## Contents

**Documentation Guide** ........................................................................................................... xxi
  - Overview .............................................................................................................................. xxi
  - Release Information ............................................................................................................ xxi
  - Tornado Host-Tool Documentation .................................................................................... xxiii
  - VxWorks Target-OS Documentation .................................................................................. xxiv
  - Online Documentation ....................................................................................................... xxv
  - Free Software Foundation Documentation ......................................................................... xxv
  - Documentation Conventions ............................................................................................... xxvi

### 1 Overview ................................................................................................................................... 1

1.1 Introduction .......................................................................................................................... 1

1.2 Cross-Development with Tornado ......................................................................................... 3

1.3 VxWorks Target Environment ............................................................................................. 4

1.4 Tornado Host IDE ................................................................................................................ 5
  - Tornado Editor ..................................................................................................................... 5
  - Project Management ............................................................................................................ 5
  - Compiler ............................................................................................................................. 6
  - WindSh Command Shell ...................................................................................................... 6
  - CrossWind Debugger .......................................................................................................... 7
  - Browser ............................................................................................................................. 8
  - WindView Software Logic Analyzer .................................................................................... 8
  - VxWorks Target Simulator ............................................................................................... 8
# Tornado 2.0 User’s Guide

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

## 1.6 Online Documentation ........................................ 12

## 1.7 Extending Tornado ............................................ 12
- Published Application Program Interfaces ............ 13
- Optional Products ......................................... 13
- Third-Party Products ..................................... 13

## 1.8 Customer Services ............................................. 14
- Customer Support ......................................... 14
- Training ....................................................... 14

## 2 Setup and Startup .................................................. 15

### 2.1 Introduction ..................................................... 15

### 2.2 Host Setup: Tornado Registry ............................... 15

### 2.3 Target Setup .................................................... 17
#### 2.3.1 Example Target Configurations ......................... 17
#### 2.3.2 Standalone PCs and Non-Networked Targets .......... 19
#### 2.3.3 Networking the Host and Target ....................... 20
   - Initializing the Host Network Software .................. 20
   - Establishing the VxWorks Target Name and IP Address 21
#### 2.3.4 Configuring the Target Hardware ....................... 21
   - Boot ROMs and Other Boot Media ....................... 22
   - Setting Board Jumpers .................................. 22
   - Board Installation and Power ......................... 23
   - Connecting a Serial Cable for Terminal Emulator .... 24
   - Connecting a Cable for Ethernet Connection ......... 24

### 2.4 Host-Target Communication Configuration ............... 24
2.4.1 Network Connections ................................................................. 27
Configuring the Target Agent For Network Connection ........ 28
2.4.2 END Connections ................................................................. 28
Configuring the Target Agent For END Connection ............ 28
2.4.3 Serial-Line Connections ....................................................... 28
Configuring the Target Agent For Serial Connection .......... 29
Configuring the Boot Program for Serial Connection .......... 29
Testing the Connection ................................................................. 29
Starting the Target Server ........................................................... 30
2.4.4 The NetROM ROM-Emulator Connection ......................... 30
Configuring the Target Agent for NetROM ......................... 32
Configuring the NetROM ............................................................ 32
Starting the Target Server .......................................................... 35
Troubleshooting the NetROM ROM-Emulator Connection ...... 35

2.5 Booting VxWorks ................................................................. 38
2.5.1 Default Boot Process ........................................................... 38
2.5.2 Entering New Boot Parameters .......................................... 39
2.5.3 Boot Program Commands ................................................. 41
2.5.4 Description of Boot Parameters ......................................... 41
2.5.5 Booting With New Parameters .......................................... 44
2.5.6 Alternate Boot Methods ...................................................... 45
2.5.7 Booting a Target Without a Network ............................... 46
2.5.8 Rebooting VxWorks ............................................................ 47

2.6 Starting Tornado ................................................................. 47
2.6.1 Toolbars and Buttons ......................................................... 48
2.6.2 Status Line ................................................................. 50
2.6.3 Window Management ....................................................... 50

2.7 Starting a Target Server ....................................................... 51

2.8 Displaying Information About the Target ......................... 52
2.9 Troubleshooting .......................... 53
  2.9.1 Things to Check ................................................................. 53
    Hardware Configuration .......................................................... 54
    Booting Problems ............................................................... 54
    Target-Server Problems ...................................................... 56
  2.9.2 Technical Support ............................................................. 57

3 Editor ................................................................. 59
  3.1 Introduction ............................................................... 59
  3.2 The Standard Toolbar ..................................................... 60
  3.3 File Management ............................................................. 61
    3.3.1 Creating a File ............................................................ 61
    3.3.2 Opening a File ........................................................... 63
    3.3.3 Saving and Closing a File ........................................... 64
  3.4 Typing and Editing .......................................................... 64
    3.4.1 Editing Mode .............................................................. 65
    3.4.2 Editing Text ............................................................... 65
    3.4.3 Navigation ................................................................. 66
    3.4.4 Search and Replace .................................................... 67
    3.4.5 Printing ...................................................................... 68
  3.5 Editor Customization ....................................................... 69

4 Projects ................................................................. 71
  4.1 Introduction ............................................................... 71
  4.2 Creating a Downloadable Application .................................. 75
    4.2.1 Creating a Project for a Downloadable Application ........... 76
    4.2.2 Project Files for a Downloadable Application .................. 80
4.2.3 Working With Application Files ...................................................... 80
Creating, Adding, and Removing Application Files .................... 81
Displaying and Modifying File Properties ..................................... 82
Opening, Saving, and Closing Files ................................................. 83
4.2.4 Building a Downloadable Application ........................................... 83
Calculating Makefile Dependencies ................................................ 84
Build Specifications ............................................................................ 85
Building an Application .................................................................... 87
4.2.5 Downloading and Running an Application .................................. 89
4.2.6 Adding and Removing Projects ....................................................... 89
4.3 Creating a Custom VxWorks Image .............................................................. 90
4.3.1 Creating a Project for VxWorks ........................................................ 90
4.3.2 Project Files for VxWorks .................................................................. 94
4.3.3 Configuring VxWorks Components ................................................ 96
Finding VxWorks Components and Configuration Macros ........ 97
Displaying Descriptions and Online Help for Components ....... 97
Including and Excluding Components ........................................... 97
Component Conflicts ......................................................................... 101
Changing Component Parameters ................................................. 102
Estimating Total Component Size ................................................... 103
4.3.4 Selecting the VxWorks Image Type ................................................. 103
4.3.5 Building VxWorks .............................................................................. 105
4.3.6 Booting VxWorks ............................................................................ 106
4.4 Creating a Bootable Application .......................................................... 106
4.4.1 Using Automated Scaling of VxWorks ........................................... 107
4.4.2 Adding Application Initialization Routines ............................ 107
4.5 Working With Build Specifications ..................................................... 107
4.5.1 Changing a Build Specification ...................................................... 108
Custom Makefile Rules ................................................................. 109
Makefile Macros .............................................................................. 110
Compiler Options ............................................................................ 111
Assembler Options ................................................................. 112
Link Order Options ................................................................. 112
Linker Options ...................................................................... 113

4.5.2 Creating New Build Specifications .............................. 114
4.5.3 Selecting a Specification for the Current Build .......... 115

4.6 Configuring the Target-Host Communication Interface ...... 116
    Configuration for an END Driver Connection .................. 117
    Configuration for Integrated Target Simulators .............. 117
    Configuration for NetROM Connection ......................... 117
    Configuration for Network Connection ......................... 119
    Configuration for Serial Connection ............................ 119
    Configuration for tyCoDrv Connection ......................... 120
    Scaling the Target Agent ............................................. 120
    Configuring the Target Agent for Exception Hooks .......... 121
    Starting the Agent Before the Kernel ......................... 121

4.7 Configuring and Building a VxWorks Boot Program ......... 123

5 Target Server ........................................................................ 127

5.1 Introduction ....................................................................... 127

5.2 Configuring and Starting a Target Server ....................... 128
    5.2.1 Using the Configure Target Servers Dialog Box ....... 129
    5.2.2 Target-Server Configuration Options .................... 131
        Authorizations ..................................................... 131
        Back End ......................................................... 132
        Core File and Symbols ....................................... 134
        Memory Cache Size .......................................... 135
        Target Server File System .................................. 136
        Console and Redirection ................................... 137
        Logging ........................................................... 138
        Miscellaneous ................................................ 139
    5.2.3 Using the Command Line ....................................... 140

5.3 Target Server Icon and Log Console Window .................. 141
6.2.7 Interrupting a Shell Command ...................................................... 173

6.3 The Shell C-Expression Interpreter ......................................................... 174
6.3.1 Data Types ..................................................................................... 174
6.3.2 Lines and Statements ....................................................................... 176
6.3.3 Expressions ...................................................................................... 176
    Literals ............................................................................................... 176
    Variable References ........................................................................... 176
    Operators ........................................................................................... 177
    Function Calls .................................................................................... 177
    Subroutines as Commands ................................................................ 179
    Arguments to Commands ................................................................... 179
    Task References .................................................................................. 179
6.3.4 The “Current” Task and Address .................................................... 180
6.3.5 Assignments .................................................................................... 180
    Typing and Assignment ..................................................................... 181
    Automatic Creation of New Variables ............................................. 181
6.3.6 Comments ....................................................................................... 182
6.3.7 Strings ............................................................................................ 182
6.3.8 Strings and Path Names ................................................................. 183
6.3.9 Ambiguity of Arrays and Pointers .................................................. 183
6.3.10 Pointer Arithmetic ......................................................................... 184
6.3.11 C Interpreter Limitations ............................................................... 184
6.3.12 C-Interpreter Primitives ................................................................. 185
6.3.13 Terminal-Control Characters ........................................................... 186
6.3.14 Redirection in the C Interpreter ....................................................... 187
    Ambiguity Between Redirection and C Operators ................................. 187
    The Nature of Redirection ................................................................... 188
    Scripts: Redirecting Shell I/O .............................................................. 188
    C-Interpreter Startup Scripts ............................................................... 190

6.4 C++ Interpretation .................................................................................. 190
6.4.1 Overloaded Function Names ............................................................ 191
6.4.2 Automatic Name Demangling ......................................................... 192
6.5 Shell Line Editing ............................................................................ 193
6.6 Object Module Load Path ............................................................... 196
6.7 Tcl: Shell Interpretation ................................................................. 197
   6.7.1 Tcl: Controlling the Target ....................................................... 198
   Tcl: Calling Target Routines ............................................................ 199
   Tcl: Passing Values to Target Routines ........................................... 199
   6.7.2 Tcl: Calling under C Control ..................................................... 199
   6.7.3 Tcl: Tornado Shell Initialization ............................................... 200
6.8 The Shell Architecture ................................................................. 201
   6.8.1 Controlling the Target from the Host ....................................... 201
   6.8.2 Shell Components ................................................................. 203
   6.8.3 Layers of Interpretation .......................................................... 204

7 Browser .............................................................................................. 207
   7.1 A System-Object Browser ........................................................... 207
   7.2 Starting the Browser ................................................................. 208
   7.3 Anatomy of the Browser Window ............................................... 208
   7.4 Browser Buttons ................................................................. 210
   7.5 Target-Information Window ..................................................... 211
   7.6 Task-List Window ................................................................. 213
   7.7 Memory-Usage Window .......................................................... 214
   7.8 Object-Information Windows ................................................... 215
      7.8.1 The Task Browser ............................................................ 216
      7.8.2 The Semaphore Browser .................................................. 217
7.8.3 The Message-Queue Browser .......................................................... 218
7.8.4 The Memory-Partition Browser ....................................................... 219
7.8.5 The Watchdog Browser ................................................................. 219
7.8.6 The Class Browser ............................................................................. 220

7.9 The Module-Information Window ....................................................... 221
7.10 The Spy Window .................................................................................. 224
7.11 The Stack-Check Window ..................................................................... 225
7.12 The Vector Table Window ..................................................................... 226
7.13 Browser Displays and Target Link Speed ............................................ 227
7.14 Troubleshooting with the Browser ...................................................... 227
    7.14.1 Memory Leaks ........................................................................... 227
    7.14.2 Memory Fragmentation .............................................................. 228
    7.14.3 Stack Overflow .......................................................................... 228
    7.14.4 Priority Inversion ....................................................................... 228

7.15 Tcl: the Browser Initialization File ..................................................... 232

8 Debugger .................................................................................................. 233

8.1 Introduction .......................................................................................... 233
8.2 Debugger GUI ....................................................................................... 234
    8.2.1 Debugger Toolbar, Buttons, and Menu Commands ..................... 235
8.3 Using the Debugger ............................................................................... 237
    8.3.1 Starting and Stopping the Debugger .......................................... 238
    8.3.2 Setting the Search Path ............................................................... 239
    8.3.3 Unloading a Module .................................................................. 240
    8.3.4 Running a Program .................................................................. 240
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3.5</td>
<td>Interrupting a Program</td>
</tr>
<tr>
<td>8.3.6</td>
<td>Setting Breakpoints</td>
</tr>
<tr>
<td>8.3.7</td>
<td>Continuing Through a Program</td>
</tr>
<tr>
<td>8.3.8</td>
<td>Attaching to a Running Task</td>
</tr>
<tr>
<td>8.3.9</td>
<td>Detaching from a Running Task</td>
</tr>
<tr>
<td>8.3.10</td>
<td>Examining Data, Memory, and the Stack</td>
</tr>
<tr>
<td>8.4</td>
<td>Source Code Display Options</td>
</tr>
<tr>
<td>8.5</td>
<td>Using the Debugger Command Line</td>
</tr>
<tr>
<td>8.5.1</td>
<td>GDB Initialization Files</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Selecting Modules to Debug</td>
</tr>
<tr>
<td>8.5.3</td>
<td>Running a Program</td>
</tr>
<tr>
<td>8.5.4</td>
<td>Re-Setting Application I/O</td>
</tr>
<tr>
<td>8.5.5</td>
<td>Using Graphically Enhanced Commands</td>
</tr>
<tr>
<td>8.5.6</td>
<td>Managing Targets</td>
</tr>
<tr>
<td>8.5.7</td>
<td>Extended Debugger Commands</td>
</tr>
<tr>
<td>8.5.8</td>
<td>Extended Debugger Variables</td>
</tr>
<tr>
<td>8.6</td>
<td>System-Mode Debugging</td>
</tr>
<tr>
<td>8.6.1</td>
<td>Entering System Mode</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Thread Facilities in System Mode</td>
</tr>
<tr>
<td></td>
<td>Displaying Summary Thread Information</td>
</tr>
<tr>
<td></td>
<td>Switching Threads Explicitly</td>
</tr>
<tr>
<td></td>
<td>Thread-Specific Breakpoints</td>
</tr>
<tr>
<td></td>
<td>Switching Threads Implicitly</td>
</tr>
<tr>
<td>8.7</td>
<td>Tcl: Debugger Automation</td>
</tr>
</tbody>
</table>
### 8.7.1 Tcl: A Simple Debugger Example

8.7.2 Tcl: Specialized GDB Commands

8.7.3 Tcl: Invoking GDB Facilities

8.7.4 Tcl: A Linked-List Traversal Macro Example

### 8.8 Tcl: Debugger Customization

8.8.1 Tcl: Debugger Initialization Files

8.8.2 Tcl: Passing Control between the Two Tornado Debugger Interpreters

8.8.3 Tcl: Experimenting with Tornado Debugger Extensions

Tcl: An Add-Symbols Command for the File Menu

Tcl: “this” Menu Command for C++

### 9 Customization

9.1 Introduction

9.2 Setting Options

9.2.1 Setting Download Options

9.2.2 Editor Preferences

9.2.3 External Editor

9.2.4 Project

9.2.5 Version Control

9.2.6 Fonts/Colors

9.2.7 Debugger

9.2.8 Tornado Registry

9.3 Customizing the Tools Menu

9.3.1 The Customize Tools Dialog Box

Macros for Customized Menu Commands

9.3.2 Examples of Tools Menu Customization
| Contents |
|-----------------|------------------|
| Version Control | 298              |
| Alternate Editor | 299              |
| Binary Utilities | 300              |
| World Wide Web  | 300              |
| 9.3.3 Customizing the Build Menu | 301 |
| 9.4 Tcl Customization Files | 303 |
| Tornado Initialization | 303 |
| HTML Help Initialization | 304 |
| 10 Customer Service | 305 |
| 10.1 Introduction | 305 |
| 10.2 WRS Support Services | 306 |
| 10.2.1 The User Profile | 306 |
| 10.2.2 Sending a Tornado Support Request (TSR) | 308 |
| History Area | 309 |
| Host Information | 309 |
| Target Information | 310 |
| Problem Information | 310 |
| Action Buttons | 311 |
| 10.3 WRS Broadcasts on the World Wide Web | 312 |
| Appendices | 315 |
| A Directories and Files | 317 |
| A.1 Introduction | 317 |
| A.2 Host Directories and Files | 318 |
| A.3 Target Directories and Files | 319 |
| A.4 Initialization and State-Information Files | 325 |
| B Tcl | 327 |
B.1 Why Tcl? ............................................................................................................ 327
B.2 A Taste of Tcl ..................................................................................................... 328
B.2.1 Tcl Variables ........................................................................................ 329
B.2.2 Lists in Tcl ........................................................................................... 330
B.2.3 Associative Arrays ............................................................................. 331
B.2.4 Command Substitution ..................................................................... 331
B.2.5 Arithmetic ........................................................................................... 332
B.2.6 I/O, Files, and Formatting ............................................................... 332
B.2.7 Procedures .......................................................................................... 333
B.2.8 Control Structures .............................................................................. 334
B.2.9 Tcl Error Handling ............................................................................. 334
B.2.10 Integrating Tcl and C Applications ................................................. 335
B.3 Tcl Coding Conventions .................................................................................. 335
B.3.1 Tcl Module Layout ............................................................................. 335
B.3.2 Tcl Procedure Layout ........................................................................ 336
B.3.3 Tcl Code Outside Procedures ........................................................... 338
B.3.4 Declaration Formats .......................................................................... 339
  Variables .............................................................................................. 339
  Procedures .......................................................................................... 340
B.3.5 Code Layout ....................................................................................... 340
  Vertical Spacing .................................................................................. 340
  Horizontal Spacing ............................................................................ 341
  Indentation ......................................................................................... 341
  Comments ........................................................................................... 343
B.3.6 Naming Conventions ........................................................................ 343
B.3.7 Tcl Style ............................................................................................... 344

C Tornado Tools Reference .................................................................................. 349

D Utilities Reference .......................................................................................... 433
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)\(^1\). The Tornado IDE also includes context-sensitive online help.

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 Tornado 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—editor, project facility, target server management, shell, browser, and the debugger.
- A chapter on customizing Tornado.
- Appendices on Tornado directories and files, the use of Tcl (Tool Command Language) in Tornado, and reference information for host tools.

Tornado API 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.

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 in the Tornado IDE. 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 for the Tornado graphical debugger, CrossWind.

NOTE: The FSF develops software under UNIX, and examples in their manuals reflect this. Nevertheless, the GNU tools operate reliably under Windows.

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 path names starting at the directory c:\tornado. 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 Windows 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 \texttt{printf()}. Combinations of keys that must be pressed simultaneously are shown with a + linking the keys. For example, \texttt{CTRL+F3} means to press the key labeled \texttt{CTRL} and the key labeled \texttt{F3}.

<table>
<thead>
<tr>
<th>Term</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>files, path names</td>
<td>\texttt{c:\tornado\host}</td>
</tr>
<tr>
<td>libraries, drivers</td>
<td>\texttt{memLib.c}</td>
</tr>
<tr>
<td>Command-prompt tools</td>
<td>\texttt{dir}</td>
</tr>
<tr>
<td>Tcl procedures</td>
<td>\texttt{wtxMemRead}</td>
</tr>
<tr>
<td>subroutines</td>
<td>\texttt{semTake()}</td>
</tr>
<tr>
<td>VxWorks boot commands</td>
<td>\texttt{p}</td>
</tr>
<tr>
<td>code display</td>
<td>\texttt{main ();}</td>
</tr>
<tr>
<td>keyboard input display</td>
<td>\texttt{wtxregd -V}</td>
</tr>
<tr>
<td>output display</td>
<td>\texttt{value = 0}</td>
</tr>
<tr>
<td>user-supplied values</td>
<td>\texttt{name}</td>
</tr>
<tr>
<td>constants</td>
<td>\texttt{INCLUDE_NFS}</td>
</tr>
<tr>
<td>keywords</td>
<td>\texttt{struct}</td>
</tr>
<tr>
<td>named key on keyboard</td>
<td>\texttt{RETURN}</td>
</tr>
<tr>
<td>key combinations</td>
<td>\texttt{ALT+SHIFT+F5}</td>
</tr>
<tr>
<td>lower-case acronyms</td>
<td>\texttt{fd}</td>
</tr>
<tr>
<td>GUI titles and commands</td>
<td>\texttt{Help}</td>
</tr>
<tr>
<td>GUI menu path</td>
<td>\texttt{Tools:Target Server:Configure}</td>
</tr>
</tbody>
</table>
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).
- An integrated development environment (IDE) that facilitates managing and building projects, establishing and managing host-target communication, and running, debugging, and monitoring VxWorks applications.

The Tornado IDE is the main focus of this manual. Key features of the IDE are:

- An integrated source-code editor.
- 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 software logic analyzer for the target simulator.
- Customization options for many features, including integration of alternate editors and configuration management (CM) tools, as well as the entire Tornado GUI itself.
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.
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 IDE

Tornado integrates the various aspects of VxWorks programming into a single environment for developing and debugging VxWorks applications. The Tornado IDE 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 the IDE.

**Tornado Editor**

The Tornado source-code editor includes the following features:

- Standard text manipulation capabilities.
- C and C++ syntax-element color highlighting.
- Debugger integration: the editor window tracks code execution.
- Compiler integration: the project-management utility links compiler warnings and errors directly to the affected source in an editor window.

The Tornado editor is described in 3. Editor.

**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.

**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 6. 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 8. Debugger. Also see GDB User’s Guide.
Browser

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

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

The browser is described in 7. 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.

For information about how to configure, start, and manage target servers see 5. Target Server and the tgtsvr reference 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. A registry should never be killed; without a registry, target servers cannot be named, and no Tornado tool can connect to a target.

For information about the Tornado registry, see 2.2 Host Setup: Tornado Registry, p.15.

**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 *Console and Redirection*, p.137 for information about how to configure a target server with a virtual console.

### 1.6 Online Documentation

Tornado online documentation includes a context-sensitive help system, online versions of all manuals, and a search facility.

- The help system provides context-sensitive help on dialog boxes, menu commands, and windows.
- 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 IDE. 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 dynamic link 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 10. 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 10. 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.
2.1 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 2.2 Host Setup: Tornado Registry, p.15 and then proceed with 2.5 Booting VxWorks, p.38.

2.2 Host Setup: Tornado Registry

The Tornado target server registry is a service that keeps track of target servers and provides your host with access to them. The registry1 must always be running; otherwise Tornado tools cannot communicate with targets. Because Tornado development tools are independent of the method of communication with the

1. The Tornado registry program is the file c:\tornado\host\x86-win32\bin\wtxregd.exe. The Windows NT service version is wtxregs.exe.
target, the registry is required regardless of whether the target communicates over a network or serial links.

The Tornado registry need not run on the same host as your tools, as long as it is accessible on the network. In fact, 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 you attempt to start a registry when one is already running on the same host, the new registry automatically detects that it is not needed. It displays the Tornado Registry window with a warning message, and then shuts itself down when you click the associated OK button.

When the Tornado registry is running on your host system, the registry icon is displayed in the Windows taskbar (except when the registry is running as a Windows service). The context menu for the icon provides options for displaying the window, displaying version information about the registry, and shutting down the registry. Double click on the icon to display the Tornado Registry window (Figure 2-1).

Figure 2-1  Tornado Registry Window and Icon

The Tornado Registry window displays log information and a list of all the target servers that have been registered with it, including information about the target system and the user. The Hide button hides the Tornado Registry window. The Kill Registry button shuts down the registry. The About button displays version information for the registry. The Refresh button refreshes information about the entries in the registry.
Your 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.

After installation you can select to use a different Tornado registry with the Tornado Registry page of the Options window (Tools>Options>Tornado Registry).

If you did not set up the registry as a Windows service when you installed Tornado with the SETUP program, you can use a Tornado service utility to do so (see Windows NT Service Manager, p.459).

For detailed information about the operation of the Tornado registry, and its command options, see the wtxregd reference page in Tornado Tools Reference, p.435.

2.3 Target Setup

This section covers bringing up VxWorks on a target with a relatively simple configuration. The VxWorks Programmer’s Guide and the VxWorks Network Programmer’s Guide elaborate 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.

2.3.1 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 2-2 illustrates a minimal cross-development configuration: the target is a bare board, connected to the host development system by a single serial line. For a configuration of this sort, use a combination of a boot mechanism that does not require a network and an alternative Tornado communications back end. See 2.3.2 Standalone PCs and Non-Networked Targets, p.19 for more information.

Another target system configuration, representing a more resource-rich development environment, is shown in Figure 2-3. This environment corresponds to the default VxWorks run-time development configuration.
The configuration in Figure 2-3 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.
Tornado PC: A PC that runs the Tornado tools, and includes a serial connection to the target (used by the boot program for initial setup).
Setup and Startup

2.3.2 Standalone PCs and Non-Networked Targets

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

**WARNING:** Tornado tools such as the shell and debugger use the TCP/IP protocol to communicate with one another. Thus, you must have TCP/IP installed on your host even if it is not a networked system. To install TCP/IP on your PC as part of the Windows network support, following the instructions in your Windows documentation.

The following steps refer to configuration choices that are required during Windows TCP/IP installation:

1. Select a physical device for the TCP/IP binding. When there is no networking hardware, select Dial-up Adapter.
2. TCP/IP configuration requires that you specify an IP address; you can choose any arbitrary address when you are not connecting to any other hosts.
3. Choose a name for your system, and enter it as the host name.
4. Disable Domain Name Service (DNS), because the service is not available.
5. Record your host name and associated IP address in the *hosts* file. (On Windows 95, c:\windows\hosts; on Windows NT, c:\winnt\system32\drivers\etc\hosts.)

These steps produce a functioning TCP/IP subsystem on your host. Because your machine is not networked, the Tornado services must all run locally: make sure that the registry is running on your system (see 2.2 Host Setup: Tornado Registry, p.15).

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 2.4.3 Serial-Line Connections, p.28.

2.3.3 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 2.4 Host-Target Communication Configuration, p.24.

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.

Initializing the Host Network Software

The TCP/IP networking package should already be installed on the Windows host where you are configuring Tornado; TCP/IP is generally installed when the operating system is first installed. If TCP/IP is not yet installed on your Tornado host, install it now. Consult your Windows documentation on installing and configuring TCP/IP for your PC.

If you are planning to boot VxWorks over the network (the default VxWorks-target boot configuration), you must have an FTP server running on the host where the VxWorks system image is stored, and the FTP server must have a user ID and password defined that your VxWorks target can use to identify itself.

On Windows 95, Tornado includes an FTP-server application, wftpd32.exe. See F. FTP Server for information on configuring this FTP server.

On Windows NT, we recommend that you install the FTP server as an NT service when you install TCP/IP.
Establishing the VxWorks Target Name and IP Address

With TCP/IP installed, you can configure the server that provides Domain Name Service (DNS) so that your Windows computer uses that server to translate system names to network IP addresses. Consult your Windows documentation on how to configure your system to take advantage of DNS.

If you do not have a domain name server at your site, you can specify how to map machine names to IP addresses in a file (discussed below) on your machine. Otherwise, you must identify targets by IP address.

Windows uses a file called hosts to record the names and IP network addresses of systems accessible on the network from the local system. The location of this file depends on which version of Windows you use:

- Windows 95 and 98: the hosts file is c:\windows\hosts
- Windows NT: the hosts file is c:\winnt\system32\drivers\etc\hosts

Each line consists of an IP address and the name (or names) of the system at that address.

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 hosts file must then contain the following lines:

```
90.0.0.1 mars
90.0.0.50 phobos
```

2.3.4 Configuring the Target Hardware

Configuring the target hardware may involve the following tasks:

- Setting up a boot mechanism.
- Jumpering the target CPU board, and any auxiliary boards (for example, Ethernet).
- 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 IDE; the file c:\tornado\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.

For specific information on a BSP’s booting method, see Help>Manuals Contents>BSP Reference.

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. For information about how to use NetROM on your target, refer to Configuration for NetROM Connection, p.117. Contact the nearest Wind River Systems office (see back cover) for information about support for other ROM emulators.

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

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

See 4.7 Configuring and Building a VxWorks Boot Program, p.123 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

2. NetROM is a trademark of Applied Microsystems Corporation.
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>BSP Reference in the Tornado IDE; the file c:\tornado\docs\BSP_Reference.html).

**Board Installation and Power**

For bare-board targets, use the power supply recommended by the board manufacturer.

If you are using a VME chassis, first install the CPU board in the first slot of the backplane (Figure 2-4). If you are using a separate Ethernet controller board, install it in the second slot of the backplane.

**Figure 2-4  Assembling VME Targets**

![Diagram of VME Target](image)

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 require the extended address and data lines of row B of the P2 bus.

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

Empty Slots

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

Connecting a Serial Cable for Terminal Emulator

Most VxWorks targets include at least one on-board serial port. This serial port must be connected to a terminal emulator (HyperTerminal), 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.

Tornado comes with terminal-emulator configurations already set up for connecting to VxWorks targets on either COM1 or COM2, in the same program folder as other Tornado programs. Use VxWorks COM1 if the serial connection from your target is to COM1, or VxWorks COM2 if the target is connected to COM2.

Connecting a Cable for Ethernet Connection

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

2.4 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 2-5 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 (2.5 Booting VxWorks, p.38).

All of the standard target server back ends included with Tornado connect to the target through the 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).

- 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 Tornado debugger (CrossWind; see 8.6 System-Mode Debugging, p.263) or the shell (6.2.6 Using the Shell for System Mode Debugging, p.169). 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 or a ROM-emulator 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 exploit a polled-mode serial driver for 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 the 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 2.5.7 Booting a Target Without a Network, p.46. Certain BSPs may provide other alternatives, such as flash memory; see the reference information for your BSP.

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.

### 2.4.1 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 `wdbrpc` 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.116 for information about configuring VxWorks for various target agent communications interfaces.

### 2.4.2 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.117 for information about configuring the VxWorks target agent for an END connection.

### 2.4.3 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.119).

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.123). The simplest way to boot over a serial connection is by using the Target Server File System. See 2.5.7 Booting a Target Without a Network, p.46.

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 to the target-agent serial port, then reset the target (see Connecting a Serial Cable for Terminal Emulator, p.24). If the WDB READY message does not appear, or if it is garbled, check the configuration of the serial port you are using on your host.

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

```c
#ifdef INCLUDE_WDB_TTY_TESTS
 /* 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_TESTS */
```

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 COM 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:** This configuration change also prevents the target from completing its boot process until a target server attaches to it. Thus, it is best to change the `wdbSioTest()` calls back to the default 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, specify the communications port with `-d`, and also specify the line speed to match the speed configured into your target:

   ```
   C:\> tgtsvr -V targetname -d com1 -B wdbserial -bps 38400
   ```

   You can also use the Tornado GUI to configure and start a target server (see 5.2 Configuring and Starting a Target Server, p.128).

### 2.4.4 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 2-6 illustrates this connection method.

Figure 2-6  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 2.5.7 Booting a Target Without a Network, p.46.
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. 117.

Configuring the NetROM

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

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

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

   There are two ways of establishing an IP address for the NetROM:
   - Connect a terminal to the NetROM serial console, and specify the IP address manually when you power up the NetROM (for Step 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.

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.
   - 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 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**

```plaintext
begin
  ifconfig le0 147.11.46.164 netmask 255.255.255.0 broadcast 147.11.46.0
  setenv filetype srecord
  setenv loadpath c:\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.

For more information regarding NetROM boot requirements, refer to NetROM documentation. Consult your system administrator to configure a network host to reply to RARP or BOOTP requests.

3. Connect the NetROM to your Ethernet network, and plug NetROM pods into the target-board ROM sockets.

**WARNING:** Do not power up either the NetROM or the target yet. Pod connections and disconnections should be made while power is off on both the NetROM and the target board.
WARNING: Some board sockets are designed to support either ROM or flash PROM. On this kind of socket, a 12V potential is applied to pin 1 each time the processor accesses ROM space. This potential may damage the NetROM. In this situation, place an extra ROM socket with pin 1 removed between the NetROM pod and the target-board socket.

WARNING: Take great care when you plug in NetROM pod(s). Double check the pod connections, especially pin 1 position and alignment. A pod connection error can damage either the NetROM itself, the target board, or both. The pins coming out of the NetROM’s DIP emulation pods are very easy to break, and the cables are expensive to replace. It is a good idea to use a DIP extender socket, because they are much cheaper and faster to replace if a pin breaks.

NetROM pod 0 differs from other pods because it implements the dual-port RAM. This special port is used by NetROM both to send data to the board and to receive data from the board: that is, the dual port is the communication path between the NetROM and the board.

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

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

The NetROM begins a telnet connection with the following prompt:

```
NETROM TELNET
NetROM>
```

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

5. Download test code to the NetROM.

One method is to type the `newimage` command at the NetROM prompt. This command uses the TFTP protocol to download the image specified by the `loadfile` environment variable from the path specified by the `loadpath` environment variable (which is `c:\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 the `download.exe` utility from AMC (see the AMC NetROM documentation).

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.

```
C:\> tgtsvr -V 90.0.0.5 -B netrom
```

In this example, `90.0.0.5` is the IP address of the NetROM. (You can also use the Tornado GUI to configure and start a target server; see 5.2 Configuring and Starting a Target Server, p.128.)

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 Configuring the Target Agent for NetROM, p.32.

**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:

```c
int wdbNetromTest = 0;
```

to:

```c
int wdbNetromTest = 1;
```

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 troubleshooting tips:

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.
2.5 Booting VxWorks

Once you have configured your host software and target hardware, you are ready to start a terminal emulator and boot VxWorks.

Select either VxWorks COM1 or VxWorks COM2 from the same program folder as other Tornado programs, according to whether the serial connection from your target is to COM1 or COM2 of your host PC. (See Connecting a Serial Cable for Terminal Emulator, p. 24.)

**WARNING:** If you are using a VxWorks image configured for a network connection (the default), you must have an FTP server running on the host where the VxWorks system image is stored. See Initializing the Host Network Software, p. 20 for more information.

2.5.1 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.3

When you power on the target hardware (and each time you reset it), the target system executes the boot program from ROM; during the boot process, the target uses its serial port to communicate with your terminal or workstation. The boot program first displays a banner page, and then starts a seven-second countdown, visible on the screen as shown in Figure 2-7. 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.

---

3. 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. 123). In the course of your developing an application, you will also build bootable applications (see 4.4 Creating a Bootable Application, p. 106).
2.5.2 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, as follows:

```
Please any key to stop auto-boot...
```

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, as follows:

```
Please any key to stop auto-boot...
```
To display the current (default) 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 2-8. (The **p** command does not actually display the lines with blank fields, although this example shows them for completeness.)

```
boot device         : ln
processor number    : 0
host name           : mars
file name           : c:\tornado\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)   : secret
flags (f)           : 0x0
target name (tn)    : phobos
startup script (s)  :
other (o)           :
```

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 ENTER. To clear a field, enter a period (.), then press ENTER. If you want to quit before completing all the parameters, type **CTRL+D**.

Network information must be entered to match your particular system configuration. The Internet addresses must match those in your system's hosts file (or those known to your Domain Name Server), as described in Establishing the VxWorks Target Name and IP Address, p. 21.

If your target has nonvolatile RAM (NVRAM), the boot parameters are stored there and retained 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.
2.5.3 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 ENTER:

[VxWorks Boot]: ?

Table 2-1 describes each of the VxWorks boot commands and their arguments.

2.5.4 Description of Boot Parameters

Each of the boot parameters is described below, with reference to the example in 2.5.2 Entering New Boot Parameters, p.39. The letters in parentheses after some parameters indicate how to specify the parameters in the command-line boot procedure described in 2.5.6 Alternate Boot Methods, p.45.

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 target identifier 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 2.5.2 Entering New Boot Parameters, p.39.)

file name
The full pathname of the VxWorks object module to be booted (c:\tornado\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.4

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

---

4. 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 5.6 Managing a Target Server, p.143.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>Help command—print a list of available boot commands.</td>
</tr>
<tr>
<td>?</td>
<td>Same as h.</td>
</tr>
<tr>
<td>@</td>
<td>Boot (load and execute the file) using the current boot parameters.</td>
</tr>
<tr>
<td>p</td>
<td>Print the current boot parameter values.</td>
</tr>
<tr>
<td>c</td>
<td>Change the boot parameter values.</td>
</tr>
<tr>
<td>l</td>
<td>Load the file using current boot parameters, but without executing.</td>
</tr>
<tr>
<td>g adrs</td>
<td>Go to (execute at) hex address adrs.</td>
</tr>
<tr>
<td>d adrs[, n]</td>
<td>Display n words of memory starting at hex address adrs. If n is omitted, the default is 64.</td>
</tr>
<tr>
<td>m adrs</td>
<td>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:</td>
</tr>
<tr>
<td>f adrs, nbytes, value</td>
<td>Fill nbytes of memory, starting at adrs with value.</td>
</tr>
<tr>
<td>t adrs1, adrs2, nbytes</td>
<td>Copy nbytes of memory, starting at adrs1, to adrs2.</td>
</tr>
<tr>
<td>s [ 0</td>
<td>1 ]</td>
</tr>
<tr>
<td>e</td>
<td>Display a synopsis of the last occurring VxWorks exception.</td>
</tr>
<tr>
<td>n netif</td>
<td>Display the address of the network interface device netif.</td>
</tr>
</tbody>
</table>
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 permission to read the VxWorks boot-image file. VxWorks must have access to this user’s FTP signon, with the FTP password provided below.

ftp password (pw)
The “user” password. This field is not required by the boot program, but you must supply it to boot over the network from a Windows host. (If you do not supply this password, the boot ROM attempts to load the run-time system image using a protocol based on the UNIX rsh utility, which is not available for Windows hosts.)

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.

2.5.5 Booting With New Parameters

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

[VxWorks Boot]: @

Figure 2-9 VxWorks Booting Display

Figure 2-9 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:
2.5.6 Alternate Boot Methods

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:c:\tornado\target\config\bsp\vxWorks e=90.0.0.50 h=90.0.0.1 u=fred pw=...
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.123 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 place boot code in flash require TFTP on the host in order to burn a new VxWorks image into flash. Tornado 2.0 ships with a version of TFTP. See your vendor documentation on how to burn flash for these boards.

2.5.7 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.123.
Before you boot the target, configure a target server with the TSFS option and start it. See Target Server File System, p. 136.

The only boot parameters required to boot the target are **boot device** and **file name** (see 2.5.4 Description of Boot Parameters, p. 41). 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.

### 2.5.8 Rebooting VxWorks

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

- Entering **CTRL+X** in the serial terminal window. (This works only if you use the preconfigured terminal emulation icons shipped with Tornado; other terminal emulators do not pass **CTRL+X** to the target, because of its standard Windows meaning.)
- Invoking **reboot( )** from the Tornado shell.
- Pressing the reset button on the target system.
- Turning 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.

### 2.6 Starting Tornado

To start Tornado, click on the Start button on the Windows taskbar and select Programs. Then select the Tornado program group (called Tornado with default installation), and click on Tornado.

When you first start Tornado, the project facility **Create Project** window is displayed (Figure 2-10).
2.6.1 Toolbars and Buttons

Five toolbars are at the top of the main window; the toolbars provide quick access to the most frequently used Tornado commands. Figure 2-11 shows the default toolbar configuration.

You can dock any toolbar at the top, side, or bottom edge of the Tornado window. You can also drag any toolbar away from the edges of the window to make it a floating palette, which you can place anywhere on the screen. To move a toolbar, position the mouse pointer between any of its buttons (or on the title bar of a floating toolbar), and then drag the toolbar to the position you prefer.

For a concise description of the purpose of any toolbar button, hold the mouse pointer over it for a short interval to display a tooltip window with the name of the button.

The Tornado Launch toolbar has a pull-down list box that shows all the target servers that are currently running and known to the Tornado registry that your
system is using (shown in Figure 2-12 as a floating palette). If no target servers are listed, or none of the ones listed represent the target you need, you must configure and start a target server.

The toolbar also contains the following buttons to launch the most frequently used interactive tools on the currently selected target.

Table 2-2  Tornado Launch Toolbar Buttons

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Starts a browser for the selected target." /></td>
<td>Starts a browser for the selected target. 7. Browser describes how to use the browser.</td>
</tr>
<tr>
<td><img src="image" alt="Starts a Tornado shell for the selected target." /></td>
<td>Starts a Tornado shell for the selected target. 6. Shell describes how to use the shell.</td>
</tr>
<tr>
<td><img src="image" alt="Starts the debugger." /></td>
<td>Starts the debugger. 8. Debugger describes how to use the debugger.</td>
</tr>
<tr>
<td><img src="image" alt="Starts the VxWorks target simulator." /></td>
<td>Starts the VxWorks target simulator. The Tornado Getting Started Guide and 4. Projects include information about using the simulator.</td>
</tr>
<tr>
<td><img src="image" alt="Starts WindView." /></td>
<td>Starts WindView. The WindView User’s Guide describes how to use the WindView software logic analyzer.</td>
</tr>
</tbody>
</table>
When you first start Tornado, most of the buttons in toolbars are disabled. Tornado enables and disables buttons and menu commands so that only the commands that are currently meaningful are enabled. For example, at first the only buttons enabled are those for creating or opening a file, and displaying Tornado help. Once a file is open, more buttons become enabled; when you make a selection, the buttons that act on selections become active, and so on.

### 2.6.2 Status Line

The fields at the bottom right of the status line (at the bottom of the main Tornado window) provide information about the current state of keyboard toggles. These fields display the following indicators:

- **CAP** when **CAPS LOCK** restricts the keyboard to upper case; blank otherwise.
- **NUM** when **NUM LOCK** sets the number pad to numeric; blank when the number pad acts as a cursor pad.
- **SCRL** when **SCROLL LOCK** is on; blank otherwise.
- **OVR** when the editor is in Overtype mode.
- **READ** when a read-only file is in the active window.
- The row and column position of the insertion point in the active window.

### 2.6.3 Window Management

Once a Tornado session is in progress, you are likely to have many specialized windows open or iconized within Tornado: one or more editor windows, browser windows inspecting assorted system objects, disassembly windows, build output windows, and so on. To manage these sub-windows, use the commands in the Window menu.

The top pane of the Window menu contains commands to rearrange or dismiss all open sub-windows.
Cascade
  Arrange all open windows so that they overlap, with the top and left of each window visible.

Tile
  Arrange all open windows to occupy the entire Tornado workspace, without overlapping.

Arrange Icons
  Line up all iconized windows at the bottom of the workspace.

Close All
  Dismiss all windows.

The remainder of the Window menu selects particular sub-windows. Build Output is always present, and opens a window that displays the output of the most recent build command from the Project menu.

At the bottom of the Window menu is a list of all currently available windows. Click on any item in the list to display its window.

2.7 Starting a Target Server

A target server manages communications between Tornado host tools and the VxWorks target agent (or an alternate agent) on the target system. A target server must be configured for the target, and started, before the host tools can interact with the target.

The target server must be configured with the same communication back end as the one built into the VxWorks image. Communication back ends include drivers for specific modes of communication between the host and target, such a network (IP), serial line, and so on.

The default communication back end for the VxWorks images shipped with Tornado is for a network connection. Configuring a target server for the default connection and image simply involves identifying the IP address of the target. It is also useful to provide an alternative to the default target server name.

To configure a target server, select Tools>Target Server>Configure. Tornado opens the Configure Target Servers dialog box (Figure 2-13).
Click the **New** button. Then enter a name for the configuration in the **Description** field, and the IP address or network name for the target in the **Target IP Name/Address** field. This name will appear in the drop-down list of the Tornado Launch toolbar.

Click on the **Launch** button to start the target server.

⚠️ **CAUTION:** If you are going to use a host-target connection other than the default one for a network connection, see §5.2 **Configuring and Starting a Target Server**, p. 128 and §4.6 **Configuring the Target-Host Communication Interface**, p. 116.

### 2.8 Displaying Information About the Target

You can check whether everything is working properly by select your target server from the drop-down list box in the Tornado Launch toolbar, and clicking the ![Info Icon]
button next to it. A Browser window similar to the one shown in Figure 2-14 should appear, displaying summary information about the target.

Figure 2-14  Browser Display

![Browser Display](image)

2.9 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 2.9.1 Things to Check, p.53 before contacting Wind River customer support. Often, you can locate the problem just by re-checking the installation steps, your hardware configuration, and so forth.

2.9.1 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.

⚠️ CAUTION: Booting systems with complex network configurations is beyond the scope of this chapter. See *VxWorks Network Programmer’s Guide*. 
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:

```
Error loading file: errno = 0x33.
```

0x33 corresponds to *errno* 51 (decimal) `ENETUNREACH`. (This is one of the POSIX error codes, defined for VxWorks in `c:\tornado\target\h\errno.h`.) If the target Internet address is not in the appropriate *hosts* file (or the DNS 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 FTP server permissions.**

Is the FTP server configured correctly? See *F. FTP Server* for more information on configuring the FTP Server under Windows 95. On Windows NT, consult your system documentation on FTP Server.

• **Helpful Troubleshooting Tools**

In tracking down configuration problems, the following network tools are helpful:

**ping**

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

```
C:\> ping venus
Pinging venus.wrs.com [91.0.10.1] with 32 bytes data:
```
Tornado 2.0
User’s Guide

arp -a
This command displays the “address resolution protocol” tables that map Internet addresses to physical (Ethernet) addresses. Your target machine is listed if at least one packet was transferred from your target to your host. The following example shows both the Internet address (91.0.10.1) and physical address (08-00-20-1b-66-e9) of venus.

C:\> arp -a
Interface: 91.0.10.26
Internet Address          Physical Address      Type
91.0.10.1           08-00-20-1b-66-e9     dynamic
91.0.10.20          00-20-af-52-1e-72     dynamic
91.0.10.254         00-00-ef-01-f1-a0     dynamic

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.

C:\> netstat -r
Route Table
Network Address          Netmask  Gateway Address   Interface  Metric
0.0.0.0          0.0.0.0      91.0.10.254  91.0.10.26       1
127.0.0.0        255.0.0.0        127.0.0.1   127.0.0.1       1
91.0.10.0    255.255.255.0       91.0.10.26  91.0.10.26       1
91.0.10.26  255.255.255.255        127.0.0.1   127.0.0.1       1
91.11.255.255  255.255.255.255       91.0.10.26  91.0.10.26       1
224.0.0.0        224.0.0.0       91.0.10.26  91.0.10.26       1
255.255.255.255  255.255.255.255       91.0.10.26  91.0.10.26       1

Active Connections
Proto  Local Address          Foreign Address        State
TCP    mercury:1025           saturn.wrs.com:nbsession ESTABLISHED
TCP    mercury:1177           earth.wrs.com:nntp     ESTABLISHED
TCP    mercury:1259           oak.oakland.edu:ftp    ESTABLISHED

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 (see Core File and Symbols, p. 134) or the equivalent -c option to `tgtsvr (D. Tornado Tools Reference)` to specify the path to a host-resident copy of the VxWorks image.

### 2.9.2 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 customer support organization. Your comments and suggestions are welcome as well.
3.1 Introduction

The Tornado source-code editor includes standard text manipulation capabilities, as well as the following specialized features:

- C and C++ syntax-element color highlighting.
- Debugger integration: the editor window tracks code execution.
- Compiler integration: links to the editor window from compiler messages.

The Tornado editor also provides features tailored to the programming environment. You can have the editor display program syntactic elements such as C or C++ keywords, preprocessor directives, and comments in color. Because the editor is integrated with the Tornado debugger (8. Debugger), the editor also keeps pace automatically with program execution during debugging sessions.

You can work on as many files simultaneously as your computer’s memory allows. This is convenient when you are working with more than one module, or want to edit both a source and header file at the same time.

Typically, developers use the Tornado editor to work with source files, header files, and makefiles. However, because it is a text editor, it can also be used on any text file. For example, you can view a bug report from one of your users, or save a note in a text file.

The Tornado editor uses standard Windows editing commands and conventions. In describing the features and usage of the editor, we assume you are already familiar with these conventions.
3.2 The Standard Toolbar

The Standard toolbar has buttons for frequently used editing commands that are also available in the File and Edit menus. Figure 3-1 shows the Standard toolbar as a floating palette.

![Standard Toolbar](image)

The following are summary descriptions of each of these buttons and the equivalent menu option. For full descriptions, see the discussion of the menu command equivalent to each button.

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File&gt;New</td>
<td>File&gt;Open</td>
<td>Create a new document. See 3.3.1 Creating a File, p.61.</td>
</tr>
<tr>
<td>File&gt;Open</td>
<td>File&gt;Save</td>
<td>Open an existing document. See 3.3.2 Opening a File, p.63.</td>
</tr>
<tr>
<td>File&gt;Save</td>
<td>Edit&gt;Cut</td>
<td>Save the current document. See 3.3.3 Saving and Closing a File, p.64.</td>
</tr>
<tr>
<td>Edit&gt;Cut</td>
<td>Edit&gt;Copy</td>
<td>Delete the selection and place it in the clipboard. See 3.4.2 Editing Text, p.65.</td>
</tr>
<tr>
<td>Edit&gt;Copy</td>
<td>Edit&gt;paste</td>
<td>Copy the selection to the clipboard. See 3.4.2 Editing Text, p.65.</td>
</tr>
<tr>
<td>Edit&gt;paste</td>
<td>File&gt;Print</td>
<td>Insert the clipboard text at the insertion point. See 3.4.2 Editing Text, p.65.</td>
</tr>
<tr>
<td>File&gt;Print</td>
<td>Help&gt;about Tornado</td>
<td>Print the document. See 3.4.5 Printing, p.68.</td>
</tr>
</tbody>
</table>
3.3 File Management

Commands in the File menu include: file management commands, printing commands, recently-opened files, and the Exit command.

The following sections describe these commands. Each section begins with a table that summarizes the buttons (if any) and keyboard shortcuts for the commands described in that section. For information about printing, see 3.4.5 Printing, p.68.

3.3.1 Creating a File

Table 3-1 Toolbar Buttons (Continued)

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td></td>
<td>Display context-sensitive help.</td>
</tr>
</tbody>
</table>

To create a new file, click File>New. A dialog box (Figure 3-2) appears where you can select the type of file you want to create. The file type you select controls such default parameters as color keywords. To enter plain text, you can use any file type.

When you select a file type, an empty window appears, ready for you to begin entering text. Figure 3-3 shows a new file window for type Source File (the type for C or C++ source, including header files).

The area at the left margin is called the attribute pane. Only the Source File file type includes an attribute pane in its display. The attribute pane indicates breakpoints and the target-program context when you use the debugger. You can turn the attribute pane off (among other editor options) from Tools>Options>Editor (see 9.2.2 Editor Preferences, p.283). If there is no attribute pane, the debugger highlights an entire line of source code to show the location of a breakpoint and to show
where the program has stopped. (You can control the color used for such highlighting; see 9.2.6 Fonts/Colors, p. 289.)
3.3.2 Opening a File

To open an existing file, click File > Open. A standard Windows file browser (Figure 3-4) allows you to select what file to open.

![Figure 3-4 Opening C++ Source Files](image)

The bottom part of the File menu lists the most recently opened files. You can choose one of these without navigating through the Open dialog box.
3.3.3 Saving and Closing a File

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>File Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>CTRL+F4</td>
<td>Close</td>
</tr>
<tr>
<td></td>
<td>CTRL+S</td>
<td>Save</td>
</tr>
<tr>
<td>n/a</td>
<td>ALT+F a</td>
<td>Save As</td>
</tr>
</tbody>
</table>

To write out the current file, click File>Save. To specify a new name (or path) for the current file, click Save As instead. (Save is disabled until you modify the current file.)

Click Close to dismiss the editor window for the current file. If the file has changed since you last saved, a confirmation dialog box offers you the opportunity to save the file before closing it.

3.4 Typing and Editing

Only one of the edit windows in Tornado is active at any one time. The active window contains a text cursor, a blinking vertical line also called an insertion point. Whatever you type appears at the text location indicated by the text cursor.

The editor is designed for editing source files. As such, it does not provide the “word wrap” feature found in many word processors. You must press ENTER to start a new line. If a line is too long for the current width of the edit window, the text scrolls horizontally as necessary to display the portion of the line you are editing.

NOTE: If you cannot type inside an editor window, check for a READ indicator on the status line at the bottom of the Tornado window. If that indicator appears, the editor is displaying a read-only file. To enable typing in that window, save a copy of the file under a different name, or use Windows to turn off the read-only file attribute.
3.4.1 Editing Mode

There are two editing modes in Tornado: overtype mode, which replaces the existing text under the cursor as you type, and insert mode (the default), which displaces text to the right while adding the characters you type. Use the INSERT key on your keyboard to toggle between these two modes. The indicator OVR appears in the status line at the bottom of the Tornado window when overtype mode is on. The editing mode does not change when you switch edit windows; the last mode you selected continues to apply, even if you switch to a window that you last edited with the other editing mode.

3.4.2 Editing Text

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Edit Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>CTRL+Z</td>
<td>Undo</td>
</tr>
<tr>
<td></td>
<td>CTRL+X</td>
<td>Cut</td>
</tr>
<tr>
<td></td>
<td>CTRL+C</td>
<td>Copy</td>
</tr>
<tr>
<td></td>
<td>CTRL+V</td>
<td>Paste</td>
</tr>
<tr>
<td>n/a</td>
<td>DEL</td>
<td>Delete</td>
</tr>
<tr>
<td>n/a</td>
<td>CTRL+A</td>
<td>Select All</td>
</tr>
</tbody>
</table>

The Edit menu supports the Windows standard editing functions: Undo, Cut, Copy, Paste, Delete, and Select All, with standard shortcuts.

The Undo command reverses the effects of the most recent change; click the command again to reverse the change before that. The editor supports up to 512 levels of undo history; you can set how much history to save (among other editor options) by clicking Tools>Options>Editor (9.2.2 Editor Preferences, p.283).
3.4.3 Navigation

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Edit Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>CTRL+G</td>
<td>Go To</td>
</tr>
<tr>
<td>n/a</td>
<td>F4</td>
<td>Next Error/Tag</td>
</tr>
<tr>
<td>n/a</td>
<td>SHIFT+F4</td>
<td>Previous Error/Tag</td>
</tr>
</tbody>
</table>

The Go To command in the Edit menu displays a dialog box that allows you to specify, by line number, what portion of the file to view.

The Next Error/Tag and Previous Error/Tag commands are enabled when errors or warnings are displayed in the Build Output window. In that situation, click these commands to examine the source-code context for each error or warning in turn. See 4.2.4 Building a Downloadable Application, p.83 for more information.

The editor uses the standard Windows keys and mouse actions for moving throughout the file. For example, the following specialized keys have the standard effects:

- **PG UP**: Display the next portion of text.
- **PG DN**: Display the previous portion of text.
- **END**: Move the cursor to the end of the line.
- **CTRL+END**: Display the end of the document.
- **HOME**: Move the cursor to the beginning of the line.
- **CTRL+HOME**: Display the start of the document.
- **← ↑ ↓ →**: Move the cursor in the direction of the arrow, one character or line at a time.
- **CTRL+←**: Move the cursor one word at a time, in the direction of the arrow.
- **CTRL+↑**: Scroll the window by one line, in the direction of the arrow, without changing the cursor position.
3.4.4 Search and Replace

Click Edit>Find to search for a string in your file. Figure 3-5 shows the Find dialog box.

Enter the string you are looking for, set the options, and click OK. The option buttons under Direction determine whether the editor searches back (Up) from the cursor position, or forwards (Down). If the text is not found, the editor displays an error message; the Find dialog box remains open, in case you need to correct the search string.

Click Find Next to search for another instance of the same string in the same search direction; click Find Previous to repeat the search, but in the opposite direction.

Click Replace to specify both a string to find and a replacement for it. Figure 3-6 shows the Replace dialog box. The buttons in the Replace dialog box allow you to replace all occurrences of a string, or examine each individual occurrence before deciding whether or not to replace it.
3.4.5 Printing

To print the current edit window, click File>Print. A standard Print dialog box (Figure 3-7) appears.

Figure 3-7 Print Dialog Box
Click Page Setup to change the text that appears at the top (Header) and bottom (Footer) of each page in printouts from the Tornado editor. The dialog box in Figure 3-8 allows you to specify the header and footer text.

![Page Setup Dialog Box](image)

In the Header and Footer boxes of the Page Setup dialog box, you can use the character sequences in Table 3-2 to request timestamps or file names and to control alignment.

<table>
<thead>
<tr>
<th>To Print</th>
<th>Specify</th>
<th>For Alignment</th>
<th>Specify</th>
</tr>
</thead>
<tbody>
<tr>
<td>File name</td>
<td>&amp;f</td>
<td>Left aligned</td>
<td>&amp;l</td>
</tr>
<tr>
<td>Page # of current page</td>
<td>&amp;p</td>
<td>Centered</td>
<td>&amp;c</td>
</tr>
<tr>
<td>Current system time</td>
<td>&amp;t</td>
<td>Right aligned</td>
<td>&amp;r</td>
</tr>
<tr>
<td>Current system date</td>
<td>&amp;d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3.5 Editor Customization

You can customize the editor by specifying fonts, colors, and options such as tab stops, scrollbar displays, and the number of levels of undo history. See 9.2 Setting Options, p.281 for details.

You can also select an alternate editor for use with Tornado. See 9.2.3 External Editor, p.284.
4
Projects

4.1 Introduction

The project facility is a key element of the Tornado IDE. 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.97). 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.
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 and Project and
Build menus.

For information about using the Tornado editor, see 3. Editor. For information
about using an alternate editor, integrating configuration management tools (such
as ClearCase) with the project facility, and other customization options, see

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
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.

If the Create Project window is open (the default when you first start Tornado), click the New tab. Otherwise, click File> New Project. Then choose the selection for a downloadable application, and click OK (Figure 4-3).
The application wizard appears. 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.

You may also enter a description of the project, which will later appear in the property sheet for the project (Figure 4-4). Finally, identify the workspace in which the project should be created. Click Next to continue.

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). Click Next.

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

3. You can modify the default behavior by un-checking the Show this window on startup box at the bottom of the window.
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 SIMHOST_OSgnu (for example, SIMNTgnu).
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:

`packageName.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 the projects, and the directories and files that make up a project (Figure 4-8).

The first level of folders in the Files view are projects. Each project folder contains:

- Project source code files, which are added to the project by the user.
- An Object Modules folder, which contains a list of objects that the project’s build will produce. The list is automatically generated by the project facility.
- An External Dependencies folder, which contains a list of make dependencies. The list is automatically generated by the project facility.

Initially, there are only the default folders for Built Objects and External Dependencies, and the `packageName.out` file. The file `packageName.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 all 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`, or press `CTRL+N`. 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. (See 3. Editor).

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.113). 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.110).

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.84, for information about how and when to calculate makefile dependencies. See Compiler Options, p.111 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 key and mouse shortcuts (such as double clicking on a file name to open the file in the editor, using CTRL+S to save a file, and so on).

See 9. Customization for information about customizing the editor, including how it handles DOS and UNIX end-of-line markers.

4.2.4 Building a Downloadable Application

The project facility uses the GNU make utility to automate compiling and linking an application. 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.

5. See the GNU Make User’s Guide for more information about make.
Binaries produced by a given build are created in a project subdirectory with the same name as the name of the build specification (projectName\buildName).

**NOTE:** All source files in a project are built using a single build specification (which includes a specific set of makefile, compiler, and linker options) at a time. If some of your source requires a different build specification from the rest, you can create a project for it in the same workspace, and customize the build specification for those files. One project's build specification can then be modified to link in the output from the other project. See *Linker Options*, p.113.

**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.107.

### Calculating Makefile Dependencies

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

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

If you do not calculate dependencies before you start a build, Tornado will prompt you to do so for any of the project files for which dependencies have not previously been 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).
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:

**Figure 4-14  Build Property Sheet**

- **objects**
  - Objects for all source files in the project.
- **archive**
  - An archive (library) file.
- **projectName.out**
  - A single, partially-linked and munched object that comprises all of the individual object modules in a project.

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.107.

**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 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. Double-clicking on the line containing the error message opens the file in the editor with a context pointer on the offending
line. The error or warning text is also displayed in the status bar at the bottom of the main Tornado window.

Use the Edit>Next Error/Tag and Edit>Previous Error/Tag menu options to navigate between errors when you are using the Tornado editor.

⚠️ **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.107 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>Toolbar>Build Toolbar menu option. The toolbar is shown free-floating in Figure 4-16, but is docked by default.

Figure 4-16  **Build Toolbar**

![Build Toolbar](image)

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

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Build button" /></td>
<td>Build&gt;Build</td>
<td>Build project.</td>
</tr>
<tr>
<td><img src="image" alt="Rebuild button" /></td>
<td>Build&gt;Rebuild All</td>
<td>Rebuild project (performing a make clean first).</td>
</tr>
<tr>
<td><img src="image" alt="Compile button" /></td>
<td>Build&gt;Compile</td>
<td>Compile selected source file.</td>
</tr>
</tbody>
</table>
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 5. Target Server 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 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 File>New Project and creating a new project when the workspace is open.

Existing projects can be added to a workspace by selecting the menu options File>Add Project to Workspace, and using the file browser to select a project file (projectName.wpj).

<table>
<thead>
<tr>
<th>Table 4-1 Build Toolbar Buttons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
</tr>
<tr>
<td>Build&gt;Dependencies</td>
</tr>
<tr>
<td>Build&gt;Stop Build</td>
</tr>
<tr>
<td>Project&gt;Download</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build&gt;Dependencies</td>
<td>Update dependencies.</td>
<td></td>
</tr>
<tr>
<td>Build&gt;Stop Build</td>
<td>Stop build.</td>
<td></td>
</tr>
<tr>
<td>Project&gt;Download</td>
<td>Download object file (or boot image for target simulator).</td>
<td></td>
</tr>
</tbody>
</table>
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.106).

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.
If the Create Project window is open (the default when you first start Tornado), click the New tab. Otherwise, click File>New Project. Then choose the selection for a bootable application, and click OK (Figure 4-17).

![Create Bootable Application](image)

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.

**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.

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.

6. You can modify the default behavior by un-checking the Show this window on startup box at the bottom of the window.
Basing a project on an existing one 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.

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:

```
c:\tornado\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.
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).

![VxWorks Project Files](image)

**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).

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.106).

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. It contains symbols for components that do not have initialization routines.
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 Built Objects 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

![VxWorks Components](image)

**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).

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
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).

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 4-27 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.103).

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**

![Including Conflicting Components diagram](image)

**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.)

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

Figure 4-29  Component Conflicts

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.

For more information about component parameters, see the *VxWorks Programmer's Guide* and the *VxWorks Network Programmer's Guide*.

**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).

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 be flagged as such, so that they are only used when a ROM-based image is built. See Compiler Options, p. 111.

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 drop-down list at the top of the workspace window.

See 4.2.4 Building a Downloadable Application, p. 83 for more information about a generic build, and 4.5 Working With Build Specifications, p. 107 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. 113.
4.3.6 Booting VxWorks

For information about booting VxWorks (and bootable applications) see 2.5 Booting VxWorks, p.38. 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.75. For information about configuring and building VxWorks, see 4.3 Creating a Custom VxWorks Image, p.90. For information about additional build options, see 4.5 Working With Build Specifications, p.107.

⚠️ 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.107 for information about modifying builds and creating new build configurations.
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 all 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.

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
- compiler options
- assembler options
linker options

For information about build targets for downloadable applications, see Build Specifications, p. 85. For information about build targets for bootable applications, see 4.3.4 Selecting the VxWorks Image Type, p. 103. 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 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 Rule](Image)

**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**

![Makefile Macros](Image)

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).

![Compiler Options](image)

**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.

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. Unchecking 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.103.
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).

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.
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).

To link an object or library (archive) 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.110).

⚠️ **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 (see Build Specifications, p.85), 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).

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.108).

**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).

![Build Specification Selection](image)

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, and so on). By default, VxWorks is configured for a network connection. Also note that before you use Tornado tools such as the shell and debugger, you must start a target server that is configured for the same mode of communication. See 2.4 Host-Target Communication Configuration, p. 24; and 5.2 Configuring and Starting a Target Server, p. 128.

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

![Target Agent Connection Options](image)

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.

Also see *Scaling the Target Agent*, p.120 and *Starting the Agent Before the Kernel*, p.121.

**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 2.4.4 *The NetROM ROM-Emulator Connection*, p.30).

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.

**WDB_NETROM_MTU**

The default is 1500 octets.

**WDB_NETROM_INDEX**

The value 0 indicates that pod zero is at byte number 0 within a ROM word.

**WDB_NETROM_NUM_ACCESS**

The value 1 indicates that pod zero is accessed only once when a word of memory is read.
WDB_NETROM_POLL_DELAY
The value 2 specifies that the NetROM is polled every two VxWorks clock ticks to see if data has arrived from the host.

WDB_NETROM_ROMSIZE
The default value is ROM_SIZE, a makefile macro that can be set for a specific build. See Makefile Macros, p.110.

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 c:\tornado\target\src\config\usrWdb.c specifies the offset of dual-port RAM from the start of the ROM address space.

\[ \text{dpOffset} = (\text{WDB_ROM_SIZE} - \text{DUALPORT_SIZE}) \times \text{WDB_NETROM_WIDTH}; \]

If your board has more than one ROM socket, this calculation gives the wrong result, because the VxWorks macro ROM_SIZE describes the total size of the ROM space—not the size of a single ROM socket. In that situation, you must adjust this calculation.

Refer to the NetROM documentation for more information on the features governed by these parameters.
Configuration for Network Connection

To configure the target agent for use with a network connection, add the WDB network connection component. (See 2.4.1 Network Connections, p.27).

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 2.4.3 Serial-Line Connections, p.28).
By default, the agent uses serial channel 1 at 9600 bps. 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

![Serial Connection Macros](image)

**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:

- WDB banner (INCLUDE_WDB_BANNER)
- VIO driver (INCLUDE_WDB_VIO)
- WDB task creation (INCLUDE_WDB_START_NOTIFY)
- WDB user event (INCLUDE_WDB_USER_EVENT)

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).
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 about 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:

   ```c
   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 6.2.6 Using the Shell for System Mode Debugging, p.169 for information on using system mode from the shell, and 8.6 System-Mode Debugging, p.263 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 `sysHwInit()` routine. Once `sysHwInit()` is working properly, you no longer need to start the agent before the kernel.

8. The code generator for `prjConfig.c` is based on a component descriptor language that specifies when components are initialized. The component descriptors are searched in a specific order, with the project directory last in the search path. This allows the `.cdf` files in the project directory to override default definitions in the generic `.cdf` files.
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 72).
Configuring Boot Parameters

To customize a boot program for your development environment, you must edit `c:\tornado\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 2.5.4 Description of Boot Parameters, p.41 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 Tornado GUI. Then 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 4-1). Then click OK.

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.116.

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

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:

```c
#define INCLUDE_TSFS_BOOT
#else
#uncomment the above line here
#endif
```

If one serial line connects the host and target, config.h must include the lines:

```c
#define INCLUDE_TSFS_BOOT
#undef CONSOLE_TTY
#define WDB_COMM_TYPE WDB_COMM_SERIAL
```

For a NetROM connection, config.h must include the lines:

```c
#define INCLUDE_TSFS_BOOT
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_NETROM
```

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 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.124).

See the VxWorks Programmer’s Guide: Local File Systems for more information about the TSFS.
5.1 Introduction

A Tornado target server runs on the host and manages communications between Tornado host tools (such as the shell, debugger, and browser), and the target system itself (Figure 5-1). A target server includes the host-resident target symbol table, and an object-module loader that inserts application modules into a running target system.

The target server communicates with the target system through the target agent, which runs on the target system either as a VxWorks task or externally from VxWorks. In order to communicate with the target agent, the target server uses a communication back end configured for the same communication protocol and transport layer as the target agent.
A target server must be configured for a target and started before host tools can interact with the target. It *must* be configured with a communications back end that supports the communication protocol and transport layer used by the target agent, or it will not work. When a target server is started, it identifies the target agent by the network name of a target board on which the target agent is running.

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 are registered with, and made accessible to users by, the service formally known as the *Tornado target registry* (also frequently referred to as the *Tornado registry*, or simply *the registry*). For information about the registry, see 2.2 Host Setup: Tornado Registry, p.15; and the *wtxregd* reference documentation in D. Tornado Tools Reference.

**NOTE:** This chapter describes various ways to configure, start, and manage target servers using the Tornado GUI. For detailed information about the operation of the target server, and its command options, see the *tgtsvr* reference material in D. Tornado Tools Reference.

Before configuring and starting a new target, make sure that the host and target are connected properly (see 2.3 Target Setup, p.17).

### 5.2 Configuring and Starting a Target Server

A target server must be configured and started before any tool (such as the debugger) can communicate with a target system. There are two ways to configure and start a server:

- With the Configure Target Servers dialog box (see 5.2.1 Using the Configure Target Servers Dialog Box, p.129).

- From the command line or from a batch file (see 5.2.3 Using the Command Line, p.140).

The Configure Target Servers dialog box provides the simplest means of configuration. The command line provides the complete set of options available for a server. Command line usage also allows you to start target servers before Tornado.
5.2.1 Using the Configure Target Servers Dialog Box

The Configure Target Servers dialog box is a GUI tool for configuring and starting a target server. To display the box, select Tools > Target Server > Configure.

When you press the New button, Tornado displays the dialog box with default options. (Figure 5-2).

If you are using the default wdbrpc back end (for IP and serial connections) you only need to enter the following information:

- A name for your configuration (the Description field).
- The target name or IP address.

NOTE: When you start a VxWorks target simulator from the Tornado GUI, you are automatically prompted to start or configure a target server for the simulator.
The communication port, if your target and host are communicating over a serial link.

Tornado supplies a default for the Description field, but descriptive configuration names are more useful, particularly if you plan to use more than one target-server configuration. You can configure a target server so that the configuration name appears as a menu option Target Server in the Tools menu with the Add description to menu checkbox.

The fields and lists displayed in the property panel section of the box change based on the selection you make from the Change Property drop-down list. These options are discussed later in this section.

Each time you specify a configuration option, the Command Line box displays the corresponding `tgtsvr` command and its parameters. The text in the box cannot be edited, but you can use the Other Options field to add options that are not exposed through the GUI (see Miscellaneous, p.139).

You can also use the Command Line box display to copy the text to a batch file in order to launch a target server automatically with this configuration. You may also find it helpful to copy the text to the command line and explore various options in conjunction with the `tgtsvr` reference documentation (see D. Tornado Tools Reference).

Control Buttons

To start a target server and save your server configuration, click the Launch button, which also saves the configuration. To save the configuration without starting a server, click OK instead. The name used for the configuration is in the Description field.

To discard all changes since you opened the Configure Target Servers dialog box, click Cancel.

The following controls are used to manage saved configurations and to create new ones:

Target Server Descriptions list
To select a configuration for modification or for starting a target server, select a name from this list. The remaining fields of the Configure Target Servers dialog box are filled in for that configuration name.

New button
To create a new target-server configuration from scratch, click this button. A default target-server configuration name appears, and the remaining fields of the Configure Target Servers dialog box are cleared.
Copy button

To create a new target-server configuration based on the one currently selected, click this button. A new default target-server configuration name appears, and the remaining fields of the Configure Target Servers dialog box are copied to the new configuration.

Remove button

To discard a configuration you no longer need, first select that configuration in the Target Server Descriptions list, then click this button.

5.2.2 Target-Server Configuration Options

This section describes all the configuration options you can specify in the Configure Target Servers dialog box. All options are listed in the Target Server Properties dropdown list.

⚠️ WARNING: The target server must be configured with the same communication back end as the one used by the VxWorks target agent. See Back End, p.132 and 4.6 Configuring the Target-Host Communication Interface, p.116.

Authorizations

By default, anyone on a networked Tornado site who connects to the Tornado registry you are using can connect to any target server registered there. Use the properties in the Authorizations list to limit who can use your target server, or to prevent others from using it altogether.

⚠️ CAUTION: The target-server authorization mechanism assumes a collaborating group of users; it is not secure against malicious use. If you are concerned about interference from malicious users, isolate your PC from any network that such users may be able to reach.

To enable the authorization mechanism, you (and anyone else who wishes to participate in the Tornado authorization scheme from a Windows host) must define an environment variable WIND_UID as identification; see Sharing and Reserving a Target Server, p.145.

The Authorizations property list (Figure 5-3) offers the following controls:
Lock on Startup

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

User ID File

To restrict this target server to a particular set of users, specify the full path and file 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 Authorized User File, p. 146.

Back End

The Back End property list is used to specify how a target server will communicate with a target (Figure 5-4).

The fields displayed for the Back End property list vary depending on the communications method you choose from the list labeled Available Back Ends. The default, wdbrpc, is suitable for targets with IP connectivity.
The standard back ends are described in Table 5-1. Other back ends may be available separately; contact your nearest Wind River Systems office (see the back cover).

Table 5-1  Communications Back Ends for Target Server

<table>
<thead>
<tr>
<th>Back End Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wdbrpc</td>
<td>WDB RPC back end. This back end is the default. It is the most frequently used back end, and supports any kind of IP connection (for example, Ethernet). Polled-mode Ethernet drivers are available for most BSPs to support system mode debugging for this type of connection.</td>
</tr>
<tr>
<td>wdbpipe</td>
<td>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.</td>
</tr>
<tr>
<td>wdbserial</td>
<td>WDB Serial back end. A back end 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.</td>
</tr>
<tr>
<td>netrom</td>
<td>WDB NetRom. A back end that communicates over a proprietary communications protocol for NetROM, a networked ROM emulator. This back end supports either system- or task-level views of the target, depending on target-agent configuration.</td>
</tr>
</tbody>
</table>

WARNING: The target server must be configured with the same communication back end as the one built into the VxWorks image. See 2.4 Host-Target Communication Configuration, p.24 and 4.6 Configuring the Target-Host Communication Interface, p.116.

The following Back End properties appear for all standard back ends:

Timeout
   How many seconds to wait for a response from the agent running on the target system (the default is 3 seconds).

Re-try
   How many times to repeat a transaction if the target agent does not appear to respond the first time.

The wdbserial back end (see Figure 5-4) also has the following properties:
Serial Port
The communications port on your PC that is connected to the target. The default serial device is COM2.

Speed (bps)
The line speed (in bits per second) that your target uses over its serial line. If your target is configured to use the default serial speed of 9600 bps, leave the Speed setting at the default. However, it is best to use the fastest available serial line speed available. When you change the line speed, you must also re-compile the target agent (see 4.6 Configuring the Target-Host Communication Interface, p. 116).

All back ends also require:

Target IP Name/Address
The IP address or network name of the target hardware for networked targets; or an arbitrary target-server name for other targets. This field appears below the Back End property list (see Figure 5-2).

Core File and Symbols
The Core File and Symbols property list gathers properties that have to do with the host-resident copy of the target-system executable.

Figure 5-5 Core File and Symbols Properties

File Path From Target
To search for an image of the software running on the target according to the path recorded on the target itself, leave File Path From Target selected. This is appropriate, for example, in configurations where the run-time image is downloaded from the host at boot time.
Target Server

File
If the run-time image file is no longer in the same location on the host that is configured into the target (or if host and target have different views of the file system), select the option button next to File, and use the adjacent text box to specify where to find the image on the host. To locate a core image using a file browser, click the button to the right of the text box.

Global Symbols, All Symbols, or No Symbols
By default, the target server records only global symbols when it manages the host-resident symbol table for the target system. To control explicitly what symbols to record, select one of the following three buttons in the Core File and Symbols property list:

- No Symbols: Do not load the symbol table at all.
- All Symbols: Include both local and global symbols.
- Global Symbols: Restore the default setting.

Synchronize Target/Host Symbol Tables
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.96 and the reference entry for `symSyncLib`.

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 enter an object format to override the default.

Memory Cache Size
In order to avoid excessive data-transfer transactions with the target, the target server maintains a cache on the host system. By default, this cache can grow up to a size of 1 MB. Change the cache size using the options in the Memory Cache Size property list (Figure 5-6).

- Default: Use the default cache size.
- Specify: To choose another target-memory cache size, click this option and specify the size of the cache in kilobytes (either in decimal or hexadecimal). A larger maximum cache size may be desirable if the memory pool used by host tools on the target is large, because transactions on memory outside the cache are far
slower. See Scaling the Target Agent, p. 120 for more information about the memory pool managed by the server on the target.

**Target Server File System**

The Target Server File System (TSFS) is a full-featured VxWorks file system that provides target access to files located on the host system. See the `tgtsvr` reference entry in D. Tornado Tools Reference and VxWorks Programmer’s Guide: Local File Systems for more information.

The TSFS provides the most convenient way to boot a target over a serial connection (see 2.5.7 Booting a Target Without a Network, p. 46).

⚠️ **WARNING:** To use the TSFS, you must include the WDB target server file system component when you build the VxWorks image. See 4.3 Creating a Custom VxWorks Image, p. 90.

---

Enable File System

Check this box to use the TSFS.
Root
Identify the root of the host file system that will be made visible to target processes using the TSFS, for example `c:\tornado`.

If you use the TSFS for booting a target, it is recommended that you use the base Tornado installation directory. If you do not do so, you must use the Core File and Symbols configuration options to specify the location of the VxWorks image (see Core File and Symbols, p.134).

Read only
Make the TSFS read-only (recommended for most purposes).

Read / Write
Allow read and write access to host files by target processes using the TSFS. When this option is specified, access to the target server is restricted to processes running under the same user ID as the server.

Console and Redirection

Tornado supports virtual I/O to the host from target applications (see Virtual I/O, p.11). By default, any virtual output from the target is broadcast to all shell windows.

However, you can use the target server to create a dedicated console for virtual I/O. The option buttons in the Console and Redirection property list (Figure 5-8) direct virtual I/O to various locations:

Figure 5-8 Virtual Console Properties

Redirect Target IO
Redirect the target global `stdin`, `stdout`, and `stderr` to the target server. If the console is not created, WTX events are sent to all WTX tools when characters come from the target.
Create Console Window

Create a virtual console window on the target server host for target I/O. The Log Console window (see 5.3 Target Server Icon and Log Console Window, p.141) is not used for this purpose.

Redirect Target Shell

Start a console window into which the target shell’s standard input, output, and error will be redirected.

NOTE: See the tgtsvr reference entry in D. Tornado Tools Reference for detailed information about these options and for other I/O options available with command-line usage.

Logging

The Logging property list (Figure 5-9) allows you to log WDB and WTX requests.

Figure 5-9 Logging Configuration Properties

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.

Max Size

The maximum size of the back-end 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).

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.

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.

Miscellaneous
The Miscellaneous property list (Figure 5-10) provides for Tornado 1.0.1 tools compatibility, and for use of options that are not available through the Configure Target Servers dialog box.

Use portmapper
To register a target server with the RPC portmapper, turn on this checkbox. 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.

Other Options
Use this field to enter any options that are not otherwise available through the Configure Target Servers dialog box.

5.2.3 Using the Command Line

In general, you will probably do most of your work from the Tornado IDE. In this case, you do not need to worry about environment variables: Tornado manages its internal environment automatically. However, if you also want to run some portions of Tornado from the command line (such as the compilation tools, or the shell), you must make sure that the appropriate variables are set in order for the tools to work properly. The file torVars.bat in c:\tornado\host\x86-win32\bin captures the requisite definitions, so that all you have to do is run torVars.bat to set the environment variables appropriately for your own Tornado installation.

You may want to start a server from the command line or from a batch file, so that your target is ready to use as soon as you enter the Tornado development environment. To do so, run c:\tornado\host\x86-win32\bin\tgtsvr.exe. You must specify the network name of your target as an argument (see Establishing the VxWorks Target Name and IP Address, p.21; and the tgtsvr reference in D. Tornado Tools Reference).

There are a number of useful command-line options to tgtsvr that control the behavior of your target server. These include the -V (verbose) option for troubleshooting, the -B option for alternative methods of communicating with the target, and several options for redirecting I/O. For detailed information on these and other command-line options, see the tgtsvr reference entry in D. Tornado Tools Reference.
5.3 Target Server Icon and Log Console Window

When a target server is started, the target server icon is displayed in the Windows taskbar (Figure 5-11). The context menu for the icon provides options for displaying the Log Console window, displaying information about the target, and shutting down the target server. The tooltip for the icon displays the name of the target server. Double-clicking on the icon opens the Log Console window. When the target server reports error messages, a yellow exclamation point is superimposed on the target server icon.

![Target Server Log Console and Icon](image)

The Log Console window displays target server log output (Figure 5-11). The Hide button hides the target server Log Console window. The Stop button stops the target server. The About button displays version information about the target server, and basic information about the target system (BSP, CPU, and operating system).

5.4 Stopping a Target Server

There are various ways of stopping a target server, including:

- Using the Kill option in the Manage Target Servers dialog box (see 5.6 Managing a Target Server, p.143).
- Using the Shutdown button in the target server Log Console window (see 5.3 Target Server Icon and Log Console Window, p.141). Note that using the title bar close button (x) merely hides the window; it does not stop the target server.
- Selecting the Shutdown option from the context menu for the target server icon in the Windows taskbar (the tooltip for the icon displays the name of the target server; see 5.3 Target Server Icon and Log Console Window, p.141).
5.5 Selecting a Target Server

If a target server for your target has already been configured and started, you can select it with either the Tornado Launch toolbar or by various means from the Tools menu.

For information about configuring and starting a target server, see 5.2 Configuring and Starting a Target Server, p.128.

Tornado Launch Toolbar

The Tornado Launch toolbar has a pull-down list box that shows all the target servers that are currently running and known to the Tornado registry that your system is using (Figure 5-12; also see 2.2 Host Setup: Tornado Registry, p.15). Click on the drop-down list box to display the available target servers; then select a target server from the list. If no target servers are listed, or none of the ones listed represent the target you need, you must configure and start a target server.

![Figure 5-12 Tornado Launch Toolbar](image)

Tools Menu

When you start a browser, shell, or debugger from the Tools menu you can specify which target server to use. Tornado displays a dialog box with summary
information about the target server currently selected with the Targets drop-down list for that tool.

Figure 5-13 shows the Launch Shell dialog box that appears when you click Tools > Shell; clicking Browser or Debugger produces a similar dialog box. Thus, you can use the Tools menu to connect a tool to an alternate target without changing the currently selected target in your Tornado session.

You can also use the Manage Target Servers dialog box to select a target server. See 5.6 Managing a Target Server, p.143.

### 5.6 Managing a Target Server

A number of target-server management commands are available to control your target servers and other networked Tornado target servers. To reach these commands, click Tools > Target Server > Manage. Tornado opens the dialog box shown in Figure 5-14.

At the top of the Manage Target Servers dialog box is a target selector—a drop-down list that you can use to select any of the targets registered with the Tornado target registry you are using. To help you see what target is selected, the Manage Target
Servers dialog box displays the target information summary (the same information as the target summary available from the browser; see 7.5 Target-Information Window, p.211). You can update the display with the (exclamation mark) button.

Once you select a target, you can select a command from the Select Action drop-down list box (Figure 5-14), then click Apply to execute the command on the selected target. Click Close to dismiss the dialog box.

The following commands are available on the Select Action drop-down list:

**Reserve**
- Reserve the target server for your own use. See Sharing and Reserving a Target Server, p.145 for more information.

**Unreserve**
- Release a previously reserved server so that others can use it.

**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.
**Kill**

Stop the currently selected target server; equivalent to pressing `CTRL+BREAK` in the target-server window.

---

**WARNING:** Close any Tornado tools that use a particular target before you kill it. 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**

Reboot the selected target and re-initialize its target server.

---

**Sharing and Reserving a Target Server**

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

---

**CAUTION:** The target-server authorization mechanism assumes a collaborating group of users; it is not secure against malicious use. If you are concerned about interference from malicious users, isolate your PC from any network that such users may be able to reach.

---

To participate in the Tornado authorization scheme, you must set the environment variable `WIND_UID` to a unique numeric user ID; see your Windows documentation.

If some Tornado users at your site use UNIX hosts, they do not need to set the `WIND_UID` environment variable; on UNIX hosts, the Tornado authorization scheme uses the system user ID.

When a target server is available to anyone, its status is `unreserved`. This status is visible in the browser (7.5 Target-Information Window, p.211) and in the target-
selection dialog for any Tornado tool (for example, Figure 5-13). 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 start it, by clicking the Lock on Startup option button in the Authorizations property panel of the Configure Target Servers dialog box. See 5.2.2 Target-Server Configuration Options, p.131. Target servers started 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 the Reserve command of the Manage Target Servers dialog box (see 5.6 Managing a Target Server, p.143). The target status becomes reserved until you release the target with the Unreserve command in the same dialog box. 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 QA 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.

**Authorized User File**

To restrict a target server to a list of users, create a file listing authorized users, and configure the target server to base authorization on that file (see Authorizations, p.131). The file should consist of one line for each user, with each line containing the user name, followed by a space, followed by the user’s numeric identification.

For users on UNIX hosts, the user names are host sign-on names, as used by system files like /etc/passwd (or its network-wide equivalent), and the numbers are the system user IDs. For users on Windows hosts, the names are mnemonic aids, and the numbers are arbitrary identifiers.

You can also use one special entry in the authorization file: a plus sign (+) to explicitly authorize any user to connect to the target server. (This might be useful to preserve the link between a target server and an authorization file when access to that target need only be restricted from time to time.)
6

Shell

6.1 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 6.7 Tcl: Shell Interpretation, p. 197.

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. 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 6-1.

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 6-1  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.162, 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
6.2 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*.

6.2.1 Starting and Stopping the Tornado Shell

There are three ways to start a Tornado shell:

- From the Tornado Launch toolbar: Click the button. This launches a shell for the currently selected target server (see Tornado Launch Toolbar, p.142).
- From the Tools menu: Click on Shell. The dialog box shown in Figure 6-2 appears, which allows you to select a target server from the Targets drop-down list.
- From the Windows command prompt: Invoke `windsh`, specifying the target-server name, as in the following example:

  ```
  C:\> windsh phobos
  ```

If you start a Tornado shell from the IDE, a shell window like the one shown in Figure 6-3 appears, with the arrow prompt (->). If you start a shell from the

---

2. As a special case of executing WindSh from the Windows command prompt, you can configure the properties of the command-prompt icon—or a shortcut—to run `windsh targetname`. 

Windows command prompt, WindSh executes in the environment where you call it, using the command-prompt 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 `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+BREAK).

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.154).

### 6.2.2 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.

#### 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:

```bash
-> printf("Hello World\n")
Hello World!
value = 13 = 0xd
```

This behavior can be dynamically modified using the Tcl procedure shConfig as follows:

```bash
-> ?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.
SH_GET_TASK_IO = off

For more information on shConfig, see WindSh Environment Variables, p.154.

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 IO redirected to WindSh, you can start the target server with the options -C -redirectShell.

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:

```
-> C:\Tor  [CTRL+D]
Tornado/ TorClass/
-> C:\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 */
}
```

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 the VxWorks Programmer’s Guide: WRS Coding Conventions.)

2. Include these files in your project. (See Creating, Adding, and Removing Application Files, p. 81.)

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 in a new browser window.

**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.

\[-(14 \times 9) / 3\]
value = 42 = 0x2a = '*'

\[-(0x1355 << 3) \& 0x0f0f\]
value = 2568 = 0xa08

\[-4.3 \times 5\]
value = 21.5

Calculations With Variables

\[-(j + k) \times 3\]
value = ...

\[-*(j + 8 \times k)\]
(...)address \(j + 8 \times k)\): value = ...

\[-x = (val1 - val2) / val3\]
new symbol "x" added to symbol table
address = ...
value = ...

\[-f = 1.41 \times 2\]
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 the environment variables listed in Table 6-1. The Tcl procedure \texttt{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.

Because \texttt{shConfig} is a Tcl procedure, use the \texttt{?} to move from the C interpreter to the Tcl interpreter. (See 6.7.2 \textit{Tcl: Calling under C Control}, p.199.)
Table 6-1  WindSh Environment Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SH_GET_TASK_IO</td>
<td>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.”</td>
</tr>
<tr>
<td>LD_CALL_XTORS</td>
<td>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.</td>
</tr>
<tr>
<td>LD_SEND_MODULES</td>
<td>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.</td>
</tr>
<tr>
<td>LD_PATH</td>
<td>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.</td>
</tr>
<tr>
<td>LD_COMMON_MATCH_ALL</td>
<td>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.</td>
</tr>
<tr>
<td>DSM_HEX_MOD</td>
<td>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.</td>
</tr>
</tbody>
</table>
Example 6-1  Using shConfig to Modify WindSh Behavior

```bash
-> ?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
```

6.2.3 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 6-2 summarizes the WindSh commands that manage VxWorks tasks.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp()</td>
<td>Spawn a task with default parameters.</td>
</tr>
<tr>
<td>sps()</td>
<td>Spawn a task, but leave it suspended.</td>
</tr>
</tbody>
</table>
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 6.6 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 6-3 summarizes the WindSh commands that report task information.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>i()</td>
<td>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.</td>
</tr>
<tr>
<td>iStrict()</td>
<td>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.</td>
</tr>
</tbody>
</table>
The `i()` command is commonly used to get a quick report on target activity. (To see this information periodically, use the Tornado browser; see 7. 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
LogTask _logTask 3aa918 0 PEND 4df10 3aa748 0 0
WdbTask 0x41288 3870f0 3 READY 23ff4 386d78 3d0004 0
FtpdTask _ftpdTask 3a2c18 55 PEND 23b28 3a2938 0 0
```

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>ENTRY</th>
<th>TID</th>
<th>STATUS</th>
<th>DELAY</th>
<th>OBJ_TYPE</th>
<th>OBJ_ID</th>
<th>OBJ_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tExcTask</td>
<td>_excTask</td>
<td>3d9e3c</td>
<td>PEND</td>
<td>0</td>
<td>MSG_Q(R)</td>
<td>3d9ff4</td>
<td>N/A</td>
</tr>
<tr>
<td>tLogTask</td>
<td>_logTask</td>
<td>3d7510</td>
<td>PEND</td>
<td>0</td>
<td>MSG_Q(R)</td>
<td>3d76c8</td>
<td>N/A</td>
</tr>
<tr>
<td>tWdbTask</td>
<td>_wdbCmdLoo</td>
<td>36dde4</td>
<td>READY</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>tNetTask</td>
<td>_netTask</td>
<td>3a43d0</td>
<td>READY</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>u0</td>
<td>_smtask1</td>
<td>36cc2c</td>
<td>PEND</td>
<td>0</td>
<td>MSG_Q_S(S)</td>
<td>370b61</td>
<td>N/A</td>
</tr>
<tr>
<td>u1</td>
<td>_smtask3</td>
<td>367c54</td>
<td>PEND</td>
<td>0</td>
<td>MSG_Q_S(S)</td>
<td>370b61</td>
<td>N/A</td>
</tr>
<tr>
<td>u3</td>
<td>_taskB</td>
<td>362c7c</td>
<td>PEND</td>
<td>0</td>
<td>SEM_B</td>
<td>8d378</td>
<td>__mySem2</td>
</tr>
<tr>
<td>u6</td>
<td>_smtask1</td>
<td>35dca4</td>
<td>PEND</td>
<td>0</td>
<td>MSG_Q_S(S)</td>
<td>370ae1</td>
<td>N/A</td>
</tr>
<tr>
<td>u9</td>
<td>_task3B</td>
<td>358ccc</td>
<td>PEND</td>
<td>0</td>
<td>MSG_Q(S)</td>
<td>8cf1c</td>
<td>__myMsgQ</td>
</tr>
</tbody>
</table>

value = 0 = 0x0
->
-> tw u1

<table>
<thead>
<tr>
<th>NAME</th>
<th>ENTRY</th>
<th>TID</th>
<th>STATUS</th>
<th>DELAY</th>
<th>OBJ_TYPE</th>
<th>OBJ_ID</th>
<th>OBJ_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u1</td>
<td>_smtask3</td>
<td>367c54</td>
<td>PEND</td>
<td>0</td>
<td>MSG_Q_S(S)</td>
<td>370b61</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x36cc2c</td>
<td>0</td>
<td>0x36e464</td>
</tr>
<tr>
<td>0x367c54</td>
<td>0</td>
<td>0x36e47c</td>
</tr>
</tbody>
</table>

value = 0 = 0x0
->

**System Information**

Table 6-4 shows the WindSh commands that display information from the symbol table, from the target system, and from the shell itself.

Table 6-4  **WindSh Commands for System Information**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>devs()</code></td>
<td>List all devices known on the target system.</td>
</tr>
</tbody>
</table>
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:

```bash
-> 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:

```bash
-> 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:

```plaintext
-> 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:

```plaintext
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 MC680x0 target:

```plaintext
-> 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
00033bd8 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 \texttt{d()} appear below.

Display starting at address 1000 decimal:

\texttt{\textbackslash{}-\texttt{d \{1000\}}}

Display starting at 1000 hex:

\texttt{\textbackslash{}d \textbackslash{}texttt{0x1000}}

Display starting at the address contained in the variable \texttt{dog}:

\texttt{\textbackslash{}d \texttt{dog}}

The above is different from a display starting at the address of \texttt{dog}. For example, if \texttt{dog} is a variable at location 0x1234, and that memory location contains the value 10000, \texttt{d()} displays starting at 10000 in the previous example and at 0x1234 in the following:

\texttt{\textbackslash{}d \texttt{&\{dog\}}}

Display starting at an offset from the value of \texttt{dog}:

\texttt{\textbackslash{}d \texttt{dog + 100}}

Display starting at the result of a function call:

\texttt{\textbackslash{}d \texttt{func \{dog\}}}

Display the code of \texttt{func()} as a simple hex memory dump:

\texttt{\textbackslash{}d \texttt{func}}

\section*{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 6-5 summarizes the WindSh commands of this type.

\begin{table}[h]
\centering
\caption{WindSh Commands for System Modification and Debugging}
\begin{tabular}{ll}
\hline
Call & Description \\
\hline
\texttt{ld()} & Load an object module into target memory and link it dynamically into the run-time. \\
\hline
\end{tabular}
\end{table}
<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unld()</td>
<td>Remove a dynamically-linked object module from target memory, and free the storage it occupied.</td>
</tr>
<tr>
<td>m()</td>
<td>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 (.).</td>
</tr>
<tr>
<td>mRegs()</td>
<td>Modify register values for a particular task.</td>
</tr>
<tr>
<td>b()</td>
<td>Set or display breakpoints, in a specified task or in all tasks.</td>
</tr>
<tr>
<td>bh()</td>
<td>Set a hardware breakpoint.</td>
</tr>
<tr>
<td>s()</td>
<td>Step a program to the next instruction.</td>
</tr>
<tr>
<td>so()</td>
<td>Single-step, but step over a subroutine.</td>
</tr>
<tr>
<td>c()</td>
<td>Continue from a breakpoint.</td>
</tr>
<tr>
<td>cret()</td>
<td>Continue until the current subroutine returns.</td>
</tr>
<tr>
<td>bdall()</td>
<td>Delete all breakpoints.</td>
</tr>
<tr>
<td>bd()</td>
<td>Delete a breakpoint.</td>
</tr>
<tr>
<td>reboot()</td>
<td>Return target control to the target boot ROMs, then reset the target server and reattach the shell.</td>
</tr>
<tr>
<td>bootChange()</td>
<td>Modify the saved values of boot parameters (see 2.5.4 Description of Boot Parameters, p.41).</td>
</tr>
<tr>
<td>sysSuspend()</td>
<td>If supported by the target-agent configuration, enter system mode. See 6.2.6 Using the Shell for System Mode Debugging, p.169.</td>
</tr>
<tr>
<td>sysResume()</td>
<td>If supported by the target agent (and if system mode is in effect), return to task mode from system mode.</td>
</tr>
<tr>
<td>agentModeShow()</td>
<td>Show the agent mode (system or task).</td>
</tr>
<tr>
<td>sysStatusShow()</td>
<td>Show the system context status (suspended or running).</td>
</tr>
<tr>
<td>quit() or exit()</td>
<td>Dismiss the shell.</td>
</tr>
</tbody>
</table>
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 6-6 summarizes these commands. For more discussion of these shell commands, see VxWorks Programmer’s Guide: C++ Development.
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.154.

### Object Display

Table 6-7 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 7. Browser.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>show()</code></td>
<td>Print information on a specified object in the shell window.</td>
</tr>
<tr>
<td><code>browse()</code></td>
<td>Display a specified object in the Tornado browser.</td>
</tr>
</tbody>
</table>
| `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.                               |
Table 6-7  WindSh Commands for Object Display  (Continued)

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>taskSwitchHookShow()</code></td>
<td>Show the list of task switch routines.</td>
</tr>
<tr>
<td><code>taskWaitShow()</code></td>
<td>Show information about the object a task is pended on. Note that <code>taskWaitShow()</code> cannot give object IDs for POSIX semaphores or message queues.</td>
</tr>
<tr>
<td><code>semShow()</code></td>
<td>Show information about a semaphore.</td>
</tr>
<tr>
<td><code>semPxShow()</code></td>
<td>Show information about a POSIX semaphore.</td>
</tr>
<tr>
<td><code>wdShow()</code></td>
<td>Show information about a watchdog timer.</td>
</tr>
<tr>
<td><code>msgQShow()</code></td>
<td>Show information about a message queue.</td>
</tr>
<tr>
<td><code>mqPxShow()</code></td>
<td>Show information about a POSIX message queue.</td>
</tr>
<tr>
<td><code>iosDrvShow()</code></td>
<td>Display a list of system drivers.</td>
</tr>
<tr>
<td><code>iosDevShow()</code></td>
<td>Display the list of devices in the system.</td>
</tr>
<tr>
<td><code>iosFdShow()</code></td>
<td>Display a list of file descriptor names in the system.</td>
</tr>
<tr>
<td><code>memPartShow()</code></td>
<td>Show partition blocks and statistics.</td>
</tr>
<tr>
<td><code>memShow()</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>smMemShow()</code></td>
<td>Display the amount of free space and statistics on memory-block allocation for the shared-memory system partition.</td>
</tr>
<tr>
<td><code>smMemPartShow()</code></td>
<td>Display the amount of free space and statistics on memory-block allocation for a specified shared-memory partition.</td>
</tr>
<tr>
<td><code>moduleShow()</code></td>
<td>Show the current status for all the loaded modules.</td>
</tr>
<tr>
<td><code>moduleIdFigure()</code></td>
<td>Report a loaded module’s module ID, given its name.</td>
</tr>
<tr>
<td><code>intVecShow()</code></td>
<td>Display the interrupt vector table. This routine displays information about the given vector or the whole interrupt vector table if <code>vector</code> is equal to -1. Note that <code>intVecShow()</code> is not supported on architectures such as ARM and PowerPC that do not use interrupt vectors.</td>
</tr>
</tbody>
</table>
**Network Status Display**

Table 6-8 summarizes the WindSh commands that display information about the VxWorks network.

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hostShow()</code></td>
<td>Display the host table.</td>
</tr>
<tr>
<td><code>ifShow()</code></td>
<td>Display the attached network interfaces.</td>
</tr>
<tr>
<td><code>inetstatShow()</code></td>
<td>Display all active connections for Internet protocol sockets.</td>
</tr>
<tr>
<td><code>ipstatShow()</code></td>
<td>Display IP statistics.</td>
</tr>
<tr>
<td><code>routestatShow()</code></td>
<td>Display routing statistics.</td>
</tr>
<tr>
<td><code>tcpstatShow()</code></td>
<td>Display all statistics for the TCP protocol.</td>
</tr>
<tr>
<td><code>tftpInfoShow()</code></td>
<td>Get TFTP status information.</td>
</tr>
<tr>
<td><code>udpstatShow()</code></td>
<td>Display statistics for the UDP protocol.</td>
</tr>
</tbody>
</table>

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()`.
6.2.4 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

\[\rightarrow \text{taskSpawn } ("tmyTask", 10, 0, 1000, myTask, fd1, 300)\]
value = ...

\[\rightarrow \text{fd } = \text{open } ("file", 0, 0)\]
new symbol "fd" added to symbol table
fd = (...address of fd...): value = ...

Invocations of Application Subroutines

\[\rightarrow \text{testFunc } (123)\]
value = ...

\[\rightarrow \text{myValue } = \text{myFunc } (1, \&_val, \text{testFunc } (123))\]
myValue = (...address of myValue...): value = ...

\[\rightarrow \text{myDouble } = \text{(double )} \text{myFuncWhichReturnsADouble } (x)\]
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.177.

6.2.5 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 \texttt{reboot}() command or CTRL+SHIFT+X to make this easy.

When you execute \texttt{reboot}() or type CTRL+SHIFT+X, the following reboot sequence occurs:

1. The shell displays a message to confirm rebooting has begun:

\[\rightarrow \text{reboot}\]
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 \texttt{tgtsvr} reference entry in \textit{D. Tornado Tools Reference}, or in \textit{Tornado Online Manuals: Tornado Tools}.

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.

\begin{itemize}
  \item CAUTION: 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 \texttt{reboot()}, try starting a new target server manually.
\end{itemize}

\section*{6.2.6 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 \textit{4.6 Configuring the Target-Host Communication Interface}, p.116. When the agent supports mode switching, the following WindSh commands control system mode:

\begin{verbatim}
sysSuspend()
\end{verbatim}

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 6-2  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().

Example output:

```
-> sysSuspend
value = 0 = 0x0
-> i
```

```
NAME     ENTRY     TID  PRI  STATUS   PC    SP   ERRNO  DELAY
---------- ---------- ----- -------- ------- ------- ------- ------
tExcTask  _excTask 3e8f98 0  PEND  47982  3e8ef4 0  0
LogTask   _logTask 3e6670 0  PEND  47982  3e65c8 0  0
MdbTask   0x3f024  398e04 3  PEND  405ac  398d50 30067 0
NetTask   _netTask 3b39e0 50 PEND  405ac  3b3988 0  0
```
Agent mode     : Extern
System context : 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
-> i
 _usrClock
00022d9a  4856       PEA   (A6)
00022d9c  2c4f      MOVEA .L   A7,A6
Return to task mode and confirm that return by calling \texttt{i()}: 

```c
-> sysResume
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:

```c
-> 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 \texttt{address} when the instruction at \texttt{address} is executed.
6.2.7 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+BREAK from Tornado or CTRL+C from the console. 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+BREAK or CTRL+C is also necessary to regain control of the shell after calling a routine on the target that ends with `exit()` rather than `return`.

Occasionally a subroutine invoked from the shell may incur a fatal error, such as a bus/address error or a privilege violation. When this happens, the failing routine is suspended. If the fatal error involved a hardware exception, the shell automatically notifies you of the exception. For example:

```
-> taskSpawn -4
   Exception number 11: Task: 0x264ed8 (tCallTask)
```

In cases like this, you do not need to type CTRL+BREAK 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.
6.3 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.

6.3.1 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 6-9.

<table>
<thead>
<tr>
<th>Element</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>int</td>
</tr>
<tr>
<td>variable used as floating-point</td>
<td>double</td>
</tr>
<tr>
<td>return value of subroutine</td>
<td>int</td>
</tr>
<tr>
<td>constant with no decimal point</td>
<td>int/long</td>
</tr>
<tr>
<td>constant with decimal point</td>
<td>double</td>
</tr>
</tbody>
</table>

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. 177. Table 6-10 shows the various data types available in the shell C interpreter, with examples of how they can be set and referenced.

### Table 6-10 Data Types in the Shell C Interpreter

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Set Variable</th>
<th>Display Variable</th>
</tr>
</thead>
</table>
| int     | 4     | x = 99       | x
|         |       |              | (int) x          |
| long    | 4     | x = 33       | x
|         |       |              | (long) x         |
| short   | 2     | x = (short)20| (short) x        |
| char    | 1     | x = 'A'      | (char) x         |
|         |       | x = (char)65 |                  |
|         |       | x = (char)0x41|                |
| double  | 8     | x = 11.2     | (double) x       |
|         |       | x = (double)11.2|               |
| float   | 4     | x = (float)5.42|             |

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. 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. You can also use `printf()` to display strings.

The shell places no type restrictions on the application of operators. For example, the shell expression:

`*(70000 + 3 * 16)`

evaluates to the 4-byte integer value at memory location 70048.

---

3. Memory allocated for string constants is never freed by the shell. See 6.3.7 Strings, p. 182 for more information.
4. `d()` is one of the WindSh commands, implemented in Tcl and executing on the host.
6.3.2 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.

6.3.3 Expressions

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

Literals

The shell interprets the literals in Table 6-11 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>
<thead>
<tr>
<th>Literal</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal numbers</td>
<td>143967</td>
</tr>
<tr>
<td>octal numbers</td>
<td>017734</td>
</tr>
<tr>
<td>hex numbers</td>
<td>0xf3ba or $f3ba</td>
</tr>
<tr>
<td>floating point numbers</td>
<td>666.666</td>
</tr>
<tr>
<td>character constants</td>
<td>'x' and '$'</td>
</tr>
<tr>
<td>string constants</td>
<td>&quot;hello world!!&quot;</td>
</tr>
</tbody>
</table>

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 6-12 in the same way as the C compiler.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>+ — * /</td>
</tr>
<tr>
<td>relational</td>
<td>== != &lt; &gt; &lt;= &gt;=</td>
</tr>
<tr>
<td>shift</td>
<td>&lt;&lt; &gt;&gt;</td>
</tr>
<tr>
<td>logical</td>
<td></td>
</tr>
<tr>
<td>bitwise</td>
<td></td>
</tr>
<tr>
<td>address and indirection</td>
<td>&amp; *</td>
</tr>
</tbody>
</table>

The shell assigns the same precedence to the operators as the C compiler. However, unlike the C compiler, the shell always evaluates both sub-expressions of the logical binary operators || and &&.

Function Calls

Shell expressions may contain calls to C functions (or C-compatible functions) whose names have been entered in the system symbol table; they may also contain function calls to WindSh commands that execute on the host.
The shell executes such function calls in tasks spawned for the purpose, with the specified arguments and default task parameters; if the task parameters make a difference, you can call `taskSpawn()` instead of calling functions from the shell directly. The value of a function call is the 4-byte integer value returned by the function. The shell assumes that all functions return integers. If a function returns a value other than an integer, the shell must know the data type being returned before the function is invoked. This requires a slightly unusual syntax because you must cast the function, not its return value. For example:

```
-> floatVar = ( float () ) funcThatReturnsAFloat (x,y)
```

The shell can pass up to ten arguments to a function. In fact, the shell always passes exactly ten arguments to every function called, passing values of zero for any arguments not specified. This is harmless because the C function-call protocol handles passing of variable numbers of arguments. However, it allows you to omit trailing arguments of value zero from function calls in shell expressions.

Function calls can be nested. That is, a function call can be an argument to another function call. In the following example, `myFunc()` takes two arguments: the return value from `yourFunc()` and `myVal`. The shell displays the value of the overall expression, which in this case is the value returned from `myFunc()`:

```
myFunc (yourFunc (yourVal), myVal);
```

Shell expressions can also contain references to function addresses instead of function invocations. As in C, this is indicated by the absence of parentheses after the function name. Thus the following expression evaluates to the result returned by the function `myFunc2()` plus 4:

```
4 + myFunc2 ()
```

However, the following expression evaluates to the address of `myFunc2()` plus 4:

```
4 + myFunc2
```

An important exception to this occurs when the function name is the very first item encountered in a statement. This is discussed in *Arguments to Commands*, p.179. 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:

\[
\text{rename ("oldname", "newname")} \\
\text{rename "oldname", "newname"}
\]

as are:

\[
\text{evtBufferAddress ( )} \\
\text{evtBufferAddress}
\]

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 \text{evtBufferAddress()}:

\[
\text{value = evtBufferAddress}
\]

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 \textit{u} for tasks started from the Tornado shell, and with a \textit{t} for VxWorks tasks started from the target itself. In addition, tasks started from a shell are prefixed by \textit{s1}, \textit{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 \texttt{sp()} in the first shell opened are given names such as \texttt{s1u0} and \texttt{s1u1}. Tasks spawned with the second shell opened have names such as \texttt{s2u0} and \texttt{s2u1}.

You are urged to adopt a similar convention for tasks named in your applications.

\section*{6.3.4 The “Current” Task and Address}

A number of commands—\texttt{c()}, \texttt{s()}, \texttt{ti()}—take a task parameter that can be omitted. If omitted, the current task is used. The \texttt{l()} and \texttt{d()} commands use the current address if no address is specified. The current task and address are set when:

\begin{itemize}
  \item A task hits a breakpoint or an exception trap. The current address is the address of the instruction that caused the break or exception.
  \item A task is single-stepped. The current address is the address of the next instruction to be executed.
  \item 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.
\end{itemize}

\section*{6.3.5 Assignments}

The shell C interpreter accepts assignment statements in the form:

\begin{verbatim}
addressExpression = expression
\end{verbatim}

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)
```
6.3.6 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. 188 below.

6.3.7 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"

Furthermore, 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. 120.

After extended development sessions in Tornado shells, the cumulative memory used for strings may be noticeable. If this becomes a problem, restart your target server.
6.3.8 Strings and Path Names

In VxWorks, the directory and file segments of path names (for target-resident files and devices) are separated with the slash character (/). This presents no difficulty when subroutines require a path-name argument, because the / character has no special meaning in C strings.

However, you can also refer from the shell to files that reside on your Windows host. For host path names, you can use either a slash for consistency with the VxWorks convention, or a backslash (\) for consistency with the Windows convention.

Because the backslash character is an escape character in C strings, you must double any backslashes that you use in path names as strings. This applies only to path names in C strings. No special syntax is required for path names that are interpreted directly by the shell.

The shell’s ld() command (System Modification and Debugging, p.162) can be used with all of these variations of path names. The following ld() invocations are all correct and equivalent:

- `ld < c:\fred\tests\zap.o`
- `ld < c:/fred/tests/zap.o`
- `ld 1,0,"c:\fred\tests\zap.o"
- `ld 1,0,"c:/fred/tests/zap.o"

6.3.9 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:

```c
char string [ ] = "hello";
```

you might be tempted to enter a shell expression like:

```bash
printf (string)
```

```bash
1   -> printf (string)
```
While this would be correct in C, the shell will pass the first 4 bytes of the string itself to `printf()`, instead of the address of the string. To correct this, the shell expression must explicitly take the address of the identifier:

\[ \text{printf}(&\text{string}) \]

To make matters worse, in C if the identifier had been declared a character pointer instead of a character array:

\[ \text{char } *\text{string} = \text{"hello";} \]

then to a compiler \( \text{➊} \) would be correct and \( \text{➋} \) 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 `&`.

### 6.3.10 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:

\[ \text{→ *(myPtr + 4) = 5} \]

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.

### 6.3.11 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 6.7 Tcl: Shell Interpretation, p.197.

• **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 work-arounds 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.

### 6.3.12 C-Interpreter Primitives

Table 6-13 lists all the primitives(commands) built into WindSh. (For discussion of these primitives by function, see 6.2.3 Invoking Built-In Shell Routines, p.156.) 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.167.)
6.3.13 Terminal-Control Characters

The terminal-control characters are slightly different when WindSh runs as a console-based application and when it runs within Tornado.

Table 6-14 lists special terminal characters frequently used for shell control in both situations. For more information on the use of these characters, see 6.5 Shell Line Editing, p.193 and 6.2.7 Interrupting a Shell Command, p.173.
6.3.14 Redirection in the C Interpreter

The shell provides a redirection mechanism for momentarily reassigning the standard input and standard output file descriptors just for the duration of the parse and evaluation of an input line. The redirection is indicated by the `<` and `>` symbols followed by file names, at the very end of an input line. No other syntactic elements may follow the redirection specifications. The redirections are in effect for all subroutine calls on the line.

For example, the following input line sets standard input to the file named `input` and standard output to the file named `output` during the execution of `copy()`:

```
-> copy < input > output
```

If the file to which standard output is redirected does not exist, it is created.

**Ambiguity Between Redirection and C Operators**

There is an ambiguity between redirection specifications and the relational operators `less than` and `greater than`. The shell always assumes that an ambiguous use of `<` or `>` specifies a redirection rather than a relational operation. Thus the ambiguous input line:

```
-> x > y
```
writes the value of the variable \( x \) to the stream named \( y \), rather than comparing the value of variable \( x \) to the value of variable \( y \). However, you can use a semicolon to remove the ambiguity explicitly, because the shell requires that the redirection specification be the last element on a line. Thus the following input lines are unambiguous:

\[-\to x; > y\]
\[-\to x > y;\]

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:

\[-\to taskSpawn (...myFunc...) > output\]

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:

1. `-> <startup`
2. `-> < c:\fred\startup`

cause the shell to read and execute the commands in the file `startup`, either on the current working directory as in 1 or explicitly on the complete path name in 2. If your working directory is `\fred`, commands 1 and 2 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 `c:\tmp\script` with the current shell history:

```
-> h > c:\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 precedes 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 by specifying its name with the -s command-line option to windsh. For example:

```
C:\> windsh -s c:\fred\startup
```

Like the autoexec.bat file, 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 6.7 Tcl: Shell Interpretation, p.197.

6.4 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 6.3 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.

---

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 6.7.3 Tcl: Tornado Shell Initialization, p.200.

190
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.

### 6.4.1 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:

```plaintext
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.

```plaintext
-> 1 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
3fe718 f22e 5438 0010   FCMP    .D      (0x10,A6),F0
3fe71e f295 0008        FB      .W      #0x8f22e
3fe720 f22e 5400 0010   FMOVE   .D      (0x10,A6),F0
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
```
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.

### 6.4.2 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 WindShprint 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 6.4.1 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
```

## 6.5 Shell Line Editing

The WindSh front end provides a history mechanism similar to the UNIX Korn-shell history facility, including a built-in line editor (with keystrokes similar to the UNIX editor vi) 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), 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.

The ESC key switches the shell from normal input mode to edit mode. The history and editing commands in Table 6-15 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.159) 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 6-15; you can then edit and execute the line with one of the commands from Table 6-15.

6. The WindSh Tcl-interpreter interface is described in 6.7 Tcl: Shell Interpretation, p.197.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h [size]</td>
<td>Display shell history if no argument; otherwise set history buffer to size.</td>
</tr>
<tr>
<td>ESC</td>
<td>Switch to line-editing mode from regular input mode.</td>
</tr>
<tr>
<td>ENTER</td>
<td>Give line to shell and leave edit mode.</td>
</tr>
<tr>
<td>CTRL+D</td>
<td>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).</td>
</tr>
<tr>
<td>[tab]</td>
<td>Complete symbol or path name (edit mode).</td>
</tr>
<tr>
<td>CTRL+H</td>
<td>Delete a character (backspace).</td>
</tr>
<tr>
<td>CTRL+U</td>
<td>Delete entire line (edit mode).</td>
</tr>
<tr>
<td>CTRL+L</td>
<td>Redraw line (works in edit mode).</td>
</tr>
<tr>
<td>CTRL+S and CTRL+Q</td>
<td>Suspend output, and resume output.</td>
</tr>
<tr>
<td>CTRL+W</td>
<td>Display HTML reference page for a routine.</td>
</tr>
</tbody>
</table>

**Movement and Search Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nG</td>
<td>Go to command number n. *</td>
</tr>
<tr>
<td>/s or ?s</td>
<td>Search for string s backward in history, or forward.</td>
</tr>
<tr>
<td>n</td>
<td>Repeat last search.</td>
</tr>
<tr>
<td>nk or n-</td>
<td>Get nth previous shell command. *</td>
</tr>
<tr>
<td>nj or n+</td>
<td>Get nth next shell command. *</td>
</tr>
<tr>
<td>nh</td>
<td>Go left n characters (also CTRL+H). *</td>
</tr>
<tr>
<td>nl or SPACE</td>
<td>Go right n characters. *</td>
</tr>
<tr>
<td>nw or nW</td>
<td>Go n words forward, or n large words. *†</td>
</tr>
<tr>
<td>ne or nE</td>
<td>Go to end of the nth next word, or nth next large word. *†</td>
</tr>
<tr>
<td>nb or nB</td>
<td>Go back n words, or n large words. *†</td>
</tr>
<tr>
<td>$</td>
<td>Go to end of line.</td>
</tr>
</tbody>
</table>
### Table 6-15  
**Shell Line-Editing Commands (Continued)**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>0</code> or <code>^</code></td>
<td>Go to beginning of line, or first nonblank character.</td>
</tr>
<tr>
<td><code>fc</code> or <code>Fc</code></td>
<td>Find character <code>c</code>, searching forward, or backward.</td>
</tr>
</tbody>
</table>

#### Insert and Change Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code> or <code>A</code></td>
<td>Append, or append at end of line (ESC ends input).</td>
</tr>
<tr>
<td><code>i</code> or <code>I</code></td>
<td>Insert, or insert at beginning of line (ESC ends input).</td>
</tr>
<tr>
<td><code>ns</code></td>
<td>Change <code>n</code> characters (ESC ends input).</td>
</tr>
<tr>
<td><code>nc SPACE</code></td>
<td>Change <code>n</code> characters (ESC ends input).</td>
</tr>
<tr>
<td><code>cw</code></td>
<td>Change word (ESC ends input).</td>
</tr>
<tr>
<td><code>cc</code> or <code>S</code></td>
<td>Change entire line (ESC ends input).</td>
</tr>
<tr>
<td><code>c$</code> or <code>C</code></td>
<td>Change from cursor to end of line (ESC ends input).</td>
</tr>
<tr>
<td><code>c0</code></td>
<td>Change from cursor to beginning of line (ESC ends input).</td>
</tr>
<tr>
<td><code>R</code></td>
<td>Type over characters (ESC ends input).</td>
</tr>
<tr>
<td><code>nrc</code></td>
<td>Replace the following <code>n</code> characters with <code>c</code>.</td>
</tr>
<tr>
<td><code>~</code></td>
<td>Toggle case, lower to upper or vice versa.</td>
</tr>
</tbody>
</table>

#### Delete Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nx</code></td>
<td>Delete <code>n</code> characters starting at cursor.</td>
</tr>
<tr>
<td><code>nX</code></td>
<td>Delete <code>n</code> character to left of cursor.</td>
</tr>
<tr>
<td><code>dw</code></td>
<td>Delete word.</td>
</tr>
<tr>
<td><code>dd</code></td>
<td>Delete entire line (also CTRL+U).</td>
</tr>
<tr>
<td><code>d$</code> or <code>D</code></td>
<td>Delete from cursor to end of line.</td>
</tr>
<tr>
<td><code>d0</code></td>
<td>Delete from cursor to beginning of line.</td>
</tr>
</tbody>
</table>

#### Put and Undo Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p</code> or <code>P</code></td>
<td>Put last deletion after cursor, or in front of cursor.</td>
</tr>
<tr>
<td><code>u</code></td>
<td>Undo last command.</td>
</tr>
</tbody>
</table>

---

* The default value for `n` is 1.
† words are separated by blanks or punctuation; large words are separated by blanks only.
6.6 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 6-3 Loading a Module: Alternate Path Names

```
-> ld < /usr/david/project/test/test.o
Loading /usr/david/project/test/test.o
WTX Error 0x2 (no such file or directory)
value = -1 = 0xffffffff
-> ?shConfig LD_PATH "/usr/david/project/test;C:\project\test"
-> ld < test.o
Loading C:\project\test\test.o
value = 17427840 = 0x109ed80
```

For more information on using LD_PATH and other shConfig facilities, see WindSh Environment Variables, p.154.

⚠️ CAUTION: If you call ld() with an explicit argument list, any instances of the backslash character in Windows paths must be doubled: "n:\fred\applic.o". If you supply the module name with the redirection symbol instead, no double backslashes are necessary.

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 6-16 lists the affected utilities. The object modules are in wind\target\lib\objcputypegnuvx.
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. 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.

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()`.
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 Windows command-line utilities on your development host.

### 6.7.1 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 \texttt{wtxtcl} procedures.

For example, \texttt{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 \texttt{wtxMemRead}, together with the memory-block routine \texttt{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.

```tcl
# 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 \texttt{wtxtcl} routines available in the Tornado shell, see the \textit{Tornado API Guide} (or the \textit{Tornado API} entry in the \textit{Tornado Online Manuals}).

All of the commands defined for the C interpreter (6.2.3 Invoking Built-In Shell Routines, p.156) are also available, with a double-underscore prefix, from the Tcl level; for example, to call \texttt{i()} from the Tcl interpreter, run the Tcl procedure \texttt{__i}. However, in many cases, it is more convenient to call a \texttt{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 \texttt{id} 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 \texttt{wtxtcl} command. In the case of the dynamic linker, the underlying
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")}
```  

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 = ul
```  

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")"}
```  

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:
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 ...}
```

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
```

### 6.7.3 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 `c:\tornado`, 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:

```
C:\> windsh phobos -e "source c:\fred\tcltest"
```
Example 6-4  **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.

```plaintext
# 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
```

6.8  **The Shell Architecture**

6.8.1  **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 6-17 shows how the shell distributes the interpretation and execution of the following simple expression:

```plaintext
-> 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 6-17 omits the following important steps, which must be carried out to link the activities in the three contexts (and two systems) shown in each column of the table:

- After every C-interpreter step, the shell program sends a request to the target server representing the next activity required.
The target server receives each such request, and determines whether to execute it in its own context on the host. If not, it passes an equivalent request on to the target agent to execute on the target.

Table 6-17 Interpreting: dir = opendir("/myDev/myFile")

<table>
<thead>
<tr>
<th>Tornado Shell (on host)</th>
<th>Target Server &amp; Symbol Table (on host)</th>
<th>Agent (on target)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse the string</td>
<td>Allocate memory for the string; return address A.</td>
<td>Write &quot;/myDev/myFile&quot;; return address A.</td>
</tr>
<tr>
<td>/myDev/myFile&quot;.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parse the name</td>
<td>Look up opendir; return address B.</td>
<td></td>
</tr>
<tr>
<td>opendir.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parse the function call</td>
<td>Spawn a task to run opendir() and signal result C when done.</td>
<td></td>
</tr>
<tr>
<td>B(A); wait for the result.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parse the symbol dir.</td>
<td>Look up dir (fails).</td>
<td></td>
</tr>
<tr>
<td>Request a new symbol table entry dir.</td>
<td>Define dir; return symbol D.</td>
<td></td>
</tr>
<tr>
<td>Parse the assignment</td>
<td>Allocate agent-pool memory for the value of dir.</td>
<td>Write the value of dir.</td>
</tr>
<tr>
<td>D=C.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The first access to server and agent is to allocate storage for the string "/myDev/myFile" on the target and store it there, so that VxWorks subroutines (notably opendir() in this case) have access to it. There is a pool of target memory reserved for host interactions. Because this pool is reserved, it can be managed from the host system. The server allocates the required memory, and informs the shell of its location; the shell then issues the requests to actually copy the string to that memory. This request reaches the agent on the target, and it writes the 14 bytes (including the terminating null) there.

The shell’s C-expression interpreter must now determine what the name opendir represents. Because opendir() is not one of the shell’s own commands, the shell looks up the symbol (through the target server) in the symbol table.

The C interpreter now needs to evaluate the function call to opendir() with the particular argument specified, now represented by a memory location on the target. It instructs the agent (through the server) to spawn a task on the target for that purpose, and awaits the result.

As before, the C interpreter looks up a symbol name (dir) through the target server; when the name turns out to be undefined, it instructs the target server to allocate storage for a new int and to make an entry pointing to it with the name dir in the symbol table. Again these symbol-table manipulations take place entirely on the host.

The interpreter now has an address (in target memory) corresponding to dir, on the left of the assignment statement; and it has the value returned by opendir(), on the right of the assignment statement. It instructs the agent (again, through the server) to record the result at the dir address, and evaluation of the statement is complete.

6.8.2 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 6-4 illustrates these components:

Line Editing

The line-editing and command history facilities are designed to be unobtrusive, and support your access to the interpreters. 6.5 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 6.7 Tcl: Shell Interpretation, p.197.

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.

6.8.3 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 6-5. First, input is examined for special editing keystrokes (described in 6.5 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 6.2.3 Invoking Built-In Shell Routines, p. 156 for more information. For a list of all WindSh primitives, see Table 6-13 List of WindSh Commands, p. 186.

Figure 6-5 Layers of Interpretation in the Shell
7

Browser

7.1 A System-Object Browser

The Tornado browser conveniently monitors the state of your target. The main browser window summarizes active system and 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.
7.2 Starting the Browser

There are two ways to start a Tornado browser:

- From the Tornado Launch toolbar, click the button. This launches a browser for the currently selected target server (see Tornado Launch Toolbar, p.142).
- From the Tools menu, click on Browser. The dialog box shown in Figure 7-1 appears, to allow you to select a target server from the Targets drop-down list.

Figure 7-1 Browser Target-Selection Dialog Box

7.3 Anatomy of the Browser Window

The top of the browser window, shown in Figure 7-2, allows you to request particular browser displays and to control other browser functionality.

Browser Window Selector

Clicking this drop-down list box displays the list of specialized browser displays you can choose from. Each specialized display is described in a section of this chapter.
You can choose among the following browser displays:

- **Target Information**: See 7.5 Target-Information Window, p.211.
- **Tasks**: See 7.6 Task-List Window, p.213.
- **Memory Usage**: See 7.7 Memory-Usage Window, p.214.
- **Object Information**: See 7.8 Object-Information Windows, p.215.
- **Stack Check**: See 7.11 The Stack-Check Window, p.225.
- **Module Information**: See 7.9 The Module-Information Window, p.221.
- **Spy Chart**: See 7.10 The Spy Window, p.224.
- **Vector Table**: See 7.12 The Vector Table Window, p.226. This selection is available only on supported architectures.

**Button Bar**

Buttons to give you fresh snapshots of your target, and to adjust browser parameters. 7.4 Browser Buttons, p.210 describes each button.

**Data Area**

Below the controls, most of the browser window is devoted to a display of the selected target information. The default Target Information window provides a summary of the target hardware and software on the current target.

When you choose a particular browser display, the contents of the data area are replaced in the currently active browser window. You can have as many browser windows open as necessary to monitor different aspects of the target system simultaneously.
7.4 Browser Buttons

The button bar at the top right of each browser window provides the following controls:

1. **Immediate-update button.** Use this button to update all browser displays immediately. You can use this button for an immediate update even if a periodic update is running.

2. **Periodic-update button.** This button is a toggle: press it to request or cancel regular updates of all browser displays (the time period between updates is one of the parameters controlled by the button, described below).

3. **Parameter adjustment button.** Press this button to adjust the parameters that govern the browser’s behavior. Figure 7-3 shows the Browser Configuration dialog box, displayed when you click this button.

Figure 7-3 Browser Configuration Dialog Box

The Browser Configuration dialog box contains the following controls:

**Symbol sort**

To select alphabetic sorting order for symbols displayed by the browser, check this box; otherwise, the browser uses numeric order.

**Spy mode**

To select differential mode for the spy window, check this box; otherwise the browser uses cumulative mode (see 7.10 The Spy Window, p.224).

**Browser/Spy update time**

Specify how often to update browser windows (when periodic updates are running).
Spy collect frequency
   Specify how many times per second to gather data for the spy window.

Reuse Browser Window
   To open a new window when you double-click on an object ID (for example, a
   task ID) in the browser, leave this box un-checked. To discard the data area of
   the current window in that situation, replacing it with data for the object you
   clicked on, check this box.

7.5 Target-Information Window

The Target Information window (Figure 7-4) displays the following summary
information about the selected target:

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

Status
This field indicates whether a target is reserved (see the User field) or unreserved. Anyone may connect to an unreserved target.¹

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

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

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

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

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

Link
The physical connection mechanism to the target.

User
The sign-on name 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.

¹ You can also restrict your target servers to permit connections only by a particular list of users; see Sharing and Reserving a Target Server, p.145.
7.6 Task-List Window

Figure 7-5 shows two examples of the task list produced by clicking Tasks in the browser window selector.

Figure 7-5  System Tasks and Application Tasks

The tasks in this window are organized into two lists:

- **WRS Tasks**

  Summary information on all operating-system tasks running on the target.

- **User Tasks**

  Summary information on all application tasks running on the target.

Each task list is marked with a small (minus sign) icon. You can double-click on this icon to hide data that is not of current interest; the browser changes the marker to a small (plus sign) to indicate that hidden data is available. To reveal the contents of a hidden task list, click the (plus-sign) icon. (This convention for controlling the display of hierarchical information is used in many parts of Tornado.)

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 7.8.1 The Task Browser, p.216.
7.7 Memory-Usage Window

Figure 7-6 shows an example of the window produced by clicking Memory Usage in the browser window selector.

The Memory Usage window has the following areas:

- Memory-Consumption Graphs

  The two bar graphs in this area 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. This represents target memory consumed by Tornado tools, for example with dynamically linked modules or for variables defined from the shell.

  The lower bar shows the memory consumed by all tasks in the target system, including both application (user) tasks and system tasks.

  The agent-memory pool is not part of VxWorks' memory. If the target server wants to allocate more memory than available in the agent-memory pool, it will allocate memory from the VxWorks memory pool and add it to the agent-memory pool.

2. To set the size of this memory pool, see *Scaling the Target Agent*, p.120.
Clicking on the lower bar produces a more detailed display of system memory (the memory display described in 7.8.4 The Memory-Partition Browser, p.219, applied to the system-memory partition; this display is shown in Figure 7-12).

In both bars, the numeric label inside the bar measures the memory currently in use. The numbers below the right edge of each graph indicate the total memory size available in each pool. All memory-size numbers are byte counts in decimal.

- **Loaded-Module List**

  This area lists each object file currently loaded on your target. This includes the VxWorks image and all dynamically linked object modules.

### 7.8 Object-Information Windows

Clicking Object Information in the browser window selector brings up a window where you can request a specialized display for VxWorks system objects (Figure 7-7). Type or paste either the name or the ID of a system object in the text box next to the Show button. Then click Show (or press ENTER) to display information about that particular object. As a convenient shorthand, we refer to the browser’s object-information windows as object browsers: task browsers, semaphore browsers, and so on.

Another way to bring up an object-information window is to click on the name of an object in the module browser (7.9 The Module-Information Window, p.221). If the object is a recognized system object, the data area for it is displayed just as if you had copied the name to the Show box.

For example, Figure 7-7 shows the Show box filled in with a request to display an object called graphSem.

![Figure 7-7 Filling in the Object-Display Box](image)
7.8.1 The Task Browser

To see more detailed information about a particular task, click on any task’s summary line in a browser Tasks window (or enter the task name or task ID in the Show box of an Object Information browser window). The browser displays a window for that task, similar to Figure 7-8.

Figure 7-8 Task Browser (Initial Display)

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 (minus sign) icons; the lines they mark categorize the task information. As in other parts of Tornado that display hierarchical data, you can hide categories by clicking on any icon, or expose hidden categories by clicking on any (plus sign) icon.

Figure 7-9 shows another task browser running on the same target, but with most of the hardware registers hidden.
7.8.2 The Semaphore Browser

To inspect a semaphore, enter either its name or its semaphore ID in the Show box of a browser window with Object Information selected. A specialized semaphore browser appears, similar to the one shown in Figure 7-10. 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.

Figure 7-10 shows a binary semaphore with several blocked tasks in its queue. As in other browser windows, you can click on the levels of the display to control detail. To start a browser for any queued task, click on the task name or ID; both are displayed for each task.

POSIX semaphores have a somewhat different collection of attributes, and the browser display for a POSIX semaphore reflects those differences. Similarly, the semaphore browser adapts to shared-memory semaphores.
7.8.3 The Message-Queue Browser

To inspect a message queue, enter its name or message-queue ID in the Show box of a browser window with Object Information selected. A message-queue browser like the one in Figure 7-11 is displayed.

In addition to 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.

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 version for POSIX message queues.
7.8.4 The Memory-Partition Browser

The memory-partition browser comes up when you enter a memory partition ID (or the name of a variable containing one) in the Show box of a browser window with Object Information selected, as do all specialized browser windows. Figure 7-12 shows the VxWorks system memory partition, memSysPartId.

By default the memory-partition browser displays the following:

- The total size of the partition.
- The number of blocks currently allocated, and their total size in bytes.
- The number of blocks currently free, and their total size in bytes.
- The totals allocated since booting the target system (headed Cumulative).
- The size and address of each block currently on the free list.

As with other object browsers, you can control the level of detail visible by clicking on the (minus sign) or (plus sign) icons beside each heading.

7.8.5 The Watchdog Browser

When the Tornado browser recognizes a watchdog-timer ID (or a variable containing one) in the Show box of a browser window with Object Information selected, it displays a window like those shown in Figure 7-13.
Before you start a timer, the display resembles the one on the left of Figure 7-13; only the state field is particularly meaningful. However, after the timer starts counting, you can see the number of ticks remaining, the address of the routine to be executed when the timer expires, and the address of the routine’s parameter.

### 7.8.6 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 of a browser window with Object Information selected, the browser displays overall information about the objects in that class.

For example, Figure 7-14 shows the display for semClassId (the semaphore class).

Figure 7-14  Class Browser (Semaphore Class)

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

```
-> lookup "ClassId"
```

### 7.9 The Module-Information Window

To inspect the memory map of any currently loaded module, click on the line that lists that module in the loaded-module list (described in 7.7 Memory-Usage Window, p. 214).

The browser opens a module-information window resembling Figure 7-16 for the selected object module.
Figure 7-15  Loaded-Module List in Memory Usage Window

```
AgentPool

VNodeId: 0

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mBox</td>
<td>0x75a3</td>
<td>0x2c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>vColor</td>
<td>0x75a4</td>
<td>0x18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>libLib</td>
<td>0x75a5</td>
<td>0x02</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>0x75a6</td>
<td>0x2c</td>
<td>0x18</td>
</tr>
</tbody>
</table>
```

Figure 7-16  Module Information Window

```
Module:
- name = mBox
- info = LDFD_GLOBAL_SYMBOLS
- DHF = 0x75a4
- group = 0x2
- Segments
  - Type: test
    - address = 0x75a4
    - size = 0x2c
- Test_Symbols
  - 0x75a5:0x600 T_neighborTest
  - 0x75a4:0x600 T_graphColoring
  - 0x75a3:0x600 T_consistencyTest
  - 0x75a2:0x600 T_dataChange
  - 0x75a1:0x600 T_graphDisp
  - 0x75a0:0x600 T_graphHtml
  - 0x75a9:0x600 T_graphControl
  - 0x75a8:0x600 T_graphCreate
  - 0x75a7:0x600 T_graphDelete
  - 0x75a6:0x600 T_graphConnect
  - 0x75a5:0x600 T_graphTest
- Other_Symbols
  - 0x75a4:0x600 T_mBoxSync56m
  - 0x75a3:0x600 T_mBoxSync56c
```
The module-information window displays information in the following categories:

**Module**
- Overall characteristics of the object module: its name, the loader flags used when the module was downloaded, the object-module format (OMF), and the group number. (The group number identifies all of the symbols from 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 numerical 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 7-1), and the symbol name (in its internal representation—C++ symbols are displayed “mangled,” and all compiled-language symbols begin with an underbar).

<table>
<thead>
<tr>
<th>Symbol Key</th>
<th>Global</th>
<th>Local</th>
<th>Symbol Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
<td>absolute</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td>bss segment</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>common (uninitialized global symbol)</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>d</td>
<td>data segment</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>t</td>
<td>text segment</td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>unrecognized symbol</td>
<td></td>
</tr>
</tbody>
</table>

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.
For symbols that represent system objects, clicking on the symbol name brings up the corresponding object browser; see 7.8 Object-Information Windows, p.215.

### 7.10 The Spy Window

Clicking Spy Chart in the browser window selector produces a display similar to Figure 7-17. This window reports on CPU utilization for each task on your target, as a percentage of CPU cycles. Besides tasks, the spy window always includes the following additional categories for CPU-cycle usage: time spent in the kernel, time spent in interrupt handlers, and idle time. These additional categories appear below all task data; you may need to use the scrollbar to see them.

Spy data is reported in one of two modes (selected with the Browser Configuration dialog box shown in Figure 7-3). When the Differential check box is off in the Browser Parameters dialog box (see 7.4 Browser Buttons, p.210), the spy window is in cumulative mode: it shows total CPU usage since you first display the spy window. Spy reports in differential mode reflect only the CPU usage since the last update.
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**.

### 7.11 The Stack-Check Window

Clicking Stack Check in the browser window selector produces a window similar to Figure 7-18. The stack-check window summarizes the current and maximum stack usage for each task currently running.

**Figure 7-18  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 shaded portion of each task’s bar.
- The portion of the stack currently in use, shown as a number of bytes, displayed within the bar graph for each task.
7.12 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 Windows simulator, PowerPC, and ARM.) The display is similar to Figure 7-19.

Figure 7-19 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.

7.13 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. On slow links, this traffic may seriously slow down overall Tornado performance. If you experience this problem, try displaying fewer objects, updating browser displays on request instead of periodically, or setting updates to a longer interval.

7.14 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.

7.14.1 Memory Leaks

The memory-consumption bar graphs in the Memory Usage window makes memory leaks easy to notice. If the allocated portion of memory grows continually, you have a problem. The memory-consumption graph in Figure 7-20 indicates a memory leak in an application that has run long enough to exhaust memory almost completely.
7.14.2 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 many applications this is the VxWorks system memory partition, memSysPartId) with the browser. Figure 7-21 shows an example of a growing free-list with many small blocks, characteristic of memory fragmentation.

7.14.3 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 (Figure 7-22).

7.14.4 Priority Inversion

Browser displays are most useful when they complement each other. For example, suppose you notice in a task-list window (as in Figure 7-23) that a task uHi, expected to be high priority, is blocked while two other tasks are ready to run.
Figure 7-21  Fragmented Memory as Seen in the Browser

Figure 7-22  Stack Overflow on Task u9
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: \texttt{uHi} is supposed to have highest priority, \texttt{uMed} medium priority, and \texttt{uLow} the lowest). You can check this immediately by clicking on each task’s summary line, thus bringing up the windows shown in Figure 7-24.

Unfortunately, that turns out to be the wrong explanation; the priorities (shown for each task under Attributes) are indeed as expected. Examining the CPU allocations with the spy window (Figure 7-25) reveals that the observed situation is ongoing; \texttt{uMed} is monopolizing the target CPU. It should certainly execute by preference to the low-priority \texttt{uLow}, but why is \texttt{uHi} not getting to run?

At this point examining the code (not shown) may seem worthwhile. Doing so, you notice that \texttt{uMed} uses no shared resources, but \texttt{uHi} and \texttt{uLow} synchronize their work with a semaphore. Could that be the problem?
Examining the semaphore with the browser (Figure 7-26) confirms this suspicion: \texttt{uHi} is blocking on the semaphore, which \texttt{uLow} cannot release because \texttt{uMed} has preempted it.

Having diagnosed the problem as a classic priority inversion, the fix is straightforward. As described in \textit{VxWorks Programmer's Guide: Basic OS}, you can revise the application to synchronize \texttt{uLow} and \texttt{uHi} with a mutual-exclusion semaphore created with the \texttt{SEM_INVERSION_SAFE} option.
7.15 Tcl: the Browser Initialization File

When the browser begins executing, it first checks for a file called .wind\browser.tcl in two places: first under c:\tornado, and then in the directory specified by the HOME environment variable (if that environment variable is defined). In each of these directories, if a browser.tcl file is found, its contents are sourced as Tcl code; this gives you an opportunity to customize the browser.
8

Debugger

8.1 Introduction

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 complex or unpredictable debugging needs, the command-line interface gives you full access to a wealth of specialized debugging commands. You can extend or automate command-line debugger interactions by using the Tcl scripting interface that allows you to develop custom debugger commands (see 8.1 Introduction, p.233).

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.

NOTE: For a tutorial introduction to the Tornado debugger, see the Tornado Getting Started Guide.
8.2 Debugger GUI

Figure 8-1 illustrates the GUI elements you can use to interact with the debugger.

The Debug menu provides the complete list of Tornado GUI debugger commands, as well as their keyboard shortcuts (8.2.1 Debugger Toolbar, Buttons, and Menu Commands, p.235).

The Debug toolbar provides buttons for the most common debugger commands, as well as for opening and closing windows that display data, memory, and stack information (see 8.2.1 Debugger Toolbar, Buttons, and Menu Commands, p.235).

You use the editor window to keep track of the code you are debugging. You can click in this window to specify information for debugger commands (such as symbol names, or lines of code). The debugger in turn uses the attribute panel, in the left margin of the editor window, to show breakpoints and the execution context. (See 8.3 Using the Debugger, p.237.)
The debugger command line window for CrossWind provides a command-line interface to the debugger (see 8.5 Using the Debugger Command Line, p.256). The window is not needed for standard debugger operations; the graphical interface provide simpler controls.

### 8.2.1 Debugger Toolbar, Buttons, and Menu Commands

The Debug toolbar has buttons for the most common debugging commands, as well as for displaying auxiliary debugger windows. The toolbar shown as a floating palette in Figure 8-2.

![Debug Toolbar](image)

Figure 8-2  Debug Toolbar

The commands in the Debug menu include alternatives to the buttons in the Debug toolbar, as well as additional debugger functions. Keyboard shortcuts are also available for all graphical debugger commands.

The debugger buttons and menu commands are described in Table 8-1.

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>Source Search Path</td>
<td>Identify source search paths for the debugger. Opens the Debugger Source Search Path dialog box. See 8.3.2 Setting the Search Path, p.239.</td>
</tr>
<tr>
<td><img src="image" alt="Run" /></td>
<td>Run</td>
<td>Run a routine on the target as a new task under debugger control. Opens the Run Task dialog box, which allows you to choose the routine. See 8.3.7 Continuing Through a Program, p.244.</td>
</tr>
<tr>
<td>n/a</td>
<td>Detach</td>
<td>Detach from the task currently under debugger control, leaving it in the state present (suspended or running) when you give the command. See 8.3.9 Detaching from a Running Task, p.247.</td>
</tr>
<tr>
<td>n/a</td>
<td>Detach and Resume</td>
<td>Detach from the task currently under debugger control, first ensuring that it is running. See 8.3.9 Detaching from a Running Task, p.247</td>
</tr>
</tbody>
</table>
Table 8-1  **Debugger Buttons and Commands (Continued)**

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>Attach</td>
<td>Attach the debugger to a task. See 8.3.8 Attaching to a Running Task, p.246.</td>
</tr>
<tr>
<td></td>
<td>Interrupt Debugger</td>
<td>Interrupt program execution. See 8.3.5 Interrupting a Program, p.241.</td>
</tr>
<tr>
<td></td>
<td>Stop Debugging</td>
<td>Stop the debugger. See 8.3.1 Starting and Stopping the Debugger, p.238.</td>
</tr>
<tr>
<td>n/a</td>
<td>Breakpoints</td>
<td>Open the Breakpoints window. See 8.3.6 Setting Breakpoints, p.241.</td>
</tr>
<tr>
<td></td>
<td>Toggle Breakpoint</td>
<td>Set or remove a task-level breakpoint on the current line of the editor window. To delete a breakpoint, click this button on a line that is already marked with the breakpoint icon. See 8.3.6 Setting Breakpoints, p.241.</td>
</tr>
<tr>
<td>n/a</td>
<td>Toggle Global Breakpoint</td>
<td>Set or remove a global breakpoint on the current line of the editor window. To delete a breakpoint, click this button on a line that is already marked with the breakpoint icon.</td>
</tr>
<tr>
<td>n/a</td>
<td>Toggle Temp. Breakpoint</td>
<td>Set or remove a temporary breakpoint. See 8.3.6 Setting Breakpoints, p.241.</td>
</tr>
<tr>
<td></td>
<td>Step Into</td>
<td>Step to the next line of code, in order of execution (not necessarily the next line displayed in the editor). See 8.3.7 Continuing Through a Program, p.244.</td>
</tr>
<tr>
<td></td>
<td>Step Over</td>
<td>Step to the next line displayed on the screen. If there is a subroutine call on the current line, the button executes that subroutine in its entirety, then stops at the line after the subroutine call. See 8.3.7 Continuing Through a Program, p.244.</td>
</tr>
<tr>
<td></td>
<td>Continue</td>
<td>Continue program execution. The task you are debugging continues until the next breakpoint, exception, or until you use Interrupt Debugger to halt it. See 8.3.7 Continuing Through a Program, p.244.</td>
</tr>
</tbody>
</table>
8.3 Using the Debugger

The debugger allows you to download object modules, to start routines under debugger control, and to take over existing tasks in the target. Tasks under debugger control execute normally until they terminate, unless they encounter a breakpoint, or you interrupt them, or some other event sends them an interrupt or a signal.

⚠️ **WARNING:** You must compile your application using debugging symbols (the `-g` GNU compiler option) to use many of the features of the debugger. The default compiler settings used by the Tornado project facility include debugging symbols.

---

### Table 8-1 Debugger Buttons and Commands (Continued)

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Step Out" /></td>
<td>Step Out</td>
<td>Finish the current subroutine. Execution continues until the current subroutine returns to its caller. See 8.3.7 Continuing Through a Program, p.244.</td>
</tr>
<tr>
<td><img src="image" alt="Watch" /></td>
<td>Watch</td>
<td>Open and close the Watch window, which displays the values of specified variables throughout the execution of the program. See Watch Window, p.248.</td>
</tr>
<tr>
<td><img src="image" alt="Variables" /></td>
<td>Variables</td>
<td>Open and close the Variables window, which displays the values of local variables. See Variables Window, p.250.</td>
</tr>
<tr>
<td><img src="image" alt="Registers" /></td>
<td>Registers</td>
<td>Open and close the Registers window, which displays values of the target registers. See Registers Window, p.251.</td>
</tr>
<tr>
<td><img src="image" alt="Back Trace" /></td>
<td>Back Trace</td>
<td>Open and close the Back Trace window, which displays stack information. See Back Trace Window, p.252.</td>
</tr>
<tr>
<td><img src="image" alt="Memory" /></td>
<td>Memory</td>
<td>Open and close the Memory window, which displays target memory information. See Memory Window, p.253.</td>
</tr>
</tbody>
</table>
8.3.1 Starting and Stopping the Debugger

There are two ways to start a debugging session:

- From the Tornado Launch toolbar: Press the button. This starts a debugging session for the currently selected target server (see Tornado Launch Toolbar, p. 98).
- From the Tools menu: Click on Debugger. The dialog box shown in Figure 8-3 appears, to allow you to select a target server from the Targets drop-down list.

When the debugger is running, you can interact with it through the editor window, through the debugger command line window, and through the Debug menu and toolbar. (See 8.2 Debugger GUI, p. 234.)

You can end the debugging session in any of the following ways:

- In the debug toolbar, press the button.
- Click on the Stop Debugging command in the Debug menu.
8.3.2 Setting the Search Path

The debugger maintains a list of directories where it searches for source code, and for the host-resident image of the VxWorks run-time (the debugger uses the latter to load debugging symbol information generated by the -g compiler option). This list is called the *source search path*.

Normally you will not need to set the source search path because the debugger can derive the path from the object file. If GDB finds the object but cannot find the source, the GUI prompts you for the source file location and remembers it.

If necessary, click on **Source Search Path** to add directories to, remove directories from, or change the order of this list. Figure 8-4 illustrates the Debugger Source Search Path dialog box.

![Debugger Source Search Path Dialog Box](image)

**WARNING:** Do not leave more than one directory containing a run-time core image in the search path. The debugger uses the first of these it encounters; if it is not the correct core image, the results are unpredictable, and the debugging session may hang.
8.3.3 Unloading a Module

Normally you will not need to unload a module. If you update and download a module with the same name, it replaces the module loaded earlier. In the unusual case where you need to unload a module without replacing it, use unload option from the context menu in the project Workspace window (see 4.2.5 Downloading and Running an Application, p.89).

You can also use the debugger command line to remove any dynamically-linked module from the target. Open the debug command-line window and use the GDB **unload** command. See 8.5 Using the Debugger Command Line, p.256 for more information about command line usage.

8.3.4 Running a Program

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>![run_icon] (F6)</td>
<td>Run</td>
<td></td>
</tr>
</tbody>
</table>

To run a subroutine under debugger control, use the Run command. The Run Task dialog box appears; use it to specify which routine to run, with what arguments. Click OK to start the task on the target.

![Run Task Dialog Box]

Specify the argument list after the routine name, with the arguments separated by spaces. The argument list must contain integers or addresses only; it may not contain floating-point or double-precision values, function calls, or user-defined C++ operations. (See **GDB User’s Guide** for other commands which allow arbitrary arguments to be passed.)

Figure 8-5 shows the Run Task dialog box with a routine name (required) and an argument list (optional). The default for required arguments that you do not
supply is zero. To set a temporary breakpoint where the routine begins execution, check the Break at Entrypoint box.

Once a task stops under debugger control (most often, at a breakpoint), you can single-step through the code, jump over subroutine calls, or resume execution. Figure 8-10 shows the debugger stopped at the entry point of the routine *BigBang*(). The context pointer indicates what statement will execute if you allow the program to resume.

**Figure 8-6  The Context Pointer**

```c
void bigBang(void) {
  struct byLightening *head, *prev, *curr;
  int i;
  head = nodeAdd(hitCount());
  prev = head;
  for (i = 0; i < NODES; i++)
    {
```

### 8.3.5 Interrupting a Program

You can interrupt execution with *Interrupt Debugger* in the Debug menu, by pressing the button, or by using the keyboard shortcut **ALT+SHIFT+F5**.

### 8.3.6 Setting Breakpoints

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Menu Command</th>
<th>Pop-up Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>n/a</td>
<td>Breakpoints</td>
<td>Breakpoints</td>
</tr>
<tr>
<td>!</td>
<td>F9</td>
<td>Toggle Breakpoint</td>
<td>Toggle Breakpoint</td>
</tr>
<tr>
<td>n/a</td>
<td>F8</td>
<td>Toggle Temp. Breakpoint</td>
<td>Toggle Temp. Breakpoint</td>
</tr>
</tbody>
</table>

To set multiple breakpoints, select Breakpoints in either the Debug or the pop-up menu (right-click in the editor window). The Breakpoints dialog box appears. (See Figure 8-7.) Enter a file name and line number in the Location box, select a scope (local task or all tasks), and click Add. The new breakpoint appears in the Breakpoints list. If Externally managed is checked, it indicates that the breakpoint was set by means other than the debugger (by the Tornado shell, for example).

The Advanced button opens the Advanced Breakpoint dialog box (Figure 8-8). This dialog lets you attach conditions to the break point as well as deleting or disabling...
it instead of keeping it when it is hit. Enter a file name and line number in the Location box. A conditional expression of type \texttt{int} can be used, which will be evaluated as true or false (all non-zero values being true), or to check if a value in memory has changed. The On Break options have the following meanings with regard to handling a breakpoint:

Keep
   Defines the breakpoint as permanent.

Delete
   Defines the breakpoint as temporary. The breakpoint is deleted after it is hit.
Disable

Defines the breakpoint as temporary. The breakpoint is disabled after it is hit (not deleted), and can be manually enabled.

To set a breakpoint on an individual line of code, place the text cursor in the line where you want the program to stop. Then click the button, select Toggle Breakpoint from the Debug menu, or right-click on the line of code and select Toggle Breakpoint from the pop-up menu. The symbol appears in the left margin of the editor window to indicate the breakpoint location, when the attribute pane is turned on for the editor (Editor Preferences, p.283). Otherwise, the entire line is highlighted to indicate the breakpoint location.

If you click Toggle Breakpoint on a line that produces no object code (such as a comment line or a declaration), the breakpoint appears on the next line that does produce object code.

You can also set temporary breakpoints by using Toggle Temp. Breakpoint from the Debug menu. A temporary breakpoint stops the program only once. The debugger deletes it automatically as soon as the program stops there. A hollow breakpoint symbol ( ) marks temporary breakpoints in the editor’s left margin, so that you can readily distinguish the two kinds of breakpoints there.

To remove either kind of breakpoint, click either of these commands with a line selected that already has a breakpoint or use the Breakpoints dialog box.

⚠️ **CAUTION:** If your application was compiled without debugging information, the debugger displays an error when you try to set a breakpoint using these commands. If you are forced to work on an object module without debugging information, you can still break at the start of any subroutine in either of the following ways: (1) Check the Break at Entrypoint check box in the Run Task dialog box when you start the task (Continuing Through a Program, p.244). (2) Use the Breakpoints dialog box (see Setting Breakpoints, p.241). In either case, when the debugger stops, it displays a Disassembly window, as it does whenever no debugging information is available for the program context.

Figure 8-9 shows the editor window with two breakpoints set: a temporary breakpoint (hollow) and a persistent breakpoint (filled in).
8.3.7 Continuing Through a Program

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Menu Command</th>
<th>Pop-up Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F5</td>
<td>Continue</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>F10</td>
<td>Step Into</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>CTRL+F5</td>
<td>Step Over</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>SHIFT+F5</td>
<td>Step Out</td>
<td>n/a</td>
</tr>
<tr>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Run to Cursor</td>
</tr>
<tr>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>Start a new task from current function</td>
</tr>
</tbody>
</table>

Once a task has been stopped under debugger control (most often, at a breakpoint), you can single-step through the code, jump over subroutine calls, or resume execution. Figure 8-10 shows the debugger stopped at the entry point of the routine `graphInit()`. The context pointer ➊ indicates what statement will execute if you allow the program to resume.

When the program is stopped, you can resume operation with the Continue command from the Debug menu. If there are no remaining breakpoints, interrupts, or signals, the task runs to completion. A common example of using Continue is to
set a breakpoint at the end of a loop, then use Continue repeatedly to stop once in each iteration through the loop, while monitoring a variable used within that loop.

To step through the code one line at a time, click Step Into. If you have auxiliary debugger windows open (8.3.10 Examining Data, Memory, and the Stack, p.247), they are updated with current values as you step through the code. If there is a subroutine call in the current line, Step Into takes you to the first line of that subroutine, not to the next line currently displayed on your screen. The only exception is for calls to system subroutines and application subroutines that are compiled without debugging information; Step Into cannot step into these.

The effect of Step Into is somewhat different if the current view in the editor shows assembly instructions (when either Disassembly or Mixed is selected from the View menu, or the current routine has no debugging symbols). In this case, Step Into advances execution to the next instruction rather than to the next line of source.

To single-step without going into other subroutines, click Step Over instead of Step Into. The Step Over command is almost the same as Step Into, but instead of stepping to the very next statement executed (which, in the case of a subroutine call, is typically not the next statement displayed), Step Over steps to the next line on the screen. If there is no intervening subroutine call, this is the same thing as Step Into. But if there is an intervening subroutine call, Step Over executes that subroutine in its entirety, then stops at the line after the subroutine call.

The display style has the same effect on Step Over as on Step Into. Step Over steps to either the next machine instruction or the next source statement, and if necessary completes a subroutine call first.
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 the Step Out command from the Debug menu in that situation: execution continues until the current subroutine completes, then the debugger regains control in the calling statement.

To continue a stopped task to a specific point without setting a breakpoint, place the cursor on the desired line of code, right-click to open the pop-up menu, and select Run to Cursor.

To start execution from a specific point, place the cursor on the desired line of code, right-click to open the pop-up menu, and select Start a new task from current function.

### 8.3.8 Attaching to a Running Task

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>ALT+F6</td>
<td>Attach</td>
</tr>
</tbody>
</table>

Click Attach to attach the debugging session to a task that is already running. If you were already debugging another task, the previous task is released from debugger control, remaining in its current state (running or stopped). Attach displays a scrolling list of the tasks running on the target (Figure 8-11). You can either select a task in the list, or type the name (or task ID) of a task in the Attach to: box. The debugger stops the task immediately after attaching.

![Figure 8-11](image)

Usually, a newly-attached task stops in a system routine; thus, the debugger displays an assembly listing in the editor window. Open the backtrace window.
using the button. Change the frame by double-clicking on the frame you want to jump to.

The first entry in the Attach dialog box is always System. Select this entry to enter system mode, as described in 8.6 System-Mode Debugging, p.263. An error display appears if your BSP is not configured to support system mode.

**8.3.9 Detaching from a Running Task**

Click Detach and Resume to release the current task from debugger control (and allow it to continue running). This allows you to exit the debugger, or switch to system mode, without killing the task you were debugging.

**8.3.10 Examining Data, Memory, and the Stack**

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Windows Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="button.png" alt="Watch" /></td>
<td>ALT+3</td>
<td>Watch</td>
</tr>
<tr>
<td><img src="button.png" alt="Variables" /></td>
<td>ALT+4</td>
<td>Variables</td>
</tr>
<tr>
<td><img src="button.png" alt="Registers" /></td>
<td>ALT+5</td>
<td>Registers</td>
</tr>
<tr>
<td><img src="button.png" alt="Memory" /></td>
<td>ALT+6</td>
<td>Memory</td>
</tr>
<tr>
<td><img src="button.png" alt="Back Trace" /></td>
<td>ALT+7</td>
<td>Back Trace</td>
</tr>
</tbody>
</table>

When a program stops under debugger control, you can use auxiliary windows to examine local and global program variables, arguments, registers, target memory, and the execution stack. The windows can be displayed docked or free-floating (Figure 8-12). When they are docked (the default), the split bars at their edges can be used to change their size.

Buttons for the auxiliary debugger windows are on the debugger toolbar, and provide a simple means for opening and closing the windows. They are also accessible from Debug>Debug Windows.
The Watch window displays the current values of symbols, throughout the execution of the program. The Watch window has four pages, which allows you to group and display sets of symbols in any manner you find useful.

To select a symbol for display in the watch window, click on or highlight the symbol name in the editor, display the pop-up menu (with the right-mouse button), and select Add to Watch. If you highlighted the symbols name, the Watch
window opens, and lists the symbol and its current value. If you simply clicked on
the symbol, the Add to Watch dialog box opens (Figure 8-13).

Figure 8-13 Add to Watch Dialog Box

Enter the name of the symbol and click OK to display the symbol and its current
value in the Watch window (Figure 8-14). Use the pop-up menu or the Delete key
to remove symbols from the window.

Figure 8-14 Debugger Watch Window
Variables Window

Click the Variables command in the Debug Windows sub-menu to open a window that shows the values of local variables. Figure 8-15 shows an example of the Variables window.

Figure 8-15  Debugger Variables Window

The contents of the Variables window always reflect the routine that is currently executing; when you step into a different routine, the new routine’s local variables replace those in the previous display.
Registers Window

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Windows Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ALT+5</td>
<td>Registers</td>
</tr>
</tbody>
</table>

Click the Registers command in the Debug menu to open a window that shows the values in the target registers. Figure 8-16 illustrates the Registers window. The contents of the window depend on the architecture of the target, and the title displayed when the window is not docked includes information about machine architecture.

Figure 8-16 Debugger Registers Window

```c
struct byLightening * nodeAdd(int hitNum) {
    int div;
    struct byLightening *node;
    node = (struct byLightening *) xalloc(sizeof(struct
    node->next = NULL;
    node->num = hitNum;
    div = LOG2F(hitNum);
    node->color = hitNum / div;
    return(node);
}

void report(struct byLightening *node) {
    while(node->next != NULL) {
        printf("hit %d at %d\n", node->num, (int)node
        node = node->next;
    }
}

void bigScan(void) {
    struct byLightening *head, *prev, *curr;
    int i;
    head = nodeAdd(hitCount());
    prev = head;
    for (i = 0, i < NODES, i++) {
        curr = nodeAdd(hitCount());
        prev->next = curr;
        prev = curr;
        report(head);
    }
```
**Back Trace Window**

To inspect the calling sequence leading to the current routine, click Back Trace in the Debug Windows sub-menu. The Back Trace window allows you to monitor the stack. You can double-click on any routine in the window to move the context pointer to that stack level in the editor window (Figure 8-17).

![Debugger Back Trace Window](image)

To inspect the calling sequence leading to the current routine, click Back Trace in the Debug Windows sub-menu. The Back Trace window allows you to monitor the stack. You can double-click on any routine in the window to move the context pointer to that stack level in the editor window (Figure 8-17).
Click the Memory command to open a window that displays a range of target memory starting at the address specified in the Start Address control field. The debugger saves each address you type in the field; you can select a previously displayed address from the drop-down list associated with this box. To update the memory display, press the button. Figure 8-18 shows the window docked and maximized.

Figure 8-18 Debugger Memory Window
The display of data in the Memory window is controlled by the Debugger page of the Options property sheet, which is accessible from Tools>Options>Debugger (see 9.2.7 Debugger, p.291).

See the description of the x (“examine”) command in GDB User’s Guide: Examining Data for a discussion of the memory-display formats.

## 8.4 Source Code Display Options

While the debugger is running, you can display your program code in the editor as source code, as symbolic disassembly of object code, and as mixed source and disassembly.

The Source, Disassembly, and Mixed Source and Disassembly commands in the View menu control the display of code in the editor:

**Source**
- Displays the original high-level language source code (usually C or C++). This is the default style of program display.

**Disassembly**
- Displays 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).

**Mixed Source and Disassembly**
- 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.

Figure 8-19 shows a mixed-mode code display.

Source display (in either the Source or the Mixed Source and Disassembly view) requires that your application modules be compiled with debugging information (the -g option with the GNU compiler) which is the default for compilations with the Tornado project facility (see Projects, p.71).
The debugger is fully operational no matter what view you select. For example, you can set breakpoints in a line of assembly code, and you can use the Step and Next commands in either assembly or source. (In views that show assembly, these commands step by instructions rather than by source lines; see 8.3.7 Continuing Through a Program, p. 244.)

The editor, however, works only on source code. Thus, when you display a view with disassembled instructions, the editor display goes into read-only mode until you either stop debugging or switch to the Source view.
8.5 Using the Debugger Command Line

The Tornado graphical interface is usually the most convenient way to run the debugger. However, you can also use the GDB command-line interface, which in some cases is the best way to perform a particular action (see Figure 8-1). The debugger command-line window provides full access to GDB commands, as well as to Tornado extensions to those commands. The command line can be displayed with Debug > Debug Windows > Debug Command Line.

NOTE: The command line drop-down list displays a history of all commands the user has entered. Commands can be selected from the list to be executed again.

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.

8.5.1 GDB Initialization Files

One use of the debugger window is to experiment with text-based commands for actions that you might want to perform automatically each time you start debugging.

When the debugger first executes GDB, it looks for a file named .gdbinit. It first looks in the directory named by the environment variable HOME (if it is defined), then in your current working directory.1 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.

1. The graphical interface to the debugger has a separate initialization file crosswind.tcl, which runs after .gdbinit.

NOTE: Disassembly takes a long time the first time you switch to a view that requires it. Subsequently, in the same debugging session, you can switch views quickly. The disassembly information is not persistent; the debugger discards it when you stop debugging (or if you close the source file with the Close command in the File menu).
A related initialization file, called `gdb.tcl`, is specifically intended for Tcl code to customize GDB with your own extensions. The Tcl code in this file executes before `.gdbinit`. The debugger searches for `gdb.tcl` in two places: first in `c:\tornado\wind`, then (if the environment variable `HOME` is defined) in `%HOME%\wind`. See 8.7 Tcl: Debugger Automation, p.267 for a discussion of extending GDB through Tcl. See also 8.8.1 Tcl: Debugger Initialization Files, p.275 for a discussion of how the Tornado debugger initialization files interact.

### 8.5.2 Selecting 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 often make it necessary to specify the object file explicitly. These factors include differing mount points on host and target, symbolic links, virtual file systems, or simply moving a file after downloading it.

The debugger recognizes the abbreviation `add` for this command.

`load filename`

Downloads an object file to the target.

This command is equivalent to the `Download` command in the Debug menu. You may sometimes find it preferable to invoke the command from the command panel—for example, when you can paste a complex path name from the clipboard instead of spending time in a file-browser interaction.

`load filename serverFilename`

Reads debugging information from `filename` while downloading `serverFilename` to the target.

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 debugging session. For example, in some complex networks different hosts may mount the same file at different points: you may want to download a file `c:\fred\applic.o` which your target server on another host sees as `\fredhost\fred\applic.o`.2

2
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 6.6 Object Module Load Path, p.196 for a more extended discussion of the same problem in the context of the shell.

`unload filename`

Undoes the effect of `load`: removes a dynamically linked file from the target, deletes its symbols from the debugger, and closes the file.

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.

### 8.5.3 Running a 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 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 6.2.6 Using the Shell for System Mode Debugging, p.169.)

**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

2. See also the description of `wtx-load-path-qualify` in 8.5.8 Extended Debugger Variables, p.261 for another way of managing how the debugger reports `load` path names to the target server.
interest before starting a routine. However, you can interrupt the running task by clicking Interrupt Debugger in the Debug menu or by pressing CTRL+BREAK from the Debugger window.

8.5.4 Re-Setting 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 output, ordinary path names refer to files on the host where the debugger is running, and path names preceded by an @ character refer to files or devices on the target. Input cannot be redirected to host files, but input redirection to target files or devices is supported with the same @ convention for consistency. 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 file grab.it:

  (gdb) run appMain > grab.it < @/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 convention used with explicit redirection on the run line allow you to specify target files or devices for I/O. For example, the following command sets the default input and output channels to target device /tyCo/0:

  (gdb) tty @/tyCo/0

• **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 8.7.3 Tcl: Invoking GDB Facilities, p.270 for details.
8.5.5 Using Graphically Enhanced Commands

The GDB frame command has a Tornado graphical extension, even when used at the command line.

frame n
Displays a summary of a stack frame, in the Debugger window. But it also has a useful side effect: it re-displays the code in the editor window, centered around the line of code corresponding to that stack frame.

Used without any arguments, this command provides a quick way of restoring the editor-window 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*.

8.5.6 Managing Targets

Instead of using the target server list in the Tornado Launch toolbar, you can select a target from the Debugger window 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. To identify the target server, any unique abbreviation will do as servername; there is no need to specify the full name known to the Tornado registry.

8.5.7 Extended Debugger Commands

The debugger also provides two kinds of extended commands:

- 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.

- **Shell Commands**

  You can run any of the Tornado shell’s primitive facilities described in 6.2.3 Invoking Built-In Shell Routines, p. 156 in the Debugger window, by inserting the prefix “wind-” before the shell command name. For example, to run the shell \( \text{td}() \) command from the debugger, enter wind-td in the Debugger window.\(^3\) 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 upper-case letter. For example, to run the \( \text{semShow}() \) command, enter wind-sem-show.

  **CAUTION:** The debugger does not include the Tornado 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 \( \text{shSymAddr} \) to translate C symbols to the Tcl form. For example, to call the shell built-in \( \text{show}() \) on a semaphore ID \( \text{mySemID} \), use the following:

  \[(\text{gdb}) \quad \text{wind-show [shSymAddr mySemId]}\]

### 8.5.8 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 the Tornado debugger (CrossWind) in addition to those described in *GDB User’s Guide*:

- **inhibit-gdbinit**
  
  Specifies whether or not to read the GDB-language initialization files (discussed in 8.8.1 Tcl: Debugger Initialization Files, p. 275). Default: `no` (that is, read initialization files).

---

3. You can exploit command completion to see a list of all the shell primitives as they appear in the debugger: type “wind-” followed by two tabs.
wtx-ignore-exit-status
 Specifies whether or not to report the explicit exit status of a routine under debugger control. When this parameter is on, 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. Default: on.

wtx-load-flags
 Specifies the option flags for the dynamic loader (Download in the Debug menu, or load in the Debugger window). 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
 Specifies 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. Setting this value to yes 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 path name can be interpreted correctly by both programs even though the absolute path name is different for the two. In this case, set wtx-load-path-qualify to no.

Default: yes.

wtx-load-timeout
 Specifies how long to wait for the target to respond during a download. 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
 Specifies the priority for transient VxWorks tasks spawned by the run command. Default: 100.

wtx-task-stack-size
 Specifies the stack size for transient tasks spawned by the run command. Default: 20,000.

wtx-tool-name
 Specifies 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. If you often run multiple debugger sessions, you can use this parameter to give each session a distinct name. Default: crosswind.
8.6 System-Mode Debugging

By default, in Tornado you debug only one task at a time. The task is selected either by using the run dialog box to create a new task, or by using the Attach dialog box (8.3.8 Attaching to a Running Task, p.246) to debug an existing task. When the debugger is attached to a task, debugger commands affect only that particular task. For example, breakpoints apply 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 8.6.2 Thread Facilities in System Mode, p.264) are useful only in system mode.

⚠️ CAUTION: 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 6.2.6 Using the Shell for System Mode Debugging, p.169, then attach to them with the thread command.

8.6.1 Entering System Mode

To debug in system mode, click on the System entry displayed in the Attach dialog box (use the Attach command in the Debug menu to display the dialog box).

You can also switch to system mode from the debugger window or an initialization file, but first make sure your debugger session is not attached to any task (type detach). 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.

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 \texttt{wdbSuspendSystemHere()}, the normal suspension point after \texttt{attach system}.

\begin{verbatim}
    WARNING: Not all targets support system mode, because the BSP must include a special driver for that purpose (see 4.6 Configuring the Target-Host Communication Interface, p.116). If your target does not support system mode, attempting to use \texttt{attach system} produces an error.
\end{verbatim}

### 8.6.2 Thread Facilities in System Mode

In system mode, the GDB thread-debugging facilities become useful. \textit{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 \textit{thread ID}, a single arbitrary number internal to the debugger.

You can use the following GDB commands to manage threads:

- \texttt{info threads}  
  Displays summary information (including thread ID) for every thread in the target system.
- \texttt{thread idNo}  
  Selects the specified thread as the current thread.
- \texttt{break linespec thread idNo}  
  Sets a breakpoint affecting only the specified thread.

For a general description of these commands, see \textit{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 \texttt{info threads} shows what thread ID corresponds to which VxWorks task. For example, immediately after attaching to the system, the \texttt{info threads} display resembles the following:

\begin{verbatim}
(gdb) info threads
  4 task 0x4fc868  tExcTask   0x444f58 in ?? ()
  3 task 0x4f9f40  tLogTask   0x444f58 in ?? ()
\end{verbatim}
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 debugger 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, its current ID.

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 routine 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 0x49fed in shell ()
(gdb) thread 3
```
As in the display shown above, each time you switch threads the debugger exhibits the newly current thread’s task ID and task name.

**Thread-Specific Breakpoints**

In system mode, unqualified breakpoints (set with graphical controls on the editor window, or in the Debugger window 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.
```

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.
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 with “Switching to” shows which thread is 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 Interrupt Debugger in the Debug menu, or by keying CTRL+BREAK. 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, which is in the thread marked + in the info threads display. However, in the course of stepping that thread, other threads may begin executing. The debugger may stop in another thread before the stepping command completes, due to an event in that other thread.

8.7 Tcl: Debugger Automation

The debugger 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
Tornado 2.0
User's Guide

level. This section discusses using the Tcl interpreter inside the Tornado enhanced GDB, at the command level.

NOTE: For information about using Tcl to customize the Tornado GUI, see 8.8 Tcl: Debugger Customization, p.274. This section is mainly of interest when you need complex debugger macros; thus, you might want to skip it 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.

8.7.1 Tcl: A Simple Debugger Example

To submit commands to the Tcl interpreter within GDB from the Tornado Debugger window, 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 8.7.3 Tcl: Invoking GDB Facilities, p.270.

You can also define Tcl procedures from the GDB command line. The following example procedure, mload, 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 `gdb.tcl`. The GDB Tcl interpreter reads this file when it begins executing. (See 8.8.1 Tcl: Debugger Initialization Files, p.275 for a discussion of where you can put this file, and of how all the Tornado debugger and GDB initialization files interact.)

### 8.7.2 Tcl: Specialized GDB Commands

The Tornado debugger (CrossWind) 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` for a Tcl procedure named `TclName`. GDB does not evaluate the arguments when `gdbName` is invoked; it passes them to the 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 8.5.1 GDB Initialization Files, p.256), 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 8.7.3 Tcl: Invoking GDB Facilities, p.270 to call `gdb tclproc` in `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:
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.334.

### 8.7.3 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 Debugger window.

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.334).

**gdbEvalScalar exprlist**

Evaluates a list of expressions `exprlist` and returns a list of single integer values (in hexadecimal), one for each element of `exprlist`. 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.334 for information about unwinding).

`gdbFileAddrInfo fileName`

Returns a Tcl list with four elements: the first source line number of `fileName` that corresponds to generated object code, the last such line number, the lowest object-code address from `fileName` in the target, and the highest object-code address from `fileName` in the target. The argument `fileName` must be the source file (.c, not .o) corresponding to code loaded in the target and in the debugger.

For example:

```
(gdb) tcl gdbFileAddrInfo vxColor.c
{239 1058 0x39e2d0 0x39fbfc}
```

`gdbFileLineInfo fileName`

Returns a Tcl list with as many elements as there are source lines of `fileName` that correspond to generated object code. Each element of the list is itself a list with three elements: the source-file line number, the beginning address of object code for that line, and the ending address of object code for that line. The argument `fileName` must be the source file (.c, not .o) of a file corresponding to code loaded in the target and in the debugger.

For example:

```
(gdb) tcl gdbFileLineInfo vxColor.c
{239 0x39e2d0 0x39e2d4} {244 0x39e2d4 0x39e2ec} ...
```

`gdbIORedirect inFile outFile [taskId]`

Redirects target input to file or device `inFile`, and target output and error streams to file or device `outFile`. If `taskId` is specified, redirect input and output only for that task; otherwise, redirect global input and output.

4. 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.
input or output unchanged, specify the corresponding argument as a dash (-).
Input may only be redirected to target files or devices; output may be
redirected either to host files or to target files or devices. Ordinary path names
indicate host files; 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 file grab.it:

```
(gdb) tcl gdbIORedirect - grab.it
```

**gdbIOClose**
Closes 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
```
8.7.4 Tcl: A Linked-List Traversal Macro Example

This section shows a Tcl procedure to traverse a linked list, printing information about each node. The example is tailored to a list where each node has the following structure:

```c
struct node
{
    int data;
    struct node *next;
}
```

A common method of list traversal in C is a `for` loop like the following:

```c
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:

```tcl
set pNode [gdbEvalScalar "$head"]
```

To get the pointer to the next element in the list:

```tcl
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:

```tcl
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 Debugger window would require prefacing the text above with the `tcl` command, and would require a backslash at the end of every line.

5. Keep in mind that for interactive exploration of a list the window (Figure 8-14) described in Watch Window, p.248 is probably more convenient.
If \texttt{pList} is a variable of type (\texttt{struct *node}), you can execute:

\begin{verbatim}
(gdb) tcl traverse pList
\end{verbatim}

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:

\begin{verbatim}
0xffeb00
0xffea2c
\end{verbatim}

It might be more useful to redefine the procedure body to print out the integer member \texttt{data}, instead. For example, replace the last line with the following:

\begin{verbatim}
{puts stdout [format "data = %d" \\
[gdbEvalScalar "((struct node *) $pNode)->data"]]
\end{verbatim}

You can bind a new GDB command to this Tcl procedure by using \texttt{tclproc} (typically, in the same Tcl script file as the procedure definition):

\begin{verbatim}
tclproc traverse traverse
\end{verbatim}

The \texttt{traverse} command can be abbreviated, like any GDB command. With these definitions, you can type the following command:

\begin{verbatim}
(gdb) trav pList
\end{verbatim}

The output now exhibits the contents of each node in the list:

\begin{verbatim}
data = 1
data = 2
\end{verbatim}

### 8.8 Tcl: Debugger Customization

Like every other Tornado tool, the debugger’s graphical user interface is “soft” (amenable to customizing) because it is written in Tcl, which is an interpreted language. The \textit{Tornado API Reference: GUI Tcl Library} describes the graphical building blocks available. You can also study the Tcl implementation of the debugger graphical interface itself, in \texttt{host\resource\tcl\CrossWind.win32.tcl}. 


8.8.1 Tcl: Debugger Initialization Files

You can write Tcl code to customize the debugger’s graphical presentation in a file called `crosswind.tcl`. The debugger looks for this file in two places: first under `c:\tornado` (or whatever directory you specify with the environment variable `WIND_BASE`), and then in the directory specified by the `HOME` environment variable (if that environment variable is defined). Use this file to collect your custom modifications, or to incorporate shared modifications 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 `gdb.tcl` initialization files for the debugger command language (see 8.7 Tcl: Debugger Automation, p.267). The following outline summarizes the role of all the Tornado debugger customization files. The files are listed in the order in which they execute:

`%WIND_BASE%\.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 Tornado debugger (that is, it is not shared by any other GDB configuration you might install).

The version of this file under `%WIND_BASE%` (for example, in `c:\tornado\.wind`) is shared by everyone who uses Tornado on the same PC.

`%HOME%\.wind\gdb.tcl`

If more than one developer uses the same PC as a Tornado host, you can use this file just like the shared version of `gdb.tcl` described above, but for user-specific customizations; each user can specify a different directory in the `HOME` environment variable.

`%HOME%\.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 the Tornado debugger; it is shared by any other GDB configuration you may install.

`.gdbinit`

Akin to the `.gdbinit` in your home directory, the version of this file in the current working directory also contains commands in GDB’s command language, and is not unique to the Tornado 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.
Tornado 2.0
User's Guide

%WIND_BASE%\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 Tornado debugger (that is, it is not shared by any other GDB configuration you might install).

The version of this file under %WIND_BASE% is shared by everyone who uses Tornado on the same PC.

%HOME%\wind\crosswind.tcl
If other developers use the same PC as a Tornado host, you can use this file in the same way as the shared version of crosswind.tcl described above, but for user-specific customizations; each user can specify a different directory in the HOME environment variable.

You can prevent the Tornado debugger 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 in gdb.tcl:

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

### 8.8.2 Tcl: Passing Control between the Two Tornado Debugger Interpreters

You can use the following specialized Tcl commands to pass control between the two Tornado debugger (CrossWind) Tcl interpreters:

**uptcl**
From the Tcl interpreter integrated with the GDB command parser, `uptcl` executes the remainder of the line in the Tornado debugger 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 `writeDebugInput` if your goal is to capture the result for presentation in the graphical layer.

**writeDebugInput**
From the graphical-interface layer, `writeDebugInput` 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 `writeDebugInput` 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 Debugger window:

```
(gdb) tcl uptcl source myXWind
```

By contrast, `downtcl` and `writeDebugInput` are likely to be embedded in Tcl procedures, because (in conjunction with the commands discussed in 8.7.3 `Tcl: Invoking GDB Facilities`, p.270) they are the path to debugger functionality from the graphical front end.

Most of the examples in 8.8.3 `Tcl: Experimenting with Tornado Debugger Extensions`, p.277, below, center around calls to `downtcl`.

### 8.8.3 Tcl: Experimenting with Tornado Debugger Extensions

The examples in this section use the Tcl extensions summarized in Table 8-2. For detailed descriptions of these and other Tornado graphical building blocks in Tcl, see Tornado API Guide: UI Tcl or Tornado Online Manuals: Tornado API.

<table>
<thead>
<tr>
<th>Tcl Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>menuItemAppend</td>
<td>Add a command or a separator to the end of an existing menu.</td>
</tr>
<tr>
<td>menuItemInsert</td>
<td>Add a command within the list of existing commands in a menu.</td>
</tr>
</tbody>
</table>

### Tcl: An Add-Symbols Command for the File Menu

As explained in 8.5.2 `Selecting Modules to Debug`, p.257, 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 8-1 illustrates a Debug menu command Add Symbols to handle this through the graphical user interface, instead of typing the `add-symbol-file` command.
Example 8-1  Add-Symbols Command

# MENU COMMAND:  "Add Symbols", additional entry under "Debug"

menuItemInsert -after {"&Debug" "Do&wnload...|Shift+F6"} \  
{"Add Symbols..."} {windAddSyms}

#############################################################################
#
# windAddSyms - called from Debug menu to add symbols from chosen object file
# This routine implements the "Add Symbols" command in the Debug menu.
# It prompts the user for a filename; if the user selects one, it tells
# GDB to load symbols from that file.
# SYNOPSIS:
# proc windAddSyms
#
# RETURNS: N/A
#
# ERRORS: N/A
#proc windAddSyms {} {
  set result [fileDialogCreate -title "Symbols from file" -filemustexist \
    -filefilters "Object Files (*.o;*.out)|*.o;*.out|\n          All Files (*.*)|*.*||" -    
    -okbuttontitle "&Add"]

  if {$result != ""} {
    # we violate good taste here by not capturing or testing the result
    # of catch, because an error notification appears in the command
    # panel.
    catch {downtcl gdb add-symbol-file [lindex $result 0]}
  }
}

Tcl: “this” Menu Command for C++

In C++ programs, one particular named value has special interest: this, which is a
pointer to the object where the currently executing routine is a member.

Example 8-2 defines a Debug menu command to launch an inspection window for
the value where this points. The Tcl primitive catch is used in the 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 Tornado debugger session.
Example 8-2  Command to Display C++ this Value

# Insert a separator line after default menu commands
menuItemAppend -separator {&Debug}

# Debug Menu Command: *this - Inspect window on current C++ class (*this)
menuItemAppend {"&Debug"} {"*this"} {
  catch (downtcl gdb display/W *this)
}
9 Customization

9.1 Introduction

Tornado not only allows you to customize the appearance of the display to match your preferences, but it also allows you to add menu entries for other tools you may wish to use. The Options entry in the Tools menu displays commands that change the fonts, colors, editor settings, and other defaults for Tornado. The Customize entry in the Tools menu opens a dialog box for adding menu items.

9.2 Setting Options

You can specify the various default options that Tornado uses through the Options entry in the Tornado Tools menu.

9.2.1 Setting Download Options

The Download page provides options for handling symbols when objects are downloaded to the target (Figure 9-1).

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.
- 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.
9.2.2 Editor Preferences

Select Options in the Tools menu, then click the Editor tab to adapt the editor to your preferences. The Editor page is shown in Figure 9-2.

Figure 9-2 Editor Page

The following choices are available on the Editor page:

Window Settings

The Horizontal and Vertical Scrollbar check boxes control which scroll bars appear in the editor window. If no scroll bars appear, you can only scroll by using the keyboard arrow keys to move past the displayed area (or by typing past the displayed area). Defaults: Vertical on, Horizontal off.

When Attribute Pane is checked (the default), Tornado reserves space in the left margin of source-code editor windows and displays state information there: breakpoint locations, the currently executing line, the current error, and the like. When Attribute Pane is not checked, Tornado conveys this information instead by highlighting entire lines in the editor window.
Tab Size
The editor uses regularly spaced tab stops; this field controls how far apart the tab stops are. If you use tabs for code indentation, smaller values in this field are useful in small windows. Default: 8.

Maximum Undo Levels
The editor keeps track of your changes to the source file, and uses this information to allow you to reverse those changes (through the Undo command in the Edit menu). This box specifies how many changes the editor keeps track of, up to a maximum of 512 changes. Default: 512.

Save Options
This panel contains two check boxes for different purposes. Both check boxes are on by default.

When Save before running Tools/Builds is checked, Tornado saves all modified text from the current editor windows before executing any command from the Tools or Project menus.

When Automatic reload of externally modified files is checked, Tornado automatically reloads any file that is modified by an external editor. This keeps the Tornado tools synchronized with an external editor.

File Format
The radio buttons in this panel can be used to determine the end-of-line format for the files you edit. The Tornado editor always saves files with a consistent end-of-line format, converting any inconsistent formats, if necessary. The DOS end of line format is a combination of carriage return and line feed. The UNIX end-of-line format is a line feed. The Automatic option preserves whatever format is present in existing files when they are opened, and uses DOS format for new files.

9.2.3 External Editor
If you are accustomed to another editor, you may want to use Tornado only as a viewer and to provide debugging context, and pass control to your preferred editor when you want to make changes to a file. Select Options in the Tools menu, then click External Editor to specify an editor other than the Tornado editor. The External Editor page is shown in Figure 9-3.

The following choices are available on the External Editor page:
Settings

The Command line text box allows you to enter the command that invokes your preferred editor. Click the button at the right of the box to see menu including a browse option and macros which allow you to capture Tornado context in your commands (File name and Line number). See Macros for Customized Menu Commands, p.296 for explanations of these macros.

The Defaults list box allows you to select from several available alternative editors. When you make a selection, the appropriate command is automatically entered in the Command line text box.

Invoke from

The Invoke from section of the page allows you to choose where your external editor will be invoked. For example, if you check File menu but not Project, selecting Open from the File menu will open the file in your external editor, but double-clicking on a file in the Project window will open the file in the Tornado editor.

9.2.4 Project

Select Options in the Tools menu, then click Projects to specify certain project attributes. The Projects page is shown in Figure 9-4.

The following choices are available on the Projects page:
Tornado 1.0.1 Compatibility

Checking the Show Tornado 1.0.1 menu items box causes a Customize item to be added to the Build menu. This allows you to perform Tornado 1.0.1-style builds in BSP directories, to use existing Tornado 1.0.1-style Build menu items, and to create additional Build menu customizations. Before you add any commands to the menu, Tornado displays the place-holder No Custom Builds as a disabled menu entry.

Selecting Customize from the Build menu opens the Customize builds dialog box. This is similar to the Customize Tools dialog box; it has the same buttons, text boxes, and macro menus available. For more information, see 9.3.1 The Customize Tools Dialog Box, p.294.

Component Properties

Checking the Show advanced component properties box adds the Definition tab to the component property window. The Definition page shows the internal schema and attributes for the component. This may be helpful for authoring or debugging components.

For example, using Tornado 1.0.1-style Build menu customizations, you can add a command that compiles the default make target in the same directory as the file currently open or selected in the Project tool. Use the $filepath macro in Working Directory and leaving Build Target blank in the Customize Builds dialog box.
9.2.5 Version Control

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. The Version Control page is shown in Figure 9-6.
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 9-1).

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 9-1).

Table 9-1  
Macros for Version Control

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>File name</td>
<td>$filename</td>
<td>Name of the active file, without path information.</td>
<td>zap.c</td>
</tr>
<tr>
<td>Comment</td>
<td>$comment</td>
<td>Prompt for a checkin or checkout comment; any parameter needed by the tool can also be entered.</td>
<td>See Figure 9-7.</td>
</tr>
</tbody>
</table>

Figure 9-7  
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.

Advanced
Clicking the Advanced button opens the Customize Tools dialog box, which can be used to put commands on the Tools menu. Anything that can not be done on the Version Control page can be done on the Customize Tools dialog box.
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.

9.2.6 Fonts/Colors

Select Options in the Tools menu, then click Fonts/Colors to change the fonts and colors that Tornado uses for all text windows: the editor window, the shell, and the debugger command window. The Fonts/Colors page is shown in Figure 9-8.

The Font selection boxes allow you to select a typeface, weight, and size in points. Different fonts are appropriate for different working environments: for example, usually a smaller point size is more desirable on a lap-top than on a large desktop display. The Sample box shows you what your selections will look like.

If you check the Syntax Coloring box, Tornado will identify various distinguished window elements by color, including syntactically distinguished text and the attribute-panel markers that appear during debugging.

You can choose which window element to color by clicking an entry in the Items list box. Table 9-2 describes the window elements available for coloring. The Foreground Color and Background Color panels allow you to control the foreground
and background independently for each window element; click on the desired color in either list to assign that color to the currently selected window element.

Table 9-2  Window Elements for Coloring

<table>
<thead>
<tr>
<th>Color Preferences Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Text (Default)</td>
<td>Editor-window color for text not otherwise highlighted.</td>
</tr>
<tr>
<td>Breakpoint Line</td>
<td>Breakpoint-symbol (or line, if no attribute pane) highlighting.</td>
</tr>
<tr>
<td>Comment</td>
<td>Editor window color for C or C++ comments.</td>
</tr>
<tr>
<td>Disassembly Line</td>
<td>Disassembly lines in editor window; entire disassembly window.</td>
</tr>
<tr>
<td>C Keyword</td>
<td>Editor-window color for C-language keywords.</td>
</tr>
<tr>
<td>C++ Keyword</td>
<td>Editor-window color for C++ keywords that are not C keywords.</td>
</tr>
<tr>
<td>Number</td>
<td>Integer literals in the editor window.</td>
</tr>
<tr>
<td>Floating Point</td>
<td>Floating-point numeric literals in the editor window.</td>
</tr>
<tr>
<td>String</td>
<td>String literals in the editor window.</td>
</tr>
<tr>
<td>Identifier</td>
<td>Symbolic names in source code (editor window).</td>
</tr>
<tr>
<td>Inspect Window</td>
<td>Window displayed by Inspect in Debug menu.</td>
</tr>
<tr>
<td>Build Output</td>
<td>Window for output from Project menu commands.</td>
</tr>
<tr>
<td>Registers Window</td>
<td>Window displayed by Registers in Debug menu.</td>
</tr>
<tr>
<td>Locals Window</td>
<td>Window displayed by Locals in Debug menu.</td>
</tr>
<tr>
<td>Text Selection</td>
<td>Current selection in any Tornado window.</td>
</tr>
<tr>
<td>Current Line</td>
<td>Context pointer (or line, if no attribute pane) highlighting.</td>
</tr>
<tr>
<td>Memory Window</td>
<td>Window displayed by Memory in Debug menu.</td>
</tr>
<tr>
<td>Backtrace Window</td>
<td>Window displayed by Back Trace in Debug menu.</td>
</tr>
<tr>
<td>Debugger</td>
<td>Debugger command window.</td>
</tr>
<tr>
<td>Shell</td>
<td>Tornado shell (WindSh) window.</td>
</tr>
<tr>
<td>Previous Context</td>
<td>Context pointer, while viewing non-current stack level.</td>
</tr>
</tbody>
</table>
Table 9-2  **Window Elements for Coloring**  *(Continued)*

<table>
<thead>
<tr>
<th>Color Preferences Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers Highlight</td>
<td>Highlighting for updates to Registers window.</td>
</tr>
<tr>
<td>Locals Highlight</td>
<td>Highlighting for updates to Locals window.</td>
</tr>
<tr>
<td>Inspect Highlight</td>
<td>Highlighting for updates to Inspect window.</td>
</tr>
<tr>
<td>Backtrace Highlight</td>
<td>Highlighting for updates to Backtrace window.</td>
</tr>
<tr>
<td>Current Error/Tag</td>
<td>Selected error line in editor, when navigating from build output.</td>
</tr>
<tr>
<td>Component Error</td>
<td>Highlighting for components and their parents that are in error in the VxWorks view of the Workspace window.</td>
</tr>
<tr>
<td>Component Highlight</td>
<td>Highlighting for components and their parents that are selected in the VxWorks view of the Workspace window during Autoscale or Component Find processes.</td>
</tr>
<tr>
<td>Component Normal</td>
<td>Text color for normal components in the VxWorks view of the Workspace window (neither errors nor selected).</td>
</tr>
</tbody>
</table>

You can always return to the original settings by clicking the following buttons:

**Restore to Default**
- Reset the currently selected window element to its default color.

**Restore All Defaults**
- Reset all window elements to their default colors.

### 9.2.7 Debugger

Select Options in the Tools menu, then click Debugger to customize various debugger attributes. The Debugger page is shown in Figure 9-9.

The following choices are available on the Debugger page:

**General**
- Checking the Hexadecimal display box forces all debugger window displays to hexadecimal. The default is to display the format chosen by gdb for each window.
- Checking the Docking views box causes all debuggers windows to dock. The default is floating windows.
Memory Window

Use the Format and Size drop-down combo boxes to select how the memory window is displayed.

Checking the Always refresh on debugger stop box causes the memory window to update whenever execution stops. This is the default. For performance reasons you may choose to uncheck the box and only update it on demand.

Auto attach to tasks

Allows you to determine the degree to which the debugger automatically attaches to a task that throws an exception. Depending on which option you select, the behavior is as follows:

- Never: the debugger never automatically attaches to the task throwing the exception.
- Only if not already attached: The debugger attaches to the task if you are not already attached to another task.
- Always: the debugger always automatically attaches to the task.

9.2.8 Tornado Registry

Select Options in the Tools menu, then click Tornado Registry to change the Tornado registry. The Tornado Registry page is shown in Figure 9-10.
A local registry resides on your local host. It lists any target servers you have running on your host. If others are using your machine as a remote registry, their target servers will also appear on the list; otherwise only your local target servers are known to Tornado. If you choose a local registry, enter the path in the box or use the browse button to select a path.

A remote registry resides on another host. All your target servers and any other networked target servers known to that registry are listed. You can specify either the host name or the IP address of the remote host.

### 9.3 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.
9.3.1 The Customize Tools Dialog Box

Click Customize in the Tools menu to open the Customize Tools dialog box (Figure 9-11).

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. 296 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 9-12.)
- 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 9-3 lists macros for editor context; in this table, the phrase active file refers to the file that is open in the active editor window (or selected in the project facility if no editor window is open).

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>File path</td>
<td>$filepath</td>
<td>Full path to the active file.</td>
<td>c:\tornado\zap.c</td>
</tr>
<tr>
<td>Dir name</td>
<td>$filedir</td>
<td>Directory containing the active file.</td>
<td>c:\tornado</td>
</tr>
<tr>
<td>File name</td>
<td>$filename</td>
<td>Name of the active file, without path information.</td>
<td>zap.c</td>
</tr>
<tr>
<td>Base name</td>
<td>$basename</td>
<td>Name of the active file, without the file extension.</td>
<td>zap</td>
</tr>
<tr>
<td>Line number</td>
<td>$line</td>
<td>The line number where the cursor is positioned in the active file.</td>
<td>15</td>
</tr>
<tr>
<td>Column number</td>
<td>$column</td>
<td>The column number where the cursor is positioned in the active file.</td>
<td>25</td>
</tr>
<tr>
<td>Selected text</td>
<td>$textsel</td>
<td>The current selection (highlighted text) in the active file.</td>
<td>hitCount(void)</td>
</tr>
</tbody>
</table>
Table 9-4 lists macros for the project facility context.

### Table 9-4  Menu-Customization Macros for Project Context

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project dir</td>
<td>$projdir</td>
<td>The name of the directory of the current project.</td>
<td>c:\proj\widget</td>
</tr>
<tr>
<td>Project build dir</td>
<td>$builddir</td>
<td>The name of the directory for the current build of the current project.</td>
<td>c:\proj\widget\default</td>
</tr>
<tr>
<td>Derived object</td>
<td>$derivedobj</td>
<td>The name of the derived object of the currently selected source file in the current project.</td>
<td>c:\proj\widget\default\zap.o</td>
</tr>
</tbody>
</table>

Table 9-5 lists macros for the global Tornado context.

### Table 9-5  Menu-Customization Macros for Global Context

<table>
<thead>
<tr>
<th>Menu Entry</th>
<th>Macro</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target name</td>
<td>$targetName</td>
<td>The full name of the target server selected in the Tornado Launch toolbar.</td>
<td>vxsim@ontario</td>
</tr>
<tr>
<td>Target</td>
<td>$target</td>
<td>The name of the target selected in the Tornado Launch toolbar.</td>
<td>vxsim</td>
</tr>
<tr>
<td>Tornado inst. dir</td>
<td>$wind_base</td>
<td>Installation directory recorded in the environment variable WIND_BASE.</td>
<td>c:\tornado</td>
</tr>
<tr>
<td>Tornado host type</td>
<td>$wind_host_type</td>
<td>Host type recorded in the environment variable WIND_HOST_TYPE.</td>
<td>x86-win32</td>
</tr>
<tr>
<td>Tornado registry</td>
<td>$wind_registry</td>
<td>Registry recorded in the environment variable WIND_REGISTRY</td>
<td>mars</td>
</tr>
</tbody>
</table>

These macros are available for custom entries in the Build menu as well as for the Tools menu. For information on making them available through the Build menu, see 9.2.4 Project, p. 285. For information on using them for custom builds, see 9.3.3 Customizing the Build Menu, p. 301.
9.3.2 Examples of Tools Menu Customization

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 9-13 illustrates the specification for a ClearCase command to uncheckout a module.

Figure 9-13 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. 296) to expand the current file to its full path name.

In this example, the Prompt for Arguments box is 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 9-14), to permit specifying other arguments if necessary.
Figure 9-14  **Prompt for Arguments Dialog Box**

![Prompt for Arguments Dialog Box](image)

**Alternate Editor**

Figure 9-15 illustrates the specification for a command to run the Windows Notepad 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.

Figure 9-15  **Custom Editor Command for Tools Menu**

![Custom Editor Command for Tools Menu](image)

Normally you would change the default editor to an external editor using the External Editor tab under Tools>Options. In the case where you only wish to use an alternate editor some of the time, you may prefer this method.
**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 9-16 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 case, the Tool Command field uses `$filedir/SIMNTgnu/$basename` to construct the path and name of the object file generated from the current source file. The Working Directory field is filled in using the browse option to locate the appropriate version of `size arch` in the correct directory.

![Figure 9-16 Object-Module Size Command for Tools Menu](image)

**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 9-17 shows the specification for a Wind River Web Page command. (For a description of the information available on the WRS home page, see 10.3 WRS Broadcasts on the World Wide Web, p.312.

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.

300
9.3.3 Customizing the Build Menu

If you are using Tornado 1.0.1 build techniques, you can add commands to the Build menu to build any object that can be generated using GNU make. Typically this requires first generating (or writing) a makefile that specifies the rules to build that object. It also requires checking Show Tornado 1.0.1 menu items on the Project tab of the Tools>Options menu (see 9.2.4 Project, p.285).

Once a makefile is in place, click Customize in the Build menu to add your own commands to the bottom part of that menu (or to remove commands you no longer need). The Customize Builds dialog box appears (Figure 9-18). This dialog box shows all custom entries already present in the Project menu (if any), and allows you to add commands to, remove commands from, or reorder the list of custom Project commands.

The Customize Builds dialog box has the following buttons:

Add
Activates the list and check boxes at the bottom of the Customize builds dialog box and enters New Build in the Menu Text list box. Replace New Build with the command description; when you click OK, the new menu item appears in the Build menu.

Remove
Deletes the selected menu item from the Build menu.
Move Up
Moves the selected menu item up one line in the Menu Contents list box and changes the displayed order on the Build menu.

Move Down
Moves the selected menu item down one line in the Menu Contents list box and changes the displayed order on the Build menu.

OK
Applies your changes to the Build menu.

Cancel
Discards your changes without modifying the Build menu.

The Menu Contents list box in Customize Builds shows all custom commands currently in the Build 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:

Menu Text
Contains the name of the custom command, as it appears in the Build menu.

Build Target
You can specify any information here that can be placed in the make command line. You must specify at least the name of the object to build, but frequently it is also useful to include additional parameters in some of the standard makefile variables (see VxWorks Programmer’s Guide: Configuration and Build).

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. 296 for explanations of these macros.

**Working Directory**
Use this field to specify the directory containing the makefile that defines the rules for building this target. (Usually, this directory is also where the source code for the object resides, and where the new object is generated.) 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 Build Target 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.

You can also use the browse feature to select makefiles with names other than the names recognized by default, `Makefile` or `makefile`; Tornado inserts the `make -f` option automatically when you select a makefile with another name). If you use the Browse button without filling out the Build Target field, the new command is labeled `Makefile`, and it builds the default target in the makefile you selected.

### 9.4 Tcl Customization Files

When Tornado begins executing, it checks for initialization files of the form `.wind\filename.tcl` in two places: first under `c:\tornado` (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:

```tcl
set htmlBrowser "newbrowser -install"
```
10.1 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:

- **Tornado**
  
The Tornado Tools menu provides a structured online system for requesting help with any WRS product. For more information, see 10.2 WRS Support Services, p.306.

- **support@wrs.com**
  
You can use this e-mail address directly to contact customer support if you prefer, although using the Tools menu is generally 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.

- **1–800–872–4977 (1–800–USA–4WRS)**
  
Within 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 send the structured information from the Tools menu over FAX instead of e-mail; see 10.2 WRS Support Services, p.306.
The remainder of this chapter describes the support facilities in Tornado. These facilities are helpful, but are not required: use the contact method you find most convenient.

NOTE: The standard Wind River Systems support contract specifies that there must be assigned contact people at your organization. Please channel your support requests through them.

10.2 WRS Support Services

The Tools>Support menu option displays a form where you can enter support requests.

10.2.1 The User Profile

The first time you select the Support command from the Tools menu, an auxiliary dialog box is displayed: the User Profile dialog box. You must enter this information once before sending your first Tornado Support Request (TSR). Thereafter, the same information is used automatically for all future TSRs you transmit. You can also get to this auxiliary dialog box by pressing the Profile button on a Tornado Support Request form, if you need to revise it later. A completed User Profile dialog box is shown in Figure 10-1 below.

The following fields appear in the User Profile dialog box:

User
   Your name; filled in by default from your Tornado registration.

Company
   Your company’s name; filled in by default from your registration.

Contact
   The person to receive replies to your support requests. Usually this is also your name; filled in by default from your registration.

Address 1
   The first line of a two-line mailing address.

1. This information is stored in the file c:\tornado\wind\profile after you enter it.
Address 2
The second line of a two-line mailing address.

Phone
A telephone number where support personnel can contact you.

E-mail
The electronic mail address, if available, to receive replies to your support requests.

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

NOTE: The standard Wind River Systems support contract specifies that there must be assigned contact people at your organization. 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.

Mail Service Configuration
Select the mail protocol your system uses. Tornado supports both the Internet standard SMTP and the Microsoft standard MAPI. If you use SMTP, also specify the name of your mail gateway (contact your site’s system administrator if you do not have this information).
10.2.2 Sending a Tornado Support Request (TSR)

See Figure 10-2 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 Tornado Support Request form by electronic mail or by FAX. 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.

Figure 10-2  Tornado Support Request Form

The Support Request form is organized into the major areas illustrated in Figure 10-2. The sample text in Figure 10-2 shows the recommended way to use the Problem Synopsis and Problem Description fields, which are discussed further below.
History Area

TSR File
The file where Tornado saves this TSR for your reference. By default Tornado stores these files under &lt;c:\tornado\wind\tsr.&lt; This is a drop-down list box; it is filled in automatically with a name for a new TSR, but you can display a list of previously saved TSRs by clicking the button to the right of the field. To view the information from any TSR in this list, click on its file name: the contents of the Tornado Support Request form are filled in with the saved TSR data.

Save button
Save the information currently specified (without transmitting a TSR). For example, use this button to save a partly completed TSR while you investigate the problem further.

New button
Clear the currently displayed TSR. As a safeguard, this button is disabled until you either save or discard the current TSR information.

Remove button
Discard the currently displayed TSR information. If you decide to start over after beginning to enter a TSR, use this button first, then New.

Host Information

OS/Version
The version of Windows where Tornado is running. 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.

Product/Release
The name and release level of the product which is the subject of this TSR. This is filled in by default with the identification of Tornado as a whole. To select a more specific component name, click the button to the right of the field and select an appropriate item in the drop-down list box. If nothing in the list is appropriate, you can type a product and release level directly in this field.

Date
The date and time when you first opened this TSR, filled in automatically. Tornado does not allow you to edit this field.
Reference No.
A place for any reference number your own organization may find useful to associate with this TSR.

Profile button
Bring up the dialog box (10.2.1 The User Profile, p.306) that records name, contact, and address information for your TSRs.

Target Information

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. The drop-down list box allows you to select any architecture for which Tornado support was installed on your system. If nothing in the list is appropriate, you can type a target architecture directly in this field. If the problem appears on multiple architectures, you can select more than one by using SHIFT+click or CTRL+click to extend the selection.

BSP
The name of the Board Support Package linked into your target run-time. The drop-down list box allows you to select any BSP for which Tornado support was installed on your system. If nothing in the list is appropriate, you can type a BSP name directly in this field.

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.

Peripherals
List any other hardware that might be involved in your problem, such as SCSI devices or Ethernet cards.

Problem Information

Problem Synopsis
A terse line to identify the problem. Make this line as informative as possible. When the WRS support group responds to your support request, their mail uses the problem synopsis as the subject line. Use keywords that help you to recognize the problem.
Problem Description
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 paste text copied from any editor. To paste into the TSR form, click to place the text cursor where you want to paste; then either press CTRL+V, or click Paste in the Edit menu.

Attachments
The Attachments box allows you to include any information you may have already saved that is relevant to this problem. The Add button brings up a standard file browser that allows you to select a file, and records the file as an attachment to your TSR. The Remove button removes the selected item from the current list of attachments. When you send the support request, Tornado adds the contents of all files listed as attachments to your problem description.

You may include object code as an attachment if necessary. Tornado recognizes binary data and encodes it for safe transmission.

Action Buttons
The buttons at the bottom right of the main Support Request form allow you to transmit the TSR or exit the TSR form.

Send Mail
Save the current support request, then send the support request through e-mail or print the trouble report. This button brings up a dialog box to handle the transmission (with the destination already filled in as specified by your saved user profile). This dialog box is shown in Figure 10-3.

Close
Discard the Support Request form. A confirmation dialog box appears to give you the opportunity to save the TSR if necessary.

Help
Describe the Support Request form.
10.3 WRS Broadcasts on the World Wide Web

Wind River Systems publishes current information on products and customer services on the World Wide Web. If you wish, you can start your Web browser from the Tornado Tools menu; for an example, see World Wide Web, p.300.

Links to the following pages are available from the WRS home page:

Corporate Information
   Information about Wind River’s history, mission, and management team.

Product Information
   Information about ordering or inquiring about Wind River Systems products.

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. This link provides information on WRS training services, course schedules, and prices.

Customer Support
   Contact information for WRS Customer Support Services, and pointers to other technical information (including access to Wind Technical Notes and to comp.os.vxworks, the newsgroup devoted to VxWorks and its development tools).

Custom Engineering Services
   Information about the contract services available from the Wind River Systems Engineering Services group.
TradeWinds Directory
Product and contact information about third-party products to work with Tornado.

Wind River Users 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.
Appendices
A

Directories and Files

A.1 Introduction

Wind River Systems products are installed in a single directory tree. The directory root is shown as `c:\tornado` in our documentation, but you may choose whatever root is most appropriate for your site. 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>
<thead>
<tr>
<th>Directory/File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>docs</td>
<td>Directory of online documentation in HTML format.</td>
</tr>
<tr>
<td>host</td>
<td>Directory of host-resident tools and associated files. Described in more detail in A.2 Host Directories and Files, p.318.</td>
</tr>
<tr>
<td>SETUP</td>
<td>Directory of SETUP program.</td>
</tr>
<tr>
<td>share</td>
<td>Directory of protocol definitions shared by both host and target software.</td>
</tr>
<tr>
<td>target</td>
<td>Directory of VxWorks target-resident software and associated files. Described in more detail in A.3 Target Directories and Files, p.319.</td>
</tr>
</tbody>
</table>
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  c:\tornado\host

<table>
<thead>
<tr>
<th>Directory/File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>include</td>
<td>Directory containing header files for the Tornado tools.</td>
</tr>
<tr>
<td>host-os</td>
<td>Host-specific directory to permit Tornado installations for multiple hosts to be installed in a single tree, and share files in other directories.</td>
</tr>
<tr>
<td>(x86-win32 for Windows hosts)</td>
<td></td>
</tr>
<tr>
<td>host-os\bin</td>
<td>Directory containing Tornado tool executables (including GNU ToolKit binaries) on a particular host. This directory must be on your execution path to use Tornado conveniently.</td>
</tr>
<tr>
<td>host-os\lib</td>
<td>Directory containing subroutine libraries for the Tornado tools.</td>
</tr>
<tr>
<td>host-os\lib\backend</td>
<td>Directory containing dynamic-link libraries that implement the communications back ends available to the target server.</td>
</tr>
<tr>
<td>host-os\lib\gcc-lib</td>
<td>Directory containing the separate programs called by the GNU compiler driver.</td>
</tr>
</tbody>
</table>
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 \( c:\tornado\target \)

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>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 <strong>all</strong>, which contains modules common to all implementations of VxWorks (<strong>system-independent</strong> modules), and a subdirectory for each port of VxWorks to specific target hardware (<strong>system-dependent</strong> modules).</td>
<td></td>
</tr>
<tr>
<td>config\all</td>
<td>Subdirectory containing system configuration modules. <em>Note that this method of configuration has been replaced by the project facility</em> (see 4. Projects). The directory includes the following files:</td>
<td><strong>bootInit.c</strong> System-independent boot ROM facilities.</td>
</tr>
<tr>
<td></td>
<td><strong>configAll.h</strong></td>
<td>Generic header file used to define configuration parameters common to all targets.</td>
</tr>
<tr>
<td></td>
<td><strong>usrConfig.c</strong>,</td>
<td>Source of the configuration module for a VxWorks development system (<strong>usrConfig.c</strong>), and a configuration module for the VxWorks boot ROM (<strong>bootConfig.c</strong>).</td>
</tr>
<tr>
<td></td>
<td><strong>bootConfig.c</strong></td>
<td></td>
</tr>
<tr>
<td>config\bspname</td>
<td>The other subdirectories of <strong>config</strong> contain system-dependent modules for each port of VxWorks to a particular target board. Each of these directories includes the following files:</td>
<td><strong>Makefile</strong> Makefile for creating boot ROMs and the VxWorks system image for a particular target.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>sysLib.c</strong>, <strong>sysALib.s</strong> Two source modules of system-dependent routines.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>sysSerial.c</strong> Driver for on-board serial ports.</td>
</tr>
<tr>
<td></td>
<td><strong>config.h</strong></td>
<td>Header file of hardware configuration parameters.</td>
</tr>
<tr>
<td></td>
<td><strong>bspname.h</strong></td>
<td>Header file for the target board.</td>
</tr>
<tr>
<td></td>
<td><strong>romInit.s</strong></td>
<td>Assembly language source for initialization code that is the entry point for the VxWorks boot ROMs and ROM-based versions of VxWorks.</td>
</tr>
<tr>
<td></td>
<td><strong>vxWorks</strong>, <strong>vxWorks.sym</strong> Complete, linked VxWorks system binary (<strong>vxWorks</strong>), and its symbol table (<strong>vxWorks.sym</strong>) created with the supplied configuration files.</td>
<td></td>
</tr>
</tbody>
</table>
Directories and Files

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bootrom,</td>
<td>bootrom.hex</td>
<td>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.</td>
</tr>
<tr>
<td>h</td>
<td></td>
<td>Directory containing all the header (include) files supplied by VxWorks. Your application modules must include several of them in order to access VxWorks facilities. The h directory also contains the following subdirectories:</td>
</tr>
<tr>
<td>h\arch</td>
<td></td>
<td>Directory containing architecture-dependent header files.</td>
</tr>
<tr>
<td>h\arpa</td>
<td></td>
<td>Directory containing a header file for use with inetLib.</td>
</tr>
<tr>
<td>h\drv</td>
<td></td>
<td>Directory containing hardware-specific header files (primarily for drivers). Not all of the subdirectories shown are present for all BSPs.</td>
</tr>
<tr>
<td>h\make</td>
<td></td>
<td>Directory containing files that describe the rules for the makefiles for each CPU and toolset.</td>
</tr>
<tr>
<td>h\net</td>
<td></td>
<td>Directory containing all the internal header (include) files used by the VxWorks network. Network drivers must include several of these header files, but no application modules should need them.</td>
</tr>
<tr>
<td>h\netinet</td>
<td></td>
<td>Directory containing Internet-specific header files.</td>
</tr>
<tr>
<td>h\private</td>
<td></td>
<td>Directory containing header files for code private to VxWorks.</td>
</tr>
<tr>
<td>h\rpc</td>
<td></td>
<td>Directory containing header files that must be included by applications using the Remote Procedure Call library (RPC).</td>
</tr>
<tr>
<td>h\sys</td>
<td></td>
<td>Directory containing header files specified by POSIX.</td>
</tr>
<tr>
<td>h\types</td>
<td></td>
<td>Directory containing header files used for defining types.</td>
</tr>
<tr>
<td>lib</td>
<td></td>
<td>Directory containing the machine-independent object libraries and modules provided by VxWorks.</td>
</tr>
<tr>
<td>lib\objcpuol\vx</td>
<td></td>
<td>Directory containing VxWorks object modules as individual files (suitable for loading dynamically to the target).</td>
</tr>
<tr>
<td>lib\libcpuol\vx.a</td>
<td></td>
<td>Archive (ar) format library containing the object modules that make up VxWorks.</td>
</tr>
<tr>
<td>src</td>
<td></td>
<td>Directory containing all source files for VxWorks.</td>
</tr>
</tbody>
</table>
### Table A-3  \texttt{c:\tornado\target} (Continued)

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src\config</td>
<td>ansi_5_0.c</td>
<td>Used to include all 5.0 ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>assertInit.c</td>
<td>Used to include the \texttt{assert} ANSI C library routine.</td>
</tr>
<tr>
<td></td>
<td>ctypeInit.c</td>
<td>Used to include the \texttt{ctype} ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>localeInit.c</td>
<td>Used to include the \texttt{locale} ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>mathInit.c</td>
<td>Used to include the \texttt{math} ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>stdioInit.c</td>
<td>Used to include the \texttt{stdio} ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>stlibInit.c</td>
<td>Used to include the \texttt{stdlib} ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>stringInit.c</td>
<td>Used to include the \texttt{string} ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>timeInit.c</td>
<td>Used to include the \texttt{time} ANSI C library routines.</td>
</tr>
<tr>
<td></td>
<td>usrDepend.c</td>
<td>Used to check module dependences for constants defined in \texttt{configAll.h} and \texttt{config.h}.</td>
</tr>
<tr>
<td></td>
<td>usrExtra.c</td>
<td>Used to include extra modules that are needed by VxWorks but not referenced in the VxWorks code.</td>
</tr>
<tr>
<td></td>
<td>usrFd.c</td>
<td>Used to mount a dosFs file system on a boot diskette (i386/i486 targets only).</td>
</tr>
<tr>
<td></td>
<td>usrKernel.c</td>
<td>Used to configure and initialize the \texttt{wind} kernel.</td>
</tr>
<tr>
<td></td>
<td>usrLde.c</td>
<td>Used to mount a dosFs file system on a boot IDE hard disk drive (i386/i486 targets only).</td>
</tr>
<tr>
<td></td>
<td>usrLoadSym.c</td>
<td>Used to load the VxWorks symbol table.</td>
</tr>
<tr>
<td></td>
<td>usrMmuInit.c</td>
<td>Used to initialize the memory management unit.</td>
</tr>
<tr>
<td></td>
<td>usrNetwork.c</td>
<td>Used to configure and initialize networking support.</td>
</tr>
<tr>
<td></td>
<td>usrScript.c</td>
<td>Used to execute a startup script when VxWorks first boots.</td>
</tr>
<tr>
<td></td>
<td>usrScsi.c</td>
<td>Used to configure and initialize SCSI support.</td>
</tr>
<tr>
<td></td>
<td>usrSmObj.c</td>
<td>Used to configure and initialize shared memory object support.</td>
</tr>
<tr>
<td>Directories</td>
<td>Files</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>usrWdb.c</strong></td>
<td>Used to configure and initialize the Tornado target agent.</td>
<td></td>
</tr>
<tr>
<td><strong>cplusHeap.c, cplusIos.c, cplusTools.c</strong></td>
<td>Used to configure Rogue Wave C++ utility classes (from optional product: Wind Foundation Classes).</td>
<td></td>
</tr>
<tr>
<td><strong>cplusVxw.c</strong></td>
<td>Used to configure VxWorks wrapper classes (from optional product: Wind Foundation Classes).</td>
<td></td>
</tr>
<tr>
<td><strong>usrCplusTools.c</strong></td>
<td>Used to initialize Rogue Wave C++ utility classes (from optional product: Wind Foundation Classes).</td>
<td></td>
</tr>
<tr>
<td><strong>usrCplusVxw.c</strong></td>
<td>Used to initialize VxWorks wrapper classes (from optional product: Wind Foundation Classes).</td>
<td></td>
</tr>
</tbody>
</table>

**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.

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

- **src\arch**: Directory containing VxWorks source code for architecture-specific modules.
- **src\cplus**: Directory containing source code for the Wind 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 the BOOTP server.

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ramDrv.c</strong></td>
<td>Block device driver that allows memory to be used as a device with VxWorks local file systems.</td>
<td></td>
</tr>
</tbody>
</table>

Table A-3 c:\tornado\target *(Continued)*
A.4 Initialization and State-Information Files

You can define initialization files to customize each of the Tornado tools. These files are not distributed as a part of Tornado; it is up to you to define them if you wish. If they are present, the initialization files are collected in a directory called .wind. Tornado looks for this directory in two places: first under c:\tornado (that is, your base installation directory), and then under the directory specified by the HOME environment variable (if defined). In each of these directories, if an initialization file is found, its contents are sourced as Tcl code.

Some Tornado tools also use the c:\tornado\.wind directory to store state information (and some optional products store both initialization and state information there).

Table A-4 and Table A-5 are a description of the kinds of files that can be stored in the c:\tornado\.wind directory.

Table A-4  Initialization Files

<table>
<thead>
<tr>
<th>Directory/File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>browser.tcl</td>
<td>Optional Tcl initialization code for the browser. See Tcl, p.327.</td>
</tr>
<tr>
<td>crosswind.tcl</td>
<td>Optional Tcl initialization code for the debugger front end. See 8.8 Tcl: Debugger Customization, p.274.</td>
</tr>
<tr>
<td>gdb.tcl</td>
<td>Optional Tcl initialization code for the debugging engine itself. See 8.8 Tcl: Debugger Customization, p.274.</td>
</tr>
<tr>
<td>Tornado.tcl</td>
<td>Optional Tcl initialization code for the Tools menu and toolbar, and for overall Tornado initialization.</td>
</tr>
<tr>
<td>windsh.tcl</td>
<td>Optional Tcl initialization code for the shell. See 6.7.3 Tcl: Tornado Shell Initialization, p.200.</td>
</tr>
<tr>
<td>wtstcl.tcl</td>
<td>Optional Tcl initialization code for wtstcl, the Tcl interpreter with WTX-protocol extensions. See the Tornado API Guide: Extending Tornado Tools.</td>
</tr>
</tbody>
</table>

Table A-5  State-Information Files

<table>
<thead>
<tr>
<th>Directory/File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>profile</td>
<td>A file of identification information used for your Tornado support requests. This information is collected and updated through the Support command in the Tools menu; see 10.2.1 The User Profile, p.306.</td>
</tr>
</tbody>
</table>
A directory recording the history of your Tornado support requests. This information is managed through the `Support` command in the `Tools` menu; see 10.2.2 *Sending a Tornado Support Request (TSR)*, p.308.
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:

- 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>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>set num 6</td>
<td>6</td>
</tr>
<tr>
<td>set y hello</td>
<td>hello</td>
</tr>
<tr>
<td>set z &quot;hello world&quot;</td>
<td>hello world</td>
</tr>
<tr>
<td>set t $z</td>
<td>hello world</td>
</tr>
<tr>
<td>set u &quot;$z $y&quot;</td>
<td>hello world hello</td>
</tr>
<tr>
<td>set v {$z $y}</td>
<td>$z $y</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>set num</td>
<td>6</td>
</tr>
<tr>
<td>set z</td>
<td>hello world</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>llength $v</td>
<td>2</td>
<td>Length of list v.</td>
</tr>
<tr>
<td>lindex $u 1</td>
<td>world</td>
<td>Second element of list u.</td>
</tr>
<tr>
<td>set long &quot;a b c d e f g&quot;</td>
<td>a b c d e f g</td>
<td>Define a longer list.</td>
</tr>
<tr>
<td>lrange $long 2 4</td>
<td>c d e</td>
<td>Select elements 2 through 4 of list long.</td>
</tr>
<tr>
<td>lreplace $long 2 4 C D E</td>
<td>a b C D E f g</td>
<td>Replace elements 2 through 4 of list long.</td>
</tr>
<tr>
<td>set V &quot;(c d e) f {h {i j} k}&quot;</td>
<td></td>
<td>Define a list of lists.</td>
</tr>
<tr>
<td>lindex $V 1</td>
<td>f</td>
<td>Some elements of V are singletons.</td>
</tr>
<tr>
<td>lindex $V 0</td>
<td>c d e</td>
<td>Some elements of V are lists.</td>
</tr>
</tbody>
</table>

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.331), 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: \textit{arrayName(keyString)}. The \textit{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>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{set taskId(tNetTask)}</td>
<td>0x4f300</td>
<td>Get element tNetTask of array \textit{taskId}.</td>
</tr>
<tr>
<td>\textit{set cpuFamily(5) m68k}</td>
<td>m68k</td>
<td>Define array \textit{cpuFamily} and an element keyed 5.</td>
</tr>
<tr>
<td>\textit{set cpuFamily(10) sparc}</td>
<td>sparc</td>
<td>Define element keyed 10 of array \textit{cpuFamily}.</td>
</tr>
<tr>
<td>\textit{set cpuId 10}</td>
<td>10</td>
<td>Define \textit{cpuId}, and use it as a key to \textit{cpuFamily}.</td>
</tr>
<tr>
<td>\textit{set cpuFamily($cpuId)}</td>
<td>sparc</td>
<td></td>
</tr>
</tbody>
</table>

B.2.4 Command Substitution

In Tcl, you can capture the result of the command as text by enclosing the command in square brackets \([\ldots]\). The Tcl interpreter substitutes the command result in the same process that is already running, which makes this an efficient operation.

<table>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{set m [lrange $long 2 4]}</td>
<td>c d e</td>
</tr>
<tr>
<td>\textit{set n [lindex $m 1]}</td>
<td>d</td>
</tr>
<tr>
<td>\textit{set o [lindex [lrange $long 2 4] 1]}</td>
<td>d</td>
</tr>
<tr>
<td>\textit{set x [lindex [lindex $V 2] 1]}</td>
<td>i j</td>
</tr>
</tbody>
</table>

The last example selects from a list of lists (defined among the examples in B.2.2 Lists in Tcl, p.330). 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>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr (2 &lt;&lt; 2) + 3</code></td>
<td>11</td>
</tr>
<tr>
<td><code>expr 0xff00 &amp; 0xf00</code></td>
<td>3840</td>
</tr>
</tbody>
</table>

### 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>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set myfile [open myfile.out w]</code></td>
<td>Open a file for writing.</td>
</tr>
<tr>
<td><code>puts $myfile [format &quot;%s %d\n&quot; &quot;you are number&quot; [expr 3+3]]</code></td>
<td>Format a string and write it to file.</td>
</tr>
<tr>
<td><code>close $myfile</code></td>
<td>Close the file.</td>
</tr>
<tr>
<td><code>file exists myfile.out</code></td>
<td>1</td>
</tr>
<tr>
<td><code>file writable myfile.out</code></td>
<td>1</td>
</tr>
<tr>
<td><code>file executable myfile.out</code></td>
<td>0</td>
</tr>
<tr>
<td><code>glob *.o</code></td>
<td><code>testCall.o foo.o bar.o</code></td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>square 4</td>
<td>16</td>
</tr>
<tr>
<td>square [sum 1 2 3]</td>
<td>36</td>
</tr>
<tr>
<td>set x &quot;squ&quot;</td>
<td>squ</td>
</tr>
<tr>
<td>set y &quot;are&quot;</td>
<td>are</td>
</tr>
<tr>
<td>$x$y 4</td>
<td>16</td>
</tr>
</tbody>
</table>

Table B-8 Calling a Tcl Procedure

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:

```tcl
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.328.

### 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.328.

**B.3 Tcl Coding Conventions**

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 10-1  Tcl File and Module Headings**

```tcl
# 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 B.3.4 Declaration Formats, p.339.

The following is an example of a standard procedure layout.

**Example 10-2 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 10-3 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.341, 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 symSortByNames 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:

```tcl
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:
    ```tcl
    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.341).
The if statement is not an exception. The conditionally executed statement always goes on a separate line from the conditional expression:

```tcl
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:

  ```tcl
  set status [fooGet $foo [expr $i + 3] $value]
  if {($value & $mask) { 
  ```

- Line up continuation lines with the part of the preceding line they continue:

  ```tcl
  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:

  ```tcl
  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.343).
- 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:
    
    ```tcl
    # This is the correct format for a single-line comment
    set foo 0
    
    # 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
    ``
  - 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.
    
    ```tcl
    # 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:

  ```tcl
  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:

  ```tcl
  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 {
            ...
          }`
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 \textit{RETURNS} block. If a complex element is returned, provide a complete description of the element layout. Do not use the \texttt{return} statement to indicate that an abnormal situation has occurred; use the \texttt{error} statement in that situation.

The following illustrates a complex return value consisting of a description:

\begin{verbatim}
# 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]]
\end{verbatim}

The following illustrates a simple return value:

\begin{verbatim}
# 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
}
\end{verbatim}

Error Conditions: The Tcl \texttt{error} command raises an error condition that can be trapped by the \texttt{catch} command. If not caught, an error condition terminates script execution. For example:

\begin{verbatim}
if {$defaultTaskId == 0} {
    error "No default task has been established."
}
\end{verbatim}

Because every error message and error code must be described in the procedure header in the \textit{ERRORS} block, it is sometimes useful to call \texttt{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)"
}
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgtsvr</td>
<td>the target board server</td>
<td>352</td>
</tr>
<tr>
<td>vxColor</td>
<td>graph coloring demo for Tornado</td>
<td>361</td>
</tr>
<tr>
<td>windsh</td>
<td>Tornado Shell</td>
<td>365</td>
</tr>
<tr>
<td>agentModeShow()</td>
<td>show the agent mode (*)</td>
<td>374</td>
</tr>
<tr>
<td>b()</td>
<td>set or display breakpoints</td>
<td>375</td>
</tr>
<tr>
<td>bd()</td>
<td>delete a breakpoint</td>
<td>376</td>
</tr>
<tr>
<td>bdall()</td>
<td>delete all breakpoints</td>
<td>376</td>
</tr>
<tr>
<td>bh()</td>
<td>set a hardware breakpoint</td>
<td>377</td>
</tr>
<tr>
<td>bootChange()</td>
<td>change the boot line</td>
<td>377</td>
</tr>
<tr>
<td>browse()</td>
<td>send a message to the browser asking it to browse an address (*)</td>
<td>378</td>
</tr>
<tr>
<td>c()</td>
<td>continue from a breakpoint</td>
<td>378</td>
</tr>
<tr>
<td>cd()</td>
<td>change the default directory</td>
<td>379</td>
</tr>
<tr>
<td>checkStack()</td>
<td>print a summary of each task’s stack usage</td>
<td>380</td>
</tr>
<tr>
<td>classShow()</td>
<td>show information about a class of objects (*)</td>
<td>380</td>
</tr>
<tr>
<td>cplusCtors()</td>
<td>call static constructors (C++)</td>
<td>381</td>
</tr>
<tr>
<td>cplusDtors()</td>
<td>call static destructors (C++)</td>
<td>382</td>
</tr>
<tr>
<td>cplusStratShow()</td>
<td>show C++ static constructors calling strategy (*)</td>
<td>382</td>
</tr>
<tr>
<td>cplusXtorSet()</td>
<td>change C++ static constructor calling strategy (C++)</td>
<td>383</td>
</tr>
<tr>
<td>cret()</td>
<td>continue until the current subroutine returns</td>
<td>384</td>
</tr>
<tr>
<td>d()</td>
<td>display memory</td>
<td>384</td>
</tr>
<tr>
<td>devs()</td>
<td>list all system-known devices</td>
<td>385</td>
</tr>
<tr>
<td>h()</td>
<td>display or set the size of shell history</td>
<td>385</td>
</tr>
<tr>
<td>help()</td>
<td>print a synopsis of selected routines</td>
<td>386</td>
</tr>
<tr>
<td>hostShow()</td>
<td>display the host table</td>
<td>387</td>
</tr>
<tr>
<td>t()</td>
<td>print a summary of each task’s TCB, task by task</td>
<td>387</td>
</tr>
<tr>
<td>icmpstatShow()</td>
<td>display statistics for ICMP</td>
<td>388</td>
</tr>
<tr>
<td>ifShow()</td>
<td>display the attached network interfaces</td>
<td>388</td>
</tr>
<tr>
<td>inetstatShow()</td>
<td>display all active connections for Internet protocol sockets</td>
<td>389</td>
</tr>
<tr>
<td>intVecShow()</td>
<td>display the interrupt vector table</td>
<td>390</td>
</tr>
</tbody>
</table>
iosDevShow() — display the list of devices in the system .............................................. 390
iosDrvShow() — display a list of system drivers ................................................................. 390
iosFdShow() — display a list of file descriptor names in the system .......................... 391
ipstatShow() — display IP statistics ................................................................................ 391
iStrict() — print a summary of all task TCBs, as an atomic snapshot (*) ................... 392
l() — disassemble and display a specified number of instructions .............................. 392
ld() — load an object module into memory ................................................................... 393
lkAddr() — list symbols whose values are near a specified value ............................... 394
lkup() — list symbols ...................................................................................................... 394
ls() — list the contents of a directory ............................................................................. 395
m() — modify memory ..................................................................................................... 395
memPartShow() — show partition blocks and statistics ..................................................... 396
memShow() — show system memory partition blocks and statistics ............................... 397
moduleIdFigure() — figure out module ID, given name or number (*) ......................... 398
moduleShow() — show the current status for all the loaded modules .............................. 398
mqPxShow() — show information about a POSIX message queue (*) ........................... 399
mRegs() — modify registers ............................................................................................. 399
msgQShow() — show information about a message queue ............................................. 400
period() — spawn a task to call a function periodically ............................................... 401
printErrno() — print the definition of a specified error status value ............................. 402
printLogo() — display the Tornado logo ......................................................................... 402
pwd() — display the current default directory ................................................................ 402
quit() — shut down WindSh (*) ..................................................................................... 403
reboot() — reset network devices and transfer control to boot ROMs ............................ 403
repeat() — spawn a task to call a function repeatedly .................................................... 404
routestatShow() — display routing statistics ................................................................. 404
s() — single-step a task ................................................................................................... 405
semPxShow() — show information about a POSIX semaphore (*) ................................. 405
semShow() — show information about a semaphore ....................................................... 406
shellHistory() — display or set the size of shell history .................................................. 407
shellPromptSet() — change the shell prompt ................................................................. 407
show() — display information on a specified object ....................................................... 408
smMemPartShow() — show user shared memory system partition blocks and statistics (*) .... 408
smMemShow() — show the shared memory system partition blocks and statistics .......... 409
so() — single-step, but step over a subroutine ................................................................ 410
sp() — spawn a task with default parameters ............................................................... 410
spsp() — spawn a task with default parameters, and leave it suspended (*) ................ 412
sysResume() — reset the agent to tasking mode (*) ......................................................... 412
sysStatusShow() — show system context status (*) ......................................................... 413
sysSuspend() — set the agent to external mode and suspend the system (*) ............... 413
taskCreateHookShow() — show the list of task create routines .................................... 414
taskDeleteHookShow() — show the list of task delete routines ..................................... 414
taskIdDefault() — set the default task ID ...................................................................... 414
taskIdFigure() — figure out the task ID of a specified task (*) ...................................... 415
taskRegsShow() — display the contents of a task’s registers ......................................... 416
taskShow() – display task information from TCBs .................................................. 416
\( \text{taskSwitchHookShow()} \) – show the list of task switch routines ........................................ 417
\( \text{taskWaitShow()} \) – show information about the object a task is pended on (*) .................. 418
tcpstatShow() – display all statistics for the TCP protocol .............................................. 419
td() – delete a task ............................................................................................................ 419
tftpInfoShow() – get TFTP status information ................................................................. 419
ti() – display complete information from a task’s TCB ................................................. 420
tr() – resume a task .......................................................................................................... 421
ts() – suspend a task .......................................................................................................... 421
tt() – display a stack trace of a task .................................................................................. 422
tw() – print info about the object the given task is pending on (*) ................................. 423
udpstatShow() – display statistics for the UDP protocol .................................................. 423
unld() – unload an object module by specifying a file name or module ID ..................... 423
version() – print VxWorks version information ............................................................... 424
w() – print a summary of each task’s pending information, task by task (*) .................. 425
wdShow() – show information about a watchdog .............................................................. 425
wtxCtest – test suite for the WTX C API ............................................................................ 426
wtxregd – the Tornado service registry ............................................................................ 426
wtxtcl – the Tornado Tcl shell ......................................................................................... 429

**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.)
tgtsvr

NAME

tgtsvr – the target board server

SYNOPSIS
tgtsvr [-A.llvm] [-B.ackend backendName] [-B.d.ebug fileName]
[-B.m.ax size] [-B.ps linespeed] [-B.r.esend number]
[-B.t.imeout timeout] [-C.onsole] [-C.ore fileName]
[-d.device device] [-d.show hostName:0]
[-f.ormat formatName] [-h.elp] [-hfc] [-L.ock]
[-m.emory nbytes] [-n.ame serverName] [-N.osyms]
[-p.ort portNumber] [-R TSFS_root] [-rredirectIO]
[-rredirectShell] [-RW] [-s.ychro] [-u.se portmapper]
[-u.sers fileName] [-V.ersion] [-v.ersion]
[-W.d.ebug fileName] [-W.f.ilter request] [-W.m.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 “$HOME”, or “$HOMEDRIVES$HOMEPATH” if “$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 exist, 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  
Name of the alternate object module format (for example, a.out or COFF) that will be managed by the target server.

-h  
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  
Restrict access to this target server to processes running under the same user ID as the server.

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

-n 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  
Do not use a core file to build the target symbol table.

-p 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  
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.

-a 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.
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 D:\tornado\target\config\mv1604\vxWorks -V \\-B wdbserial -b 38400 myTarget
```

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 -redirectShell -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**


---

**vxColor**

**NAME**

vxColor – graph coloring demo for Tornado

**SYNOPSIS**

```
vxColor targetServer [-V.verbose]
```

**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:

<table>
<thead>
<tr>
<th>Contextual Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>Starts a coloring session.</td>
</tr>
<tr>
<td>COLOR EDIT</td>
<td>Menu for changing region colors:</td>
</tr>
<tr>
<td>- RANDOM</td>
<td>randomly initializes each region’s color.</td>
</tr>
</tbody>
</table>
During a coloring session, the contextual controls are:

- **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
make CPU=MC68040
```
Solaris simulator:

```make
make CPU=SIMPARCSOLARIS
```

Intel i960CA board:

```make
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/CPUTOOLtest/vxColor.o`
  a CPU specific demo object module.

BUGS

If the NumLock key is on, the demo will hang.

SEE ALSO

`tgsr`, *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
b() – 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
tit() – display complete information from a task’s TCB
tr() – resume a task
ts() – suspend a task
tt() – display a stack trace of a task
twt() – 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
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 \( i() \), \( ti() \), \( dl() \), \( l() \), \( ts() \), \( tr() \), \( td() \), \( ld() \). 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:

\( w() \) – 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 \( i() \), \( ti() \), \( dl() \), \( l() \), \( ts() \), \( tr() \), \( td() \), \( ld() \). 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:
Tornado 2.0
User’s Guide

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,
  ```
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:

```bash
# 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)};
    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:

```bash
-> 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:

```bash
-> lkup "foo"
__GLOBAL__$I$foo(int) 0x003760f8 text  (cptest.o)
_foo(char *) 0x00375fcb text  (cptest.o)
value = 0 = 0x0
```

Try to call “talk” and you’ll be asked which one to invoke:

```bash
-> 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_COMMONMATCH_ALL [on | off]
Set the loader behavior for common symbols. If LD_COMMONMATCH_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_COMMONMATCH_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.

```bash
-> ?shConfig
SH_GET_TASK_IO = on
LD_CALL_XTORS = target
LD_SEND_MODULES = on
LD_COMMONMATCH_ALL = on
LD_PATH = C:/ProjX/lib/objR4650gnutest;/C:/ProjY/lib/objR4650gnu vx/
-> ?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.
agentModeShow()

NAME

agentModeShow() – show the agent mode (*)

SYNOPSIS

int agentModeShow (void)

SEE ALSO
tgtsvr, launch, browser, Tornado User’s Guide

Commands marked with (*) have no equivalents in the VxWorks shell.
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**

```c
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:

```bash
-> 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:

```bash
-> 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 +1`st 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 breakpoint 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**

```c
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 table is full.

**SEE ALSO**
winds, *b*(), *bd*(), *Tornado User’s Guide: Shell*

---

**bootChange()**

**NAME**
bootChange() – change the boot line

**SYNOPSIS**

```c
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**
winds, *Tornado User’s Guide: Shell*
**browse()**

**NAME**

`browse()` – send a message to the browser asking it to browse an address (*)

**SYNOPSIS**

```c
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**

```c
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.
cd()

NAME

`cd()` – change the default directory

SYNOPSIS

```c
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`, `tr()`, `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  

<table>
<thead>
<tr>
<th>NAME</th>
<th>ENTRY</th>
<th>TID</th>
<th>SIZE</th>
<th>CUR</th>
<th>HIGH</th>
<th>MARGIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>t28</td>
<td>_foo</td>
<td>23e1c78</td>
<td>9208</td>
<td>832</td>
<td>3632</td>
<td>5576</td>
</tr>
</tbody>
</table>

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, \texttt{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:

\begin{verbatim}
-> lkup "ClassId"
\end{verbatim}

There is no target-resident version of \texttt{classShow()}.  

RETURNS
N/A  

SEE ALSO
\texttt{windsh}, \textit{Tornado User's Guide: Shell}

---

\textbf{cplusCtors()}

\textbf{NAME}
\texttt{cplusCtors()} – call static constructors (C++)

\textbf{SYNOPSIS}
\begin{verbatim}
void cplusCtors
(const char * moduleName /* name of loaded module */
)
\end{verbatim}

\textbf{DESCRIPTION}
This function is used to call static constructors under the manual strategy (see \texttt{cplusXtorSet()}). \texttt{moduleName} is the name of an object module that was \textit{munched} before loading. If \texttt{moduleName} is 0, then all static constructors, in all modules loaded by the target server loader, are called.

\textbf{EXAMPLES}
The following example shows how to initialize the static objects in modules called \texttt{applx.out} and \texttt{apply.out}:

\begin{verbatim}
-> cplusCtors "applx.out"
value = 0 = 0x0
-> cplusCtors "apply.out"
value = 0 = 0x0
\end{verbatim}

The following example shows how to initialize all the static objects that are currently loaded, with a single invocation of \texttt{cplusCtors()}:

\begin{verbatim}
-> cplusCtors
value = 0 = 0x0
\end{verbatim}

\textbf{RETURNS}
N/A
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.

- \texttt{cplusStratShow}
  
  C++ ctors/dtors strategy set to MANUAL
  
  value = 0 = 0x0

RETURNS

N/A

SEE ALSO

\texttt{windsh}, \texttt{cplusXtorsSet()}, \texttt{Tornado User's Guide: Shell}

\underline{cplusXtorSet()}

NAME

\texttt{cplusXtorSet()} – change C++ static constructor calling strategy (C++)

SYNOPSIS

\begin{verbatim}
void cplusXtorSet
  (  
    int strategy /* constructor calling strategy */
  )
\end{verbatim}

DESCRIPTION

This command sets the C++ static constructor calling strategy to \textit{strategy}. The default strategy is 0.

There are two static constructor calling strategies: \textit{automatic} and \textit{manual} represented by the following numeric codes:

\begin{center}
<table>
<thead>
<tr>
<th>Strategy</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>manual</td>
<td>0</td>
</tr>
<tr>
<td>automatic</td>
<td>1</td>
</tr>
</tbody>
</table>
\end{center}

Under the manual strategy, a module’s static constructors and destructors are called by \texttt{cplusCtors()} and \texttt{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

\texttt{windsh}, \texttt{cplusStratShow()}, \texttt{cplusCtors()}, \texttt{cplusDtors()}, \texttt{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 \textit{width} is omitted or zero, it defaults to the previous value. If \textit{width} is an invalid number, it is set to 1. The valid values for \textit{width} are 1, 2, 4, and 8. The number of units \texttt{d()} displayed is rounded up to the nearest number of full lines.

**RETURNS**

N/A

**SEE ALSO**

\texttt{windsh}, \texttt{m()}, \texttt{Tornado User's Guide: Shell}

---

**devs()**

**NAME**

\texttt{devs()} – list all system-known devices

**SYNOPSIS**

\texttt{void devs (void)}

**DESCRIPTION**

This command displays a list of all devices known to the I/O system.

**RETURNS**

N/A

**SEE ALSO**

\texttt{windsh}, \texttt{iosDevShow()}, \texttt{Tornado User's Guide: Shell}

---

**h()**

**NAME**

\texttt{h()} – display or set the size of shell history

**SYNOPSIS**

\texttt{void h}

\hspace{3em} (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 \texttt{size} is specified, that number of the most recent commands is saved for display. The value of \texttt{size} is initially 20.

**RETURNS**

N/A

**SEE ALSO**

\texttt{windsh}, \texttt{shellHistory()}, \texttt{Tornado User's Guide: Shell}
`help()`

### NAME

`help()` – print a synopsis of selected routines

### SYNOPSIS

```c
void help (void)
```

### DESCRIPTION

This command prints the following list of the calling sequences for commonly used routines.

```c
help                        Print this list
h         [n]               Print (or set) shell history
i         [task]            Summary of tasks\x00d5 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\x00d5 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        [symns[,noAbort][,"name"]]] Load stdin, or file, into memory
           (symns = 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
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>ENTRY</th>
<th>TID</th>
<th>PRI</th>
<th>STATUS</th>
<th>PC</th>
<th>SP</th>
<th>ERRNO</th>
<th>DELAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>tExcTask</td>
<td>_excTask</td>
<td>20fcb00</td>
<td>0</td>
<td>PEND</td>
<td>200c5fc</td>
<td>20fca6c</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tLogTask</td>
<td>_logTask</td>
<td>20f5b8</td>
<td>0</td>
<td>PEND</td>
<td>200c5fc</td>
<td>20f5b20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tRlogind</td>
<td>_rlogind</td>
<td>20f3b0</td>
<td>2</td>
<td>PEND</td>
<td>2038614</td>
<td>20f3b0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tTelnetd</td>
<td>_telnetd</td>
<td>20f20</td>
<td>2</td>
<td>PEND</td>
<td>2038614</td>
<td>20f20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tNetTask</td>
<td>_netTask</td>
<td>20f7398</td>
<td>50</td>
<td>PEND</td>
<td>2038614</td>
<td>20f7340</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

value = 57 = 0x39 = 9

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**

```c
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**

```c
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**

```c
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**

```c
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**

```c
void iosDrvShow (void)
```

**DESCRIPTION**

This routine displays a list of all drivers in the driver list.
**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**  
winds, 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**  
winds, 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:

```bash
-> 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.
NAME
ld() – load an object module into memory

SYNOPSIS

```c
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  

```c  
    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  

```c  
    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**

```c
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**

```c
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.
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:
Do not change this register, but continue, prompting at the next register.

_set 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, _mt_, 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:

<table>
<thead>
<tr>
<th>Information</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Queue Id</td>
<td>0x3f8c20</td>
</tr>
<tr>
<td>Task Queuing</td>
<td>FIFO</td>
</tr>
<tr>
<td>Message Byte Len</td>
<td>150</td>
</tr>
<tr>
<td>Messages Max</td>
<td>50</td>
</tr>
<tr>
<td>Messages Queued</td>
<td>0</td>
</tr>
<tr>
<td>Receivers Blocked</td>
<td>1</td>
</tr>
<tr>
<td>Send timeouts</td>
<td>0</td>
</tr>
<tr>
<td>Receive timeouts</td>
<td>0</td>
</tr>
</tbody>
</table>

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**

```c
int period
  (  
    int     secs, /* period in seconds */
    FUNCPTTR    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**

```c
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**

```c
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**

```c
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  
```c
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  

RoutingShow()  

Name  
routestatShow() – display routing statistics  

Synopsis  
```c
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:

<table>
<thead>
<tr>
<th>Semaphore name</th>
<th>namedSem</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_open() count</td>
<td>2</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TID</th>
<th>PRI</th>
<th>DELAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>tExcTask</td>
<td>3fd678</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>tLogTask</td>
<td>3f8ac0</td>
<td>0</td>
<td>611</td>
</tr>
</tbody>
</table>

RETURNS
OK or ERROR.

SEE ALSO
windsh, browse(), Tornado User’s Guide: Shell

406
**shellHistory()**

**NAME**

shellHistory() – display or set the size of shell history

**SYNOPSIS**

```c
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**

```c
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**

```c
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**

```c
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**

```c
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**

```bash
-> 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**
```c
int sps
(  
    FUNCPTTR 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**
winds, sp(), Tornado User's Guide: Shell

**sysResume()**

**NAME**
sysResume() – reset the agent to tasking mode (*)

**SYNOPSIS**
```c
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**
winds, 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**

```c
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**

```c
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**

```c
int taskIdDefault
```
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 \texttt{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 \texttt{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.

The most recent non-zero task ID.

\textit{taskIdFigure()} – figure out the task ID of a specified task (*)

This command returns the task ID of a task specified either by name or by number. If \texttt{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 \texttt{ERROR}.

There is no target-resident version of \textit{taskIdFigure}().

A task ID, or \texttt{ERROR}.

\textit{windsh, taskIdDefault()}, \textit{Tornado User’s Guide: Shell}
**taskRegsShow( )**

**NAME**

`taskRegsShow()` – display the contents of a task’s registers

**SYNOPSIS**

```c
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):

```plaintext
-> taskRegsShow (taskIdToId ("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**

```c
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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Task name</td>
</tr>
<tr>
<td>ENTRY</td>
<td>Symbol name or address where task began execution</td>
</tr>
<tr>
<td>TID</td>
<td>Task ID</td>
</tr>
<tr>
<td>PRI</td>
<td>Priority</td>
</tr>
<tr>
<td>STATUS</td>
<td>Task status, as formatted by taskStatusString( )</td>
</tr>
<tr>
<td>PC</td>
<td>Program counter</td>
</tr>
<tr>
<td>SP</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>ERRNO</td>
<td>Most recent error code for this task</td>
</tr>
<tr>
<td>DELAY</td>
<td>If task is delayed, number of clock ticks remaining in delay (0 otherwise)</td>
</tr>
</tbody>
</table>

RETURNS
N/A

SEE ALSO
windsh, Tornado User’s Guide: Shell

\[
\text{taskSwitchHookShow( )}
\]

NAME
\text{taskSwitchHookShow( )} – show the list of task switch routines

SYNOPSIS
\text{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
\textit{tcpstatShow()} \\
\textbf{NAME} \hspace{1cm} \textit{tcpstatShow()} -- display all statistics for the TCP protocol \\
\textbf{SYNOPSIS} \hspace{0.5cm} \texttt{void tcpstatShow (void)} \\
\textbf{DESCRIPTION} \hspace{0.5cm} This routine displays detailed statistics for the TCP protocol. \\
\textbf{RETURNS} \hspace{0.5cm} N/A \\
\textbf{SEE ALSO} \hspace{0.5cm} windsh, Tornado User's Guide: Shell \\

\textit{td()} \\
\textbf{NAME} \hspace{1cm} \textit{td()} -- delete a task \\
\textbf{SYNOPSIS} \hspace{0.5cm} \texttt{void td (void)} \\
\textbf{DESCRIPTION} \hspace{0.5cm} This command deletes a specified task. It is equivalent to the target routine \texttt{taskDelete()}. \\
\textbf{RETURNS} \hspace{0.5cm} N/A \\
\textbf{SEE ALSO} \hspace{0.5cm} windsh, Tornado User's Guide: Shell \\

\textit{tftpInfoShow()} \\
\textbf{NAME} \hspace{1cm} \textit{tftpInfoShow()} -- get TFTP status information \\
\textbf{SYNOPSIS} \hspace{0.5cm} \texttt{void tftpInfoShow} \\
\hspace{1cm} \texttt{(TFTP_DESC * pTftpDesc /* TFTP descriptor */)} \\
\textbf{DESCRIPTION} \hspace{0.5cm} This routine prints information associated with TFTP descriptor \texttt{pTftpDesc}. \\
\textbf{EXAMPLE} \hspace{0.5cm} A call to \texttt{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**

```c
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**

```c
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**

```c
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*
NAME

`tt()` – display a stack trace of a task

SYNOPSIS

```c
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)
666 _logTask +12 : _read (5, 3f8a10, 20)
460 _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
\textbf{w()}

\textbf{NAME}\n
\textit{w()} – print a summary of each task’s pending information, task by task (*)

\textbf{SYNOPSIS}\n
\texttt{void w (}
\begin{verbatim}
    int taskNameOrId /* task name or task ID */
\end{verbatim}
\texttt{)}

\textbf{DESCRIPTION}\n
This routine calls \texttt{taskWaitShow()} in quiet mode on all tasks in the system, or a specified task if the argument is given.

\textbf{RETURNS}\n
N/A

\textbf{SEE ALSO}\n
\texttt{windsh}, \texttt{taskWaitShow()}, \texttt{tw()}, \textit{Tornado User's Guide: Shell}

\textbf{wdShow()}

\textbf{NAME}\n
\textit{wdShow()} – show information about a watchdog

\textbf{SYNOPSIS}\n
\texttt{STATUS \texttt{wdShow}}
\begin{verbatim}
    (\texttt{WDOG_ID \texttt{wdId} /* watchdog to display */})
\end{verbatim}

\textbf{DESCRIPTION}\n
This command displays the state of a watchdog.

\textbf{EXAMPLE}\n
\begin{verbatim}
-> \texttt{wdShow myWdId}
Watchdog Id : 0x3dd46c
State : OUT_OF_Q
Ticks Remaining : 0
Routine : 0
Parameter : 0
\end{verbatim}

\textbf{RETURNS}\n
OK or ERROR.

\textbf{SEE ALSO}\n
\texttt{windsh}, \texttt{browse()}, \textit{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.

wtxregd

NAME
wtxregd – the Tornado service registry

SYNOPSIS
wtxregd [-d directory registryDirectory] [-h elp] [-pd pingDelay]

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 wtstoolAttach() 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 wtstoolInfoQ(), 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*pingNumbers. 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 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,flc 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

### Utilities Reference

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>aoutToBin</td>
<td>extract text and data segments from a BSD a.out object module</td>
<td>435</td>
</tr>
<tr>
<td>aoutToBinDec</td>
<td>extract text and data segments from a.out file on x86</td>
<td>435</td>
</tr>
<tr>
<td>binToAsm</td>
<td>convert a binary file to an assembly file</td>
<td>435</td>
</tr>
<tr>
<td>coffArmToBin</td>
<td>extract text/data segments from ARM COFF object file</td>
<td>436</td>
</tr>
<tr>
<td>coffHex</td>
<td>convert a COFF format object file into Motorola hex records</td>
<td>436</td>
</tr>
<tr>
<td>coffHex960</td>
<td>convert a COFF format object file into Motorola hex records</td>
<td>437</td>
</tr>
<tr>
<td>coffHexArm</td>
<td>convert a COFF format object file into Motorola hex records</td>
<td>437</td>
</tr>
<tr>
<td>coffToBin</td>
<td>extract text and data segments from a COFF object file</td>
<td>438</td>
</tr>
<tr>
<td>elfHex</td>
<td>convert a ELF format object file into Motorola hex records</td>
<td>439</td>
</tr>
<tr>
<td>elfToBin</td>
<td>extract text and data segments from an ELF file</td>
<td>439</td>
</tr>
<tr>
<td>hex</td>
<td>convert an a.out format object file into Motorola hex records</td>
<td>439</td>
</tr>
<tr>
<td>hexDec</td>
<td>convert a.out format object file to Motorola hex records for x86</td>
<td>440</td>
</tr>
<tr>
<td>htmlBook</td>
<td>html Book Index generator</td>
<td>441</td>
</tr>
<tr>
<td>htmlLink</td>
<td>install cross references in HTML documents</td>
<td>443</td>
</tr>
<tr>
<td>makeStatTbl</td>
<td>make a table of status values</td>
<td>444</td>
</tr>
<tr>
<td>makeSymTbl</td>
<td>make a table of symbols</td>
<td>445</td>
</tr>
<tr>
<td>memdrvbuild</td>
<td>filesystem builder for memDrv</td>
<td>446</td>
</tr>
<tr>
<td>munch</td>
<td>extract initializers and finalizers from C++ object modules</td>
<td>447</td>
</tr>
<tr>
<td>refgen</td>
<td>Tornado Reference documentation generator</td>
<td>448</td>
</tr>
<tr>
<td>romsize</td>
<td>compute the size of a ROM image</td>
<td>454</td>
</tr>
<tr>
<td>symgen</td>
<td>Tornado Reference documentation generator</td>
<td>454</td>
</tr>
<tr>
<td>vxencrypt</td>
<td>encryption program for loginLib</td>
<td>455</td>
</tr>
<tr>
<td>vxsize</td>
<td>compute the size of a vxWorks image</td>
<td>456</td>
</tr>
<tr>
<td>xsym</td>
<td>extract the symbol table from a BSD a.out object module</td>
<td>456</td>
</tr>
<tr>
<td>xsymc</td>
<td>extract the symbol table from a COFF object module</td>
<td>457</td>
</tr>
<tr>
<td>xsymcArm</td>
<td>extract the symbol table from an ARM COFF object module</td>
<td>457</td>
</tr>
<tr>
<td>xsymDec</td>
<td>extract the symbol table from an a.out object module for x86</td>
<td>458</td>
</tr>
</tbody>
</table>
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, `format` 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

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
-1 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

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
-1
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.

---

**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

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
-1 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.

hex

NAME
hex – convert an a.out format object file into Motorola hex records

SYNOPSIS
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 than 16M; and S3 records are generated for addresses greater than 16M.

OPTIONS
-1 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

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 than 16M; and S3 records are generated for addresses greater than 16M.

OPTIONS
-1 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:
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>
(<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>} {\<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

makeStatTbl

NAME
makeStatTbl – make a table of status values

SYNOPSIS
makeStatTbl hdr [...]

444
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.
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() and memDrvDeleteFiles(), where mount is the argument to the -d option, with non-alphabetic characters converted to _. This first function mounts the packed files using...
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:

```c
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**

```
wxctcl $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

```
rm68k partialImage.o partialUserImage.o user1.o user2.o \
   | wtxtcl %WIND_BASE%\host\src\hutils\munch.tcl -asm 68k > ctdt.c
```

UNIX

```
rm68k 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 \n   -fno-builtin -c hello.cpp
rm68k hello.o | wtxtcl %WIND_BASE%\host\src\hutils\munch.tcl \n   -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-builtins \n   -c hello.cpp
rm68k hello.o | wtxtcl $WIND_BASE/host/src/hutils/munch.tcl \n   -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] 
```
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):

<table>
<thead>
<tr>
<th>Note</th>
<th>Markup</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>\</code> text to end of line</td>
<td>Comment in documentation.</td>
</tr>
<tr>
<td></td>
<td>‘text ...’ or ‘text ...’</td>
<td>Boldface text.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;...&gt;</code></td>
<td>Italicized text.</td>
</tr>
<tr>
<td>[1]</td>
<td><code>\</code> or <code>/</code></td>
<td>The character <code>\</code>.</td>
</tr>
<tr>
<td>[2]</td>
<td><code>&lt;</code> or <code>&gt;</code> &amp; <code>&lt;</code> or <code>&gt;</code></td>
<td>The characters <code>&lt;</code> or <code>&gt;</code> &amp; <code>&lt;</code> or <code>&gt;</code>.</td>
</tr>
<tr>
<td></td>
<td><code>\</code></td>
<td>The character <code>\</code> within a table.</td>
</tr>
<tr>
<td>[3]</td>
<td><code>\ss</code> ... <code>\se</code></td>
<td>Preformatted text.</td>
</tr>
<tr>
<td></td>
<td><code>cs</code> ... <code>ce</code></td>
<td>Literal text (code).</td>
</tr>
<tr>
<td></td>
<td><code>bs</code> ... <code>be</code></td>
<td>Literal text, with smaller display.</td>
</tr>
<tr>
<td>[4]</td>
<td><code>\is</code> ... <code>\i</code> ... <code>\ie</code></td>
<td>Itemized list.</td>
</tr>
<tr>
<td>[5]</td>
<td><code>\ml</code> ... <code>\m</code> ... <code>\m</code> ... <code>\me</code></td>
<td>Marker list.</td>
</tr>
<tr>
<td>[6]</td>
<td><code>\ts</code> ... <code>\te</code></td>
<td>A table.</td>
</tr>
<tr>
<td></td>
<td><code>\sh</code> text</td>
<td>A subheading.</td>
</tr>
<tr>
<td></td>
<td><code>\tb</code> reference</td>
<td>A reference followed by newline.</td>
</tr>
</tbody>
</table>

**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
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

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>ampersand</td>
<td>bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
</tr>
<tr>
<td>#</td>
<td>octothorpe</td>
<td>bitwise NAND</td>
</tr>
</tbody>
</table>

- **-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 sourceFileList**: Prints reassuring messages to indicate something is happening. 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:

```bash
  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

```bash
  % 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


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

---

**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
The Tornado utility \texttt{servutil.exe} (located in \texttt{c:\tornado\host\x86-win32\bin}) can be used to install and remove Windows NT services.

\textit{Installing a Service}

Figure E-1 shows the \texttt{servutil} dialog box, filled out to install \texttt{Portmapper} as a service.

![Figure E-1 Installing Portmapper as a Service](image)

Figure E-2 illustrates how to fill out the \texttt{servutil} dialog box to install the Tornado registry as a service.

The Service Dependency Name field is filled in with “Tornado PortMapper” to specify that \texttt{Portmapper} must be started before the service manager starts the Tornado registry.
Figure E-2 **Installing the Tornado Registry as a Service**

Figure E-3 illustrates how to remove a service. First select **Remove Service**, then type in the name of the service that you want to remove and click **OK**.

**Removing a Service**

Figure E-3 **Removing a Service**

**NOTE:** To find the names of existing services, click **Service** in the Control Panel.
If you have an Ethernet or other fast IP connection between your development host and target, you can take advantage of the image-download facility that is built into the VxWorks boot ROMs. This facility allows you to establish a completely new VxWorks run-time system on your target when you change the VxWorks configuration or when you link your application statically into the run-time. The boot-ROM download mechanism depends on the Internet transfer protocol known as FTP (file transfer protocol). Thus, to download new run-time images from your Windows host, you must configure and run an FTP server there.

On Windows NT, we recommend that you install (as an NT service) the FTP server provided by the operating system, when you install TCP/IP.

For Windows 95, Tornado includes an FTP server called WFTPD, a product of Texas Imperial Software. Your copy is licensed as part of the Tornado license. The remainder of this appendix discusses how to set up WFTPD as an FTP server on Windows 95.

The WFTPD utility is distributed as c:\tornado\host\x86-win32\bin\wftpd32.exe. If you plan to use FTP as the normal means of booting
VxWorks, it is convenient to start `wftpd32` automatically when you start your Windows host. To do this, add `wftpd32` to Start Menu Programs (Start>Settings>Taskbar>Start Menu Programs).

### F.3 WFTPD Configuration

Before an FTP client can connect to WFTPD, you must specify the following information:

- A user ID. Choose whatever arbitrary name you wish as the user ID for the VxWorks boot ROM. Be sure to specify the same name as the `user (u)` VxWorks boot parameter described in 2.5.4 Description of Boot Parameters, p.41.
- A password for that user ID. Use any memorable arbitrary string, and be sure to specify the same string as the `ftp password (pw)` VxWorks boot parameter described in 2.5.4 Description of Boot Parameters, p.41.
- A home directory. The VxWorks boot ROM does not require this information, but WFTPD refuses to connect to a client unless you specify a home directory. Any directory will do, so long as you permit sufficient disk access for the VxWorks boot ROM to read the boot image on your Windows disk.

To specify this information, carry out the following steps:

1. Open the WFTPD window and select the Users/rights command from the Security menu (Figure F-1).
2. WFTPD displays the User / Rights Security Dialog box shown in Figure F-2. Click the New User button; another dialog box appears where you can enter the user name (also shown in Figure F-2).
3. After you specify the user name and click OK, WFTPD displays a third dialog box where you can specify the password (Figure F-3). Because the password does not display as you type it, you must type it twice to make sure the correct password is recorded.
4. After defining the new user ID and password, be sure to fill in the Home Directory text box before dismissing the User / Rights Security Dialog box by clicking Done.
Figure F-1  **WFTPD Security Menu**

![WFTPD Security Menu](image1)

Figure F-2  **Adding a New User for WFTPD**

![Adding a New User for WFTPD](image2)

Figure F-3  **WFTPD Password Dialog Box**

![WFTPD Password Dialog Box](image3)
Symbols

? (C/Tcl interpreter toggle)  197
   quick access  199
@ command (booting)  44
@ prefix (WindSh)  185

A

About Tornado dialog box  60
add command (debugger)  257
Add New Build Specification window  114
Add Symbols command (Debug menu)  277
   code example  278
address(es), memory
   current, setting   180
add-symbol-file command (debugger)  257
agent, see target agent
agentModeShow()  163
   mode switching, under   170
ALT+F (open recently-used file)  63
ALT+F t (print)  68
ALT+F3 (find)  67
ALT+Fa (save as)  64

APIs
   see also Tornado API Guide; Tornado API Reference
documentation   xxiii
application I/O (debugger)  259
   see also bootable applications; downloadable applications
   bootable   73
   displaying information about (browser)   221–224
   downloadable   72
   loading during debugging   257
   option flags   262
   unloading   240

Application Program Interfaces, see APIs
application wizard   77
arp command   56
Arrange icons command   51
arrays (WindSh)   183
arrow keys   66
assignments (WindSh)   180
Attach command (ALT+F6) (Debug menu)   246
attach system command (debugger)   263
Attribute Pane checkbox (Options window)   283
attribute panes   61
authorizations, see security
auto scaling   107

B

-B option (target server)   140
b()   164
system mode, in 170
back ends, communications 132
Back Trace command (ALT+7) (Debug menu) 252
backplane
  installing boards in 23
  processor number 41
bd() 163
bdall() 163
bh() 164
board support packages (BSP) 74
  boot media 22
  creating projects 92
  documentation xxv
  pre-Tornado 2.x versions, using 92
boot media 22
boot programs
  boot parameters, configuring 124
  building 124
  configuring 123–126
  creating new 46
  serial connection, configuring for 29
  TSFS, for 125
boot ROMs
  see also NetROM ROM emulator
  emulators, substituting ROM 22
  installing 22
  system image (VxWorks), building a new 461
bootable applications 73
  auto scaling 107
  creating 106–107
  initialization routines, adding application 107
bootChange() 163
booting 38–47
  @ command 44
  commands 41
  display, boot 44
  parameters 39–44
  command-line format 45
  displaying current, at boot time 40
  nonvolatile RAM, effect of 46
  setting 39
rebooting 47
TFTP, requiring 46
troubleshooting 54
TSFS, using 46
CTRL+BREAK (interrupt) 173
break thread command (debugger) 266–267
breakpoints
  multiple, setting 241
  setting 243
  shell commands for 164
  temporary, setting 243
  threads, setting in 266–267
Breakpoints command (Debug menu) 241
browse() 165
browser (HTML)
  specifying 304
browser (Tornado) 207–232
  application module information, displaying 221–224
  behavior of, controlling 210
  buttons 210
  class information, displaying 220
  CPU utilization, reporting 224
  customizing 232
  interrupt/exception vector table, displaying 226
  limitations 227
  memory partition information, displaying 219
  memory usage data, displaying 214
  message queue information, displaying 218
  object information, displaying 215–221
  semaphore information, displaying 217
  spy utility 224
  stack checks, displaying 225
  starting 208
  target information, displaying 211
  task information, displaying 216
  tasks, listing running 213
  troubleshooting with 227–231
  updating displays 210
  watchdog information, displaying 219
  window display, choosing 208
Browser command (Tools menu) 208
Browser Configuration dialog box 210
browser.tcl 232
BSP, see board support packages
Build menu
  adding/removing custom commands 301–303
Build Output command (Window menu) 51
Build Output window 87
build specifications 107–115
  assembler options, specifying 112
  changing 108
  compiler options, specifying 111
  creating new 114
  current build, selecting for 115
  link order, specifying 112
  linker options, specifying 113
  makefile macros, specifying 110
  makefile rules, working with 109
  property sheets, working with 108–113
Build toolbar
  buttons 88
  displaying 88
bus, see VMEbus
buttons
  browser, Tornado 210
  Build toolbar 88
  debugger 235–237
  standard toolbar 60
  Tornado Launch toolbar 49
  WindView event triggering 50

C

C interpreter (WindSh) 174–190
  address, setting current 180
  arrays 183
  assignments 180
  commands, built-in 185
  comments 182
  data types, handling 174
  function calls 177
  limitations 184
  literals 176
  operators 177
  pointers 183
  redirecting I/O 187–190
  startup scripts 190
  statements, handling 176
  strings 182
    declaring 175
    subroutines as commands 179
    tasks
      current, setting 180
      referencing 179
    Tcl interpreter
      quick access to 199
      toggling to 197
    variable references 176
c() 163
  current task, using 180
  system mode, in 170
C++ support 190–193
  demangling function names 192
  host and target strategies, using different 165
  overloaded function names, reconciling 191
  shell commands for 164
  cables, connecting 24
call command (debugger) 258
CAPS LOCK key 50
Cascade command 51
cd() 160
character arrays, see strings
characters, control, see control characters
checkStack() 158
classes (kernel objects)
  displaying information about (browser) 220
classShow() 165
Close All command 51
Close command (File menu) 64
code
  Tcl conventions 335–347
  unsupported, directory for 324
code examples
  debugger extensions 278–279
  NetROM startup file 33
  system mode debugging 170
colors, changing window 289–291
command-line parameters 45
commands, built-in (WindSh)
  C++ support 164
  debugging 162
  network information, displaying 167
  system information, reporting 159
  system, modifying the 162
  target routines, differentiating from 185
task information, reporting 157
tasks, managing 156
VxWorks objects, displaying 165
comments (WindSh) 182
compiling, see application modules; GNU Make
User’s Guide; GNU ToolKit User’s Guide
cross-development 3–12
CrossWind 233–279
see also debugger; debugging; GDB User’s Guide
crosswind.tcl 275
CTRL+A (select all) 65
CTRL+BREAK (interrupt) 173
CTRL+C
  copy character
  editor, from 65
  shell, in 187
  interrupt (console) 173
CTRL+D
  completing symbol and path names 152
  function synopsis, printing 152
  quitting 150
CTRL+END (end of file) 66
CTRL+F4 (close file) 64
CTRL+G (go to) 66
CTRL+H (delete) 187
CTRL+HOME (beginning of file) 66
CTRL+N (new file) 61
CTRL+O (open file) 63
CTRL+P (print) 68
CTRL+S (save file) 64
CTRL+SHIFT+X (reboot) 168
CTRL+U (delete line) 187
CTRL+V (paste)
  editor, from 65
  shell, in 187
CTRL+W (function reference page) 153
CTRL+X
  cut 65
  reboot 47
CTRL+Z (undo edit) 65
cursor, text 64
  navigating 66
customer services (WRS) 305–313
Customize Tools dialog box 294–300
  macros, using 296
customizing Tornado 281–304
see also initialization files
Build menu 301–303
colors, changing window 289–291
destructor attributes 291
download options, setting 281
editor 283–285
alternate, specifying 284
fonts, changing screen 289–291
registry, Tornado 292
Tcl files, using 303
Tools menu 293–300
alternate editor command, creating (example) 299
binary utilities commands, creating (example) 300
macros for custom commands 296
version control command, creating (example) 298
Web link to WRS, creating (example) 300
Tornado 1.0.1 compatibility, setting 286
version control, configuring 287
customizing VxWorks 90–106
booting VxWorks 106
building VxWorks 105
image type, selecting 103
project files, creating 94
projects, creating 90
VxWorks components
configuring 96–103
dependencies, calculating 98
descriptions of 97
excluding 97–102
finding 97
including 97–102
parameters, accessing 102
size of components only, estimating image 103
system image size, recalculating 101
warnings, conflict 101
Cut command (Edit menu) 65

D

d() 161
current address, using 180
data types 174
data variables 176
Debug menu 235
Add Symbols command 277
Attach command 246
Back Trace command 252
Breakpoints command 241
Continue command 244
Detach command 247
Download command 262
Interrupt Debugger command 241
Memory command 253
Registers command 251
Run command 240
Source Search Path command 239
Step Into command 245
Step Out command 244
Step Over command 245
Stop Debugging command 238
Toggle Breakpoint command 243
Toggle Temp. Breakpoint command 243
Variables command 250
Watch command 248
debugger (CrossWind) 233–279
see also debugging; GDB User’s Guide
application I/O 259
auto-attaching to tasks 292
behavior, controlling 261–262
buttons 235–237
command-line interface 256–262
customizing 291
console window 256–262
code examples 278–279
GDB Tcl interpreter 267–274
customizing 291
dependencies, calculating 98
expressions, evaluating 270
GDB command line, using 268
GDB facilities, invoking 270–272
GUI Tcl interpreter, switching to 276
I/O file descriptors, closing 272
linked lists, traversing 273–274
list of Tcl elements, returning 271
redireciting I/O 271
routine names, returning 272
scripts, debugging GDB-based 269
symbols
returning local 272
testing for 272
verbose error printing 269
.gdbinit, disabling 261
GUI 234–237
GUI Tcl interpreter 274–279
   extensions, creating 277–279
   code examples 278–279
   GDB Tcl interpreter, switching to 276
hexadecimal, displaying 291
initialization files 256
   customizing GUI 275–276
   internal parameters, changing 261–262
menu commands 235–237
quitting 238
redirecting I/O 259
source search path, setting 239
starting 238
toolbar 235–237
unloading modules 240
documentation xxv
   conventions xxvi–xxvii
   cross-references, format of xxvi
   online xxv
   WTX protocol requests, sending 260
DEL key (delete) 65
   projects, removing 90
delete character
BACKSPACE 187
CTRL+H 187
Delete command (Edit menu) 65
delete-line character (CTRL+U) 187
demangling, see name demangling
demo host directory 323
Dependents dialog box 84
Detach command (Debug menu) 247
development environment 2–12
des() 159
directory tree 317–324
disassembling (I()) 161
Disassembly command (View menu) 254
docs directory 317
see also online documentation
documentation xxi–xxvii
VxWorks xxiv–xxv
Domain Name Service (DNS) 21
Download command (Debug menu)
  option flags 262
download options 281
downloadable applications 72
  building 83–89
  errors and messages, displaying 87
  specifications, providing 85
  creating 75–90
  project files
    adding 81
    closing 83
    creating 80
    linking 81
    opening 83
    properties, modifying 82
    property information, displaying 82
    removing 82
    saving 83
  projects
    adding 89
    creating new 76
    removing 89
downloading 89
downcl command (debugger) 276
DSM_HEX_MOD 155
dual mode 25

E

Edit menu 65–67
  Copy command 65
  Cut command 65
  Delete command 65
  Find command 67
  Find Next command 67
  Find Previous command 67
  Go To command 66
  Next Error/Tag command 66
  Paste command 65
  Previous Error/Tag command 66
  Replace command 67
  Select All command 65

Undo command 65
  edit mode, shell (WindSh) 193–195
  input mode, toggling to 187
editor (external)
  alternate
    Tools menu command for, creating
      (example) 299
  specifying 284
editor (Tornado) 59–69
  see also buttons; Edit menu; keyboard shortcuts
  copying 65
  customizing 69
  deleting 65
  errors, going to 66
  insert mode 65
  navigating 66
  overtype mode 65
  pasting 65
  preferences, setting 283–285
  replacing text 67
  save options 284
  searching 67
  selecting all 65
  tabs, setting 284
  undoing edits 65
END (end of line) 66
Enhanced Network Driver (END) 28
environment variables, host
  HOME 325
  shell behavior, controlling 154
environment, see cross-development; development environment; host environment; target environment
errorInfo global variable (debugger) 270
errors
  Tcl 334
  unwinding 334
ESC key (edit/input mode toggle) 187
  event triggering, starting 50
exit() 150
expressions, C language (WindSh) 176–180
  see also data variables; function calls; literals; operators
External Dependencies folder 80
makefile dependencies, listing 84
external mode, see system mode

F

F3 (find next) 67
F4 (next error) 66

File menu 61
  Add Project to Workspace command 89
  Close command 64
  New Project command 89
  Open command 63
  Page Setup command 68
  Print command 68
  Save As command 64
  Save command 64

file tree 317–324
files
  see also editor; navigation
  attribute panes 61
  closing 64
  opening 63
  page, setting up 68
  printing 68
  saving 64
  saving as 64
  types 61

Find command (Edit menu) 67
Find dialog box 67
Find Next command (Edit menu) 67
Find Object dialog box 97
Find Previous command (Edit menu) 67

FIXED.TXT xxii
fonts, changing screen 289–291
frame command (debugger) 260
Free Software Foundation documentation xxv
FTP (File Transfer Protocol)
  password, user (boot parameter) 43
  server (WFTPD) (Windows 95) 461–462
  configuring 462
  starting 461
ftp password 462
function calls (WindSh) 177
  parentheses, omitting 179

reference page, displaying HTML 153
synopses, printing 152

G

-g compiler option
  limitations 111
GDB (GNU debugger)
  see also debugger; debugging
  customizing with Tcl 257
  initialization files 256
gdb command (debugger) 270
gdb.tcl 257
gdbEvalScalar command (debugger) 270
gdbFileAddrInfo command (debugger) 271
gdbFileLineInfo command (debugger) 271
.gdbinit 256
gdbIOClose command (debugger) 272
gdbIORedirect command (debugger) 271
gdbLocalsTags command (debugger) 272
gdbStackTraceTags command (debugger) 272
gdbSymbol command (debugger) 272
gdbSymbolExists command (debugger) 272
GNU binary utilities, working with 300
GNU ToolKit 6
  see also linking; GNU ToolKit User’s Guide
Go To command (Edit menu) 66

H

h target directory 321
h() 160
header files
  Tornado tools 318
  VxWorks-supplied 321
help
  context-sensitive xxv
  workspace, in 75
online xxv
  reference documentation 97
Help menu
  About Tornado command 60
Manuals Contents command xxv
Manuals Index command xxv
help set command (debugger) 261
help() 160
history facility, shell 193
HOME host environment variable 325
HOME key (beginning of line) 66
host communication modes 24
directory and file tree 318–319
network software
  configuring 20–21
  initializing 20
  system names and addresses, establishing 21
setting up 15–17
host directory 318–319
host shell, see shell
host-os host directory 318
hosts file (Windows) 21
hostShow() 167

i() 158
  system mode, in 170
icmpstatShow() 167
ifShow() 167
include host directory 318
INCLUDE_TSFS_BOOT_VIO_CONSOLE 125
INCLUDE_WDB_BANNER 120
INCLUDE_WDB_START_NOTIFY 120
INCLUDE_WDB_TTY_TEST 121
INCLUDE_WDB_USER_EVENT 120
INCLUDE_WDB_VIO 120
info threads command (debugger) 264
inhibit-gdbinit (debugger) 261
initialization files 325
initializing
  network software, host 20
input mode (WindSh) 187
INSERT key 65
insertion point 64
installing

see also Tornado Getting Started Guide
boot ROMs 22
target boards 23
  in backplane 23
Internet, see World Wide Web
Interrupt Debugger command (ALT+SHIFT+F5) (Debug menu) 241
interrupt key
  console (CTRL+C) 173
  Tornado (CTRL+BREAK) 173
interrupt/exception vector table 226
intVecShow() 166
I/O
  redirecting
    C interpreter 187–190
    debugging, during 259
    shConfig, using 151
    virtual 11
iosDevShow() 166
iosDrvShow() 166
iosFdShow() 166
ipstatShow() 166
iStrict() 157

jumps 22

kernel 4
keyboard settings 50
keyboard shortcuts
  ALT+3 (display symbol values) 248
  ALT+4 (display local variables) 250
  ALT+5 (display register values) 251
  ALT+6 (target memory) 253
  ALT+7 (back trace) 252
  ALT+D h (source search path) 239
  ALT+F (open recently-used file) 63
  ALT+F t (print) 68
  ALT+F3 (find) 67
ALT+F6 (attach debugging) 246
ALT+Fa (save as) 64
ALT+SHIFT+F5 (interrupt) 241
arrow keys 66
BACKSPACE (delete) 187
CTRL+A (select all) 65
CTRL+BREAK (interrupt) 173
CTRL+C
    copy
editor, from 65
    shell, in 187
    interrupt (console) 173
CTRL+D
    completing symbol and path names 152
    function synopsis, printing 152
    quitting 150
CTRL+END (end of file) 66
CTRL+F4 (close file) 64
CTRL+F5 (step over) 245
CTRL+G (go to) 66
CTRL+H (delete) 187
CTRL+HOME (beginning of file) 66
CTRL+N (new file) 61
CTRL+O (open file) 63
CTRL+P (print) 68
CTRL+S (save file) 64
CTRL+SHIFT+X (reboot) 168
CTRL+U (delete line) 187
CTRL+V (paste)
editor, from 65
    shell, in 187
CTRL+W (function reference page) 153
CTRL+X
    cut 65
    reboot 47
CTRL+Z (undo edit) 65
DEL (delete) 65
END (end of line) 66
ESC key (input/edit mode toggle) 187
F10 (step into) 245
F3 (find next) 67
F4 (next error) 66
F5 (continue debugging) 244
F6 (run task) 240
F8 (temporary breakpoint) 243
F9 (toggle breakpoint) 243
HOME key (beginning of line) 66
line editing, shell (WindSh) 193–195
navigation, for file 66
PG DN (page down) 66
PG UP (page up) 66
SHIFT+F3 (find previous) 67
SHIFT+F4 (previous error) 66
SHIFT+F5 (step out) 244
kill, see stopping

L

l() 161
    current address, using 180
Launch Browser dialog box 208
Launch Shell dialog box 149
ld() 164
    path name orthography 183
LD_CALL_XTORS 155
LD_COMMON_MATCH_ALL 155
LD_PATH 155
LD_SEND_MODULES 155
lib target directory 321
LICENSE.TXT 318
linking
    see also GNU ToolKit User’s Guide
dynamic 164
linkSyms.c 94
list (Tcl) 330
literals (WindSh) 176
    see also strings
lkAddr() 160
lkup() 160
load command (debugger) 257
Lock on Startup button (Configure Target Servers dialog box) 146
locking (target server) 146
Log Console window (target server) 141
ls() 160
M

\texttt{m()} 164
Macros tab (property sheet) 110
makefile dependencies, calculating 84
\texttt{Makefile} file 95
Manage command (Tools\textgreater Target Server menu) 143–146
Manage Target Servers dialog box 143–146
authorizations, setting 146
Select Action drop-down list 144
mangling, \textit{see} name demangling
Manuals Contents command (Help menu) xxv
Manuals Index command (Help menu) xxv
memory
  cache size, changing target server 135
  fragmentation, troubleshooting 228
  leaks, troubleshooting 227
  target, manipulating 164
  usage, displaying data about 214
Memory command (\texttt{ALT}+\texttt{6}) (Debug menu) 253
memory partitions
  displaying information about (browser) 219
\texttt{memPartShow()} 166
\texttt{memShow()} 166
message queues
  displaying information about (browser) 218
Mixed Source and Disassembly command (View menu) 254
\texttt{moduleIdFigure()} 166
\texttt{moduleShow()} 166
\texttt{mqPxShow()} 166
\texttt{mRegs()} 163
\texttt{msgQShow()} 166

N

name demangling (C++) 192
navigation 66
  \texttt{Go To} command 66
NetROM ROM emulator 22
  back end connection, as 30–37
code example 33
  configuring 32–35
target agent for, configuring 32
target server, starting 35
troubleshooting 35–37
\texttt{netstat} command 56
\texttt{netstatShow()} 167
networks
  configuring simple 20–21
target-server back end connection, as 27
New command (File menu) 61
Next Error/Tag command (Edit menu) 66
\texttt{NUM LOCK} key 50

O

Object Modules folder 80
object modules, \textit{see} application modules
online documentation xxv
  function reference pages, displaying 153
Open command (File menu) 63
operators (WindSh) 177
  using \texttt{<} and \texttt{>} 187
optional VxWorks products 13
  VxMP 5
  VxSim, target simulator 8
  VxVMI 4
  WindView 8
Options command (Tools menu) 281
Options window 281–293

P

Page Setup command (File menu) 68
Page Setup dialog box 69
paste character (\texttt{CTRL}+\texttt{V})
  editor, from 65
  shell, in 187
Paste command (Edit menu) 65
path names
  documentation, in xxvi
  shell and target server 196
typing conventions 183
\texttt{period()} 157
Tornado 2.0
User's Guide

PG DN (page down) 66
PG UP (page up) 66
ping command 55
pointers (WindSh) 183
  arithmetic, handling 184
Previous Error/Tag command (Edit menu) 66
primitives (WindSh)
  debugging 261
Print command (File menu) 68
Print dialog box 68
printErrno() 160
printLogo() 160
priority inversion troubleshooting 228
prjComps.h 95
prjConfig.c 95
prjParams.h 95
problem lists xxii
PROBLEMS.TXT xxii
projectName.wpj 95
projects 71–126
  see also bootable applications; customizing VxWorks; downloadable applications; target agent
  adding 89
  boot programs, using new 123–126
  creating
    based on existing project 91
    bootable applications, for 106–107
    downloadable applications, for 76
    customized VxWorks, for 90
  removing 90
pwd() 160

Q
quit() 150
quitting
  see also stopping
  debugger 238
  shell sessions 150

R
README.HTML 318
README.TXT 318
reboot character (CTRL+X) 47
reboot() 168
rebooting 47
  shell, from the 168
  redirection, I/O (WindSh) 187–190
  scripts, using 188
  shConfig, using 151
reference pages
  tools, Tornado 349
  utilities 433
Registers command (ALT+s) (Debug menu) 251
registry, Tornado 15–17
  see also wxregd
    changing 17
    customizing 292
release notes xxii
  see also Tornado Release Notes
repeat() 157
Replace command (Edit menu) 67
Replace dialog box 67
resource host directory 319
ROM, see boot ROMs; NetROM ROM emulator
romInit.s 95
romStart.c 95
routestatShow() 167
routines, see function calls; commands, built-in
run command (debugger) 258
  resetting I/O 259
  system mode, not in 263
Run command (F6) (Debug menu) 240
Index

S

s() 164
  current task, using 180
  system mode, in 170
Save As command (File menu) 64
Save command (File menu) 64
SCROLL LOCK key 50
  searching 67
security
  target server authorizations 145
Select All command (Edit menu) 65
semaphores
  displaying information about (browser) 217
semPxShow() 166
semShow() 166
serial lines
  target-server back end connection, as 28–30
testing 29
servutil.exe service utility (Windows NT) 459–460
set command (debugger) 261–262
SETUP directory 317
setup, see configuration
setup.log log file 318
SH_GET_TASK_IO 155
share directory 317
shConfig (Tcl)
  redirecting I/O 151
  shell environment variables, setting 154
shell (WindSh) 147–205
  see also C interpreter; C++ support; commands, built-in
  C control structures 185
  C interpreter 174–190
  C++ support 190–193
  calculating in 154
  commands, built-in 156–167
    differentiating from target routines 156
    list of 185
  components 203
  control characters 186
  data conversion 153
  debugging from 148
  edit mode 193–195
    input mode, toggling to 187
  environment variables, setting 154
  function synopsis, printing 152
  history facility 193
  input mode 187
  interpretation, layers of 204
  interrupting 173
  line editing 193–195
  linker, dynamic 164
  path names
    orthography 196
    typing 152
  preprocessor macros 185
  rebooting 168
  redirecting I/O 151
  starting 149
  startup scripts 190
  stopping 150
  system mode debugging 169–172
  target from host, controlling 201
  target routines from, running 167
  target symbol names, typing 152
  Tcl interpreter 197–201
    initializing with 200
    starting 197
  types, compound and derived 185
Shell command (Tools menu) 149
shellHistory() 160
shellPromptSet() 160
SHIFT+F3 (find previous) 67
SHIFT+F4 (previous error) 66
show command (debugger) 261
show routines (shell) 165
show() 165
shParse utility (Tcl) 199
simulator, target (VxSim) 8
  starting 49
single-stepping
  shell commands for 164
smMemPartShow() 166
smMemShow() 166
so() 163
source code directories (VxWorks) 321
Source command (View menu) 254
source control, see version control
source search path 239

477
Source Search Path command (ALT+D h) (Debug menu) 239

sp() 156
  system mode, in 170
sps() 156
spy utility (browser) 224
  see also spyLib(1)
  reporting mode, selecting 224
src target directory 321
src/demo host directory 323
stacks
  overflow, troubleshooting 228
  usage, checking (browser) 225
standalone systems, using 19
standard toolbar 60
starting
  browser, Tornado 49
  debugger 49
  event triggering (WindView) 50
  shell, Tornado 49
  target servers 51
  Tornado 47–51
  VxSim target simulator 49
  WindView 49
state-information files 325
statements (WindSh) 176
status line (Tornado) 50
Step Into command (F10) (Debug menu) 245
Step Out command (SHIFT+F5) (Debug menu) 244
Step Over command (CTRL+F5) (Debug menu) 245
Stop Debugging command (Debug menu) 238
stopping
  shell (WindSh) 150
  target servers 141
strings 182
  declaring 175
subroutines, see function calls; commands, built-in
Support command (Tools menu) 306
support, see technical support; customer services
symbol table
  synchronizing 135
  target server, configuring 134
sysALib.s 95
sysLib.c 95
sysResume() 163

mode switching, under 170
sysStatusShow() 163
  mode switching, under 170
sysSuspend() 163
  mode switching, under 169
system controller 23
system image
  see also customizing VxWorks
  boot image options 124
  VxWorks
    boot ROMs, building with 461
    creating custom 90–106
system mode 25
  debugging 263–267
  code example 170
  initiating 263
  shell, from the 169–172
  threads, managing 264–267
  shell commands, using 169
  target commands, using 169
  target agent 9

T

-T option (Tcl) 197
target
  see also browser (Tornado)
  communication modes 24
  configuring 17–19
  directory and file tree 319–324
  information about, displaying (browser) 211
  monitoring state of (browser) 207–232
  target agent 9
  communication modes 25
  END drivers, configuring 117
  exception hooks, configuring for 121
  kernel, starting before 121
  NetROM, configuring for 32
    macros, specifying 117
  networks, configuring for 119
  scaling 120
  serial connections, configuring for 119
  simulators, configuring integrated target 117
  system mode 9
  task mode 9
Index

TyCoDrv connections, configuring 120
target board
  backplane, installing boards in 23
  cables, connecting 24
  configuring 21–24
  installing 23
  jumpers, setting 22
  processor number 41
  serial port 24
  terminal emulator 24
target directory 319–324
target environment 4–5
Target Information window 211
target name (tn) (boot parameter) 43
target server 127–146
  see also WTX protocol
  authorizations 145
  back ends 132
  configuring 128–140
    command line, using a 140
    options, specifying 131–138
  connections
    END (Enhanced Network Driver) 28
    network 27
    ROM emulator 30–37
    serial line 28–30
  core file 134
  icon 141
  information about, displaying (browser) 211
  locking 146
  log output, displaying 141
  memory cache size, changing 135
  rebooting 145
  reserving 146
  security 145
  selecting 142
  sharing 145
  starting 51
  status information, displaying 52
  stopping 141
    Manage Target Servers dialog box 145
  symbol table 134
    synchronizing 135
  troubleshooting 56
  un-registering 144
  un-reserving 145
  virtual consoles, creating 137
Target Server File System (TSFS) 136
  boot program for, configuring 125
Target wtx command (debugger) 260
task mode 25
  debugging 263
  target agent 9
taskCreateHookShow() 165
taskDeleteHookShow() 165
taskIdDefault() 157
taskIdFigure() 158
taskRegsShow() 165
taskShow() 165
taskSwitchHookShow() 166
taskWaitShow() 166
Tcl (tool command language) 327–335
  arithmetic expressions 332
  arrays, associative 331
  browser initialization files 232
  C applications, integrating with 335
  coding conventions 335–347
    code layout 340–343
    declarations 339
    module layout 335
    naming 343
    outside procedures 338–339
    procedures 336–338
    programming style 344–347
    command substitution 331–332
    control structures 334
    customizing
Tornado 2.0
User’s Guide

browser, Tornado 232
debugger 274–279
GDB 257
initialization files 303
shell 200
debugger (CrossWind)
    GDB interpreter 267–274
    GUI interpreter 274–279
error handling 334
files 332
formatting 332
I/O 332
linked lists, traversing 273–274
lists 330
procedures, defining 333–334
shell (WindSh) 197–201
C interpreter
    quick access from 199
toggling to 197
initialization files 200
target, controlling the 198
Tcl interpreter, starting 197
Tk graphics library 328
unwinding 334
variables 329–330
tcl command (debugger) 268–269
tcl host directory 319
Tcl_CreateCommand() 335
tcldebug command (debugger) 269
tclerror command (debugger) 269
-Tclmode option (Tcl) 197
tclproc command (debugger) 268
TCP/IP
    Tornado tools, for 19
tcpstatShow() 167
td() 157
technical support (WRS) 305–313
    filing Technical Support Requests (TSR) 308–311
terminal characters, see control characters
terminating, see stopping
text cursor 64
    navigating 66
tftpInfoShow() 167
tgtsvr.exe program 140

options, setting 140
third-party products 13
this menu command (debugger: C++) 278
code example 279
thread command (debugger) 265
thread IDs 264
    breakpoints and system context 267
    managing 264–267
    summary information, displaying 264
    switching
        explicitly 265
        implicitly 267
tl() 158
    current task, using 180
Tile command 51
Tk graphics library (Tcl) 328
Toggle Breakpoint command (F9) (Debug menu) 243
Toggle Temp. Breakpoint command (F8) (Debug menu) 243
toolbars 48
    see also buttons
        Debug 235–237
        standard 60
Tornado Launch 48
toolchain 73
Tools menu
    adding/removing custom commands 294–300
    Browser command 208
    Customize command 293
customizing 293–300
    alternate editor, for 299
    binary utilities, for 300
    version control, for 298
    Web link to WRS, for 300
Debugger command 238
    No Custom Tools placeholder 293
    Options command 281
Shell command 149
    Support command 306
    Target Server submenu
        Manage command 143–146
target server, selecting a 142
tools, development 4–9
  see also browser; debugger; editor; project
  facility; shell; target server
  adding/removing Tools menu
  commands 294–300
  communications, managing 127
  TCP/IP, using 19
  Tornado settings, changing 281–293
  using, guidelines for 4
  tools, Tornado (reference pages) 349
  Tornado 1.0.1 compatibility 286
  Tornado Launch toolbar
    Browser button 208
    buttons 49
    debugger, starting 238
    shell, starting a 149
  Tornado Online Manuals
    12
  Tornado Registry icon 16
  Tornado Registry window 16
  Tornado.tcl 303
  torVars.bat 140
  tr() 157
  training classes (WRS) 14
  troubleshooting 53–57
    booting problems 54
    hardware configuration 54
    memory fragmentation 228
    memory leaks 227
    NetROM ROM emulator connection 35–37
    priority inversion 228
    stack overflow 228
    target-server problems 56
  tfl() 157
  TSFS, see Target Server File System
  tt() 158
  tty command (debugger) 259
  tto() 158

U

  udpstatShow() 167
  Undo command (Edit menu) 65
  unld() 163
  unload command (debugger) 258
  unloading modules 240
  unsupported target directory 324
  unwinding (Tcl) 334
  uptcl command (debugger) 276
  User Profile dialog box 306
  userAppInit.c 95
    initialization calls, adding 107
  utilities, Tornado (reference pages) 433

V

  -V option (target server) 140
  Variables command (ALT+4) (Debug menu) 250
  variables, see data variables; environment variables, host; Tcl variables
  version control 287
    Uncheckout command, creating (example) 298
  version number, displaying Tornado 60
  version() 160
  View menu
    Disassembly command 254
    Mixed Source and Disassembly command 254
    Source command 254
  virtual console 137
  virtual I/O 11
    consoles, creating dedicated 137
  VMEbus
    backplane, installing boards in 23
    system controller 23
  VX_NO_STACK_FILL
    checkStack(), using 158
  VxMP (option) 5
  VxSim 8
    starting 49
  VxVMI (option) 4
  VxWorks 4–5
    see also VxWorks Network Programmer’s Guide;
    booting 38–47
    configuring 17–19
    documentation xxiv–xxv
    optional products 13
      VxMP 5
VxSim, target simulator 8
VxVMI 4
WindView 8
rebooting 47

W

\text{w(\ )} 158
Watch command (ALT+3) (Debug menu) 248
watchdogs
displaying information about (browser) 219
WDB NetRom back end (target server) 133
WDB Pipe back end (target server) 133
WDB RPC back end (target server) 133
WDB Serial back end (target server) 133
WDB\_BP\_MAX 121
WDB\_NETROM\_INDEX 117
WDB\_NETROM\_MTU 117
WDB\_NETROM\_NUM\_ACCESS 117
WDB\_NETROM\_POLL\_DELAY 118
WDB\_NETROM\_ROMSIZE 118
WDB\_NETROM\_TYPE 118
WDB\_NETROM\_WIDTH 118
wdShow() 166
WFTPD FTP server (Windows 95) 461–462
\text{configuring} 462
starting 461
\text{.wind} directory 325
wind kernel, see kernel
Wind River Users Group 313
WIND\_UID 145
windHelp.tcl 304
Window menu 50
Arrange Icons command 51
Build Output command 51
Cascade command 51
Close All command 51
Tile command 51
windows
active 64
colors, changing display 289–291
displaying currently available 51
editing in 64–67
fonts, changing screen 289–291
managing 50
navigating within 66
settings, changing 283
Windows NT
service manager 459–460
WindSh 147–205
see also C interpreter; C++ support; shell
windsh 149
-Tclmode, using 197
windsh.tcl 200
WindView 8
event triggering, starting 50
starting 49
workspace 73
adding project to 89
Workspace window 74
project files, working with 80
workspaceName.wsp 95
World Wide Web
links to WRS 312
Tools menu link to WRS, creating
(example) 300
\text{writeDebugInput} command (debugger) 276
WTX protocol 260
see also target server
requests, sending 260
\text{wtx\_ignore\_exit\_status} (debugger) 262
\text{wtx\_load\_flags} (debugger) 262
\text{wtx\_load\_path\_qualify} (debugger) 262
\text{wtx\_load\_timeout} (debugger) 262
\text{wtx\_task\_priority} (debugger) 262
\text{wtx\_task\_stack\_size} (debugger) 262
\text{wtx\_tcl} procedures 198
\text{wtx\_tool\_name} (debugger) 262