presumed dead from the registry queue. This self-managing feature prevents you from having to manually unregister target servers that are not responding (see TARGET SERVERS PING section for further informations).

Tornado tools need to know the host on which the registry daemon runs. The host name of this machine or its IP address is published through the environment variable **WIND_REGISTRY**.

RESTORING THE DATA BASE

After an registry shutdown, a recovery of the data base is done from the data base backup file whose location has been specified by the **-d** option. (The default location is **$WIND_BASE**/.wind.)

If the target servers registered on the same host as the registry are not responding at restoration time, they will be removed from the registry queue.

TARGET SERVERS PING

While processing a ***wtxToolAttach*( )** on a target server, the registry pings this target server. If the ping is unsuccessful (after 3 seconds of pending ping), the target server is considered to be not responding.

While processing a ***wtxInfoQ*( )**, the target servers that are not responding are pinged if the *pingDelay* has expired (see PING DELAY section).

A not responding target server will be unregistered (and then considered to be dead) if it has been unsuccessfully pinged at least *pingNumbers* times AND if the time between the first and last unsuccessful ping time is at least equal to *pingDelay*pingNnumbers*. This is to avoid having a single failed ping lead to an unregistered target server.

This mechanism is used to allow the Tornado tools to survive network latencies or short time failures.

PING DELAY

This delay (in seconds) can be specified through the **-pd** option, and its default value is set to 120. It specifies after how many seconds a target server that does not respond should be pinged again.

In combination with the **-pn** option (see PING NUMBER section), this value sets the time after which a target server that does not respond is considered dead.

See the EXAMPLE section for examples on how to use the **-pd** option.

PING NUMBER

This number, set by the **-pn** option, fixes the number of times a target server that does not respond has to be unsuccessfully pinged to be considered dead.

See the EXAMPLE section for examples on how to use the **-pn** option.

WARNING: If either the *pingDelay* or *pingNumbers* variable is set to 0, the installation will not forgive any network latencies, and a target server could be unregistered just because the network was slow for a short time.

CAVEAT

The vertical tab character is not supported in any of the registry fields and should never be used.

OPTIONS

-d | -directory
Specify the directory containing the registry database file.

-h | -help
Display a help message summarizing target-registry usage and options.

-pd *pingDelay*
Set the delay between two pings on a dead target server.

-pn *pingNumbers*
Set the number of failed pings before considering a target server dead.

-use_portmapper
Use the local portmapper to register the target servers RPC services. This flag must be set if version 1.x Tornado tools or target servers have to connect using this registry.

-V | -Verbose
Turn on verbose mode. By default, the registry daemon is silent. Verbose mode displays information, warnings, and error messages.

-v | -version
Print the Tornado version.

EXAMPLES

Display the Tornado registry on line help.

```
wtxregd -h
```

Start a Tornado registry in verbose mode.

```
wtxregd -V
```

Start a Tornado registry in verbose mode and save data base file in /tmp.

```
wtxregd -V -d /tmp
```

Start a Tornado registry using the local portmapper to register target servers.

```
wtxregd -use_portmapper
```

Start a Tornado registry with 30 seconds between two unsuccessful target server pings.

```
wtxregd -V -pd 30
```

Start the Tornado registry with only two failed pings allowed.

```
wtxregd -V -pn 2
```

Start the Tornado registry so that it unregisters target servers as soon as they seem to be unreachable.

```
wtxregd -V -pn 0
wtxregd -V -pd 0
```

ENVIRONMENT VARIABLES

WIND_BASE
the root location of the Tornado tree.

FILES

WINDBASE/.wind/wtxregd.*hostName*
default location for the backup registry file (can be changed through the **-d[irectory]** option).

SEE ALSO

tgtsvr, **launch**, **browser**, **windsh**, **wtxreg** (UNIX only), *Tornado User's Guide*

wtxtcl

NAME

wtxtcl – the Tornado Tcl shell

SYNOPSIS

```
wtxtcl [file]
```

DESCRIPTION

wtxtcl is the Tornado Tcl shell. This shell provides all the Tcl facilities plus the WTX Tcl APIs.

INTRODUCTION

The WTX Tcl API provides a binding of the WTX protocol to the Tcl language. This allows Tcl scripts to be written that interact with the WTX environment. Every WTX protocol request is available to the Tcl interface. The names of all WTX Tcl API commands are derived from the protocol request names according to the conventions discussed in the *Tornado API Programmer's Guide: Tcl Coding Conventions*. In other words, underscores are removed and all words but the first are capitalized. For example, **WTX_MEM_READ** becomes **wtxMemRead**.

The Tcl API is accessible directly by means of the wtxtcl tool, from WindSh by typing **?**, in CrossWind by using **tcl**, and in launch and the browser when these tools are started with the **-T** option.

ERROR MESSAGES

A wtxtcl command can return one of several types of errors:

A WTX error
: See the reference for **WTX** in the online *Tornado API Reference* for a list of WTX errors.

A Tcl command parameter parsing error
: See **host/src/libwpwr/wtxtcl/wtparse.c**, which is the file where these errors are generated.

A wtxtcl error
: See the listings in the ERRORS section of the function documentation for examples.

MEMORY BLOCKS

The **memBlockXxx** routines implement a memory block data type. The goal is to allow efficient management of blocks of target memory provided by WTX from Tcl programs. In particular, the memory blocks are not converted to string form except when a Tcl program requests it.

WTX routines that return (or accept) blocks of target memory must supply block handles provided by this library.

Blocks have both a logical size, specified by their creator, and an allocation size. The allocation size is the amount of heap memory allotted for the block data. The routines obtain memory to enlarge blocks in chunks.

Blocks are coded for the endianness of the target that supplied them, and the **memBlockSet** and **memBlockGet** routines automatically swap 16- and 32-bit quantities when they are stored to or retrieved from a block.

USING A SCRIPT FILE

Specifying a *file* in the wtxtcl command line makes wtxtcl use this file as script file.

wtxtcl then executes all specified procedures from the script file. This does not act like a sourced file in that **wtxtcl** exits after having executed the script file.

Example First write the **myWtxTcl.tcl** script

```
# myWtxTcl.tcl - user defined WTX / Tcl script file
#
# modification history
# --------------------
# 01a,02jun98,fle written
#*/

#
# DESCRIPTION
# This tcl script file contains user-defined WTX-Tcl procedures
#
##############################################################################
#
# cpuNumGet - gets the CPU number of the specified tgtSvr
#
# SYNOPSIS
#   cpuNumGet tgtSvr
#
# PARAMETERS
#   tgtSvr : the target server to get CPU number from
#
# RETURNS: The CPU number or -1 on error
#
# ERRORS: N/A
#
proc cpuNumGet { tgtSvr } {
    set cpuNum -1
    if { [catch "wtxToolAttach $tgtSvr" tgtSvrName] } {
        return $cpuNum
    }
    if { [catch "wtxTsInfoGet" tsInfo] } {
        return $cpuNum
    }
    wtxToolDetach
```

```
    set cpuNum [lindex [lindex $tsInfo 2] 0]
    return $cpuNum
}
##########################################################################
#
# main - command to execute at wtxtcl startup
#
puts stdout [cpuNumGet $argv]
```

Then launch wtxtcl with the file name as argument and the target server name as the argument to **myWtxTcl.tcl**

```
wtxtcl myWtxTcl.tcl
-1
wtxtcl myWtxTcl.tcl vxsim0
61
```

SEE ALSO **WTX** entry in the online *Tornado API Reference*, *Tornado API Programmer's Guide: The WTX Protocol*, *Tornado User's Guide*

wtxTest

NAME **r** – Tcl script, wtxtcl test suite launcher

DESCRIPTION This tool thoroughly checks each WTX (Wind River Tool Exchange) protocol service. Several unit tests test various features of the WTX services. This test tool, based on exclusively Tcl scripts is a convenient way to run the WTX services test suite from the UNIX shell.

When started, the test tool needs to connect to a specified target server (see **tgtsvr**) whose identifier is given as a required parameter. It is possible to use a regular expression for the server identifier. For example, the following will run all the tests on vxsim1@aven:

```
aven% tclsh wtxtest.tcl sim1
```

If there is an ambiguity in your target server name, an error message is displayed.

The *testName* parameter is optional. If omitted, the test tool will apply all tests in the test suite (use the **-l** option to see the list of tests). It is possible to run a subset of tests by using a glob-style expression. For example, the following will run all the memory tests:

```
aven% tclsh wtxtest.tcl vxsim1@aven Mem
```

The following will run all the context and event tests:

```
aven% tclsh wtxtest.tcl vxsim1@aven Cont Eve
```

OPTIONS

-h
Print a help message about test tool usage and options.

-i ***expr***
This option must be followed by a Tcl expression. This expression is evaluated before any other work by **wtxtcl**. This option is used mainly to debug the test suite.

-l [*globexp***]**
List all the unit tests in the test suite. It is possible to get only the tests that match a certain pattern by specifying a glob-style expression.

-t ***timeout***
This option allows a timeout to be set for all the tests in the test suite.

-I ***invalidAddr***
This option allows an invalid address to be specified when it cannot be determined by the test suite.

-SUPPRESS
Do not run tests with invalid addresses.

-tsload
Use direct access by target server to load an object module.

-r [*files***]**
Create the reference files required to test the target server in loopback mode. This option should be invoked only if the reference files are missing.

ENVIRONMENT VARIABLES

WIND_BASE
specifies the root location of the Tornado tree.

FILES

The files in the following directories are required by the test tool:

/tmp and **/dev**
directories for temporary files.

$WIND_BASE/host/src/test/wtxtcl
directory for unit test files.

$WIND_BASE/target/lib/obj*CPUtool*test
directory for loader test files.

$WIND_BASE/host/resource/test
directory for resource files required to run the test suite.

SEE ALSO

tgtsvr, **launch**, **wtxregd**, *Tornado API Programmer's Guide*

D

Utilities Reference

aoutToBin – extract text and data segments from a BSD a.out object module 439
aoutToBinDec – extract text and data segments from a.out file on x86 439
binToAsm – convert a binary file to an assembly file 439
coffArmToBin – extract text/data segments from ARM COFF object file 440
coffHex – convert a COFF format object file into Motorola hex records 440
coffHex960 – convert a COFF format object file into Motorola hex records 440
coffHexArm – convert a COFF format object file into Motorola hex records 441
coffToBin – extract text and data segments from a COFF object file 442
elfHex – convert a ELF format object file into Motorola hex records 444
elfToBin – extract text and data segments from an ELF file 444
hex – convert an a.out format object file into Motorola hex records 446
hexDec – convert a.out format object file to Motorola hex records for x86 447
htmlBook – html Book Index generator 447
htmlLink – install cross references in HTML documents 450
makeStatTbl – make a table of status values 453
makeSymTbl – make a table of symbols 454
memdrvbuild – filesystem builder for memDrv 454
munch – extract initializers and finalizers from C++ object modules 456
refgen – Tornado Reference documentation generator 457
romsize – compute the size of a ROM image 463
syngen – Tornado Reference documentation generator 463
vxencrypt – encryption program for loginLib 464
vxsize – compute the size of a vxWorks image 464
xsym – extract the symbol table from a BSD a.out object module 467
xsymc – extract the symbol table from a COFF object module 467
xsymcArm – extract the symbol table from an ARM COFF object module 468
xsymDec – extract the symbol table from an a.out object module for x86 468

Typographic Conventions

The reference entries in this Appendix use the following special characters to present information concisely:

[…] – Square brackets enclose optional parts of the command line. (Used in *Synopsis* sections only.)

. – A period within a command-line option name shows how to abbreviate the option. For example, **-f.ormat** means that you can use either **-f** or **-format**, with the same effect. (*Synopsis* sections only.)

| – Alternation; two alternative options or parameters are shown with this separator when you may choose one or the other. For example, **-N | -Nosyms** indicates that **-N** and **-Nosyms** are alternative spellings of the same option. (*Options* sections only.)

aoutToBin

NAME **aoutToBin** – extract text and data segments from a BSD **a.out** object module

SYNOPSIS

```
aoutToBin < inFile > outfile
```

DESCRIPTION This tool extracts the text and data segments from a BSD **a.out** object module, on standard input, and writes it to standard output as a simple binary image.

aoutToBinDec

NAME **aoutToBinDec** – extract text and data segments from **a.out** file on x86

SYNOPSIS

```
aoutToBinDec < inFile > outfile
```

DESCRIPTION This tool extracts the text and data segments from a BSD **a.out** object module, on standard input, and writes it to standard output as a simple binary image.

binToAsm

NAME **binToAsm** – convert a binary file to an assembly file

SYNOPSIS

```
binToAsm file
```

DESCRIPTION This tool uses **od** to produce an ASCII hexadecimal representation of a binary file. The output of **od** is massaged to produce an assembly file which begins with the label **_binArrayStart**, and ends with the label **_binArrayEnd**.

A C program would reference the data as follows:

```
extern UCHAR binArrayStart [];    /* binary image */
extern UCHAR binArrayEnd;         /* end of binary image */
```

SEE ALSO UNIX documentation for **od**

coffArmToBin

NAME **coffArmToBin** – extract text/data segments from ARM COFF object file

SYNOPSIS

```
coffArmToBin < infile > outfile
```

DESCRIPTION This tool extracts the text and data segments from a COFF object module on standard input, and writes it to standard output as a simple binary image.

coffHex

NAME **coffHex** – convert a COFF format object file into Motorola hex records

SYNOPSIS

```
coffHex [-a adrs] [-l] [-v] [-p PC] [-s SP] file
```

DESCRIPTION This program generates a Motorola hex format (S-record) file from a COFF format object module. Normally, the entry address in the object module is used as the starting address of the output module. However, if the **-a** flag is used, then *adrs* is used as the starting address. Normally S1 records are generated for addresses less than 64K; S2 records are generated for addresses greater than 64K and less the 16M; and S3 records are generated for addresses greater than 16M.

OPTIONS

-l	generate only S2 records.
-a *adrs*	use *adrs* as the entry address, rather than the address in the object module.
-v	output vector information at address 0.
-p *PC*	use *PC* as the PC in the vector (meaningless without **-v**).
-s *SP*	use *SP* as the SP in the vector (meaningless without **-v**).

coffHex960

NAME **coffHex960** – convert a COFF format object file into Motorola hex records

SYNOPSIS

```
coffHex960 [-[TD]imifile,offset][-a offset] [-l] file
```

DESCRIPTION

This program generates a Motorola hex format (S-record) file from a COFF format object module. Normally, the entry address in the object module is used as the starting address of the output module in ROM. However, if the **-a** flag is used, then *offset* is used as the starting address. *offset* is a hex value.

Normally S1 records are generated for addresses less than 64K; S2 records are generated for addresses greater than 64K and less than 16M; and S3 records are generated for addresses greater than 16M.

The text or data section from a second (optional) file can also be processed. This file is usually the IMI file for an Intel 960 processor.

Example:

```
coffHex960 -Dimi,0x0 -a 0x800 bootrom
```

OPTIONS

-l	generate only S2 records.
-a *offset*	use *offset* as the ROM entry address, rather than the address in the object module.
-[TD]*imifile,offset*	use either the text (T) or data(D) section from the imi file. Use *address* as the IMI offset in ROM.

coffHexArm

NAME

coffHexArm – convert a COFF format object file into Motorola hex records

SYNOPSIS

```
coffHexArm [-a offset] [-l] file
```

DESCRIPTION

This program generates a Motorola hex format (S-record) file from a COFF format object module. Normally, the entry address in the object module is used as the starting address of the output module in ROM. However, if the **-a** flag is used, then *offset* is used as the starting address. *offset* is a hex value.

Normally S1 records are generated for addresses less than 64K; S2 records are generated for addresses greater than 64K and less than 16M; and S3 records are generated for addresses greater than 16M.

Example:

```
coffHex960 -a 0x800 bootrom
```

OPTIONS

-l	generate only S2 records.

-a *offset* use *offset* as the ROM entry address, rather than the address in the object module.

coffToBin

NAME **coffToBin** – extract text and data segments from a COFF object file

SYNOPSIS

```
coffToBin < infile > outfile
```

DESCRIPTION This tool extracts the text and data segments from a COFF object module on standard input, and writes it to standard output as a simple binary image.

configUlipSolaris

NAME **configUlipSolaris** – User-Level IP (ULIP) start/stop script

SYNOPSIS

```
configUlipSolaris [ start | stop ]
```

DESCRIPTION This script installs the pseudo-network ULIP driver in a Solaris (SunOS 5.X) kernel. It loads the device driver into the kernel, creates 16 devices, and configures the devices' Internet addresses and routing information.

By convention, addresses are 127.0.1.0 through 127.0.1.31 for the 32 processors/processes that may interact at once. The **/dev** directory will have entries from **/dev/ul0** to **/dev/ul31**.

This script must be run as **root**.

OPTIONS

start Start ULIP services. If ULIP devices already exist they are unplumbed first, and any running **loopd** daemons are stopped. After this the **ul** driver is installed in the kernel, the **loopd** daemon is started, and the new devices are plumbed.

stop Stop ULIP services. ULIP devices are unplumbed, the **loopd** daemon is stopped, and the **ul** driver is removed from the kernel.

CAVEAT If VxSim is using any of the 16 ULIP devices when this script is executed with the start or stop command, an error will occur like the following:

```
host# /etc/init.d/ulip stop
```

```
stopping ULIP services...
Device busy
Cannot unload module: ul
Will be unloaded upon reboot.
host#
```

To ensure ULIP starts properly when this occurs: terminate all VxSim processes, execute this script with the **stop** command for a second time, and then execute the following script with the **start** command.

```
host#
host# ps -af | grep vxWorks
user  3006  2963  0 12:12:25 pts/3    0:00 vxWorks -p 1
user  3164  3016  0 12:19:00 pts/3    0:00 grep vxWorks
host# kill 3006
host#
host# /etc/init.d/ulip stop
stopping ULIP services...
host# /etc/init.d/ulip start
starting ULIP services...
host#
```

At this point ULIP services should be running properly.

SEE ALSO *VxWorks Programmer's Guide: VxSim*

deflate

NAME **deflate** - deflate (compress) a file

SYNOPSIS **`deflate < infile > outfile`**

DESCRIPTION This tool reads from standard input and writes a compressed image to standard output. It is used to compress boot ROM images, and is called by the BSP makefiles when you make a compressed ROM image such as a boot ROM.

For integrity checking, a magic byte (**Z_DEFLATED**) is added to the beginning of the compressed stream and a 16-bit checksum (a standard IP checksum) is added to the end of the stream.

The compression stream itself is built using the public domain zlib software, which has been modified by Wind River Systems. For more information, see the zlib home page at **http://quest.jpl.nasa.gov/zlib/**.

elfHex

NAME **elfHex** – convert a ELF format object file into Motorola hex records

SYNOPSIS
```
elfHex [-a adrs] [-l] [-v] [-p PC] [-s SP] file
```

DESCRIPTION This program generates a Motorola hex format (S-record) file from a ELF format object module. Normally, the entry address in the object module is used as the starting address of the output module. However, if the **-a** flag is used, then *adrs* is used as the starting address. Normally S1 records are generated for addresses less than 64K; S2 records are generated for addresses greater than 64K and less the 16M; and S3 records are generated for addresses greater than 16M.

OPTIONS

-l	generate only S2 records.
-a *adrs*	use *adrs* as the entry address, rather than the address in the object module.
-v	output vector information at address 0.
-p *PC*	use *PC* as the PC in the vector (meaningless without **-v**).
-s *SP*	use *SP* as the SP in the vector (meaningless without **-v**).

elfToBin

NAME **elfToBin** – extract text and data segments from an ELF file

SYNOPSIS
```
elfToBin < inFile > outfile
```

DESCRIPTION This tool extracts the text and data segments from an ELF object module, on standard input, and writes it to standard output as a simple binary image.

elfToBsd

NAME **elfToBsd** – convert MIPS ELF object modules to BSD **a.out** format

SYNOPSIS
```
elfToBsd < infile_elf > outfile_bsd
```

DESCRIPTION This tool converts MIPS ELF format object modules to BSD-style **a.out** object modules.

elfXsyms

NAME **elfXsyms** – extract the symbol table from an ELF file

SYNOPSIS

```
elfXsyms < objMod > symTbl
```

DESCRIPTION This tool reads an object module (UNIX SVR4 ELF format) from standard input and writes an object module to standard output. The output object module contains only the symbol table, with no code, but is otherwise a normal, executable object module.

This tool is used to generate the VxWorks symbol table, **vxWorks.sym**.

SEE ALSO **makeSymTbl**, UNIX SVR4 ELF documentation

extractBsp

NAME **extractBsp** – extract a BSP from a VxWorks installation

SYNOPSIS

```
extractBsp -b bspName -s vxworksDir [-d outfile]
```

DESCRIPTION This tool extracts all files that make up a specified BSP from a given VxWorks installation and places them in a tar file. The tar file can then be installed into a VxWorks tree with **installOption**.

This tool provides a means of installing older BSPs in later releases of VxWorks that support them. It is useful as an installation alternative for those who need to install a previously released BSP into a new VxWorks release but cannot locate the original BSP distribution tape.

OPTIONS

-b *bspName*	the name of the BSP as it appears in the **$WIND_BASE/target/config** directory.
-s *vxworksDir*	the base directory of the VxWorks installation from which you want to extract the BSP.
-d *outfile*	By default, the resulting tar file is placed in the current directory with the name ***bspName*_tar**. If **-d** is used and *outfile* is a directory, then the resulting tar file is created there and given the name ***bspName*_tar**; otherwise the file is given the name *outfile*.

EXAMPLE

The following example extracts an MVME147 BSP. The script is run from the newly installed VxWorks tree, **$WIND_BASE/target**, and extracts the BSP from the VxWorks 5.1.x tree, **/usr/vw.5.1**. The **installOption** script is then run to install the BSP in the **$WIND_BASE**/target directory.

```
% cd $WIND_BASE/target
% extractBsp -b mv147 -s /usr/vw.5.1
% installOption -f mv147_tar
```

NOTE: While **extractBsp** correctly collects all the files that comprise a given BSP, it may actually extract more than you need, depending on how many BSPs are in the *vxworksDir* installation. This is because the script extracts all files found in **h/drv** and **src/drv**. For this reason, the preferred alternative is to install BSPs from the original distribution tape.

SEE ALSO

installOption

hex

NAME

hex – convert an **a.out** format object file into Motorola hex records

SYNOPSIS

```
hex [-a adrs] [-l] [-v] [-p PC] [-s SP] file
```

DESCRIPTION

This program generates a Motorola hex format (S-record) file from an **a.out** format object module. Normally, the entry address in the object module is used as the starting address of the output module. However, if the **-a** flag is used, then *adrs* is used as the starting address. Normally S1 records are generated for addresses less than 64K; S2 records are generated for addresses greater than 64K and less the 16M; and S3 records are generated for addresses greater than 16M.

OPTIONS

-l	generate only S2 records.
-a ***adrs***	use *adrs* as the entry address, rather than the address in the object module.
-v	output vector information at address 0.
-p ***PC***	write *PC* as the program counter (PC) in the vector (meaningless without **-v**). The default PC is the entry point, if you specify **-v** without **-p**.
-s ***SP***	write *SP* as the stack pointer (SP) in the vector (meaningless without **-v**). The default SP is the entry point, if you specify **-v** without **-s**.

hexDec

NAME **hexDec** – convert **a.out** format object file to Motorola hex records for x86

SYNOPSIS `hexDec [-a adrs] [-l] [-v] [-p PC] [-s SP] [-b] file`

DESCRIPTION This program generates a Motorola hex format (S-record) file from an **a.out** format object module. Normally, the entry address in the object module will be used as the starting address of the output module. However, if the **-a** flag is used, then *adrs* is used as the starting address. Normally S1 records are generated for addresses less than 64K; S2 records are generated for addresses greater than 64K and less the 16M; and S3 records are generated for addresses greater than 16M.

OPTIONS

-l	generate only S2 records.
-a ***adrs***	use *adrs* as the entry address, rather than the address in the object module.
-v	output vector information at address 0.
-p ***PC***	write *PC* as the program counter (PC) in the vector (meaningless without **-v**). The default PC is the entry point, if you specify **-v** without **-p**.
-s ***SP***	write *SP* as the stack pointer (SP) in the vector (meaningless without **-v**). The default SP is the entry point, if you specify **-v** without **-s**.
-b	output signature bytes, size, and checksum for additional BIOS ROM.

htmlBook

NAME **htmlBook** – html Book Index generator

SYNOPSIS `htmlBook [-h] [-nonav] [-noupdate] [-skip` *dirName*`] [`*dirList*`]`

DESCRIPTION This tool generates the hierarchy of HTML indexes for the HTML manuals within a directory tree. The top-level index is called **books.html**.

htmlBook reads all linkage files generated by **refgen**, by **htmlLink**, or inserted manually, extracts all the book and chapter/section names, then creates the proper indexes.

Each book has a single entry in the top-level index, which points to a book-level index. If the book was not created by refgen, then it is possible to have this link point directly to the book's table of contents. If the book was created by refgen, then there will be a book index, which will have chapter/section entries pointing to chapter/section indexes. These chapter/section indexes then point to the specific entries in the reference manual.

OPTIONS

-h	Display a simple description of **htmlBook**.
-nonav	Do not create the alphabetic navigation bar at the top of the chapter/section index.
-noupdate	Do not generate new **LIB.SUB** and **RTN.SUB** files, just uses any existing ones.
-skip ***dirName***	Do not process the subdirectory *dirName*. Create no links to or from it. If several directories are to be skipped, then the **-skip** option must be repeated for each of the directories to be skipped.

NON-REFERENCE DOCUMENTS

In order to generate index links to HTML documents not generated by refgen, a LIB (linkage) file must be associated with the new document (in the same directory as the html index for the document). This file allows **htmlBook** to generate the appropriate index entries.

This LIB file should contain a single linkage entry. A linkage entry consists of a Tcl list with eight elements (enclosed in curly braces) defined as follows:

Entry	This field normally represents the name of library. If you are creating a LIB file for an externally generated HTML document, this entry should be the name of the document. This field would normally appear in the chapter/section index, and if the *Category* field (see below) were not empty, the category name (between square brackets) would be appended to the entry name. For an externally generated HTML document, however, the *Chapter Name* field (see below) should be empty, and then this name will only be used internally. The *Link* field (see below) will then be used for the top-level book index.
Short Description	This is a short description of the entry that will appear in the chapter/section index, if present.
Pattern	This field is used only by **htmlLink** to specify what pattern should be replaced with the text in the *Link* field (next field) in all HTML files.
Link	This specifies the HTML text to be inserted, after the path is corrected, as the link for the entry. For an external document, this is usually a link to the table of contents of the book. The path in this link should be relative to the LIB file, and should begin with a ./ so that it can be corrected for the actual location of the indexes.
Book Name	For a reference entry, this field specifies the book the library entry belongs to (for example, Tornado Reference or BSP Reference). This field would appear in the top-level index. For an external document, this field is only used internally.

Chapter / Section Name This field specifies the chapter or section that the entry belongs in. In combination with the *Book Name* field, this allows two levels of indexes (for example, VxWorks Reference Manual>ANSI Libraries). If there is no chapter name, the *Link* field will be used (after the path is corrected) in the top-level index.

Category This field makes it possible to distinguish help entries that have the same name, but belong to different libraries (for example, **wtxToolAttach** for C, Java or Tcl APIs).

File This is used by **htmlLink** to generate the **RTN.SUB** and **LIB.SUB** files. The relative path of the **LIB** file containing this entry is stored in this field. If this field is empty, the **LIB** file is in the local directory.

EXAMPLE

If the **LIB** file contains the following line (all on one line):

```
{HTMLWorks} {HTMLWorks user\xd5 s guide} {<b>HTMLWorks</b>}
{<b><a href="./guide.html">HTMLWorks</a></b>} {Tornado Optional Products}
{HTMLWorks} {} {}
```

then the top-level HTML index (**book.html**) will contain a link named *Tornado Optional Products* to a book HTML index, and this book index will contain a link to a section named *HTMLWorks*. If other **LIB** files have entries with *Tornado Optional Products* as their book name, the corresponding section names will be listed along with *HTMLWorks*.

The *HTMLWorks* section index will then have links for all the **LIB** file entries that give *Tornado Optional Products* as their book name, and *HTMLWorks* as their section name.

To make a direct link from the top-level index and the table of contents of an externally generated HTML document, just provide a **LIB** file with an empty *chapter name* field. The following example will create a *Tornado User's Guide* entry in the top-level (**book.html**) index, and the link will go directly to *path***/tug/TUG.html**, where:

path is the directory the **LIB** file is in.
tug comes from the *Path* field.
TUG.html is extracted from the *Link* field.

```
{Tornado user\xd5 s guide} {The Tornado user\xd5 s guide} \
{<b>Tornado User\xd5 s Guide</b>}{<b><a href="./TUG.html">Tornado \
User\xd5 s Guide</a></b>}{Tornado user\xd5 s guide} {} {} {tug}
```

NOTE: The above entry must be on a single line.

FILES

updateDocTime This is an empty file that fixes the last doc update time. Such a file is generated in each directory in *dirList*.

books.html The top-level HTML index, containing all books that **htmlBook** found. All other index file names are derived from the book or chapter/section names by replacing any non-alphanumeric characters with an underscore.

RETURNS N/A

ERRORS N/A

FILES **$WIND_BASE/docs/book.html**

SEE ALSO **refgen**, **windHelpLib**, **htmlLink**

htmlLink

NAME **htmlLink** – install cross references in HTML documents

SYNOPSIS **`htmlLink [-h] [-R] [-skip`** *dirName*`]` `[`*directory*`]`

DESCRIPTION This tool uses the information in the linkage files **RTN** and **LIB** to create HTML links from references in documents to the appropriate reference entries. It can also concatenate the various **RTN** and **LIB** files under each directory into **RTN.SUB** and **LIB.SUB** files recursively.

This command is used to create all the cross reference links to HTML reference manuals previously generated with **refgen**. It should only be used after all the HTML documentation files have been built for a given project.

OPTIONS

-h Display a brief message describing the usage of this tool.

-R Make links in all subdirectories (recursion).

-skip *dirName* Do not process directory *dirName* when **-R** flag is in effect. Each directory to be skipped must be listed separately, as:

htmlLink -R -skip notes -skip lost+found -skip private

directory The directory to be processed. If the -R flag is used, all its subdirectories will also be processed. If no directory is specified, the current working directory is processed.

RETURNS N/A

ERRORS N/A

EXAMPLE

Building cross-references for a BSP help files.

Go to the BSP directory and build the HTML documentation:

```
% cd $WIND_BASE/target/config/myBsp
% make man
```

Generate cross-references in the BSP help files:

```
% cd $WIND_BASE/docs/vxworks/bsp/myBsp
% htmlLink .
```

SEE ALSO

refgen, **windHelpLib**, **htmlBook**

installOption

NAME

installOption – generic installer for unbundled products

SYNOPSIS

```
installOption [-f file] [-n] [-l]
```

DESCRIPTION

This shell script installs the contents of a Tornado BSP or optional product tape (for example, mv167, VxVMI, VxMP) into an existing Tornado directory tree. This script issues the required commands to read files from the tape and add them to the Tornado archive(s). The old archive contents are backed up prior to adding the new files. This back-up is done only once; consecutive installations will not back up the archive(s).

OPTIONS

-f *file*	Use *file* as the tape device. If **-f** is not present, use the default device.
-n	Show what would be done if **installOption** were run without **-n**, but do not actually run it.
-l	Summarize the contents of the tape; no installation is performed.

PROCEDURE

To install a BSP or an optional product:

1. Place the BSP or optional product tape in an appropriate tape device (see below).
2. Change directories so that the root directory of the Tornado directory tree (for example, **/usr/wind**) is the current working directory.

3. Invoke this script by typing:

```
% ./host/host/bin/installOption
```

where *host* is the name of the UNIX host type plus the name of the operating system running on the host (for example, "sun4-sunos4"). See the above section for option flags.

The *GNU ToolKit* for the target CPU architecture(s) must already be installed before this script is run, because the appropriate archiver must be used to update the Tornado library.

CAVEAT

This script reads the BSP or optional product tape twice. You must use a rewinding tape device.

installUlipSolaris

NAME

installUlipSolaris – install ULIP files on Solaris host

SYNOPSIS

```
installUlipSolaris [ basic | uninstall ]
```

DESCRIPTION

This script installs the mechanisms and files to make ULIP services work on Solaris (SunOS 5.X). When run with no options, **installUlipSolaris** provides a mechanism for ULIP to automatically start and stop when Solaris is booted and shutdown.

The first action performed by this script is to stop all ULIP services if they are installed and working. Upon successful installation, this script also starts ULIP services before exiting.

This script must be run as **root**.

The following files are produced when installing:

```
/etc/init.d/ulip            - ULIP start/stop script
/sbin/loopd                 - loopd daemon
/kernel/drv/ul              - ULIP driver
/kernel/drv/ul.conf         - ULIP driver configuration file
```

When run with no arguments the script also installs the following:

```
/etc/rc2.d/S80ulip          - Link to /etc/init.d/ulip
/etc/rc0.d/K10ulip          - Link to /etc/init.d/ulip
```

OPTIONS

basic Install all the files above except the links to **/etc/init.d/ulip**, and then start ULIP services. Before any installation takes place, this script ensures that ULIP services have been stopped. Using this option, ULIP services can be stopped and started by running the script **/etc/init.d/ulip** (see the **configUlipSolaris** manual page), but it does not provide the mechanism to start and stop ULIP when Solaris changes system run levels.

uninstall Removes the files above. Before any removal, this script ensures that ULIP services have been stopped.

SEE ALSO *VxWorks Programmer's Guide: VxSim*

makeStatTbl

NAME **makeStatTbl** – make a table of status values

SYNOPSIS

```
makeStatTbl hdir [...]
```

DESCRIPTION This tool creates an array of type **SYMBOL** that contains the names and values of all the status codes defined in the **.h** files in the specified directories. All status codes must be prefixed with "**S_**" to be included in this table, with the exception of status codes in the UNIX-compatible header file **errno.h**. In each *hdir* there must be a ***ModNum.h** file which defines the module numbers, e.g., "**M_**". The generated code is written to standard output.

The symbol array is accessible through the global variable **statTbl**. The array contains **statTblSize** elements. These are the only external declarations in the generated code.

This tool's primary use is for creating the VxWorks status table used by ***printErrno*( )**, but may be used by applications as well. For an example, see **$WIND_BASE/target/config/all/statTbl.c**, which is generated by this tool from **$WIND_BASE/target/h/***.

FILES

hdir/***ModNum.h**	module number file for each h directory
symLib.h	symbol header file

SEE ALSO **errnoLib**, **symLib**

makeSymTbl

NAME **makeSymTbl** – make a table of symbols

SYNOPSIS

```
makeSymTbl objMod
```

DESCRIPTION This tool creates the C code for a symbol table structure containing the names, addresses, and types of all global symbols in the specified object module; the generated code is written to standard output. ***usrRoot*( )** in **usrConfig.c** inserts these symbols in the **standAloneSymTbl**.

This tool is used only when creating a standalone system. Normally, it is not needed, since the symbol table is constructed by reading and interpreting the symbol table contained in the system load module (**a.out**format), either from the local boot disk or from a host system over the network.

The generated symbol table is an array of type **SYMBOL** accessible through the global variable **standTbl**. The array contains **standTblSize** elements. These are the only external declarations in the generated code.

For an example, see the file **$WIND_BASE/target/config/***bspname***/symTbl.c**, which is generated by this tool for **vxWorks.st** in the same directory.

FILES

symLib.h	symbol table header file
a_out.h	UNIX BSD 4.3 object module header file

SEE ALSO **xsym**

memdrvbuild

NAME **memdrvbuild** – filesystem builder for **memDrv**

DESCRIPTION The **memdrvbuild** utility is designed to be used in conjunction with the ***memDrv*( )** memory file driver. It packs files and directories into a compilable C file which can be built into VxWorks, and then mounted as a filesystem using ***memDrv*( )**.

memdrvbuild converts a directory hierarchy, and all of the files contained within it, into a single C source file. The individual files are converted into C data arrays containing the contents of the files (which may be in any format), along with administrative data describing the names and sizes of the files. This constructed C file can then be built into VxWorks in the normal way.

The generated C file also contains two function which can be called to register or unregister all of the packed files and directories; they are mounted as a filesystem using ***memDevCreateDir*( )**, and unmounted using ***memDevDelete*( )**. An include file is also generated, containing a declaration of this function.

USAGE

This utility is invoked as:

```
memdrvbuild [ -o filebase ] [ -m mount ] directory
```

where:

-o *filebase*
The base name for the generated **.c** and **.h** files. Defaults to the name of the source directory.

-m *mount*
The name ("mount point") under which directory will be mounted on the target. Defaults to the name of the source directory.

directory
The source directory containing the files to be packed into a **memDrv** filesystem.

The output **.c** file contains two function, called ***memDrvAddFiles****mount*( ) and ***memDrvDeleteFiles****mount*( ), where *mount* is the argument to the **-d** option, with non-alpha characters converted to _. This first function mounts the packed files using ***memDevCreate*( )**. The second function unmounts the packed files using ***memDevDelete*( )**. The **.h** file contains a declaration of those functions.

Each file will be mounted with the name *mount*/*file*, where *file* is the pathname of the file below the indicated source directory.

EXAMPLE

Given a directory **docs**, containing a number of files:

```
memdrvbuild -m /mem -o memFiles docs
```

will produce two files, **memFiles.c** and **memFiles.h**. **memFiles.c** will contain the data for the packed files, plus two functions:

```
STATUS memDrvAddFiles_mem (void);
STATUS memDrvDeleteFiles_mem (void);
```

When called, the first function will mount all of the contained files in the filesystem under **/mem**. For example, the file **docs/fred.html** would be mounted as **/mem/fred.html**. The second function will unmount all of the contained files in the filesystem under **/mem**.

munch

NAME **munch** – extract initializers and finalizers from C++ object modules

SYNOPSIS

```
wtxtcl $WIND_BASE/host/src/hutils/munch.tcl [-asm arch]
```

DESCRIPTION This tool produces data structures used to call constructors and destructors for static objects. It is used when compiling C++ modules.

Given an ordered list of C++ object modules, munch produces a C program containing Ctors/Dtors arrays where the initialization order is the one promised below.

The modules should be ordered on the command line so that **a.o** is to the left of **b.o** if any initializers that appear in **a.o** but in no module to the right of **a.o** should be called before any initializers in **b.o**. Notice that each initializer is called only once (in its rightmost position) even though it may appear in more than one module. Finalizers are run in the reverse order.

If you are using a GNU compiler you should invoke munch with the -asm flag. This causes munch to output assembly directives which can be compiled without any special flags. On the other hand, if you do not supply the **-asm** flag the compiler may have to be coerced into accepting non-standard characters in identifiers (such as **$**). On GNU compilers this is achieved with the **-fdollars-in-identifiers** option.

EXAMPLES Consider a project consisting of two C++ modules, **user1.o** and **user2.o**, linked with a user library **myLib.a** and a VxWorks library (say **libMC68040gnuvx.a**). Then the ctors file can be generated using:

Windows

```
nm68k partialImage.o partialUserImage.o user1.o user2.o \
      | wtxtcl %WIND_BASE%\host\src\hutils\munch.tcl -asm 68k > ctdt.c
```

UNIX

```
nm68k partialImage.o partialUserImage.o user1.o user2.o \
      | wtxtcl $WIND_BASE/host/src/hutils/munch.tcl -asm 68k > ctdt.c
```

Here **partialUserImage.o** is the result of linking **user1.o** and **user2.o** against **myLib.a** and **partialImage.o** is the result of linking **partialUserImage.o** against **libMC68040gnuvx.a**.

This will ensure that the VxWorks library is initialized before the user library which in turn is initialized before any of the project modules. The latter are initialized in the order they appear on the command line.

The following commands will compile **hello.cpp**, then munch **hello.o**, resulting in the munched file **hello.out**, suitable for loading by the Tornado module loader:

Windows

```
   cc68k -I%WIND_BASE%\target\h -DCPU=MC68020 -nostdinc \
        -fno-builtin -c hello.cpp
```

```
nm68k hello.o | wtxtcl %WIND_BASE%\host\src\hutils\munch.tcl \
      -asm 68k > ctdt.c
cc68k -c ctdt.c
ld68k -r -o hello.out hello.o ctdt.o
```

UNIX

```
cc68k -I$WIND_BASE/target/h -DCPU=MC68020 -nostdinc -fno-builtin \
      -c hello.cpp
nm68k hello.o | wtxtcl $WIND_BASE/host/src/hutils/munch.tcl
      -asm 68k > ctdt.c
cc68k -c ctdt.c
ld68k -r -o hello.out hello.o ctdt.o
```


refgen

NAME

refgen – Tornado Reference documentation generator

SYNOPSIS

refgen [-book *bookName***] [-chapter** *chapterName***] [-config** *configFile***]**
[-cpp] [-expath *pathList***] [-exbase** *basedir***] [-h] [-int]**
[-l *logFile***] [-mg] [-out** *outDir***] [-verbose]** *fileList*

DESCRIPTION

This tool implements a table-driven process for extraction of documentation from source. Input tables define a "meta-syntax" that specifies the details of how documentation is embedded in source files for a particular programming language. Similarly, output tables define the mark-up details of the documentation output.

OVERALL CONVENTIONS

Some conventions about how documentation is embedded in source code do not depend on the source language, and can therefore not be changed from the configuration tables.

Overall Input Conventions

Routines are organized into *libraries*, and each library begins with a **DESCRIPTION** section. If a **DESCRIPTION** heading is not present, the description section is taken to be the first comment block after the modification history. Some input languages (such as shellscript) may optionally begin with a section headed **SYNOPSIS** instead.

The first line in a library source file is a one-line title in the following format:
sourceFileName - simple description

That is, the line begins (after whatever start-of-comment character is required) with the name of the file containing it, separated by space, hyphen, and space from a simple description of the library.

The first line in a routine's description (after the source-language-dependent routine delimiter) is a one-line title in the same format.

Routine descriptions are taken to begin after the routine-title line, whether or not a **DESCRIPTION** tag is present explicitly.

Section headings are specified by all-caps strings beginning at a newline and ending with either a newline or a colon.

Italics, notably including *Text variables*--that is, words in the documentation that are not typed literally, but are instead meant to be replaced by some value specific to each instance of use--are marked in the source by paired angle brackets. Thus, to get the output *textVar*, type **<textVar>**.

Boldface words are obtained as follows: **General mechanism:** surround a word with single quotes in the source. **Special words:** words ending in "Lib" or in a recognized filename suffix are automatically rendered in bold. For example, **fileName.c**, **object.o**, **myStuff.tcl** all appear in boldface.

Simple lists can be constructed by indenting lines in the source from the margin (or from the comment-continuation character, if one is required in a particular source language). For example:

```
line one
line two
```

Overall Output Conventions

Library descriptions are automatically prefaced by a synopsis of the routines in that library, constructed from the title lines of all routines.

For most input languages, a **SYNOPSIS** section is supplied automatically for each routine as well, extracted from the routine definition in a language-dependent manner specified in the input meta-syntax tables. Input languages that do not support this have empty strings for **$SYNTAX(declDelim)**; in such languages, the **SYNOPSIS** section must be explicitly present as part of the subroutine comments.

For some languages, the routine definition is also used to create a **PARAMETERS** section automatically.

The online form of documentation is assumed to fit into a structure involving a parent file (which includes a list of libraries) and a routine index. Several of the procedures in this library require names or labels for these files, in order to include links to them in the output. The parent file and routine index need not actually exist at the time that procedures in this library execute.

DESCRIPTION tags are supplied automatically for all description sections, whether or not the tag is present in the source file.

SEE ALSO sections are always present in the output for routine descriptions, whether or not they are present in the source. **SEE ALSO** sections for routine descriptions automatically include a reference to the containing library.

OUTPUT DIRECTIVES

The following keywords, always spelled in all capital letters and appearing at the start of a line, alter the text that is considered for output. Some directives accept an argument in a specific format, on the same line.

NOROUTINES
Do not generate subroutine documentation from this file (must appear in the library section).

NOMANUAL
Suppresses the section where it appears: either an entire routine's documentation, or the library documentation. Routine documentation can also be suppressed in language-specific ways, specified by matching a regular expression in the meta-syntactic list **LOCALdecls**. See **refDocGen** for a command line option that overrides this.

INTERNAL
Suppresses a section (that is, all text from the directive until the next heading, if any). See **refDocGen** for a command line option that overrides this.

APPEND *filename*
Include documentation from *filename* in the output as if its source were appended to the file containing the **APPEND** keyword.

EXPLICIT MARKUP

refgen supports a simple markup language that is meant to be inconspicuous in source files, while supporting most common output needs.

The following table summarizes refgen explicit markup (numbered notes appear below the table):

Note	Markup	Description
	\" *text to end of line*	Comment in documentation.
	'text ...' or ‘text ...’	Boldface text.
	<...>	Italicized text.
[1]	\\ or \/	The character \.
[2]	\< \> \& \‘ \’	The characters < > & ‘ ’.
	|	The character | within a table.
[3]	\ss ... \se	Preformatted text.
	\cs ... \ce	Literal text (code).
	\bs ... \be	Literal text, with smaller display.
[4]	\is \i ... \i ... \ie	Itemized list.

Note	Markup	Description
[5]	\ml \m ... \m ... \me	Marker list.
[6]	\ts ... \te	A table.
	\sh text	A subheading.
	\tb reference	A reference followed by newline.

Notes on Markup

[1] The escape sequence \\ is easier to remember for backslash, but \/ is sometimes required if the literal backslash is to appear among other text that might be confused with escape sequences. \/ is always safe.

[2] < and > must be escaped to distinguish from embedded HTML tags. & must be escaped to distinguish from HTML character entities. Single quotes must be escaped to distinguish from the **refgen** automatic bolding convention.

[3] Newlines and whitespace are preserved between \ss and \se, but formatting is not otherwise disabled. These are useful for including references to text variables in examples.

[4] \is and \ie mark the beginning and end of a list of items. \i is only supported between \is and \ie. It marks the start of an item: that is, it forces a paragraph break, and exempts the remainder of the line where it appears from an increased indentation otherwise applied to paragraphs between \is and \ie.

Thus, the following:

```
\is
\i Boojum
A boojum is a rare
tree found in Baja
California.
\i Snark
A snark is a different
matter altogether.
\ie
```

Yields output approximately like the following:

Boojum
A boojum is a rare tree found in Baja California.

Snark
A snark is a different matter altogether.

[5] \ml and \me delimit marker lists; they differ from itemized lists in the output format--the marker beside \m appears on the same line as the start of the following paragraph, rather than above.

[6] Between \ts and \te, a whole region of special syntax reigns, though it is somewhat simplified from the original mangen's table syntax.

Three conventions are central:

(a) Tables have a heading section, and a body section; these are delimited by a line containing only the characters - (hyphen), + (plus) and | (stile or bar), and horizontal whitespace.

(b) Cells in a table, whether in the heading or the body, are delimited by the | character. \| may be used to represent the | character.

(c) Newline marks the end of a row in either heading or body.

Thus, for example, the following specifies a three-column table with a single heading row:

```
\ts
Key | Name       | Meaning
----|------------|------------
\& | ampersand  | bitwise AND
\| | stile      | bitwise OR
#  | octothorpe | bitwise NAND
\te
```

Key	Name	Meaning
&	ampersand	bitwise AND
|	stile	bitwise OR
#	octothorpe	bitwise NAND

The cells need not align in the source, though it is easier to read it (and easier to avoid errors while editing) if they do.

PARAMETERS

-book This option allows you to define which book the documented routine or library will belong to. The default value is "Wind River Systems Reference Manual."

-chapter Related to the **-book** option, this option allows you to set the documented routine or library chapter. The default value is set to "Wind River Systems Reference Libraries."

-category	Related to the **-book** option, this option allows you to set the documented routine or library category. It can be used for example to differentiate routines available for different host platforms.
-config *configname*	Reads configuration information from the file *configname* if this optional parameter is present; the default configuration is **C2html**.
-cpp	This option specifies that the list of given files is to be treated as C++ files. Special processing is then done to recover all the class members.
-expath *pathList*	Preempts **EXPANDPATH** settings recorded within files with the explicitly-supplied colon-delimited path list.
-exbase *basedir*	To simplify working under incomplete trees, uses *basedir* rather than **WIND_BASE** as the root for expand-file searching.
-int	If this optional parameter is present, formats all available documentation, even if marked **INTERNAL** or **NOMANUAL**, and even for **local** routines.
-l *logFile*	Specifies that *logFile* is to be used to log **refgen** errors. If **-l** option is not specified, standard output is used.
-mg	Converts from **mangen** markup "on the fly" (in memory, without saving the converted source file).
-out *outputDirectoryName*	Saves output documentation file in *outputDirectoryName*.
-verbose	Prints reassuring messages to indicate something is happening.
sourceFileList	Any other parameters are taken as names of source files from which to extract and format documentation.

EXAMPLE Generate HTML manual pages for a BSP **sysLib.c** file:

```
% cd $WIND_BASE/target/config/myBsp
% refgen -mg -book BSP_Reference -chapter myBsp -category myBsp \
-out $WIND_BASE/vxworks/bsp/myBsp sysLib.c
```

INCLUDES **refLib.tcl**

SEE ALSO *VxWorks Programmer's Guide: C Coding Conventions*, **htmlLink**, **htmlBook**, **windHelpLib**

romsize

NAME **romsize** – compute the size of a ROM image

SYNOPSIS

```
romsize [-k nnn] [-b xxx] file
```

DESCRIPTION This tool calculates the size of a specified ROM image. The size of the ROM can be specified using the **-k** or **-b** option; the default is 128 Kb. If the image size (text + data) is greater than the ROM size, a warning is displayed.

OPTIONS

-k *nnn* specifies the size of the ROM in kilobytes; the default is 128K.
-b *xxx* specifies the size of the ROM in bytes base 16.

EXAMPLE

```
% romsize -k 256 bootrom
bootrom: 244988(t) + 59472(d) = 304460 (42316 over)
Warning: image is larger than 262144 bytes!
```

SEE ALSO UNIX documentation for **size**

syngen

NAME **syngen** – Tornado Reference documentation generator

SYNOPSIS

```
syngen [-config configFile] [-clean] [-d outDir] [-l logFile] [-h]
[-V] fileList
```

DESCRIPTION Use this utility to collect synopsis-only output from *fileList*

This utility opens all the files from *fileList* and appends the found synopses specified through the **-d *outDir*** option.

The **-config** option allows you to specify which configuration file to use in case an unknown file extension appears (Known extensions are **.c**, **.cc**, **.cpp**, **.tcl**, **.sh**, **.java**, **.s**, **.nr** and **.pcl**). The default value for *configFile* is set to C.

The **-clean** option specifies that the synopsis file will be cleaned, not built.

PARAMETERS

-clean	specifies a synopsis cleaning operation.
-config	specifies the *configFile* to be used to parse *fileList*.
-d	specifies that *outDir* is to be used to append all the synopses.
-h	displays the help message.
-l	specifies that *logFile* will be used as the log receiver.
-V	sets verbose mode on.
fileList	list of files to be parsed by **syngen**.

SEE ALSO **refgen**, **htmlLink**, **htmlBook**, **windHelpLib**

vxencrypt

NAME **vxencrypt** – encryption program for **loginLib**

SYNOPSIS

```
vxencrypt
```

DESCRIPTION This tool generates the encrypted equivalent of a supplied password. It prompts the user to enter a password, and then displays the encrypted version.

The encrypted password should then be supplied to VxWorks using the ***loginUserAdd*( )** routine. This is only necessary if you have enabled login security by defining **INCLUDE_SECURITY**. For more information, see the reference entry for **loginLib**.

This tool contains the default encryption routine used in ***loginDefaultEncrypt*( )**. If the default encryption routine is replaced in VxWorks, the routine in this module should also be replaced to reflect the changes in the encryption algorithm.

SEE ALSO **loginLib**, *VxWorks Programmer's Guide: Target Shell*

vxsize

NAME **vxsize** – compute the size of a vxWorks image

SYNOPSIS

```
vxsize [-v h_adr l_adr] [-k kbytes] [ -b hex] file
```

DESCRIPTION This tool calculates the size of a specified VxWorks image and compares the size of the image with the system image as it would sit in RAM, i.e. the difference of

RAM_HIGH_ADRS and **RAM_LOW_ADRS**. If the image size (text + data + bss) is greater than the difference, a warning is displayed.

OPTIONS

-v *h_adr l_adr*	specifies the size of the *h_adr* and *l_adr* addresses in bytes base 16.
-k *kbytes*	specifies the size of the system image in RAM in kilobytes.
-b *hex*	specifies the size of the system image in RAM in bytes base 16.

EXAMPLE

```
% vxsize -v 0010000 002000 vxWorks
vxWorks: 312720(t) + 28596(d) + 32430(b) = 373746 (57344 over)
Warning: image is larger than 316402 bytes!
```

SEE ALSO UNIX documentation for **size**

wind_host_type

NAME **wind_host_type** – return the host type in the Tornado format

SYNOPSIS

```
wind_host_type
```

DESCRIPTION This tool returns the host type of the machine on which it is run, in the Tornado format. In particular, it is intended to facilitate the setting of the environment variable **WIND_HOST_TYPE**. For example, if you are a C shell user, you can add the following line to your **.cshrc** file:

```
setenv WIND_HOST_TYPE \xd4 $WIND_BASE/host/resource/wind_host_type\xd4
```

windman

NAME **windman** – UITclSh GUI for quick help retrieving

SYNOPSIS

```
windman
```

DESCRIPTION The Tornado software suite includes the *Tornado Online Manuals*, a hypertext publication with complete reference information for Tornado and VxWorks. This information can be displayed from any of the **Help** menus of the Tornado tools or by invoking **windman** from the UNIX command line.

The **windman** interface is a UITcl interface with two tabs that allow you to navigate around the Tornado reference information. This interface provides both a table of contents and a quick search index. It contains two sheets:

CONTENT SHEET This sheet allows you to navigate through the Wind River reference files tree structure. You can navigate through the books, chapters, and topics by expanding the tree branches. When you reach the leaves of the tree (the function or library names), double-clicked on a name to open a web browser displaying the reference entry.

INDEX SHEET This sheet has two fields that allow you to locate topics in different ways.

Entry Name Field This text field allows you to type the name of the function or library you wish to retrieve. As you type each letter the selection moves to the closest name found in the entry list field. This is a case-sensitive interface.

Entry List Field This is a complete list of all the available routine and library references. It allows you to select any item from the list.

Double-click on the selected list item or click the Display button to open a browser window containing the reference entry.

SEE ALSO **refgen**, **syngen**, **htmlBook**, **htmlLink**, **windHelpLib**

xlinkHppa

NAME **xlinkHppa** – fix debug info in a partially linked HP-PA SOM COFF object module

SYNOPSIS

```
xlinkHppa objMod
```

DESCRIPTION This tool reads a partially linked object module (HP-PA SOM format) and rewrites it, with relocations applied to the debugging information. This program must be used if you want correct symbolic debugging information in partially linked object files.

FILES **som_coff.h** - HP-PA SOM COFF object module header file

SEE ALSO **xsymHppa**, HP-PA SOM documentation

xsym

NAME **xsym** – extract the symbol table from a BSD **a.out** object module

SYNOPSIS

```
xsym < objMod > symTbl
```

DESCRIPTION This tool reads an object module (UNIX BSD 4.3 **a.out** format) from standard input, and writes an object module to standard output. The output module contains only the symbol table, with no code, but is otherwise a normal, executable object module.

This tool is used to generate the VxWorks symbol table, **vxWorks.sym**.

FILES **a_out.h** - UNIX BSD 4.3 object module header file

SEE ALSO **makeSymTbl**, UNIX BSD 4.3 **a.out** documentation

xsymc

NAME **xsymc** – extract the symbol table from a COFF object module

SYNOPSIS

```
xsymc < objMod > symTbl
```

DESCRIPTION This tool reads an object module (UNIX SYSV COFF format) from standard input and writes an object module to standard output. The output module contains only the symbol table, with no code, but is otherwise a normal, executable object module.

This tool is used to generate the VxWorks symbol table, **vxWorks.sym**.

FILES **ecoff.h** - UNIX SYSV COFF object module header file

SEE ALSO **makeSymTbl**, UNIX SYSV COFF documentation

xsymcArm

NAME **xsymcArm** – extract the symbol table from an ARM COFF object module

SYNOPSIS

```
xsymcArm < objMod > symTbl
```

DESCRIPTION This tool reads an object module (UNIX SYSV COFF format) from standard input and writes an object module to standard output. The output module contains only the symbol table, with no code, but is otherwise a normal, executable object module.

This tool is used to generate the VxWorks symbol table, **vxWorks.sym**.

FILES **ecoff.h** - UNIX SYSV COFF object module header file

SEE ALSO **makeSymTbl**, UNIX SYSV COFF documentation

xsymDec

NAME **xsymDec** – extract the symbol table from an a.out object module for x86

SYNOPSIS

```
xsymDec < objMod > symTbl
```

DESCRIPTION This tool reads an object module (UNIX BSD 4.3 **a.out** format) from standard input, and writes an object module to standard output. The output module contains only the symbol table, with no code, but is otherwise a normal, executable object module.

This tool is used to generate the VxWorks symbol table, **vxWorks.sym** for the x86.

FILES **a_out.h** - UNIX BSD 4.3 object module header file

SEE ALSO **makeSymTbl**, UNIX BSD 4.3 **a.out** documentation

xsymHppa

NAME **xsymHppa** – extract the symbol table from an HP-PA SOM COFF object module

SYNOPSIS `xsymHppa < objMod > symTbl`

DESCRIPTION This tool reads an object module (HP-PA SOM format) from standard input and writes an object module to standard output. The output module contains only the symbol table, with no code, but is otherwise a normal, executable object module.

This tool is used to generate the VxWorks symbol table, **vxWorks.sym**.

FILES **som_coff.h** - HP-PA SOM COFF object module header file

SEE ALSO **makeSymTbl**, HP-PA SOM documentation

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:

${WIND_BASE}/host/resource/app-defaults/Tornado
Comprehensive definitions for all resources common to all the Tornado tools.

${WIND_BASE}/host/resource/app-defaults/Tornado-grayscale
Grayscale-monitor overrides for resources common to all the Tornado tools.

${WIND_BASE}/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**).

${WIND_BASE}/host/resource/app-defaults/*toolName***-color**
Color-monitor overrides specific to a particular Tornado tool.

${WIND_BASE}/host/resource/app-defaults/*toolName***-grayscale**
Grayscale-monitor overrides specific to a particular Tornado tool.

Index

Symbols

? (C/Tcl interpreter toggle) 198
    quick access 200–201
@ command (booting) 51
@ prefix (WindSh) 168

A

About menu
    Tornado command 57
add command (CrossWind) 253
Add New Build Specification window 135
address(es), memory
    current, setting 181
add-symbol-file command (CrossWind) 253
Admin menu (launcher) 83
    Authorize command 84
    FTP WRS command 84
    Install CD command 83
agent, *see* target agent
***agentModeShow*( )** 164
    mode switching, under 170
animation 56
APIs
    see also Tornado API Guide; *Tornado API Reference*
    documentation xxi
application I/O (CrossWind) 256
application modules
    see also bootable applications; downloadable applications
    bootable 93
    displaying information about (browser) 221
    downloadable 92
    loading 253
    unloading 254
Application Program Interfaces, *see* APIs
application wizard 97
arrays (WindSh) 184
Assembly command (CrossWind) 241
assignments (WindSh) 182–183
Attach System command (CrossWind) 240
attach system command (CrossWind) 262
Attach Task command (CrossWind) 239
Authorize command (launcher) 84
auto scaling 127

B

***b*( )** 163
    mode switching, under 170
    using 164
back ends, communications 80–81
backplane
    installing boards in 29–30
    processor number 48

Backtrace command (CrossWind) 243
Bash
  host environment, configuring 21
***bd*( )** 163
***bdall*( )** 163
***bh*( )** 164
board support packages (BSP) 94
  boot media 28
  creating projects 113
  documentation xxiii
  pre-Tornado 2.x versions, using 113
boot media 28
boot programs
  boot parameters, configuring 145
  building 145
  configuring 144–147
  TSFS, for 146
boot ROMs
  installing 28
  reprogramming 53
bootable applications 93
  auto scaling 127
  creating 127–128
  initialization routines, adding application 128
***bootChange*( )** 163
booting 45–54
  @ command 51
  alternative procedures 52
  commands 48
  parameters 46–51
    command-line format 52
    displaying current, at boot time 46
    nonvolatile RAM 53
    reprogramming boot ROMs 53
    setting 46–47
  troubleshooting 59–62
    boot display, using 51
Bourne shell
  host environment, configuring 21
break command (CrossWind) 255
break thread command (CrossWind) 264–265
breakpoints
  commands for handling, built-in shell 164
  deleting 245
  disabling 245
  hardware, setting 245
  setting
    buttons, using CrossWind 244–245
    command line, using 255
  temporary, setting 245
  threads, setting in 264–265
***browse*( )** 165
browser (HTML)
  specifying 291
browser (Tornado) 207–232
  application module information, displaying 221
  application task list 212
  behavior of, controlling 211
  buttons 210
  class information, displaying 220
  CPU utilization, reporting 224
  customizing 231
  interrupt/exception vector table, displaying 223
  limitations 227
  loaded module list 213
  memory consumption graphs 212
  memory partition information, displaying 218
  menu bar 210
  message queue information, displaying 217
  object information, displaying 213–220
  quitting 210
  semaphore information, displaying 215
  spy utility 224
  stack checks, displaying 226
  starting 208
  state indicator bar 209
  symbols, setting sort order for 211
  system task list 212
  task information, displaying 214
  troubleshooting with 227–231
  updating displays 210
    time intervals, setting 212
  watchdog information, displaying 219
browser command 208
browser.tcl 231
BSP, *see* board support packages
Build Output window 108
build specifications 128–136

assembler options, specifying 133
changing 128
compiler options, specifying 132
creating new 135
current build, selecting for 136
link order, specifying 133
linker options, specifying 134
makefile macros, specifying 131
makefile rules, working with 130
property sheets, working with 129–135
Build toolbar
buttons 109
displaying 109
bus, *see* VMEbus
busy box 56
buttons
browser, Tornado 210
Build toolbar 109
CrossWind 244–252
launcher, Tornado 72
toolbar
Browser 67
CrossWind 67
Project 67
VxSim 67
WindSh 67
WindView 68

C

C interpreter (WindSh) 174–190
addresses, setting current 181
arguments, using 180
arrays 184
assignments 182–183
commands, built-in 156–168
differentiating from target routines 168
list of 186
comments 183
data types, handling 175–177
function calls 179–180
limitations 185–186
literals 177
operators 178
redirection vs. relational 188
pointers 184–185
redirecting I/O 186–190
shConfig, using 174
scripts 189–190
statements, handling 177
strings 183
declaring 176
subroutines as commands 180
tasks
current, setting 181
referencing 181
Tcl
expressions, embedding 200
interpreter
quick access to 200–201
toggling to 198
variables 178
creating 182
***c*( )** 163
mode switching, under 170
omitting task parameters 181
C++ support 191–193
commands, built-in (WindSh) 165
demangling function names 193
overloaded function names, reconciling 191–192
C/C++ command (CrossWind) 241
cables, connecting 30
cache, target server 79
call command (CrossWind) 255
***cd*( )** 160
character arrays, *see* strings
characters, control, *see* control characters
***checkStack*( )** 158
classes (kernel objects)
displaying information about (browser) 220
***classShow*( )** 166
code
conventions 326–337
Tcl 330–334
unsupported, directory for 313
code examples
CrossWind extensions 276–278
launcher, customizing 85–90

shConfig, using 156
sytem mode debugging 171
command-line parameters 52
commands, built-in (WindSh) 156–168
C++ development 165
debugging 163
command line, in the 259
displaying objects and object information 165–167
list of 186
network information, displaying 167
network status, displaying 167
show routines 165–167
system information, obtaining 160–162
system modification 163
target routines, differentiating from 168
target-resident code, requiring 197
task information, obtaining 158
tasks, managing 157
commands, universal menu 57
comments (WindSh) 183
compiling, *see* application modules; *GNU **Make** User's Guide*; *GNU ToolKit User's Guide*
component 93
config target directory 308
configuration management, *see* version control
configuring
directories and files 308
host environment 20–24
jumpers 29
network software, host 26–27
target 24–26
target hardware 28–30
target servers 73–81
Connect Target Servers command (CrossWind) 240
context-sensitive help xxiii
workspaces, in 95
continue command (CrossWind) 248
control characters (shell) 186
see also specific control characters; **tyLib**(1)
conventions
coding 326–337
documentation xxiv–xxv
interface (GUI) 56–57
Tcl
code, layout of 330–334
declarations 329
module headings 326
naming 334
outside procedures 329
procedures 327–328
programming style 334–337
typographic 342, 438
core file 77
***cplusCtors*( )** 165
***cplusDtors*( )** 165
***cplusStratShow*( )** 165
***cplusXtorSet*( )** 165
Create Project window 97
Create Target Server form 76
options, specifying 76–81
Create... command (launcher) 72
cross-development 3–12
CrossWind 233–278
see also debugging; *GDB User's Guide*
application modules
loading 253
unloading 254
auxiliary debugger displays, controlling 243
breakpoints, setting
buttons, using GUI 244–245
command line, using 255
buttons 244–252
defining new 251–252
summary of 234
code, displaying
regular expressions, containing 255
selecting display mode 241
specified line 254
command-line facilities 258
command-line interface 252–261
commands 252–261
information about, displaying 258
shell, built-in 163
command line, in the 259
customizing 273–278
code examples 276–278
debugger state, providing information about 258
detaching session from task 261

download options 260
download timeout 260
editor, invoking 249
execution while debugging, starting 255
exit status reporting 260
expressions, evaluating 255
GDB Tcl interpreter 266–272
commands, submitting 266–267
expressions, evaluating 269
GDB command line, using 266
GDB facilities, invoking 268–271
GUI Tcl interpreter, switching to 274
I/O file descriptors, closing 270
linked lists, traversing 271–272
list of Tcl elements, returning 269
naming new commands 266
redirecting I/O 270
routine names, returning 270
scripts, debugging GDB-based 267
symbolic addresses 270
symbols
returning local 270
testing for 271
verbose error printing 268
GDB, running 252–261
.gdbinit, disabling 260
GUI Tcl interpreter 273–278
extensions, creating
code examples 275–278
GDB Tcl interpreter, switching to 274
initialization files 273
inspection windows, launching 249–251
interrupts 247
I/O, redirecting 256
name, debugger session 261
object files, specifying 253
pointer values, displaying 257
program state, providing information about 258
quitting 238
registers, displaying machine 243
single-stepping 247
stack frame summaries, displaying 257
stack size 260
stack traces, displaying current 243
starting 234
state indicator 236
structure browsers 250
subroutines
finishing current 248
moving through stack 248
symbol values
displaying 257
monitoring 249–251
printing 249
system mode 261–265
targets
connecting to 257
task execution, continuing 248
task mode 261
task priority 260
Tcl, using 243
threads, managing 262–265
WTX protocol requests, sending 259
CrossWind command (Support menu) 294
crosswind command-line command 234
crosswind.tcl 243
debugger GUI, customizing 273
re-initializing debugger 243
CTRL+C (interrupt key) 187
shell commands, interrupting 173
shell, terminating the 151
CTRL+D
completing symbol and path names 152
end-of-file 151
function synopsis, printing 153
CTRL+H (delete) 187
CTRL+L (clear input/output) 258
CTRL+Q (resume) 188
CTRL+S (suspend) 187
CTRL+SHIFT+X (reboot) 169
CTRL+U (delete line) 187
CTRL+W
function reference page, displaying 154
HTML help, launching 154
CTRL+X (reboot) 187
CTRL+Z (suspend shell) 188
Customer Information form 295
customer services (WRS) 293–301
Customize Tools dialog box 283–289

macros, using 285
customizing Tornado 279–291
see also initialization files
download options, setting 279
editor, specifying alternate 290
Tcl files, using 290
Tools menu 282–289
alternate editor command, creating (example) 288
binary utilities commands, creating (example) 288
macros for custom commands 285
version control command, creating (example) 287
Web link to WRS, creating (example) 289
version control 281
customizing VxWorks 110–127
booting VxWorks 127
building VxWorks 126
image type, selecting 124
project files, creating 113
projects, creating 111
VxWorks components
configuring 117–124
dependencies, calculating 119
descriptions of 118
excluding 118–123
finding 118
including 118–123
parameters, accessing 124
size of components only, estimating image 124
system image size, recalculating 121
warnings, conflict 122

D

***d*( )** 162
omitting address parameter 181
strings, displaying 176
data types 175–177
data variables (WindSh) 178
creating 182
debugger (CrossWind) 233–278
see also CrossWind; debugging
debugging
see also CrossWind; *GDB User's Guide*
commands, built-in shell 163
command line, in the 259
disassembling 161
remote source-level (CrossWind) 233–278
shell, from the 150
system mode
code example 171
DEL key (delete)
projects, removing 110
delete character (**CTRL+H**) 187
delete command (CrossWind) 245
delete-line character (**CTRL+U**) 187
demangling, *see* name demangling 193
demo host directory 307
Dependencies dialog box 105
detach command (CrossWind) 261
Detach menu command (CrossWind) 261
Detach Task/System command (CrossWind) 240
development environment 1–12
***devs*( )** 160
dialogs, *see* forms
directory tree 305–313
disable command (CrossWind) 245
disassembler (***l*( )**) 161
display /W command (CrossWind) 257
docs directory 305
see also online documentation
documentation xix–xxv
conventions xxiv–xxv
online xxiii
VxWorks xix–xxiii
down command (CrossWind) 249
Download command (CrossWind) 238
download options 279
downloadable applications 92
building 104–109
errors and messages, displaying 108
specifications, providing 106
creating 95–110
project files
adding 103
closing 103

creating 100
linking 103
opening 103
properties, modifying 103
property information, displaying 103
removing 103
saving 103
projects
adding 110
creating new 97
removing 110
downloading 109
downtcl command (CrossWind) 274
dragging with mouse 57
drivers, *see* threads
DSM_HEX_MOD 156
dual mode, definition 32

E

edit mode, shell (WindSh) 193–196
see also **ledLib**(1)
input mode toggle (**ESC** key) 188
EDITOR host environment variable 22
specifying 290
editor, alternate
specifying 290
Tools menu command for, creating (example) 288
end-of-file character (**CTRL+D**) 151
environment variables, host 20–23
EDITOR 22
HOME 201
LD_LIBRARY_PATH (Solaris 2) 22
PATH 21
PRINTER 23
shell behavior, controlling 155
SHLIB_PATH 21
WIND_BASE 20
WIND_HOST_TYPE 20–21
WIND_REGISTRY 20
environment, *see* cross-development; development environment; host environment; target environment
eof (**stty**) 187
erase (**stty**) 187
errorInfo global variable 268
errors
Tcl 324
unwinding 324
ESC key (input/edit mode toggle) 188
/etc/hosts 27
/etc/hosts.equiv 27
Ethernet
cable connections 30
***exit*( )** 151
CTRL+C, using 173
expressions, C language (WindSh), *see* data variables; function calls; literals; operators
External Dependencies folder 102
makefile dependencies, listing 105
external mode, *see* system mode

F

File menu
Add Project to Workspace command 110
CrossWind
Download command 238
New Project command 110
Quit command 57
browser, in 210
CrossWind, in 238
file names, shell and target server 196–197
file tree 305–313
Find Object dialog box 118
finish command (CrossWind) 248
FIXED.TXT xx
flow-control characters (**CTRL+Q** and **CTRL+S**) 187
folders 57
forms, operating with keyboard 57
frame command (CrossWind) 257
Free Software Foundation documentation xxiii
FTP (File Transfer Protocol)
password, user 50
FTP WRS command (launcher) 84
function calls (WindSh) 179–180

arguments, passing 179
nesting 179
parentheses, omitting 180
reference page, displaying HTML 154
stepping over 247
synopses, printing 153

G

-g compiler option
limitations 132
GDB (GNU debugger)
see also CrossWind; debugging
running 252–261
Tcl interpreter 266–272
gdb command (CrossWind) 268
GDB Online command (debugger) 244
gdbEvalScalar command (CrossWind) 269
gdbFileAddrInfo command (CrossWind) 269
gdbFileLineInfo command (CrossWind) 269
.gdbinit 253
gdbIOClose command (CrossWind) 270
gdbIORedirect command (CrossWind) 270
gdbLocalsTags command (CrossWind) 270
gdbStackFrameTags command (CrossWind) 270
gdbSymbol command (CrossWind) 270
gdbSymbolExists command (CrossWind) 271
GNU binary utilities, working with 288
GNU command (Support menu) 294
GNU ToolKit, *see* linking; *GNU ToolKit User's Guide*

H

h target directory 309
***h*( )** 160
scripts, creating shell 189
using 194
header files
Tornado tools 306
VxWorks-supplied 309
help
context-sensitive xxiii
workspaces, in 95
online xxiii
reference documentation 118
technical support (WRS) 293–301
help command (CrossWind) 258
Help menu 57
browser 210
debugger
GDB Online command 244
On CrossWind command 244
Manuals Contents command xxiii
Manuals Index command xxiii
***help*( )** 160
hierarchical displays 57
history facility, shell 193
HOME host environment variable 201
host
access to, VxWorks 27
directory and file tree 306–307
network software
configuring 26–27
initializing 26
type, naming 20–21
host directory 306–307
host environment
configuring 20–24
Bash, Bourne shell, or Korn shell, using 21
C shell, using 21
host shell, *see* shell
host-os host directory 306
***hostShow*( )** 167

I

***i*( )** 158
mode switching, under 170
***icmpstatShow*( )** 167
***ifShow*( )** 167
include host directory 306
INCLUDE_WDB_BANNER 142
INCLUDE_WDB_START_NOTIFY 142
INCLUDE_WDB_TTY_TEST 142
INCLUDE_WDB_USER_EVENT 142
INCLUDE_WDB_VIO 142

info command (CrossWind) 258
Info menu (launcher) 301
info threads command (CrossWind) 262–263
inhibit-gdbinit (CrossWind) 260
initialization files 314
  browser 231
  CrossWind 273
  GDB 253
  launcher 85
  shell (WindSh) 201
initializing
  network software, host 26
Install CD command (launcher) 83
installing
  see also Tornado Getting Started Guide
  boards in backplane 29–30
  boot ROMs 28
  directory 20
interaction, common features 56–57
Internet
  see also World Wide Web
  addresses
    host, of 50
    target, of 50
interrupt key (**CTRL+C**) 187
  shell commands, interrupting 173
  shell, terminating the 151
interrupt service routines (ISR), *see* threads
interrupt/exception vector table 223
interrupts, sending (CrossWind) 247
intr (**stty**) 187
***intVecShow*( )** 167
I/O
  redirecting 186–190
    debugging, during 256
    shConfig, using 174
  virtual 12
***iosDevShow*( )** 166
***iosDrvShow*( )** 166
***iosFdShow*( )** 166
***ipstatShow*( )** 167
***iStrict*( )** 158

J

jumpers 29

K

kernel 4
keyboard
  forms (dialogs), using with 57
  menus, using with 57
keyboard shortcuts
  CTRL+C (interrupt) 187
  CTRL+D
    completing symbol and path names 152
    end-of-file 151
    function synopsis, printing 153
  CTRL+H (delete) 187
  CTRL+Q (resume output) 188
  CTRL+S (suspend output) 187
  CTRL+SHIFT+X (reboot) 169
  CTRL+U (delete line) 187
  CTRL+W
    HTML help, launching 154
    reference pages, displaying HTML 154
  CTRL+X (reboot) 187
  CTRL+Z (suspend shell) 188
  ESC key (input/edit mode toggle) 188
  shell line editing 193–196
kill (**stty**) 187
Kill command (launcher) 73
Kill Task command (CrossWind) 240
killing
  see also quitting
  target servers 73
Korn shell
  host environment, configuring 21

L

***l*( )** 161
  omitting address parameter 181
launcher 65–90

access, restricting 84
browser, starting 208
buttons 72
connecting tools and targets 68
customer support, accessing WRS 83
customizing with Tcl 84–90
code examples 85–90
initialization file 85
quitting 66
starting 66
support requests, sending 294
target list 66
toolbar 66
training, accessing WRS 83
Web browser, linking to 301
***ld*( )** 163
using 164
LD_CALL_XTORS 155
LD_COMMON_MATCH_ALL 156
LD_LIBRARY_PATH host environment variable (Solaris 2) 22
LD_PATH 155
LD_SEND_MODULES 155
working with 197
lib target directory 310
LICENSE.TXT 306
line editor 193–196
see also **ledLib**(1)
linking
see also GNU ToolKit User's Guide
dynamic 164
linkSyms.c 115
list (Tcl) 320
list command (CrossWind) 254
literals (WindSh) 177
see also strings
***lkAddr*( )** 160
***lkup*( )** 160
load command (CrossWind) 253
login, remote, *see* remote login
logo, Wind River Systems 56
***ls*( )** 160

M

***m*( )** 163
Makefile 116
makefile dependencies, calculating 105
mangling, *see* name demangling
Manuals Contents command (Help menu) xxiii
Manuals Index command (Help menu) xxiii
memory
consumption of, reporting (browser) 212
fragmentation, troubleshooting 228
leaks, troubleshooting 227
target, manipulating 164
memory partitions
displaying information about (browser) 218
***memPartShow*( )** 166
***memShow*( )** 166
menus
see also specific GUI menus
commands, universal 57
operating with keyboard 57
message queues
displaying information about (browser) 217
Mixed command (CrossWind) 241
***moduleIdFigure*( )** 167
***moduleShow*( )** 167
mouse
dragging with 57
selecting with 57
***mqPxShow*( )** 166
***mRegs*( )** 163
***msgQShow*( )** 166

N

name demangling (C++) 193
NetROM ROM emulator 37–45
target agent for, configuring 38–42
troubleshooting 42–45
***netstatShow*( )** 167
Network Information Service (NIS) 26
networks
configuring simple 26–27
status, displaying 167

O

Object Modules folder 102
On CrossWind command (debugger) 244
online documentation xxiii
    function reference pages, displaying 154
    HTML help, launching 154
Open Boot Prom protocol 28
Open Workspace window 97
operators (WindSh) 178
    redirection vs. relational 188
optional VxWorks products 13
    VxMP 5
    VxSim, target simulator 9
    VxVMI 4
    WindView 8
Options command (Tools menu) 279

P

PATH host environment variable 21
path names
    shell and target server 196–197
***period*( )** 157
    target-resident code, requiring 197
pointers (WindSh) 184–185
    arithmetic, handling 184
print * command (CrossWind) 249
print command (CrossWind) 249
PRINTER host environment variable 23
***printErrno*( )** 160
***printf*( )**
    strings, displaying 176
***printLogo*( )** 160
priority inversion
    troubleshooting 229
prjComps.h 116
prjConfig.c 116
prjParams.h 116
problem lists xx
PROBLEMS.TXT xx
processor number 48
Project command (Support menu) 294
project facility 91–144
    bootable applications, creating 127–128
    customizing VxWorks 110–127
    downloadable applications, creating 95–110
    makefile dependencies, calculating 105
    manual configuration, versus 92
    target-host communication interface, configuring 137–144
    terminology 92
projectName.**wpj** 116
projects 91–147
    see also bootable applications; customizing VxWorks; downloadable applications; target agent
    adding 110
    boot programs, using new 144–147
    creating
        based on existing project 113
        bootable applications, for 127–128
        downloadable applications, for 97
    customized VxWorks, for 111
    removing 110
***pwd*( )** 160

Q

quit (**stty**) 187
Quit command (File menu) 57
***quit*( )** 151
quitting
    browser 210
    CrossWind 238
    launcher 66
    shell 151

R

README.HTML 306
README.TXT xx
Reattach command (launcher) 72
reboot character (**CTRL+X**) 187
Reboot command (launcher) 73
***reboot*( )** 163

using 169
rebooting 54
shell, from the 169
redirection, I/O (WindSh) 186–190
shConfig, using 174
reference pages
tools, Tornado 339
utilities 437
Registers command (CrossWind) 243
registry, Tornado 11
remote, using a 19
segregating targets 19
setting up 18–20
release notes xx
see also Tornado Release Notes
remote file access, restricting 27
remote login security 27
***repeat*( )** 157
target-resident code, requiring 197
Reread All command (CrossWind) 243
Reread Home command (CrossWind) 243
Reserve command (launcher) 73
resource host directory 307
rev command (CrossWind) 255
.rhosts 27
ROM, *see* boot ROM; NetROM ROM emulator
romInit.s 116
romStart.c 116
***routestatShow*( )** 167
routines, *see* function calls; commands, built-in
run command (CrossWind) 255
run-time image (on host) 77

S

***s*( )** 163
mode switching, under 170
omitting task parameters 181
using 164
scripts
shell 189–190
startup 190
search command (CrossWind) 255
security 27
selecting with mouse 57
semaphores
displaying information about (browser) 215
***semPxShow*( )** 166
***semShow*( )** 166
serial lines
targets, configuring 34–36
testing 35
Set Hardware Breakpoint window 245
SETUP directory 305
setup.log log file 306
SH_GET_TASK_IO 155
share directory 305
shConfig (Tcl)
code example 156
redirecting I/O 174
shell environment variables, setting 155
shell (WindSh) 149–205
see also C interpreter; C++ support; commands, built-in
C control structures 185
C interpreter 174–190
C++ support 191–193
calculating in 154
commands, built-in 156–168
differentiating from target routines 168
list of 186
components 204–205
control characters 186
data conversion 154
debugging from 150
displaying information from 160–162
edit mode 193–196
see also **ledLib**(1)
environment variables, setting 155
function synopsis, printing 153
history 193
host environment, configuring 21
interpretation, layers of 205–206
interrupting (**CTRL+C**) 173
path names, typing 152
preprocessor facilities 185
quitting 151
reboot character (**CTRL+X**) 187
rebooting from 169

redirecting I/O 186–190
shConfig, using 174
starting 151
strings 176
subroutines as commands 180
suspend character (**CTRL+Z**) 188
system mode 170–173
target from host, controlling 202
target routines from, running 168
target symbol names, typing 152
Tcl interpreter 198–202
initializing with 201
types
compound 185
derived 185
***shellHistory*( )** 160
***shellPromptSet*( )** 160
SHLIB_PATH host environment variable (HP-UX) 21
show command (CrossWind) 258
show routines 165–167
***show*( )** 165
shParse utility (Tcl) 200
si command (CrossWind) 247
simulator, target (VxSim) 9
single-stepping
commands for handling, built-in shell 164
function calls, stepping over 247
next line of code 247
***smMemPartShow*( )** 167
***smMemShow*( )** 166
source code directories (VxWorks) 310
source control, *see* version control
Source menu (CrossWind) 241
Assembly command 241
C/C++ command 241
Mixed command 241
***sp*( )** 157
mode switching, under 171
***sps*( )** 157
spy utility (browser) 224
see also **spyLib**(1)
data gathering intervals, setting 211
mode, setting 211
target-resident code, requiring 197
update intervals, setting 211
src target directory 310
src/demo host directory 307
stack traces
displaying current 243
stacks
overflow, troubleshooting 228
usage, checking (browser) 226
start (**stty**) 188
starting
browser, Tornado 208
CrossWind 234
launcher 66
shell, Tornado 151
target servers 54
Tornado 55
startup scripts 190
state-information files 314
statements (WindSh) 177
step command (CrossWind) 247
stepi command (CrossWind) 247
stop (**stty**) 187
stopping, *see* quitting
strings 183
shell, and the 176
stty command 186
subroutines, *see* function calls; commands, built-in
Support menu (launcher) 294
CrossWind command 294
GNU command 294
Project command 294
Tornado command 294
VxWorks command 294
WindSurf command 294
WindView command 295
support, *see* technical support; customer services
susp (**stty**) 188
suspend shell character (**CTRL+Z**) 188
symbol table
displaying information from 160–162
synchronizing 78
symbols
values, monitoring 249–251
sysALib.s 116
sysLib.c 116

***sysResume*( )** 164
mode switching, under 170
***sysStatusShow*( )** 164
mode switching, under 170
***sysSuspend*( )** 163
mode switching, under 170
system address 27
system image
see also customizing VxWorks
boot image options 145
VxWorks
creating custom 110–127
system information, displaying 160–162
system mode 31
debugging 261–265
code example 171
threads, managing 262–265
initiating, from CrossWind 262
shell 170–173
target agent 10
system name 27

T

-T option (Tcl) 198
target
see also browser (Tornado)
configuring 24–26
ROM emulation 37–42
serial-only 34–36
connecting to tools 68
directory and file tree 308–313
information from, displaying 160–162
memory, manipulating 164
monitoring state of (browser) 207–232
target agent 10
END drivers, configuring 138
exception hooks, configuring for 142
kernel, starting before 142
NetROM, configuring for 138
networks, configuring for 140
scaling 141
serial connections, configuring for 141
simulators, configuring integrated target 138
system mode 10
task mode 10
tyCoDrv connections, condiguring 141
target board
backplane, installing boards in 29–30
cables, connecting 30
configuring 28–30
jumpers, setting 29
processor number 48
target directory 308–313
target environment 4–5
target list (launcher) 66
Target menu (launcher)
Create... command 72
Kill command 73
Reattach command 72
Reboot command 73
Reserve command 73
Unregister command 72
Unreserve command 73
Target Server File System (TSFS)
boot program for, configuring 146
target servers 11
see also launcher; WTX protocol
access, restricting 82
agent, connecting target 31–36
back ends, communications 80–81
configuring 73–81
networked targets 75
options 76–81
serial targets 75
Create Target Server buttons 76
file names 196–197
locking 82
managing 72–83
memory cache 79
path names 196–197
reserving 82–83
restricting users 77
saved configurations, working with 75
selecting 69–71
sharing 82–83
starting 54
symbol table synchronizing 78
troubleshooting 62–63

unreserved 82
virtual console, using a 78
WTX log setup 81
target wtx command (CrossWind) 257
Targets menu (CrossWind)
Attach System command 240
Attach Task command 239
Connect Target Servers command 240
Detach command 261
Detach Task/System command 240
Kill Task command 240
task mode 31
debugger 261
target agent 10
taskCreateHookShow( ) 166
taskDeleteHookShow( ) 166
taskIdDefault( ) 157
taskIdFigure( ) 158
taskRegsShow( ) 166
tasks
see also threads
CPU utilization, reporting 224
current, setting 181
debugging
detaching CrossWind session 261
multiple tasks, switching among 261
displaying information about (browser) 214
execution, continuing 248
IDs 181
information about, obtaining 158
interrupting (CrossWind) 247
managing 157
names 181
referencing 181
registers for, displaying machine 243
taskShow( ) 166
taskSwitchHookShow( ) 166
taskWaitShow( ) 166
Tcl (tool command language) 317–325
arithmetic expressions 322
arrays, associative 321
browser initialization files 231
C applications, integrating with 325
coding conventions 326–337
command substitution 321–322
control structures 324
CrossWind GDB interpreter 266–272
CrossWind GUI interpreter 273–278
customizing
browser, Tornado 231
initialization files 290
launcher 84–90
shell 201
error handling 324
files 322
formatting 322
I/O 322
launcher initialization file 85
linked lists, traversing 271–272
lists 320
procedures, defining 323–324
shell (WindSh) 198–202
C interpreter
quick access from 200–201
toggling to 198
initialization files 201
target, controlling the 199–200
Tk graphics library 318
unwinding 324
variables 319
tcl command (CrossWind) 266–267
tcl host directory 307
Tcl menu (CrossWind) 243
Reread All command 243
Reread Home command 243
Tcl_CreateCommand( ) 325
tcldebug command (CrossWind) 267
tclerror command (CrossWind) 268
-Tclmode option (Tcl) 198
tclproc command (CrossWind) 267
tcpstatShow( ) 167
td( ) 157
technical support (WRS) 293–301
filing Technical Support Requests (TSR) 296–301
terminal characters, *see* control characters
terminating, *see* quitting
tftpInfoShow( ) 167
tgtsvr command 73
see also target servers

starting target servers 54
third-party products 14
thread command (CrossWind) 263
thread IDs 262
breakpoints and system context 265
threads 261
see also tasks
breakpoints, setting 264–265
managing 262–265
summary information, displaying 262–263
switching
explicitly 263
implicitly 265
***ti*( )** 158
omitting task parameters 181
Tk graphics library (Tcl) 318
toolbars
Build 109
launcher 66
toolchain 93
Tools menu
adding/removing custom commands 283–289
Customize command 282
customizing 282–289
alternate editor, for 288
binary utilities, for 288
version control, for 287
Web link to WRS, for 289
No Custom Tools placeholder 282
Options command 279
tools, development 4–9
see also browser; CrossWind; launcher; project facility; WindSh
adding/removing Tools menu commands 283–289
connecting to targets 68
environment variables, setting host 20–22
launching 71
using, guidelines for 4
tools, Tornado (reference pages) 339
Tornado command
About menu 57
Support menu 294
Tornado.tcl 290
***tr*( )** 157
training classes (WRS) 14
troubleshooting 58–62
boot display, using 51
booting problems 59–62
hardware configuration 58–59
memory fragmentation 228
memory leaks 227
NetROM ROM emulator connection 42–45
priority inversion 229
stack overflow 228
target-server problems 62–63
***ts*( )** 157
***tt*( )** 158
target-resident code, requiring 197
ttySend command (CrossWind) 274
***tw*( )** 158

U

***udpstatShow*( )** 167
***unld*( )** 163
unload command (CrossWind) 254
Unregister command (launcher) 72
Unreserve command (launcher) 73
unsupported target directory 313
unwinding (Tcl) 324
up command (CrossWind) 248
uptcl command (CrossWind) 274
userAppInit.c 116
initialization calls, adding 128
utilities, Tornado (reference pages) 437

V

variables, *see* data variables; environment variables, host; Tcl variables
version control
customizing 281
Uncheckout command, creating (example) 287
***version*( )** 160
virtual console

creating 77
virtual I/O 12
VMEbus
backplane, installing boards in 29–30
system controller 29
VX_NO_STACK_FILL 158
VxMP (option) 5
VxSim 9
VxVMI (option) 4
VxWorks 4–5
see also VxWorks Network Programmer's Guide; *VxWorks Programmer's Guide*; *VxWorks Reference Manual*
booting 45–54
configuring 24–26
documentation xxii–xxiii
optional products 13
VxMP 5
VxSim, target simulator 9
VxVMI 4
WindView 8
rebooting 54
VxWorks command (Support menu) 294

W

***w*( )** 158
watchdogs
displaying information about (browser) 219
WDB_BP_MAX 142
WDB_COMM_TYPE 35
WDB_NETROM_INDEX 138
WDB_NETROM_MTU 138
WDB_NETROM_NUM_ACCESS 139
WDB_NETROM_POLL_DELAY 139
WDB_NETROM_ROMSIZE 139
WDB_NETROM_TYPE 139
WDB_NETROM_WIDTH 139
***wdShow*( )** 166
.wind directory 314
wind kernel, *see* kernel
Wind River Systems logo 56
Wind River Users Group 301
WIND_BASE host environment variable 20
WIND_BUILDTOOL environment variable 290
WIND_HOST_TYPE host environment variable 20–21
WIND_REGISTRY host environment variable 19
windHelp.tcl 291
Windows menu (CrossWind) 243
Backtrace command 243
Registers command 243
WindSh 149–205
see also C interpreter; C++ support; shell
windsh command 151
Tcl interpreter, starting 198
windsh.tcl 201
WindSurf, accessing 294
WindView 8
WindView command (Support menu) 295
workspace 93
adding project to 110
Workspace window 94
project files, working with 101
workspaceName.**wsp** 116
World Wide Web
links to WRS 301
Tools menu link to WRS, creating (example) 289
WTX protocol 257
see also target servers
debugger, sending requests in 259
target server options 81
wtx-ignore-exit-status (CrossWind) 260
wtx-load-flags (CrossWind) 260
wtx-load-path-qualify (CrossWind) 260
wtx-load-timeout (CrossWind) 260
wtx-task-priority (CrossWind) 260
wtx-task-stack-size (Crosswind) 260
WTXTCL protocol 199
wtx-tool-name (CrossWind) 261

X

X Window System
color and grayscale, setting 23
customizing 471–472
target-server virtual console, using 78