<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Setup and Startup</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>Editor</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>Projects</td>
<td>75</td>
</tr>
<tr>
<td>5</td>
<td>Command-Line Configuration and Build</td>
<td>151</td>
</tr>
<tr>
<td>6</td>
<td>VxSim</td>
<td>197</td>
</tr>
<tr>
<td>7</td>
<td>Shell</td>
<td>219</td>
</tr>
<tr>
<td>8</td>
<td>Target Server</td>
<td>281</td>
</tr>
<tr>
<td>9</td>
<td>Browser</td>
<td>303</td>
</tr>
<tr>
<td>10</td>
<td>Debugger</td>
<td>329</td>
</tr>
<tr>
<td>11</td>
<td>Building VxDCOM Applications</td>
<td>377</td>
</tr>
<tr>
<td>12</td>
<td>Customization</td>
<td>399</td>
</tr>
<tr>
<td>A</td>
<td>Directories and Files</td>
<td>427</td>
</tr>
<tr>
<td>B</td>
<td>Makefile Details</td>
<td>441</td>
</tr>
<tr>
<td>C</td>
<td>Tcl</td>
<td>447</td>
</tr>
<tr>
<td>D</td>
<td>Coding Conventions</td>
<td>455</td>
</tr>
<tr>
<td>E</td>
<td>Windows NT Service Manager</td>
<td>489</td>
</tr>
<tr>
<td>F</td>
<td>FTP Server</td>
<td>491</td>
</tr>
<tr>
<td>G</td>
<td>VxWorks Initialization Timeline</td>
<td>495</td>
</tr>
</tbody>
</table>
## Overview

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

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

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

1.4 Tornado Host IDE ............................................................................................. 5

1.5 Host-Target Interface ....................................................................................... 9

1.6 Customer Services ............................................................................................ 12
## 2 Setup and Startup

### 2.1 Introducing Tornado

### 2.2 Setting Up the Host

### 2.3 Setting Up the Default Target Hardware

#### 2.3.1 Default Target Configuration

#### 2.3.2 Networking the Host and Target

- Initializing the Host Network Software
- Establishing the VxWorks Target Name and IP Address

#### 2.3.3 Configuring the Target Hardware

- Setting Up a Boot Mechanism
- Setting Board Jumpers
- Board Installation and Power
- Connecting a Serial Cable for the Terminal Emulator
- Connecting a Cable for the Ethernet Connection

### 2.4 Host-Target Communication Configuration

#### 2.4.1 Network Connections

- Configuring the Target Agent For Network Connection

#### 2.4.2 END Connections

- Configuring the Target Agent For END Connection

#### 2.4.3 Serial-Line Connections

- Configuring the Target Agent For Serial Connection
- Configuring the Boot Program for Serial Connection
- Testing the Connection
- Starting the Target Server

#### 2.4.4 The NetROM ROM-Emulator Connection

- Configuring the Target Agent for NetROM
- Configuring the NetROM
- Starting the Target Server
- Troubleshooting the NetROM ROM-Emulator Connection
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.5 The Transparent Mode Driver (TMD)</td>
<td>37</td>
</tr>
<tr>
<td>Configuring the Target Agent for TMD</td>
<td>38</td>
</tr>
<tr>
<td>Configuring visionICE II/visionPROBE II</td>
<td>38</td>
</tr>
<tr>
<td>Starting the Target Server</td>
<td>39</td>
</tr>
<tr>
<td>2.5 Booting VxWorks</td>
<td>41</td>
</tr>
<tr>
<td>2.5.1 Default Boot Process</td>
<td>41</td>
</tr>
<tr>
<td>2.5.2 Entering New Boot Parameters</td>
<td>42</td>
</tr>
<tr>
<td>2.5.3 Boot Program Commands</td>
<td>44</td>
</tr>
<tr>
<td>2.5.4 Description of Boot Parameters</td>
<td>45</td>
</tr>
<tr>
<td>2.5.5 Booting With New Parameters</td>
<td>47</td>
</tr>
<tr>
<td>2.5.6 Alternate Boot Methods</td>
<td>48</td>
</tr>
<tr>
<td>2.5.7 Booting a Target Without a Network</td>
<td>50</td>
</tr>
<tr>
<td>2.5.8 Rebooting VxWorks</td>
<td>50</td>
</tr>
<tr>
<td>2.6 Starting Tornado</td>
<td>51</td>
</tr>
<tr>
<td>2.6.1 Toolbars and Buttons</td>
<td>51</td>
</tr>
<tr>
<td>2.6.2 Status Line</td>
<td>53</td>
</tr>
<tr>
<td>2.6.3 Window Management</td>
<td>54</td>
</tr>
<tr>
<td>2.7 Starting a Target Server</td>
<td>54</td>
</tr>
<tr>
<td>2.8 Displaying Information About the Target</td>
<td>56</td>
</tr>
<tr>
<td>2.9 Troubleshooting</td>
<td>57</td>
</tr>
<tr>
<td>2.9.1 Things to Check</td>
<td>57</td>
</tr>
<tr>
<td>Hardware Configuration</td>
<td>57</td>
</tr>
<tr>
<td>Booting Problems</td>
<td>58</td>
</tr>
<tr>
<td>Target-Server Problems</td>
<td>60</td>
</tr>
<tr>
<td>2.9.2 Technical Support</td>
<td>61</td>
</tr>
<tr>
<td>3 Editor</td>
<td>63</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>63</td>
</tr>
</tbody>
</table>
Configuring VxWorks ................................................................. 87
Structuring Your Projects ......................................................... 87

4.3 Creating a Downloadable Application ........................................ 93
  4.3.1 Creating a Project for a Downloadable Application .............. 94
  4.3.2 Project Files for a Downloadable Application ..................... 97
  4.3.3 Working With Application Files ........................................... 98
       Creating, Adding, and Removing Application Files .......... 99
       Displaying and Modifying File Properties ..................... 100
       Opening, Saving, and Closing Files ............................ 100
  4.3.4 Building a Downloadable Application ............................ 101
       Calculating Makefile Dependencies ............................ 102
       Build Specifications ............................................... 103
       Building an Application .......................................... 106
  4.3.5 Downloading an Application ..................................... 108
  4.3.6 Adding and Removing Projects ................................. 109

4.4 Creating a Custom VxWorks Image ...................................... 109
  4.4.1 Creating a Project for VxWorks .................................. 110
  4.4.2 Project Files for VxWorks ........................................ 113
  4.4.3 Configuring VxWorks Components .............................. 116
       Finding VxWorks Components and Configuration Macros .... 117
       Displaying Descriptions and Online Help for Components .... 117
       Including and Excluding Components ........................ 118
       Component Conflicts .............................................. 122
       Changing Component Parameters .............................. 123
       Estimating Total Component Size ............................. 123
  4.4.4 Selecting the VxWorks Image Type ............................. 124
  4.4.5 Building VxWorks ................................................. 125
       Using the Build Menu ........................................ 126
       Using the Command Line ...................................... 127
  4.4.6 Booting VxWorks ............................................... 128
Creating a Bootable Application ............................................................... 128
  4.5.1 Using Automated Scaling of VxWorks ............................................ 128
  4.5.2 Adding Application Initialization Routines .................................... 128

Working With Build Specifications .............................................................. 129
  4.6.1 Changing a Build Specification ....................................................... 130
    Custom Makefile Rules ................................................................. 131
    Makefile Macros .............................................................................. 132
    Compiler Options .............................................................................. 132
    Assembler Options ............................................................................ 134
    Link Order Options ........................................................................... 134
    Linker Options ................................................................................... 135
  4.6.2 Creating New Build Specifications .................................................. 136
  4.6.3 Selecting a Specification for the Current Build ................................. 137

Configuring the Target-Host Communication Interface .............................. 138
  Configuration for an END Driver Connection ...................................... 139
  Configuration for Integrated Target Simulators .................................. 139
  Configuration for NetROM Connection ............................................ 139
  Configuration for Network Connection ............................................ 141
  Configuration for Serial Connection ................................................. 141
  Configuration for tyCoDrv Connection ............................................. 142
  Scaling the Target Agent ................................................................... 142
  Configuring the Target Agent for Exception Hooks .............................. 143
  Starting the Agent Before the Kernel .............................................. 143

Configuring and Building a VxWorks Boot Program .................................. 145

Building a Custom Boot ROM .................................................................... 148

Command-Line Configuration and Build .................................................... 151
  5.1 Introduction ..................................................................................................... 151
  5.2 Building, Loading, and Unloading Application Modules ...................... 152
    5.2.1 Using VxWorks Header Files ......................................................... 153
      VxWorks Header File: vxWorks.h ..................................................... 153
Other VxWorks Header Files ......................................................... 154
ANSI Header Files ............................................................................. 154
ANSI C++ Header Files .................................................................... 154
The -I Compiler Flag .......................................................................... 154
VxWorks Nested Header Files .......................................................... 155
Internal Header Files ......................................................................... 155
VxWorks Private Header Files .......................................................... 156

5.2.2 Compiling Application Modules Using GNU Tools .................... 156
The GNU Tools ................................................................................... 156
Cross-Development Commands ...................................................... 157
Defining the CPU Type ..................................................................... 158
Compiling C Modules With the GNU Compiler ................................ 159
Compiling C++ Modules .................................................................. 160

5.2.3 Compiling Application Modules Using Diab Tools ..................... 160
The Diab Tools .................................................................................... 161
Cross-Development Commands ...................................................... 161
Defining the CPU Type ..................................................................... 162
Compiling C Modules With the Diab Compiler ................................ 163
Compiling C++ Modules .................................................................. 164

5.2.4 Static Linking (Optional) .......................................................... 164
5.2.5 Downloading an Application Module ....................................... 165
5.2.6 Module IDs and Group Numbers .............................................. 166
5.2.7 Unloading Modules .................................................................... 167

5.3 Configuring VxWorks ................................................................. 167
5.3.1 The Board Support Package (BSP) ............................................ 168
The System Library ............................................................................ 169
Virtual Memory Mapping .................................................................. 170
Configuration Files .......................................................................... 170
BSP Initialization Modules ............................................................. 170
BSP Documentation ......................................................................... 170

5.3.2 The Environment Variables ...................................................... 171

5.3.3 The Configuration Header Files ................................................. 172
The Global Configuration Header File: configAll.h ...................... 172
The BSP-specific Configuration Header File: config.h ................... 172
Selection of Optional Features ......................................................... 173
5.3.4 The Configuration Module: usrConfig.c .............................. 173
5.3.5 Alternative VxWorks Configurations ................................. 174
Scaling Down VxWorks ................................................................. 174
Executing VxWorks from ROM ................................................... 176
5.4 Building a VxWorks System Image ........................................... 178
5.4.1 Available VxWorks Images .................................................. 178
5.4.2 Rebuilding VxWorks with make ........................................... 180
5.4.3 Including Customized VxWorks Code ................................. 181
5.4.4 Linking the System Modules ............................................... 182
5.4.5 Creating the System Symbol Table Module ....................... 184
5.5 Makefiles for BSPs and Applications ....................................... 185
5.5.1 Make Variables ................................................................. 186
Variables for Compilation Options ........................................... 187
Variables for BSP Parameters ..................................................... 188
Variables for Customizing the Run-Time .................................. 189
5.5.2 Using Makefile Include Files for Application Modules ......... 190
5.6 Creating Bootable Applications ............................................... 191
5.6.1 Linking Bootable Applications ............................................. 191
5.6.2 Creating a Standalone VxWorks System with a Built-in Symbol Table 193
5.6.3 Creating a VxWorks System in ROM ................................. 194
5.7 Building Projects from a BSP .................................................. 195
6 VxSim .............................................................................................. 197
6.1 Introduction .............................................................................. 197
6.2 Integrated Simulator .............................................................. 198
Installation and Configuration .................................................... 199
## Contents

Starting VxSim ................................................................. 199  
Changing the Simulator Boot Line ................................. 199  
Rebooting VxSim ................................................................. 200  
Exiting VxSim ................................................................. 200  
Back End ................................................................................. 201  
System-Mode Debugging .................................................... 201  
Symbols .................................................................................... 201  
File Systems ............................................................................. 201  

6.3 Building Applications ......................................................... 202  
Defining the CPU Type ......................................................... 203  
The Toolkit Environment ..................................................... 203  
Compiling C and C++ Modules ............................................. 203  
Linking an Application to VxSim ........................................... 203  

6.4 Architecture Considerations ............................................... 204  
Supported Configurations .................................................. 204  
Endianess .................................................................................. 204  
Simulator Timeout ................................................................. 204  
The BSP Directory ................................................................. 205  
Interrupts .................................................................................. 205  
Clock and Timing Issues ....................................................... 206  
WindView Instrumentation on the Windows Simulator ........ 207  

6.5 Configuring the VxSim Full Simulator ................................. 207  
Installing VxSim Network Drivers ..................................... 208  
Uninstalling VxSim Network Drivers .................................. 211  
Configuring VxSim for Networking ..................................... 212  
Running Multiple Simulators .............................................. 213  
System Mode Debugging ..................................................... 213  
IP Addressing ........................................................................... 213  

7 Shell ........................................................................................................... 219  

7.1 Introduction ........................................................................ 219  

7.2 Using the Shell ........................................................................ 221  
7.2.1 Starting and Stopping the Tornado Shell ................. 221  
7.2.2 Downloading From the Shell ...................................... 223
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.3 Shell Features</td>
<td>.................................................................</td>
<td>223</td>
</tr>
<tr>
<td>I/O Redirection</td>
<td>.................................................................</td>
<td>223</td>
</tr>
<tr>
<td>Target Symbol and Path Completion</td>
<td>.................................................................</td>
<td>224</td>
</tr>
<tr>
<td>Synopsis Printing</td>
<td>.................................................................</td>
<td>225</td>
</tr>
<tr>
<td>HTML Help</td>
<td>.................................................................</td>
<td>225</td>
</tr>
<tr>
<td>Data Conversion</td>
<td>.................................................................</td>
<td>226</td>
</tr>
<tr>
<td>Data Calculation</td>
<td>.................................................................</td>
<td>226</td>
</tr>
<tr>
<td>Calculations With Variables</td>
<td>.................................................................</td>
<td>226</td>
</tr>
<tr>
<td>WindSh Environment Variables</td>
<td>.................................................................</td>
<td>227</td>
</tr>
<tr>
<td>7.2.4 Invoking Built-In Shell Routines</td>
<td>.................................................................</td>
<td>229</td>
</tr>
<tr>
<td>Task Management</td>
<td>.................................................................</td>
<td>229</td>
</tr>
<tr>
<td>Task Information</td>
<td>.................................................................</td>
<td>230</td>
</tr>
<tr>
<td>System Information</td>
<td>.................................................................</td>
<td>232</td>
</tr>
<tr>
<td>System Modification and Debugging</td>
<td>.................................................................</td>
<td>236</td>
</tr>
<tr>
<td>C++ Development</td>
<td>.................................................................</td>
<td>238</td>
</tr>
<tr>
<td>Object Display</td>
<td>.................................................................</td>
<td>238</td>
</tr>
<tr>
<td>Network Status Display</td>
<td>.................................................................</td>
<td>240</td>
</tr>
<tr>
<td>Resolving Name Conflicts between Host and Target</td>
<td>.................................................................</td>
<td>241</td>
</tr>
<tr>
<td>7.2.5 Running Target Routines from the Shell</td>
<td>.................................................................</td>
<td>241</td>
</tr>
<tr>
<td>7.2.6 Rebooting from the Shell</td>
<td>.................................................................</td>
<td>242</td>
</tr>
<tr>
<td>7.2.7 Using the Shell for System Mode Debugging</td>
<td>.................................................................</td>
<td>242</td>
</tr>
<tr>
<td>7.2.8 Interrupting a Shell Command</td>
<td>.................................................................</td>
<td>246</td>
</tr>
<tr>
<td>7.3 The Shell C-Expression Interpreter</td>
<td>.................................................................</td>
<td>247</td>
</tr>
<tr>
<td>7.3.1 Data Types</td>
<td>.................................................................</td>
<td>247</td>
</tr>
<tr>
<td>7.3.2 Lines and Statements</td>
<td>.................................................................</td>
<td>249</td>
</tr>
<tr>
<td>7.3.3 Expressions</td>
<td>.................................................................</td>
<td>249</td>
</tr>
<tr>
<td>Literals</td>
<td>.................................................................</td>
<td>250</td>
</tr>
<tr>
<td>Variable References</td>
<td>.................................................................</td>
<td>250</td>
</tr>
<tr>
<td>Operators</td>
<td>.................................................................</td>
<td>251</td>
</tr>
<tr>
<td>Function Calls</td>
<td>.................................................................</td>
<td>251</td>
</tr>
<tr>
<td>Subroutines as Commands</td>
<td>.................................................................</td>
<td>252</td>
</tr>
<tr>
<td>Arguments to Commands</td>
<td>.................................................................</td>
<td>253</td>
</tr>
<tr>
<td>Task References</td>
<td>.................................................................</td>
<td>253</td>
</tr>
<tr>
<td>7.3.4 The “Current” Task and Address</td>
<td>.................................................................</td>
<td>254</td>
</tr>
</tbody>
</table>
7.3.5 Assignments ................................................................. 254
    Typing and Assignment .................................................. 254
    Automatic Creation of New Variables ............................ 255
7.3.6 Comments ............................................................... 255
7.3.7 Strings ................................................................. 256
7.3.8 Strings and Path Names .......................................... 256
7.3.9 Ambiguity of Arrays and Pointers ......................... 257
7.3.10 Pointer Arithmetic .................................................. 258
7.3.11 C Interpreter Limitations ...................................... 258
7.3.12 C-Interpreter Primitives ........................................ 259
7.3.13 Terminal-Control Characters ............................... 260
7.3.14 Redirection in the C Interpreter ............................ 261
    Ambiguity Between Redirection and C Operators ............ 261
    The Nature of Redirection ........................................... 262
    Scripts: Redirecting Shell I/O ..................................... 262
    C-Interpreter Startup Scripts ...................................... 264
7.4 C++ Interpretation ....................................................... 264
    Overloaded Function Names ........................................ 265
    Automatic Name Demangling ....................................... 266
7.5 Shell Line Editing ....................................................... 267
7.6 Object Module Load Path ............................................ 270
7.7 Tcl: Shell Interpretation .............................................. 271
    Tcl: Controlling the Target ........................................ 272
    Tcl: Calling Target Routines ..................................... 273
    Tcl: Passing Values to Target Routines ....................... 273
    Tcl: Calling under C Control ..................................... 273
7.7.3 Tcl: Tornado Shell Initialization ............................ 274
7.8 The Shell Architecture ................................................................. 275
  7.8.1 Controlling the Target from the Host ................................. 275
  7.8.2 Shell Components ............................................................... 277
  7.8.3 Layers of Interpretation ......................................................... 279

8 Target Server ....................................................................................... 281
  8.1 Introduction .................................................................................. 281
  8.2 Configuring and Starting a Target Server .................................... 282
    8.2.1 Using the Configure Target Servers Dialog Box .................. 283
    8.2.2 Target-Server Configuration Options .................................... 285
      Authorizations ........................................................................... 285
      Back End .................................................................................. 286
      Core File and Symbols ............................................................. 288
      Memory Cache Size ................................................................ 289
      Target Server File System ....................................................... 290
      Console and Redirection .......................................................... 291
      Logging .................................................................................... 292
      Miscellaneous ......................................................................... 293
      Target Memory Pool ................................................................. 294
    8.2.3 Using the Command Line ....................................................... 294
  8.3 Target Server Icon and Log Console Window .............................. 295
  8.4 Stopping a Target Server ............................................................. 296
  8.5 Selecting a Target Server ............................................................. 296
    Tornado Launch Toolbar ............................................................... 296
    Tools Menu ................................................................................ 297
  8.6 Managing a Target Server ............................................................ 298
    Sharing and Reserving a Target Server ........................................ 299

9 Browser ............................................................................................... 303
  9.1 A System-Object Browser ........................................................... 303
9.2 Starting the Browser .......................................................... 304
9.3 Anatomy of the Browser Window ...................................... 304
9.4 Browser Buttons ................................................................. 306
9.5 Target-Information Window .............................................. 307
9.6 Task-List Window ............................................................... 309
9.7 Memory-Usage Window ...................................................... 310
9.8 Object-Information Windows ............................................. 311
   9.8.1 The Task Browser .................................................... 312
   9.8.2 The Semaphore Browser ........................................ 313
   9.8.3 The Message-Queue Browser ................................. 314
   9.8.4 The Memory-Partition Browser ............................... 315
   9.8.5 The Watchdog Browser ......................................... 316
   9.8.6 The Class Browser ................................................ 317
9.9 The Module-Information Window ...................................... 318
9.10 The Spy Window ............................................................... 320
9.11 The Stack-Check Window ................................................ 321
9.12 The Vector Table Window ................................................. 322
9.13 Browser Displays and Target Link Speed ........................... 324
9.14 Troubleshooting with the Browser ..................................... 324
   9.14.1 Memory Leaks ......................................................... 324
   9.14.2 Memory Fragmentation .......................................... 325
   9.14.3 Stack Overflow ....................................................... 325
   9.14.4 Priority Inversion .................................................... 326
9.15 Tcl: the Browser Initialization File ...................................... 328
10 Debugger ................................................................................................................... 329

10.1 Introduction ........................................................................................................... 329

10.2 Debugger GUI ....................................................................................................... 330
  10.2.1 Debugger Toolbar, Buttons, and Menu Commands ........................................... 331

10.3 Using the Debugger ............................................................................................. 333
  10.3.1 Starting and Stopping the Debugger ................................................................. 334
  10.3.2 Setting the Search Path .................................................................................... 335
  10.3.3 Unloading a Module ......................................................................................... 336
  10.3.4 Running a Program ......................................................................................... 336
  10.3.5 Interrupting a Program .................................................................................... 337
  10.3.6 Setting Breakpoints ......................................................................................... 337
  10.3.7 Continuing Through a Program ..................................................................... 340
  10.3.8 Attaching to a Running Task ......................................................................... 343
  10.3.9 Attaching Automatically ............................................................................... 344
  10.3.10 Detaching from a Running Task .................................................................. 344
  10.3.11 Examining Data, Memory, and the Stack ...................................................... 344
    Watch Window ....................................................................................................... 345
    Variables Window ................................................................................................. 347
    Registers Window ................................................................................................. 348
    Back Trace Window .............................................................................................. 349
    Memory Window ................................................................................................... 350

10.4 Source Code Display Options ............................................................................... 351

10.5 Using the Debugger Command Line ..................................................................... 352
  10.5.1 GDB Initialization Files ................................................................................... 353
  10.5.2 Selecting Modules to Debug ............................................................................ 353
  10.5.3 Running a Program ......................................................................................... 355
  10.5.4 Re-Setting Application I/O ............................................................................. 355
  10.5.5 Using Graphically Enhanced Commands ....................................................... 356
10.5.6 Managing Targets ................................................................. 356
10.5.7 Extended Debugger Commands ........................................... 357
10.5.8 Extended Debugger Variables ............................................. 358
10.6 System-Mode Debugging .......................................................... 359
  10.6.1 Configuring System Mode ...................................................... 360
  10.6.2 Entering System Mode ......................................................... 361
  10.6.3 Thread Facilities in System Mode ......................................... 361
    Displaying Summary Thread Information .................................... 362
    Switching Threads Explicitly ................................................... 363
    Thread-Specific Breakpoints ..................................................... 363
    Switching Threads Implicitly .................................................... 364
10.7 Tcl: Debugger Automation ....................................................... 365
  10.7.1 Tcl: A Simple Debugger Example ........................................ 365
  10.7.2 Tcl: Specialized GDB Commands ........................................ 366
  10.7.3 Tcl: Invoking GDB Facilities ............................................... 368
  10.7.4 Tcl: A Linked-List Traversal Macro Example ....................... 370
10.8 Tcl: Debugger Customization .................................................. 372
  10.8.1 Tcl: Debugger Initialization Files ....................................... 372
  10.8.2 Tcl: Passing Control between the Two Tornado Debugger Interpreters 374
  10.8.3 Tcl: Experimenting with Tornado Debugger Extensions ......... 375
    Tcl: An Add-Symbols Command for the File Menu .................... 375
    Tcl: “this” Menu Command for C++ .................................... 376
11 Building VxDCOM Applications .................................................. 377
  11.1 Introduction .............................................................................. 377
  11.2 The VxDCOM Development Process ....................................... 378
11.3 Configuring a VxDCOM Bootable Image .................................................... 379
  11.3.1 Adding VxDCOM Component Support ........................................ 379
  11.3.2 Configuring the DCOM Parameters ............................................... 380

11.4 Using the VxDCOM Wizard ................................................................. 381
  11.4.1 Choosing the Project Type ............................................................ 381
  11.4.2 Creating a COM/DCOM Skeleton Project ..................................... 382
    Defining the CoClass ................................................................. 382
    Choosing CoClass Options ......................................................... 386
    Generating the Skeleton Files ................................................... 388
  11.4.3 Importing Existing Files into a New Project .................................. 388
    Porting Existing Applications .................................................... 388
    Editing IDL Files ......................................................................... 389
    Adding Non-Automation Types ................................................... 389

11.5 The Generated Output .......................................................................... 389
  Output Directories ............................................................................. 390
  Project Work Files ........................................................................... 390
  Server Output Files ......................................................................... 391
  Client Output Files ......................................................................... 392

11.6 Implementing the Server and Client ....................................................... 392

11.7 Building and Linking the Application .................................................... 393

11.8 Registering, Deploying, and Running Your Application ......................... 394
  11.8.1 Registering Proxy DLLs on Windows ......................................... 394
  11.8.2 Register the Type Library .......................................................... 395
  11.8.3 Registering the Server ............................................................... 395
  11.8.4 Authenticating the Server .......................................................... 396
  11.8.5 Activating the Server ................................................................. 397

12 Customization ....................................................................................... 399

12.1 Introduction .......................................................................................... 399
12.2 Toolbars and Status Bar ................................................................. 399
12.3 Setting Options ............................................................................... 400
  12.3.1 Setting Download Options .................................................... 400
  12.3.2 Editor Preferences ................................................................. 402
  12.3.3 External Editor ...................................................................... 403
  12.3.4 Project .................................................................................. 405
  12.3.5 Version Control ..................................................................... 406
  12.3.6 Fonts/Colors ......................................................................... 408
  12.3.7 Debugger ............................................................................... 411
  12.3.8 Tornado Registry ................................................................. 412
  12.3.9 Installation and Licenses ....................................................... 413
12.4 Customizing the Tools Menu ....................................................... 413
  12.4.1 The Customize Tools Dialog Box ........................................... 413
    Macros for Customized Menu Commands .................................... 416
  12.4.2 Examples of Tools Menu Customization ................................ 418
    Version Control ........................................................................... 418
    Alternate Editor .......................................................................... 419
    Binary Utilities ........................................................................... 420
    World Wide Web .......................................................................... 420
  12.4.3 Customizing the Build Menu ................................................ 421
12.5 Tcl Customization Files ................................................................. 423
  Tornado Initialization ................................................................. 423
  HTML Help Initialization ............................................................. 424

Appendices ....................................................................................... 425

A Directories and Files ....................................................................... 427
A.1 Introduction .................................................................................. 427
A.2 Host Directories and Files .......................................................... 428
A.3 Target Directories and Files ................................................................. 430
A.4 Initialization and State-Information Files .......................................... 438

B Makefile Details ...................................................................................... 441
B.1 Introduction ......................................................................................... 441
B.2 Customizing the VxWorks Makefile ..................................................... 441
B.3 Commonly Used Makefile Macros ......................................................... 442

C Tcl .......................................................................................................... 447
C.1 Why Tcl? ............................................................................................. 447
C.2 Introduction to Tcl .............................................................................. 448
  C.2.1 Tcl Variables .................................................................................. 448
  C.2.2 Lists in Tcl .................................................................................... 449
  C.2.3 Associative Arrays ......................................................................... 450
  C.2.4 Command Substitution .................................................................. 451
  C.2.5 Arithmetic ....................................................................................... 451
  C.2.6 I/O, Files, and Formatting .............................................................. 452
  C.2.7 Procedures ....................................................................................... 452
  C.2.8 Control Structures ......................................................................... 453
  C.2.9 Tcl Error Handling ......................................................................... 454
  C.2.10 Integrating Tcl and C Applications ............................................. 454

D Coding Conventions ................................................................................. 455
D.1 Introduction .......................................................................................... 455
D.2 File Heading .......................................................................................... 456
D.3 C Coding Conventions ........................................................................... 457
  D.3.1 C Module Layout ............................................................................ 457
1.1 Introduction

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

- VxWorks, a high-performance real-time operating system.
- Application-building tools (compilers and associated programs).
- 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.

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

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

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

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

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

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

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

Tornado maintains a complete host-resident symbol table for the target. This symbol table is incremental: the server incorporates symbols as it downloads each object module. You can examine variables, call subroutines, spawn tasks,
disassemble code in memory, set breakpoints, trace subroutine calls, and so on, all using the original symbolic names.

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

### 1.3 VxWorks Target Environment

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

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

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

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

Tornado is also compatible with the VxWorks optional components VxMP and VxFusion. VxMP provides for synchronization of tasks on different CPUs over a back plane, while VxFusion allows that synchronization to take place over any kind of connection including Ethernet.
For detailed information on VxWorks and on its optional components, see the VxWorks Programmer’s Guide and the VxWorks Network Programmer’s Guide. For information on exactly what functions your architecture supports, see the appropriate Architecture Supplement.

1.4 Tornado Host 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 summarizes 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 GNU tools originally developed by the Free Software Foundation (FSF). The Tornado project facility provides a GUI for the GNU tools that is powerful and easy to use.


WindSh Command Shell

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

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

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

The shell is described in 7. Shell.
CrossWind Debugger

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

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

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

The debugger is described in 10. Debugger. Also see the GDB User’s Guide.

Browser

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

- Summaries of active tasks (classified as system tasks or application tasks).
- The state of particular tasks, including register usage, priority, and other attributes.
- Comparative CPU usage by the entire collection of tasks.
- Stack consumption by all tasks.
- Memory allocation.
- Summary of modules linked dynamically into the run-time system.
- Structure of any loaded object module.
- Operating-system objects such as semaphores, message queues, memory partitions, and watchdog timers.
The browser is described in 9. Browser.

**WindView Software Logic Analyzer**

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

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

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

**VxSim Target Simulator**

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

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

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

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

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

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

Target Agent

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

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

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

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

A key function of the agent is to service the requests of the host-resident object-module loader. If the agent is linked into the run-time and stored in ROM, the target server automatically initializes the symbol table from the host-resident image of the target run-time system as it starts. From this point on, all downloads are incremental in nature, greatly reducing download time.
The agent itself is scalable; you can choose what features to include or exclude. This permits the creation of final-production configurations that still allow field testing, even when very little memory can be dedicated to activities beyond the application’s purpose.

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

### Tornado Target Server

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

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

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

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

For information about how to configure, start, and manage target servers see 8. Target Server and the `tgtsvr` reference entry in Help>Manuals Contents>Tornado Reference/Tornado Tools.

### Tornado Registry

Tornado provides a central target server registry that allows you to select a target server by a convenient name. The registry associates a target server’s name with the network address needed to connect with that target server. You can see the
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.291 for information about how to configure a target server with a virtual console.
1.6 Customer Services

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

Customer Support

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

http://www.windriver.com/support

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

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

- **E-mail.** You can contact Wind River Customer Support by sending e-mail to support@windriver.com.

- **1-800-872-4977 (1-800-USA-4WRS).** Within North America, you can contact Customer Support with a toll-free voice telephone call. For telephone access outside North America, see the Support Web site shown above.

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

WindSurf

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

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

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

You do not need much of this chapter if all you want to do is connect to a target that is already set up on your network. If this is the case, read 2.2 Setting Up the Host, p.16 and then proceed with 2.5 Booting VxWorks, p.41.

**Tornado Configuration (once only)**

1. Make sure that there is a Tornado registry running at your site.

2. Make sure your host environment includes the right definitions, on the host system where you attach the target.

**Target Configuration (once for each new target)**

3. Modify your host network tables so that you can communicate with your target.

4. Create and install the VxWorks boot ROM (or equivalent) in your target.

5. Set up physical connections (serial, Ethernet) between your target and your host.
6. Define a Tornado target server to connect to the new target.

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

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

---

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

---

**Target Servers and Target Agents**

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

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

---

**Target Agent Modes**

All of the standard 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 10.6 System-Mode Debugging, p.359) or the shell (7.2.7 Using the Shell for System Mode Debugging, p.242). In order to support a system-mode agent, the target communication path must work in polled mode (because the external agent needs to communicate to the host even when the
system is suspended). Thus, the choice of communication path can affect what debugging modes are available.

Communication Paths

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

Nevertheless, there are situations where you may wish to use a non-network connection, such as a serial line 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. Older versions of VxWorks network interface drivers such as netif do not support polled operations and so cannot be used as a connection for system-mode debugging.

Note that the target-server back end connection is not always the same as the connection used to load the VxWorks image into target memory. For example, you can boot VxWorks over Ethernet, but use a serial line connection to 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. 50. Certain BSPs may provide other alternatives, such as flash memory; see the reference information for your BSP.

2.2 Setting Up the Host

In order for your target to communicate with the Tornado host tools, you need to set up a Tornado registry on either your own or a networked host, as well as TCP/IP on your own host.
The Tornado target server registry is a service that keeps track of target servers and provides your host with access to them. The registry must always be running; otherwise Tornado tools cannot communicate with targets. Because Tornado development tools are independent of the method of communication with the 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 pop-up 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 2-2).

Figure 2-2  Tornado Registry Window and Icon

1. The Tornado registry program is the file c:\tornado\host\x86-win32\bin\wtxregd.exe. The Windows NT service version is wtxregds.exe.
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 E. Windows NT Service Manager).

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

TCP/IP

You need to install TCP/IP in your host whether or not your host is on a network. See your Windows documentation on how to install it.

WARNING: Tornado tools such as the shell and the 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 part of a networked system. To install TCP/IP on your PC as part of the Windows network support, follow 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. First select Start>Settings>Control Panel>Network. Select the Protocols tab, click Properties, and select the appropriate networking card. When there is no networking hardware, select Dial-up Adapter.

2. TCP/IP configuration requires that you specify an IP address, which is normally your host IP address. If you are not connecting to any other hosts, you can choose any arbitrary address.

3. Choose a name for your system, and enter it as the host name.
4. If you are not using a network connection, disable Domain Name Service (DNS), because the service is not available.

5. Record your host name and associated IP address in the hosts file. The file is located in the c:\winnt\system32\drivers\etc\ directory and may be called hosts or lmhosts.

These steps produce a functioning TCP/IP subsystem on your host.

**NOTE:** If your machine is not networked, the Tornado services must all run locally; make sure that the registry is running on your system (see Tornado Registry, p.17).

### Environment Variables

If you plan to use the Tornado tools from the UI all system variables are set during installation. However, if you plan to use the command line you will need to set some variables by hand. The easiest way is to go to the bin directory and execute torVars.bat.

```bash
C:\> cd installDir/host/x86-win32/bin
C:\> torVars.bat
```

### 2.3 Setting Up the Default Target Hardware

This section covers bringing up VxWorks on a hardware target with the relatively simple configuration matching the default software image. In the next section, 2.4 Host-Target Communication Configuration, p.26, additional options are discussed. 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 Default Target Configuration

VxWorks is a flexible system that has been ported to many different hardware platforms. The default VxWorks run-time development configuration is shown in Figure 2-3. The pre-built VxWorks images shipped with your BSP include all the necessary components to run on this hardware 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).

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

For detailed information about your particular target Board Support Package (BSP), see Help>Manuals Contents>BSP Reference in the Tornado IDE.
2.3.2 Networking the Host and Target

IP networking over Ethernet is the most desirable way to connect a development target to your host, because of the high bandwidth it provides. This section describes setting up simple IP connections to a target over Ethernet. To read about other communication strategies, see 2.4 Host-Target Communication Configuration, p.26.

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

- Initializing the host network software and setting permissions.
- Establishing the VxWorks system name and network address 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. For appropriate settings, see TCP/IP, p.18.

To use the default VxWorks configuration and boot VxWorks over the network, 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.

Tornado includes an FTP-server application, wftpd32.exe. See F. FTP Server for information on configuring this FTP server. 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 called `hosts` on your machine. Otherwise, you must identify targets by IP address.

The `hosts` file records 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 NT: the `hosts` file is `c:\winnt\system32\drivers\etc\hosts`
- Windows 2000 and XP: the `hosts` file is `c:\winnt\system32\drivers\etc\lmhosts`

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
```

### 2.3.3 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).
Setting Up a Boot Mechanism

Tornado is shipped with the VxWorks images shown in Table 2-1.

<table>
<thead>
<tr>
<th>Compiled with GNU</th>
<th>Compiled with Diab</th>
</tr>
</thead>
<tbody>
<tr>
<td>vxWorks</td>
<td>vxWorks</td>
</tr>
<tr>
<td>vxWorks_rom</td>
<td>vxWorks_rom</td>
</tr>
<tr>
<td>vxWorks_romCompress</td>
<td>vxWorks_romCompress</td>
</tr>
<tr>
<td>vxWorks_romResident</td>
<td>vxWorks_romResident</td>
</tr>
</tbody>
</table>

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

Boot ROM
Most BSPs boot from ROMs.

Floppy Disk
Some BSPs for systems that include floppy drives use boot diskettes instead of a boot ROM. For example, the BSPs for Pentium 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 to write the boot program into flash memory is supplied by the board vendor.

For specific information on a BSP’s booting method, see Help>Manuals Contents>BSP Reference. Instructions for making a floppy disk for booting a Pentium target are in the VxWorks for Pentium Architecture Supplement.

You may also wish to replace a boot ROM, even if it is available, with a ROM emulator. This is particularly desirable if your target has no Ethernet capability, because the ROM emulator can be used to provide connectivity at near-Ethernet speeds. Tornado includes support for one such device, NetROM. For information about how to use NetROM on your target, refer to Configuration for NetROM Connection, p. 139. Contact the nearest Wind River office (see copyright page) for information about support for other ROM emulators.

2. NetROM is a trademark of Applied Microsystems Corporation.
For cases where boot ROMs are used to boot VxWorks, install the appropriate set of boot ROMs on your target board(s). When installing boot ROMs, be careful to:

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

See 4.8 Configuring and Building a VxWorks Boot Program, p.145 for instructions on creating a new boot program with parameters customized for your site.

**Setting Board Jumpers**

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

![CPU system controller](image-url)
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 the Terminal Emulator**

Most VxWorks targets include at least one on-board serial port. This serial port must be connected to a terminal emulator (HyperTerminal) to use the default configuration. After initial configuration of the boot parameters and getting started with VxWorks, you may wish to configure VxWorks to boot automatically without a terminal. Refer to the CPU board hardware documentation for proper connection of the RS-232 signals.

Tornado comes with terminal-emulator software already configured for connecting to VxWorks targets on either COM1 or COM2; the software is 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 the 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

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

2.4.1 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 wdbpipe back end (see 8.2 Configuring and Starting a Target Server, p.282).

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

Configuring the Target Agent For Network Connection

The default VxWorks system image is configured for a networked target. See 4.7 Configuring the Target-Host Communication Interface, p.138 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. 139 for information about configuring the VxWorks target agent for an END connection.

2.4.3 Serial-Line Connections

Figure 2-5 illustrates a minimal cross-development configuration: the target is a bare board, connected to the host development system by a single serial line. For a configuration of this sort, use a combination of a boot mechanism that does not require a network and an alternative Tornado communications back end.

Figure 2-5  A Minimal Tornado Configuration

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

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

- Configure and build a boot program for the serial connection, because the default boot configuration uses an FTP download from the host.
• Reconfigure and rebuild VxWorks with a target agent configuration for a serial connection.

• Configure and start a target server for a serial connection.

For more information, see 4.7 Configuring the Target-Host Communication Interface, p.138.

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

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

**Configuring the Target Agent For Serial Connection**

To configure the target agent for a raw serial communication connection, reconfigure and rebuild VxWorks with a serial communication interface for the target agent (see Configuration for Serial Connection, p.141).

**Configuring the Boot Program for Serial Connection**

When you connect the host and target exclusively over serial lines, you must configure and build a boot program for the serial connection because the default boot configuration uses an FTP download from the host (see 4.8 Configuring and Building a VxWorks Boot Program, p.145). 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.50.

**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 the Terminal Emulator, p.25). 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_TEST
       /* test in polled mode if the kernel hasn't started */
       if (taskIdCurrent == 0)
           wdbSioTest (pSioChan, SIO_MODE_POLL, 0);
       else
           wdbSioTest (pSioChan, SIO_MODE_INT, 0);
#endif /* INCLUDE_WDB_TTY_TEST */
```

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

With this configuration, attach any terminal emulator on the host to the 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 8.2 Configuring and Starting a Target Server, p.282).

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 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.50.

**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.139.

**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 \texttt{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**

```
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
```
2

Setup and Startup

```
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 8.2 Configuring and Starting a Target Server, p.282.)

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.31.

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

```
NetROM> tgtreset
```

**Troubleshooting the NetROM ROM-Emulator Connection**

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

**Download Configuration**

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

**Initialization**

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

**RAM Configuration**

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

```
int wdbNetromTest = 0;
```

to:

```
int wdbNetromTest = 1;
```
NOTE: There are two versions of `wdbNetromPktDrv.c`. The one for the 400 series is located in `target/src/drv/wdb` and the one for the 500 series is located in `target/src/drv/wdb/amc500`. Be sure to edit the appropriate one.

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

```
% telnet NetROM_IPaddress 1235
```

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

```
WDB NetROM communication ready
```

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

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

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

**Communication**

Type the following at the NetROM prompt:

```
NetROM> set debugecho on
```

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

```
int wdbNetromDebug = 0;
```

to:

```
int wdbNetromDebug = 1;
```

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

(1) Kill the target server.
(2) Type `tgtreset` at the NetROM prompt.
(3) Reboot your target.
(4) Start the target server using the `-Bd` option to log transactions between the target server and the agent to a log file. Use the target server `-Bt` option to increase the timeout period. (This is necessary whenever the NetROM debug echo feature is enabled, because `debugecho` slows down the connection.)

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

2.4.5 The Transparent Mode Driver (TMD)

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

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

The TMD is added to the current build by selecting the VxWorks tab in the project dialog window. Expand the VxWorks entry associated with your project, and from the list that appears, select development tool components>select WDB connection>WDB visionTMD connection.

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

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

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

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

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

Configuring visionICE II/visionPROBE II

The debugger being used must be configured correctly to download the vxWorks image created in the previous steps to the target. The debugger will also be used to start the image running once it has been downloaded. Once the image is running on the target, the TM Driver will also be available since the WDB agent that is included in the vxWorks image uses the TM Driver as the connection mechanism.
Instructions for configuring the debugger and downloading and executing the vxWorks image on the target are provided for two of Wind Rivers debuggers in the *Transparent Mode Driver User’s Guide.*

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

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

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

### Starting the Target Server

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

1. With the VxWorks image running on the target, return to the Tornado project window. Select Tools>Target Server>Configuration from the main menu.
2. Click the **New** button at the top right side of the dialog window, enter a description, and check the **Add description to menu** check box. In the space beside **Target Server Name**, enter the same description as you placed in the **Description** space. This will result in a link being created in the Tornado Tools menu that will automatically launch the target server when selected.

3. Define **Target Server Properties** by selecting an item from the drop-down menu and then completing the related properties as shown in Table 2-2.

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

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

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

For more information about using either the GUI or the command line to configure target servers, see 2.7 *Starting a Target Server*, p.54.
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 the Terminal Emulator*, p.25.)

**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.21 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.

Unless your target CPU has nonvolatile RAM (NVRAM), you will eventually find it useful to build a new version of the boot loader that includes all parameters required for booting a VxWorks image (see 4.8 Configuring and Building a VxWorks Boot Program, p.145). In the course of your developing an application, you will also build bootable applications (see 4.5 Creating a Bootable Application, p.128).

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.
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.

Figure 2-8  Boot Configuration Example
The boot program displays the VxWorks boot prompt, as follows:

[VxWorks 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:ffffff00
inet on backplane (b) :
host inet (h) : 90.0.0.1
gateway inet (g) :
user (u) : fred
ftp password (pw)(blank=use rsh) : 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 either h or ? at the boot prompt, followed by ENTER:

[VxWorks Boot]: ?

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

<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></td>
<td>ENTER: Do not change that address, but continue prompting at the next address.</td>
</tr>
<tr>
<td></td>
<td>number: Set the 16-bit contents to number.</td>
</tr>
<tr>
<td></td>
<td>. (dot): Do not change that address, and quit.</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>
</tbody>
</table>
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.42. 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.48.

**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.42.)

**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.3

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

3. 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 8.6 Managing a Target Server, p.298.
**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:

- The boot program first initializes its network interfaces.
- While VxWorks is booting, you can see the size of each VxWorks section (text, data, and bss) as it is loaded.
- After the system is completely loaded, the boot program displays the entry address and transfers control to the loaded VxWorks system.
- When VxWorks starts up, it begins just as the boot ROM did, by initializing its network interfaces; the network-initialization messages appear again, sometimes accompanied by other messages about optional VxWorks facilities.
- After this point, VxWorks is up and ready to attach to the Tornado tools.
- The boot display may be useful for troubleshooting. The following hints refer to Figure 2-9. For more troubleshooting ideas, see 2.9 Troubleshooting, p.57.
- The initial “Attaching network interface” is displayed without the corresponding “done,” verify that the system controller is configured properly and the Ethernet board is properly jumpered.
- If “Loading...” is displayed without the size of the VxWorks image, this may indicate problems with the Ethernet cable or connection, or an error in the network configuration (for example, a bad host or gateway Internet address).
- If the line “Starting at” is printed and there is no further indication of activity from VxWorks, this generally indicates there is a problem with the boot image.
- If “Attaching network interface” is displayed without the “done,” this may indicate there is a problem with the network driver in the newly loaded VxWorks image.

2.5.6 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 4.8 Configuring and Building a VxWorks Boot Program, p.145 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 4.8 Configuring and Building a VxWorks Boot Program, p.145.

Before you boot the target, configure a target server with the TSFS option and start it. See Target Server File System, p.290.

The only boot parameters required to boot the target are boot device and file name (see 2.5.4 Description of Boot Parameters, p.45). 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

The top of the main window contains five toolbars; 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.

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
Table 2-4  Tornado Launch Toolbar Buttons

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Button Image]</td>
<td>Starts a browser for the selected target. 9. Browser describes how to use the browser.</td>
</tr>
<tr>
<td>![Button Image]</td>
<td>Starts a Tornado shell for the selected target. 7. Shell describes how to use the shell.</td>
</tr>
<tr>
<td>![Button Image]</td>
<td>Starts the debugger. 10. Debugger describes how to use the debugger.</td>
</tr>
<tr>
<td>![Button Image]</td>
<td>Starts the VxWorks target simulator. 6. VxSim describes the simulator. The Tornado Getting Started Guide and 4. Projects include information about using the simulator.</td>
</tr>
</tbody>
</table>

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 that selection 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 as 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 8.2 Configuring and Starting a Target Server, p.282 and 4.7 Configuring the Target-Host Communication Interface, p.138.

### 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 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
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. 57 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 the 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 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 if you are using `wftpd32.exe` (shipped by Wind River). Otherwise, consult your system documentation on the FTP Server shipped with it.

- **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:
  Reply from 91.0.10.1: bytes=32 time=10ms TTL=255
  Reply from 91.0.10.1: bytes=32 time<10ms TTL=255
  Reply from 91.0.10.1: bytes=32 time<10ms TTL=255
  Reply from 91.0.10.1: bytes=32 time<10ms TTL=255
  ```

  **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

<table>
<thead>
<tr>
<th>Network Address</th>
<th>Netmask</th>
<th>Gateway Address</th>
<th>Interface</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>0.0.0.0</td>
<td>91.0.10.254</td>
<td>91.0.10.26</td>
<td>1</td>
</tr>
<tr>
<td>127.0.0.0</td>
<td>255.0.0.0</td>
<td>127.0.0.1</td>
<td>127.0.0.1</td>
<td>1</td>
</tr>
<tr>
<td>91.0.10.0</td>
<td>255.255.255</td>
<td>91.0.10.26</td>
<td>91.0.10.26</td>
<td>1</td>
</tr>
<tr>
<td>91.0.10.26</td>
<td>255.255.255</td>
<td>127.0.0.1</td>
<td>127.0.0.1</td>
<td>1</td>
</tr>
<tr>
<td>91.11.255.255</td>
<td>255.255.255</td>
<td>91.0.10.26</td>
<td>91.0.10.26</td>
<td>1</td>
</tr>
<tr>
<td>224.0.0.0</td>
<td>224.0.0.0</td>
<td>91.0.10.26</td>
<td>91.0.10.26</td>
<td>1</td>
</tr>
<tr>
<td>255.255.255.255</td>
<td>255.255.255</td>
<td>91.0.10.26</td>
<td>91.0.10.26</td>
<td>1</td>
</tr>
</tbody>
</table>

Active Connections

<table>
<thead>
<tr>
<th>Proto</th>
<th>Local Address</th>
<th>Foreign Address</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>mercury:1025</td>
<td>saturn.wrs.com:nbsession</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>TCP</td>
<td>mercury:1177</td>
<td>earth.wrs.com:nntp</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>TCP</td>
<td>mercury:1259</td>
<td>oak.oakland.edu:ftp</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

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.288) or the equivalent -c option to tgtsvr (in the online Tornado API Reference under Tornado Tools) 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 customer support organization. Your comments and suggestions are welcome as well. For information about customer support, see 1.6 Customer Services, p.12.
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 (10. 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.

Figure 3-1  Standard Toolbar

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;Open]</td>
<td>File&gt;Open</td>
<td>Open an existing document. See 3.3.2 Opening a File, p.67.</td>
</tr>
<tr>
<td>![File&gt;Save]</td>
<td>File&gt;Save</td>
<td>Save the current document. See 3.3.3 Saving and Closing a File, p.68.</td>
</tr>
<tr>
<td>![Edit&gt;Cut]</td>
<td>Edit&gt;Cut</td>
<td>Delete the selection and place it in the clipboard. See 3.4.2 Editing Text, p.69.</td>
</tr>
<tr>
<td>![Edit&gt;Copy]</td>
<td>Edit&gt;Copy</td>
<td>Copy the selection to the clipboard. See 3.4.2 Editing Text, p.69.</td>
</tr>
<tr>
<td>![Edit&gt;Paste]</td>
<td>Edit&gt;Paste</td>
<td>Insert the clipboard text at the insertion point. See 3.4.2 Editing Text, p.69.</td>
</tr>
<tr>
<td>![File&gt;Print]</td>
<td>File&gt;Print</td>
<td>Print the document. See 3.4.5 Printing, p.72.</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.72.

3.3.1 Creating a File

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>File Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="File New" /></td>
<td>CTRL+N</td>
<td>New</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 12.3.2 Editor Preferences, p.402). 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 12.3.6 Fonts/Colors, p.408.)

3.3.2 Opening a File

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>File Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Open button]</td>
<td>CTRL+O</td>
<td>Open</td>
</tr>
<tr>
<td>n/a</td>
<td>ALT+F num</td>
<td>File Name</td>
</tr>
</tbody>
</table>

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

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

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>![image]</td>
<td>CTRL+X</td>
<td>Cut</td>
</tr>
<tr>
<td>![image]</td>
<td>CTRL+C</td>
<td>Copy</td>
</tr>
<tr>
<td>![image]</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 (12.3.2 Editor Preferences, p.402).
3.4.3 Navigation

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 4.3.4 Building a Downloadable Application, p.101 for more information.

The editor uses the standard Windows keys and mouse actions for moving throughout the file.

3.4.4 Search and Replace

Click Edit>Find to search for a string in your file. You can also open the Find dialog box with CTRL+F or ALT+F3. 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
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.

Figure 3-5  **Find Dialog Box**

<table>
<thead>
<tr>
<th>Find</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Find what:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Match whole word only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Match case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Find Next</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-6  **Replace Dialog Box**

<table>
<thead>
<tr>
<th>Replace</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Find what:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace with</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Match whole word only</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Match case</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace All</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

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.
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 12.3 Setting Options, p.400 for details.

You can also select an alternate editor for use with Tornado. See 12.3.3 External Editor, p.403.
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:

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

NOTE: For a tutorial introduction to the project facility and its use with the integrated version of the VxWorks target simulator and other Tornado tools, see the Tornado Getting Started Guide.
There are several key terms that you must understand before you can use the project facility documentation effectively:

**Downloadable application**

A downloadable application consists of one or more relocateable object modules, which can be downloaded and dynamically linked to VxWorks, and then started from the shell or debugger. A novel aspect of the Tornado
development environment is the dynamic loader, which allows objects to be loaded onto a running system. This provides much faster debug cycles compared with having to rebuild and re-link the entire operating system. A downloadable application can consist of a single file containing a simple “hello world” routine, or a complex application consisting of many files and modules that are partially linked as a single object (which is created automatically by the project facility as `projectName.out`).

**Bootable application**

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

**Project**

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

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

**Workspace**

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

**Component**

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

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

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

BSP

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

Project Facility GUI

The main components of the project facility GUI are:

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

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:

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

A context-sensitive menu is available in each of the workspace views. A right-mouse click displays the menu. The first section of the menu provides commands relevant to the GUI object you have selected. The second section displays commands relevant to the current page of the window. And the third section displays global commands that are relevant to the entire workspace (Figure 4-2).
Many of the pop-up menu options are also available under the File, 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 12. Customization.

Figure 4-2  Workspace Window Pop-up Menu

![Workspace Window Pop-up Menu](image)

**Workspace Icons**

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

<table>
<thead>
<tr>
<th>Icon</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>All panes</td>
<td>Workspace</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>Files and Builds panes</td>
<td>Downloadable application</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>All panes</td>
<td>Bootable system</td>
</tr>
</tbody>
</table>
4.2 Planning Your Projects

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

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

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

4.2.1 Getting a Functional BSP

To get a functioning BSP, you can:

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

Tornado 2.2 BSP

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

Tornado 2.0 BSP

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

Third-party or Tornado 1.0.1 BSP

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

You may wish to enable the Tornado 1.0.1 compatibility mode, which exposes menu items to execute BSP builds in a BSP directory. (See 12. Customization.) Once your BSP builds, you may proceed to create a bootable project from it immediately. See 5.7 Building Projects from a BSP, p.195.

Using a Custom BSP For Custom Hardware

Creating a BSP

If you need to create your own BSP, refer to the VxWorks BSP Developer’s Guide (a separate product available from Wind River). If you wish to develop the BSP and the application code in parallel, you may wish to begin application development on the VxWorks simulator. See Using the Simulator BSP, p.83.

Using a Pre-Existing BSP with the Project Facility

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

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

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

Using the Simulator BSP

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

4.2.2 Creating a Bootable Project Based on a BSP

Using the VxWorks Simulator

Integrated Simulator With Basic Functionality

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

Integrated Simulator With Added Functionality

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

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

NOTE: If you do not make your BSP Tornado 2.0 compliant, Tornado will not be able to provide project-based support for customizing, configuring, or building it.
Networking: If you need network support, you will need the VxWorks full simulator (VxSim), which can be purchased from Wind River as an optional product.

The process for creating a project, and configuring it, is identical for both the integrated and full simulator.

Base this project on the simulator BSP (either the integrated simulator or the optional product), or the pre-built, default simulator project (*simhost_vx.wpj*). At this point, your project will build an image identical to the integrated simulator as you received it from Wind River. Now add any components you need using the VxWorks tab in the Workspace view (see 4.4.3 Configuring VxWorks Components, p.116).

**Using a Real Target**

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

**Image Size Considerations**

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

**4.2.3 Developing and Adding Your Application Source Code**

**Adding Existing Application Source Code**

**Use Your Existing Build System**

If you already have a working application, or if your application is very large, you may want to use your own build system. You can use the project facility to link the output of an external build into VxWorks and even start external builds (see External Build System, p.87).
Alternatively, you may want to use the project facility to configure your VxWorks image and produce a makefile. You can build your application outside Tornado and call the project facility-generated makefile from your build to produce a final image.

**Integrate It with Your Bootable Project**

Use this approach if your edit-compile-reboot cycle is relatively quick. Add the files to the bootable project using the Add File(s) to Project context menu available in the File tab of the Workspace view. Then edit the VxWorks initialization file, `usrAppInit.c`, adding calls to your application’s initialization and startup routines. The VxWorks application initialization component is required, and is included by default. See 4.4 Creating a Custom VxWorks Image, p.109 and 4.5 Creating a Bootable Application, p.128.

**Create a Separate, Downloadable Project for Your Code**

Use this method if rebooting your target is inconvenient, and if your code is modular enough that it can be added to the running target without interrupting execution or if you have the means to start and stop your application. Create a downloadable project using the Downloadable Project Wizard. Add application files with the Add File(s) to Project context menu. Build your downloadable project. Boot the target using the appropriate image described in 4.2.2 Creating a Bootable Project Based on a BSP, p.83. Download the partially linked and munched `.out` file produced by your project. See 4.3 Creating a Downloadable Application, p.93.

**Creating New Application Source Code**

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

**Building with Custom Build Rules**

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

---

2. For information about munching, see the VxWorks Programmer’s Guide: C++ Development.
Create a Build Rule Specific to the Source File

This ensures that the custom rule will be invoked only to process the specified source file. For example, you may wish to add a custom rule to process a yacc file into a C source file. To create a custom rule, see 4.6 Working With Build Specifications, p.129.

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

Create a Custom Rule for the Build

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

**NOTE:** Custom build rules cannot be copied between projects. If you will use either form of custom build rules, and know that you will be migrating files between projects, you may wish to put files with similar build settings into separate projects. These projects can then be built and linked together. For more information, see the hierarchical sub-project model discussed in Sub-Projects, p.88.

**Developing Architecture-Independent Applications**

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

**Migrating Files**

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

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

If you have a number of files that must be built with special build rules or flags, it may be easier to create a new project to build these particular files, and then build that project as a sub-project of your main project. For more information, see Structuring Your Projects, p.87.

Using Configuration Management

Tornado provides configuration management integration. To enable and configure it, see 12. Customization.

Configuring VxWorks

VxWorks must be configured to support the calls your application makes to it, or you will not be able to link your image. If your BSP provides a “bare-bones” VxWorks configuration, you may wish to use Project>Auto Scale to detect and add most of the VxWorks functionality you require. Auto Scale will compile your code, analyze the symbols in your object modules, map them to components, and offer to include those components. There may be some components that Auto Scale does not detect. If you Auto Scale, build, and still get link errors, you will need to add the additional components from the workspace VxWorks tab. For information on using Auto Scale, see 4.5.1 Using Automated Scaling of VxWorks, p.128.

Structuring Your Projects

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

Single Project

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

External Build System

The Tornado workspace is very convenient for configuring VxWorks, building small applications, or building, downloading, and debugging small parts of a large application. It is not designed to handle a complete build of a large, modular
application, which often requires sub-projects (though this can be achieved using custom rules and macros). For this reason, you may want to use an external build system to build your application, then link it to VxWorks using the EXTRA_MODULES or LIBS macros. You can write a custom rule to invoke your external build process; see Sub-Projects, p. 88. Alternatively, your build can kick off a VxWorks build and link your application code as the final step.

Sub-Projects

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

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

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

Example 1 Using Sub-Projects

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

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

Assumptions:

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

- The two downloadable projects are called Project1 and Project2, and they also reside in directories of the same name. Each contains a build specification called SIMNTgnu based on the PC simulator toolchain.
• **Project1** contains a C source file called `foo.c`, containing a function called `Test()`.

• **Project2** contains a C source file `goo.c`, which in turn contains a function called `Test2()`.

• `Test()`, which calls `Test2()`, is the main application entry point.

• Dependencies have been generated for each of the two downloadable projects, and they have been saved. This creates makefiles for them. Without the makefiles, the build fails.

Go first to the Build tab. Expand the project **Master**. Double-click on the build specification default to display the build property sheet. In the build property sheet, select the Rules page. Click New/Edit and add the new rules.

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

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

**SYNTAX:**
```
target : dependencies
  commands
```

**RULES:**
```makefile
..//Project1/SIMNTgnu :
  - mkdir $@

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

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

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

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

CleanProject1 : ../../Project1/SIMNTgnu
  $(MAKE) -C ../../Project1/SIMNTgnu -f ../../Project1/Makefile
     BUILD_SPEC=SIMNTgnu clean
```
CleanProject2 : ../../Project2/SIMNTgnu

$make -C ../../Project2/SIMNTgnu -f ../../Project2/Makefile

BUILD_SPEC=SIMNTgnu clean

clean : CleanProject1 CleanProject2

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

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

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

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

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

```c
void Test(void);

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

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

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

NOTE: The sub-project objects in the example (Project1.pl and Project2.pl) need not have been generated by Tornado downloadable projects. They could also have been the result of an external build system.
To modify this example to integrate an external application build system, you could, for instance:

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

Example 4-2 **Avoiding Absolute Paths**

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

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

If you cannot organize your source files and projects in this way, we provide two environment variables to allow you to define the root of your source directory tree or your project directory tree: \texttt{WIND\_SOURCE\_BASE} and \texttt{WIND\_PROJ\_BASE}.

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

1. **Avoiding absolute paths to source files**

Suppose your project is in directory \texttt{t:source\_root\myproj} and your source files are organized as follows:

\begin{verbatim}
  t:\source_root\myproj\foo1.c
  t:\source_root\src\foo2.c
  t:\source_root\myproj\src\foo3.c
\end{verbatim}

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

\begin{verbatim}
$(PRJ\_DIR)/foo1.c
$(PRJ\_DIR)/../src/foo2.c
$(PRJ\_DIR)/src/foo3.c
\end{verbatim}
Organizing your source files and projects in any of these ways is the recommended procedure for avoiding absolute paths in project files and makefiles.

If you cannot organize your source files and projects in one of these ways, the \texttt{WIND\_SOURCE\_BASE} environment variable allows you to define the root of your source directory tree. To illustrate the use of \texttt{WIND\_SOURCE\_BASE}, assume you wish to add the file:

\texttt{t:\other\_source\_root\goo\_c}

If \texttt{WIND\_SOURCE\_BASE} is not defined, it appears in the project file with an absolute path (we do not support \texttt{(PRJ\_DIR)/././}). However, before you add the file, you can use Tools->Options->Workspace to define:

\texttt{WIND\_SOURCE\_BASE = t:\other\_source\_root}

Then \texttt{t:\other\_source\_root\goo\_c} is recorded in the project file as:

\texttt{$(WIND\_SOURCE\_BASE)/goo\_c}$

The major drawback to the \texttt{WIND\_SOURCE\_BASE} variable is that all users of \texttt{myproj} must have \texttt{WIND\_SOURCE\_BASE} defined or the workspace cannot find \texttt{goo\_c}.

2. Avoiding absolute paths to sub-projects

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

Assume the master project is in directory \texttt{t:\prj\root\master} and your sub-projects are organized as follows:

\texttt{t:\prj\root\sub1}
\texttt{t:\prj\root\master\sub2}
\texttt{t:\prj\root\subprojects\sub3}
\texttt{t:\prj\root\master\subprojects\sub4}

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

\texttt{$(PRJ\_DIR)/../sub1}
\texttt{$(PRJ\_DIR)/sub2}
\texttt{$(PRJ\_DIR)/../subprojects/sub3}
\texttt{$(PRJ\_DIR)/subprojects/sub4}$
Organizing your projects in any of the above ways is the recommended procedure for avoiding absolute paths in project files and makefiles.

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

\[ t:\\text{other\_prj\_root}\backslash\text{sub5} \]

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

\[ \text{WIND\_PROJ\_BASE=\text{t:\\text{other\_prj\_root}}} \]

Now when you add \text{t:\other\_prj\_root\sub5}, this sub-project will be recorded in the master project file as:

\[ $(\text{WIND\_PROJ\_BASE})/\text{sub5} \]

The major drawback to the `WIND_PROJ_BASE` variable is that all users of the master project have to define `WIND_PROJ_BASE`, or the workspace is unable to find \text{sub5}.

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

### 4.3 Creating a Downloadable Application

A downloadable application is a collection of relocateable object modules that can be downloaded and dynamically linked to VxWorks, and started from the shell or debugger. A downloadable application can consist of a single “hello world” routine or a complex application.
To create a downloadable application, you must:
1. Create a project for a downloadable application.
2. Write your application, or use an existing one.
3. Add the application files to the project.
4. Build the project.

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

### 4.3.1 Creating a Project for a Downloadable Application

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

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.

3. You can modify the default behavior by un-checking the **Show this window on startup** box at the bottom of the window.
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 (Figure 4-4). Finally, identify the workspace in which the project should be created. Click Next to continue.

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

Identify the toolchain with which the downloadable application will be built. You can do so by referencing an existing project, or by identifying a toolchain. Basing a project on an existing one means that the new project will reference the same source files and build specifications as the one on which it was based. Once the new project has been created, its build specifications can be modified without affecting the original project, but changes to any shared source files will be reflected in both.

For example, to create a project that will run on the target simulator, select A toolchain and select the default option for the target simulator from the drop-down list (Figure 4-5). Click Next.
The wizard confirms your selections (Figure 4-6). Click Finish.

4. The default toolchain names for target simulators take the form SIMHOST_OSGnu (for example, SIMNTgnu).
The Workspace window appears, containing a folder for the project. Note that the window title includes the name of the workspace (Figure 4-7).

Figure 4-7  Initial Workspace Window for a Downloadable Application

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

4.3.2 Project Files for a Downloadable Application

The project facility generates a set of files whose contents are based on your selection of project type, toolchain, build options, and build configurations. During typical use of the project facility you need not be concerned with these files, except to avoid accidental deletion, to check them in or out of a source management system, or to share your projects or workspaces with others. The files are created in the directories you identify for the workspace and project. The files initially created are:

`projectName.wpj`
Contains information about the project used for generating the project makefile.
workspaceName.wsp

Contains information about the workspace, including which projects belong to it.

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

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

4.3.3 Working With Application Files

The Files view of the Workspace window displays information about the projects, and the directories and files that make up a project (Figure 4-8).

Figure 4-8  Workspace Files View

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

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

• An External Dependencies folder, which contains a list of make dependencies. The list is automatically generated by the project facility.

Initially, there are only the default folders for Built Objects and External Dependencies, and the projectName.out file. The file projectName.out is created as a single, partially-linked module when the project is built. It comprises all of the individual object modules in a project for a downloadable application, and provides for downloading them 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).

Figure 4-9  New File Dialog Box

Add existing files to a project by right-clicking in the Workspace window, selecting Add Files or Add Files from project from the pop-up menu, and then using the associated dialog box to locate and select the file(s).
To link object files with your project, use the PRJ_LIBS macro or the Linker page of the build specification property sheet (see Linker Options, p.135). 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.132).

Remove files from the project by right-clicking on the file name and selecting Remove from the pop-up menu, or by selecting the file name and pressing DELETE.

⚠️ CAUTION: Adding a file to a project or removing a file from a project does not affect its existence in the file system. The project facility does not copy, move, or delete user source files; merely the project facility’s references to them. Removing a file from one workspace context does not affect references to it in any others, nor its existence on disk. However, if a file is included in more than one project or workspace, an edit made in one context will be reflected in all. (If this behavior is not desired, copy source files to another directory before adding them to a project.)

Displaying and Modifying File Properties

To display information about the properties of a file, right-click on the file name in the Workspace window, and select Properties from the pop-up menu. The extent of information displayed depends on the type of file and whether or not make dependencies have been generated. In the case of source code, a Properties sheet for the file appears, displaying information about make dependencies; general file attributes such as modification date; and the associated make target, custom dependencies, and commands used for the build process (Figure 4-10).

See Calculating Makefile Dependencies, p.102, for information about how and when to calculate makefile dependencies. See Compiler Options, p.132 for information about overriding default compiler options for individual files.

Opening, Saving, and Closing Files

The File and pop-up menus provide 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 12. Customization for information about customizing the editor, including how it handles DOS and UNIX end-of-line markers.
4.3.4 Building a Downloadable Application

The project facility uses the GNU make utility to automate compiling and linking an application. It automatically creates a makefile prior to building the project. But before it can create a makefile, the makefile dependencies must be calculated. The calculation process, which is based on the project files’ preprocessor #include statements, is also an automated feature of the project facility.

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 new project for it in the same workspace, and customize the build specification for those files. One project’s build specification can then be modified to link in the output from the other project. See *Sub-Projects*, p.88.

---

5. See the GNU Make User’s Guide for more information about make.
Calculating Makefile Dependencies

To calculate makefile dependencies, select Dependencies from the workspace pop-up 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 prompts you to do so for any of the project files for which dependencies have not previously been calculated. Dependencies are not calculated for each build specification. If your dependencies change for different build specifications (for example, if they are CPU-dependent), then you may want to:

- Create a new project for each build specification.
- Regenerate dependencies when you switch build specifications.

Tornado assumes that your header files are in either your project directory or installDir\target\h. If you have placed files in other locations, you need to make two changes to your project build specification. Right-click on the name of your build specification (for example, SIMNTgnu). Select Properties from the pop-up
menu. On the C/C++ compiler tab, click the Include paths button. Add a separate entry for each directory path.

The Advanced option allows you to speed up the build process by specifying paths in which none of the dependencies could have changed since the last build. The timestamps for the files in the specified paths are not checked (Figure 4-13).

**Build Specifications**

Each build specification for a downloadable application consists of a set of options for makefile rules and macros, as well as for the compiler, assembler, and linker. A default build specification is defined when you create your project. To display its property sheet, double-click on the build specification name in the Builds view of the workspace to display the property sheet for the build specification.

**Rules Tab**

The Rules page (Figure 4-14) allows you to select from the following build target options:
Objects for all source files in the project.

Archive

An archive (library) file.
A single, partially-linked and munched object that comprises all of the individual object modules in a project. This is the correct module to download. For information on munching, see the VxWorks Programmer’s Guide: C++ Development. For more information on linking, see Linker Options, p.135.

projectName.pl
A single, partially-linked but not munched object that comprises all of the individual objects modules in a project. This file is provided for sub-project support. It is not intended for download since it has not been munched. See Linker Options, p.135.

You can use the project facility to change the options for a given build specification, create and save new build specifications, and select the specification to use for a build. You can, for example, create one build specification for your project that includes debug information, and another that does not. For more information, see 4.6 Working With Build Specifications, p.129.

**NOTE:** It is sometimes useful to build an application for the target simulator, and then to create a new build specification to build it for a real target.

**Macros Tab**

The Macros tab contains pre-set build macros. Do not delete the pre-existing macros; while you can reenter them, the value will be lost. You can add and delete your own macros by typing in the Name window and then clicking the Add button.

Macros that are useful with bootable projects:

- **EXTRA_MODULES** Extra object modules to link into the VxWorks image.
- **LIBS** Libraries against which VxWorks is linked.
- **POST_BUILD_RULE** Shell commands to execute after the build has completed.
- **RAM_HIGH_ADRS** RAM address where the boot ROM data segment is loaded. It must be a high enough value to ensure loading VxWorks does not overwrite part of the ROM program.
- **RAM_LOW_ADRS** Beginning address to use for the VxWorks run-time in RAM.

**WARNING:** RAM_HIGH_ADRS and RAM_LOW_ADRS are also defined in config.h; the two definitions must match!
Macros that are useful with downloadable projects:

**PRJ_LIBS** Libraries or modules against which a downloadable application is linked.

**POST_BUILD_RULE** Shell commands to execute after the build has completed.

**Environment Variables**

If you are using the Diab tools, you must have two settings in place:

- Be sure that `installDir\host\diab\WIN32\bin` is in the system path.
- Be sure that the environment variable `DIABLIBLE` is set to `installDir\host\diab`.

There is a batch file called `torVars.bat` in `installDir\host\x86-win32\bin` that will set `DIABLIBLE` for you.

**Building an Application**

To build a project with the default options, select the name of the project (or any subordinate object in its folder) and then select `Build 'projectName.out'` from the pop-up menu. If you have created build specifications in addition to the default, you can select the build specification you want to use from the Build 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.
To force a rebuild of all project objects, select Rebuild All from the pop-up menu (which performs a make clean before the build).

Build Toolbar

The Build toolbar provides quick access to build commands. Display of the toolbar 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.
The Build toolbar commands (Table 4-2) are also available from the main menus and the Workspace pop-up menu.

Table 4-2  Build Toolbar Buttons

<table>
<thead>
<tr>
<th>Button</th>
<th>Menu</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build&gt;Build</td>
<td>Build</td>
<td>Build project.</td>
</tr>
<tr>
<td>Build&gt;Rebuild All</td>
<td>Rebuild All</td>
<td>Rebuild project (performing a make clean first).</td>
</tr>
<tr>
<td>Build&gt;Compile</td>
<td>Compile</td>
<td>Compile selected source file.</td>
</tr>
<tr>
<td>Build&gt;Dependencies</td>
<td>Dependencies</td>
<td>Update dependencies.</td>
</tr>
<tr>
<td>Build&gt;Stop Build</td>
<td>Stop Build</td>
<td>Stop build.</td>
</tr>
<tr>
<td>Project&gt;Download</td>
<td>Download</td>
<td>Download object file (or boot image for target simulator).</td>
</tr>
</tbody>
</table>

4.3.5 Downloading an Application

Before you can download and run an application, you must boot VxWorks on the target system, have access to a Tornado registry, and configure and start a target server. For more information, see 2. Setup and Startup and 8. Target Server.

You can download an entire project from the project workspace by selecting Download 'project\Name.out' from the pop-up menu for the Files view, or by using the download button on the Build toolbar. You can download individual object modules by selecting the file name and then selecting the Download 'filename.o' option from the pop-up menu. However, you may inadvertently introduce errors by downloading individual object modules out of sequence. We strongly recommend that you always download the partially-linked project\Name.out file.

C++ projects must be downloaded as project\Name.out because this file is produced from application files and munched for proper static constructor initialization.
To run a downloaded application, use WindSh or CrossWind. For more information see 7. Shell or 10. Debugger.

To unload a project from the target, use theUnload 'projectName.out' option on the pop-up menu.

4.3.6 Adding and Removing Projects

New projects can be added to a workspace by selecting 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).

Projects can be removed from a workspace by selecting the project name in the Files view, and then selecting the Remove option from the pop-up menu, or by selecting the project name and pressing DELETE.

NOTE: When you remove a project, you only remove it from the workspace. The project directory and its associated files are not removed from disk.

4.4 Creating a Custom VxWorks Image

The Tornado distribution includes a VxWorks system image for each target shipped. The system image is a binary module that can be booted and run on a target system. The system image consists of all desired system object modules linked together into a single non-relocateable object module with no unresolved external references. In most cases, you will find the supplied system image adequate for initial development. However, later in the cycle you may want to create a custom VxWorks image.

VxWorks is a flexible, scalable operating system with numerous facilities that can be tuned, and included or excluded, depending on the requirements of your application and the stage of the development cycle. For example, various networking and file system components may be required for one application and not another, and the project facility provides a simple means for either including them in, or excluding them from, a VxWorks application. In addition, it may be
useful to build VxWorks with various target tools during development (such as the target-resident shell), and then exclude them from the production application.

Once you create a customized VxWorks, you can boot your target with it and then download and run applications. You can also create a bootable application simply by linking your application to VxWorks and adding application startup calls to the VxWorks system initialization routines (see 4.5 Creating a Bootable Application, p.128).

### 4.4.1 Creating a Project for VxWorks

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

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).

The application wizard appears (Figure 4-18). This wizard is a tool that guides you through the steps of creating a new project.

---

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

![Figure 4-18 Application Wizard: Step One for Bootable Application](image)

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.
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 mbx860 target, select An existing project and then select mbx860_gnu.wpj (or mbx860_diab.wpj if you have purchased the Diab tools) from the drop-down list (Figure 4-19). Click Next.

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

NOTE: If you are creating a customized VxWorks image or a bootable application, the project will be generated faster if you base it on an existing project rather than on a BSP. This is because the project facility does not have to regenerate configuration information from BSP configuration files. All Tornado 2.x BSPs include both GNU and Diab project files for this purpose. Options for BSP projects are available in the drop-down list for existing projects. For example, the mbx860 BSP project files are:

- `installDir\target\proj\mbx860_gnu\mbx860_gnu.wpj`
- `installDir\target\proj\mbx860_diab\mbx860_diab.wpj`
If, on the other hand, you must base your project on a BSP, select A BSP and choose one of the BSP’s you installed or the appropriate simulator. If your BSP offers both GNU and Diab toolchains, select a toolchain as well. Click Next.

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

The Workspace window appears.

### 4.4.2 Project Files for VxWorks

The project facility generates, or includes copies of, a variety of files for a VxWorks project. The names of the files that you may need to work with are displayed in the workspace File view (Figure 4-21).
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 `usrAppInit.c`, however, when you create a bootable application (see 4.5 Creating a Bootable Application, p.128).

CAUTION: Do not check in `linkSyms.c`, `prjConfig.c`, `prjComps.h`, or `prjParams.h`; these files are regenerated whenever the project file changes.

The VxWorks project files serve the following purposes:

- **linkSyms.c**
  A dynamically generated configuration file (therefore not to be checked in to your version control system) that includes code from the VxWorks archive by creating references to the appropriate symbols. It contains symbols for components that do not have initialization routines.
prjConfig.c
A dynamically generated configuration file (therefore not to be checked in to your version control system) 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
A dynamically generated configuration file (therefore not to be checked in) that contains the preprocessor definitions (macros) used to include VxWorks components.

Makefile
The makefile used for building an application or VxWorks. Created when the project is built, based on the build specification selected at that time.

prjParams.h
A dynamically generated configuration file (therefore not to be checked in) that contains component parameters.

projectName.wpj
Contains information about the project used for generating the project makefile, as well as project source files such as prjConfig.c.

workspaceName.wsp
Contains information about the workspace, including which projects belong to it.

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

You can also build your project from the command line. When you do so, you must create the makefile first. See Using the Command Line, p.127.

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.

### 4.4.3 Configuring VxWorks Components

The VxWorks view of the workspace displays all VxWorks components available for the target. The names of components that are selected for inclusion appear in **bold** type. The names of components that are excluded appear in plain type. The names of components that have not been installed appear in *italics*. Note that the names of folders appear in bold type if any (but not necessarily all) of their components are included. (Figure 4-22.)

Figure 4-22 **VxWorks Components**
Finding VxWorks Components and Configuration Macros

You can locate individual components and configuration parameters in the component tree, based on their macro names, with the Find Object dialog box. Open the dialog box with the pop-up menu for the VxWorks view (Figure 4-23).

Figure 4-23  Find Object

NOTE: The Find Object dialog box is particularly helpful in conjunction with VxWorks documentation, which discusses VxWorks configuration in terms of preprocessor symbols, rather than the descriptive names used in the project facility GUI.

Displaying Descriptions and Online Help for Components

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

To display online reference documentation, double-click on the topic of your choice displayed in the Help Link box of the property sheet. The corresponding HTML reference material is displayed in a Web browser (Figure 4-24).
Including and Excluding Components

VxWorks components that are not needed for a project can be excluded, and components that have been excluded can be included again. The pop-up menu provides Include and Exclude options for components you select in the VxWorks view. You can also use the DELETE key to exclude options.
Projects

Tornado automatically determines component dependencies each time a component is included or excluded. That is, it determines if a component you want to include is dependent upon other components that have not been included in the project, or if a component that you are deleting is required by other components. When a component is included, any components it requires are automatically included. When a component is excluded, any dependent components are also excluded. In either case, a dialog box provides information about dependencies and the option of cancelling the requested action. For example, if you exclude POSIX clocks, the dialog box informs you that the ANSI time component would be excluded (Figure 4-25).

![Exclude VxWorks Component](image)

**Figure 4-25 Exclude VxWorks Component**

**WARNING:** The results of calculating dependencies is not necessarily identical for inclusion and removal. Including a component you previously excluded does not automatically include the components that were dependent on that component, and that were therefore excluded with it. For example, excluding the POSIX clocks component automatically excludes the ANSI time component, which is dependent on it. But if the POSIX clocks component is subsequently included, there are no components required by it, so the ANSI time component is not automatically included (Figure 4-26).
You can also include folders of components. However, not all components in a folder are necessarily included by default (nor would it always be desirable to do so, as there might be conflicts between components). Tornado offers a choice about what components to include. For example, if you include target shell components, not all of the components are included by default, and you are prompted to accept or modify the default selection (Figure 4-27).

Tornado automatically calculates an estimate of the change in the size of the image resulting from the inclusion or exclusion, as well as the new image size. The Include and Exclude dialog boxes display this information. (Also see Estimating Total Component Size, p.123).

Some folders contain component options that are explicitly combinative or mutually exclusive (in the sense of being potentially in conflict). These folders are called selections, and their names are preceded by a checkbox icon in the folder tree. You can make or change your selection 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 4-27 Including a Component Folder

Figure 4-28 Including Conflicting Components
Component Conflicts

If you include components that potentially conflict, or are missing a required component, Tornado warns you of the conflict by displaying a message box with a warning, and by highlighting the full folder path to the source of the conflict. The property sheet for the folder also displays error information in its Errors page. For example, if you attempt to include both symbol table initialization components, a warning is first displayed. Once you acknowledge the warning, the folder names development tool components, symbol table components, select symbol table initialization are highlighted. You can display the property sheet for the folder for a description of the problem and how to correct it. (See Figure 4-29 for all GUI elements.)

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

In the VxWorks view, the pop-up menu provides access to component parameters (preprocessor macros). For example, selecting the operating system components folder, then Params for 'operating system components' from the pop-up menu (or double-clicking on the folder name), displays a dialog box that allows you to change the values of the parameters defined for the operating system components (Figure 4-30). Parameters specific to individual components can be accessed similarly.

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

Estimating Total Component Size

To calculate and display the estimated size of the components included in an image, select the project name (in any of the workspace views), then select Properties from the pop-up menu, and select the Size tab in the property sheet that appears (Figure 4-31). Note that this estimate is for the components only, and does not include the BSP or any application code.
4.4.4 Selecting the VxWorks Image Type

The default VxWorks is a RAM-based image. If you want to create something other than the default images, use one of the other build specifications created when you created your project:

- default_rom
- default_romCompress
- default_romResident

These build specifications are identical to the default, except that instead of building the \texttt{vxWorks} rule, \texttt{default\_rom}, for example, is set up to build the \texttt{vxWorks\_rom} rule. (Figure 4-32).

The options available for a VxWorks image are:

\begin{itemize}
  \item \texttt{vxWorks}
    A RAM-based image, usually loaded into memory by a VxWorks boot ROM. This is the default development image.
  \item \texttt{default\_rom}
    A ROM-based image that copies itself to RAM before executing. This image generally has a slower startup time, but a faster execution time than \texttt{vxWorks\_romResident}. It is also available in alternate formats as \texttt{vxWorks\_rom.bin}
\end{itemize}
and vxWorks_rom.hex. The .hex options are variants of the main options, with Motorola S-Record output. The .bin options are variants of the main options with binary output.

default_romCompress

A compressed ROM image that copies itself to RAM and decompresses before executing. It takes longer to boot than vxWorks_rom but takes up less space than other ROM-based images (nearly half the size). The run-time execution is the same speed as vxWorks_rom. It is also available in .bin and .hex formats.

default_romResident

A ROM-resident image. Only the data segment is copied to RAM on startup. It has the fastest startup time and uses the smallest amount of RAM. Typically, however, it runs slower than the other ROM images because ROM access is slower. It is also available in .bin and .hex formats.

NOTE: Project files used only for a ROM-based image can be flagged as such, so that they are only used when a ROM-based image is built. See Compiler Options, p.132.

### 4.4.5 Building VxWorks

VxWorks projects are built in the same manner as downloadable applications. To build a project with the default options, select the name of the project (or any subordinate object in its folder) and then select the Build option from the pop-up menu. The name of the build specification that will be used is displayed in the Build drop-down list at the top of the workspace window.
For more information about a generic build, see 4.3.4 Building a Downloadable Application, p. 101. For information about modifying builds and creating new build configurations, see 4.6 Working With Build Specifications, p. 129.

**Using the Build Menu**

The build menu allows you to choose boot ROMs or BSPs to build. Figure 4-33 illustrates the Build dialog box in a Tornado system that has a PowerPC BSP and the Diab compiler installed.

![Figure 4-33 Rebuilding VxWorks from the Build Menu](image)

Select a BSP
The drop-down list includes all BSPs you have installed. You will have at least one integrated simulator BSP, and you have probably installed at least one additional BSP.

Select an Image to build
The available BSP make targets appear in the drop down list. For information on the targets, see Build Specifications, p. 103.

The standard make target clean (which erases all objects that can be built by the BSP makefile) is also in this list.

⚠️ **WARNING**: Be sure not to use make clean in your VxWorks tree, in other words, in installDir\target\src. The make clean command is designed to force a complete rebuild of your project files. If you use it in installDir\target\src, you will remove VxWorks objects for which you do not have source and which you will therefore be unable to recreate.
Select a tool

If you have installed both the GNU and Diab compilers, you can select which one to use.

When you click OK in the build dialog box, Tornado builds the corresponding object in the BSP directory. Output from the build goes to a Build Output window, which you can use as a diagnostic aid.

To rebuild VxWorks, select the vxWorks target name in the Select an Image list. For example, Figure 4-33 shows the vxWorks target selected for the mbx860 BSP.

NOTE: All source files in a project are built using a single build specification (which includes a specific set of makefile, compiler, and linker options) at a time. If some of your source requires a different build specification from the rest, you can create a project for it in the same workspace, and customize the build specification for those files. One project’s build specification can then be modified to link in the output from the other project. See Linker Options, p.135.

WARNING: The default compiler options include debugging information. Using debugging with the optimization set to anything but zero may produce unexpected results. See 4.6 Working With Build Specifications, p.129 for information about modifying builds and creating new build configurations.

Using the Command Line

Using the command line allows you to automate builds. Projects must be created and configured in the GUI, and dependencies must be generated there as well. However, makefile generation and building are possible from the command line.

To generate the makefile, use the makegen utility and to build, use make.

Change to the build directory and type make with flags, for example:

```
% cd installDir/target/proj/Project1/mbx860_gnu
% makegen
% mkdir default_rom
% cd default_rom
% make -f ../Makefile BUILDSPEC=default_rom
```

Your build output will be in the default_rom directory and the build will use the defaultRom build specification.
4.4.6 Booting VxWorks

For information about booting VxWorks (and bootable applications), see 2.5 Booting VxWorks, p. 41. VxWorks images for the target simulator can be downloaded and booted with the pop-up menu Start command.

4.5 Creating a Bootable Application

A bootable application is completely initialized and functional after a target has been booted, without requiring interaction with Tornado development tools. For information about developing your application, 4.2.3 Developing and Adding Your Application Source Code, p. 84.

4.5.1 Using Automated Scaling of VxWorks

The auto scale feature of the project facility determines if your code, or your custom version of VxWorks, requires any components that are not included in your VxWorks project, and adds them as required. It also provides information about components that may not be required for your application. To automatically scale VxWorks, select Auto Scale from the pop-up menu in the VxWorks view of the workspace window to display the Auto Scale dialog box, and click OK.

NOTE: The auto scale feature detects only statically calculable dependencies between the application code and VxWorks. Some components may be needed even if they are not called by your application. This is especially true for servers such as WDB, NFS, and so on.

4.5.2 Adding Application Initialization Routines

When VxWorks boots, it initializes 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 calls to usrAppInit(). Figure 4-34, for example, illustrates the addition of a call to runItAll(), the main routine in the application file helloWorld.c.
4.6 Working With Build Specifications

The project facility allows you to create, modify, and select specifications for any number of builds. Default build specifications are defined when you create your project.

- While a BSP is usually designed for one CPU, you can create build specifications for different image types and optimization levels, specifications for builds that include debugging information and builds that do not, and so on.
- A downloadable project can have build specifications for different CPUs, for example, for a simulator and a real target.
4.6.1 Changing a Build Specification

Each build specification consists of a set of options that define the VxWorks image type (for VxWorks and bootable application projects), makefile rules, macros, as well as compiler, assembler, and linker options.

NOTE: For detailed information about compiler, assembler, and linker options, see the GNU ToolKit User’s Guide or the Diab C/C++ Compiler User’s Guide.

You can change default or other previously defined build options by double-clicking on the build name in the Builds view of the workspace window. The build’s property sheet appears (Figure 4-35).

Figure 4-35  Build Property Sheet
You can use the property sheet to modify:

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

For information about build targets for downloadable applications, see *Build Specifications*, p. 103. For information about build targets for bootable applications, see 4.4.4 Selecting the VxWorks Image Type, p. 124. Other features of the build property sheet are covered in the following sections.

**Custom Makefile Rules**

The buttons at the bottom of the build property sheet allow you to create, edit, or delete makefile rules (default project entries cannot be deleted; only those created by a user can be deleted). When you click the New/Edit button, the Create or Edit Rule dialog box appears (Figure 4-36). Once you have created or edited an entry, click OK. Note that the default is to invoke the rule before building the project (see checkbox). If the default is not selected, the rule is only invoked if it is the rule currently selected for the build (with the drop-down list in the Rules page of the build property sheet) or if it is a dependency of the currently selected rule. New rules are added to the *projectName.wpj* file and written to the makefile prior to a build.

![Figure 4-36 Makefile Rule](image-url)
Makefile Macros

Select the Macros tab of the build specification property sheet to view the makefile macros associated with the current project, build specification, and rules (Figure 4-37).

Figure 4-37  Makefile Macros

You can use the Macros page to modify the values of existing makefile macros, as well as to create new rules to be executed at the end of the build. Use the Delete button to delete a macro from the build. To change an existing macro, modify the value and click the Add/Set button. To add a macro, change the name and value of an existing macro, and click the Add/Set button.

The recommended way to link library (archive) files to your bootable project is to add the libraries to the list defined by the LIBS macro and the modules to the list defined by the EXTRA_MODULES macro. Use the PRJ_LIBS macro for downloadable projects.

Compiler Options

The C/C++ compiler page of the build specification property sheet displays compiler options. You can edit the options displayed in the text box (Figure 4-38). You can also click the Include Paths button and use the dialog box to enter and order your include paths.
If you use the -I option to add include file paths for the build, you must use forward slashes (/) rather than standard Windows backward slashes (\). Failure to do so causes dependencies from directories other than the current directory to try to make a target of the header file, preventing you from building real targets.

You can override the default compiler flags for individual files by right-clicking on the file name in the Files view, selecting Properties from the pop-up menu, and specifying a new set of options in the Build page of the property sheet. Unchecking the Use default build rule for this file box allows you to edit the fields in this page (Figure 4-40).

---

**WARNING:** The default compiler options include debugging information. Using debug information with the optimization option set to anything but zero may produce unpredictable results. Selecting Include debug info automatically sets optimization to zero. This can be changed by editing the option.
If the file should be used only when building a ROM-based image, check the Build for ROM images only box. See 4.4.4 Selecting the VxWorks Image Type, p.124.

**Assembler Options**

Select the assembler tab of the build specification property sheet to view assembler options. You can edit the options displayed in the text box (Figure 4-41).

**Link Order Options**

Select the Link Order tab of the build specification property sheet to view module link order (Figure 4-42). You can change the link order using the Down and Up buttons to ensure that static C++ constructors and destructors are invoked in the correct order.
Linker Options

Select the linker tab of the build specification property sheet to view linker options. You can edit the options displayed in the text box (Figure 4-43).

To link an object or library (archive) file with a project, you can list the full path to the file here. However, the recommended way to link library (archive) files is to add the libraries to the list defined by the LIBS, EXTRA_MODULES, or PRJ_LIBS macros on the Macros tab (see Makefile Macros, p.132).

⚠️ 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.103) or as a partial link (projectName.pl), and then link it with the current project.
4.6.2 Creating New Build Specifications

You can create new build specifications for a project with the Add New Build Specification window, which is displayed with the New Build option on the pop-up menu. For example, one build specification can be created that includes debug information, and another that does not; specifications can be created for different image types, optimization levels, and so on. You can create a new build specification by copying from an existing specification, or by creating it as a default specification for a given toolchain (Figure 4-44).

Once you have created a new build specification, use the build specification property sheet to define it (see 4.6.1 Changing a Build Specification, p.130).

NOTE: Within a bootable project, you are restricted to the toolchains that support the CPU required by the BSP. You can still create multiple build specifications (for example, with different optimization levels or rules).
4.6.3 Selecting a Specification for the Current Build

When you want to build your project, select the build specification from the Build Spec drop-down list (Figure 4-45).

Figure 4-45  Build Specification Selection

Binaries produced by a build are created in the buildName subdirectory of your project directory.
4.7 Configuring the Target-Host Communication Interface

⚠️ **WARNING:** During development you must configure VxWorks with the target agent communication interface required for the connection between your host and target system (network, serial, NetROM, and so on). By default, VxWorks is configured for a network connection. Also note that before you use Tornado tools such as the shell and debugger, you must start a target server that is configured for the same mode of communication. See 2.4 Host-Target Communication Configuration, p.26; and 8.2 Configuring and Starting a Target Server, p.282.

To display the options for the communication interface for the target agent in the VxWorks view, select development tool components > WDB agent components > select WDB connection (Figure 4-46).

![Target Agent Connection Options](image)

To select an interface, select it from the list and select the Include 'component name' option from the pop-up menu. (You can also make a selection by double-clicking on the select WDB connection option to display the property sheet, and then making the selection from the Components page.)

To display general information about a component, or to change its parameters, simply double-click on its name, which displays its property sheet (see
Figure 4-47). The options for the target agent communication interface are described below.

**NOTE:** Also see *Scaling the Target Agent*, p.142 and *Starting the Agent Before the Kernel*, p.143.

### Configuration for an END Driver Connection

When VxWorks is configured with the standard network stack, the target agent can use an END (Enhanced Network driver) connection. Add the WDB END driver connection component. This connection has the same characteristics as the network connection, but also has a polled network interface that allows system and task mode debugging.

### Configuration for Integrated Target Simulators

To configure a target agent for an image that will run with the VxWorks integrated target simulator, add the WDB simulator pipe connection component.

### Configuration for NetROM Connection

To configure the target agent to use a NetROM communication path, add the WDB netROM connection component. (See 2.4.4 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.132.

WDB_NETROM_TYPE
The default value of 400 specifies the old 400 series.

WDB_NETROM_WIDTH
The value 1 indicates that the ROMs support 8-bit access. To change this to 16- or 32-bit access, specify the value 2 or 4, respectively.

The size of the NetROM dual-port RAM is 2 KB. The NetROM permits this 2 KB buffer to be assigned anywhere in the pod 0 memory space. The default position for the NetROM dual-port RAM is at the end of the pod 0 memory space. The following line in installDir\target\src\config\usrWdb.c specifies the offset of dual-port RAM from the start of the ROM address space.

\[ dpOffset = (WDB_ROM_SIZE - DUALPORT_SIZE) * WDB_NETROM_WIDTH; \]

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

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

To configure the target agent for use with a network connection, add the WDB network connection component. (See 2.4.1 Network Connections, p.26).

The default MTU is 1500 octets. To change it, display the component property sheet, select the Params tab, select the WDB_MTU item and change the value associated with it (Figure 4-48).

![Network Connection Macro](image)

**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.27).

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.

If your target has a single serial channel, you can use the target server virtual console to share the channel between the console and the target agent. You must configure your project with the CONSOLE_TTY parameter set to NONE and the WDB_TTY_CHANNEL parameter set to 0. See Console and Redirection, p.291 for more information regarding the target server virtual console.

---

7. VxWorks serial channels are numbered starting at zero. Thus Channel 1 corresponds to the second serial port if the board’s ports are labeled starting at 1. If your board has only one serial port, you must change WDB_TTY_CHANNEL to 0 (zero).
When multiplexing the virtual console with WDB communications, excessive output to the console may lead to target server connection failures. The following actions may help resolve this problem:

- Decrease the amount of data being transmitted to the virtual console from your application.
- Increase the timeout period of the target server (see Back End, p.286).
- Increase the baud rate of the target agent and the target server connection.

**Configuration for tyCoDrv Connection**

To configure a target agent to use a serial connection, add the WDB tyCoDrv connection component. Display the component property sheet and select the Params tab to display and modify macro values.

**Scaling the Target Agent**

In a memory-constrained system, you may wish to create a smaller agent. To reduce program text size, you can remove the following optional agent facilities:

- WDB banner (INCLUDE_WDB_BANNER)
- VIO driver (INCLUDE_WDB_VIO)
- WDB task creation (INCLUDE_WDB_START_NOTIFY)
- WDB user event (INCLUDE_WDB_USER_EVENT)
These components are in the development tool components>WDB agent components>WDB agent services folder path.

You can also reduce the maximum number of WDB breakpoints with the WDB_BP_MAX parameter of the WDB breakpoints component. If you are using a serial connection, you can also set the INCLUDE_WDB_TTY_TEST parameter to FALSE.

If you are using a communication path which supports both system and task mode agents, then by default both agents are started. Since each agent consumes target memory (for example, each agent has a separate execution stack), you may wish to exclude one of the agents from the target system. You can configure the target to use only a task-mode or only a system-mode agent with the WDP task debugging or WDB system debugging options (which are in the folder path development tool components>WDB agent components>select WDB mode).

Configuring the Target Agent for Exception Hooks

If your application (or BSP) uses excHookAdd() to handle exceptions, host tools will still be notified of all exceptions (including the ones handled by your exception hook). If you want to suppress host tool notifications, you must exclude the component WDB exception notification. When this component is excluded, the target server is not notified 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 G. VxWorks Initialization Timeline for an outline of the overall VxWorks initialization sequence.)

In some cases (for example, if you are doing a BSP port for the first time), you may want to start the agent before the kernel starts, and initialize the kernel under the control of the Tornado host tools. To make that change, perform the following steps when you configure VxWorks:
1. Choose a communication path that can support a system-mode agent (NetROM or raw serial). The END communication path cannot be used as it requires that the system be started before it is initialized.

2. Change your configuration so that only WDB system debugging is selected (in folder path development tool components > WDB agent components > select WDB mode). By default, the task mode starts two agents: a system-mode agent and a task-mode agent. Both agents begin executing at the same time, but the task-mode agent requires the kernel to be running.

3. Create a configuration descriptor file called `fileName.cdf` (for example, `wdb.cdf`) in your project directory that contains the following lines:

   ```
   InitGroup usrWdbInit {
       INIT_RTN    usrWdbInit (); wdbSystemSuspend ();
       _INIT_ORDER usrInit
       INIT_BEFORE INCLUDE_KERNEL
   }
   ```

   This causes the project code generator to make the `usrWdbInit()` call earlier in the initialization sequence. It will be called from `usrInit()`, just before the component kernel is started.8

   After the target server connects to the system-mode target agent, you can resume the system to start the kernel under the agent’s control. (See 7.2.7 Using the Shell for System Mode Debugging, p.242 for information on using system mode from the shell, and 10.6 System-Mode Debugging, p.359 for information on using it from the debugger.)

   After connecting to the target agent, set a breakpoint in `usrRoot()`, then continue the system. The routine `kernelInit()` starts the multi-tasking kernel with `usrRoot()` as the entry point for the first task. Before `kernelInit()` is called, interrupts are still locked. By the time `usrRoot()` is called, interrupts are unlocked.

   Errors before reaching the breakpoint in `usrRoot()` are most often caused by a stray interrupt: check that you have initialized the hardware properly in the BSP `sysHwInit()` routine. Once `sysHwInit()` is working properly, you no longer need to start the agent before the kernel.

---

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

The default boot image included with Tornado for your BSP is configured for a networked development environment. The boot image consists of a minimal VxWorks configuration and a boot loader mechanism. You need to configure and build a new boot program (and install it on your boot medium) if:

- You are working with a target that is not on a network.
- You do not have a target with NVRAM, and do not want to enter boot parameters at the target console each time it boots.
- You want to use an alternate boot process, such as booting over the Target Server File System (TSFS).

⚠️ CAUTION: When the agent is started before the kernel, there is no way for the host to get the agent’s attention until a breakpoint occurs. There are two reasons for this:
1) For the NetROM connection, the agent cannot spawn the NetROM polling task to check periodically for incoming packets from the host. 2) For other types of connections, only system mode is supported and the WDB communication channel is set to work in polled mode only. On the other hand, the host does not really need to get the agent’s attention: you can set breakpoints in `usrRoot()` to verify that VxWorks can get through this routine. Once `usrRoot()` is working, you can start the agent after the kernel (that is, within `usrRoot()`), after which the polling task is spawned normally.

⚠️ WARNING: If you are using the serial connection, take care that your serial driver does not cause a stray interrupt when the kernel is started, because the serial-driver interrupt handlers are not installed until after `usrRoot()` begins executing; the calling sequence is `usrRoot() ➜ sysClkConnect() ➜ sysHwInit2() ➜ intConnect()`. You may want to modify the driver so that it does not set a channel to interrupt mode until the hardware is initialized. This can be done by setting a flag in the BSP after serial interrupts are connected.
Configuring Boot Parameters

To customize a boot program for your development environment, you must edit `installDir\target\config\bspname\config.h` (the configuration file for your BSP). The file contains the definition of `DEFAULT_BOOT_LINE`, which includes parameters identifying the boot device, IP addresses of host and target, the path and name of the VxWorks image to be loaded, and so on. For information about the boot line parameters defined by `DEFAULT_BOOT_LINE`, see 2.5.4 Description of Boot Parameters, p.45 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.

Figure 4-50 Build Boot ROM
The three main options for boot images are:

bootrom
   A compressed boot image.

bootrom_uncmp
   An uncompressed boot image.

bootrom_res
   A ROM-resident boot image.

**TSFS Boot Configuration**

The simplest way to boot a target that is not on a network is over the TSFS (which does not involve configuring SLIP or PPP). The TSFS can be used to boot a target connected to the host by one or two serial lines, or a NetROM connection.

⚠️ **WARNING:** The TSFS boot facility is not compatible with WDB agent network configurations. See §4.7 Configuring the Target-Host Communication Interface, p.138.

To configure a boot program for TSFS, edit the boot line parameters defined by `DEFAULT_BOOT_LINE` in `config.h` (or change the boot parameters at the boot prompt). The boot device parameter must be `tsfs`, and the file path and name must be relative to the root of the host file system defined for the target server (see Configuring Boot Parameters, p.146 and Target Server File System, p.290).

Regardless of how you specify the boot line parameters, you must reconfigure (as described below) and rebuild the boot image.

If two serial lines connect the host and target (one for the target console and one for WDB communications), `config.h` must include the lines:

```
#undef CONSOLE_TTY
#define CONSOLE_TTY 0
#undef WDB_TTY_CHANNEL
#define WDB_TTY_CHANNEL 1
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_SERIAL
#define INCLUDE_TSFS_BOOT
```

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

```
#undef CONSOLE_TTY
#define CONSOLE_TTY NONE
#undef WDB_TTY_CHANNEL
#define WDB_TTY_CHANNEL 0
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_SERIAL
#define INCLUDE_TSFS_BOOT
```
For a NetROM connection, `config.h` must include the lines:

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

With any of these TSFS configurations, you can also use the target server console to set the boot parameters by defining the `INCLUDE_TSFS_BOOT_VIO_CONSOLE` macro in `config.h`. This disables the auto-boot mechanism, which might otherwise boot the target before the target server could 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.146).

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

### 4.9 Building a Custom Boot ROM

If your boot strategy utilizes a boot ROM, and this boot ROM requires a new driver, you will need to rebuild the boot ROM. Boot ROMs are not yet fully supported as projects in Tornado 2.2. To build a boot ROM, Select Build>Build Boot Rom from the Tornado main menu bar. From the dialog, select the BSP and boot ROM target you wish to build.

If the boot ROM you wish to build is not shown, do the following:

1. Enable extended build options by using the Tools>Options menu from the main menu bar to bring up the Tools Options dialog box. Select the Project pane and select the appropriate check box.

2. Invoke the Build>Customize menu item to bring up the custom build dialog. Click the Add button to bring up a template dialog. Enter menu text (for example, “Build My boot ROM”), the name of the boot ROM image (for example, `bootrom.hex`), and the BSP directory (for example, `installDir\target\config\mv162` for a Windows host).
3. Close the dialog, return to the Build menu, and invoke the newly created menu item. This will build the boot ROM image in the BSP directory.

You can also use the command line as described in *Using the Command Line*, p.127.
5
Command-Line Configuration and Build

5.1 Introduction

The Tornado distribution includes several VxWorks system images for each target shipped. (See 4.4.4 Selecting the VxWorks Image Type, p.124.) 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-relocatable object module with no unresolved external references.

In most cases, you will find the supplied system image entirely adequate for initial development. However, later in the cycle you may want to configure the operating system to reflect your application requirements.

This chapter describes in detail the manual cross-development procedures used to create and run VxWorks systems and applications as well as how to configure the system by directly editing configuration files.

The following topics are included:

- Building, loading, running, and unloading VxWorks applications manually.
- Using VxWorks configuration files and configuration options and parameters.
- Creating common alternative configurations of VxWorks.
- Rebuilding VxWorks system images, bootable applications, and ROM images using manual methods.

There are two approaches to system configuration in Tornado 2.2:

Use the project facility and the GUI

You can use the project facility for configuring and building, with or without a command-line or automated build. If this is your choice, you do not need any of the information in this chapter. See 4. Projects.
Edit configuration files and build from the command line

The remainder of this chapter summarizes the steps and issues involved in this choice.

⚠️ WARNING: Use of the project facility for configuring and building applications is largely independent of the methods used prior to Tornado 2.x (which included manually editing the configuration file `config.h`). The project facility provides the recommended and simpler means for configuration and build, although the manual method as described in this chapter may still be used.

To avoid confusion and errors, the two methods should not be used together for the same project. The one exception is for any configuration macro that is not exposed through the project facility GUI (which may be the case, for example, for some BSP driver parameters). In this case, a configuration file must be edited, and the project facility will implement the change in the subsequent build.

Note that the project facility overrides any changes made to a macro in `config.h` that is also exposed through the project facility. If you are using the project facility, only edit macros in `config.h` which can not be configured through the project facility.

VxWorks has been ported to numerous target systems and can support many different hardware configurations. Some of the cross-development procedures discussed in this chapter depend on the specific system and configuration you are running. The procedures in this chapter are presented in generic form, and may differ slightly on your particular system.

For information specific to an architecture family, see the appropriate VxWorks Architecture Supplement. Information specific to particular target boards is provided with each BSP.

### 5.2 Building, Loading, and Unloading Application Modules

In the Tornado development environment, application modules for the target system are created and maintained on a separate development host. First, the source code, generally in C or C++, is edited and compiled to produce a relocatable object module. Application modules use VxWorks facilities by virtue of including header files that define operating system interfaces and data structures. The
resulting object modules can then be loaded and dynamically linked into a running VxWorks system over the network.

The procedure for configuring a customized VxWorks image is described in 5.3 Configuring VxWorks, p.167. In the interim, you can use the default images shipped with Tornado.

The following sections describe in detail the procedures for carrying out cross-development manually (without using the project facility).

### 5.2.1 Using VxWorks Header Files

Many application modules make use of VxWorks operating system facilities or utility libraries. This usually requires that the source module refer to VxWorks header files. The following sections discuss the use of VxWorks header files.

VxWorks header files supply ANSI C function prototype declarations for all global VxWorks routines. The ANSI C prototypes are conditionally compiled; to use them, the preprocessor constant \_\_STDC\_\_ must be defined. ANSI C compilers define this constant by default. VxWorks provides all header files specified by the ANSI X3.159-1989 standard.

VxWorks system header files are in the directory \installDir\target\h and its subdirectories.

**NOTE:** The notation ${(WIND_BASE)} is used in makefiles to refer to the Tornado installation directory (installDir). If you run the compiler from the command prompt, type %WIND_BASE% instead.

**VxWorks Header File: vxWorks.h**

The header file vxWorks.h contains many basic definitions and types that are used extensively by other VxWorks modules. Many other VxWorks header files require these definitions. Thus, this file must be included first by every application module that uses VxWorks facilities. Include vxWorks.h with the following line:

```
#include "vxWorks.h"
```
Other VxWorks Header Files

Application modules can include other VxWorks header files as needed to access VxWorks facilities. For example, an application module that uses the VxWorks linked-list subroutine library must include the `lstLib.h` file with the following line:

```c
#include "lstLib.h"
```

The API reference entry for each library lists all header files necessary to use that library.

ANSI Header Files

All ANSI-specified header files are included in VxWorks. Those that are compiler-independent or more VxWorks-specific are provided in `installDir\target\h` while a few that are compiler-dependent (for example `stddef.h` and `stdarg.h`) are provided by the compiler installation. Each toolchain knows how to find its own internal headers; no special compile flags are needed.

ANSI C++ Header Files

Each Wind River compiler has its own C++ libraries and C++ headers (such as `iostream` and `new`). The C++ headers are located in the compiler installation directory rather than in `installDir\target\h`. No special flags are required to enable the compilers to find these headers.

NOTE: In previous Tornado releases we recommended the use of the flag `-nostdinc`. This flag should not be used with the current release since it prevents the compilers from finding headers such as `stddef.h`. In this release, host header files will not be pulled in even though `-nostdinc` is not used.

The -I Compiler Flag

By default, the compiler searches for header files first in the directory of the source module and then in its internal subdirectories. In general, `installDir\target\h` should always be searched before the compilers’ other internal subdirectories; to ensure this, always use the following flag for compiling under VxWorks:

```
-I %WIND_BASE%\target\h
```
Some header files are located in subdirectories. To refer to header files in these subdirectories, be sure to specify the subdirectory name in the include statement, so that the files can be located with a single `-I` specifier. For example:

```
#include "vxWorks.h"
#include "sys\stat.h"
```

**VxWorks Nested Header Files**

Some VxWorks facilities make use of other, lower-level VxWorks facilities. For example, the `tty` management facility uses the ring buffer subroutine library. The `tty` header file `tyLib.h` uses definitions that are supplied by the ring buffer header file `rngLib.h`.

It would be inconvenient to require you to be aware of such include-file interdependencies and ordering. Instead, all VxWorks header files explicitly include all prerequisite header files. Thus, `tyLib.h` itself contains an include of `rngLib.h`. (The one exception is the basic VxWorks header file `vxWorks.h`, which all other header files assume is already included.)

Generally, explicit inclusion of prerequisite header files can pose a problem: a header file could get included more than once and generate fatal compilation errors (because the C preprocessor regards duplicate definitions as potential sources of conflict). However, all VxWorks header files contain conditional compilation statements and definitions that ensure that their text is included only once, no matter how many times they are specified by include statements. Thus, an application module can include just those header files it needs directly, without regard to interdependencies or ordering, and no conflicts arise.

**Internal Header Files**

Internal header files are, for the most part, not intended for use by applications. The following subdirectories are exceptions, and are sometimes required by application programs:

- `installDir\target\h\net`, which is used by network drivers for specific network controllers.
- `installDir\target\h\rpc`, which is used by applications using the remote procedure call library.
- `installDir\target\h\sys`, which is used by applications using standard POSIX functions.
VxWorks Private Header Files

Some elements of VxWorks are internal details that may change and so should not be referenced in your application. The only supported uses of a module’s facilities are through the public definitions in the header file, and through the module’s subroutine interfaces. Your adherence ensures that your application code is not affected by internal changes in the implementation of a VxWorks module.

Some header files mark internal details using HIDDEN comments:

```c
/* HIDDEN */
...
/* END HIDDEN */
```

Internal details are also hidden with private header files: files that are stored in the directory \installDir\target\h\private. The naming conventions for these files parallel those in \installDir\target\h with the library name followed by .h. For example, the private header file for semLib is \installDir\target\h\private\semLib.h.

5.2.2 Compiling Application Modules Using GNU Tools

Tornado includes a full-featured C and C++ compiler and associated tools, collectively called the GNU ToolKit. Extensive documentation for this set of tools is available in the GNU ToolKit User’s Guide. This section provides some general orientation about the source of these tools, and describes how the tools are integrated into the Tornado development environment.

NOTE: The GNU tools are not available for the ColdFire architecture; the Diab tools are the default toolset. Diab tools are available as an optional product for the ARM/StrongARM/XScale, MIPS, PowerPC, and SuperH architectures. See 5.2.3 Compiling Application Modules Using Diab Tools, p.160.

The GNU Tools

GNU (“GNU’s Not UNIX!”) is a project of the Free Software Foundation started by Richard Stallman and others to promote free software. To the FSF, free software is software whose source code can be copied, modified, and redistributed without restriction. GNU software is not in the public domain; it is protected by copyright and subject to the terms of the GNU General Public License, a legal document designed to ensure that the software remains free—for example, by prohibiting
proprietary modifications and concomitant restrictions on its use. The General Public License can be found in the file COPYING that accompanies the source code for the GNU tools, and in the section titled Free Software at the back of the GNU ToolKit User’s Guide.

It is important to be aware that the terms under which the GNU tools are distributed do not apply to the software you create with them. In fact, the General Public License makes no requirements of you as a software developer at all, as long as you do not modify or redistribute the tools themselves. On the other hand, it gives you the right to do both of these things, provided you comply with its terms and conditions. It also permits you to make unrestricted copies for your own use.

The Wind River GNU distribution consists of the GNU ToolKit, which contains GNU tools modified and configured for use with your VxWorks target architecture. The source code for these tools is available upon request.

Cross-Development Commands

The GNU cross-development tools in Tornado have names that clearly indicate the target architecture. This allows you to install and use tools for more than one architecture, and to avoid confusion with corresponding host native tools. A suffix identifying the target architecture is appended to each tool name. For example, the cross-compiler for the PowerPC processor family is called ccppc, and the assembler asppc. The suffixes used are shown in Table 5-1. Note that the GNU ToolKit User’s Guide refers to these tools by their generic names (without a suffix).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Command Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM, StrongARM, XScale</td>
<td>arm</td>
</tr>
<tr>
<td>MC680x0</td>
<td>68k</td>
</tr>
<tr>
<td>MIPS</td>
<td>mips</td>
</tr>
<tr>
<td>Pentium</td>
<td>pentium</td>
</tr>
<tr>
<td>PowerPC</td>
<td>ppc</td>
</tr>
<tr>
<td>SuperH</td>
<td>sh</td>
</tr>
<tr>
<td>VxSim Solaris, VxSim PC</td>
<td>simso, simpc</td>
</tr>
</tbody>
</table>
Defining the CPU Type

Tornado supports multiple target architectures. To accommodate this support, several VxWorks header files contain conditional compilation directives based on the definition of the variable CPU. When using these header files, the variable CPU must be defined in one of the following places:

- the source modules
- the header files
- the compilation command line

To define CPU in the source modules or header files, add the following line:

```c
#define CPU cputype
```

To define CPU on the compilation command line, add the following flag:

```
-DCPU=cputype
```

The constants shown in Table 5-2 are supported values for `cputype`.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM, StrongARM,</td>
<td>ARMARCH4, ARMARCH4_T,</td>
</tr>
<tr>
<td>XScale</td>
<td>ARMARCH5, ARMARCH5_T,</td>
</tr>
<tr>
<td></td>
<td>ARM7TDMI, ARM7TDMI_T,</td>
</tr>
<tr>
<td></td>
<td>ARM710A, ARM810, ARMSA110, XSCALE</td>
</tr>
<tr>
<td>MC680x0</td>
<td>MC68000, MC68010, MC68020*, MC68040, MC68060, MC68LC040†, CPU32</td>
</tr>
<tr>
<td>MIPS</td>
<td>MIPS32, MIPS64</td>
</tr>
<tr>
<td>Pentium</td>
<td>PENTIUM2, PENTIUM3, PENTIUM4</td>
</tr>
<tr>
<td>PowerPC</td>
<td>PPC403, PPC405, PPC440, PPC603, PPC604, PPC860</td>
</tr>
<tr>
<td>VxSim Solaris, VxSim PC</td>
<td>SIMSPARC SOLARIS, SIMNT</td>
</tr>
<tr>
<td>SuperH</td>
<td>SH7600, SH7700, SH7750</td>
</tr>
</tbody>
</table>

* MC68020 is the appropriate value for both the MC68020 and the MC6830 CPUs.
† MC68LC040 is the appropriate value for both the MC68LC040 and the MC68EC040.

With makefiles, the CPU definition can be added to the definition of the flags passed to the compiler (usually CFLAGS).
In the source code, the file `vxWorks.h` must be included before any other files with dependencies on the CPU flag.

As well as specifying the CPU value, you must usually run the compiler with one or more option flags to generate object code optimally for the particular architecture variant. These option flags usually begin with `-m`; see *Compiling C Modules With the GNU Compiler*, p.159.

**Compiling C Modules With the GNU Compiler**

The following is an example command to compile an application module for a VxWorks PowerPC 604 system:

```
c:\> cppc -mcpu=604 -mstrict-align -I %WIND_BASE%\target\h \
  -DCPU=PPC603 -DTOOL_FAMILY=gnu -DTOOL=gnu -c applic.c
```

This compiles the module `applic.c` into an object file `applic.o`.

Below we give summary descriptions of the target-independent flags used in the example. Flags that are specific to a particular target architecture are described in the relevant architecture supplement. For more information on any of these flags, see the *GNU ToolKit User’s Guide*.

-`g`
  
  Generate debugging information.

-`c`
  
  Compile only to produce a relocatable object file. The result is an object module with the suffix `.o`, in this case, `applic.o`.

-`-DCPU=CPU`
  
  Required; defines the CPU type.

-`-DTOOL_FAMILY=gnu`
  
  Optional; defines the compilation toolkit used to compile VxWorks. If not entered, it is derived from `-DTOOL=`.

-`-DTOOL=gnu`
  
  Required; specifies the compilation toolkit and the tool environment. For more information, see the *GNU ToolKit User’s Guide*.

-`-I$(WIND_BASE)/target/h`
  
  Include VxWorks header files. (See 5.2.1 Using VxWorks Header Files, p.153.)
-fno-builtin
Use library calls even for common library subroutines such as memcpy. Used by VxWorks for historical reasons. There is no need for application code to use this flag.

-Wall
Turn on all warnings. This flag is optional.

-ansi
Reject non-ANSI-compliant code. This flag is optional.

-O
Perform basic optimizations.

-O2
Perform most supported optimizations (except those involving a space-speed trade-off).

Compiling C++ Modules

The GNU compiler drivers can be used to compile both C and C++ source files. C++ source files are recognized by their extension (typically .cc, .cpp, or .C). For complete information on using C++, including a detailed discussion of compiling C++ modules, see the VxWorks Programmer’s Guide: C++ Development.

CAUTION: Different versions of C++ run-time support are provided for the GNU and Diab toolchains. For this reason, you cannot combine C++ objects compiled with GNU with C++ objects compiled with Diab. All C++ applications must be compiled with the same tool.

5.2.3 Compiling Application Modules Using Diab Tools

For more information about the Diab tools, see the Diab C/C++ Compiler User’s Guide. The Diab tools are the only tools available for ColdFire. Diab tools are available as an optional product for the ARM/StrongARM/XScale, MIPS, PowerPC, and SuperH architectures.
The Diab Tools

The Diab C/C++ compiler suites are high performance programming tools. In addition to the benefits of state-of-the-art optimization, they reduce time spent creating reliable code because the compilers and other tools include many built-in, customizable, checking features which help detect problems earlier.

The compilers are particularly helpful in speeding up or reducing the size of existing programs developed with other tools.

With over 250 command-line options and special pragmas, and a powerful linker command language for arranging code and data in memory, the Diab C/C++ compiler suites can be customized to meet the needs of any embedded systems project. A number of options are specifically designed to be compatible with other tools to ease porting of existing code.

If you are using the Diab tools, you need to be sure that two settings are in place:

- Be sure that `installDir\host\diab\WIN32\bin` is in your path.
- Be sure that the environment variable `DIABLIB` is set to `installDir\host\diab`.

There is a batch file called `torVars.bat` in `installDir\host\x86-win32\bin` that will set `DIABLIB` for you.

Cross-Development Commands

The Diab cross-development tools in Tornado are always called by the same names: `dcc`, `dld`, and so forth. The architecture-specific version of the tool is specified by the `-t` option in the command line or makefile. For Tornado 2.2, the `-t` option always includes the architecture family and the VxWorks specifier, for example:

`-tPPC403FS:vxworks55`

When you install Diab in the Tornado tree, the defaults are set correctly for the architecture you installed. You can use the command `dcc -Xshow-target` to display the value and `dctrl -t` to change it.

You may need to change the architecture family and its characteristics (for example, `PPC403FS`). Detailed information is available in the Diab C/C++ Compiler User’s Guide: Selecting a Target and Its Components.
Defining the CPU Type

Tornado supports multiple target architectures. To accommodate this, several VxWorks header files contain conditional compilation directives based on the definition of the variable CPU. When using these header files, the variable CPU must be defined in one of the following places:

- the source modules
- the header files
- the compilation command line

To define CPU in the source modules or header files, add the following line:

```
#define CPU cputype
```

To define CPU on the compilation command line, add the following flag:

```
-DCPU=cputype
```

The constants shown in Table 5-2 are supported values for cputype.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>ARMARCH4, ARMARCH4_T, ARMARCH5_T, ARMARCH5_T</td>
</tr>
<tr>
<td>ColdFire</td>
<td>MCF5200, MCF5400</td>
</tr>
<tr>
<td>SuperH</td>
<td>SH7600, SH7700, SH7700, SH7750</td>
</tr>
<tr>
<td>MIPS</td>
<td>MIPS32, MIPS64</td>
</tr>
<tr>
<td>PowerPC</td>
<td>PPC403, PPC405, PPC440, PPC603, PPC604, PPC860</td>
</tr>
<tr>
<td>StrongARM, XScale</td>
<td>STRONGARM, XSCALE</td>
</tr>
</tbody>
</table>

With makefiles, the CPU definition can be added to the definition of the flags passed to the compiler (usually CFLAGS).

In the source code, the file vxWorks.h must be included before any other files with dependencies on the CPU flag.

As well as specifying the CPU value, you must usually run the compiler with one or more option flags to generate object code optimally for the particular architecture variant. For detailed information, see Diab C/C++ Compiler User’s Guide: Selecting a Target and Its Components.
Compiling C Modules With the Diab Compiler

The following is an example command to compile an application module for a VxWorks PowerPC 604 system:

```
c:\> dcc -g -tPPC403FS:vxworks55 -Xmismatch-warning=2 -ew1554,1551,1552, 1086,1047,1547 -Xclib-optim-off -Xansi -Xstrings-in-text=0 -Wa,-Xsemi-is-newline-ei1516,1643,1604 -Xlocal-data-area-static-only -W:c++:,-Xexceptions -Xsize-opt -Wall -TM:\wind_base\target\h -DCPU=PPC604 -DTOOL=diab -c applic.c
```

This compiles the module `applic.c` into an object file `applic.o`.

Below we give summary descriptions of the target independent flags used in the example. Flags that are specific to a particular target architecture are described in the relevant architecture supplement. For more information on any of these flags see the *Diab C/C++ Compiler User's Guide*.

- `-g` Generate debugging information.
- `-tppc403fs:vxworks55` Specifies the processor family and the compilation environment. See Cross-Development Commands, p.161.
- `-Xname_or_number[=value]` Control the compilation process when behavior other than the default is needed. Most `-X` options can be set either by name `-Xname` or by number `-Xn`. Options control such behaviors as debugging, optimization, and syntax.
- `-ewn[,]n,...` For each message number in the comma-separated list, change the severity level of the message to `warning`.
- `-wa,argument` Pass `argument` to the assembler.
- `-ein[,]n,...` For each message number in the comma-separated list, change the severity level of the message to information (equivalent to `ignore`).
- `-W:c++: -Xexception` Pass the argument `-Xexception` to the C++ compiler.
- `-D arch family` Specifies the architecture family.
-DCPU=CPU
   Required; defines the CPU type.

-DTOOL_FAMILY=diab
   Optional; defines the compilation toolkit used to compile VxWorks. If not entered, it is derived from -DTOOL=.

-DTOOL=diab
   Required; specifies the tool and tool environment. For more information, see the Diab C/C++ Compiler User’s Guide.

-I$(WIND_BASE)/target/h
   Include VxWorks header files. (See 5.2.1 Using VxWorks Header Files, p.153)

### Compiling C++ Modules

The Diab compiler uses **dcc** to invoke the C compiler and **dplus** to invoke the C++ compiler. For complete information on using C++, including a detailed discussion of compiling C++ modules see the VxWorks Programmer’s Guide: C++ Development and the Diab C/C++ Compiler User’s Guide.

⚠️ **CAUTION:** Different versions of C++ run-time support are provided for the GNU and Diab toolchains. For this reason, you cannot combine C++ objects compiled with GNU with C++ objects compiled with Diab. All C++ applications must be compiled with the same tool.

### 5.2.4 Static Linking (Optional)

After you compile an application module, you can load it directly into the target with the Tornado dynamic loader (through the shell or through the debugger).

In general, application modules do not need to be prelinked before being downloaded to the target. The exception is when several application modules cross reference each other. In this case, the modules should be linked to form a single downloadable module. When using C++, this prelinking should be done before the **munch** step (see the VxWorks Programmer’s Guide: C++ Development).

The following example is a command to link several application modules, using the GNU linker for the PowerPC family of processors.

```
c:\> ldppc -o applic.o -r applic1.o applic2.o applic3.o
```
Similarly, the following example is a command to link several application modules, using the Diab linker for the PowerPC family of processors.

```
c:\> dld -o applic.o -r applic1.o applic2.o applic3.o
```

This creates the object module `applic.o` from the object modules `applic1.o`, `applic2.o`, and `applic3.o`. The `-r` option is required, because the object-module output must be left in relocatable form so that it can be downloaded and linked to the target VxWorks image.

Any VxWorks facilities called by the application modules are reported by the linker as unresolved externals. These are resolved by the Tornado loader when the module is loaded into VxWorks memory.

**WARNING:** Do not link each application module with the VxWorks libraries. Doing this defeats the load-time linking feature of Tornado, and wastes space by writing multiple copies of VxWorks system modules on the target.

### 5.2.5 Downloading an Application Module

After application object modules are compiled (and possibly linked by the host ldarch command), they can be dynamically loaded into a running VxWorks system by invoking the Tornado module loader. You can do this either from the Tornado shell using the built-in command `ld()`, or from the debugger using the Debug menu or the `load` command.

The following is a typical load command from the Tornado shell:

```
$> ld <applic.o
```

This relocates the code from the host file `applic.o`, linking to previously loaded modules, and loads the object module into the target’s memory. Once an application module is loaded into target memory, any subroutine in the module can be invoked directly from the shell, spawned as a task, connected to an interrupt, and so on.

**NOTE:** The order in which modules are loaded using `ld()` is important. A downloaded module can call into a previously downloaded module to resolve symbols. However, the opposite is not true. For example, given two modules, `applic1.o` and `applic2.o`, in which `applic1.o` can stand alone, but `applic2.o` relies on symbols that are defined in `applic1.o`, `ld()` will perform the necessary linking only if `applic1.o` is loaded before `applic2.o`. 
The shell `ld()` command, by default, adds only global symbols to the symbol table. During debugging, you may want local symbols as well. To get all symbols loaded (including local symbols), you can use the GDB command `load` from the debugger. Because this command is meant for debugging, it always loads all symbols. Alternatively, you can load all symbols by calling the shell command `ld()` with a full argument list instead of the shell-redirection syntax shown above. When you use an argument list, you can get all symbols loaded by specifying a 1 as the first argument, as in the following example:

```
-> ld 1,0,"applic.o"
```

In the previous examples, the object module `applic.o` resides in the shell’s current working directory. Normally, you can use either relative path names or absolute path names to identify object modules to `ld()`. If you use a relative path name, the shell converts it to an absolute path (using its current working directory) before passing the download request to the target server. In order to avoid trouble when you call `ld()` from a shell that is not running on the same host as its target server, Tornado supplies the `LD_SEND_MODULES` facility; see 7. Shell. If you are using a remote target server and `ld()` fails with a “no such file” message, be sure that `LD_SEND_MODULES` is set to “on.”

⚠️ **CAUTION:** If you call `ld()` with an explicit argument list, any backslash characters in the module-name argument must be doubled. If you supply the module name with the redirection symbol, as in the earlier example in this section, no double backslashes are needed. See 7. Shell for more discussion of this issue.

For more information about loader arguments, see the discussion of `ld()` in the reference entry for `windsh`.

For information about the target-resident version of the loader (which also requires the target-resident symbol table), see the `VxWorks Programmer’s Guide: Target Tools` and the `VxWorks` reference entry for `loadLib`. For information on booting `VxWorks`, see 2.5 Booting `VxWorks`, p.41.

### 5.2.6 Module IDs and Group Numbers

When a module is loaded, it is assigned a module ID and a group number. Both the module ID and the group number are used to reference the module. The module ID is returned by `ld()` as well as by the target-resident loader routines. When symbols are added to the symbol table, the associated module is identified by the group number (a small integer). (Due to limitations on the size of the symbol table, the module ID is inappropriate for this purpose.) All symbols with the same group
number are from the same module. When a module is unloaded, the group number is used to identify and remove all the module’s symbols from the symbol table.

5.2.7 Unloading Modules

Whenever you load a particular object module more than once, using the target server (from either the shell or the debugger), the older version is unloaded automatically. You can also unload a module explicitly; both the Tornado shell and the target-resident VxWorks libraries include an unloader. To remove a module from the shell, use the shell routine `unld();` see the reference entry for `windsh`.

For information about the target-resident version of the unloader (which also requires the target-resident symbol table and loader), see the VxWorks Programmer's Guide: Target Tools and the VxWorks reference entry for `unldLib`.

After a module has been unloaded, any calls to routines in that module fail with unpredictable results. Take care to avoid unloading any modules that are required by other modules. One solution is to link interdependent files using the static linker `ldarch` as described in 5.2.4 Static Linking (Optional), p.164, so that they can only be loaded and unloaded as a unit.

5.3 Configuring VxWorks

The configuration of VxWorks is determined by the configuration header files `installDir\target\config\all\configAll.h` and `installDir\target\config\bspname\config.h`. These files are used by the `usrConfig.c`, `bootConfig.c`, and `bootInit.c` modules as they run the initialization routines distributed in the directory `installDir\target\src\config` to configure VxWorks.

The VxWorks distribution includes the configuration files for the default development configuration. You can create your own versions of these files to better suit your particular configurations; this process is described in the following subsections. In addition, if you need multiple configurations, environment variables can be set so you can move easily between them.
Including optional components in your VxWorks image can significantly increase the image size. VxWorks BSP Developer's Guide you receive a warning from \texttt{vxsize} when building VxWorks, or if the size of your image becomes greater than that supported by the current setting of \texttt{RAM\_HIGH\_ADRS}, be sure to see Scaling Down VxWorks, p. 174 and 5.6 Creating Bootable Applications, p. 191 for information on how to resolve the problem.

\begin{itemize}
\item \textbf{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 \texttt{config.h} or \texttt{configAll.h}). The project facility provides the recommended and simpler means for configuration and building; the manual method is described in this section.
\end{itemize}

To avoid confusion and errors, the two methods should not be used together for the same project. One exception is for any configuration macro that is not accessible through the project facility GUI (which may be the case, for example, for some BSP driver parameters). You can use a \texttt{Find Object} dialog box to determine if a macro is accessible or not (see Finding VxWorks Components and Configuration Macros, p. 117). 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):

\begin{verbatim}
  project facility
  config.h
  configAll.h
\end{verbatim}

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

Another exception is that you may configure a BSP using manual methods and then use provided make targets to create a project for application development. See 5.7 Building Projects from a BSP, p. 195.

\subsection*{5.3.1 The Board Support Package (BSP)}

The directory \texttt{installDir\target\config\bspname} contains the Board Support Package (BSP), which consists of files for the particular hardware used to run VxWorks, such as a VME board with serial lines, timers, and other devices. The files include: \texttt{Makefile}, \texttt{sysLib.c}, \texttt{sysALib.s}, \texttt{romInit.s}, \texttt{bspname.h}, and \texttt{config.h}. 
Wind River-supplied BSPs conform to a standard, introduced with BSP Version 1.1. The standard is fully described in the *VxWorks BSP Developer’s Guide*.

**The System Library**

The file `sysLib.c` provides the board-level interface on which VxWorks and application code can be built in a hardware-independent manner. The functions addressed in this file include:

- **Initialization functions**
  - initialize the hardware to a known state
  - identify the system
  - initialize drivers, such as SCSI or custom drivers

- **Memory/address space functions**
  - get the on-board memory size
  - make on-board memory accessible to the external bus (optional)
  - map local and bus address spaces
  - enable/disable cache memory
  - set/get nonvolatile RAM (NVRAM)
  - define the board’s memory map (optional)
  - virtual-to-physical memory map declarations for processors with MMUs

- **Bus interrupt functions**
  - enable/disable bus interrupt levels
  - generate bus interrupts

- **Clock/timer functions**
  - enable/disable timer interrupts
  - set the periodic rate of the timer

- **Mailbox/location monitor functions (optional)**
  - enable mailbox/location monitor interrupts

The `sysLib` library does not support every feature of every board. Some boards may have additional features, others may have fewer, others still may have the same features with a different interface. For example, some boards provide some `sysLib` functions by means of hardware switches, jumpers, or PALs, instead of by software-controllable registers.

The configuration modules `usrConfig.c` and `bootConfig.c` in `installDir\target\config\all` are responsible for invoking this library’s routines at
the appropriate time. Device drivers can use some of the memory mapping routines and bus functions.

**Virtual Memory Mapping**

For boards with MMU support, the data structure `sysPhysMemDesc` defines the virtual-to-physical memory map. This table is typically defined in `sysLib.c`, although some BSPs place it in a separate file, `memDesc.c`. It is declared as an array of the data structure `PHYS_MEM_DESC`. No two entries in this descriptor can overlap; each entry must be a unique memory space.

The `sysPhysMemDesc` array should reflect your system configuration, and you may encounter a number of reasons for changing the MMU memory map, for example: the need to change the size of local memory or the size of the VME master access space, or because the address of the VME master access space has been moved. For information on virtual memory mapping, as well as an example of how to modify `sysPhysMemDesc`, see VxWorks Programmer’s Guide: Virtual Memory Interface.

⚠️ **CAUTION:** A bus error can occur if you try to access memory that is not mapped.

**Configuration Files**

The file `config.h` specifies which VxWorks facilities are included in your system image. The file `bspname.h` specifies BSP-specific capabilities.

**BSP Initialization Modules**

The following files initialize the BSP:

- The file `romInit.s` contains assembly-level initialization routines.
- The file `sysALib.s` contains initialization and system-specific assembly-level routines.

**BSP Documentation**

The file `target.nr` in the `installDir\target\config\bspname` directory is the source of the online reference entry for target-specific information. (You can also view the
5.3.2 The Environment Variables

You can use Tornado environment variables to build variations of system configurations. In general, your Tornado environment consists of three parts: the host code (Tornado), the target code, and the configuration files discussed in this section. On Windows hosts, the IDE automatically locates Tornado code in the following locations:

- **Host code**: 
  - `installDir\host\hosttype\bin` or 
  - `%WIND_BASE%\host\x86-win32\bin`
- **Target code**: 
  - `installDir\target` or 
  - `TGT_DIR = %WIND_BASE%\target`
- **Configuration code**: 
  - `installDir\target\config\all` or 
  - `CONFIG_ALL = %TGT_DIR%\config\all`

To use different versions of `usrConfig.c`, `bootConfig.c`, and `bootInit.c`, store them in a different directory and change the value of `CONFIG_ALL`. To use different target code, point to the alternate directory by changing the value of `TGT_DIR`.

You can change the value of `CONFIG_ALL` by changing it either in your makefile or on the command line. The value of `TGT_DIR` must be changed on the command line.

**NOTE:** Changing `TGT_DIR` will change the default value of `CONFIG_ALL`. If this is not what you want, reset `CONFIG_ALL` as well.

To change `CONFIG_ALL` in your makefile, add the following command:

```
CONFIG_ALL = %WIND_BASE%\target\config\newDir
```

To change `CONFIG_ALL` on the command line, do the following:

```
c:\> make ... CONFIG_ALL = %WIND_BASE%\target\config\newDir
```

To change `TGT_DIR` on the command line, do the following:

```
c:\> make ... TGT_DIR = %ALT_DIR%\target
```
5.3.3 The Configuration Header Files

You can control VxWorks’s configuration by including or excluding definitions in the global configuration header file `configAll.h` and in the target-specific configuration header file `config.h`. This section describes these files.

The Global Configuration Header File: `configAll.h`

The `configAll.h` header file, in the directory `installDir\target\config\all`, contains default definitions that apply to all targets, unless they are redefined in the target-specific header file `config.h`. The following options and parameters are defined in `configAll.h`:

- kernel configuration parameters
- I/O system parameters
- NFS parameters
- selection of optional software modules
- selection of optional device controllers
- cache modes
- maximum number of different shared memory objects
- device controller I/O addresses, interrupt vectors, and interrupt levels
- miscellaneous addresses and constants

The BSP-specific Configuration Header File: `config.h`

The BSP-specific header file, `config.h`, is located in the directory `installDir\target\config\bspname`. This file contains definitions that apply only to the specific target, and can also redefine default definitions in `configAll.h` that are inappropriate for the particular target. For example, if a target cannot access a device controller at the default I/O address defined in `configAll.h` because of addressing limitations, the address can be redefined in `config.h`.

The `config.h` header file includes definitions for the following parameters:

- default boot parameter string for boot ROMs
- interrupt vectors for system clock and parity errors
- device controller I/O addresses, interrupt vectors, and interrupt levels
- shared memory network parameters
- miscellaneous memory addresses and constants
Selection of Optional Features

VxWorks ships with optional features and device drivers that can be included in, or omitted from, the target system. These are controlled by macros in the project facility or the configuration header files that cause conditional compilation in the installDir\target\config\all\usrConfig.c module.

The distributed versions of the configuration header files configAll.h and config.h include all the available software options and several network device drivers. If you are not using the project facility (see 4. Projects), you define a macro by moving it from the EXCLUDED FACILITIES section of the header file to the INCLUDED SOFTWARE FACILITIES section. For example, to include the ANSI C assert library, make sure the macro INCLUDE_ANSI_ASSERT is defined; to include the Network File System (NFS) facility, make sure INCLUDE_NFS is defined. Modification or exclusion of particular facilities is discussed in detail in 5.3.5 Alternative VxWorks Configurations, p.174.

5.3.4 The Configuration Module: usrConfig.c

Use of the VxWorks configuration header files to configure your VxWorks system should meet all of your development requirements. Users should not resort to changing the Wind River-supplied usrConfig.c, or any other module in the directory installDir\target\config\all. If, however, an extreme situation requires such a change, we recommend you copy all the files in installDir\target\config\all to another directory, and add a CONFIG_ALL macro to your makefile to point the make system to the location of the modified files. For example, add the following to your makefile after the first group of include statements:

```
# ../myAll contains a copy of all the ../all files
CONFIG_ALL = ../myAll
```

1. To see the available macros with their descriptions, see installDir\target\config\all\configAll.h (for macros applicable to all BSPs) and installDir\target\config\bspname\config.h (for macros applicable to a specific BSP).
5.3.5 Alternative VxWorks Configurations

The discussion of the `usrConfig` module in 5.3.4 The Configuration Module: `usrConfig.c`, p.173 outlined the default configuration for a development environment. In this configuration, the VxWorks system image contains all of the VxWorks modules that are necessary to allow you to interact with the system through the Tornado host tools.

However, as you approach a final production version of your application, you may want to change the VxWorks configuration in one or more of the following ways:

- Change the configuration of the target agent.
- Decrease the size of VxWorks.
- Run VxWorks from ROM.

The following sections discuss the latter two alternatives to the typical development configuration. For a discussion on reconfiguring the target agent, see 4. Projects.

Scaling Down VxWorks

In a production configuration, it is often desirable to remove some of the VxWorks facilities to reduce the memory requirements of the system, to reduce boot time, or for security purposes.

Optional VxWorks facilities can be omitted by commenting out or using `#undef` to undefine their corresponding control constants in the header files `configAll.h` or `config.h`. For example, logging facilities can be omitted by undefining `INCLUDE_LOGGING`, and signalling facilities can be omitted by undefining `INCLUDE_SIGNALS`.

VxWorks is designed to make it easy to exclude facilities you do not need. However, not every BSP is organized in this way. If you wish to minimize the size of your system, be sure to examine your BSP code and eliminate references to facilities you do not need. Even though you exclude them, if your code refers to them, your exclusion will be overridden.

Excluding Kernel Facilities

The definition of the following constants in `configAll.h` is optional, because referencing any of the corresponding kernel facilities from the application automatically includes the kernel service:
– INCLUDE_SEM_BINARY
– INCLUDE_SEM_MUTEX
– INCLUDE_SEM_COUNTING
– INCLUDE_MSG_Q
– INCLUDE_WATCHDOGS

These configuration constants appear in the default VxWorks configuration to ensure that all kernel facilities are configured into the system, even if not referenced by the application. However, if your goal is to achieve the smallest possible system, exclude these constants; this ensures that the kernel does not include facilities you are not actually using.

There are two other configuration constants that control optional kernel facilities: INCLUDE_TASK_HOOKS and INCLUDE_CONSTANT_RDY_Q. Define these constants in configAll.h if the application requires either kernel callouts (use of task hook routines) or a constant-insertion-time, priority-based ready queue. A ready queue with constant insert time allows the kernel to operate context switches with a fixed overhead regardless of the number of tasks in the system. Otherwise, the worst-case performance degrades linearly with the number of ready tasks in the system. Note that the constant-insert-time ready queue uses 2 KB for the data structure; some systems do not have sufficient memory for this. In those cases, the definition of INCLUDE_CONSTANT_RDY_Q may be omitted, thus enabling use of a smaller (but less deterministic) ready queue mechanism.

Excluding Network Facilities

In some applications it may be appropriate to eliminate the VxWorks network facilities. For example, in the ROM-based systems or standalone configurations described in 5.6 Creating Bootable Applications, p.191, there may be no need for network facilities.

To exclude the network facilities, be sure the following constants are not defined:
– INCLUDE_NETWORK
– INCLUDE_NET_INIT
– INCLUDE_NET_SYM_TBL
– INCLUDE_NFS
– INCLUDE_RPC

Option Dependencies

Option dependencies are coded in the file instaldir/target/src/config/usrDepend.c, so that when a particular option is chosen, everything required is included. This assures you of a working system with minimum effort. Although you can exclude the features that you do not need
by undefining them in config.h and configAll.h, you should be aware that in some cases they may not be excluded because of dependencies.

For example, you cannot use telnet without running the network. Therefore, if in your configAll.h file, the option INCLUDE_TELNET is selected but the option INCLUDE_NET_INIT is not, usrDepend.c defines INCLUDE_NET_INIT for you. Because the network initialization requires the network software, the userDepend.c file also defines INCLUDE_NETWORK.

Because most of the dependencies are taken care of in usrDepend.c, that file is currently included in usrConfig.c. This simplifies the build process and the selection of options. However, you can change or add dependencies if you choose.

**Executing VxWorks from ROM**

You can put VxWorks or a VxWorks-based application into ROM; this is discussed in 5.6.3 Creating a VxWorks System in ROM, p. 194. For an example of a ROM-based VxWorks application, see the VxWorks boot ROM program. The file installDir\target\config\all\bootConfig.c is the configuration module for the boot ROM, replacing the file usrConfig.c provided for the default VxWorks development system.

In such ROM configurations, the text and data segments of the boot or VxWorks image are first copied into the system RAM, then the boot procedure or VxWorks executes in RAM. On some systems where memory is a scarce resource, it is possible to save space by copying only the data segment to RAM. The text segment remains in ROM and executes from that address space, and thus is termed ROM resident. The memory that was to be occupied by the text segment in RAM is now available for an application (up to 300 KB for a standalone VxWorks system). Note that ROM-resident VxWorks is not supported on all boards; see the reference entry for your target if you are not sure that your board supports this configuration.

The drawback of a ROM-resident text segment is the limited data widths and lower memory access time of the EPROM, which causes ROM-resident text to execute more slowly than if it was in RAM. This can sometimes be alleviated by using faster EPROM devices or by reconfiguring the standalone system to exclude unnecessary system features.

Aside from program text not being copied to RAM, the ROM-resident versions of the VxWorks boot ROMs and the standalone VxWorks system are identical to the conventional versions. A ROM-resident image is built with an uncompressed
version of either the boot ROM or standalone VxWorks system image. VxWorks target makefiles include entries for building these images; see Table 5-4.

Table 5-4 Makefile ROM-Resident Images

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Image File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS and PowerPC</td>
<td>bootrom_res_high</td>
<td>ROM-resident boot ROM image. The data segment is copied from ROM to RAM at address RAM_HIGH_ADRS.</td>
</tr>
<tr>
<td></td>
<td>vxWorks.res_rom_res_low</td>
<td>ROM-resident standalone system image without compression. The data segment is copied from ROM to RAM at address RAM_LOW_ADRS.</td>
</tr>
<tr>
<td></td>
<td>vxWorks.res_rom_nosym_res_low</td>
<td>ROM-resident standalone system image without compression or symbol table. Data segment is copied from ROM to RAM at address RAM_LOW_ADRS.</td>
</tr>
<tr>
<td>All Other Targets</td>
<td>bootrom_res</td>
<td>ROM-resident boot ROM image.</td>
</tr>
<tr>
<td></td>
<td>vxWorks.res_rom</td>
<td>ROM-resident standalone system image without compression.</td>
</tr>
<tr>
<td></td>
<td>vxWorks.res_rom_nosym</td>
<td>ROM-resident system image without compression or symbol table. Ideal for the Tornado environment.</td>
</tr>
</tbody>
</table>

* All images have a corresponding file in Motorola S-record or Intel Hex format with the same file name plus the extension .hex and one in binary format with the extension .bin.

Because of the size of the system image, 512 KB of EPROM is recommended for the ROM-resident version of the standalone VxWorks system. More space is probably required if applications are linked with the standalone VxWorks system. For a ROM-resident version of the boot ROM, 256 KB of EPROM is recommended. If you use ROMs of a size other than the default, modify the value of ROM_SIZE in the target makefile and config.h.

A separate make target, vxWorks.res_rom_nosym, has been created to provide a ROM-resident image without the symbol table. This is intended to be a standard ROM image for use with the Tornado environment where the symbol table resides.
on the host system. Being ROM-resident, the debug agent and VxWorks are ready almost immediately after power-up or restart.

The data segment of a ROM-resident standalone VxWorks system is loaded at \texttt{RAM\_LOW\_ADRS} (defined in the makefile) to minimize fragmentation. The data segment of ROM-resident boot ROMs is loaded at \texttt{RAM\_HIGH\_ADRS}, so that loading VxWorks does not overwrite the resident boot ROMs. For a CPU board with limited memory (under 1 MB of RAM), make sure that \texttt{RAM\_HIGH\_ADRS} is less than \texttt{LOCAL\_MEM\_SIZE} by a margin sufficient to accommodate the data segment. Note that \texttt{RAM\_HIGH\_ADRS} is defined in both the makefile and \texttt{config.h}. These definitions must agree.

Figure 5-1 shows the memory layout for ROM-resident boot and VxWorks images. The lower portion of the diagram shows the layout for ROM; the upper portion shows the layout for RAM. \texttt{LOCAL\_MEM\_LOCAL\_ADRS} is the starting address of RAM. For the boot image, the data segment gets copied into RAM above \texttt{RAM\_HIGH\_ADRS} (after space for \texttt{bss} is reserved). For the VxWorks image, the data segment gets copied into RAM above \texttt{RAM\_LOW\_ADRS} (after space for \texttt{bss} is reserved). Note that for both images the text segment remains in ROM.

5.4 Building a VxWorks System Image

You can redefine the VxWorks configuration in two ways: interactively, as described 4. Projects, or by editing VxWorks configuration files as described in 5.3 Configuring VxWorks, p.167. In either case, after you alter the configuration, VxWorks must be rebuilt to incorporate the changes. This includes recompiling certain modules and re-linking the system image. This section explains the procedures for rebuilding the VxWorks system image using manual techniques.

5.4.1 Available VxWorks Images

There are three types of VxWorks images:

- Boot application images
- Downloadable VxWorks images
- ROM-based VxWorks images
Figure 5-1  **ROM-Resident Memory Layout**

**BOOT IMAGE**

ROM

- ROM_TEXT_ADRS
  - data
  - bss

- RAM_HIGH_ADRS
  - data
  - bss

- LOCAL_MEM_LOCAL_ADRS

**VXWORKS IMAGE**

ROM

- ROM_TEXT_ADRS
  - data

- RAM
  - bss

- LOCAL_MEM_LOCAL_ADRS
  - RAM_LOW_ADRS

= copied to RAM
Boot ROM images come in three flavors: compressed, uncompressed, and ROM-resident.

- `bootrom`: normal compressed boot ROM
- `bootrom_uncmp`: uncompressed boot ROM
- `bootrom_res`: ROM-resident boot ROM

Downloaded VxWorks images come in two basic varieties, Tornado and standalone. (Here “Tornado” is a Vxworks image that uses the host-based tools and symbol table, while “standalone” is an image that uses the target tools and symbol table.)

- `vxWorks`: basic Tornado uses host shell and symbol table
- `vxWorks_st`: standalone image has target shell and symbol table

ROMmed VxWorks images include:

- `vxWorks_rom`: Tornado in ROM (uncompressed)
- `vxWorks_st_rom`: `vxWorks_st` in ROM (compressed)
- `vxWorks_res_rom`: `vxWorks_st` ROM-resident
- `vxWorks_res_rom_nosym`: Tornado, ROM-resident

### 5.4.2 Rebuilding VxWorks with make

VxWorks uses the GNU `make` facility to recompile and relink modules. A file called `Makefile` in each VxWorks target directory contains the directives for rebuilding VxWorks for that target. See the GNU Make User's Guide for a detailed description of GNU `make` and of how to write makefiles.

If you choose to use manual techniques on Windows hosts, you must use the command line for building individual application modules. You can use either the command line or the project facility in Tornado 1.0.1 compatibility mode to rebuild BSPs. For information on how to implement Tornado 1.0.1 compatibility mode, see 12.4.3 Customizing the Build Menu, p. 421.

To rebuild VxWorks from the Windows command prompt (or from a batch file), change to the `config` directory for the desired target and invoke `make` as follows:

```
C:\> cd installDir\target\config\bspname
C:\installDir\target\config\bspname> make
```

`make` compiles and links modules as necessary, based on the directives in the target directory’s makefile.
To rebuild VxWorks when only header files change, use one of the following methods:

```
c:\> make clean VxWorks
```

Or:

```
c:\> make clean
c:\> make
```

Either method removes all existing .o files, and then recreates the new .o files required by VxWorks.

### 5.4.3 Including Customized VxWorks Code

The directory `installDir\target\src\usr` contains the source code for certain portions of VxWorks that you may wish to customize. For example, `usrLib.c` is a popular place to add target-resident routines that provide application-specific development aids. For a summary of other files in this directory, see A. Directories and Files.

If you modify one of these files, an extra step is necessary before rebuilding your VxWorks image: you must replace the modified object code in the appropriate VxWorks archive. The makefile in `installDir\target\src\usr` automates the details; however, because this directory is not specific to a single architecture, you must specify the value of the CPU variable on the `make` command line:

```
c:\> make CPU=putype TOOL=tool
```

If you do this frequently on a Windows host, you can record the CPU definition in the Build Target field of a custom command in the Project menu; see 12.4.3 Customizing the Build Menu, p.421.

This step recompiles all modified files in the directory, and replaces the corresponding object code in the appropriate architecture-dependent directory. After that, the next time you rebuild VxWorks, the resulting system image includes your modified code.
The following example illustrates replacing `usrLib` with a modified version, rebuilding the archives, and then rebuilding the VxWorks system image. For the sake of conciseness, the `make` output is not shown. The example assumes the `epc4` (I80386) BSP; replace the BSP directory name and CPU value as appropriate for your environment.

```
c:\> cd %WIND_BASE%\target\src\usr

<installDir>target\src\usr> copy usrLib.c usrLib.c.orig

<installDir>target\src\usr> copy <develDir>\usrLib.c usrLib.c

<installDir>target\src\usr> make CPU=PPC660

<installDir>target\src\usr> cd %WIND_BASE%\target\config\epc4

<installDir>target\config\epc4> make
```

5.4.4 Linking the System Modules

The commands used to link a VxWorks system image are somewhat complicated. Fortunately, it is not necessary to understand those commands in detail because they are included in the `makefile` in each VxWorks target directory. However, for completeness, this section gives an explanation of the flags and parameters used to link VxWorks modules.

VxWorks operating system modules are distributed in the form of archive libraries. One set of archives is provided for each target architecture. These archives are located under `installDir/target/lib`. For more details about the archive directory structure, see the Tornado Migration Guide: Binary Compatibility.

These modules are combined with the configuration module `usrConfig.o` by the `ccarch` command on the host. (The file `usrConfig.c` is described in 5.3.4 The Configuration Module: `usrConfig.c`, p.173.) The following are example commands for building and linking a VxWorks system using the GNU compiler for PowerPC.

For a partial image (partially linked):

```
ccppc -r -nostdlib -Wl,-X -o vxWorks.tmp sysALib.o sysLib.o \
  miiLib.obj usrConfig.o version.o -Wl,--start-group \
  -L\vobs\wpwr\target\lib\ppc\PPC604\gnu \n  -L\vobs\wpwr\target\lib\ppc\PPC604\common \-lplus \-lgmcplus \
  -lvxcom \lvxdscom \-larch \-lcommoncc \-ldcc \-ldrv \-lgcc \-larch \-ldrv \-lgcc \-linet \-los \
  -lrpc \-lsecure \-ltfss \-lusb \-lvxfusion \-lvxmp \-lvxvmi \-lwdb \
  -lvx\-lwind \lwindview \\vobs\wpwr\target\lib\libPPC604\gnu\vux.a -Wl,--end-group
```

For the final image (fully linked):

```
ldppc -X -N -e _sysInit -Ttext 00100000 -o vxWorks dataSegPad.o \
vxWorks.tmp ctdt.o -T \vobs\wpwr\target\h\tool\gnu\ldscripts\link.RAM
```
The meanings of the flags in these commands are as follows:

- **-r** Generate relocatable output.

- **-nostdlib**
  Do not use the standard system libraries.

- **-wloption**
  Pass option as an option to the linker.

- **-X** Eliminate some compiler-generated symbols from the symbol table.

- **-o vxWorks**
  Name the output object module vxWorks.

---start-group archives --end-group
  The specified archives are searched repeatedly until no new undefined references are created. archives should be a list of archive files. They may be either explicit file names, or -l options.

- **-larch**
  List of all the archive files added to the list of files to link. ld searches its path-list for occurrences of libarch.a for every archive specified.

- **-Lsearchdir**
  List of all the paths that ld will search for archive libraries. The directories are searched in the order in which they are specified on the command line. All -L options apply to all -l options, regardless of the order in which the options appear.

- **-N** Do not configure the output object module for a virtual-memory system.

- **-Ttext 1000**
  Specify the relocation address as a hexadecimal constant; in this example, 1000 hexadecimal. This is the address where the system must be loaded in the target, and is also the address where execution starts. Some target systems have limitations on where this relocation address can be.

- **-e _sysInit**
  Define the entry point to vxWorks. _sysInit() is the first routine in sysALib.o, which is the first module loaded by ldarch.

**sysALib.o and sysLib.o**
Modules that contain CPU-dependent initialization and support routines. The module sysALib.o must be the first module specified in the ldarch command.
usrConfig.o
The configuration module (described in detail in 5.3.4 The Configuration Module: usrConfig.c, p.173). If you have several different system configurations, you may maintain several different configuration modules, either in installDir\target or in your own directory.

version.o
A module that defines the creation date and version number of this vxWorks object module. It is created by compiling the output of makeVersion, an auxiliary tool in the installDir\host\host-os\bin directory.

installDir\target\lib\libcpugnuvx.a
A VxWorks 5.4.x archive, included for backward compatibility. VxWorks is completely specified by the libraries indicated by -l and -L. This library might be used by optional or third-party products.

Additional object modules
You can link additional object modules (with .o suffix) into the run-time VxWorks system by naming them on the ldarch command line. An easy way to do this is to use the variable MACH_EXTRA in the BSP makefiles. Define this variable and list the object modules to be linked with VxWorks. Note that during development, application object modules are generally not linked with the system (unless they are needed by the usrConfig module), because it is more convenient to load them incrementally from the host after booting VxWorks. See 5.6 Creating Bootable Applications, p.191 for more detail on linking application modules in a bootable system.

5.4.5 Creating the System Symbol Table Module

The Tornado target server uses the VxWorks symbol table on the host system, both for dynamic linking and for symbolic debugging. The symbol table file is created by the supplied tool xsym. Processing an object module with xsym creates a new object module that contains all the symbols of the original file, but with no code or data. The line in the makefile that creates this file executes the command:

```
xsym < vxWorks > vxWorks.sym
```

The file vxWorks.sym is downloaded to the target to build the target symbol table when INCLUDE_NET_SYM_TBL is included.
5.5 Makefiles for BSPs and Applications

Makefiles for VxWorks applications are easy to create by exploiting the makefiles and `make` include files shipped with VxWorks BSPs. This section discusses how the VxWorks BSP makefiles are structured. For more information, see B. Makefile Details. An example of how to utilize this structure for application makefiles is in 5.5.2 Using Makefile Include Files for Application Modules, p. 190.

A set of supporting files in `installDir\target\h\make` makes it possible for each BSP or application makefile to be terse, specifying only the essential parameters that are unique to the object being built.

Example 5-1 shows the makefile from the `installDir\target\config\mbx860` directory; the makefile for any other BSP is similar. Two variables are defined at the start of the makefile: `CPU`, to specify the target architecture; and `TOOL` to identify what compilation tools to use. Based on the values of these variables and on the environment variables defined as part of your Tornado configuration, the makefile selects the appropriate set of definitions from `installDir\target\h\make`. After the standard definitions, several variables define properties specific to this BSP. Finally, the standard rules for building a BSP on your host are included.

Example 5-1  Makefile for mbx860

```make
# Makefile - makefile for target/config/mbx860
#
# Copyright 1984-2001 Wind River Systems, Inc.
# Copyright 1997,1998 Motorola, Inc., All Rights Reserved
#
# DESCRIPTION
# This file contains rules for building VxWorks for the
# MBX Board with a PowerPC 860 or PowerPC 821 processor.
#
# INCLUDES
#
# CPU              = PPC860
# TOOL             = gnu
#
TGT_DIR = $(WIND_BASE)/target

include $(TGT_DIR)/h/make/defs.bsp
#include $(TGT_DIR)/h/make/make.$(CPU)$TOOL
#include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)

## Only redefine make definitions below this point, or your definitions will
## be overwritten by the makefile stubs above.
```

TARGET_DIR = mbx860
VENDOR = Motorola
BOARD = MBX860

## The constants ROM_TEXT_ADRS, ROM_SIZE, and RAM_HIGH_ADRS are defined
## in config.h and Makefile. All definitions for these constants must be
## identical.

ROM_TEXT_ADRS = FE000100 # ROM entry address
ROM_SIZE = 00080000 # number of bytes of ROM space
RAM_LOW_ADRS = 00010000 # RAM text/data address
RAM_HIGH_ADRS = 00200000 # RAM text/data address
USR_ENTRY = usrInit
BOOT_EXTRA = mbxI2c.o mbxALib.o
MACH_EXTRA = mbxALib.o
RELEASE += bootrom.bin

## Only redefine make definitions above this point, or the expansion of
## makefile target dependencies may be incorrect.

include $(TGT_DIR)/h/make/rules.bsp

The following make include file defines variables. This file is useful for
application-module makefiles, as well as for BSP makefiles.

defs.bsp
  Standard variable definitions for a VxWorks run-time system.

The following include file defines make targets and the rules to build them. This
file is usually not required for building application modules in separate
directories, because most of the rules it defines are specific to the VxWorks run-
time system and boot programs.

rules.bsp
  Rules defining all the standard targets for building a VxWorks run-time
  system (described in 5.4 Building a VxWorks System Image, p.178 and
  5.6 Creating Bootable Applications, p.191). The rules for building object code
  from C, C++, or assembly language are also spelled out here.

5.5.1 Make Variables

The variables defined in the make include files provide convenient defaults for
most situations, and allow individual makefiles to specify only the definitions that
are unique to each. This section describes the make variables most often used to
specify properties of BSPs or applications. The following lists are not intended to be comprehensive; see the `make` include files for the complete set.

⚠️ **CAUTION:** Certain `make` variables are intended specifically for customization; see *Variables for Customizing the Run-Time*, p.189. Be very cautious about overriding other variables in BSP makefiles. They are described in the following sections primarily for expository purposes.

### Variables for Compilation Options

The variables grouped in this section are useful for either BSP makefiles or application-module makefiles. They specify aspects of how to invoke the compiler.

**CFLAGS**

The complete set of option flags for any invocation of the C compiler. This variable gathers the options specified in `CC_COMPILER`, `CC_WARNINGS`, `CC_OPTIM`, `CC_INCLUDE`, `CC_DEFINES`, and `ADDED_CFLAGS`.

**C++FLAGS**

The complete set of option flags for any invocation of the C++ compiler. This variable gathers together the options specified in `C++_COMPILER`, `C++_WARNINGS`, `CC_OPTIM`, `CC_INCLUDE`, `CC_DEFINES`, and `ADDED_C++FLAGS`.

**CC_COMPILER**

Option flags specific to compiling the C language. Default: `-ansi -nostdinc`.

**C++_COMPILER**

Option flags specific to compiling the C++ language. Default: `-ansi -nostdinc`.

**CC_WARNINGS**

Option flags to select the level of warning messages from the compiler, when compiling C programs. Two predefined sets of warnings are available: `CC_WARNINGS_ALL` (the compiler’s most comprehensive collection of warnings) and `CC_WARNINGS_NONE` (no warning flags). Default: `CC_WARNINGS_ALL`.

**C++_WARNINGS**

Option flags to select the level of warning messages from the compiler, when compiling C++ programs. The same two sets of flags are available as for C programs. Default: `CC_WARNINGS_NONE`. 
CC_OPTIM
Optimization flags. Three sets of flags are predefined for each architecture:
CC_OPTIM_DRIVER (optimization level appropriate to a device driver),
CC_OPTIM_TARGET (optimization level for BSPs), and
CC_OPTIM_NORMAL (optimization level for application modules).
Default: CC_OPTIM_TARGET.

CC_INCLUDE
Standard set of header-file directories. To add application-specific header-
file paths, specify them in EXTRA_INCLUDE.

CC_DEFINES
Definitions of preprocessor constants. This variable is predefined to
propagate the makefile variable CPU to the preprocessor, to include any
constants required for particular target architectures, and to include the
value of the makefile variable EXTRA_DEFINE. To add application-specific
constants, specify them in EXTRA_DEFINE.

Variables for BSP Parameters

The variables included in this section specify properties of a particular BSP, and are
thus recorded in each BSP makefile. They are not normally used in application-
module makefiles.

TARGET_DIR
Name of the BSP (used for dependency lists and name of documentation
reference entry). The value matches the bspname directory name.

ROM_TEXT_ADRS
Address of the ROM entry point. Also defined in config.h; the two
definitions must match.

ROM_SIZE
Number of bytes available in the ROM. Also defined in config.h; the two
definitions must match.

RAM_HIGH_ADRS
RAM address where the boot ROM data segment is loaded. Must be a high
enough value to ensure loading VxWorks does not overwrite part of the
ROM program. Also defined in config.h; the two definitions must match.
See 5.6 Creating Bootable Applications, p. 191 for more discussion.

RAM_LOW_ADRS
Beginning address to use for the VxWorks run-time in RAM.
HEX_FLAGS
GNU `objcopy` flags. These vary by architecture; for more information, see the GNU ToolKit User’s Guide.

LDFLAGS
Linker options for the static link of VxWorks and boot ROMs.

ROM_LDFLAGS
Additional static-link option flags specific to boot ROM images.

Variables for Customizing the Run-Time

The variables listed in this section make it easy to control what facilities are statically linked into your run-time system. You can specify values for these variables either from the make command line, or from your own makefiles (when you take advantage of the predefined VxWorks make include files).

CONFIG_ALL
Location of a directory containing the architecture-independent BSP configuration files. Set this variable if you maintain several versions of these files for different purposes. Default: `installDir\target\config\all`.

LIB_EXTRA
Linker options to include additional archive libraries (you must specify the complete option, including the `-L` for each library). These libraries appear in the link command before the standard VxWorks libraries.

MACH_EXTRA
Names of application modules to include in the static link to produce a VxWorks run-time. See 5.6 Creating Bootable Applications, p.191.

BOOT_EXTRA
Names of application modules to include in the static link to produce a VxWorks boot image but not in a normal VxWorks image.

ADDED_MODULES
Do not define a value for this variable in makefiles. This variable is reserved for adding modules to a static link from the make command line. Its value is used in the same way as MACH_EXTRA, to include additional modules in the link. Reserving a separate variable for use from the command line avoids the danger of overriding any object modules that are already listed in MACH_EXTRA.
EXTRA_INCLUDE
Preprocessor options to define any additional header-file directories required for your application (specify the complete option, including -I).

EXTRA_DEFINE
Definitions for application-specific preprocessor constants (you must specify the complete option, including the -D).

ADDED_CFLAGS
Application-specific compiler options for C programs.

ADDED_C++FLAGS
Application-specific compiler options for C++ programs.

5.5.2 Using Makefile Include Files for Application Modules

You can exploit the VxWorks makefile structure to put together your own application makefiles quickly and tersely. If you build your application directly in a BSP directory (or in a copy of one), you can use the makefile in that BSP, by specifying variable definitions (Variables for Customizing the Run-Time, p. 189) that include the components of your application.

You can also take advantage of the Tornado makefile structure if you develop application modules in separate directories. Example 5-2 illustrates the general scheme. Include the makefile headers that specify variables, and list the object modules you want built as dependencies of a target. This simple scheme is usually sufficient, because the Tornado makefile variables are carefully designed to fit into the default rules that make knows about.2

NOTE: The target name exe is the Tornado convention for a default make target. You may either use that target name (as in Example 5-2), or define a different default rule in your makefiles. However, there must always be an exe target in makefiles based on the Tornado makefile headers (even if the associated rules do nothing).

Example 5-2 Skeleton Makefile for Application Modules

```
# Makefile - makefile for ...
#
# Copyright ...
#
```

2. However, if you are working with C++, it may be also convenient to copy the .cpp.out rule from installDir\target\h\make\rules.bsp into your application’s makefile.
5.6 Creating Bootable Applications

As you approach a final version of your application, you will probably want to add modules to the bootable system image, and include startup of your application with the system initialization routines. In this way, you can create a bootable application, which is completely initialized and functional after booting, without requiring any interaction with the host-resident development tools.

5.6.1 Linking Bootable Applications

Linking the application with VxWorks is a two-step process. You must include the application initialization code in config.h, and you must modify the makefile to link the application statically with VxWorks.

To include the application code in config.h, you must:

- **#define INCLUDE_USER_APPL** (change **#undef** to **#define**)
- Modify the code fragment that defines **USER_APPL_INIT**. A template is provided; modify it as necessary to start your application:

```c
#define USER_APPL_INIT
{
    IMPORT int myAppInit();
    taskSpawn ("myApp", 30, 0, 5120,
               myAppInit, 0x1, 0x2, 0x3, 0,0,0,0,0,0,0,0);
}
```
To include your application modules in the bootable system image, add the names of the application object modules (with the .o suffix) to MACH_EXTRA in the makefile. For example, to link the module myMod.o, add a line like the following:

```
MACH_EXTRA = myMod.o
```

Building the system image with the application linked in is the final part of this step. In the target directory, execute the following command:

```
c:\> make vxWorks
```

Application size is usually an important consideration in bootable applications. Generally, VxWorks boot ROM code is copied to a start address in RAM above the constant RAM_HIGH_ADRS, and the ROM in turn copies the downloaded system image starting at RAM_LOW_ADRS. The values of these constants are architecture dependent, but in any case the system image must not exceed the space between the two. Otherwise the system image overwrites the boot ROM code while downloading, thus killing the booting process.

To help avoid this, the last command executed when you make a new VxWorks image is `vxsize`, which shows the size of the new executable image and how much space (if any) is left in the area below the space used for ROM code:

```
vxsize 386 -v 00100000 00020000 vxWorks
vxWorks: 612328(t) + 69456(d) + 34736(b) = 716520 (235720 bytes left)
```

If your new image is too large, `vxsize` issues a warning. In this case, you can reprogram the boot ROMs to copy the ROM code to a sufficiently high memory address by increasing the value of RAM_HIGH_ADRS in config.h and in your BSP’s makefile (both values must agree). Then rebuild the boot ROMs by executing the following command:

```
c:\> make bootrom.hex
```

The binary image size of typical boot ROM code is 128KB or less. This small size is achieved through compression; see Boot ROM Compression, p. 195. The compressed boot image begins execution with a single uncompressed routine, which uncompresses the remaining boot code to RAM. To avoid uncompressing and thus initialize the system a bit faster, you can build a larger, uncompressed boot ROM image by specifying the `make` target bootrom_uncmp.hex.
5.6.2 Creating a Standalone VxWorks System with a Built-in Symbol Table

It is sometimes useful to create a VxWorks system that includes a copy of its own symbol table. The procedure for building such a system is somewhat different from the procedure described in 5.6.1 Linking Bootable Applications, p. 191. No change is necessary to *usrConfig.c*. A different make target, *vxWorks.st*, specifies the standalone form of VxWorks:

```
C:\> make vxWorks.st
```

The rules for building *vxWorks.st* create a module *usrConfig_st.o*, which is the *usrConfig.c* module compiled with the STANDALONE flag defined. The STANDALONE flag causes the *usrConfig.c* module to be compiled with the built-in system symbol table, the target-resident shell, and associated interactive routines.

The STANDALONE flag also suppresses the initialization of the network. If you want to include network initialization, define STANDALONE_NET in either of the header files `installDir\target\config\bspname\config.h` or `installDir\target\config\all\configAll.h`.

VxWorks is linked as described previously, except that the first pass through the loader does not specify the final load address; thus the output from this stage is still relocatable. The `makeSymTbl` tool is invoked on the loader output; it constructs a data structure containing all the symbols in VxWorks. This structure is then compiled and linked with VxWorks itself to produce the final bootable VxWorks object module.

To include your own application in the system image, add the object modules to the definition of MACH_EXTRA and follow the procedures discussed in 5.6.1 Linking Bootable Applications, p. 191.

Because *vxWorks.st* has a built-in symbol table, there are some minor differences in how it treats VxWorks symbols, in contrast with the host symbol table used by Tornado tools through the target server. First, VxWorks symbol table entries cannot be deleted from the *vxWorks.st* symbol table. Second, no local (static) VxWorks symbols are present in *vxWorks.st*.

---

3. *vxWorks.st* suppresses network initialization, but it includes the network. The STANDALONE option defines INCLUDE_STANDALONE_SYM_TBL and INCLUDE_NETWORK, and undefines INCLUDE_NET_SYM_TBL and INCLUDE_NET_INIT. The alternative option STANDALONE_NET includes INCLUDE_NET_INIT.
5.6.3 Creating a VxWorks System in ROM

To put VxWorks or a VxWorks-based application into ROM, you must enter the object files on the loader command line in an order that lists the module `romInit.o` before `sysALib.o`. Also specify the entry point option `-e _romInit`. The `romInit()` routine initializes the stack pointer to point directly below the text segment. It then calls `bootInit()`, which clears memory and copies the `vxWorks` text and data segments to the proper location in RAM. Control is then passed to `usrInit()`. A good example of a ROM-based VxWorks application is the VxWorks boot ROM program itself. The file `installDir\target\config\all\bootConfig.c` is the configuration module for the boot ROM, replacing the file `usrConfig.c` provided for the default VxWorks development system. The makefiles in the target-specific directories contain directives for building the boot ROMs, including conversion to a file format suitable for downloading to a PROM programmer. Thus, you can generate the ROM image with the following `make` command:

```
c:\> make bootrom.hex
```

Tornado makefiles also define a ROMable VxWorks run-time system suitable for use with Tornado tools as the target `vxWorks.res_rom_nosym`. To generate this image in a form suitable for writing ROMs, run the following command:

```
c:\> make vxWorks.res_rom_nosym.hex
```

VxWorks target makefiles also include the entry `vxWorks.st_rom` for creating a ROMable version of the standalone system described in 5.6.2 Creating a Standalone VxWorks System with a Built-in Symbol Table, p.193. The image `vxWorks.st_rom` differs from `vxWorks.st` in two respects: (1) `romInit` code is loaded as discussed above, and (2) the portion of the system image that is not essential for booting is compressed by approximately 40 percent using the VxWorks `compress` tool (see Boot ROM Compression, p.195).

To build the form of this target that is suitable for writing into a ROM (most often, this form uses the Motorola S-record format), enter:

```
c:\> make vxWorks.st_rom.hex
```

When adding application modules to a ROMable system, size is again an important consideration. Keep in mind that by using the `compress` tool, a configuration that normally requires a 256-KB ROM may well fit into a 128-KB ROM. Be sure that `ROM_SIZE` (in both `config.h` and the makefile) reflects the capacity of the ROMs used.
Boot ROM Compression

VxWorks boot ROMs are compressed to about 40 percent of their actual size using a binary compression algorithm, which is supplied as the tool `compress`. When control is passed to the ROMs on system reset or reboot, a small (8 KB) uncompression routine, which is *not* itself compressed, is executed. It then uncompresses the remainder of the ROM into RAM and jumps to the start of the uncompressed image in RAM. There is a short delay during the uncompression before the VxWorks prompt appears. The uncompression time depends on CPU speed and code size; it takes about 4 seconds on an MC68030 at 25 MHz.

This mechanism is also available to compress a ROMable VxWorks application. The entry for `vxWorks.st_rom` in the architecture-independent portion of the makefile, `installDir\target\h\make\rules.bsp`, demonstrates how this can be accomplished. For more information, see also the reference manual entries for `bootInit` and `compress`.

5.7 Building Projects from a BSP

In some cases, you may wish to change and customize your BSP using the techniques described in this chapter, and then build the VxWorks image for use by application developers using the project facility. This is the one case where you can “mix” the two methods of configuration. Using one of these `make` targets creates a project based on the BSP you have created with all your customizations. This project can serve as a base for further development.

⚠️ **WARNING:** If you make additional changes to the configuration files of your BSP after you have begun using it with the project facility, these changes will not be available for subsequent project facility use because the project files override `config.h` and other configuration files.

The following `make` targets are available:

- **make prj_default**
  Builds a single project using the default toolchain and building all four default build specifications. These are `default`, `default_rom`, `default_romCompress`, and `default_romResident`. 
make prj_gnu
Builds a single project using the GNU toolchain and building all four default
build specifications.

make prj_diab
Builds a single project using the Diab toolchain and building all four default
build specifications.

make prj_diab_def
Builds a single project using the Diab toolchain and building a single, default
build specification.

make prj_gnu_def
Builds a single project using the GNU toolchain and building a single, default
build specification.

make bsp2prj
The same as prj_default_one, which is prj_default with only the single,
default build specification.

  c:\> make [CPU=XXXX TOOL=YYYY] bsp2prj

A Tcl script is also available for bsp2prj; you can run it from the DOS prompt
(order is important):

  c:\> wtxtcl /vobs/wpwr/host/src/hutils/bsp2prj.tcl [CPU TOOL]
6
VxSim
Integrated Simulator and Full Simulator (Optional)

6.1 Introduction

VxSim, the VxWorks simulator, is a port of VxWorks to the various host architectures. It provides a simulated target for use as a prototyping and test-bed environment. In most regards, its capabilities are identical to a true VxWorks system running on target hardware. Users link in applications and rebuild the VxWorks image exactly as they do in any VxWorks cross-development environment using a standard BSP.

The difference between VxSim and the VxWorks target environment is that in VxSim the image is executed on the host machine itself as a host process. There is no emulation of instructions, because the code is for the host’s own architecture. A communication mechanism is provided to allow VxSim to obtain an Internet IP address and communicate with the Tornado tools on the host (or with other nodes on the network) using the VxWorks networking tools.

Because target hardware interaction is not possible, device driver development may not be suitable for simulation. However, the VxWorks scheduler is implemented in the host process, maintaining true tasking interaction with respect to priorities and preemption. This means that any application that is written in a portable style and with minimal hardware interaction should be portable between VxSim and VxWorks.

The basic functionality of VxSim is included with the Tornado tools and is preconfigured to allow immediate access to the simulated target. The integrated simulator does not include networking and provides only single instance usage. The VxSim full simulator is an optional product providing for networking and multiple instance usage.
The key differences between VxSim and other BSPs are summarized below. For a detailed discussion of subtle implementation differences which may affect application development, see 6.4 Architecture Considerations, p.204.

**Integrated Simulator**

VxSim has only a few differences from VxWorks:

- **Drivers.** Because device drivers require direct hardware interaction, most VxWorks device drivers are not available with VxSim.

- **File System.** VxSim defaults to using a pass-through file system (passFs) to access files directly on the workstation. (See the online reference for **passFsLib** under VxWorks Reference Manual> Libraries.) Most VxWorks targets default to using **netDrv** to access files on the host.

- **Networking.** Networking is not available in the base product.

**Full Simulator**

The VxSim full simulator provides full network capability for your simulator. This optional product also allows you to run more than one instance of VxSim on your host.

In order to simulate the network IP connectivity of a VxWorks target, the VxSim full simulator includes special drivers that operate using IP addresses. The ULIP network interface is available for Windows hosts.

All interfaces provide an I/O-based interface for IP networking that allows VxSim processes to be addressed at the IP level. When multiple programs are run, they can send packets to each other directly. This is because the host hands the packets back and forth; that is, the host OS effectively becomes a router with multiple interfaces.

### 6.2 Integrated Simulator

All the functionality of the integrated simulator is available with the optional full simulator. All the information in this section applies to both versions of VxSim. For information specific to the full simulator product, see 6.5 Configuring the VxSim Full Simulator, p.207.
Installation and Configuration

Tornado 2.2 comes configured with basic VxSim on all hosts. Installing and starting Tornado as described in the Tornado Getting Started Guide installs the integrated VxSim.

Starting VxSim

VxSim automatically starts when you request a function that requires a connection to a target. For example, when you request download of a module, if you have not started a target server VxSim and a target server are automatically started.

You can also start VxSim from the VxSim icon in Tornado environment or from the command line or the Start>Run dialog box using the command \texttt{vxWorks}. Available options are:

\begin{itemize}
  \item \texttt{-processorNumber}
    \begin{itemize}
    \item Set the processor number [0-15] (default is 0)
    \end{itemize}
  \item \texttt{-ram bytes}
    \begin{itemize}
    \item Set the memory size in decimal (default is 2Mbytes)
    \end{itemize}
\end{itemize}

\textbf{WARNING:} On real targets, you can use \texttt{bootChange()} to boot another VxWorks core file on the next reboot. On simulators, changing the core file using \texttt{bootChange()} has no effect; in other words, on the next reboot, the simulator will not start with the core file set in the boot line.

Changing the Simulator Boot Line

Because the hardware environment is different from the simulator environment, \texttt{bootChange() } does not behave the same way on simulators as it does on real targets.

<table>
<thead>
<tr>
<th>Table 6-1</th>
<th>Simulator Boot Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>Comments</td>
</tr>
<tr>
<td>boot device</td>
<td>Do not change.</td>
</tr>
<tr>
<td>processor number</td>
<td>Do not change.</td>
</tr>
<tr>
<td>host name</td>
<td>Do not change.</td>
</tr>
</tbody>
</table>
Rebooting VxSim

As with other targets, you can reboot VxSim by typing \texttt{CTRL+X} in the VxSim window.

Exiting VxSim

Exit VxSim by closing the VxSim window.
**Back End**

The integrated simulator uses the pipe back end (INCLUDE_WDB_COMM_PIPE), which is configured by default, to communicate with the target session.

**System-Mode Debugging**

System-mode debugging allows developers to suspend the entire VxWorks operating system. One notable application of system mode is to debug ISRs, which—because they run outside any task context—are not visible to debugging tools in the default task mode. For more discussion of system mode, see the chapters 7. Shell and 10. Debugger.

The Windows integrated simulator is configured by default for system mode debugging.

**Symbols**

Particular care must be taken when using absolute symbols from loaded object modules in the simulators. The VxWorks simulators execute within the memory space of a host operating system. Their actual execution address space is more constrained than is the case for the real VxWorks operating system. Absolute references to addresses must be carefully chosen in order to point to memory areas actually existing and allocated to the simulator. The values of absolute symbols defined within object modules are not modified by the loader so these values (in other words, addresses) must be set correctly by the code developers.

**File Systems**

VxSim can use any VxWorks file system.

The default file system is the pass-through file system, ntPassFs, which is unique to VxSim. ntPassFs allows direct access to any files on the host. Essentially, the VxWorks functions open(), read(), write(), and close() eventually call the host equivalents in the host library libc.a. With ntPassFs, you can open any file available on the host, including NFS-mounted files. By default, the

---

1. System mode is sometimes also called external mode, reflecting that the target agent operates externally to the VxWorks system in this mode.
**INCLUDE_NTPASSFS** macro is enabled to cause this file system to be mounted on startup.

In the target shell, a path name should be prefixed by **host**: to prevent the Windows disk device from interpreting it as a VxWorks device. Moreover, you can use both / and \ in the path; be careful when using \ in a string as the next character will be interpreted as a special character. For example, all these usages yield expected results:

- `ld < host:c:/test/testFile.o`
- `ld < host:c:\test\testFile.o`
- `cd "host:c:/test"
- `cd "host:c:\test"

However, the following does not work:

- `cd "host:c:\test"

In WindSh, do not use the **host**: prefix in the path name.

For more information on ntPassFs, see the reference entry for **ntPassFsLib**. For more information on other VxWorks file systems, see the *VxWorks Programmer’s Guide: Local File Systems*.

### 6.3 Building Applications

The following sections describe how to use the VxSim compilers. The recommended way to build VxSim modules is to use the project tool. For complete information on this tool, see 4. *Projects*. If you are using manual methods in your project, the information required for manual builds and loading is summarized below.

This information applies to using manual methods on both the integrated version of VxSim and the full simulator product.
Defining the CPU Type

Setting the preprocessor variable CPU ensures that VxWorks and your applications build with the appropriate features enabled. Define this variable to SIMNT for all Windows hosts.

The Toolkit Environment

All VxWorks simulators use the GNU C/C++ compiler.

⚠️ CAUTION: The compiler used by the Tornado tools to compile VxSim applications for Windows is the GNU C/C++ compiler rather than any MicroSoft tools. This provides for greater compatibility between VxSim and VxWorks environments.

Compiling C and C++ Modules

Only the GNU compiler is supported for SIMNT: the Diab compiler is not supported. If you compile using the IDE build facilities, default build settings are already in place. If you wish to modify the defaults, or if you wish to build from the command line, the following information may be helpful.

The following is an example of a compiler command line for VxSim development. The file to be compiled in this example has a base name of applic.

```
% cc -g -mpentium -ansi -D_RW_MULTI_THREAD -D_REENTRANT -fno-built-in
   -fno-defer-pop -I. -IinstallDir/target/h/ -DCPU=SIMNT -D_TOOL_FAMILY=gnu
   -D_TOOL=gnu -c applic.c
```

Option Definitions

The options shown in the example and other compiler options are detailed in the online version of the GNU ToolKit User’s Guide. Wind River supports compiler options used in building Wind River software; see the Guide for a list. Other options are not supported, although they are available with the tools as shipped.

Linking an Application to VxSim

Linking and loading for VxSim are identical to other BSPs. See the VxWorks Programmer’s Guide: Configuration and Build.
6.4 Architecture Considerations

The information in this section highlights differences between VxSim (both the integrated and full versions) and other VxWorks BSPs. These differences should be taken into consideration as you develop applications on VxSim that will eventually be ported to another target architecture.

VxSim uses the VxWorks scheduler, which behaves the same way as for any other VxWorks architecture (see the *VxWorks Programmer's Guide: Basic OS*). The BSP is extensible; for example, pseudo-drivers can be written for additional timers, serial drivers, and so forth.

The rest of this section discusses some details of the VxSim implementation. Differences between VxSim and other VxWorks environments are noted where appropriate.

**Supported Configurations**

Most of the optional features and device drivers for VxWorks are supported by VxSim. The few that are not are hardware devices (SCSI, Ethernet), ROM configurations, and so on. The BSP makefile builds only the images `vxWorks` and `vxWorks.st` (standalone VxWorks).

**Endianness**

The Windows simulator uses a little-endian environment.

**Simulator Timeout**

Occasionally a simulator session loses its target server connection due to the many things competing for CPU time on the host. If you find that your application is frequently losing its target server connection, adjust the back end timeout (`-Bt`) and back end retry (`-Br`) parameters when starting the target server with Tornado. To do this from the Configure Target Servers dialog, select Miscellaneous and set the appropriate options using the Other Options field. For example, you may want to increase the back end timeout from 1 to 3 and the resend parameter from 3 to 4:

```
-Bt 3 -Br 4
```
You can also add this string to the `tgtsvr` command when you start the target server from the command line.

**The BSP Directory**

Aside from the following exceptions, the VxSim BSP is the same as a VxWorks BSP:

- The `sysLib.c` module contains the same essential functions: `sysModel()`, `sysHwInit()`, and `sysClkConnect()` through `sysNvRamSet()`. Because there is no bus, `sysBusToLocalAdrs()` and related functions have no effect.

- The file `winSio.c` ultimately calls the host OS `read()` and `write()` routines on the process’s standard input and output. Nevertheless, it supports all the functionality provided by `tyLib.c`.

- The `simpcDrv.a` file is the library for `simpc` BSP drivers.

- The configuration header `config.h` is minimal:
  - It does not reference a `bspname.h` file.
  - Most network devices are excluded.
  - The boot line has no fixed memory location. Instead, it is stored in the variable `sysBootLine` in `sysLib.c`.

- The `Makefile` is the standard version for VxWorks BSPs. It does not build boot ROM images (although the makefile rules remain intact); it can only build `vxWorks` and `vxWorks.st` (standalone) images. The final linking does not arrange for the TEXT segment to be loaded at a fixed area in RAM, but follows the usual loading model. The makefile macro `MACH_EXTRA` is provided so that users can easily link their application modules into the VxWorks image if they are using manual build methods.

The BSP file `sysLib.c` can be extended to emulate the eventual target hardware more completely.

**Interrupts**

Windows messages are used to simulate hardware interrupts. For example, VxSim uses messages 0xc000 - 0xc010 to simulate interrupts from ULIP, the pipe back end, and so forth. The messages are the VxSim equivalent to Interrupt Service Routines (ISRs) on other VxWorks targets. You can install ISRs in VxSim to handle these “interrupts.” Not all VxWorks functions can be called from ISRs; see the `VxWorks`
Programmer’s Guide: Basic OS. To run ISR code during a future system clock interrupt, use the watchdog timer facilities. To run ISR code during auxiliary clock interrupts, use the sysAuxClkxxx() functions.

Table 6-2 shows how the message table is set up.

Table 6-2 Interrupt Assignments

<table>
<thead>
<tr>
<th>Interrupts</th>
<th>Assigned To</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xc000-0xc010</td>
<td>host messages</td>
</tr>
<tr>
<td>0xc011 on</td>
<td>available for user messages</td>
</tr>
</tbody>
</table>

Pseudo-drivers can be created to use these interrupts. Interrupt code must be connected with the standard VxWorks intConnect() mechanism.

For example, to install an ISR that logs a message whenever host message WM_TIMER_CLOCK arrives, execute the following:

```c
intConnect (0xc011, logMsg, "Help!\n")
```

Then send message 0xc011 to VxSim from a host task. Every time the message is received, the ISR (logMsg() in this case) runs.

If a VxSim task reads from a host device, the task would normally block while reading; however, this would stop the VxSim process entirely until data is ready. Instead the device is put into asynchronous mode so that a message is sent whenever data becomes ready. In this case, an input ISR reads the data, puts it in a buffer, and unblocks some waiting task.

Since VxSim uses the task’s stack when taking interrupts, the task stacks are artificially inflated to compensate. You may notice this if you spawn a task of a certain size and then examine the stack size.

Clock and Timing Issues

The execution times of VxSim functions are not, in general, the same as on a real target. For example, the VxWorks intLock() function is normally very fast because it just writes to the processor status register. However, under VxSim, intLock() is relatively slow because it takes a host semaphore, allowing other processes to run.

The clock facilities are provided by the Windows API SetTimer() for both the system and auxiliary clocks. The problem with using SetTimer() for the target system clock is that it produces inaccurate timings when VxSim is swapped out as
a host process. On the other hand, the timing of VxSim is, in general, different than on an actual target, so this is not really a problem.

**NOTE:** Because VxSim is a host process, it shares resources with all other processes and is swapped in and out. In addition, the kernel's idle loop has been modified to suspend VxSim until a signal arrives (rather than busy waiting), thus allowing other processes to run.

The `spy()` facility is built on top of the auxiliary clock. The task monitoring occurs during each interrupt of the auxiliary clock to see which task is executing or if the kernel is executing. Because the profiling timer includes host system time and user time, discrepancies can occur, especially if intensive host I/O occurs.

**WindView Instrumentation on the Windows Simulator**

Due to some specifics of the Windows emulator, there is a critical section at the end of the `intUnlock()` routine where interrupts can occur but scheduling is forbidden. If a reschedule is necessary at the end of the interrupt, it is not done when the interrupt is exited but is delayed until the end of the critical section of the `intUnlock()` routine.

This behavior can have an impact on a WindView graph. If a high priority task is made ready within an interrupt handler, this task may not be run when the interrupt is exited but instead may run slightly later.

**NOTE:** This delay of the rescheduling has no impact on the relative scheduling between VxWorks tasks.

### 6.5 Configuring the VxSim Full Simulator

This section contains information pertaining only to the VxSim full simulator. (All information in previous sections also pertains to that product, as well as to the integrated version.) The VxSim full simulator provides networking facilities. Most of the special considerations associated with it are network considerations.

If you purchase the VxSim optional full simulator for networking, you must take additional configuration steps:
- Install the VxSim full simulator using SETUP, either when you install Tornado 2.2 or at a later time. (For more information, see the Tornado Getting Started Guide.)

- Install the appropriate network driver on your host. (See Installing VxSim Network Drivers, p.208.)

- Configure VxWorks to use networking, rebuild it, and download it using either the project facility or manual methods. (See Configuring VxSim for Networking, p.212.)

**WARNING:** Project facility configuration and building of projects is independent of the methods used for configuring and building applications prior to Tornado 2.x (which included manually editing config.h and configAll.h). Use of the project facility is the recommended, and is much simpler. However, the manual method may still be used (see 5. Command-Line Configuration and Build for details). Avoid using the two methods together for the same project except where specific BSP and driver macros are not available in the project facility.

- Be sure to correctly set target server options for the full simulator:
  - Click the Launch Simulator icon. Select Custom-built simulator and set the path to a full simulator project directory. Click OK. The VxSim Launch: Launch Target Server window pops up. Click on Details. Select Full simulator to set default target server options for the full simulator. Finally, click OK to launch a target server.
  - Click on Tools>TargetServer>Configure. In the Configure Target Servers window, select wdbrpc in the Available Back Ends list, and set the IP address of the simulator in the Target Name/IP Address field. Set the target server name to vxsim. The target server command line should be:

    `tgtsvr.exe 192.168.255.1 -n vxsim -V -B wdbrpc`

    (By default you should also see WindView options)

**Installing VxSim Network Drivers**

The SETUP tool writes the appropriate host drivers on your disk, but they must be installed on your host operating system.

For Windows hosts, the simpB BSP includes an NDIS driver called the ULIP driver. Follow these steps to add the ULIP driver to your Windows host.
Installing ULIP on a Windows NT Host

1. From the Start menu select Settings>Control Panel>Network.
2. Click the Adapters tab in the Network window, click the Add button, click Have Disk in the Select Network Adaptor window, select the host\x86-win32\bin directory of your Tornado installation, and click OK.
3. Click on Ulip Virtual Adapter and click OK. ULIP is added to the network adapters list.
4. Enter an IP address of the form nn.nn.nn.254 (for example,192.168.255.254).
5. Then select Show bindings for all adapters from the Bindings tab, select Ulip Virtual Adapter>WINS Client (TCP/IP)>NetBIOS Interface, and click Disable and OK.
6. If you want to enable network communication between simulators and the outside world, select Settings>Control Panel>Network from the Start menu, then click the Protocols tab and double-click on TCP/IP Protocol. From the Routing tab of the TCP/IP Properties window, check Enable IP Forwarding. Click OK and Close.
7. Restart the computer so the new settings take effect.

Installing ULIP on a Windows 2000 Host

1. From the Start menu, select Settings>Control Panel>Add/Remove hardware. This launches the driver installation wizard.
2. Click the Next button. Then click the subsequent Next button. This displays the Choose a Hardware Device dialog box.
3. From the dialog box, select the line Add a new device and click the Next button.
4. Select the option, No, I want to select the hardware from a list, and click again on the Next button.
5. Select Network adapters, and click on Next. This stage can take several minutes.
6. The Select Network Adapter dialog box appears. Click on the Have disk button, and select the host\x86-win32\bin directory of your Tornado installation and click OK.
7. Then select the line WindRiver Ulip, and click on Next. A new dialog box appears, click again on Next.
8. Select Yes from the Digital Signature Not Found dialog box.

9. When the next dialog box appears, ensure it confirms that the installation completed without errors, and finally, click on the Finish button.

10. Starting again from the Windows desktop, right click on My Network Places, and select Properties.

11. Right click on the last Local Area Connection # icon and select Properties. (You can rename this Local Area Connection # title with WindRiver Ulip by selecting Rename instead of Properties).

12. Select the line Internet Protocol (TCP/IP) and click on Properties. (If the TCP/IP protocol is not installed, refer to the MS Windows help for information on installing it).

13. Enter an IP address of the form \nnn.nnn.nnn.254 (for example, 192.168.255.254) and 255.255.255.0 as subnet mask, and click on Advanced.

14. In the WINS tab, select Disable NetBIOS over TCP/IP and click OK to properly close all the remaining windows.

15. If you want to enable network communication between simulators and the outside world, select Setting>Control Panel>Administrative tools from the Start menu, double-click on Services, then search for the Routing and Remote access service. If it is disabled, right-click on it and change the startup type to Automatic so it will be started when Windows starts. Click OK to validate your change. The service can now be started using the play icon on the main service window.

Installing ULIP on a Windows XP Host

1. From the Start menu select Settings>Control Panel>Add Hardware. This launches the driver installation wizard. Click Next.

2. The wizard searches for recently connected hardware; this stage can take several minutes. Then select Yes, I have already connected the hardware, and click Next.

3. From the Installed hardware list, select Add a new hardware device, and click Next.

4. Select Install the hardware that I manually select from a list [Advanced], click Next.

5. From the Common hardware types list, select Network adapters, click Next.

6. Click Have Disk, and select the host\x86-win32\bin directory of your Tornado installation.

7. Select WindRiver ULIP, and click Next twice
8. The Hardware Installation dialog reports that “WindRiver ULIP has not passed Windows Logo testing to verify its compatibility with Windows XP. You can safely ignore this warning and click Continue Anyway.

9. Then, from the Control Panel, launch Network Connections and right-click WindRiver ULIP connection (You can rename this connection).

10. From the WindRiver ULIP Properties panel, select Internet Protocol (TCP/IP), and click Properties.

11. Check Use the following IP address, and enter an IP address of the form \texttt{nnn.nnn.nnn.254} (for example, 192.168.255.254), and 255.255.255.0 as Subnet mask, and click on Advanced.

12. In the WINS tab, select Disable NetBIOS over TCP/IP and click OK to properly close all the remaining windows.

13. If you want to enable network communication between simulators and the outside world, select Setting>Control Panel>Administrative tools from the Start menu, double-click on Services, then search for the Routing and Remote access service. If it is disabled, right-click on it and change the startup type to Automatic so it will be started when Windows starts. Click OK to validate your change. The service can now be started using the play icon on the main service window.

**Uninstalling VxSim Network Drivers**

**Uninstalling ULIP on a Windows NT Host**

1. From the Start menu select Settings>Control Panel>Network.

2. Click the Adapters tab in the Network window.

3. Select Ulip Virtual Adapter, and click Remove.

**Uninstalling ULIP on a Windows 2000 or XP Host**

1. From the Start menu select Settings>Control Panel>System.

2. From the Hardware tab of the System Properties window, click the Device Manager button.

3. From the Device manager window, right-click on Network adapters>WindRiver ULIP, and select Uninstall.
Configuring VxSim for Networking

As with any other BSP, adding components to VxWorks requires including them, rebuilding VxWorks, the downloading and restarting it. The easiest method for doing this is to use the project facility. However, if you have used manual methods in your project, you should continue to use those methods.

For a discussion of networking as it relates to VxSim, see 6.5 Configuring the VxSim Full Simulator, p.207.

Using the Project Facility

Use the Create Project facility to create a bootable VxWorks image.

- On the VxWorks tab in the Project Workspace window, select the folder called network components.
- Right click and select Include 'network components' from the pop-up menu. Click OK to accept the defaults.
- Change WDB connection from WDB simulator pipe connection to WDB END driver connection or to WDB network connection. Then rebuild and download VxWorks.

For more information on using the configuration tool, see 4. Projects.

Using Manual Techniques

Edit target/config/simpc/config.h, and replace:

```c
#if TRUE
#undef INCLUDE_NETWORK
<...>
```

With:

```c
#if FALSE
#undef INCLUDE_NETWORK
<...>
```

Then rebuild and download VxWorks.

You must also change your target server configuration from wdbpipe to wdbrpc.

- If you are launching your target server from the command line:
  Replace -B wdbpipe by -B wdbrpc in the target server command line, and add the IP address of the simulator. The target server launch options should include the following:

```
-V `-B wdbrpc RW 192.168.255.1
```
If you are configuring and launching your target server from the GUI:

- Click the Launch Simulator icon. Select Custom-built simulator and set the path to a full simulator project directory. Click OK. In the Launch Target Server window, click on Details. Select Full simulator to enable the default target server options and click OK to launch the target server.

- Click on Tools>TargetServer>Configure. In the Configure Target Servers window, select wdbrpc in the Available Back Ends list, set the IP address of the simulator in the Target Name/IP Address field, and set the target server name to vxsim.

The target server command line should be:

tgtsvr.exe 192.168.255.1 -n vxsim1 -V -B wdbrpc

For additional information on configuring BSPs using manual methods, see the VxWorks Network Programmer’s Guide.

Running Multiple Simulators

When you install the optional VxSim component, your system is automatically configured to run up to 16 simulators. When you start VxSim from the GUI, you can specify the processor number to use from the VxSim Launch window. The processor number must be a positive number ranging from 0 (first instance: vxsim0) to 15 (last instance: vxsim15).

To start VxSim from the command line, the command takes the following form (where n is the processor number):

c:\> installDir\target\proj\fullSimulator\default> vxWorks -p n

System Mode Debugging

The full simulator does not support system mode debugging because of an incompatibility between the END and RPC back ends.

IP Addressing

All of the networking facilities available under VxWorks—for example, sockets, RPC, NFS—are available with VxSim. For VxSim to communicate with the outside
world, it must have its own target IP address as provided through a network interface.

Internet addressing is handled slightly differently among the available network interfaces. For each VxSim process, there are three associated IP addresses:

- Target IP – the address of each VxSim process, internal to your host.
- Local IP – your host’s address on the VxSim network, internal to your host.
- Host IP – your host’s address according to the network at your site.

The target IP address and the local IP address communicate according to the protocol of the chosen network interface. The host IP address is not directly relevant to the VxSim network.

Addressing is according to processor number, such that when you run VxSim with processor number $n$ (using the command `vxWorks -p n`), the network addresses packets as shown:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Target IP</th>
<th>Local IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>vxsim0</td>
<td>192.168.255.$n+1</td>
<td>local IP $n=0</td>
</tr>
<tr>
<td>vxsim1</td>
<td>192.168.255.$n+1</td>
<td>local IP $n=1</td>
</tr>
</tbody>
</table>

192.168.255.254 is the default host IP addr for the ULIP adapter; this configuration assigns IP addresses 192.168.255.1 through 192.168.255.16 to sixteen devices. If those IP addresses are not suitable, the address of the ULIP adapter may be changed, but do not use 127.$n.n.n$, which is reserved for other purposes on Windows. No change is necessary in the simulator configuration, since it gets its address dynamically from the ULIP adapter.
Aliases can be defined to assign names to simulator addresses, by adding the entries to the appropriate file:

```
| Windows NT  | C:\WINNT\system32\drivers\etc\hosts |
| Windows 2000 | C:\WINNT\system32\drivers\etc\hosts |
| Windows XP   | C:\Windows\system32\drivers\etc\hosts |
```

The following example shows a typical `hosts` file:

```
192.168.255.1  vxsim0
192.168.255.2  vxsim1
...             ...
192.168.255.16 vxsim15
```

### Choosing Processor Numbers for Distinct Devices

When you run VxSim with ULIP and specify processor number `n` (with the command `vxWorks -p n`), VxSim for Windows attaches to the IP number you specified when installing ULIP, which must not be 127.n.n.n.

Only one process at a time can open the same ULIP device; this is enforced in the ULIP driver. Thus, if you want multiple VxSim targets to use ULIP, you must give each of them a distinct processor number. If another VxSim process is already running with the same processor number, then the ULIP device cannot be opened (`ulip0` corresponds to processor 0), and the following message is displayed during the startup of VxSim:

```
Error: There is already a simulator running with this processor number. Please restart with a different number (/p <number> option).
Hit Any Key to Exit
```

When you run VxSim on NT, host entries are not automatically created until a simulator is started.

**WARNING:** The VxSim ULIP driver will not attach to a network interface if it is already in use, that is, `ul0` can be used by only one VxSim process. Use the `-p` flag to run VxSim with a different processor number; see *Starting VxSim*, p. 199.

### Setting Up Remote Access

You can add host-specific routing entries to the local host to allow remote hosts to connect to a local VxSim “target.” IP addresses are set up only for the host where the network simulation software is installed. The network interface does not have to be installed remotely; the remote host uses the local host as the gateway to the VxSim target.
In the example shown in Figure 6-2, host1 can communicate with vxsim0 or vxsim1 if IP forwarding is enabled. (See *Installing VxSim Network Drivers*, p. 208 for more information on how to enable IP forwarding.)

Contrast Figure 6-2 below with Figure 6-1 to see the way addresses are set up, paying particular attention to the addressing algorithm described in *IP Addressing*, p. 213.

To communicate from host1 to vxsim0 and vxsim1, type the following commands from host1 (requires administrative privilege):

```
c:\> route ADD 192.168.255.1 90.0.0.1 METRIC 1
```

```
c:\> route ADD 192.168.255.2 90.0.0.1 METRIC 1
```

Verify the success of the above commands by pinging vxsim0 from host1:

```
% ping 192.168.255.1
```

**NOTE:** If the remote host (host1) is a UNIX host, use the following commands (as root): on Solaris: % route add host 192.168.255.1 90.0.0.1 1

on Linux: % route add -host 192.168.255.1 gw 90.0.0.1 metric 1

To allow a VxSim process on one host to communicate with a VxSim process on a different host, you must make sure that the two VxSim processes have different IP
addresses. You must also make additional host-specific routes using unique
addresses for each process.

For example, to ping `vxsim2` from `host0` above, you must add an additional route
from `host0` as follows:

```bash
C:\> route ADD 192.168.255.3 90.0.0.2 METRIC 1
```

**NOTE:** You assign addresses for the Windows simulator when you install ULIP.
You can use 192.168.255.n or any other number except 127.n.n.n+1;127 is reserved
for other purposes on Windows.

To enable remote access to a simulator, IP forwarding must be enabled. For more
information on how to enable IP forwarding, see *Installing VxSim Network Drivers*,
p.208.
7.1 Introduction

The Tornado shell, WindSh, allows you to download application modules, and to invoke both VxWorks and application module subroutines. This facility has many uses: interactive exploration of the VxWorks operating system, prototyping, interactive development, and testing.

WindSh can interpret most C language expressions; it can execute most C operators and resolve symbolic data references and subroutine invocations. You can also interact with the shell through a Tcl interpreter, which provides a full set of control structures and lower-level access to target facilities. For a more detailed explanation of the Tcl interface, see 7.7 Tcl: Shell Interpretation, p.271.

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 7-1.

---

1. A target-resident version of the shell is also available; for more information, see VxWorks Programmer’s Guide: Target Shell.
The shell is where you directly exercise control; it receives your commands and executes them locally on the host, dispatching requests to the target server for any action involving the symbol table or target-resident programs or data.

The target server manages the symbol table and handles all communications with the remote target, dispatching function calls and sending their results back as needed. (The symbol table itself resides entirely on the host, although the addresses it contains refer to the target system.)

The target agent is the only component that runs on the target; it is a minimal monitor program that mediates access to target memory and other facilities.

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. 236, 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
7.2 Using the Shell

The shell reads lines of input from an input stream, parses and evaluates each line, and writes the result of the evaluation to an output stream. With its default C-expression interpreter, the shell accepts the same expression syntax as a C compiler with only a few variations.

The following sections explain how to start and stop the shell and provide examples illustrating some typical uses of the shell’s C interpreter. In the examples, the default shell prompt for interactive input in C is “->”. User input is shown in bold face and shell responses are shown in a plain roman face.

7.2.1 Starting and Stopping the Tornado Shell

There are 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.296).
- From the Tools menu: Click on Shell. The dialog box shown in Figure 7-2 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 7-3 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.

- ability to call user routines
- ability to create and examine variables symbolically
- ability to examine and modify memory
- exception trapping
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.227).

### 7.2.2 Downloading From the Shell

One of the most useful shell features for interactive development is the dynamic linker. With the shell command `ld()`, you can download and link new portions of the application.

```bash
-> ld < /home/moduleDir/module.o
```

Because the linking is dynamic, you only have to rebuild the particular piece you are working on, not the entire application. Download can be cancelled with `CTRL+C` or by clicking `Cancel` in the load progress indicator window. The dynamic linker is discussed further in *VxWorks Programmer’s Guide: Configuration and Build*.

The WTX error `(0x10197) EXCHANGE_TIMEOUT` may occur when a WTX request keeps the target server busy longer than 30 seconds (default timeout). This may happen when loading a large object module. A Tcl procedure is available to change the default timeout. From WindSh use `wtxTimeout sec` where `sec` is the number of seconds before timeout:

```bash
-> ?wtxTimeout 120
```

### 7.2.3 Shell Features

The shell provides many features which simplify your development and testing activities. These include command name and path completion, command and function synopsis printing, automatic data conversion, calculation with most C operators and variables, and help on all shell and VxWorks functions.

**I/O Redirection**

Developers often call routines that display data on standard output or accept data from standard input. By default the standard output and input streams are directed to the same window as the Tornado shell. For example, in a default configuration of Tornado, invoking `printf()` from the shell window gives the following display:
-> printf("Hello World\n")
Hello World!
value = i3 = 0xd
->

This behavior can be dynamically modified using the Tcl procedure \texttt{shConfig} as follows:

-> ?shConfig SH_GET_TASK_IO off
->
-> printf("Hello World!\n")
value = i3 = 0xd
->

The shell reports the \texttt{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 \texttt{shConfig}. If you issue the command without an argument, all parameters are listed. Use an argument to list only one parameter.

-> ?shConfig SH_GET_TASK_IO
SH_GET_TASK_IO = off

For more information on \texttt{shConfig}, see \textit{WindSh Environment Variables}, p. 227.

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 \texttt{-C -redirectShell}.

\textbf{Target Symbol and Path Completion}

Start to type any target symbol name or any existing directory name and then type \texttt{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 \texttt{CTRL+D} again until the path or symbol is complete.
**Synopsis Printing**

Once you have typed the complete function name, typing CTRL+D again prints the function synopsis and then reprints the function name ready for your input:

```
-> _taskIdDefault [CTRL+D]
taskIdDefault() - set the default task ID (WindSh)

int taskIdDefault
{
    int tid /* user-supplied task ID; if 0, return default */
}

-> _taskIdDefault
```

If the routine exists on both host and target, the WindSh synopsis is printed. To print the target synopsis of a function add the meta-character @ before the function name.

You can extend the synopsis printing function to include your own routines. To do this, follow these steps:

1. Create the files that include the new routines following Wind River Coding Conventions. (See the VxWorks Programmer’s Guide: Coding Conventions.)
2. Include these files in your project. (See Creating, Adding, and Removing Application Files, p.99.)
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.

Typing CTRL+W without a typing a space after the function name launches the HTML help tool if the function name is unique. If not, CTRL+W acts as CTRL+D and returns a list of functions whose names begin with the string you entered.

**Data Conversion**

The shell prints all integers and characters in both decimal and hexadecimal, and if possible, as a character constant or a symbolic address and offset.

```
-> 68
value = 68 = 0x44 = 'D'

-> 0xf5de
value = 62942 = 0xf5de = _init + 0x52

-> 's'
value = 115 = 0x73 = 's'
```

**Data Calculation**

Almost all C operators can be used for data calculation. Use “(" and ")” to force order of precedence.

```
-> (14 * 9) / 3
value = 42 = 0x2a = '*'

-> (0x1355 << 3) & 0x0f0f
value = 2568 = 0xa08

-> 4.3 * 5
value = 21.5
```

**Calculations With Variables**

```
-> (j + k) * 3
value = ...

-> *(j + 8 * k)
(...address (j + 8*k)...): value = ...
```
new symbol "x" added to symbol table
address = ...
value = ...

new symbol "f" added to symbol table
f = (...address of f...): value = 2.82

Variable f gets an 8-byte floating point value.

new symbol "ddd" added to symbol table.
ddd = 0xba0e2c: value = 5.2

new symbol "eee" added to symbol table.
eee = 0xba0e24: value = 10.5

new symbol "fff" added to symbol table.
fff = 0xba0e1c: value = 15.7

WindSh Environment Variables

WindSh allows you to change the behavior of a particular shell session by setting the environment variables listed in Table 7-1. The Tcl procedure shConfig allows you to display and set how I/O redirection, C++ constructors and destructors, loading, and the load path are defined and handled by the shell.

<table>
<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>
</tbody>
</table>
Because \texttt{shConfig} is a Tcl procedure, use the \texttt{?} to move from the C interpreter to the Tcl interpreter. (See 7.7.2 \textit{Tcl: Calling under C Control}, p.273.)

**Example 7-1** Using \texttt{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
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{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 \texttt{LD_SEND_MODULES} if “off”, the target server must be able to see the module to load it.</td>
</tr>
<tr>
<td>\texttt{LD_PATH}</td>
<td>Sets the search path for modules using the separator “;”. When a \texttt{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>\texttt{LD_COMMON_MATCH_ALL}</td>
<td>Sets the loader behavior for common symbols. If it is set to \texttt{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 \texttt{off}, the loader does not try to find an existing symbol. It creates an entry for each common symbol.</td>
</tr>
<tr>
<td>\texttt{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>
7.2.4 Invoking Built-In Shell Routines

Some of the commands (or routines) that you can execute from the shell are built into the host shell, rather than running as function calls on the target. These facilities parallel interactive utilities that can be linked into VxWorks itself. By using the host commands, you minimize the impact on both target memory and performance.

The following sections give summaries of the Tornado WindSh commands. For more detailed reference information, see the `windsh` reference entry (either online, or in .).

⚠️ **WARNING:** Most of the shell commands correspond to similar routines that can be linked into VxWorks for use with the target-resident version of the shell (VxWorks Programmer’s Guide: Target Shell). However, the target-resident routines differ in some details. For reference information on a shell command, be sure to consult the `windsh` entry in the online Tornado Tools Reference. Although there is usually an entry with the same name in the VxWorks API Reference, it describes a related target routine, not the shell command.

### Task Management

Table 7-2 summarizes the WindSh commands that manage VxWorks tasks.

<table>
<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>
<tr>
<td>tr( )</td>
<td>Resume a suspended task.</td>
</tr>
<tr>
<td>ts( )</td>
<td>Suspend a task.</td>
</tr>
<tr>
<td>td( )</td>
<td>Delete a task.</td>
</tr>
<tr>
<td>period()</td>
<td>Spawn a task to call a function periodically.</td>
</tr>
<tr>
<td>repeat()</td>
<td>Spawn a task to call a function repeatedly.</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 7.6 Object Module Load Path, p.270.) These tasks may be controlled like any other tasks on the target; for example, you can suspend or delete them with `ts()` or `td()` respectively.

### Task Information

Table 7-3 summarizes the WindSh commands that report task information.

<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>
<tr>
<td>ti()</td>
<td>Display task information. This command gives all the information contained in a task’s TCB. This includes everything shown for that task by an i() command, plus all the task’s registers, and the links in the TCB chain. If <code>task</code> is 0 (or the argument is omitted), the current task is reported on.</td>
</tr>
<tr>
<td>w()</td>
<td>Print a summary of each task’s pending information, task by task. This routine calls <code>taskWaitShow()</code> in quiet mode on all tasks in the system, or a specified task if the argument is given.</td>
</tr>
</tbody>
</table>
The \texttt{i()} command is commonly used to get a quick report on target activity. (To see this information periodically, use the Tornado browser; see \textit{9. Browser}). If nothing seems to be happening, \texttt{i()} is often a good place to start investigating. To display summary information about all running tasks:

\begin{verbatim}
- > i

NAME ENTRY TID PRI STATUS PC SP ERRNO DELAY
--------- ----------- -------- --- -------- ------- -------- ------- -----
tExcTask _excTask 3ad290 0 PEND 4df10 3ad0c0 0 0
tLogTask _logTask 3aa918 0 PEND 4df10 3aa748 0 0
tWdbTask 0x41288 3870f0 3 READY 23ff4 386d78 3d0004 0
tNetTask _netTask 3a59c0 50 READY 24200 3a5730 0 0
tFtpdTask _ftpdTask 3a2c18 55 PEND 23b28 3a2938 0 0
value = 0 = 0 x 0
\end{verbatim}

The \texttt{w()} and \texttt{tw()} commands allow you to see what object a VxWorks task is pending on. \texttt{w()} displays summary information for all tasks, while \texttt{tw()} displays object information for a specific task. Note that the \texttt{OBJ_NAME} field is used only for objects that have a symbolic name associated with the address of their structure.

\begin{verbatim}
- > w

NAME ENTRY TID STATUS DELAY OBJ_TYPE OBJ_ID OBJ_NAME
---------- ---------- ----------------- ----- ---------- -------- ----------
tExcTask _excTask 3d9e3c PEND 0 MSG_Q(R) 3d9ff4 N/A
tLogTask _logTask 3d7510 PEND 0 MSG_Q(R) 3d76c8 N/A
tWdbTask _wdbCmdLoo 36dde4 READY 0 0
NetTask _netTask 3a43d0 READY 0 0
tFtpdTask _ftpdTask 3a2c18 READY 0 0
value = 0 = 0 x 0
\end{verbatim}

The \texttt{w()} and \texttt{tw()} commands allow you to see what object a VxWorks task is pending on. \texttt{w()} displays summary information for all tasks, while \texttt{tw()} displays object information for a specific task. Note that the \texttt{OBJ_NAME} field is used only for objects that have a symbolic name associated with the address of their structure.
Tornado 2.2
User’s Guide

value = 0 = 0x0
->
tw ul

<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>ul</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>ul</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>ul</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>ul</td>
<td>_task3B</td>
<td>358ccc</td>
<td>PEND</td>
<td>0</td>
<td>MSG_S(S)</td>
<td>8cf1c</td>
<td>_myMsgQ</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
--------  ----------  ----------
0x36cc2c  0         0x36e464
0x367c54  0         0x36e47c

value = 0 = 0x0
->

System Information

Table 7-4 shows the WindSh commands that display information from the symbol table, from the target system, and from the shell itself.

### Table 7-4 WindSh Commands for System Information

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devs()</td>
<td>List all devices known on the target system.</td>
</tr>
<tr>
<td>lknup()</td>
<td>List symbols from symbol table.</td>
</tr>
<tr>
<td>lknAddr()</td>
<td>List symbols whose values are near a specified value.</td>
</tr>
<tr>
<td>d()</td>
<td>Display target memory. You can specify a starting address, size of memory units, and number of units to display.</td>
</tr>
<tr>
<td>l()</td>
<td>Disassemble and display a specified number of instructions.</td>
</tr>
<tr>
<td>printErrno()</td>
<td>Describe the most recent error status value.</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:

```
-> lkup "dsm"
_dsmData 0x00049d08 text (vxWorks)
_dsmNBytes 0x00049d76 text (vxWorks)
_dsmInst 0x00049d28 text (vxWorks)
mydsm 0x003c6510 bss (vxWorks)
```

Case is significant, but position is not (`mydsm` is shown, but `myDsm` would not be). To explicitly write a search that would match either `mydsm` or `myDsm`, you could write the following:

```
-> lkup "[dD]sm"
```

Regular-expression searches of the symbol table can be as simple or elaborate as required. For example, the following simple regular expression displays the names of three internal VxWorks semaphore functions:

```
-> lkup "sem.Take"
_semBTake 0x0002aeec text (vxWorks)
_semCTake 0x0002b268 text (vxWorks)
_semMTake 0x0002bc48 text (vxWorks)
value = 0 = 0x0
```
Another information command is a symbolic disassembler, l(). The command syntax is:

l [adr, n]]

This command lists n disassembled instructions, starting at adr. If n is 0 or not given, the n from a previous l() or the default value (10) is used. If adr is 0, l() starts from where the previous l() stopped, or from where an exception occurred (if there was an exception trap or a breakpoint since the last l() command).

The disassembler uses any symbols that are in the symbol table. If an instruction whose address corresponds to a symbol is disassembled (the beginning of a routine, for instance), the symbol is shown as a label in the address field. Symbols are also used in the operand field. The following is an example of disassembled code for an MC680x0 target:

```
-> l printf
 printf

00033bce 4856 PEA (A6)
00033bd0 2c4f MOVEA .L A7,A6
00033bd2 4878 0001 PEA 0x1
00033bd6 4879 0003 460e PEA _fioFormatV + 0x780
00033bdc 486e 000c PEA (0xc,A6)
00033be0 2f2e 0008 MOVE .L (0x8,A6),-(A7)
00033be4 6100 02a8 BSR _fioFormatV
00033be7 4e5e UNLK A6
```

This example shows the printf() routine. The routine does a LINK, then pushes the value of std_out onto the stack and calls the routine fioFormatV(). Notice that symbols defined in C (routine and variable names) are prefixed with an underbar (_) by the compiler.

Perhaps the most frequently used system information command is d(), which displays a block of memory starting at the address which is passed to it as a parameter. As with any other routine that requires an address, the starting address can be a number, the name of a variable or routine, or the result of an expression.

Several examples of variations on d() appear below.

Display starting at address 1000 decimal:

```
-> d (1000)
```

Display starting at 1000 hex:

```
-> d 0x1000
```
Display starting at the address contained in the variable dog:

-> d dog

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

-> d &dog

Display starting at an offset from the value of dog:

-> d dog + 100

Display starting at the result of a function call:

-> d func(dog)

Display the code of func() as a simple hex memory dump:

-> d func

When you use cd() in the host shell, you are changing the working directory on the host. It does not change the directory on the target. WindSh has no knowledge of the target file system. Thus if you mount a drive on the target from the host shell and try to cd() to it, you see the following:

-> cd "/ata0/"
  couldn’t change working directory to "\ata0": no such file or directory
  value = -1 = 0xffffffff

However, the result is different if you execute cd() and ls() on the target by prefixing the commands with @:

-> @cd "/ata0/"
  value = 0 = 0x0

-> @ls
  IO.SYS
  MSDOS.SYS
  DRVSPACE.BIN

@cd and @ls only work if you have the component INCLUDE_DISK_UTIL included in your target image.

The above also applies if you wish to use the target server file system (TSFS) from WindSh: cd "/tgtsvr" does not work but @cd "/tgtsvr" does. To use TSFS you must
have the TSFS component, INCLUDE_WDB_TSFS, installed in VxWorks and start
the target server with the '-R dirName' or '-R dirName -RW' option.

System Modification and Debugging

Developers often need to change the state of the target, whether to run a new
version of some software module, to patch memory, or simply to single-step a
program. Table 7-5 summarizes the WindSh commands of this type.

Table 7-5 WindSh Commands for System Modification and Debugging

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
</table>
| ld()   | Load an object module into target memory and link it dynamically into
        | the run-time. |
| unld() | Remove a dynamically-linked object module from target memory, and
        | free the storage it occupied. |
| m()    | Modify memory in width (byte, short, or long) starting at adr. The m() command displays successive words in memory on the terminal; you can change each word by typing a new hex value, leave the word unchanged and continue by typing ENTER, or return to the shell by typing a dot (.). |
| mRegs()| Modify register values for a particular task. |
| b()    | Set or display breakpoints, in a specified task or in all tasks. |
| bh()   | Set a hardware breakpoint. |
| s()    | Step a program to the next instruction. |
| so()   | Single-step, but step over a subroutine. |
| c()    | Continue from a breakpoint. |
| cret() | Continue until the current subroutine returns. |
| bdall()| Delete all breakpoints. |
| bd()   | Delete a breakpoint. |
| reboot()| Return target control to the target boot ROMs, then reset the target server and reattach the shell. |
Table 7-5  **WindSh Commands for System Modification and Debugging (Continued)**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bootChange()</td>
<td>Modify the saved values of boot parameters (see 2.5.4 Description of Boot Parameters, p.45).</td>
</tr>
<tr>
<td>sysSuspend()</td>
<td>If supported by the target-agent configuration, enter system mode. See 7.2.7 Using the Shell for System Mode Debugging, p.242.</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>

The `m()` command provides an interactive way of manipulating target memory. The remaining commands in this group are for breakpoints and single-stepping. You can set a breakpoint at any instruction. When that instruction is executed by an eligible task (as specified with the `b()` command), the task that was executing on the target suspends, and a message appears at the shell. At this point, you can examine the task’s registers, do a task trace, and so on. The task can then be deleted, continued, or single-stepped.

If a routine called from the shell encounters a breakpoint, it suspends just as any other routine would, but in order to allow you to regain control of the shell, such suspended routines are treated in the shell as though they had returned 0. The suspended routine is nevertheless available for your inspection.

When you use `s()` to single-step a task, the task executes one machine instruction, then suspends again. The shell display shows all the task registers and the next instruction to be executed by the task.

You can use the `bh()` command to set hardware breakpoints at any instruction or data element. Instruction hardware breakpoints can be useful to debug code running in ROM or Flash EPROM. Data hardware breakpoints (watchpoints) are useful if you want to stop when your program accesses a specific address. Hardware breakpoints are available on some BSPs; see your BSP documentation to determine if they are supported for your BSP. The arguments of the `bh()` command are architecture specific. For more information, run the `help()` command. The number of hardware breakpoints you can set is limited by the hardware; if you exceed the maximum number, you will receive an error.
Certain WindSh commands are intended specifically for work with C++ applications. Table 7-6 summarizes these commands. For more discussion of these shell commands, see *VxWorks Programmer’s Guide: C++ Development*.

**Table 7-6  WindSh Commands for C++ Development**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cplusCtors()</td>
<td>Call static constructors manually.</td>
</tr>
<tr>
<td>cplusDtors()</td>
<td>Call static destructors manually.</td>
</tr>
<tr>
<td>cplusStratShow()</td>
<td>Report on whether current constructor/destructor strategy is manual or automatic.</td>
</tr>
<tr>
<td>cplusXtorSet()</td>
<td>Set constructor/destructor strategy.</td>
</tr>
</tbody>
</table>

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.227.

**Object Display**

Table 7-7 summarizes the WindSh commands that display VxWorks objects. The browser provides displays that are analogous to the output of many of these routines, except that browser windows can update their contents periodically; see *9. Browser*.

**Table 7-7  WindSh Commands for Object Display**

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>show()</td>
<td>Print information on a specified object in the shell window.</td>
</tr>
<tr>
<td>browse()</td>
<td>Display a specified object in the Tornado browser.</td>
</tr>
<tr>
<td>classShow()</td>
<td>Show information about a class of VxWorks kernel objects. List available classes with:</td>
</tr>
<tr>
<td></td>
<td><code>-&gt; lookup &quot;ClassId&quot;</code></td>
</tr>
<tr>
<td>taskShow()</td>
<td>Display information from a task’s TCB.</td>
</tr>
</tbody>
</table>
Table 7-7  WindSh Commands for Object Display (Continued)

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskCreateHookShow()</td>
<td>Show the list of task create routines.</td>
</tr>
<tr>
<td>taskDeleteHookShow()</td>
<td>Show the list of task delete routines.</td>
</tr>
<tr>
<td>taskRegShow()</td>
<td>Display the contents of a task's registers.</td>
</tr>
<tr>
<td>taskSwitchHookShow()</td>
<td>Show the list of task switch routines.</td>
</tr>
<tr>
<td>taskWaitShow()</td>
<td>Show information about the object a task is pended on. Note that taskWaitShow() cannot give object IDs for POSIX semaphores or message queues.</td>
</tr>
<tr>
<td>semShow()</td>
<td>Show information about a semaphore.</td>
</tr>
<tr>
<td>semPxShow()</td>
<td>Show information about a POSIX semaphore.</td>
</tr>
<tr>
<td>wdShow()</td>
<td>Show information about a watchdog timer.</td>
</tr>
<tr>
<td>msgQShow()</td>
<td>Show information about a message queue.</td>
</tr>
<tr>
<td>mqPxShow()</td>
<td>Show information about a POSIX message queue.</td>
</tr>
<tr>
<td>iosDrvShow()</td>
<td>Display a list of system drivers.</td>
</tr>
<tr>
<td>iosDevShow()</td>
<td>Display the list of devices in the system.</td>
</tr>
<tr>
<td>iosFdShow()</td>
<td>Display a list of file descriptor names in the system.</td>
</tr>
<tr>
<td>memPartShow()</td>
<td>Show partition blocks and statistics.</td>
</tr>
<tr>
<td>memShow()</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>smMemShow()</td>
<td>Display the amount of free space and statistics on memory-block allocation for the shared-memory system partition.</td>
</tr>
<tr>
<td>smMemPartShow()</td>
<td>Display the amount of free space and statistics on memory-block allocation for a specified shared-memory partition.</td>
</tr>
<tr>
<td>moduleShow()</td>
<td>Show the current status for all the loaded modules.</td>
</tr>
</tbody>
</table>
Table 7-8 summarizes the WindSh commands that display information about the VxWorks network.

Table 7-8  WindSh Commands for Network Status Display

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

7.2.5 Running Target Routines from the Shell

All target routines are available from WindSh. This includes both VxWorks routines and your application routines. Thus the shell provides a powerful tool for testing and debugging your applications using all the host resources while having minimal impact on how the target performs and how the application behaves.

Invocations of VxWorks Subroutines

```plaintext
-> taskSpawn ("tmyTask", 10, 0, 1000, myTask, fd1, 300)
value = ...

-> fd = open ("file", 0, 0)
new symbol "fd" added to symbol table
fd = (...address of fd...): value = ...
```

Invocations of Application Subroutines

```plaintext
-> testFunc (123)
value = ...

-> myValue = myFunc (1, &val, testFunc (123))
myValue = (...address of myValue...): value = ...

-> myDouble = (double ()) myFuncWhichReturnsADouble (x)
myDouble = (...address of myDouble...): value = ...
```

For situations where the result of a routine is something other than a 4-byte integer, see Function Calls, p.251.
7.2.6 Rebooting from the Shell

In an interactive real-time development session, it is sometimes convenient to restart everything to make sure the target is in a known state. WindSh provides the `reboot()` command or 
\texttt{CTRL+SHIFT+X} to make this easy.

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

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

\[
\text{-} > \textbf{reboot} \\
\text{Rebooting...} \\
\]

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 \texttt{-Bt} (timeout) and \texttt{-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 the online \textit{Tornado Tools Reference}.

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:

\[
\text{Target connection has been lost. Restarting shell...} \\
\text{Waiting to attach to target server...} \\
\]

5. When WindSh establishes contact with the new target server, it displays the Tornado shell logo and awaits your input.

\[\textbf{CAUTION:}\] If the target server timeout (\texttt{-Bt}) and retry count (\texttt{-Br}) are too low for your target and your connection method, the new target server may abandon execution before the target finishes rebooting. The default timeout is one second, and the default retry count is three; thus, by default the target server waits three seconds for the target to reboot. If the shell does not restart in a reasonably short time after a `reboot()`, try starting a new target server manually.

7.2.7 Using the Shell for System Mode Debugging

The bulk of this chapter discusses the shell in its most frequent style of use: attached to a normally running VxWorks system, through a target agent running
in task mode. You can also use the shell with a system-mode agent. Entering system mode stops the entire target system: all tasks, the kernel, and all ISRs. Similarly, breakpoints affect all tasks.

⚠️ **CAUTION:** When you use system mode debugging, you cannot execute expressions that call target-resident routines. You must use `sp()` to spawn a task with the target-resident routine as the entry point argument. A newly-spawned task will not execute until you allow the kernel to run long enough to schedule that task.

Depending on how the target agent is configured, you may be able to switch between system mode and task mode; see 4.7 Configuring the Target-Host Communication Interface, p.138. When the agent supports mode switching, the following WindSh commands control system mode:

- `sysSuspend()`: Enter system mode and stop the target system.
- `sysResume()`: Return to task mode and resume execution of the target system.

The following commands are to determine the state of the system and the agent:

- `agentModeShow()`: Show the agent mode (`system` or `task`).
- `sysStatusShow()`: Show the system context status (`suspended` or `running`).

The following shell commands behave differently in system mode:

- `b()`: Set a system-wide breakpoint; the system stops when this breakpoint is encountered by any task, or the kernel, or an ISR.
- `c()`: Resume execution of the entire system (but remain in system mode).

⚠️ **WARNING:** If you are running either CrossWind or Look! you must not use `c()` from the shell to continue; instead continue from the debugger itself. Using `c()` from the shell when the debugger is running will confuse the debugger.

- `i()`: Display the state of the system context and the mode of the agent.
s()

Single-step the entire system.

sp()

Add a task to the execution queue. The task does not begin to execute until you continue the kernel or step through the task scheduler.

The following example shows how to use system mode debugging to debug a system interrupt.

Example 7-2 System-Mode Debugging

In this case, \texttt{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 \texttt{i()} or \texttt{sysStatusShow()}.

\begin{verbatim}
-> sysSuspend
value = 0 = 0x0
-> i

\begin{tabular}{cccccccc}
\hline
NAME & ENTRY & TID & PRI & STATUS & PC & SP & ERRNO & DELAY \\
\hline
_tExcTask & _excTask & 3e8f98 & 0 & PEND & 47982 & 3e8ef4 & 0 & 0 \\
_tLogTask & _logTask & 3e6670 & 0 & PEND & 47982 & 3e65c8 & 0 & 0 \\
_tWdbTask & & 0x3f024 & 3 & PEND & 405ac & 398d50 & 30067 & 0 \\
_tNetTask & _netTask & 3b39e0 & 50 & PEND & 405ac & 3b3988 & 0 & 0 \\
\hline
\end{tabular}

Agent mode : Extern
System context : Suspended
value = 0 = 0x0
-> sysStatusShow
System context is suspended
value = 0 = 0x0

Next, set the system mode breakpoint on the entry point of the interrupt handler you want to debug. Since the target agent is running in system mode, the breakpoint will automatically be a system mode breakpoint, which you can confirm with the \texttt{b()} command. Resume the system using \texttt{c()} and wait for it to enter the interrupt handler and hit the breakpoint.

\begin{verbatim}
-> b usrClock
value = 0 = 0x0
-> b
0x0002d9a: _usrClock               Task: SYSTEM Count: 0
value = 0 = 0x0
-> c
value = 0 = 0x0
->
Break at 0x0002d9a: _usrClock               Task: SYSTEM
\end{verbatim}
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
```

```
NAME ENTRY TID PRI STATUS PC SP ERRNO DELAY
--------- ---------- -------- ----- ------- ------- ------- ----- ----- 
tExcTask эксTask 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)
3f2bc _logTask +18 : _msgQReceive (3e62e4, 3e60dc, 20, ffffffff)
27e64 _msgQReceive +1ba: _qJobGet ([3e62e8, ffffffff, 0, 0, 0])
value = 0 = 0x0
-> 1

_1srClock
00022d9a  4856                     PEA         (A6)
00022d9c  2c4f                     MOVEA .L  A7,A6
00022d9e  61ff 0002 3d8c           BSR         _tickAnnounce
00022da4  4e5e                     UNLK        A6
00022da6  4e75                     RTS
00022da8  352e 3400                MOVE .W  (0x3400,A6),-(A2)
00022db0  3234 2031                MOVE .W  (0x2031,A4,D2.W*1),D1
00022db4  3939 382c 2031            MOVE .W  0x382c2031,-(A4)
00022dba  343a 3337                MOVE .W  (0x3337,PC),D2
value = 0 = 0x0
```

```
-> s
D0 = 3e D1 = 3700 D2 = 3000 D3 = 3b09dc
D4 = 0 D5 = 0 D6 = 0 D7 = 0
A0 = 230b8 A1 = 3b3318 A2 = 3b3324 A3 = 7e094
A4 = 38a7c0 A5 = 0 A6/fp = bc8b0 A7/sp = bc8b4
SR = 2604 PC = 230ba
value = 0 = 0x0
```

```
Return to task mode and confirm that return by calling i():
```
```
-> 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:

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

The application breaks on `address` when the instruction at `address` is executed.

### 7.2.8 Interrupting a Shell Command

Occasionally it is desirable to abort the shell’s evaluation of a statement. For example, an invoked routine may loop excessively, suspend, or wait on a semaphore. This may happen as the result of errors in arguments specified in the invocation, errors in the implementation of the routine itself, or simply oversight as to the consequences of calling the routine.

To regain control of the shell in such cases, press the interrupt character on the keyboard, usually `CTRL+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.

**CAUTION:** `CTRL+BREAK` and `CTRL+C` do not interrupt non-blocking functions. If the task transmitting the break request is of lower priority than the task to be interrupted, the request is not conveyed until the original task completes.

Pressing `CTRL+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.

### 7.3 The Shell C-Expression Interpreter

The C-expression interpreter is the most common command interface to the Tornado shell. This interpreter can evaluate almost any C expression interactively in the context of the attached target. This includes the ability to use variables and functions whose names are defined in the symbol table. Any command you type is interpreted as a C expression. The shell evaluates that expression and, if the expression so specifies, assigns the result to a variable.

#### 7.3.1 Data Types

The most significant difference between the shell C-expression interpreter and a C compiler lies in the way that they handle data types. The shell does not accept any C declaration statements, and no data-type information is available in the symbol table. Instead, an expression’s type is determined by the types of its terms.

Unless you use explicit type-casting, the shell makes the following assumptions about data types:
In an assignment statement, the type of the left hand side is determined by the type of the right hand side.

- If floating-point numbers and integers both appear in an arithmetic expression, the resulting type is a floating-point number.

- Data types are assigned to various elements as shown in Table 7-9.

**Table 7-9  Shell C Interpreter Data-Type Assumptions**

<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.251. Table 7-10 shows the various data types available in the shell C interpreter, with examples of how they can be set and referenced.

**Table 7-10  Data Types in the Shell C Interpreter**

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
<th>Set Variable</th>
<th>Display Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>4</td>
<td>( x = 99 )</td>
<td>( x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x = ) (int) 99</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>( x = 33 )</td>
<td>( x )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x = ) (long) 33</td>
<td>(long) x</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>( x = ) (short) 20</td>
<td>(short) x</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>( x = 'A' )</td>
<td>(char) x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x = ) (char) 65</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x = ) (char) 0x41</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>( x = 11.2 )</td>
<td>(double) x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( x = ) (double) 11.2</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>( x = ) (float) 5.42</td>
<td>(float) x</td>
</tr>
</tbody>
</table>
Strings, or character arrays, are not treated as separate types in the shell C interpreter. To declare a string, set a variable to a string value.\(^3\) For example:

```
-> ss = "shoe bee doo"
```

The variable `ss` is a pointer to the string `shoe bee doo`. To display `ss`, enter:

```
-> d ss
```

The `d()` command displays memory where `ss` is pointing.\(^4\) You can also use `printf()` to display strings.

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

```
*(70000 + 3 * 16)
```

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

### 7.3.2 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.

### 7.3.3 Expressions

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

---

3. Memory allocated for string constants is never freed by the shell. See 7.3.7 Strings, p. 256 for more information.
4. `d()` is one of the WindSh commands, implemented in Tcl and executing on the host.
Literals

The shell interprets the literals in Table 7-11 in the same way as the C compiler, with one addition: the shell also allows hex numbers to be preceded by $ instead of 0x.

<table>
<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.

Some C compilers prefix user-defined identifiers with an underbar, so that myVar is actually in the symbol table as _myVar. In this case, the identifier can be entered either way to the shell—the shell searches the symbol table for a match either with or without a prefixed underbar.

You can also access data in memory that does not have a symbolic name in the symbol table, as long as you know its address. To do this, apply the C indirection operator “*” to a constant. For example, *0x10000 refers to the 4-byte integer value at memory address 10000 hex.
Operators

The shell interprets the operators in Table 7-12 in the same way as the C compiler.

Table 7-12
Operators in the Shell C Interpreter

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>+  -  *  /  unary -</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().

```plaintext
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:

```plaintext
4 + myFunc2 ( )
```

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

```plaintext
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. 253.

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:

```plaintext
0x10000 ( )
```

You can assign the address of a function to a variable and then dereference the variable to call the function as in the following example:

```plaintext
-> aaa=printf
-> (* aaa)("The clock speed is %d\n" ,sysClkRateGet())
```

### Subroutines as Commands

Both VxWorks and the Tornado shell itself provide routines that are meant to be called from the shell interactively. You can think of these routines as commands, rather than as subroutines, even though they can also be called with the same syntax as C subroutines (and those that run on the target are in fact subroutines). All the commands discussed in this chapter fall in this category. When you see the word
command, you can read subroutine, or vice versa, since their meaning here is identical.

Arguments to Commands

In practice, most statements input to the shell are function calls, often to invoke VxWorks facilities. To simplify this use of the shell, an important exception is allowed to the standard expression syntax required by C. When a function name is the very first item encountered in a shell statement, the parentheses surrounding the function’s arguments may be omitted. Thus the following shell statements are synonymous:

- `rename ("oldname", "newname")`
- `rename "oldname", "newname"

as are:

- `evtBufferAddress ( )`
- `evtBufferAddress`

However, note that if you wish to assign the result to a variable, the function call cannot be the first item in the shell statement—thus, the syntactic exception above does not apply. The following captures the address, not the return value, of `evtBufferAddress`:

- `value = evtBufferAddress`

Task References

Most VxWorks routines that take an argument representing a task require a task ID. However, when invoking routines interactively, specifying a task ID can be cumbersome since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, shell expressions can reference a task by either task ID or task name. The shell attempts to resolve a task argument to a task ID as follows: if no match is found in the symbol table for a task argument, the shell searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

By convention, task names are prefixed with a `u` for tasks started from the Tornado shell, and with a `t` for VxWorks tasks started from the target itself. In addition, tasks
started from a shell are prefixed by s1, s2, and so on to indicate which shell they were started from. This avoids name conflicts with entries in the symbol table. The names of system tasks and the default task names assigned when tasks are spawned use this convention. For example, tasks spawned with the shell command `sp()` in the first shell opened are given names such as s1u0 and s1u1. Tasks spawned with the second shell opened have names such as s2u0 and s2u1. You are urged to adopt a similar convention for tasks named in your applications.

7.3.4 The “Current” Task and Address

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

- A task hits a breakpoint or an exception trap. The current address is the address of the instruction that caused the break or exception.
- A task is single-stepped. The current address is the address of the next instruction to be executed.
- Any of the commands that use the current task or address are executed with a specific task parameter. The current address will be the address of the byte following the last byte that was displayed or disassembled.

7.3.5 Assignments

The shell C interpreter accepts assignment statements in the form:

```
addressExpression = expression
```

The left side of an expression must evaluate to an addressable entity; that is, a legal C value.

**Typing and Assignment**

The data type of the left side is determined by the type of the right side. If the right side does not contain any floating-point constants or noninteger type-casts, then the type of the left side will be an integer. The value of the right side of the assignment is put at the address provided by the left side. For example, the following assignment sets the 4-byte integer variable x to 0x1000:
The following assignment sets the 4-byte integer value at memory address 0x1000 to the current value of $x$:

$$\rightarrow *0x1000 = x$$

The following compound assignment adds 300 to the 4-byte integer variable $x$:

$$\rightarrow x += 300$$

The following adds 300 to the 4-byte integer at address 0x1000:

$$\rightarrow *0x1000 += 300$$

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:

$$\rightarrow fd = open("file", 0)$$

### 7.3.6 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. 262 below.
7.3.7 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 \( x \):

\[-\rightarrow x = "hello there"\]

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:

\[-\rightarrow printf ("hello there")\]

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.142.

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.

7.3.8 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. 236) 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"

### 7.3.9 Ambiguity of Arrays and Pointers

In a C expression, a nonsubscripted reference to an array has a special meaning, namely the address of the first element of the array. The shell, to be compatible, should use the address obtained from the symbol table as the value of such a reference, rather than the contents of memory at that address. Unfortunately, the information that the identifier is an array, like all data type information, is not available after compilation. For example, if a module contains the following:

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

you might be tempted to enter a shell expression like:

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:

2. `printf (&string)`

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

```c
char *string = "hello";
```

then to a compiler 1 would be correct and 2 would be wrong! This is especially confusing since C allows pointers to be subscripted exactly like arrays, so that the value of `string[0]` would be “h” in either of the above declarations.
The moral of the story is that array references and pointer references in shell expressions are different from their C counterparts. In particular, array references require an explicit application of the address operator &.

### 7.3.10 Pointer Arithmetic

While the C language treats pointer arithmetic specially, the shell C interpreter does not, because it treats all non-type-cast variables as 4-byte integers.

In the shell, pointer arithmetic is no different than integer arithmetic. Pointer arithmetic is valid, but it does not take into account the size of the data pointed to. Consider the following example:

\[ \text{-> } *(\text{myPtr} + 4) = 5 \]

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.

### 7.3.11 C Interpreter Limitations

Powerful though it is, the C interpreter in the shell is not a complete interpreter for the C language. The following C features are not present in the Tornado shell:

- **Control Structures**
  
  The shell interprets only C expressions (and comments). The shell does not support C control structures such as if, goto, and switch statements, or do, while, and for loops. Control structures are rarely needed during shell interaction. If you do come across a situation that requires a control structure, you can use the Tcl interface to the shell instead of using its C interpreter directly; see 7.7 Tcl: Shell Interpretation, p.271.

- **Compound or Derived Types**
  
  No compound types (struct or union types) or derived types (typedef) are recognized in the shell C interpreter. You can use CrossWind instead of the shell for interactive debugging, when you need to examine compound or derived types.
Macros

No C preprocessor macros (or any other preprocessor facilities) are available in the shell. CrossWind does not support preprocessor macros either, but indirect workarounds are available using either the shell or CrossWind. For constant macros, you can define variables in the shell with similar names to the macros. To avoid intrusion into the application symbol table, you can use CrossWind instead; in this case, use CrossWind convenience variables with names corresponding to the desired macros. In either case, you can automate the effort of defining any variables you need repeatedly, by using an initialization script.

For the first two problems (control structures, or display and manipulation of types that are not supported in the shell), you might also consider writing auxiliary subroutines to provide these services during development; you can call such subroutines at will from the shell, and later omit them from your final application.

7.3.12 C-Interpreter Primitives

Table 7-13 lists all the primitives (commands) built into WindSh. (For discussion of these primitives by function, see 7.2.4 Invoking Built-In Shell Routines, p. 229.) 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. 241.)

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>agentModeShow()</td>
<td>ipstatShow()</td>
<td>smMemPartShow()</td>
</tr>
<tr>
<td>b()</td>
<td>iStrict()</td>
<td>smMemShow()</td>
</tr>
<tr>
<td>bd()</td>
<td>l()</td>
<td>so()</td>
</tr>
<tr>
<td>bdall()</td>
<td>ld()</td>
<td>sp()</td>
</tr>
<tr>
<td>bh()</td>
<td>lkAddr()</td>
<td>sps()</td>
</tr>
<tr>
<td>bootChange()</td>
<td>lkup()</td>
<td>sysResume()</td>
</tr>
<tr>
<td>browse()</td>
<td>ls()</td>
<td>sysStatusShow()</td>
</tr>
<tr>
<td>c()</td>
<td>m()</td>
<td>sysSuspend()</td>
</tr>
<tr>
<td>cd()</td>
<td>memPartShow()</td>
<td>taskCreateHookShow()</td>
</tr>
</tbody>
</table>
7.3.13 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 7-14 lists special terminal characters frequently used for shell control in both situations. For more information on the use of these characters, see 7.5 Shell Line Editing, p.267 and 7.2.8 Interrupting a Shell Command, p.246.

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkStack()</td>
<td>memShow()</td>
<td>taskDeleteHookShow()</td>
</tr>
<tr>
<td>classShow()</td>
<td>moduleIdFigure()</td>
<td>taskIdDefault()</td>
</tr>
<tr>
<td>cplusCtors()</td>
<td>moduleShow()</td>
<td>taskIdFigure()</td>
</tr>
<tr>
<td>cplusDtors()</td>
<td>mqPxShow()</td>
<td>taskRegsShow()</td>
</tr>
<tr>
<td>cplusStratShow()</td>
<td>mRegs()</td>
<td>taskShow()</td>
</tr>
<tr>
<td>cplusXtorSet()</td>
<td>msgQShow()</td>
<td>taskSwitchHookShow()</td>
</tr>
<tr>
<td>cret()</td>
<td>period()</td>
<td>taskWaitShow()</td>
</tr>
<tr>
<td>d()</td>
<td>printErrno()</td>
<td>tcpstatShow()</td>
</tr>
<tr>
<td>devs()</td>
<td>printLogo()</td>
<td>td()</td>
</tr>
<tr>
<td>h()</td>
<td>pwd()</td>
<td>tftpInfoShow()</td>
</tr>
<tr>
<td>help()</td>
<td>quit()</td>
<td>t()</td>
</tr>
<tr>
<td>hostShow()</td>
<td>reboot()</td>
<td>tr()</td>
</tr>
<tr>
<td>i()</td>
<td>repeat()</td>
<td>ts()</td>
</tr>
<tr>
<td>icmpstatShow()</td>
<td>routestatShow()</td>
<td>tt()</td>
</tr>
<tr>
<td>ifShow()</td>
<td>s()</td>
<td>tw()</td>
</tr>
<tr>
<td>inetstatShow()</td>
<td>semPxShow()</td>
<td>udpstatShow()</td>
</tr>
<tr>
<td>intVecShow()</td>
<td>semShow()</td>
<td>unld()</td>
</tr>
<tr>
<td>iosDevShow()</td>
<td>shellHistory()</td>
<td>version()</td>
</tr>
<tr>
<td>iosDrvShow()</td>
<td>shellPromptSet()</td>
<td>w()</td>
</tr>
<tr>
<td>iosFdShow()</td>
<td>show()</td>
<td>wdShow()</td>
</tr>
</tbody>
</table>
7.3.14 Redirection in the C Interpreter

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

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

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

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

Ambiguity Between Redirection and C Operators

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

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

\[
\text{-} \rightarrow \ x; \ > \ y \\
\text{-} \rightarrow \ x \ > \ y; \\
\]

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:

\[
\text{-} \rightarrow \text{taskSpawn (...myFunc...) > output} \\
\]

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 .).

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 7.7 Tcl: Shell Interpretation, p.271.

7.4 C++ Interpretation

Tornado supports both C and C++ as development languages; see VxWorks Programmer's Guide: C++ Development for information about C++ development. Because C and C++ expressions are so similar, the WindSh C-expression interpreter supports many C++ expressions. The facilities explained in 7.3 The Shell C-Expression Interpreter, p.247 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 7.7.3 Tcl: Tornado Shell Initialization, p.274.
To exercise C++ facilities that are missing from the C-expression interpreter, you can compile and download routines that encapsulate the special C++ syntax. Fortunately, the Tornado dynamic linker makes this relatively painless.

### 7.4.1 Overloaded Function Names

If you have several C++ functions with the same name, distinguished by their argument lists, call any of them as usual with the name they share. When the shell detects the fact that several functions exist with the specified name, it lists them in an interactive dialogue, printing the matching functions’ signatures so that you can recall the different versions and make a choice among them.

You make your choice by entering the number of the desired function. If you make an invalid choice, the list is repeated and you are prompted to choose again. If you enter 0 (zero), the shell stops evaluating the current command and prints a message like the following, with \texttt{xxx} replaced by the function name you entered:

```
undefined symbol: xxx
```

This can be useful, for example, if you misspelled the function name and you want to abandon the interactive dialogue. However, because WindSh is an interpreter, portions of the expression may already have executed (perhaps with side effects) before you abandon execution in this way.

The following example shows how the support for overloaded names works. In this example, there are four versions of a function called \texttt{xmin()}. Each version of \texttt{xmin()} returns at least two arguments, but each version takes arguments of different types.

```
-> 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
3fe71a f22e 5438 0010 FCMP .D  (0x10,A6),F0
3fe720 f295 0008       FB     .W  #0x8f22e
3fe724 f22e 5400 0010 FMOVE .D  (0x10,A6),F0
3fe72a f277 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\texttt{\_\_\_\_xmin()} \texttt{(double,double)}, then the version that computes the minimum of two \texttt{double} values is selected. Next, the disassembler is invoked again, this time selecting the version that computes the minimum of two \texttt{int} values. Note that a different routine is disassembled in each case.

### 7.4.2 Automatic Name Demangling

Many shell debugging and system information functions display addresses symbolically (for example, the \texttt{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 \textit{name mangling} and can be a source of frustration when the mangled names are exposed to the developer.

To avoid this confusion, the debugging and system information routines in WindSh print C++ function names in a demangled representation. Whenever the shell prints an address symbolically, it checks whether the name has been mangled. If it has, the name is demangled (complete with the function’s class name, if any, and the type of each of the function’s arguments) and printed.
The following example shows the demangled output when `lkup()` displays the addresses of the `xmin()` functions mentioned in 7.4.1 Overloaded Function Names, p.265.

```
-> lkup "xmin"
  xmin(double,double)    0x003fe710 text         (templex.out)
  xmin(long,long)        0x003fe754 text         (templex.out)
  xmin(int,int)          0x003fe73a text         (templex.out)
  xmin(float,float)      0x003fe6ee text         (templex.out)
value = 0 = 0x0
```

### 7.5 Shell Line Editing

The WindSh front end provides a history mechanism similar to the UNIX Korn-shell history facility, including a built-in 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 7-15 are available in edit mode.

Some line-editing commands switch the line editor to insert mode until an ESC is typed (as in `vi`) or until an ENTER gives the line to one of the shell interpreters. ENTER always gives the line as input to the current shell interpreter, from either input or edit mode.

In input mode, the shell history command `h()` (described in System Information, p.232) displays up to 20 of the most recent commands typed to the shell; older commands are lost as new ones are entered. You can change the number of commands kept in history by running `h()` with a numeric argument. To locate a line entered previously, press ESC followed by one of the search commands listed in Table 7-15; you can then edit and execute the line with one of the commands from Table 7-15.

---

6. The WindSh Tcl-interpreter interface is described in 7.7 Tcl: Shell Interpretation, p.271.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>h [size]</code></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 7-15  **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>n</code> or <code>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>nr</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.
7.6 Object Module Load Path

In order to download an object module dynamically to the target, both WindSh and the target server must be able to locate the file. If path naming conventions are different between WindSh and the target server, the two systems may both have access to the file, but mounted with different path names. This situation arises often in environments where UNIX and Windows systems are networked together, because the path naming convention is different: the UNIX /usr/fred/applic.o may well correspond to the Windows n:\fred\applic.o. If you encounter this problem, check to be sure the LD_SEND_MODULES variable of shConfig is set to “on” or use the LD_PATH facility to tell the target server about the path known to the shell.

Example 7-3  Loading a Module: Alternate Path Names

```bash
- > 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.227.

⚠️ 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 7-16 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 C. Tcl.) In the interim, you can do a great deal of development work with the shell C interpreter alone.

To toggle between the Tcl interpreter and the C interpreter in the shell, type the single character `. The shell prompt changes to remind you of the interpreter state: the prompt `->` indicates the C interpreter is listening, and the prompt `tcl>` indicates the Tcl interpreter is listening.7 For example, in the following interaction we use the C interpreter to define a variable in the symbol table, then switch into the Tcl interpreter to define a similar Tcl variable in the shell itself, and finally switch back to the C interpreter:

```
-} hello=hi there
new symbol "hello" added to symbol table.
hello = 0x3616e8: value = 3544824 = 0x3616f8 = hello + 0x10
-} ?
tcl> set hello (hi there)
hi there
tcl> ?
-}
```

If you start `windsh` from the Windows command line, you can also use the option `-Tclmode` (or `-T`) to start with the Tcl interpreter rather than the C interpreter.

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.

### 7.7.1 Tcl: Controlling the Target

In the Tcl interpreter, you can create custom commands, or use Tcl control structures for repetitive tasks, while using the building blocks that allow the C interpreter and the WindSh commands to control the target remotely. These building blocks as a whole are called the `wtxtcl` procedures.

For example, `wtxMemRead` returns the contents of a block of target memory (given its starting address and length). That command in turn uses a special memory-block datatype designed to permit memory transfers without unnecessary Tcl data conversions. The following example uses `wtxMemRead`, together with the memory-block routine `memBlockWriteFile`, to write a Tcl procedure that dumps target memory to a host file. Because almost all the work is done on the host, this procedure works whether or not the target run-time environment contains I/O libraries or any networked access to the host file system.

```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 `wtxtcl` routines available in the Tornado shell, see the Tornado API Programmer’s Guide (or the online Tornado API Reference).

All of the commands defined for the C interpreter (7.2.4 Invoking Built-In Shell Routines, p.229) are also available, with a double-underscore prefix, from the Tcl level; for example, to call `i()` from the Tcl interpreter, run the Tcl procedure `__i`. However, in many cases, it is more convenient to call a `wtxtcl` routine instead, because the WindSh commands are designed to operate in the C-interpreter context. For example, you can call the dynamic linker using `ld` from the Tcl interpreter, but the argument that names the object module may not seem intuitive: it is the address of a string stored on the target. It is more convenient to call the underlying `wtxtcl` command. In the case of the dynamic linker, the underlying
Shell

wtxtcl command is `wtxObjModuleLoad`, which takes an ordinary Tcl string as its argument, as described in the online WTX Tcl API.

**Tcl: Calling Target Routines**

The `shParse` utility allows you to embed calls to the C interpreter in Tcl expressions; the most frequent application is to call a single target routine, with the arguments specified (and perhaps capture the result). For example, the following sends a logging message to your VxWorks target console:

```
tcl> shParse {logMsg("Greetings from Tcl\n")}
```

You can also use `shParse` to call WindSh commands more conveniently from the Tcl interpreter, rather than using their `wtxtcl` building blocks. For example, the following is a convenient way to spawn a task from Tcl, using the C-interpreter command `sp()`, if you do not remember the underlying `wtxtcl` command:

```
tcl> shParse {sp appTaskBegin}
task spawned: id = 25e388, name = u1
```

**Tcl: Passing Values to Target Routines**

Because `shParse` accepts a single, ordinary Tcl string as its argument, you can pass values from the Tcl interpreter to C subroutine calls simply by using Tcl facilities to concatenate the appropriate values into a C expression.

For example, a more realistic way of calling `logMsg()` from the Tcl interpreter would be to pass as its argument the value of a Tcl variable rather than a literal string. The following example evaluates a Tcl variable `tclLog` and inserts its value (with a newline appended) as the `logMsg()` argument:

```
tcl> shParse "logMsg("$tclLog\n")"
```

**7.7.2 Tcl: Calling under C Control**

To dip quickly into Tcl and return immediately to the C interpreter, you can type a single line of Tcl prefixed with the `?` character (rather than using `?` by itself to toggle into Tcl mode). For example:
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:

```c
-> ? set appTasks {appFrobStart appGetStart appPutStart ...}
appFrobStart appGetStart appPutStart ...
```

Then whenever you need to restart the whole list of tasks, you can use something like the following:

```c
-> ? foreach it $appTasks {shParse "sp($it)"}
task spawned: id = 25e388, name = u0
task spawned: id = 259368, name = u1
task spawned: id = 254348, name = u2
task spawned: id = 24f328, name = u3
```

### 7.7.3 Tcl: Tornado Shell Initialization

When you execute an instance of the Tornado shell, it begins by looking for a file called `.wind\windsh.tcl` in the directory specified by the `HOME` environment variable (if that environment variable is defined). In each of these directories, if the file exists, the shell reads and executes its contents as Tcl expressions before beginning to interact. You can use this file to automate any initialization steps you perform repeatedly.

You can also specify a Tcl expression to execute initially on the `windsh` command line, with the option `-e tclExpr`. For example, you can test an initialization file before saving it as `.wind\windsh.tcl` using this option, as follows:

```bash
C:\> windsh phobos -e "source c:\fred\tcltest"
```
Example 7-4  **Shell Initialization File**

This file causes I/O for target routines called in WindSh to be directed to the
target’s standard I/O rather than to WindSh. It changes the default C++ strategy
to automatic for this shell, sets a path for locating load modules, and causes
modules not to be copied to the target server.

```
# Redirect Task I/O to WindSh
shConfig SH_GET_TASK_IO off
# Set C++ strategy
shConfig LD_CALL_XTORS on
# Set Load Path
shConfig LD_PATH "/folk/jmichel/project/app;/folk/jmichel/project/test"
# Let the Target Server directly access the module
shConfig LD_SEND_MODULES off
```

### 7.8 The Shell Architecture

#### 7.8.1 Controlling the Target from the Host

Tornado integrates host and target resources so well that it creates the illusion of
executing entirely on the target itself. In reality, however, most interactions with
any Tornado tool exploit the resources of both host and target. For example,
Table 7-17 shows how the shell distributes the interpretation and execution of the
following simple expression:

```
-> dir = opendir("/myDev/myFile")
```

Parsing the expression is the activity that controls overall execution, and
dispatches the other execution activities. This takes place on the host, in the shell’s
C interpreter, and continues until the entire expression is evaluated and the shell
displays its result.

To avoid repetitive clutter, Table 7-17 omits the following important steps, which
must be carried out to link the activities in the three contexts (and two systems)
shown in each column of the table:

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

**Table 7-17 Interpreting: dir = opendir ("/myDev/myFile")**

<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 &quot;/myDev/myFile&quot;.</td>
<td>Allocate memory for the string; return address A.</td>
<td>Write &quot;/myDev/myFile&quot;; return address A.</td>
</tr>
<tr>
<td>Parse the name opendir.</td>
<td>Look up opendir; return address B.</td>
<td></td>
</tr>
<tr>
<td>Parse the function call opendir(A); wait for the result.</td>
<td>Spawn a task to run opendir() and signal result C when done.</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 D=C.</td>
<td>Allocate agent-pool memory for the value of dir.</td>
<td>Write the value of dir.</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.

### 7.8.2 Shell Components

The Tornado shell includes two interpreters, a common front end for command entry and history, and a back end that connects the shell to the global Tornado environment to communicate with the target server. Figure 7-4 illustrates these components:
The line-editing and command history facilities are designed to be unobtrusive, and support your access to the interpreters. 7.5 Shell Line Editing, p.267 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 7.7 Tcl: Shell Interpretation, p.271.

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

In daily use, the shell seems to be a seamless environment; but in fact, the characters you type in WindSh go through several layers of interpretation, as illustrated by Figure 7-5. First, input is examined for special editing keystrokes (described in 7.5 Shell Line Editing, p.267). Then as much interpretation as possible is done in WindSh itself. In particular, execution of any subroutine is first attempted in the shell itself; if a shell built-in (also called a primitive) with that name exists, the built-in runs without any further checking. Only when a subroutine call does not match any shell built-ins does WindSh call a target routine. See 7.2.4 Invoking Built-In Shell Routines, p.229 for more information. For a list of all WindSh primitives, see Table 7-13 List of WindSh Commands, p.259.

Figure 7-5  Layers of Interpretation in the Shell
8.1 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 8-1). A target server includes the host-resident target symbol table, and an object-module loader that inserts application modules into a running target system.

Figure 8-1  Target Server in Development Environment

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 Setting Up the Host, p. 16; and the wtxregd reference documentation in . .

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 . .

Before configuring and starting a new target, make sure that the host and target are connected properly (see 2.3 Setting Up the Default Target Hardware, p. 19).

8.2 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 8.2.1 Using the Configure Target Servers Dialog Box, p. 283).
- From the command line or from a batch file (see 8.2.3 Using the Command Line, p. 294).

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.

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.
8.2.1 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 8-2).

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 the target server will use to connect to the target.
- 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 \texttt{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 \textit{Miscellaneous}, p.293).

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 \texttt{tgtsvr} reference documentation (see \textit{Tornado API Reference: Tornado Tools Reference}).

\textbf{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:

\textbf{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.

\textbf{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.

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

### 8.2.2 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 drop-down 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.286 and 4.7 Configuring the Target-Host Communication Interface, p.138. |

**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.299.

**Figure 8-3 Authorizations Properties**
The Authorizations property list (Figure 8-3) 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.300.

Back End

The Back End property list is used to specify how a target server will communicate with a target (Figure 8-4).

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 8-1. Other back ends may be available separately; contact your nearest Wind River office (see the back cover).

<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>
</tbody>
</table>
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 8-4) 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 4.7 Configuring the Target-Host Communication Interface, p. 138).

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 8-2).

**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 8-5 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.

- **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.

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

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 4.4.3 Configuring VxWorks Components, p.116 and the reference entry for `symSyncLib`.

To use symbol and module synchronization, the `WIND_REGISTRY` environment variable must be set to a host name or an IP address that the VxWorks target can access. It cannot be left as the default value, `localhost`. Change `WIND_REGISTRY` by selecting the Environment tab in the Start>Settings>Control Panel>System dialog box.

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 8-6).
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.142 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 the online 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.50).
WARNING: To use the TSFS, you must include the WDB target server file system component when you build the VxWorks image. See 4.4 Creating a Custom VxWorks Image, p.109).

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.288).

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

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 8-8) direct virtual I/O to various locations:

Figure 8-8 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 8.3 Target Server Icon and Log Console Window, p.295) 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 Tornado API Reference: 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 8-9) allows you to log WDB and WTX requests.

Figure 8-9 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 8-10) provides for Tornado 1.0.1 tools compatibility, and for use of options that are not available through the Configure Target Servers dialog box.

**Figure 8-10** **Miscellaneous Configuration Properties**

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 the online 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.

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

**Target Memory Pool**

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

**Figure 8-11** Target Memory Pool Configuration Properties

8.2.3 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 entry).

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.

### 8.3 Target Server Icon and Log Console Window

When a target server is started, the target server icon is displayed in the Windows taskbar (Figure 8-12). 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.

The Log Console window displays target server log output (Figure 8-12). 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).
8.4 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 8.6 Managing a Target Server, p.298).

- Using the Shutdown button in the target server Log Console window (see 8.3 Target Server Icon and Log Console Window, p.295). 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 8.3 Target Server Icon and Log Console Window, p.295).

**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.

8.5 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 8.2 Configuring and Starting a Target Server, p.282.

*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 8-13; also see 2.2 Setting Up the Host, p.16). 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.
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 8-14 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 8.6 Managing a Target Server, p.298.
8.6 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 8-15.

Figure 8-15  Manage Target Servers Dialog Box

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 9.5 Target-Information Window, p.307). You can update the display with the ! button.

Once you select a target, you can select a command from the Select Action drop-down list box (Figure 8-15), 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.299 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

1. Strictly speaking, there is another layer of authorization defining who is meant by “any user.” The file c:\tornado\wind\userlock is a Tornado-wide authorization file, used as the default list of authorized users for any target server that does not define its own authorized-users file. This file has the same format described below for individual authorization files.
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 (9.5 Target-Information Window, p.307) and in the target-selection dialog for any Tornado tool (for example, Figure 8-14). 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 8.2.2 Target-Server Configuration Options, p.285. 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 8.6 Managing a Target Server, p.298). 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.285). 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.)
9

Browser

9.1 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.
9.2 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.296).
- From the Tools menu, click on Browser. The dialog box shown in Figure 9-1 appears, to allow you to select a target server from the Targets drop-down list.

9.3 Anatomy of the Browser Window

The top of the browser window, shown in Figure 9-2, 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 9.5 Target-Information Window, p.307.
- **Tasks**  
  See 9.6 Task-List Window, p.309.
- **Memory Usage**  
  See 9.7 Memory-Usage Window, p.310.
- **Object Information**  
  See 9.8 Object-Information Windows, p.311.
- **Stack Check**  
  See 9.11 The Stack-Check Window, p.321.
- **Module Information**  
  See 9.9 The Module-Information Window, p.318.
- **Spy Chart**  
  See 9.10 The Spy Window, p.320.
- **Vector Table**  
  See 9.12 The Vector Table Window, p.322. This selection is available only on supported architectures.

**Button Bar**

Buttons to give you fresh snapshots of your target, and to adjust browser parameters. 9.4 Browser Buttons, p.306 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.
9.4 Browser Buttons

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

- **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.

- **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).

- **Parameter adjustment button.** Press this button to adjust the parameters that govern the browser’s behavior. Figure 9-3 shows the Browser Configuration dialog box, displayed when you click this button.

![Browser Configuration Dialog Box](image)

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 9.10 The Spy Window, p.320).

- **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.

9.5 Target-Information Window

Figure 9-4  Target Information Window

The Target Information window (Figure 9-4) 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.299.
9.6 Task-List Window

Figure 9-5 shows two examples of the task list produced by clicking Tasks in the browser window selector.

Figure 9-5  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 9.8.1 The Task Browser, p.312.
9.7 Memory-Usage Window

Figure 9-6 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 separate from the memory controlled by VxWorks'. If the target server wants to allocate more memory than is available in the agent-memory pool, it transfers memory from the VxWorks memory pool to the agent-memory pool and then allocates it from the agent-memory pool.

  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

---

2. To set the size of this memory pool, see *Scaling the Target Agent*, p. 142.
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.

### 9.8 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 9-7). 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 (9.9 The Module-Information Window, p.318). 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 9-7 shows the Show box filled in with a request to display an object called `graphSem`.

![Figure 9-7 Filling in the Object-Display Box](image)
9.8.1 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 9-8.

Figure 9-8 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 depend 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 (minus sign) icon, or expose hidden categories by clicking on any (plus sign) icon.

Figure 9-9 shows another task browser running on the same target, but with most of the hardware registers hidden.
9.8.2 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 9-10. 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 9-10 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.
9.8.3 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 9-11 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.
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 9-12 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.
9.8.5 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 9-13.

Before you start a timer, the display resembles the one on the left of Figure 9-13; 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.

9.8.6 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 9-14 shows the display for `semClassId` (the semaphore class).

Figure 9-14  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:

```bash
-> l k u p  "C l a s s I d"
```
9.9 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 9.7 Memory-Usage Window, p.310).

The browser opens a module-information window resembling Figure 9-16 for the selected object module.

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 9-1), and the symbol name (in its internal representation—C++ symbols are displayed “mangled,” and all compiled-language symbols begin with an underbar).

Table 9-1  Key Letters in Symbol Displays

<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></td>
<td>absolute</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td></td>
<td>bss segment</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>common (uninitialized global symbol)</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 9.8 Object-Information Windows, p.311.

### 9.10 The Spy Window

Clicking Spy Chart in the browser window selector produces a display similar to Figure 9-17. 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 9-3). When the Differential check box is off in the Browser Parameters dialog box (see 9.4 Browser Buttons, p.306), 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.
9.11 The Stack-Check Window

Clicking Stack Check in the browser window selector produces a window similar to Figure 9-18. The stack-check window summarizes the current and maximum stack usage for each task currently running.

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.
9.12 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 9-19.

NOTE: It is possible for a task to write beyond the end of its stack while not writing to the last part of its stack. This will not be detected by checkStack(), the underlying routine for the stack-check window.
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.
9.13 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.

9.14 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.

9.14.1 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 9-20 indicates a memory leak in an application that has run long enough to exhaust memory almost completely.

Figure 9-20 A Memory Leak as Seen in the Browser
9.14.2 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 9-21 shows an example of a growing free-list with many small blocks, characteristic of memory fragmentation.

![Fragmented Memory as Seen in the Browser](image)

9.14.3 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 9-22).

Figure 9-22  **Stack Overflow on Task u9**

### 9.14.4 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 9-23) that a task **uHi**, expected to be high priority, is blocked while two other tasks are ready to run.

Figure 9-23  **Browser: uHi Pended**
An immediate thing to check is whether the three tasks really have the expected priority relationship (in this example, the names are chosen to suggest the intended priorities: uHi is supposed to have highest priority, uMed medium priority, and uLow the lowest). You can check this immediately by clicking on each task’s summary line, thus bringing up the windows shown in Figure 9-24.

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 9-25) reveals that the observed situation is ongoing; uMed is monopolizing the target CPU. It should certainly execute by preference to the low-priority uLow, but why is uHi not getting to run?
At this point examining the code (not shown) may seem worthwhile. Doing so, you notice that uMed uses no shared resources, but uHi and uLow synchronize their work with a semaphore. Could that be the problem?

Examining the semaphore with the browser (Figure 9-26) confirms this suspicion: uHi is blocking on the semaphore, which uLow cannot release because uMed has preempted it.

![uHi Blocked on Semaphore](image)

Having diagnosed the problem as a classic priority inversion, the fix is straightforward. As described in VxWorks Programmer’s Guide: Basic OS, you can revise the application to synchronize uLow and uHi with a mutual-exclusion semaphore created with the SEM_INVERSION_SAFE option.

### 9.15 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.
10.1 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 10.1 Introduction, p.329).

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.
10.2 Debugger GUI

Figure 10-1 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 (10.2.1 Debugger Toolbar, Buttons, and Menu Commands, p.331).

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 10.2.1 Debugger Toolbar, Buttons, and Menu Commands, p.331).

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 10.3 Using the Debugger, p.333.)
The debugger command line window for CrossWind provides a command-line interface to the debugger (see 10.5 Using the Debugger Command Line, p.352). The window is not needed for standard debugger operations; the graphical interface provide simpler controls.

### 10.2.1 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 10-2.

![Debug Toolbar](image)

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 10-1.

<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 10.3.2 Setting the Search Path, p.335.</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 10.3.7 Continuing Through a Program, p.340.</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 10.3.10 Detaching from a Running Task, p.344.</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 10.3.10 Detaching from a Running Task, p.344.</td>
</tr>
</tbody>
</table>
### Table 10-1 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 10.3.8 Attaching to a Running Task, p.343.</td>
</tr>
<tr>
<td>![Image]</td>
<td>Interrupt Debugger</td>
<td>Interrupt program execution. See 10.3.5 Interrupting a Program, p.337.</td>
</tr>
<tr>
<td>![Image]</td>
<td>Stop Debugging</td>
<td>Stop the debugger. See 10.3.1 Starting and Stopping the Debugger, p.334.</td>
</tr>
<tr>
<td>n/a</td>
<td>Breakpoints</td>
<td>Open the Breakpoints window. See 10.3.6 Setting Breakpoints, p.337.</td>
</tr>
<tr>
<td>![Image]</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 10.3.6 Setting Breakpoints, p.337.</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 10.3.6 Setting Breakpoints, p.337.</td>
</tr>
<tr>
<td>![Image]</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 10.3.7 Continuing Through a Program, p.340.</td>
</tr>
<tr>
<td>![Image]</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 10.3.7 Continuing Through a Program, p.340.</td>
</tr>
<tr>
<td>![Image]</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 10.3.7 Continuing Through a Program, p.340.</td>
</tr>
</tbody>
</table>
10.3 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 compiler option) to use many of the features of the debugger. The default compiler settings used by the Tornado project facility include debugging symbols.
10.3.1 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 10-3 appears, to allow you to select a target server from the Targets drop-down list.

Figure 10-3 Debugger Target-Selection Dialog Box

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 10.2 Debugger GUI, p.330.)

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.
10.3.2 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). 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 10-4 illustrates the Debugger Source Search Path dialog box.

Figure 10-4 Debugger Source Search Path Dialog Box

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>n/a</td>
<td>ALT+D h</td>
<td>Source Search Path</td>
</tr>
</tbody>
</table>

WARNING: Do not leave more than one directory containing a VxWorks kernel 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.
10.3.3 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 the unload option from the context menu in the project Workspace window (see 4.3.5 Downloading an Application, p.108).

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 10.5 Using the Debugger Command Line, p.352 for more information about command line usage.

10.3.4 Running a Program

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

Figure 10-5 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 10-5 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 10-10 shows the debugger stopped at the entry point of the routine **BigBang()**. The context pointer 1 indicates what statement will execute if you allow the program to resume.

### 10.3.5 Interrupting a Program

You can interrupt execution with **Interrupt Debugger** in the Debug menu, by pressing the $F9$ button, or by using the keyboard shortcut **ALT+SHIFT+F5**.

### 10.3.6 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>$F9$</td>
<td></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 10-7.) 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).

Figure 10-7  The Breakpoints Dialog Box

![The Breakpoints Dialog Box](image)

**NOTE:** The breakpoints you set in this way will affect only the task to which the debugger is attached. If you want your breakpoint to stop all tasks when the attached task hits the breakpoint, set it using `gbreak` from the command line.

The Advanced button opens the Advanced Breakpoint dialog box (Figure 10-8). This

Figure 10-8  The Advanced Breakpoint Dialog Box

![The Advanced Breakpoint Dialog Box](image)
dialog lets you attach conditions to the breakpoint 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 persistent. It remains active until disabled or deleted.

- **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 \texttt{ Breakpoint} 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 \textsuperscript{ } 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 (12.3.2 Editor Preferences, p.402). 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 ( \textsuperscript{ } ) 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 (10.3.7 Continuing Through a Program, p.340). (2) Use the Breakpoints dialog box (see 10.3.6 Setting Breakpoints, p.337). 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.

NOTE: GDB software watchpoints are very intrusive. They should only be used with real-time programs if the overhead is acceptable.

Figure 10-9 shows the editor window with two breakpoints set: a temporary breakpoint (hollow) and a persistent breakpoint (filled in).

Figure 10-9  Breakpoint Marks in Editor Window

10.3.7 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><img src="image" alt="Continue" /></td>
<td>F5</td>
<td>Continue</td>
<td>n/a</td>
</tr>
</tbody>
</table>

340
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 10-10 shows the debugger stopped at the entry point of the routine `graphInit()`. The context pointer 1 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 (10.3.11 Examining Data, Memory, and the Stack, p.344),...
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.

NOTE: The Run to Cursor option is implemented using the GDB until location command; the GDB command always stops on return from the current function to prevent the program from running free. If you use Run to Cursor with the program stopped in a subroutine, it will stop at the return point no matter where the cursor is; to run to the cursor, select Run to Cursor again.

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.
10.3.8 Attaching to a Running Task

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 10-11). You can either select a task in the list, or type the name (or task ID) of a task in the Attach to box. By default, the debugger attaches without stopping the task. The task will stop when it hits a breakpoint.

Figure 10-11  Attach Dialog Box

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 10.6 System-Mode Debugging, p.359. An error display appears if your BSP is not configured to support system mode.
10.3.9 Attaching Automatically

By default, the debugger always attaches to the next task to register an exception. Thus your debugger is always attached to the task that needs debugging.

This default behavior can be changed by selecting Tools>Options>Debugger. The following options can be selected:

- never: do not auto-attach at all
- safely: only auto-attach if we are not attached to any task
- always: always attach to the next task that gets an exception

10.3.10 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.

10.3.11 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>![Watch icon]</td>
<td>ALT+3</td>
<td>Watch</td>
</tr>
<tr>
<td>![Variables icon]</td>
<td>ALT+4</td>
<td>Variables</td>
</tr>
<tr>
<td>![Registers icon]</td>
<td>ALT+5</td>
<td>Registers</td>
</tr>
<tr>
<td>![Memory icon]</td>
<td>ALT+6</td>
<td>Memory</td>
</tr>
<tr>
<td>![Back Trace icon]</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 10-12). 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.

**NOTE:** The windows described in this section update each time your program stops in the debugger. Each update highlights values that changed since the previous display.

**Watch Window**

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Windows Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Watch Window" /></td>
<td>ALT+3</td>
<td>Watch</td>
</tr>
</tbody>
</table>
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 10-13).

![Add to Watch Dialog Box](image)

Enter the name of the symbol and click OK to display the symbol and its current value in the Watch window (Figure 10-14). Use the pop-up menu or the Delete key to remove symbols from the window.

![Debugger Watch Window](image)
Variables Window

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Windows Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Debugger" /></td>
<td>ALT+4</td>
<td>Variables</td>
</tr>
</tbody>
</table>

Click the Variables command in the Debug Windows sub-menu to open a window that shows the values of local variables. Figure 10-15 shows an example of the Variables window.

![Debugger Variables Window](image)

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.
Click the Registers command in the Debug menu to open a window that shows the values in the target registers. Figure 10-16 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 10-16 Debugger Registers Window
Back Trace Window

<table>
<thead>
<tr>
<th>Button</th>
<th>Shortcut</th>
<th>Debug Windows Menu Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Back Trace" /></td>
<td>ALT+7</td>
<td>Back Trace</td>
</tr>
</tbody>
</table>

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 10-17).

Figure 10-17 Debugger Back Trace Window
Memory Window

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 10-18 shows the window docked and maximized.

Figure 10-18  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 the description of the x ("examine") command in GDB User’s Guide: Examining Data for a discussion of the memory-display formats.
10.4 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 10-19 shows a mixed-mode code display.

Figure 10-19  Mixed Source and Disassembly Display
Source display (in either the Source or the Mixed Source and Disassembly view) requires that your application modules be compiled with debugging information which is the default for compilations with the Tornado project facility (see 4Projects, p.75).

NOTE: For some source lines, compilers generate code that is not contiguous, because it is sometimes more efficient to interleave the object code from separate source lines. In this situation, the mixed-mode display rearranges the assembly listing to group all object code below the line that generates it.

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 10.3.7 Continuing Through a Program, p.340.)

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.

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).

10.5 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 10-1). 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.
The following sections summarize some particularly useful commands, and describe commands added by Wind River that are not part of other versions of GDB.

### 10.5.1 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. If it finds the file in either directory, the debugger commands in it are executed; if it finds the file in both directories, the commands in both are executed.

A related initialization file, 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 `installDir\.wind`, then (if the environment variable `HOME` is defined) in `homeDir\.wind`. See 10.7 Tcl: Debugger Automation, p.365 for a discussion of extending GDB through Tcl. See also 10.8.1 Tcl: Debugger Initialization Files, p.372 for a discussion of how the Tornado debugger initialization files interact.

### 10.5.2 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

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

   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.
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 `\fred\host\fred\applic.o`.

Use the *serverFilename* argument to specify what file to download from the server’s point of view. (You must also specify the *filename* argument from the local point of view for the benefit of the debugger itself.)

See 7.6 *Object Module Load Path*, p.270 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*.

---

2. See also the description of **wtx-load-path-qualify** in 10.5.8 *Extended Debugger Variables*, p.358 for another way of managing how the debugger reports **load** path names to the target server.
10.5.3 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 7.2.7 Using the Shell for System Mode Debugging, p.242.)

**call expr**
If a task is already running (and suspended, so that the debugger has control), you can evaluate any source-language expression (including subroutine calls) with the `call` command. This provides a way of exploring the effects of other subroutines without abandoning the suspended call. Subroutine arguments in the expression `expr` may be of any type, including floating-point or double precision.

When you run a routine from the debugger using one of these commands, the routine runs until it encounters a breakpoint or a signal, or until it completes execution. The normal practice is to set one or more breakpoints in contexts of interest before starting a routine. However, you can interrupt the running task by clicking **Interrupt Debugger** in the Debug menu or by pressing `CTRL+BREAK` from the Debugger window.

10.5.4 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 10.7.3 Tcl:
*Invoking GDB Facilities*, p.368 for details.

### 10.5.5 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*.

### 10.5.6 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.

### 10.5.7 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 Programmer’s Guide: WTX Protocol* for descriptions of WTX protocol requests. Convert protocol message names to lower case, and use hyphens in place of underbars. For example, issue the message WTX_CONSOLE_CREATE as the command wtx-console-create.

- **Shell Commands**

  You can run any of the Tornado shell’s primitive facilities described in 7.2.4 *Invoking Built-In Shell Routines*, p.229 in the Debugger window, by inserting the prefix “wind-” before the shell command name. For example, to run the shell `td()` command from the debugger, enter wind-td in the 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 `semShow()` command, enter wind-sem-show.

---

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.
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 \texttt{shSymAddr} to translate C symbols to the Tcl form. For example, to call the shell built-in \texttt{show} on a semaphore ID \texttt{mySemID}, use the following:
\begin{verbatim}
(gdb) wind-show [shSymAddr mySemId]
\end{verbatim}

\section*{10.5.8 Extended Debugger Variables}

You can change many details of the debugger’s behavior by using the \texttt{set} command to establish alternative values for internal parameters. (The \texttt{show} command displays these values; you can list the full set interactively with \texttt{help set}.)

The following additional \texttt{set} parameters are available in the Tornado debugger (CrossWind) in addition to those described in \textit{GDB User’s Guide}:

\textbf{inhibit-gdbinit}  
Specifies whether or not to read the GDB-language initialization files (discussed in 10.8.1 \textit{Tcl: Debugger Initialization Files}, p.372). Default: \texttt{no} (that is, read initialization files).

\textbf{wtx-ignore-exit-status}  
Specifies whether or not to report the explicit exit status of a routine under debugger control. When this parameter is \texttt{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 \texttt{off} to see the explicit exit status as part of the termination message. Default: \texttt{on}.

\textbf{wtx-load-flags}  
Specifies the option flags for the dynamic loader (\texttt{Download} in the \textit{Debug} menu, or \texttt{load} in the Debugger window). These flags are described in the discussion of \texttt{ld()} in \textit{VxWorks Programmer’s Guide: Configuration and Build}. Default: \texttt{LOAD_GLOBAL_SYMBOLS} (4).

\textbf{wtx-load-path-qualify}  
Specifies whether the debugger translates a relative path specified in the \texttt{load} argument to an absolute path when instructing the target server to download a file. Setting this value to \texttt{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. To reset this parameter to 120 seconds, use:

(gdb) set wtx-load-timeout 120

**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.

### 10.6 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 (10.3.8 Attaching to a Running Task, p.343) 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 debugging. (All the material in 10.3 Using the Debugger, p.333 applies to task mode debugging).

Tornado also supports an alternate form of debugging, where you can switch among multiple tasks (also called threads) and even examine execution in system routines such as ISRs. This alternative mode is called system mode debugging; it is also sometimes called external mode. In system mode, you can use global breakpoints to stop the entire system whenever any task hits the breakpoint.

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 10.6.3 Thread Facilities in System Mode, p.361) 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 7.2.7 Using the Shell for System Mode Debugging, p.242, then attach to them with the thread command.

### 10.6.1 Configuring System Mode

In order for system mode debugging to work properly, the following items must be set correctly.

<table>
<thead>
<tr>
<th># Define</th>
<th>Required Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#define WDB_COMM_TYPE</code></td>
<td><code>WDB_SERIAL</code></td>
</tr>
<tr>
<td><code>#define WDB_MODE</code></td>
<td><code>WDB_MODE_DUAL</code></td>
</tr>
<tr>
<td><code>#define INCLUDE_PC_CONSOLE</code></td>
<td><code>PC_CONSOLE</code></td>
</tr>
<tr>
<td><code>#define PC_CONSOLE</code></td>
<td><code>CONSOLE_TTY</code></td>
</tr>
<tr>
<td><code>#define CONSOLE_TTY</code></td>
<td><code>NUM_TTY</code></td>
</tr>
<tr>
<td><code>#define WDB_TTY_CHANNEL</code></td>
<td><code>WDB_TTY_DEV_NAME</code></td>
</tr>
</tbody>
</table>
10.6.2 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 wdbSuspendSystemHere (), the normal suspension point after attach system.

⚠️ WARNING: Not all targets support system mode, because the BSP must include a special driver for that purpose (see 4.7 Configuring the Target-Host Communication Interface, p.138). If your target does not support system mode, attempting to use attach system produces an error.

10.6.3 Thread Facilities in System Mode

In system mode, the GDB thread-debugging facilities become useful. Thread is the general term for processes with some independent context, but a shared address space. In VxWorks, each task is a thread. The system context (including ISRs and drivers) is also a thread. GDB identifies each thread with a thread ID, a single arbitrary number internal to the debugger.

<table>
<thead>
<tr>
<th># Define</th>
<th>Required Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define WDB_TTY_BAUD</td>
<td>As appropriate.</td>
</tr>
</tbody>
</table>
You can use the following GDB commands to manage threads:

**info threads**
Displays summary information (including thread ID) for every thread in the target system.

**thread idNo**
Selects the specified thread as the current thread.

**break linespec thread idNo**
Sets a breakpoint affecting only the specified thread.

For a general description of these commands, see *GDB User’s Guide: Debugging Programs with Multiple Threads*. The sections below discuss the thread commands in the context of debugging a VxWorks target in system mode.

**Displaying Summary Thread Information**

The command **info threads** shows what thread ID corresponds to which VxWorks task. For example, immediately after attaching to the system, the **info threads** display resembles the following:

```
(gdb) info threads
 4 task 0x4fc868   tExcTask    0x444f58 in ?? ()
 3 task 0x4f9f40   tLogTask    0x444f58 in ?? ()
 2 task 0x4c7380 + tNetTask    0x4151e0 in ?? ()
 1 task 0x4b0a24   tWdbTask    0x4184fe in ?? ()
(gdb)
```

In the **info threads** output, the left-most number on each line is the debugger’s thread ID. The single asterisk (*) at the left margin indicates which thread is the current thread. The current thread identifies the “most local” perspective: debugger commands that report task-specific information, such as **bt** and **info regs** (as well as the corresponding 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  0x49fe4 in shell ()
(gdb) thread 3
[Switching to task 0x3aa9d8   tFtpdTask ]
#0  0x66454 in semBTake ()
(gdb) print/x $i0
$3 = 0x3bdb50
```

As in the display shown above, each time you switch threads the debugger exhibits the newly current thread’s 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:
Internally, the debugger still gets control every time any thread encounters the breakpoint; but if the thread ID is not the one you specified with the break command, the debugger silently continues program execution without prompting you.

⚠️ **CAUTION:** Because the thread ID for the system context is not constant, it is not possible to set a breakpoint specific to system context. The only way to stop when a breakpoint is encountered in system context is to use a non-task-specific breakpoint.

### Switching Threads Implicitly

Your program may not always suspend in the thread you expect. If any breakpoint or other event (such as an exception) occurs while in system mode, in any thread, the debugger gets control. Whenever the target system is stopped, the debugger switches to the thread that was executing. If the new current thread is different from the previous value, a message beginning with “Switching to” shows which thread is suspended:

```
(gdb) thread 2
(gdb) cont
Continuing.
```

Interrupt...
Program received signal SIGINT, Interrupt.
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.

### 10.7 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 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 10.8 Tcl: Debugger Customization, p.372. 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.

#### 10.7.1 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 10.7.3 *Tcl: Invoking GDB Facilities*, p.368.

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 10.8.1 *Tcl: Debugger Initialization Files*, p.372 for a discussion of where you can put this file, and of how all the Tornado debugger and GDB initialization files interact.)

### 10.7.2 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 10.5.1 GDB Initialization Files, p.353), because **tclproc** is a GDB command rather than a Tcl command. However, if you want to keep the **tclproc** definition together with supporting Tcl code, you can exploit the **gdb** Tcl extension described in 10.7.3 Tcl: Invoking GDB Facilities, p.368 to call **gdb tclproc** in **gdb.tcl**.

**tlcdebug**

Toggles Tcl debugging mode. Helps debug Tcl scripts that use GDB facilities. When Tcl debugging is ON, all GDB commands or other GDB queries made by the Tcl interpreter are printed.

**tclerror**

Toggles Tcl verbose error printing, to help debug Tcl scripts. When verbose error mode is ON, the entire stack of error information maintained by the Tcl interpreter appears when a Tcl error occurs that is not caught. Otherwise, when verbose error mode is OFF, only the innermost error message is printed. For example:

```
(gdb) tcl puts stdout [expr $x+2]
can’t read “x”: no such variable

(gdb) tclerror
TCL verbose error reporting is ON.

(gdb) tcl puts stdout [expr $x+2]
can’t read “x”: no such variable
while executing
"expr $x..."
invoked from within
"puts stdout [expr $x..."
```

Tcl also stores the error stack in a global variable, **errorInfo**. To see the error stack when Tcl verbose error mode is OFF, examine this variable as follows:

```
(gdb) tcl $errorInfo
```

For more information about error handling in Tcl, see C.2.9 Tcl Error Handling, p.454.
10.7.3 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 C.2.9 Tcl Error Handling, p.454).

**gdbEvalScalar exprlist**
Evaluates a list of expressions `exprlist` and returns a list of single integer values (in hexadecimal), one for each element of `exprlist`.\(^4\) 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 C.2.9 Tcl Error Handling, p.454 for information about unwinding).

---

\(^4\) 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.
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. To leave either 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

10.7.4 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:

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

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

5. Keep in mind that for interactive exploration of a list the window (Figure 10-14) described in Watch Window, p.345 is probably more convenient.
for (pNode = pHead; pNode; pNode = pNode->next)
...

We imitate this code in Tcl, with the important difference that all Tcl data is in strings, not pointers.

The argument to the Tcl procedure will be an expression (called head in our procedure) representing the first node of the list.

Use gdbEvalScalar to convert the GDB expression for a pointer into a Tcl string:

    set pNode [gdbEvalScalar "$head"]

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

    set pNode [gdbEvalScalar "((struct node *) $pNode)->next"]

Putting these lines together into a Tcl for loop, the procedure (in a form suitable for a Tcl script file) looks like the following:

    proc traverse head {
        for {set pNode [gdbEvalScalar "$head"]} {
            {$pNode} {
                (set pNode [gdbEvalScalar "((struct node *)$pNode)->next"])
                {puts stdout $pNode}
            }
        }
    }

In the body of the loop, the Tcl command puts prints the address of the node.

To type the procedure directly into the Debugger window would require prefacing the text above with the tcl command, and would require a backslash at the end of every line.

If pList is a variable of type (struct *node), you can execute:

    (gdb) tcl traverse pList

The procedure displays the address of each node in the list. For a list with two elements, the output would look something like the following:

    0xffeb00
    0xffea2c

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

    {puts stdout [format "data = %d" \
        [gdbEvalScalar "((struct node *) $pNode)->data"]]}
You can bind a new GDB command to this Tcl procedure by using \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}

10.8 Tcl: Debugger Customization

Like every other Tornado tool, the debugger’s graphical user interface is “soft”
( amendable to customizing) because it is written in Tcl, which is an interpreted
language. The Tornado API Reference 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}.

10.8.1 Tcl: Debugger Initialization Files

You can write Tcl code to customize the debugger’s graphical presentation in a file
called \texttt{crosswind.tcl}. The debugger looks for this file in two places: first under
\texttt{c:\tornado} (or whatever directory you specify with the environment variable
\texttt{WIND\_BASE}), and then in the directory specified by the \texttt{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 \texttt{.gdbinit} and \texttt{gdb.tcl} initialization files for the debugger command
language (see 10.7 Tcl: Debugger Automation, p.365). The following outline
summarizes the role of all the Tornado debugger customization files. The files are
listed in the order in which they execute:
installDir\.wind\gdb.tcl
Use this file to customize the Tcl interpreter built into GDB itself (for example, to define Tcl procedures for new GDB commands). This file is unique to the Tornado debugger (that is, it is not shared by any other GDB configuration you might install). When issuing commands intended for GDB, you must prepend them with gdb.

The version of this file under installDir (for example, in installDir\.wind) is shared by everyone who uses Tornado on the same PC.

homeDir\.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.

homeDir\.gdbinit
Use this file for any initialization you want to perform in GDB’s command language rather than in Tcl. This file is not unique to the Tornado debugger; it is shared by any other GDB configuration you may install.

.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.

installDir\.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.

homeDir\.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:

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

### 10.8.2 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.

**downtowncl**
From the graphical-interface layer, `downtowncl` executes the remainder of the line in the Tcl interpreter integrated with GDB. The result of `downtowncl` is whatever GDB output the command generates. Use `downtowncl` 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 `downtowncl` 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, `downtowncl` and `writeDebugInput` are likely to be embedded in Tcl procedures, because (in conjunction with the commands discussed in 10.7.3 Tcl: Invoking GDB Facilities, p.368) they are the path to debugger functionality from the graphical front end.
Most of the examples in 10.8.3 Tcl: Experimenting with Tornado Debugger Extensions, p.375, below, center around calls to `downtcl`.

### 10.8.3 Tcl: Experimenting with Tornado Debugger Extensions

The examples in this section use the Tcl extensions summarized in Table 10-3. For detailed descriptions of these and other Tornado graphical building blocks in Tcl, see *Tornado API Programmer’s Guide* or the online *Tornado API Reference*.

<table>
<thead>
<tr>
<th>Table 10-3</th>
<th>Tornado UI Tcl Procedures Used in Debugger Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Tcl Extension</strong></td>
</tr>
<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 10.5.2 Selecting Modules to Debug, p.353, 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 10-1 illustrates a Debug menu command *Add Symbols* to handle this through the graphical user interface, instead of typing the `add-symbol-file` command.

**Example 10-1 Add-Symbols Command**

```tcl
# MENU COMMAND: "Add Symbols", additional entry under "Debug"
menuItemInsert -after {"&Debug" "Do&wnload...\tShift+F6"} \
    {"Add Symbols..."} {windAddSyms}
```

```tcl
# 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:
#  \$windAddSyms
#
```

375
# RETURNS: N/A
#
# ERRORS: N/A
# proc windAddSyms {} {
#   set result [fileDialogCreate -title "Symbols from file" -filemustexist \
#     -filefilters "Object Files (*.o;*.out)|*.o;*.out| \
#     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 10-2 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 10-2 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}
}
11

Building VxDCOM Applications

11.1 Introduction

This chapter describes the step-by-step process of creating and building VxDCOM applications using Tornado. To more easily create VxDCOM applications, the basic VxDCOM support includes the following tools:

an application wizard
Let you easily generate skeleton code for a basic VxDCOM application, without having to define CoClasses and interfaces in IDL (the Interface Definition Language).

a C++ template class library
Facilitates writing client and server implementation code.

an IDL compiler
Compiles IDL file, generating the necessary proxy/stub and header files required by VxDCOM.

For a detailed description of the VxDCOM technology, see the VxWorks Programmer’s Guide: VxDCOM Applications.
11.2 The VxDCOM Development Process

VxDCOM clients and servers for VxWorks can be created either as bootable or downloadable applications. The following step-by-step overview summarizes the VxDCOM development process:

**Step 1: Create a Bootable Image with VxDCOM Support**
Build a VxWorks bootable image with VxDCOM support components. You will need this image whether you are creating a bootable or a downloadable application. This step is covered in *Configuring a VxDCOM Bootable Image* on p. 379.

**Step 2: Configure Any DCOM Component Parameters**
Optionally configure the parameters for the DCOM components. This step is covered in *Configuring the DCOM Parameters* on p. 380.

**Step 3: Generate Skeleton Project Files with the VxDCOM Wizard**
Run the VxDCOM wizard from the command line. Use the wizard to define the CoClass and interfaces, to choose the server model and client skeleton program. This generates skeleton files for header prototypes, coclass definitions, interface and library definitions, and so on. These steps are described in *Using the VxDCOM Wizard* on p. 381.

**Step 4: Implement the Server and Client**
Complete the implementation of the server by editing the CoClass files to implement the interface methods. These files also contain (auto-generated) code to auto-register the server. This step is covered in *Implementing the Server and Client* on p. 392.

**Step 5: Add the Files to Project and Build It**
Choose a bootable or downloadable application model. You can use either the project facility or the **makefile**, generated by the wizard, to build your application. Build and link the application. This step is covered in *Building and Linking the Application* on p. 393.

(a) Build the application.
(b) Ensure that it is correctly linked with proxy/stub code.
(c) Build the client program, if your project includes one.
Step 6: Register and Deploy Your VxDCOM Application

Deploy your application by registering the type library and setting the proper configurations for server authentication. These steps, listed below, are described in Registering, Deploying, and Running Your Application on p. 394.

(a) Register any necessary proxy DLLs on Windows.
(b) Register the type library.
(c) Register the server.
(d) Authenticate the server.
(e) Run the application and activate the server.

11.3 Configuring a VxDCOM Bootable Image

Whether you choose to create a bootable or downloadable client or server, you need a VxWorks bootable image that contains a kernel and VxDCOM support. You need such an image for all VxDCOM applications.

If you are creating a bootable application, add your VxDCOM application files to the bootable system image. If you are creating a downloadable application, add your VxDCOM files to this downloadable module. Then download that module to the bootable system containing the VxDCOM support.

11.3.1 Adding VxDCOM Component Support

After you have created the kernel, add the appropriate VxDCOM support components. Some components are required for all VxDCOM applications, some are required only for DCOM, and some are optional, providing additional functionality. This section describes VxDCOM support and when to add it to your kernel.

COM Core Component

This component is called COM_CORE and provides support for C COM projects as well as C++ COM and DCOM projects.
COM Support Component

The COM support component is required for all C++ COM and DCOM applications.

DCOM Support Components

The DCOM and DCOM_PROXY support components are required for DCOM applications. The DCOM component provides support for distributed COM. The DCOM_PROXY component provides support for proxy/stub code. Both of these components require the basic COM support, described above.

OPC Program Support

The DCOM_OPC support component is required if you are writing your own OPC server. If you are using the VxOPC product it is not required. For more information, see the VxWorks Programmer’s Guide: VxDCOM Applications.

ComShow Routines Support

The COM_SHOW support component is required only if you want to use the COM show routines. Including this component adds diagnostic routines that may be used to interrogate the registry held within the VxWorks run-time.

DCOMShow Routines Support

The DCOM_SHOW component is required only if you want to debug the VxDCOM wire protocol. Including this component adds a significant overhead; therefore, it should only be used at the request of Wind River Customer Support to provide debug information. It should not be shipped as part of a production system.

11.3.2 Configuring the DCOM Parameters

If your application includes DCOM, you can optionally configure parameters for this component. Change the parameter values from within the project facility by selecting Properties from the popup menu on the appropriate binary component.

The DCOM parameter descriptions, their default values, and the value ranges are described in VxWorks Programmer’s Guide: VxDCOM Applications.
11.4 Using the VxDCOM Wizard

The VxDCOM Wizard lets you generate a completely new VxDCOM project, or import an existing COM or DCOM server. To run the wizard, type at the command line:

```
installDir/host/hostType/bin/torVars.bat
comwizard projDir
```

where the `projDir` is the directory for your project. The files generated by the wizard are generated in this directory. The directory name you type in is the default name for your CoClass for new projects. You have an option to modify this name in the wizard.

11.4.1 Choosing the Project Type

The first page of the VxDCOM wizard, shown in Figure 11-1, lets you choose the project type.

Figure 11-1 Choosing the VxDCOM Project Type
The options are:

Create COM/DCOM Skeleton Project
Create skeleton files for a new COM or DCOM component. For details, see Creating a COM/DCOM Skeleton Project on p. 382.

Import existing files into new Project
Create a project using existing COM application code. For details, see Importing Existing Files into a New Project on p. 388.

Choose the type of project and click Next.

11.4.2 Creating a COM/DCOM Skeleton Project

If you are creating a new project, you simply specify your server type and implementation language, and define your CoClass and interfaces using the wizard GUI. You do not have to write them in IDL (Interface Definition Language); nor do you have to write proxy/stub code.

From the information you enter, the wizard generates the primary IDL definition file and the additional files appropriate to the server model and client program selected. These output files are described in The Generated Output on p. 389.

Defining the CoClass

Figure 11-2 shows the CoClass Input dialog of the wizard. From this dialog you define the CoClass by adding interface methods and parameters. This dialog defaults to a CoClass named for the argument you passed to comwizard.

You can modify the name of the CoClass (or any items), by highlighting the item and choosing Edit.
To define the CoClass, you add interfaces and interface methods, and you specify parameter types for those methods. You typically do this in steps:

- Add one or more interfaces to the CoClass.
- For each interface, add one or more methods.
- For each method, add one or more parameters, specifying the attribute and type of each.

When your CoClass definition is complete, click Next.

Adding Items

To add an item, highlight the item and choose Add, as shown in Figure 11-3. This opens the appropriate Add dialog. These dialogs let you enter names for new interfaces, interface methods, or interface method parameters.
Adding Method Parameters

When adding interface method parameters, you must also specify the attribute and type of each parameter. As shown in Figure 11-4, edit boxes for the parameter attribute and type appear with defaults. To change these, select the correct option from the dropdown listboxes. Once you click Apply, your selections will appear in the CoClass definition.

- **Method Parameter Attributes.** When selecting interface method parameters, all [out] parameters and [out] parameter combinations must be pointers. The attribute options for these parameters are part of the IDL language and are documented in the IDL reference section of the VxWorks Programmer’s Guide: VxDCOM Applications.

- **Method Parameter Types.** When selecting the data type of the interface method parameter, the dropdown listbox displays a list of automation data types, as shown in Figure 11-5. Using automation data types provides built-in marshaling support under DCOM.
Figure 11-4  Adding Method Parameters

Figure 11-5  Method Parameter Data Types
It is also possible to use non-automation types, however these are not available from the VxDCOM wizard dialog and may require additional linking. If you need to use non-automation data types, see the Adding Non-Automation Types, p.389.

The `widl` tool compiles both automation data types and non-automation data types. The types compiled by `widl` are listed in the `VxWorks Programmer's Guide: VxDCOM Applications`.

Choosing CoClass Options

Once your CoClass definition is complete, clicking Next displays the CoClass Options dialog of the wizard, shown in Figure 11-6. From this dialog you specify the CoClass server model, implementation language, and any optional client programs. Then click Next.

![Diagram of CoClass Options dialog]

Server Models

Choose the Server Model for which you want to generate project files.
COM
A COM server model uses the COM technology entirely within the VxWorks system. A COM server is limited to communication with a COM C or C++ client on the same VxWorks target. You can use COM components to design object-oriented code based on the internal use of COM interfaces.

DCOM
A DCOM server model uses a distributed, component-based system (Distributed COM). DCOM extends the basic COM technology across process and machine boundaries by using a client-server application-level protocol for remote procedure calls. You can use a VxDCOM server, for example, to connect the desktop PC with distributed objects over a network.

Language
Choose the language used to implement the server CoClass.

C++
This language can be used for either COM or DCOM applications.

C
This language can be used only for COM, not DCOM, applications.

Client Skeleton Programs
Depending upon the server model, you can select from among several client application types. If you have a COM server, you can have only one C++ COM client project. If you have a DCOM server, you can choose more than one client program and clients of any type. Choosing a client program is optional.

C++ COM Client Program
The client must exist on the same VxWorks communication server as the VxWorks COM server.

C++ DCOM Client Program
The DCOM client can reside on either a VxWorks target or on a PCs running Windows NT. The communication server must be a VxWorks DCOM target server. When a client is on a PC, or another target, the DCE network layer is used. When the client is on a target, even though DCOM calls are made, it detects that the client is on the same host, and uses COM instead.

Visual Basic DCOM Client Program
A Visual Basic client program (such as Excel Basic, Word Basic, Access Basic, and so on) must reside on a PC running Windows NT. The client is written in Visual Basic, and the communication server is a DCOM VxWorks target server.
**Generating the Skeleton Files**

The last dialog of the wizard, Project Creation, appears. Simply click Finish to generate your project.

**11.4.3 Importing Existing Files into a New Project**

**Porting Existing Applications**

If you have an existing COM application, choose the option to Import existing files into new Project and choose Next. This brings up the Import Existing Project dialog shown in Figure 11-7.

![Figure 11-7 Adding Existing VxDCOM Files](image)

The files to add are the .idl file, the server implementation file, and the CoClass header file.
**Editing IDL Files**

If you wish to edit your .idl file by hand, you can refer to the VxWorks Programmer's Guide for details on the IDL file structure and the correct syntax for defining elements in that file. Remember to specify an HRESULT as the return type for all interface methods and to use automation data-types and directional attributes appropriately for your interface parameters.

If you do not use automation data types, refer to *Adding Non-Automation Types*, p.389 below. As a guide for writing in these files you can use the wizard GUI dialogs and wizard generated IDL files, and the CoMathDemo.idl file.

If for any reason you need to add additional interface definitions manually to the auto-generated .idl file, for each additional interface you must:

- Generate a new GUID (you can use the UUIDGEN utility to do this).
- Specify this value as the [uuid] attribute of your interface.

**Adding Non-Automation Types**

If you are using non-automation data types, the simplest way to add them is to generate all of the simple automation data types for the interface first using the wizard, generate the skeleton code. Then edit the .idl file and server implementation file by hand, adding the interface methods that could not be added with the wizard because they use non-automation types.

If you are defining interface methods that use non-automation types, you would not use the [oleautomation] attribute in your interface definition.

For more information on IDL, see the Microsoft documentation. Be aware that some of that information applies only to RPC interfaces and not to COM interfaces.

**11.5 The Generated Output**

The wizard generates output files in several subdirectories of your project name directory referred to here as basename. The content of those directories differs depending upon the options you selected in the wizard.
Output Directories

When the wizard finishes running it generates output files and creates 2 directories, which are:

- basename
- basename/Client

Subsequent sections describe the additional files that appear in these two directories. Depending upon the type of server and client you selected in the wizard, the content of these directories differs.

Project Work Files

These are the primary files to identify for working with your project. You can use these files to write code, compile, edit, or browse. Some are used for compiling, linking, building, and deployment of your application.

Makefile
The makefile used for building the project from the command line. Using this file is the default method for building the downloadable module. For details, see 11.7 Building and Linking the Application, p.393.

basename.idl
This is the IDL file, identified by the .idl extension. This file is automatically compiled by widl when you build your Tornado project.

basenameImpl.cpp
This is the CoClass implementation file for your server, identified by the Impl.cpp extension. This is the file in which you implement the interface methods of your server CoClass.

basenameImpl.h
This is the CoClass header file, identified by the Impl.h extension. This file contains definitions and prototypes generated from the items you defined in the wizard.

Client/basenameClient.cpp
This is the skeleton implementation file for your client if you choose COM as your server and client model. It is identified by the Client.cpp extension. You edit this file to add client code to activate the COM server.
Client/basenameDCOMClient.cpp
This is the skeleton implementation file for your DCOM client application. It is identified by the DCOMClient.cpp extension. You edit this file to add client code to activate the DCOM server.

Client/basename.rgs (DCOM-only)
This is the registry-script file, identified by the .rgs extension. This file is used by the VxDOM build tools to register the CoClass after it has been built. It is generated only when the DCOM server is selected. You only need to edit this file if you are writing a DCOM application, in which case you modify the ‘vxworks.target’ entry.

Client/nmakefile (C++ Client only)
This is the host-side (client) makefile, always generated as nmakefile. It is used (with nmake) to build the MFC client application for Win32 or for VxWorks.

Server Output Files

The COM and DCOM server files differ only in the proxy stub code file used for marshaling.

COM Server Output
These are the files output to the basename directory for a COM server application.
- basename.h
- basename.idl
- basenameImpl.cpp
- basenameImpl.h
- basename_i.c

DCOM Server Output
These are the files output to the basename directory for a COM server application.
- basename.h
- basename.idl
- basenameImpl.cpp
- basenameImpl.h
- basename_i.c
- basename_ps.cpp
This file is output to the basename/Client directory, whether or not a client type is selected from the wizard:

- basename.rgs file

**Client Output Files**

If you selected more than one client program with your server choice, then all files necessary for each client are output into the basename/Client directory. Remember that, for DCOM server projects, the basename/Client directory also contains the basename.rgs file. However, this file is only used by DCOM clients to register the type library.

**COM Client Output**

For a COM client, the following files are generated in the Client directory:

- basenameClient.cpp

**DCOM C++ Client Output**

For a DCOM C++ client, the following files are generated in the Client directory:

- basenameDCOMClient.cpp
- nmakefile

**DCOM Visual Basic Client Output**

For a DCOM Visual Basic client, the following files are generated in the Client directory:

- basenameClient.bas

11.6 Implementing the Server and Client

Once your system is created, you are ready to implement your CoClass methods for the server component and write the code for any client programs.

- COM server code in the implementation file basenameImpl.cpp, located in your project directory.
11.7 Building and Linking the Application

Add the appropriate files, generated by the wizard, to your project. These files include the implementation code for the server and client. If you are creating a bootable application, add the files to the bootable system image. If you are creating a downloadable application, add the files to a downloadable module, ensuring that the module is downloaded to a bootable system containing the required VxDCOM support for your application.

Then, build the project. Although you can use the project facility from the IDE to build, the recommended method is to use the wizard-generated Makefile from the command line. The generated Makefile automatically runs widl. If you are building from the project facility, you must run widl manually. Running widl generates the proxy/stub code, the identification GUIDs, and the interface header prototypes. The build process also links the proxy/stub code.

Linking Proxy/Stub Code

The proxy/stub components generated by widl are required for the marshaling method used to remotely invoke interface methods across task boundaries. In order for your application to use marshaling, server and client code must be linked with proxy/stub code. The proxy/stub code source file is basename_ps.cpp and the derived object file is basename_ps.o. You will find these in the workspace under each client and server component. The proxy/stub code is automatically linked to your client and server components when the system is built in the IDE.

1. Win32 uses type library based marshaling. VxDCOM generates custom marshaling code that is linked into the object module.
On an NT machine, proxy/stub code is not required for automation data types.

**Building the Win32 Client**

To build the client application for Win32, be sure that you have Visual C++ installed with the path to `nmake` configured in the command shell. Then, from the command line, run:

```
nmake -f nmakefile
```

---

**11.8 Registering, Deploying, and Running Your Application**

This section describes registering proxy DLLs, the type library, and the server, authenticating and activating the server.

**11.8.1 Registering Proxy DLLs on Windows**

If your project uses non-automation types, you must register the proxy DLLs on Windows.

Typically, when you define your interface you use automation data types. These are the types you can select from the VxDCOM wizard for your interface method parameters. When you use these types, the parameters are defined by the `oleautomation` attribute, to indicate that they are automation types. This signals to the Win32 Automation Marshaler that no extra Win32 proxy/stub DLLs are required because the marshaling of these types is handled automatically.

However, if your project requires non-automation types then you cannot specify the `oleautomation` attribute nor automate the marshaling of the parameter types. For interfaces that use non-automation types, you must generate and install your own Win32 proxy/stub DLLs. The DLL containing the interface proxy must be distributed with all client applications that use that new interface.

The specific non-automation types that `widl` can compile are described in the *VxWorks Programmer’s Guide: VxDCOM Applications*.

The details of generating Win32 proxy/stub DLLs is outside of the scope of this document. Please refer to the Win32SDK MIDL documentation for details.
11.8.2 Register the Type Library

The type library is a binary file with a .tlb extension, that stores information about a DCOM object’s properties and methods in a form that is accessible to other applications at run time. Windows’ client applications use a type library to determine which interfaces an object supports and how to invoke the interface methods. For this reason, the type library must be registered. To register the type library, run \installDir\host\x86-win32\bin\vxreg32.exe. This command adds the type library to the Windows Registry at its current location, so that the object can be accessed from the Windows host. If you add interfaces to the IDL file, the type library must also be re-registered so that the interface becomes known to the Windows Automation Marshaler.

The registry will contain an entry called ‘vxworks.target’ that needs to be modified to point to the actual target, so that it is available from, for example, Visual Basic. There are two ways to change the target machine entry and register the type library:

- Edit the basename.rgs file (generated by the wizard) by changing the ‘vxworks.target’ entry to the target IP address, and then run vxreg32 as described above.
- Run vxreg32 first, and then change the Windows registry entry for the target by opening up DCOMCNFG and using that tool to change the location.

11.8.3 Registering the Server

You must register your DCOM server classes in the VxWorks COM registry, so that client applications can locate them. There is one server-class per object-module. The object-module contains the server-class code, the proxy/stub code, and a registration object. This registration object registers the server-class with the VxWorks Registry when its constructor runs, guaranteeing that modules are automatically registered, regardless of whether they are pre-linked or downloaded. This auto-registration process does not rely on constructor ordering and can, thus, be safely performed at any point during system initialization.

To automatically register a DCOM server at load time or system startup, include the AUTOREGISTER_COCLASS macro in the CoClass implementation file. This will create an entry for the class in the VxWorks COM registry. AUTOREGISTER_COCLASS associates the CoClass with its CLSID, and registers the module name in the VxRegistry against its CLSID.
This macro is automatically included when the skeleton implementation file is generated. Thus, you do not need to add the macro code yourself (unless you work with files that were not auto-generated.)

The AUTOREGISTER_COCLASS macro takes three arguments: the server implementation CoClass, priority-scheme, and the default priority. The priority parameters are used as part of the real-time extension priority schemes, and are discussed in the *VxWorks AE Programmer's Guide: Writing COM and DCOM Clients and Servers*.

### 11.8.4 Authenticating the Server

The DCOM registry under VxWorks is not the complex, multi-purpose registry that Win32 supports, but is designed specifically for that purpose of:

- Allowing COM server classes to register their CLSIDs (class-IDs).
- Providing a link to an instantiation procedure, given that CLSID.
- Determining the correct proxy/stub configuration for any given interface, given its IID (unique identifier).

Therefore, the VxWorks registry works on a simple associative lookup method, keyed by the interface and class identifiers. The value that can be stored in the registry-entry is simply a string, which is used by various internal functions to look up proxy/stub classes, system-provided objects (such as the standard marshaler), and other objects identified by these values.

VxWorks also exposes non-Win32-compliant API calls to provide access to this registry. However, user/application code does not need to use these APIs, since widl automatically generates the code for registration of server-classes.

VxDCOM can participate in the basic NTLM authentication procedures when acting as a server, but not as a client. It can recognize incoming authentication requests from NT and correctly take part in the challenge/response transaction, but by default will not take any action based upon those transactions. Future versions of VxDCOM may more fully support the NTLM security system depending upon NT domain security, network trusts, and so on. However, for now the safest and most predictable approach is to disable client-side security by using either the registry/DCOMCNFG tool or by calling CoInitializeSecurity().
11.8.5 Activating the Server

After you have completed these steps you can run your program. See the VxWorks Programmer's Guide: VxDCOM Applications for descriptions of the MathDemo program client code, which activates the demo server component.

All VxDCOM threads must be created with VX_FP_TASK.
12

Customization

12.1 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.

12.2 Toolbars and Status Bar

Use the View>Toolbars menu item to customize the toolbars that are displayed in the IDE. All toolbars can be floating or docked. Each menu item acts as a toggle to display or hide the specified toolbar. The options are:

Standard Toolbar
- Standard Windows functions including Open, Save, Cut, Paste, and Print.

Launch Toolbar
- Launch the browser, the shell, the debugger, the simulator, WindView, and triggering.

Debug Toolbar
- Debugger functions including start debugging, run a function, step, and watch.
WindView Toolbar
WindView zoom functions, filtering, and numerical analysis.

Build Toolbar
Build, compile current file, update dependencies, and download.

Use the View>StatusBar menu item to toggle display of the status bar at the bottom of the Tornado window.

12.3 Setting Options

You can specify the various default options that Tornado uses through the Options entry in the Tornado Tools menu.

12.3.1 Setting Download Options

The Download page provides options for handling symbols when objects are downloaded to the target (Figure 12-1).

Figure 12-1 Download Page
The options are as follows:

**System Symbol Table**

The system symbol table offers two check box options:

- **LOAD_LOCAL_SYMBOLS**
  Only local symbols are added to the system symbol table

- **LOAD_GLOBAL_SYMBOLS** (default)
  Only external symbols are added to the system symbol table

In order to obtain the other symbol options, you can:

- Check nothing, the equivalent of **LOAD_NO_SYMBOLS**, which adds no symbols to the system symbol table.
- Check both boxes, the equivalent of **LOAD_ALL_SYMBOLS**, which adds all symbols to the system symbol table.

**Common Symbol Resolution**

The common symbol resolution options are mutually exclusive.

- **LOAD_COMMON_MATCH_NONE**
  Allocate common symbols, but do not search for any matching symbols (the default)

- **LOAD_COMMON_MATCH_USER**
  Allocate common symbols, but search for matching symbols in user-loaded modules

- **LOAD_COMMON_MATCH_ALL** (default)
  Allocate common symbols, but search for matching symbols in user-loaded modules and the target-system core file

**C++ Constructors**

The C++ constructors options are mutually exclusive:

- **LOAD_CPLUS_XTOR_AUTO** (default)
  C++ ctors/dtors management is explicitly turned on

- **LOAD_CPLUS_XTOR_MANUAL**
  C++ ctors/dtors management is explicitly turned off

**Miscellaneous**

- **LOAD_HIDDEN_MODULE** (not default)
  Load the module but make it invisible to **WTX_MSG_MODULE_LIST**

The Set Defaults buttons resets the options to their defaults.
WARNING: The first time you start Tornado, you must open the Tools>Options>Download tab and set the defaults. Otherwise you may receive unexpected results.

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

12.3.2 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 12-2.

Figure 12-2 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.

12.3.3 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 12-3.

**Figure 12-3**  **External Editor Page**

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.416 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.
12.3.4 Project

Select Options in the Tools menu, then click Projects to specify certain project attributes. The Projects page is shown in Figure 12-4.

The following choices are available on the Projects page:

Enable extended Build Options
Checking the Enable extended Build Options box causes Standard BSP Builds to be added to the Build tab. For more information on command-line builds, see 5. Command-Line Configuration and Build.

Component Properties
Checking the Show advanced component properties box adds the Definition tab to the component property window. The Definition page shows the internal schema and attributes for the component. This may be helpful for authoring or debugging components.

Build Environment Variables
The Build Environment Variables section displays two environment variables, WIND_SOURCE_BASE and WIND_PROJ_BASE, that provide flexibility in locating and sharing project files. In most cases these variables are not needed. For information on when you might need them and how to use them, see Sub-Projects, p.88.
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.

![Customize Builds Example: Current Default](image)

**12.3.5 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 12-6.

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 12-1).

**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 12-1).
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.

NOTE: If you use Visual SourceSafe as your version control system, you must also have “Assume project based on working folder” checked in SourceSafe. This is not the default, but it affects only command-line SourceSafe use. Go to SourceSafe Explorer, Tools->Options->Command Line Options, check this option, and exit SourceSafe Explorer: the change does not take effect until you exit SourceSafe Explorer.

12.3.6 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 12-8.
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 12-2 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>
<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>
</tbody>
</table>
### Table 12-2  Window Elements for Coloring  *(Continued)*

<table>
<thead>
<tr>
<th>Color Preferences Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<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 Auto Scale 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.
12.3.7 Debugger

Select Options in the Tools menu, then click Debugger to customize various debugger attributes. The Debugger page is shown in Figure 12-9.

Figure 12-9  Debugger Page

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 throwing the exception.

12.3.8 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 12-10.

Figure 12-10 Tornado Registry Page

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.
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.

12.3.9 Installation and Licenses

Press one of the buttons to launch the Tornado SETUP program. You can change the following settings that you specified from Setup when you installed Tornado:

- Reconfigure Registry
- Register File Types
- Reconfigure Licensing

12.4 Customizing the Tools Menu

You can add entries to the Tools menu to allow easy access to additional tools. Before you add any commands in this part of the menu, Tornado displays the placeholder No Custom Tools as a disabled menu entry. The Customize command in the Tools menu allows you to add (or remove) entries at the end of the Tools menu.

12.4.1 The Customize Tools Dialog Box

Click Customize in the Tools menu to open the Customize Tools dialog box (Figure 12-11).

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. 416 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 12-12.)

- 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 12-3 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 12-4 lists macros for the project facility context.
Table 12-5 lists macros for the global Tornado context.
These macros are available for custom entries in the Build menu as well as for the Tools menu. For information on making them available though the Build menu, see 12.3.4 Project, p.405. For information on using them for custom builds, see 12.4.3 Customizing the Build Menu, p.421.
12.4.2 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 12-13 illustrates the specification for a ClearCase command to uncheckout a module.

Figure 12-13 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.416) 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 12-14), to permit specifying other arguments if necessary.
Alternate Editor

Figure 12-15 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.

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 12-16 illustrates the specification for a command to run the sizearch 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 sizearch in the correct directory.

World Wide Web

You can add a Tools command to link your Web browser directly to announcements from Wind River (and to related Internet resources). Figure 12-17 shows the specification for a Wind River Web Page command. (For a description of the information available on the Wind River home page, see 1.6 Customer Services, p.12.

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.
12.4.3 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 *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 12.3.4 Project, p.405).

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 12-18). 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.

---

**Figure 12-18** Dialog Box: Customize Builds

![Dialog Box: Customize Builds](image)
in your commands. See *Macros for Customized Menu Commands*, p.416 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.

---

### 12.5 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 Programmer’s Guide* or the online *Tornado API Reference*. 

---
HTML Help Initialization

The windHelp.tcl file allows you to specify a different HTML browser. The default is NetScape Communicator. To change the default, create windHelp.tcl with the following contents:

```tcl
set htmlBrowser "newbrowser -install"
```
Appendices
A.1 Introduction

Wind River 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.428.</td>
</tr>
<tr>
<td>Setup</td>
<td>Directory of the SETUP program, including the WIND subdirectory where PDF versions of documentation are located.</td>
</tr>
<tr>
<td>share</td>
<td>Directory of protocol definitions shared by both host and target software, including <code>agents\wtx</code>, <code>tgtsvr</code>, and <code>wtx</code> directories.</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.430.</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>
<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\include</td>
<td>Directory containing OS-specific header files.</td>
</tr>
<tr>
<td>host-os\jre118</td>
<td>Directory containing Java Runtime Environment-related files.</td>
</tr>
<tr>
<td>host-os\lib</td>
<td>Directory containing subroutine libraries for the Tornado tools. Subdirectories include backend (implementing the communications back ends available to the target server) and gcc-lib (containing the separate programs called by the GNU compiler driver).</td>
</tr>
<tr>
<td>host-os\host-type</td>
<td>Routines and scripts required by the host machine.</td>
</tr>
<tr>
<td>Directory/File</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>host-os\x-wrs-vxworks</td>
<td>Directory containing component programs and libraries for the GNU ToolKit configured for architecture x.</td>
</tr>
<tr>
<td>java</td>
<td>Directory containing various .jar files.</td>
</tr>
<tr>
<td>resource</td>
<td>Directory containing host-independent supporting files for the Tornado interactive tools.</td>
</tr>
<tr>
<td>resource\app-defaults</td>
<td>Directory containing default settings for the Tornado tools.</td>
</tr>
<tr>
<td>resource\bitmaps</td>
<td>Directory containing icons and button definitions for the Tornado tools.</td>
</tr>
<tr>
<td>resource\doctools</td>
<td>Directory containing tools for creating reference entries from source code.</td>
</tr>
<tr>
<td>resource\help</td>
<td>Directory containing online help.</td>
</tr>
<tr>
<td>resource\loader</td>
<td>Directory of object-module format information for the dynamic linker.</td>
</tr>
<tr>
<td>resource\synopsis</td>
<td>Directory containing entries that support WindSh command completion and look-up.</td>
</tr>
<tr>
<td>resource\target</td>
<td>Directory containing the target information database.</td>
</tr>
<tr>
<td>resource\tcl</td>
<td>Directory containing Tcl source code for the Tornado tools, including sub-directories with dialog descriptions and other definitions for each tool.</td>
</tr>
<tr>
<td>resource\test</td>
<td>Directory containing tests for the WTX protocol.</td>
</tr>
<tr>
<td>resource\vxdcom</td>
<td>Directory containing VxDCOM images and template.</td>
</tr>
<tr>
<td>resource\wdb</td>
<td>Mappings for WDB protocol extensions.</td>
</tr>
<tr>
<td>resource\userlock</td>
<td>Global authorization file for Tornado users.</td>
</tr>
<tr>
<td>src</td>
<td>Directory containing source for host utilities and examples, including demo (VxWorks sample programs) and windview (WindView and triggering sample programs).</td>
</tr>
<tr>
<td>tcl</td>
<td>Directory containing the standard Tcl and Tk distribution.</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>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td></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 all, which contains modules common to all implementations of VxWorks (system-independent modules), and a subdirectory for each port of VxWorks to specific target hardware (system-dependent modules).</td>
</tr>
<tr>
<td>config\all</td>
<td></td>
<td>Subdirectory containing system configuration modules. Note that this method of configuration has been replaced by the project facility (see 4. Projects).</td>
</tr>
<tr>
<td></td>
<td>bootInit.c</td>
<td>System-independent boot ROM facilities.</td>
</tr>
<tr>
<td></td>
<td>configAll.h</td>
<td>Generic header file used to define configuration parameters common to all targets.</td>
</tr>
<tr>
<td></td>
<td>usrConfig.c, bootConfig.c</td>
<td>Source of the configuration module for a VxWorks development system (usrConfig.c), and a configuration module for the VxWorks boot ROM (bootConfig.c).</td>
</tr>
<tr>
<td>config\bspname</td>
<td></td>
<td>Directory containing system-dependent modules for each port of VxWorks to a particular target CPU. Each of these directories includes the files listed below. In addition, drivers specific to each BSP are located here. See 4.4.4 Selecting the VxWorks Image Type, p.124.</td>
</tr>
</tbody>
</table>
### Directories and Files

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00bsp.cdf</td>
<td></td>
<td>Project facility configuration file that overrides the generic BSP components in <code>comps/vxWorks/00bsp.cdf</code> with BSP-specific versions of components and parameters.</td>
</tr>
<tr>
<td>00html.cdf</td>
<td></td>
<td>Project facility configuration file for HTML.</td>
</tr>
<tr>
<td>bootrom, bootrom.hex</td>
<td></td>
<td>VxWorks boot ROM code, as object module (<code>bootrom</code>) and as an ASCII file (<code>bootrom.hex</code>) in Motorola S-record format or Intel hex format (i960 targets), suitable for downloading over a serial connection to a PROM programmer. For more information see 4.8 Configuring and Building a VxWorks Boot Program, p. 145.</td>
</tr>
<tr>
<td>config.h</td>
<td></td>
<td>Header file of hardware configuration parameters.</td>
</tr>
<tr>
<td>bspname.h</td>
<td></td>
<td>Header file for the target board.</td>
</tr>
<tr>
<td>Makefile</td>
<td></td>
<td>Makefile for creating boot ROMs and the VxWorks system image for a particular target.</td>
</tr>
<tr>
<td>romInit.s</td>
<td></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>sysLib.c, sysALib.s</td>
<td></td>
<td>Two source modules of system-dependent routines.</td>
</tr>
<tr>
<td>sysSerial.c</td>
<td></td>
<td>Driver for on-board serial ports.</td>
</tr>
<tr>
<td>vxWorks, vxWorks.sym</td>
<td></td>
<td>Complete, linked VxWorks system binary (<code>vxWorks</code>), and its symbol table (<code>vxWorks.sym</code>) created with the supplied configuration files.</td>
</tr>
<tr>
<td>config\comps</td>
<td></td>
<td>Directory containing source and configuration files.</td>
</tr>
<tr>
<td>config\simulator</td>
<td></td>
<td>Directory containing configuration files and executables for the host simulator.</td>
</tr>
</tbody>
</table>
Table A-3  *installDir\target* (Continued)

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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 <em>inetLib</em>.</td>
</tr>
<tr>
<td>h\dhcp</td>
<td></td>
<td>Directory containing header files for use with dhcp.</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\resolv</td>
<td></td>
<td>Directory containing header files for use with name service.</td>
</tr>
<tr>
<td>h\rip</td>
<td></td>
<td>Directory containing header files for use with rip.</td>
</tr>
<tr>
<td>Directories</td>
<td>Files</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</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\tffs</td>
<td></td>
<td>Directory containing header files for use with TrueFFS.</td>
</tr>
<tr>
<td>h\tool</td>
<td></td>
<td>Directory containing header files for Diab and GNU.</td>
</tr>
<tr>
<td>h\types</td>
<td></td>
<td>Directory containing header files used for defining types.</td>
</tr>
<tr>
<td>h\wdb</td>
<td></td>
<td>Directory containing header files for use with WDB.</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\objcutool\test</td>
<td></td>
<td>Directory containing \texttt{vxColor.o}, a test program, for the specified target.</td>
</tr>
<tr>
<td>lib\archfamily</td>
<td></td>
<td>Directory containing VxWorks object modules as individual files (suitable for loading dynamically to the target).</td>
</tr>
<tr>
<td>lib/lib\cputool\vx.a</td>
<td></td>
<td>An archive (\texttt{ar}) format library containing the object modules that make up VxWorks.</td>
</tr>
<tr>
<td>proj</td>
<td></td>
<td>Directory containing the default VxWorks images and the default location for projects to be created.</td>
</tr>
<tr>
<td>src</td>
<td></td>
<td>Directory containing all source files for VxWorks.</td>
</tr>
<tr>
<td>src/arch</td>
<td></td>
<td>Directory containing makefiles to build source.</td>
</tr>
<tr>
<td>src/config</td>
<td></td>
<td>Directory containing files used to force inclusion of specific VxWorks modules.</td>
</tr>
</tbody>
</table>
Table A-3  \textit{installDir\textbackslash target} (Continued)

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansi_5_0.c</td>
<td>Used to include all 5.0 ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>assertInit.c</td>
<td>Used to include the \textit{assert} ANSI C library routine.</td>
<td></td>
</tr>
<tr>
<td>cplusdiabComplex.c, cplusdiabComplexIo.c, cplusdiabIos.c, cplusdiabIosLang.c, cplusdiabStl.c, cplusdiabString.c, cplusdiabStringIo.c, cplusgnuComplex.c, cplusgnuComplexIo.c, cplusgnuIos.c, cplusgnuIosLang.c, cplusgnuStl.c, cplusgnuString.c, cplusgnuStringIo.c</td>
<td>Used to include the various groups of C++ routines.</td>
<td></td>
</tr>
<tr>
<td>ctypeInit.c</td>
<td>Used to include the \textit{ctype} ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>localeInit.c</td>
<td>Used to include the \textit{locale} ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>mathInit.c</td>
<td>Used to include the \textit{math} ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>stdioInit.c</td>
<td>Used to include the \textit{stdio} ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>stdlibInit.c</td>
<td>Used to include the \textit{stdlib} ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>stringInit.c</td>
<td>Used to include the \textit{string} ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>timeInit.c</td>
<td>Used to include the \textit{time} ANSI C library routines.</td>
<td></td>
</tr>
<tr>
<td>usrAta.c</td>
<td>Used to include the ATA initialization routines.</td>
<td></td>
</tr>
<tr>
<td>Directories</td>
<td>Files</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>usrBreakpoint.c</td>
<td>Used to include the breakpoint management routines.</td>
<td></td>
</tr>
<tr>
<td>usrCplusTools.c</td>
<td>Placeholder files.</td>
<td></td>
</tr>
<tr>
<td>usrCplusVxw.c</td>
<td>Used to check module dependences for constants defined in <code>configAll.h</code> and <code>config.h</code>.</td>
<td></td>
</tr>
<tr>
<td>usrDepend.c</td>
<td>Used to activate <code>dsp</code> support.</td>
<td></td>
</tr>
<tr>
<td>usrDsp.c</td>
<td>Used to include extra modules that are needed by VxWorks but not referenced in the VxWorks code.</td>
<td></td>
</tr>
<tr>
<td>usrExtra.c</td>
<td>Used to mount a dosFs file system on a boot diskette (i386/i486 targets only).</td>
<td></td>
</tr>
<tr>
<td>usrFd.c</td>
<td>Used to mount a dosFs file system on a boot IDE hard disk drive (i386/i486 targets only).</td>
<td></td>
</tr>
<tr>
<td>usrKernel.c</td>
<td>Used to configure and initialize the <code>wind</code> kernel.</td>
<td></td>
</tr>
<tr>
<td>usrLoadSym.c</td>
<td>Used to load the VxWorks symbol table.</td>
<td></td>
</tr>
<tr>
<td>usrMmuInit.c</td>
<td>Used to initialize the memory management unit.</td>
<td></td>
</tr>
<tr>
<td>usrNetwork.c</td>
<td>Used to configure and initialize networking support.</td>
<td></td>
</tr>
<tr>
<td>usrPcmcia.c</td>
<td>Used to configure and initialize PCMCIA support.</td>
<td></td>
</tr>
<tr>
<td>usrScript.c</td>
<td>Used to execute a startup script when VxWorks first boots.</td>
<td></td>
</tr>
<tr>
<td>usrScsi.c</td>
<td>Used to configure and initialize SCSI support.</td>
<td></td>
</tr>
<tr>
<td>usrSmObj.c</td>
<td>Used to configure and initialize shared memory object support.</td>
<td></td>
</tr>
<tr>
<td>Directories</td>
<td>Files</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>usrTffs.c</td>
<td>Used to configure and initialize TrueFFS support.</td>
</tr>
<tr>
<td></td>
<td>usrWdb.c</td>
<td>Used to configure and initialize the Tornado target agent.</td>
</tr>
<tr>
<td></td>
<td>usrWindview.c</td>
<td>Used to configure and initialize WindView.</td>
</tr>
<tr>
<td>src\demo</td>
<td></td>
<td>Directory containing sample application modules for demonstration purposes,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>including both the source and the compiled object modules ready to be loaded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>into VxWorks.</td>
</tr>
<tr>
<td>src\demo\1</td>
<td></td>
<td>Directory containing a simple introductory demo program as well as a server/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>client socket demonstration.</td>
</tr>
<tr>
<td>src\demo\color</td>
<td></td>
<td>Directory containing the VxColor example application.</td>
</tr>
<tr>
<td>src\demo\cplusplus</td>
<td></td>
<td>Directory containing the factory example application.</td>
</tr>
<tr>
<td>src\demo\dg</td>
<td></td>
<td>Directory containing a simple datagram facility, useful for demonstrating</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and testing datagrams on VxWorks and/or other TCP/IP systems.</td>
</tr>
<tr>
<td>src\demo\start</td>
<td></td>
<td>Directory containing the program used with the Tornado Getting Started Guide</td>
</tr>
<tr>
<td>src\demo\wind</td>
<td></td>
<td>Directory containing the windDemo example application.</td>
</tr>
<tr>
<td>src\drv</td>
<td></td>
<td>Directory containing source code for supported board device drivers.</td>
</tr>
<tr>
<td>src\usr</td>
<td></td>
<td>Directory containing user-modifiable code. Not every file in the directory is</td>
</tr>
<tr>
<td></td>
<td>devSplit.c</td>
<td>Provides a routine to split the device name from a full path name.</td>
</tr>
</tbody>
</table>
### Table A-3  \textit{installDir}\textbackslash target (Continued)

<table>
<thead>
<tr>
<th>Directories</th>
<th>Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>Contains the makefile rules for building the vx library.</td>
<td></td>
</tr>
<tr>
<td>memDrv.c</td>
<td>Pseudo-device driver that allows memory to be accessed as a VxWorks character (non-block) device.</td>
<td></td>
</tr>
<tr>
<td>ramDiskCbio.c</td>
<td>RAM-disk driver with a CBIO interface which can be utilized directly by \textit{dosFsLib} without using \textit{dcacheCbio}.</td>
<td></td>
</tr>
<tr>
<td>ramDrv.c</td>
<td>Block device driver that allows memory to be used as a device with VxWorks local file systems.</td>
<td></td>
</tr>
<tr>
<td>usrLib.c</td>
<td>Library of routines designed for interactive invocation, which can be modified or extended if desired.</td>
<td></td>
</tr>
<tr>
<td>src\wdb</td>
<td>Directory containing target agent communication support.</td>
<td></td>
</tr>
<tr>
<td>wdbUdpLib.c</td>
<td>Implements communication methods for the target agent using a lightweight UDP/IP stack.</td>
<td></td>
</tr>
<tr>
<td>wdbUdpSockLib.c</td>
<td>Initializes UDP socket routines for the target agent.</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Directories</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>src\arch</td>
<td>Directory containing VxWorks source code for architecture-specific modules.</td>
</tr>
<tr>
<td>src\cplus</td>
<td>Directory containing source code for the Wind Foundation Classes.</td>
</tr>
<tr>
<td>src\libc</td>
<td>Directory containing the source files for the ANSI C library.</td>
</tr>
<tr>
<td>src\math</td>
<td>Directory containing the source files for various math routines (non-ANSI).</td>
</tr>
<tr>
<td>src\netwrs</td>
<td>Directory containing the source files for the VxWorks network subsystem modules.</td>
</tr>
</tbody>
</table>
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 wind 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 <code>CTcl</code>, p.447.</td>
</tr>
<tr>
<td>crosswind.tcl</td>
<td>Optional Tcl initialization code for the debugger front end. See <code>10.8 Tcl: Debugger Customization</code>, p.372.</td>
</tr>
<tr>
<td>gdb.tcl</td>
<td>Optional Tcl initialization code for the debugging engine itself. See <code>10.8 Tcl: Debugger Customization</code>, p.372.</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 <code>7.7.3 Tcl: Tornado Shell Initialization</code>, p.274.</td>
</tr>
<tr>
<td>wtxtcl.tcl</td>
<td>Optional Tcl initialization code for <code>wtxtcl</code>, the Tcl interpreter with WTX-protocol extensions. See the Tornado API Programmer’s Guide: Extending Tornado Tools.</td>
</tr>
</tbody>
</table>

### Table A-5 wind 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 <code>1.6 Customer Services</code>, p.12.</td>
</tr>
<tr>
<td>.wind/tsr</td>
<td>A directory recording the history of your Tornado support requests. This information is managed through the Support command in the Tools menu; see <code>1.6 Customer Services</code>, p.12.</td>
</tr>
</tbody>
</table>
B.1 Introduction

Each BSP has a makefile for building VxWorks. This makefile, called Makefile, is abbreviated to declare only the basic information needed to build VxWorks with the BSP. The makefile includes other files to provide target and VxWorks specific rules and dependencies. In particular, a file of the form depend.bspname is included. The first time that make is run in the BSP directory, it creates the depend.bspname file.

The Makefile in the BSP directory is used only when building from the traditional command line. It is not used when building projects from the project tool. Each build option for a project has its own makefile that the tool uses to build the project modules.

When projects are created from a BSP, the BSP makefile is scanned once and the make parameters are captured into the project. Any changes made to the BSP makefile after a project has been created do not affect that project. Only projects built from the BSP after the change is made are affected.

B.2 Customizing the VxWorks Makefile

The BSP makefile provides several mechanisms for configuring the VxWorks build. Although VxWorks configuration is more commonly controlled at compile-time by macros in configAll.h and bspname/config.h.
Most of the makefile macros fall into two categories: macros intended for use by the BSP developer, and macros intended for use by the end user. When building a VxWorks image, the needs of these two audiences differ considerably. Maintaining two separate compile-time macro sets lets the `make` separate the BSP-specific requirements from user-specific requirements.

Macros containing `EXTRA` in their name are intended for use by the BSP developer to specify additional object modules that must be compiled and linked with all VxWorks images.

Macros containing `ADDED` in their name are intended for use by the end-user on the `make` command line. This allows for easy compile time options to be specified by the user, without having to repeat any BSP-specific options in the same macro declaration.

### B.3 Commonly Used Makefile Macros

Of the 135 or so makefile macros, this document discusses only the most commonly used.

**MACH_EXTRA**

You can add an object module to VxWorks by adding its name to the skeletal makefile. To include `fooLib.o`, for example, add it to the `MACH_EXTRA` definition in `Makefile`. This macro causes the linker to link it into the output object.

Finally, regenerate VxWorks with `make`. The module will now be included in all future VxWorks builds. If necessary, the module will be made from `fooLib.c` or `fooLib.s` using the `.c.o` or `.s.o` makefile rule.

**MACH_EXTRA** can be used for drivers that are not included in the VxWorks driver library. BSPs do not usually include source code for Ethernet and SCSI device drivers; thus, when preparing your system for distribution, omit the driver source file and change the object file’s name from `.o` to `.obj` (update the makefiles, too).

Now the end user can build VxWorks without the driver source, and `rm *.o` will not inadvertently remove the driver’s object file.
LIB_EXTRA

The LIB_EXTRA makefile variable makes it possible to include library archives in the VxWorks build without altering the standard VxWorks archive or the driver library archive. Define LIB_EXTRA in Makefile to indicate the location of the extra libraries.

The libraries specified by LIB_EXTRA are provided to the link editor when building any VxWorks or boot ROM images. This is useful for third-party developers who want to supply end users with network or SCSI drivers, or other modules in object form, and find that the MACH_EXTRA mechanism described earlier in this chapter does not suit their needs.

The extra libraries are searched first, before Wind River libraries, and any references to VxWorks symbols are resolved properly.

EXTRA_INCLUDE

The makefile variable EXTRA_INCLUDE is available for specifying additional header directory locations. This is useful when the user or BSP provider has a separate directory of header files to be used in addition to the normal directory locations.

EXTRA_INCLUDE = -I../myHdrs

The normal order of directory searching for #include directives is:

$(INCLUDE_CC) (reserved for compiler specific uses)
$(EXTRA_INCLUDE)
.
$(CONFIG_ALL)
$(TGT_DIR)/h
$(TGT_DIR)/src/config
$(TGT_DIR)/src/drv

EXTRA_DEFINE

The makefile variable EXTRA_DEFINE is available for specifying compile time macros required for building a specific BSP. In the following example the macro BRD_TYPE is given the value MB934. This macro is defined on the command line for all compiler operations.

EXTRA_DEFINE = -DBRD_TYPE=MB934
By default a minimum set of macro names are defined on the compiler command line. This is primarily used to pass the same memory addresses used in both the compiling and linking operations.

These default macro definitions include:

```bash
-DCPU=$(CPU)
```

**ADDED_CFLAGS**

Sometimes it is inconvenient to modify `config.h` to control VxWorks configuration. **ADDED_CFLAGS** is useful for defining macros without modifying any source code.

Consider the hypothetical Acme XYZ-75 BSP that supports two hardware configurations. The XYZ-75 has a daughter board interface, and in this interface either a Galaxy-A or a Galaxy-B module is installed. The drivers for the modules are found in the directory `src/drv/multi`.

The macro `GALAXY_C_FILE` determines which driver to include at compile-time. The file named by `GALAXY_C_FILE` is `#include`d by `sysLib.c`.

The default configuration (Galaxy-A module installed) is established in `config.h`:

```bash
#ifndef GALAXY_C_FILE
#define GALAXY_C_FILE "multi/acmeGalaxyA.c"
#endif /* GALAXY_C_FILE */
```

When `make` is called normally, VxWorks supports the XYZ-75 with the Galaxy-A module installed. To override the default and build VxWorks for the XYZ-75/Galaxy-B configuration, do the following:

```
% make ADDED_CFLAGS='-DGALAXY_C_FILE="multi/acmeGalaxyB.c"' vxWorks
```

For ease of use, you can encapsulate the lengthy command line within a shell script or independent makefile.

To ensure that a module is incorporated in `vxWorks`, remove the module’s object file and `vxWorks` before running `make`.

**ADDED_MODULES**

The `ADDED_MODULES` makefile variable makes it possible to add modules to VxWorks without modifying any source code.

While `MACH_EXTRA` requires the makefile to be modified, `ADDED_MODULES` allows one or more extra modules to be specified on the `make` command line. For
example, to build VxWorks with the BSP VTS support library included, copy
pkLib.c to the target directory and enter:

```bash
% make ADDED_MODULES=pkLib.o vxWorks
```

One disadvantage of using `ADDED_MODULES` is that makefile dependencies are
not generated for the module(s). In the above example, if `pkLib.c`, `pkLib.o`, and
`vxWorks` already exist, you must remove `pkLib.o` and `vxWorks` before running
`make` to force the latest `pkLib.c` to be incorporated into `vxWorks`.

**CONFIG_ALL**

Under extreme circumstances, the files in the `config/all` directory might not be
flexible enough to support a complex BSP. In this case, copy all the `config/all` files
to the BSP directory (`config/bspname`) and edit the files as necessary. Then redefine
the `CONFIG_ALL` makefile variable in `Makefile` to direct the build to the altered
files. To do this, define `CONFIG_ALL` to equal the absolute path to the BSP’s
`config/bspname` directory as shown in the following example:

```
CONFIG_ALL = $(TGT_DIR)/config/bspname/
```

The procedure described above works well if you must modify all or nearly all the
files in `config/all`. However, if you know that only one or two files from `config/all`
need modification, you can copy just those files to the BSP’s `config/bspname`
directory. Then, instead of changing the `CONFIG_ALL` makefile macro, change one
or more of the following (which ever are appropriate).

**USRCONFIG**

The path to an alternate `config/all/usrConfig.c` file.

**BOOTCONFIG**

The path to an alternate `config/all/bootConfig.c` file.

**BOOTINIT**

The path to an alternate `config/all/bootInit.c` file.

**DATASEGPAK**

The path to an alternate `config/all/dataSegPad.s` file.

**CONFIG_ALL_H**

The path to an alternate `config/all/configAll.h` file.

**TGT_DIR**

The path to the target directory tree, normally `$(WIND_BASE)/target`.
COMPRESS
The path to the host’s compression program. This is the program that
compresses an executable image. The binary image is input through stdin, and
the output is placed on the stdout device. This macro can contain
command-line flags for the program if necessary.

BINHEX
The path to the host’s object-format-to-hex program. This program is called
using HEX_FLAGS as command line flags. See target/h/make/rules.bsp for
actual calling sequence.

HEX_FLAGS
Command line flags for the $(BINHEX) program.

BOOT_EXTRA
Additional modules to be linked with compressed ROM images. These
modules are not linked with uncompressed or ROM-resident images, just
compressed images.

EXTRA_DOC_FLAGS
Additional preprocessor flags for making man pages. The default
documentation flags are -DDOC -DINCLUDE_SCSI. If EXTRA_DOC_FLAGS is
defined, these flags are passed to the man page generation routines in addition
to the default flags.
C
Tcl

C.1 Why Tcl?

Tcl is a scripting language which is designed to be embedded in applications. It can be embedded in applications that present command-line interfaces (the Tornado shell, for example) as well as in those that do not (such as the browser). Almost any program can benefit from the inclusion of such a language, because it provides a way for users to combine the program’s features in new and unforeseen ways to meet their own needs. Many programs implement a command-line interface that is unique to the particular application. However, application-specific command line interfaces often have weak languages. Tcl holds some promise of unifying application command languages. This has an additional benefit: the more programs use a common language, the easier it is for everyone to learn to use each additional program that incorporates the language.

To encourage widespread adoption, John Ousterhout (the creator of Tcl) has placed the language and its implementation in the public domain.

Tk is often mentioned in conjunction with Tcl. Tk is a graphics library that extends Tcl with graphical-interface facilities. Tornado does not currently use Tk, but you may find Tk useful for your own Tcl applications.
C.2 Introduction to Tcl

Tcl represents all data as ordinary text strings. As you might expect, the string-handling features of Tcl are particularly strong. However, Tcl also provides a full complement of C-like arithmetic operators to manipulate strings that represent numbers.

The examples in the following sections exhibit some of the fundamental mechanisms of the Tcl language, in order to provide some of the flavor of working in Tcl. However, this is only an introduction.

For documentation on all Tcl interfaces in Tornado (as well as on C interfaces), see the Tornado API Programmer’s Guide from Wind River.

For the Tcl language itself, the following generally available books are helpful:

- Welch, Brent: Practical Programming in Tcl and Tk (Prentice Hall, 1995) – Useful both as a quick Tcl reference and as a tutorial.

C.2.1 Tcl Variables

The Tcl set command defines variables. Its result is the current value of the variable, as shown in the following examples:

<table>
<thead>
<tr>
<th>Table C-1 Setting Tcl Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tcl Expression</strong></td>
</tr>
<tr>
<td>set num 6</td>
</tr>
<tr>
<td>set y hello</td>
</tr>
<tr>
<td>set z &quot;hello world&quot;</td>
</tr>
<tr>
<td>set t $z</td>
</tr>
<tr>
<td>set u &quot;$z $y&quot;</td>
</tr>
<tr>
<td>set v {($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>
<thead>
<tr>
<th>Table C-2</th>
<th>Evaluating Tcl Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcl Expression</td>
<td>Result</td>
</tr>
<tr>
<td>set num</td>
<td>6</td>
</tr>
<tr>
<td>set z</td>
<td>hello world</td>
</tr>
</tbody>
</table>

### C.2.2 Lists in Tcl

Tcl provides special facilities for manipulating lists. In Tcl, a list is just a string, with the list elements delimited by spaces, as shown in the following examples:

<table>
<thead>
<tr>
<th>Table C-3</th>
<th>Using Tcl Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcl Expression</td>
<td>Result</td>
</tr>
<tr>
<td>llength $v</td>
<td>2</td>
</tr>
<tr>
<td>lindex $u 1</td>
<td>world</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>
</tr>
</tbody>
</table>
This powerful technique, especially combined with recursive command substitution (see C.2.4 Command Substitution, p.451), can provide a little of the flavor of Lisp in Tcl programs.

### C.2.3 Associative Arrays

Tcl arrays are all associative arrays, using a parenthesized key to select or define a particular element of an array: `arrayName(keyString)`. The `keyString` may in fact represent a number, giving the effect of ordinary indexed arrays. The following are some examples of expressions involving Tcl arrays:

<table>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set taskId(tNetTask) 0x4f300</td>
<td>Get element tNetTask of array taskId.</td>
<td></td>
</tr>
<tr>
<td>set cpuFamily(5) m68k</td>
<td>Define array cpuFamily and an element keyed 5.</td>
<td></td>
</tr>
<tr>
<td>set cpuFamily(10) sparc</td>
<td>Define element keyed 10 of array cpuFamily.</td>
<td></td>
</tr>
<tr>
<td>set cpuId 10 10</td>
<td>Define cpuId, and use it as a key to cpuFamily.</td>
<td></td>
</tr>
<tr>
<td>set cpuFamily($cpuId) sparc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The last examples use curly braces to delimit list items, yielding “lists of lists.”
C.2.4 Command Substitution

In Tcl, you can capture the result of the command as text by enclosing the command in square brackets [ … ]. The Tcl interpreter substitutes the command result in the same process that is already running, which makes this an efficient operation.

The last example selects from a list of lists (defined among the examples in C.2.2 Lists in Tcl, p. 449). This and the previous example show that you can nest Tcl command substitutions readily. The Tcl interpreter substitutes the most deeply nested command, then continues substituting recursively until it can evaluate the outermost command.

C.2.5 Arithmetic

Tcl has an `expr` command to evaluate arithmetic expressions. The `expr` command understands numbers in decimal and hexadecimal, as in the following examples:

<table>
<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>
**C.2.6 I/O, Files, and Formatting**

Tcl includes many commands for working with files and for formatted I/O. Tcl also has many facilities for interrogating file directories and attributes. The following examples illustrate some of the possibilities:

<table>
<thead>
<tr>
<th>Tcl Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set myfile [open myfile.out w]</td>
<td>Open a file for writing.</td>
</tr>
<tr>
<td>puts $myfile [format &quot;%s %d\n&quot; \</td>
<td>Format a string and write it to file.</td>
</tr>
<tr>
<td>&quot;you are number&quot; [expr 3+3]]</td>
<td></td>
</tr>
<tr>
<td>close $myfile</td>
<td>Close the file.</td>
</tr>
<tr>
<td>file exists myfile.out</td>
<td>1</td>
</tr>
<tr>
<td>file writable myfile.out</td>
<td>1</td>
</tr>
<tr>
<td>file executable myfile.out</td>
<td>0</td>
</tr>
<tr>
<td>glob *.o</td>
<td>testCall.o foo.o bar.o</td>
</tr>
</tbody>
</table>

**C.2.7 Procedures**

Procedure definition in Tcl is straightforward, and resembles many other languages. The command `proc` builds a procedure from its arguments, which give the procedure name, a list of its arguments, and a sequence of statements for the procedure body. In the body, the `return` command specifies the result of the procedure. For example, the following defines a procedure to compute the square of a number:

```tcl
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:

```tcl
proc sum {args} {
    set accum 0
    foreach item $args {
        incr accum $item
    }
}
Defined Tcl procedures are called by name, and can be used just like any other Tcl command. The following examples illustrate some possibilities:

```tcl
proc shEventDispatch {event} {
    set handlerProc "[lindex $event 0]_Handler"
    if {[info procs $handlerProc] != ""} {
        $handlerProc $event
    } {
        #event has no handler--do nothing.
    }
}
```

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.
    }
}
```

### C.2.8 Control Structures

Tcl provides all the popular control structures: conditionals (if), loops (while, for, and foreach), case statements (switch), and explicit variable-scope control (global, upvar, and uplevel variable declarations). By using these facilities, you can even define your own control structures. While there is nothing mysterious about these
facilities, more detailed descriptions are beyond the scope of this summary. For detailed information, see the books cited at the beginning of C.2 Introduction to Tcl, p.448.

C.2.9 Tcl Error Handling

Every Tcl procedure, whether built-in or script, normally returns a string. Tcl procedures may signal an error instead: in a defined procedure, this is done with the error command. This starts a process called unwinding. When a procedure signals an error, it passes to its caller a string containing information about the error. Control is passed to the calling procedure. If that procedure did not provide for this possibility by using the Tcl catch command, control is passed to its caller in turn. This recursive unwinding continues until the top level, the Tcl interpreter, is reached.

As control is passed along, any procedure can catch the error and take one of two actions: signal another error and provide error information, or work around the error and return as usual, ending the unwinding process.

At each unwinding step, the Tcl interpreter adds a description of the current execution context to the Tcl variable errorInfo. After unwinding ends, you can display errorInfo to trace error information. Another variable, errorCode, may contain diagnostic information, such as an operating system dependent error code returned by a system call.

C.2.10 Integrating Tcl and C Applications

Tcl is designed to integrate with C applications. The Tcl interpreter itself is distributed as a library, ready to link with other applications. The core of the Tcl integration strategy is to allow each application to add its own commands to the Tcl language. This is accomplished primarily through the subroutine Tcl_CreateCommand() in the Tcl interpreter library, which associates a new Tcl command name and a pointer to an application-specific routine. For more details, consult the Tcl books cited at the beginning of C.2 Introduction to Tcl, p.448.
D.1 Introduction

This document defines the Wind River standard for all C code and for the accompanying documentation included in source code. The conventions are intended, in part, to encourage higher quality code; every source module is required to have certain essential documentation, and the code and documentation is required to be in a format that has been found to be readable and accessible.

The conventions are also intended to provide a level of uniformity in the code produced by different programmers. Uniformity allows programmers to work on code written by others with less overhead in adjusting to stylistic differences. Also it allows automated processing of the source; tools can be written to generate reference entries, module summaries, change reports, and so on.

The conventions described here are grouped as follows:

- **File Headings.** Regardless of the programming language, a single convention specifies a heading at the top of every source file.
- **C Coding Conventions**
- **TCL Coding Conventions.**
Every file containing C code—whether it is a header file, a resource file, or a file that implements a host tool, a library of routines, or an application—must contain a **standard file heading**. The conventions in this section define the standard for the heading that must come at the beginning of every source file.

The file heading consists of the blocks described below. The blocks are separated by one or more empty lines and contain no empty lines within the block. This facilitates automated processing of the heading.

- **Title**: The title consists of a one-line comment containing the tool, library, or application name followed by a short description. The name must be the same as the file name. This line will become the title of automatically generated reference entries and indexes.

- **Copyright**: The copyright consists of a single-line comment containing the appropriate copyright information.

- **Modification History**: The modification history consists of a comment block: in C, a multi-line comment. Each entry in the modification history consists of the version number, date of modification, initials of the programmer who made the change, and a complete description of the change. If the modification fixes an SPR, then the modification history must include the SPR number.

The version number is a two-digit number and a letter (for example, 03c). The letter is incremented for internal changes, and the number is incremented for large changes, especially those that materially affect the module’s external interface.

The following example shows a standard file heading from a C source file:

```
Example D-1 Standard File Heading (C Version)

/* fooLib.c - foo subroutine library */

/* Copyright 1984-1995 Wind River Systems, Inc. */

/*
 modification history
 -------------------
 02a,15sep92,nfs added defines MAX_FOOS and MIN_FATS.
 01b,15feb86,dnw added routines fooGet() and fooPut();
     added check for invalid index in fooFind().
 01a,10feb86,dnw written.
 */
```
D.3 C Coding Conventions

These conventions are divided into the following categories:

- Module Layout
- Subroutine Layout
- Code Layout
- Naming Conventions
- Style
- Header File Layout
- Documentation Generation

D.3.1 C Module Layout

A module is any unit of code that resides in a single source file. The conventions in this section define the standard module heading that must come at the beginning of every source module following the standard file heading. The module heading consists of the blocks described below; the blocks should be separated by one or more blank lines.

After the modification history and before the first function or executable code of the module, the following sections are included in the following order, if appropriate:

- **General Module Documentation:** The module documentation is a C comment consisting of a complete description of the overall module purpose and function, especially the external interface. The description includes the heading INCLUDE FILES: followed by a list of relevant header files.

- **Includes:** The includes block consists of a one-line C comment containing the word includes followed by one or more C pre-processor #include directives. This block groups all header files included in the module in one place.

- **Defines:** The defines block consists of a one-line C comment containing the word defines followed by one or more C pre-processor #define directives. This block groups all definitions made in the module in one place.

- **Typedefs:** The typedefs block consists of a one-line C comment containing the word typedefs followed by one or more C typedef statements, one per line. This block groups all type definitions made in the module in one place.

- **Globals:** The globals block consists of a one-line C comment containing the word globals followed by one or more C declarations, one per line. This block
groups together all declarations in the module that are intended to be visible outside the module.

- **Locals:** The locals block consists of a one-line C comment containing the word *locals* followed by one or more C declarations, one per line. This block groups together all declarations in the module that are intended not to be visible outside the module.

- **Forward Declarations:** The forward declarations block consists of a one-line C comment containing the words *forward declarations* followed by one or more ANSI C function prototypes, one per line. This block groups together all the function prototype definitions required in the module. Forward declarations must only apply to local functions; other types of functions belong in a header file.

The format of these blocks is shown in the following file example (which also includes the file heading specified earlier).

Example D-2  
**C File and Module Headings**

```c
/* fooLib.c - foo subroutine library */

/* Copyright 1984-1995 Wind River Systems, Inc. */

/*
 modification history
---------------------
 02a,15sep92,nfs added defines MAX_FOOS and MIN_FATS.
 01b,15feb86,dnw added routines fooGet() and fooPut();
            added check for invalid index in fooFind().
 01a,10feb86,dnw written.
*/

/* DESCRIPTION
This module is an example of the Wind River Systems C coding conventions.
*/

*/ INCLUDE FILES: fooLib.h */

*/ includes */

#include "vxWorks.h"
#include "fooLib.h"

*/ defines */
#define MAX_FOOS 112 /* max # of foo entries */
#define MIN_FATS 2   /* min # of FAT copies */

*/ typedefs */
```
typedef struct fooMsg /* FOO_MSG */
{
    VOIDFUNCPTTR func; /* pointer to function to invoke */
    int arg [FOO_MAX_ARGS]; /* args for function */
} FOO_MSG;

/* globals */
char * pGlobalFoo; /* global foo table */

/* locals */
LOCAL int numFoosLost; /* count of foos lost */

/* forward declarations */
LOCAL int fooMat (list * aList, int fooBar, BOOL doFoo);
FOO_MSG fooNext (void);
STATUS fooPut (FOO_MSG inPar);

D.3.2 C Subroutine Layout

The following conventions define the standard layout for every subroutine.

Each subroutine is preceded by a C comment heading consisting of documentation
that includes the following blocks. There should be no blank lines in the heading,
but each block should be separated with a line containing a single asterisk (*) in the
first column.

- **Banner**: This is the start of a C comment and consists of a slash character (/)
followed by 75 asterisks (*) across the page.

- **Title**: One line containing the routine name followed by a short, one-line
description. The routine name in the title must match the declared routine
name. This line becomes the title of automatically generated reference entries
and indexes.

- **Description**: A full description of what the routine does and how to use it.

- **Returns**: The word RETURNS: followed by a description of the possible result
values of the subroutine. If there is no return value (as in the case of routines
declared void), enter:

  RETURNS: N/A

  Mention only true returns in this section—not values copied to a buffer given
as an argument.
• **Error Number:** The word **ERRNO:** followed by all possible **errno** values returned by the function. No description of the **errno** value is given, only the **errno** value and only in the form of a defined constant.\(^1\)

The subroutine documentation heading is terminated by the C end-of-comment character (`*/`), which must appear on a single line, starting in column one.

The subroutine declaration immediately follows the subroutine heading.\(^2\) The format of the subroutine and parameter declarations is shown in *D.3.3 C Declaration Formats*, p.460.

Example D-3  **Standard C Subroutine Layout:**

```c
/****************************
 * fooGet - get an element from a foo
 *
 * This routine finds the element of a specified index in a specified foo. The value of the element found is copied to <pValue>.
 *
 * RETURNS: OK, or ERROR if the element is not found.
 *
 * **ERRNO:**
 *  S_fooLib_BLAH
 *  S_fooLib_GRONK
 */

STATUS fooGet
{
    FOO foo,     /* foo in which to find element */
    int index,   /* element to be found in foo */
    int * pValue /* where to put value */
}
```

---

**D.3.3 C Declaration Formats**

Include only one declaration per line. Declarations are indented in accordance with *Indentation*, p.464, and are typed at the current indentation level.

The rest of this section describes the declaration formats for variables and subroutines.

---

1. A list containing the definitions of each **errno** is maintained and documented separately.
2. The declaration is used in the automatic generation of reference entries.
Variables

- For basic type variables, the type appears first on the line and is separated from the identifier by a tab. Complete the declaration with a meaningful one-line comment. For example:

```c
unsigned rootMemNBytes; /* memory for TCB and root stack */
int rootTaskId;       /* root task ID */
BOOL roundRobinOn;    /* boolean for round-robin mode */
```

- The * and ** pointer declarators belong with the type. For example:

```c
FOO_NODE * pFooNode;     /* foo node pointer */
FOO_NODE ** ppFooNode;   /* pointer to the foo node pointer */
```

- Structures are formatted as follows: the keyword `struct` appears on the first line with the structure tag. The opening brace appears on the next line, followed by the elements of the structure. Each structure element is placed on a separate line with the appropriate indentation and comment. If necessary, the comments can extend over more than one line; see Comments, p.466, for details. The declaration is concluded by a line containing the closing brace, the type name, and the ending semicolon. Always define structures (and unions) with a `typedef` declaration, and always include the structure tag as well as the type name. Never use a structure (or union) definition to declare a variable directly. The following is an example of acceptable style:

```c
typedef struct symtab /* SYMTAB - symbol table */
{
    OBJ_CORE   objCore;     /* object maintenance */
    HASH_ID    nameHashId;  /* hash table for names */
    SEMAPHORE  symMutex;    /* symbol table mutual exclusion sem */
    PART_ID    symPartId;   /* memory partition id for symbols */
    BOOL       sameNameOk;  /* symbol table name clash policy */
    int        nSymbols;    /* current number of symbols in table */
} SYMTAB;
```

This format is used for other composite type declarations such as `union` and `enum`.

The exception to never using a structure definition to declare a variable directly is structure definitions that contain pointers to structures, which effectively declare another `typedef`. This exception allows structures to store pointers to related structures without requiring the inclusion of a header that defines the type.

For example, the following compiles without including the header that defines `struct fooInfo` (so long as the surrounding code never delves inside this structure):
CORRECT:

```c
typedef struct tcbInfo
{
    struct fooInfo * pfooInfo;
    ...
} TCB_INFO;
```

By contrast, the following cannot compile without including a header file to define the type `FOO_INFO`:

INCORRECT:

```c
typedef struct tcbInfo
{
    FOO_INFO * pfooInfo;
    ...
} TCB_INFO;
```

**Subroutines**

There are two formats for subroutine declarations, depending on whether the subroutine takes arguments.

- For subroutines that take arguments, the subroutine return type and name appear on the first line, the opening parenthesis on the next, followed by the arguments to the routine, each on a separate line. The declaration is concluded by a line containing the closing parenthesis. For example:

```c
int lstFind
(
    LIST * pList, /* list in which to search */
    NODE * pNode /* pointer to node to search for */
)
```

- For subroutines that take no parameters, the word `void` in parentheses is required and appears on the same line as the subroutine return type and name. For example:

```c
STATUS fppProbe (void)
```
D.3.4 C Code Layout

The maximum length for any line of code is 80 characters.

The rest of this section describes the conventions for the graphic layout of C code, and covers the following elements:

- vertical spacing
- horizontal spacing
- indentation
- comments

**Vertical Spacing**

- Use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines.
- Do not put more than one declaration on a line. Each variable and function argument must be declared on a separate line. Do not use comma-separated lists to declare multiple identifiers.
- Do not put more than one statement on a line. The only exceptions are the `for` statement, where the initial, conditional, and loop statements can go on a single line:

```
for (i = 0; i < count; i++)
```

or the `switch` statement if the actions are short and nearly identical (see the `switch` statement format in Indentation, p.482).

The `if` statement is not an exception: the executed statement always goes on a separate line from the conditional expression:

```
if (i > count)
  i = count;
```

- Braces ({ and }) and `case` labels always have their own line.

**Horizontal Spacing**

- Put spaces around binary operators, after commas, and before an open parenthesis. Do not put spaces around structure members and pointer operators. Put spaces before open brackets of array subscripts; however, if a
subscript is only one or two characters long, the space can be omitted. For example:

```c
status = fooGet (foo, i + 3, &value);
foo.index
pFoo->index
fooArray [(max + min) / 2]
string[0]
```

- Line up continuation lines with the part of the preceding line they continue:

```c
a = (b + c) *
   (d + e);
status = fooList (foo, a, b, c,
                  d, e);
if ((a == b) &&
    (c == d))
...```

**Indentation**

- Indentation levels are every four characters (columns 1, 5, 9, 13, ...).
- The module and subroutine headings and the subroutine declarations start in column one.
- Indent one indentation level after:
  - subroutine declarations
  - conditionals (see below)
  - looping constructs
  - switch statements
  - case labels
  - structure definitions in a `typedef`
- The `else` of a conditional has the same indentation as the corresponding `if`. Thus the form of the conditional is:

```c
if ( condition )
{
    statements
}
else
{
    statements
}
```
The form of the conditional statement with an `else if` is:

```c
if ( condition )
{
    statements
}
else if ( condition )
{
    statements
}
else
{
    statements
}
```

The general form of the `switch` statement is:

```c
switch ( input )
{
    case 'a':
        ...    // Here can be a very short statement
        break;
    case 'b':
        ...    // Here can be a very short statement
        break;
    default:
        ...    // Here can be a very short statement
        break;
}
```

If the actions are very short and nearly identical in all cases, an alternate form of the switch statement is acceptable:

```c
switch ( input )
{
    case 'a': x = aVar; break;
    case 'b': x = bVar; break;
    case 'c': x = cVar; break;
    default: x = defaultVar; break;
}
```

- Comments have the same indentation level as the section of code to which they refer (see Comments, p.484).
- Section braces (`{` and `}`) have the same indentation as the code they enclose.
Comments

- Place comments within code so that they precede the section of code to which they refer and have the same level of indentation. Separate such comments from the code by a single blank line.
  - Begin single-line comments with the open-comment and end with the close-comment, as in the following:
    ```
    /* This is the correct format for a single-line comment */
    foo = MAX_FOO;
    ```
  - Begin and end multi-line comments with the open-comment and close-comment on separate lines, and precede each line of the comment with an asterisk (*), as in the following:
    ```
    /*
    * This is the correct format for a multiline comment
    * in a section of code.
    */
    foo = MIN_FOO;
    ```
- Compose multi-line comments in declarations and at the end of code statements with one or more one-line comments, opened and closed on the same line. For example:
  ```
  int foo
  {
    int a, /* this is the correct format for a */
      /* multiline comment in a declaration */
    BOOL b /* standard comment at the end of a line */
  }

  { /* when necessary, a comment about a line */
    day = night; /* of code can be done this way */
  }
  ```

D.3.5 C Naming Conventions

The following conventions define the standards for naming modules, routines, variables, constants, macros, types, and structure and union members. The purpose of these conventions is uniformity and readability of code.
When creating names, remember that the code is written only once, but read many times. Assign names that are meaningful and readable; avoid obscure abbreviations.

Names of routines, variables, and structure and union members are composed of upper- and lowercase characters and no underbars. Capitalize each “word” except the first:

aVariableName

Names of defined types (defined with `typedef`), and constants and macros (defined with `#define`), are all uppercase with underbars separating the words in the name:

A_CONSTANT_VALUE

Every module has a short prefix (two to five characters). The prefix is attached to the module name and all externally available routines, variables, constants, macros, and `typedefs`. (Names not available externally do not follow this convention.)

fooLib.c module name
fooObjFind subroutine name
fooCount variable name
FOO_MAX_COUNT constant
FOO_NODE type

Names of routines follow the `module-noun-verb` rule. Start the routine name with the module prefix, followed by the noun or object that the routine manipulates. Conclude the name with the verb or action the routine performs:

fooObjFind foo - object - find
sysNvRamGet system - NVRAM - get
taskSwitchHookAdd task - switch hook - add

Every header file defines a preprocessor symbol that prevents the file from being included more than once. This symbol is formed from the header file name by prefixing `__INC` and removing the dot (`.`). For example, if the header file is called `fooLib.h`, the `multiple inclusion guard symbol` is:

`__INCfooLibh`

Pointer variable names have the prefix `p` for each level of indirection. For example:

`FOO_NODE * pFooNode;
FOO_NODE ** ppFooNode;
FOO_NODE *** pppFooNode;`
D.3.6 C Style

The following conventions define additional standards of programming style:

- **Comments**: Insufficiently commented code is unacceptable.
- **Numeric Constants**: Use `#define` to define meaningful names for constants. Do not use numeric constants in code or declarations (except for obvious uses of small constants like 0 and 1).
- **Boolean Tests**: Do not test non-booleans as you test a boolean. For example, where `x` is an integer:
  
  **CORRECT**:
  ```c
  if (x == 0)
  ```
  
  **INCORRECT**:
  ```c
  if (!x)
  ```
  
  Similarly, do not test booleans as non-booleans. For example, where `libInstalled` is declared as `BOOL`:
  
  **CORRECT**:
  ```c
  if (libInstalled)
  ```
  
  **INCORRECT**:
  ```c
  if (libInstalled == TRUE)
  ```

- **Private Interfaces**: Private interfaces are functions and data that are internal to an application or library and do not form part of the intended external user interface. Place private interfaces in a header file residing in a directory named `private`. End the name of the header file with an uppercase `P` (for `private`). For example, the private function prototypes and data for the commonly used internal functions in the library `blahLib` would be placed in the file `private/blahLibP.h`.

- **Passing and Returning Structures**: Always pass and return pointers to structures. Never pass or return structures directly.

- **Return Status Values**: Routines that return status values should return either `OK` or `ERROR` (defined in `vxWorks.h`). The specific type of error is identified by setting `errno`. Routines that do not return any values should return `void`.

- **Use Defined Names**: Use the names defined in `vxWorks.h` wherever possible. In particular, note the following definitions:
- Use **TRUE** and **FALSE** for boolean assignment.
- Use **EOS** for end-of-string tests.
- Use **NULL** for zero pointer tests.
- Use **IMPORT** for **extern** variables.
- Use **LOCAL** for **static** variables.
- Use **FUNCPTR** or **VOIDFUNCPTR** for pointer-to-function types.

### D.3.7 C Header File Layout

Header files, denoted by a .h extension, contain definitions of status codes, type definitions, function prototypes, and other declarations that are to be used (through `#include`) by one or more modules. In common with other files, header files must have a **standard file heading** at the top. The conventions in this section define the header file contents that follow the standard file heading.

#### Structural

The following structural conventions ensure that generic header files can be used in as wide a range of circumstances as possible, without running into problems associated with multiple inclusion or differences between ANSI C and C++.

- To ensure that a header file is not included more than once, the following must bracket all code in the header file. This follows the standard file heading, with the `#endif` appearing on the last line in the file.

```c
#ifndef __INCfooLibh
#define __INCfooLibh
...
#endif /* __INCfooLibh */
```

See D.3.5 C Naming Conventions, p.466, for the convention for naming preprocessor symbols used to prevent multiple inclusion.

- To ensure C++ compatibility, header files that are compiled in both a C and C++ environment must use the following code as a nested bracket structure, subordinate to the statements defined above:

```c
#ifndef __cplusplus
extern "C" {
#ifndef __cplusplus
...
#endif /* __cplusplus */
}```
Order of Declaration

The following order is recommended for declarations within a header file:

1. Statements that include other header files.
2. Simple defines of such items as error status codes and macro definitions.
3. Type definitions.
4. Function prototype declarations.

Example D-4  Sample C Header File

The following header file demonstrates the conventions described above:

```c
/* bootLib.h - boot support subroutine library */
/* Copyright 1984-1993 Wind River Systems, Inc. */
/*
modification history
--------------------
01g,22sep92,rrr added support for c++.
01f,04jul92,jcf cleaned up.
01e,26may92,rrr the tree shuffle.
01d,04oct91,rrr passed through the ansification filter,
-changed VOID to void
-changed copyright notice
01c,05oct90,shl added ANSI function prototypes;
added copyright notice.
01b,10aug90,dnw added declaration of bootParamsErrorPrint().
01a,18jul90,dnw written.
*/

#ifndef __INCbootLibh
#define __INCbootLibh
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

/* defines */
#define BOOT_DEV_LEN 20 /* max chars in device name */
#define BOOT_HOST_LEN 20 /* max chars in host name */
#define BOOT_ADDR_LEN 30 /* max chars in net addr */
#define BOOT_FILE_LEN 80 /* max chars in file name */
#define BOOT_USR_LEN 20 /* max chars in user name */
```

470
/* coding conventions */

#define BOOT_PASSWORD_LEN 20 /* max chars in password */
#define BOOT_OTHER_LEN 80 /* max chars in "other" field */
#define BOOT_FIELD_LEN 80 /* max chars in boot field */

/* typedefs */

typedef struct bootParams /* BOOT_PARAMS */
{
    char bootDev [BOOT_DEV_LEN]; /* boot device code */
    char hostName [BOOT_HOST_LEN]; /* name of host */
    char targetName [BOOT_HOST_LEN]; /* name of target */
    char ead [BOOT_ADDR_LEN]; /* ethernet internet addr */
    char had [BOOT_ADDR_LEN]; /* backplane internet addr */
    char gad [BOOT_ADDR_LEN]; /* gateway internet addr */
    char bootFile [BOOT_FILE_LEN]; /* name of boot file */
    char startupScript [BOOT_FILE_LEN]; /* name of startup script */
    char usr [BOOT_USR_LEN]; /* user name */
    char passwd [BOOT_PASSWORD_LEN]; /* password */
    char other [BOOT_OTHER_LEN]; /* avail to application */
    int procNum; /* processor number */
    int flags; /* configuration flags */
} BOOT_PARAMS;

/* function declarations */

extern STATUS bootBpAnchorExtract (char * string, char ** pAnchorAdrs);
extern STATUS bootNetmaskExtract (char * string, int * pNetmask);
extern STATUS bootScanNum (char ** ppString, int * pValue, BOOL hex);
extern STATUS bootStructToString (char * paramString, BOOT_PARAMS * pBootParams);
extern char * bootStringToStruct (char *bootString, BOOT_PARAMS * pBootParams);
extern void bootParamsErrorPrint (char *bootString, BOOT_PARAMS * pBootParams);
extern void bootParamsPrompt (char *string);
extern void bootParamsShow (char *paramString);

#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif /* __INCbootLibh */

D.3.8 Documentation Format Conventions for C

This section specifies the text-formatting conventions for source-code derived documentation. The Wind River tool refgen is used to generate reference entries (in HTML format) for every module automatically. All modules must be able to generate valid reference entries. This section is a summary of basic documentation format issues; for a more detailed discussion, see the Tornado BSP Developer’s Kit User’s Guide: Documentation Guidelines.
Layout

To work with refgen, the documentation in source modules must be laid out following a few simple principles. The file sample.c in installDir/target/unsupported/tools/mangen provides an example and more information.

Lines of text should fill out the full line length (assume about 75 characters); do not start every sentence on a new line.

Format Commands

Documentation in source modules can be formatted with UNIX nroff/troff formatting commands, including the standard man macros and several Wind River extensions to the man macros. Some examples are described in the sections below. Such commands should be used sparingly.

Any macro (or “dot command”) must appear on a line by itself, and the dot (. ) must be the first character on the logical line (in the case of subroutines, this is column 3, because subroutine comment sections begin each line with an asterisk plus a space character).

Special Elements

- **Parameters:** When referring to a parameter in text, surround the name with the angle brackets, < and >. For example, if the routine getName() had the following declaration:

```c
void getName

    { int tid, /* task ID */
      char * pTname /* task name */
          }
```

You might write something like the following:

This routine gets the name associated with a specified task ID and copies it to <pTname>.
**Subroutines:** Include parentheses with all subroutine names, even those generally construed as shell commands. Do not put a space between the parentheses or after the name (unlike the Wind River convention for code):

CORRECT: \( \text{taskSpawn()} \)

INCORRECT: \( \text{taskSpawn()}, \text{taskSpawn()}, \text{taskSpawn} \)

Note that there is one major exception to this rule. In the subroutine title, do not include the parentheses in the name of the subroutine being defined:

CORRECT:

```c
/*************************************************/
/* xxxFunc - do such and such*/
```

INCORRECT:

```c
/*************************************************/
/* xxxFunc() - do such and such*/
```

Avoid using a library, driver, or routine name as the first word in a sentence, but if you must, do not capitalize it.

**Terminal Keys:** Enter the names of terminal keys in all uppercase, as in \texttt{TAB} or \texttt{ESC}. Prefix the names of control characters with \texttt{CTRL+}; for example, \texttt{CTRL+C}.

**References to Publications:** References to chapters of publications should take the form \texttt{Title: Chapter}. For example, you might say:

For more information, see the \textit{VxWorks Programmer's Guide: I/O System}.

References to documentation volumes should be set off in italics. For general cases, use the \texttt{.I} macro. However, in \texttt{SEE ALSO} sections, use the \texttt{.PG} and \texttt{.TG} macros for the \textit{VxWorks Programmer's Guide} and \textit{Tornado User's Guide}, respectively.

**Section-Number Cross-References:** Do not use the UNIX parentheses-plus-number scheme to cross-reference the documentation sections for libraries and routines:

CORRECT:

```c
\texttt{sysLib, vxTas()}
```

INCORRECT:

```c
\texttt{sysLib(1), vxTas(2)}
```
Formatting Displays

- **Code**: Use the `.CS` and `.CE` macros for displays of code or terminal input/output. Introduce the display with the `.CS` macro; end the display with `.CE`. Indent such displays by four spaces from the left margin. For example:

```
  * .CS
  * struct stat statStruct;
  *  fd = open("file", READ);
  *  status = ioctl(fd, FIOFSTATGET, &statStruct);
  * .CE
```

- **Board Diagrams**: Use the `.bS` and `.bE` macros to display board diagrams under the BOARD LAYOUT heading in the `target.nr` module for a BSP. Introduce the display with the `.bS` macro; end the display with `.bE`.  

### Table D-1 Format of Special Elements

<table>
<thead>
<tr>
<th>Component</th>
<th>Input</th>
<th>Output (mangen + troff)</th>
</tr>
</thead>
<tbody>
<tr>
<td>library in title</td>
<td><code>sysLib.c</code></td>
<td><code>sysLib</code></td>
</tr>
<tr>
<td>library in text</td>
<td><code>sysLib</code></td>
<td>(same)</td>
</tr>
<tr>
<td>subroutine in title</td>
<td><code>sysMemTop</code></td>
<td><code>sysMemTop()</code></td>
</tr>
<tr>
<td>subroutine in text</td>
<td><code>sysMemTop()</code></td>
<td>(same)</td>
</tr>
<tr>
<td>subroutine parameter</td>
<td><code>&lt;ptid&gt;</code></td>
<td>(same)</td>
</tr>
<tr>
<td>terminal key</td>
<td><code>TAB, ESC, CTRL+C</code></td>
<td>(same)</td>
</tr>
<tr>
<td>emphasis</td>
<td><code>2mustP</code></td>
<td><code>must</code></td>
</tr>
</tbody>
</table>
Tables: Tables built with `tbl` are easy as long as you stick to basics, which suffice in almost all cases. Tables always start with the `.TS` macro and end with a `.TE`. The `.TS` should be followed immediately by a line of options terminated by a semicolon (`;`); then by one or more lines of column specification commands followed by a dot (`.`). For more details on table commands, refer to any UNIX documentation on `tbl`. The following is a basic example:

```
.TS
center; tab(1);
l1f3 l1f3

<table>
<thead>
<tr>
<th>Command</th>
<th>Op Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>INQUIRY</td>
<td>(0x12)</td>
</tr>
<tr>
<td>REQUEST SENSE</td>
<td>(0x03)</td>
</tr>
<tr>
<td>TEST UNIT READY</td>
<td>(0x00)</td>
</tr>
</tbody>
</table>

.TE
```

General stylistic considerations are as follows:
- Redefine the tab character using the `tab` option; keyboard tabs cannot be used by `tbl` tables. Typically the pipe character (`|`) is used.
- Center small tables on the page.
- Expand wide tables to the current line length.
- Make column headings bold.
- Separate column headings from the table body with a single line.
- Align columns visually.

Do not use `.CS/.CE` to build tables. This markup is reserved for code examples.

Lists: List items are easily created using the standard `man` macro `IP`. Do not use the `.CS/.CE` macros to build lists. The following is a basic example:

```
.IP "FIODISKFORMAT"
Formats the entire disk with appropriate hardware track and sector marks.
.IP "FIODISKINIT"
Initializes a DOS file system on the disk volume.
```
D.4 Tcl Coding Conventions

These conventions are divided into the following categories:

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

D.4.1 Tcl Module Layout

A *module* is any unit of code that resides in a single Tcl file. The conventions in this section define the standard module heading that must come at the beginning of every Tcl module following the standard file heading. The module heading consists of the blocks described below; the blocks are separated by one or more blank lines.

After the modification history and before the first function or executable code of the module, the following sections are included in the following order, if appropriate:

- **General Module Documentation:** The module documentation is a block of single-line Tcl comments beginning by the keyword `DESCRIPTION` and consisting of a complete description of the overall module purpose and function, especially the external interface. The description includes the heading `RESOURCE FILES` followed by a list of relevant Tcl files sourced inside the file.

- **Globals:** The globals block consists of a one-line Tcl comment containing the word `globals` followed by one or more Tcl declarations, one per line. This block groups together all declarations in the module that are intended to be visible outside the module.

The format of these blocks is shown in the following example (which also includes the Tcl version of the file heading):

Example D-5  Tcl File and Module Headings

```
# Browser.tcl - Browser Tcl implementation file
#
# Copyright 1994-1995 Wind River Systems, Inc.
```
# modification history
# --------------------
# 02b, 30oct95, jco added About menu and source browser.tcl in wind.
# 02a, 02sep95, pad fixed communications loss with license daemon (SPR #1234).
# 01c, 05mar95, jcf upgraded spy dialog
# 01b, 08feb95, p_m take care of loadFlag in wtxObjModuleInfoGet.
# 01a, 06dec94, c_s written.
#
# DESCRIPTION
# This module is the Tcl code for the browser. It creates the main window and
# initializes the objects in it, such as the task list and memory charts.
#
# RESOURCE FILES
# wpwr/host/resource/tcl/shelbrws.tcl
# wpwr/host/resource/tcl/app-config/Browser/*.tcl
# ...
#*/
#
# globals

set browserUpdate 0 ;# no auto update by default

D.4.2 Tcl Procedure Layout

The following conventions define the standard layout for every procedure in a
module.

Each procedure is preceded by the procedure documentation, a series of Tcl
comments that includes the following blocks. The documentation contains no
blank lines, but each block is delimited with a line containing a single pound
symbol (#) in the first column.

- **Banner**: A Tcl comment that consists of 78 pound symbols (#) across the page.
- **Title**: One line containing the routine name followed by a short, one-line
description. The routine name in the title must match the declared routine
name. This line becomes the title of automatically generated reference entries
and indexes.
- **Description**: A full description of what the routine does and how to use it.
- **Synopsis**: The word SYNOPSIS: followed by the synopsis of the
procedure—its name and parameter list between .tS and .tE macros. Optional
parameters are shown in square brackets. A variable list of arguments is
represented by three dots (...).
Parameters: For each parameter, the .IP macro followed by the parameter name on one line, followed by its complete description on the next line. Include the default value and domain of definition in each parameter description.

Returns: The word RETURNS: followed by a description of the possible explicit result values of the subroutine (that is, values returned with the Tcl return command).

RETURNS:
A list of 11 items: vxTicks taskId status priority pc sp errno timeout entry priNormal name

If the return value is meaningless enter N/A:

RETURNS: N/A

Errors: The word ERRORS: followed by all the error messages or error code (or both, if necessary) raised in the procedure by the Tcl error command.

ERRORS:
"Cannot find symbol in symbol table"

If no error statement is invoked in the procedure, enter N/A.

ERRORS: N/A

The procedure documentation ends with an empty Tcl comment starting in column one.

The procedure declaration follows the procedure heading and is separated from the documentation block by a single blank line. The format of the procedure and parameter declarations is shown in D.3.3 C Declaration Formats, p.460.

The following is an example of a standard procedure layout.

Example D-6 Standard Tcl Procedure Layout

```
# browse - browse an object, given its ID
#
# This routine is bound to the "Show" button, and is invoked when
# that button is clicked. If the argument (the contents of...
#
# SYNOPSIS
# .tS
# browse [objAddr | symbol | &symbol]
# .tE
```
# PARAMETERS
# .IP <objAddr>
# the address of an object to browse
# .IP <symbol>
# a symbolic address whose contents is the address of
# an object to browse
# .IP <&symbol>
# a symbolic address that is the address of an object to browse
#
# RETURNS: N/A
#
# ERRORS: N/A
#
proc browse {args} {
    ...
}

### D.4.3 Tcl Code Outside Procedures

Tcl allows code that is not in a procedure. This code is interpreted immediately when the file is read by the Tcl interpreter. Aside from the global-variable initialization done in the globals block near the top of the file, collect all such material at the bottom of the file.

However, it improves clarity—when possible—to collect any initialization code in an initialization procedure, leaving only a single call to that procedure at the bottom of the file. This is especially true for dialog creation and initialization, and more generally for all commands related to graphic objects.

Tcl code outside procedures must also have a documentation heading, including the following blocks:

- **Banner**: A Tcl comment that consists of 78 pound symbols (#) across the page.

- **Title**: One line containing the file name followed by a short, one-line description. The file name in the title must match the file name in the file heading.

- **Description**: A description of the out-of-procedure code.

The following is a sample heading for Tcl code outside all procedures.
Example D-7  Heading for Out-of-Procedure Tcl Code

```
# 01Spy.tcl - Initialization code
#
# This code is executed when the file is sourced. It executes the module
# entry routine which does all the necessary initialization to get a
# runnable spy utility.
#
# Call the entry point for the module
spyInit
```

**D.4.4 Declaration Formats**

Include only one declaration per line. Declarations are indented in accordance with *Indentation*, p. 482, and begin at the current indentation level. The remainder of this section describes the declaration formats for variables and procedures.

**Variables**

For global variables, the Tcl `set` command appears first on the line, separated from the identifier by a tab character. Complete the declaration with a meaningful comment at the end of the same line. Variables, values, and comments should be aligned, as in the following example:

```
set rootMemNBytes 0 ;# memory for TCB and root stack
set rootTaskId 0 ;# root task ID
set symSortByName 1 ;# boolean for alphabetical sort
```

**Procedures**

The procedure name and list of parameters appear on the first line, followed by the opening curly brace. The declarations of global variables used inside the procedure begin on the next line, one on each separate line. The rest of the procedure code begins after a blank line. For example:

```
proc lstFind {list node} {
    global firstNode
    global lastNode
    ...
}
```
D.4.5 Code Layout

The maximum length for any line of code is 80 characters. If more than 80 characters are required, use the backslash character to continue on the next line.

The rest of this section describes conventions for the graphic layout of Tcl code, covering the following elements:

- vertical spacing
- horizontal spacing
- indentation
- comments

Vertical Spacing

- Use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines.
- Do not put more than one declaration on a line. Each variable and function argument must be declared on a separate line.
- Do not put more than one statement on a line. The only exceptions are:
  - A for statement where the initial, conditional, and loop statements can be written on a single line:
    
    ```
    for {set i 0} {$i < 10} {incr i 3} {
    
    - A switch statement whose actions are short and nearly identical (see the switch statement format in Indentation, p. 482).

    The if statement is not an exception. The conditionally executed statement always goes on a separate line from the conditional expression:

    ```
    if {$i > $count} {
        set i $count
    }
    ```

- Opening braces ({}), defining a command body, are always on the same line as the command itself.
- Closing braces () and switch patterns always have their own line.
Horizontal Spacing

- Put spaces around binary operators. Put spaces before an open parenthesis, open brace and open square bracket if it follows a command or assignment statement. For example:

```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 + $c) * \n    ($d + $e)]
set status [fooList $foo $a $b $c \n    $d $e]
if {($a == $b) && \n    ($c == $d)} {
```

Indentation

- Indentation levels are every four characters (columns 1, 5, 9, 13, ...).

- The default tab width must be eight characters (assumed to be at columns 1, 9, 17, ...). The intermediate indentations are achieved with spaces.

- The module and procedure headings and the procedure declarations start in column one.

- The closing brace of a command body is always aligned on the same column as the command it is related to:

```tcl
while { condition }
{
    statements
}
```

```tcl
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:

```plaintext
if { condition } {
    statements
} else {
    statements
}
```

The form of the conditional statement with an **elseif** is:

```plaintext
if { condition } {
    statements
} elseif { condition } {
    statements
} else {
    statements
}
```

The general form of the **switch** statement is:

```plaintext
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:

```plaintext
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.484).

Opening body braces (`) have no specific indentation; they follow the command on the same line.
Comments

- Place comments within code so that they precede the section of code to which they refer and have the same level of indentation. Separate such comments from the code by a single blank line.
  - Begin single-line comments with the pound symbol as in the following:
    
    ```
    # This is the correct format for a single-line comment
    set foo 0
    ```
  - Multi-line comments have each line beginning with the pound symbol as in the example below. Do not use a backslash to continue a comment across lines.
    
    ```
    # This is the CORRECT format for a multiline comment
    # in a section of code.
    set foo 0
    # This is the INCORRECT format for a multiline comment \ in a section of code.
    set foo 0
    ```

- Comments on global variables appear on the same line as the variable declaration, using the semicolon (;) character:
  
  ```
  set day night ;# This is a global variable
  ```

D.4.6 Naming Conventions

The following conventions define the standards for naming modules, routines and variables. The purpose of these conventions is uniformity and readability of code.

- When creating names, remember that code is written once but read many times. Make names meaningful and readable. Avoid obscure abbreviations.

- Names of routines and variables are composed of upper- and lowercase characters and no underbars. Capitalize each “word” except the first:

  ```
  aVariableName
  ```
• Every module has a short prefix (two to five characters). The prefix is attached to the module name and to all externally available procedures and variables. (Names that are not available externally need not follow this convention.)

<table>
<thead>
<tr>
<th>fooLib.tcl</th>
<th>module name</th>
</tr>
</thead>
<tbody>
<tr>
<td>fooObjFind</td>
<td>procedure name</td>
</tr>
<tr>
<td>fooCount</td>
<td>variable name</td>
</tr>
</tbody>
</table>

• 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:

<table>
<thead>
<tr>
<th>fooObjFind</th>
<th>foo - object - find</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysNvRamGet</td>
<td>system - non volatile RAM - get</td>
</tr>
<tr>
<td>taskInfoGet</td>
<td>task - info - get</td>
</tr>
</tbody>
</table>

### D.4.7 Tcl Style

The following conventions define additional standards of programming style:

• **Comments**: Insufficiently commented code is unacceptable.

• **Procedure Length**: Procedures should have a reasonably small number of lines, less than 50 if possible.

• **Case Statement**: Do not use the case keyword. Use switch instead.

• **expr and Control Flow Commands**: Do not use expr in commands such as if, for or while except to convert a variable from one format to another:

  CORRECT:
  ```tcl```
  ```tcl
  if {$id != 0} {
  ```
  CORRECT:
  ```tcl```
  ```tcl
  if { [expr $id] != 0 } {
  ```
  INCORRECT:
  ```tcl```
  ```tcl
  if { [expr $id != 0] } {
  ```

• **expr and incr**: Do not use expr to increment or decrement the value of a variable. Use incr instead.
CORRECT:

    incr index

CORRECT:

    incr index -4

INCORRECT:

    set index [expr $index + 1]

- **wtxPath and wtxHostType**: Use these routines when developing tools for Tornado. With no arguments, `wtxPath` returns the value of the environment variable `WIND_BASE` with a "/" appended. With an argument list, the result of `wtxPath` is an absolute path rooted in `WIND_BASE` with each argument as a directory segment. Use this command in Tornado tools to read resource files. The `wtxHostType` call returns the host-type string for the current process (the environment variable `WIND_HOST_TYPE` if properly set, has the same value). For example:

    source [wtxPath host resource tcl]wtxcore.tcl
    set backenddir [wtxPath host [wtxHostType] lib backend]*

- **catch Command**: The `catch` command is very useful to intercept errors raised by underlying procedures so that a script does not abort prematurely. However, use the `catch` command with caution. It can obscure the real source of a problem, thus causing errors that are particularly hard to diagnose. In particular, do not use `catch` to capture the return value of a command without testing it. Note also that if the intercepted error cannot be handled, the error must be resubmitted exactly as it was received (or translated to one of the defined errors in the current procedure):

CORRECT:

    if [catch "dataFetch $list" result] {
        if ($result == "known problem") {
            specialCaseHandle
        } else {
            error $result
        }
    }

INCORRECT:

    catch "dataFetch $list" result
• **if then else Statement:** In an if command, you may omit the keyword *then* before the first command body; but do not omit *else* if there is a second command body.

**CORRECT:**

```tcl
if {$id != 0} {
  ...
} else {
  ...
}
```

**INCORRECT:**

```tcl
if {$id !=0} then {
  ...
} {
  ...
}
```

• **Return Values:** Tcl procedures only return strings; whatever meaning the string has (a list for instance) is up to the application. Therefore each constant value that a procedure can return must be described in the procedure documentation, in the `RETURNS:` block. If a complex element is returned, provide a complete description of the element layout. Do not use the `return` statement to indicate that an abnormal situation has occurred; use the `error` statement in that situation.

The following illustrates a complex return value consisting of a description:

```tcl
# Return a list of 11 items: vxTicks taskId status priority pc
# sp errno timeout entry priNormal name
return [concat [lrange $tiList 0 1] [lrange $tiList 3 end]]
```

The following illustrates and simple return value:

```tcl
# This code checks whether the VxMP component is installed:
if [catch "wtxSymFind -name smObjPoolMinusOne" result] {
  if ([wtxErrorName $result] == "SYMTBL_SYMBOL_NOT_FOUND") {
    return -1 # VxMP is not installed
  } else {
    error $result
  }
} else {
  return 0  # VxMP is installed
}
```
- **Error Conditions:** The Tcl `error` command raises an error condition that can be trapped by the `catch` command. If not caught, an error condition terminates script execution. For example:

```tcl
if {$defaultTaskId == 0} {
    error "No default task has been established."
}
```

Because every error message and error code must be described in the procedure header in the `ERRORS:` block, it is sometimes useful to call `error` in order to replace an underlying error message with an error expressed in terms directly relevant to the current procedure. For example:

```tcl
if [catch "wtxObjModuleLoad $periodModule" status] {
    error "Cannot add period support module to Target ($status)"
}
```
The Tornado utility `servutil.exe` (located in `c:\tornado\host\x86-win32\bin`) can be used to install and remove Windows NT services.

**Installing a Service**

Figure E-1 illustrates how to fill out the `servutil` dialog box to install the Tornado registry as a service.

**Figure E-1  **

*Installing the Tornado Registry as a Service*
Removing a Service

Figure E-2 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.

**Figure E-2  Removing a Service**

![Service Manager](image)

**NOTE:** To find the names of existing services, click Service in the Control Panel.
F.1 Introduction

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.

We recommend that you install (as an NT service) the FTP server provided by the operating system, when you install TCP/IP.

As an alternative, 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.
F.2 Starting WFTPD

The WFTPD utility is distributed as installDir\host\x86-win32\bin\wftpd32.exe. If you plan to use FTP as the normal means of booting VxWorks, one of the following options may work for you:

- To start wftpd32 manually, include it in the Tornado program group.
- To start wftpd32 automatically when you start your Windows NT host, add wftpd32 to Start Menu Programs (Start>Settings>Taskbar>Start Menu Programs).
- To start wftpd32 automatically when you start your Windows 2000 host, add wftpd32 to Startup Folder. From the Taskbar and Start Menu select the Advanced tab and the Add button. In the Create Shortcut wizard browse to wftpd32.exe and choose Start Menu>Programs>Startup Folder.

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.45.
- 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.45.
- 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.

NOTE: You can run the FTP server as a restricted user, but you cannot add new users and passwords if you are a restricted user. A non-restricted user must add the new users and passwords for you.
G.1 Introduction

This section covers the initialization sequence for VxWorks in a typical development configuration. The steps are described in sequence of execution. This is not the only way VxWorks can be bootstrapped on a particular processor. There are often more efficient or robust techniques unique to a particular processor or hardware; consult your hardware’s documentation.

For final production, the sequence can be revisited to include diagnostics or to remove some of the generic operations that are required for booting a development environment, but that are unnecessary for production. This description can provide only an approximate guide to the processor initialization sequence and does not document every exception to this time-line.

The early steps of the initialization sequence are slightly different for ROM-based versions of VxWorks; for information, see G.9 Initialization Sequence for ROM-Based VxWorks, p.508.

For a summary of the initialization time-line, see Table G-1. The following sections describe the initialization in detail by routine name. For clarity, the sequence is divided into a number of main steps or function calls. The key routines are listed in the headings and are described in chronological order.
G.2 The VxWorks Entry Point: sysInit()

The first step in starting a VxWorks system is to load a system image into main memory. This usually occurs as a download from the development host, under the control of the VxWorks boot ROM. Next, the boot ROM transfers control to the VxWorks startup entry point, sysInit(). This entry point is configured by RAM_LOW_ADRS in the makefile and in config.h. The VxWorks memory layout is different for each architecture; for details, see the appendix that describes your architecture.

The entry point, sysInit(), is in the system-dependent assembly language module, sysALib.s. It locks out all interrupts, invalidates caches if applicable, and initializes processor registers (including the C stack pointer) to default values. It also disables tracing, clears all pending interrupts, and invokes usrInit(), a C subroutine in the usrConfig.c module. For some targets, sysInit() also performs some minimal system-dependent hardware initialization, enough to execute the remaining initialization in usrInit(). The initial stack pointer, which is used only by usrInit(), is set to occupy an area below the system image but above the vector table (if any).

G.3 The Initial Routine: usrInit()

The usrInit() routine (in usrConfig.c) saves information about the boot type, handles all the initialization that must be performed before the kernel is actually started, and then starts the kernel execution. It is the first C code to run in VxWorks. It is invoked in supervisor mode with all hardware interrupts locked out.

Many VxWorks facilities cannot be invoked from this routine. Because there is no task context as yet (no TCB and no task stack), facilities that require a task context cannot be invoked. This includes any facility that can cause the caller to be preempted, such as semaphores, or any facility that uses such facilities, such as printf(). Instead, the usrInit() routine does only what is necessary to create an initial task, usrRoot(). This task then completes the startup.

The initialization in usrInit() includes the following:
Cache Initialization

The code at the beginning of \texttt{usrInit()} initializes the caches, sets the mode of the caches and puts the caches in a safe state. At the end of \texttt{usrInit()}, the instruction and data caches are enabled by default.

Zeroing Out the System bss Segment

The C and C++ languages specify that all uninitialized variables must have initial values of 0. These uninitialized variables are put together in a segment called \texttt{bss}. This segment is not actually loaded during the bootstrap, because it is known to be zeroed out. Because \texttt{usrInit()} is the first C code to execute, it clears the section of memory containing \texttt{bss} as its very first action. While the VxWorks boot ROMs clear all memory, VxWorks does not assume that the boot ROMs are used.

Initializing Interrupt Vectors

The exception vectors must be set up before enabling interrupts and starting the kernel. First, \texttt{intVecBaseSet()} is called to establish the vector table base address.

\begin{center}
\textbf{NOTE:} There are exceptions to this in some architectures; see the appendix that describes your architecture for details.
\end{center}

After \texttt{intVecBaseSet()} is called, the routine \texttt{excVecInit()} initializes all exception vectors to default handlers that safely trap and report exceptions caused by program errors or unexpected hardware interrupts.

Initializing System Hardware to a Quiescent State

System hardware is initialized by calling the system-dependent routine \texttt{sysHwInit()}. This mainly consists of resetting and disabling hardware devices that can cause interrupts after interrupts are enabled (when the kernel is started). This is important because the VxWorks ISRs (for I/O devices, system clocks, and so on), are not connected to their interrupt vectors until the system initialization is completed in the \texttt{usrRoot()} task. However, do not attempt to connect an interrupt handler to an interrupt during the \texttt{sysHwInit()} call, because the memory pool is not yet initialized.
G.4 Initializing the Kernel

The `usrInit()` routine ends with calls to two kernel initialization routines:

- `usrKernelInit()` (defined in `usrKernel.c`)
  calls the appropriate initialization routines for each of the specified optional kernel facilities (see Table G-1 for a list).

- `kernelInit()` (part of `kernelLib.c`)
  initiates the multitasking environment and never returns. It takes the following parameters:
  - The application to be spawned as the “root” task, typically `usrRoot()`.
  - The stack size.
  - The start of usable memory; that is, the memory after the main text, data, and `bss` of the VxWorks image. All memory after this area is added to the system memory pool, which is managed by `memPartLib`. Allocation for dynamic module loading, task control blocks, stacks, and so on, all come out of this region. See G.5 Initializing the Memory Pool, p.499.
  - The top of memory as indicated by `sysMemTop()`. If a contiguous block of memory is to be preserved from normal memory allocation, pass `sysMemTop()` less the reserved memory.
  - The interrupt stack size. The interrupt stack corresponds to the largest amount of stack space any interrupt-level routine uses, plus a safe margin for the nesting of interrupts.
  - The interrupt lock-out level. For architectures that have a `level` concept, it is the maximum level. For architectures that do not have a level concept, it is the mask to disable interrupts. See the appendix that describes your architecture for details.

`kernelInit()` calls `intLockLevelSet()`, disables round-robin mode, and creates an interrupt stack if supported by the architecture. It then creates a root stack and TCB from the top of the memory pool, spawns the root task, `usrRoot()`, and terminates the `usrInit()` thread of execution. At this time, interrupts are enabled; it is critical that all interrupt sources are disabled and pending interrupts cleared.
G.5 Initializing the Memory Pool

VxWorks includes a memory allocation facility, in the module `memPartLib`, that manages a pool of available memory. The `malloc()` routine allows callers to obtain variable-size blocks of memory from the pool. Internally, VxWorks uses `malloc()` for dynamic allocation of memory. In particular, many VxWorks facilities allocate data structures during initialization. Therefore, the memory pool must be initialized before any other VxWorks facilities are initialized.

Note that the Tornado target server manages a portion of target memory to support downloading of object modules and other development functions. VxWorks makes heavy use of `malloc()`, including allocation of space for loaded modules, allocation of stacks for spawned tasks, and allocation of data structures on initialization. You are also encouraged to use `malloc()` to allocate any memory your application requires. Therefore, it is recommended that you assign to the VxWorks memory pool all unused memory, unless you must reserve some fixed absolute memory area for a particular application use.

The memory pool is initialized by `kernelInit()`. The parameters to `kernelInit()` specify the start and end address of the initial memory pool. In the default `usrInit()` distributed with VxWorks, the pool is set to start immediately following the end of the booted system, and to contain all the rest of available memory.

The extent of available memory is determined by `sysMemTop()`, which is a system-dependent routine that determines the size of available memory. If your system has other noncontiguous memory areas, you can make them available in the general memory pool by later calling `memAddToPool()` in the `usrRoot()` task.

G.6 The Initial Task: usrRoot()

When the multitasking kernel starts executing, all VxWorks multitasking facilities are available. Control is transferred to the `usrRoot()` task and the initialization of the system can be completed. For example, `usrRoot()` performs the following:

- initialization of the system clock
- initialization of the I/O system and drivers
- creation of the console devices
- setting of standard in and standard out
- installation of exception handling and logging
– initialization of the pipe driver
– initialization of standard I/O
– creation of file system devices and installation of disk drivers
– initialization of floating-point support
– initialization of performance monitoring facilities
– initialization of the network
– initialization of optional facilities
– initialization of WindView (see the WindView User’s Guide)
– initialization of target agent
– execution of a user-supplied startup script

To review the complete initialization sequence within `usrRoot()`, see `installDir/target/config/all/ usrConfig.c`.

Modify these initializations to suit your configuration. The meaning of each step and the significance of the various parameters are explained in the following sections.

**Initialization of the System Clock**

The first action in the `usrRoot()` task is to initialize the VxWorks clock. The system clock interrupt vector is connected to the routine `usrClock()` (described in G.7 The System Clock Routine: `usrClock()`, p.505) by calling `sysClkConnect()`. Then, the system clock rate (usually 60Hz) is set by `sysClkRateSet()`. Most boards allow clock rates as low as 30Hz (some even as low as 1Hz), and as high as several thousand Hz. High clock rates (>1000Hz) are not desirable, because they can cause system thrashing.1

The timer drivers supplied by Wind River include a call to `sysHwInit2()` as part of the `sysClkConnect()` routine. Wind River BSPs use `sysHwInit2()` to perform further board initialization that is not completed in `sysHwInit()`. For example, an `intConnect()` of ISRs can take place here, because memory can be allocated now that the system is multitasking.

**Initialization of the I/O System**

If `INCLUDE_IO_SYSTEM` is defined in `configAll.h`, the VxWorks I/O system is initialized by calling the routine `iosInit()`. The arguments specify the maximum

---

1. *Thrashing* occurs when clock interrupts are so frequent that the processor spends too much time servicing the interrupts, and no application code can run.
number of drivers that can be subsequently installed, the maximum number of files that can be open in the system simultaneously, and the desired name of the “null” device that is included in the VxWorks I/O system. This null device is a “bit-bucket” on output and always returns end-of-file for input.

The inclusion or exclusion of INCLUDE_IO_SYSTEM also affects whether the console devices are created, and whether standard in, standard out, and standard error are set; see the next two sections for more information.

**Creation of the Console Devices**

If the driver for the on-board serial ports is included (INCLUDE_TTY_DEV), it is installed in the I/O system by calling the driver’s initialization routine, typically ttyDrv(). The actual devices are then created and named by calling the driver’s device-creation routine, typically ttyDevCreate(). The arguments to this routine includes the device name, a serial I/O channel descriptor (from the BSP), and input and output buffer sizes.

The macro NUM_TTY specifies the number of tty ports (default is 2), CONSOLE_TTY specifies which port is the console (default is 0), and CONSOLE_BAUD_RATE specifies the bps rate (default is 9600). These macros are specified in configAll.h, but can be overridden in config.h for boards with a nonstandard number of ports.

PCs can use an alternative console with keyboard input and VGA output; see your PC workstation documentation for details.

**Setting of Standard In, Standard Out, and Standard Error**

The system-wide standard in, standard out, and standard error assignments are established by opening the console device and calling ioGlobalStdSet(). These assignments are used throughout VxWorks as the default devices for communicating with the application developer. To make the console device an interactive terminal, call ioctl() to set the device options to OPT_TERMINAL.

**Installation of Exception Handling and Logging**

Initialization of the VxWorks exception handling facilities (supplied by the module excLib) and logging facilities (supplied by logLib) takes place early in the
execution of the root task. This facilitates detection of program errors in the root
task itself or in the initialization of the various facilities.

The exception handling facilities are initialized by calling `excInit()` when
`INCLUDE_EXC_HANDLING` and `INCLUDE_EXC_TASK` are defined. The `excInit()`
routine spawns the exception support task, `excTask()`. Following this
initialization, program errors causing hardware exceptions are safely trapped and
reported, and hardware interrupts to uninitialized vectors are reported and
dismissed. The VxWorks signal facility, used for task-specific exception handling,
is initialized by calling `sigInit()` when `INCLUDE_SIGNALS` is defined.

The logging facilities are initialized by calling `logInit()` when
`INCLUDE_LOGGING` is defined. The arguments specify the file descriptor of the
device to which logging messages are to be written, and the number of log message
buffers to allocate. The logging initialization also includes spawning the logging
task, `logTask()`.

**Initialization of the Pipe Driver**

If named pipes are desired, define `INCLUDE_PIPE` in `configAll.h` so that `pipeDrv()`
is called automatically to initialize the pipe driver. Tasks can then use pipes to
communicate with each other through the standard I/O interface. Pipes must be
created with `pipeDevCreate()`.

**Initialization of Standard I/O**

VxWorks includes an optional standard I/O package when `INCLUDE_STDIO` is
defined.

**Creation of File System Devices and Initialization of Device Drivers**

Many VxWorks configurations include at least one disk device or RAM disk with
a dosFs, rt11Fs, or rawFs file system. First, a disk driver is installed by calling the
driver’s initialization routine. Next, the driver’s device-creation routine defines a
device. This call returns a pointer to a `BLK_DEV` structure that describes the device.

The new device can then be initialized and named by calling the file system’s
device-initialization routine—`dosFsDevInit()`, `rt11FsDevInit()`, or
`rawFsDevInit()`—when the respective constants `INCLUDE_DOSFS`,
`INCLUDE_RT11FS`, and `INCLUDE_RAWFS` are defined. (Before a device can be
initialized, the file system module must already be initialized with dosFsInit(), rt11FsInit(), or rawFsInit(). The arguments to the file system device-initialization routines depend on the particular file system, but typically include the device name, a pointer to the BLK_DEV structure created by the driver’s device-creation routine, and possibly some file-system-specific configuration parameters.

**Initialization of Floating-Point Support**

Support for floating-point I/O is initialized by calling the routine floatInit() when INCLUDE_FLOATING_POINT is defined in configAll.h. Support for floating-point coprocessors is initialized by calling mathHardInit() when INCLUDE_HW_FP is defined. Support for software floating-point emulation is initialized by calling mathSoftInit() when INCLUDE_SW_FP is defined. See the appropriate architecture appendix for details on your processor’s floating-point support.

**Inclusion of Performance Monitoring Tools**

VxWorks has two built-in performance monitoring tools. A task activity summary is provided by spyLib, and a subroutine execution timer is provided by timexLib. These facilities are included by defining the macros INCLUDE_SPY and INCLUDE_TIMEX, respectively, in configAll.h.

**Initialization of the Network**

Before the network can be used, it must be initialized with the routine usrNetInit(), which is called by usrRoot() when the constant INCLUDE_NET_INIT is defined in one of the configuration header files. (The source for usrNetInit() is in installDir/target/src/config/usrNetwork.c.) The routine usrNetInit() takes a configuration string as an argument. This configuration string is usually the “boot line” that is specified to the VxWorks boot ROMs to boot the system (see Tornado Getting Started). Based on this string, usrNetInit() performs the following:

- Initializes network subsystem by calling the routine netLibInit().
- Attaches and configures appropriate network drivers.
- Adds gateway routes.
- Initializes the remote file access driver netDrv, and adds a remote file access device.
- Initializes the remote login facilities.
- Optionally initializes the Remote Procedure Calls (RPC) facility.
- Optionally initializes the Network File System (NFS) facility.

As noted previously, the inclusion of some of these network facilities is controlled by definitions in `configAll.h`. The network initialization steps are described in the *VxWorks Network Programmer’s Guide*.

**Initialization of Optional Products and Other Facilities**

Shared memory objects are provided with the optional product VxMP. Before shared memory objects can be used, they must be initialized with the routine `usrSmObjInit()` (in `installDir/target/src/config/usrSmObj.c`), which is called from `usrRoot()` if `INCLUDE_SM_OBJ` is defined.

⚠️ **CAUTION:** The shared memory objects library requires information from fields in the VxWorks boot line. The functions are contained in the `usrNetwork.c` file. If no network services are included, `usrNetwork.c` is not included and the shared memory initialization fails. The project facility calculates all dependencies but if you are using manual configuration, either add `INCLUDE_NETWORK` to `configAll.h` or extract the bootline cracking routines from `usrNetwork.c` and include them elsewhere.

Basic MMU support is provided if `INCLUDE_MMU_BASIC` is defined. Text protection, vector table protection, and a virtual memory interface are provided with the optional product VxVMI, if `INCLUDE_MMU_FULL` is defined. The MMU is initialized by the routine `usrMmuInit()`, located in `installDir/target/src/config/usrMmuInit.c`. If the macros `INCLUDE_PROTECT_TEXT` and `INCLUDE_PROTECT_VEC_TABLE` are also defined, text protection and vector table protection are initialized.

Wind River compilers support the C++ language. Run-time C++ support is enabled by defining `INCLUDE_CPLUS`. Additional C++ facilities can also be included by defining the appropriate `INCLUDE_CPLUS_XXX` macros. For more details see the *VxWorks Programmer’s Guide: C++ Development*.
Initialization of WindView

Kernel instrumentation is provided with the optional product WindView. It is initialized in `usrRoot()` when `INCLUDE_WINDVIEW` is defined in `configAll.h`. Other WindView configuration constants control particular initialization steps; see the WindView User’s Guide: Configuring WindView.

Initialization of the Target Agent

If `INCLUDE_WDB` is defined, `wdbConfig()` in `installDir/target/src/config/usrWdb.c` is called. This routine initializes the agent’s communication interface, then starts the agent. For information on configuring the agent, see 5.3 Configuring VxWorks, p.167.

Execution of a Startup Script

The `usrRoot()` routine executes a user-supplied startup script if the target-resident shell is configured into VxWorks, `INCLUDE_STARTUP_SCRIPT` is defined, and the script’s file name is specified at boot time with the startup script parameter (see Tornado Getting Started). If the parameter is missing, no startup script is executed.

G.7 The System Clock Routine: `usrClock()`

Finally, the system clock ISR `usrClock()` is attached to the system clock timer interrupt by the `usrRoot()` task described G.6 The Initial Task: `usrRoot()`, p.499. The `usrClock()` routine calls the kernel clock tick routine `tickAnnounce()`, which performs OS bookkeeping. You can add application-specific processing to this routine.
G.8 Initialization Summary

Table G-1 shows a summary of the entire VxWorks initialization sequence for typical configurations. For a similar summary applicable to ROM-based VxWorks systems, see G.9 Initialization Sequence for ROM-Based VxWorks, p.508.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Activity</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysInit()</td>
<td>(a) lock out interrupts</td>
<td>sysALib.s</td>
</tr>
<tr>
<td></td>
<td>(b) invalidate caches, if any</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c) initialize system interrupt tables with default stubs (i960 only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(d) initialize system fault tables with default stubs (i960 only)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(e) initialize processor registers to known default values</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(f) disable tracing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(g) clear all pending interrupts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(h) invoke usrInit() specifying boot type</td>
<td></td>
</tr>
<tr>
<td>usrInit()</td>
<td>(a) invoke optional sysHwInit0()</td>
<td>usrConfig.c</td>
</tr>
<tr>
<td></td>
<td>(b) invoke cacheInit()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c) zero bss (uninitialized data)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(d) save bootType in sysStartType</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(e) invoke excVecInit() to initialize all system and default interrupt vectors</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(f) invoke sysHwInit()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(g) invoke usrKernelInit()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(h) enable caches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(i) invoke kernelInit()</td>
<td></td>
</tr>
<tr>
<td>usrKernelInit()</td>
<td>The following routines are invoked if their configuration constants are defined.</td>
<td>usrKernel.c</td>
</tr>
</tbody>
</table>

506
Table G-1  VxWorks Run-time System Initialization Sequence (Continued)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Activity</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) classLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b) taskLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c) taskHookInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(d) semBLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(e) semMLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(f) semCLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(g) semOLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(h) wdLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(i) msgQLibInit( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(j) qInit( ) for all system queues</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(k) workQInit( )</td>
<td></td>
</tr>
<tr>
<td>kernelInit( )</td>
<td>Initialize and start the kernel.</td>
<td>kernelLib.c</td>
</tr>
<tr>
<td></td>
<td>(a) invoke intLockLevelSet( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b) create root stack and TCB from top of memory pool</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c) invoke taskInit( ) for usrRoot( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(d) invoke taskActivate( ) for usrRoot( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(e) usrRoot( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>usrRoot( ) Initialize I/O system, install drivers, and create devices</td>
<td>usrConfig.c</td>
</tr>
<tr>
<td></td>
<td>as specified in configAll.h and config.h. See usrConfig.c for a complete</td>
<td></td>
</tr>
<tr>
<td></td>
<td>list of optional kernel facilities initialized.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(a) Initialize memory partitions and MMU</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(b) sysClkConnect( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c) sysClkRateSet( )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(d) selectInit( )</td>
<td></td>
</tr>
</tbody>
</table>
The early steps of system initialization are somewhat different for the ROM-based versions of VxWorks: on most target architectures, the two routines `romInit()` and `romStart()` execute instead of the usual VxWorks entry point, `sysInit()`.

ROM Entry Point: `romInit()`

At power-up the processor begins executing at `romInit()` (defined in `installDir/target/config/bspname/romInit.s`). The `romInit()` routine disables interrupts, puts the boot type (cold/warm) on the stack, performs hardware-dependent initialization (such as clearing caches or enabling DRAM), and branches to `romStart()`. The stack pointer is initialized to reside below the data section in the case of ROM-resident versions of VxWorks (in RAM versions, the stack pointer instead resides below the text section).
Copying the VxWorks Image: `romStart()`

Next, the `romStart()` routine (in `installDir/target/config/all/bootInit.c` loads the VxWorks system image into RAM. If the ROM-resident version of VxWorks is selected, the data segment is copied from ROM to RAM and memory is cleared. If VxWorks is not ROM resident, all of the text and code segment is copied and decompressed from ROM to RAM, to the location defined by `RAM_HIGH_ADRS` in `Makefile`. If VxWorks is neither ROM resident nor compressed, the entire text and data segment is copied without decompression straight to RAM, to the location defined by `RAM_LOW_ADRS` in `Makefile`.

Overall Initialization for ROM-Based VxWorks

Beyond `romStart()`, the initialization sequence for ROM-based VxWorks resembles the normal sequence, continuing with the `usrInit()` call.

Table G-2 summarizes the complete initialization sequence. For details on the steps after `romInit()` and `romStart()`, see G.8 Initialization Summary, p.506.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Activity</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <code>romInit()</code></td>
<td>(a) disable interrupts</td>
<td><code>romInit.s</code></td>
</tr>
<tr>
<td></td>
<td>(b) save boot type (cold/warm)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c) hardware-dependent initialization</td>
<td></td>
</tr>
<tr>
<td>2. <code>romStart()</code></td>
<td>(a) copy data segment from ROM to RAM; clear memory</td>
<td><code>bootInit.c</code></td>
</tr>
<tr>
<td></td>
<td>(b) copy code segment from ROM to RAM, decompressing if necessary</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(c) invoke <code>usrInit()</code> with boot type</td>
<td></td>
</tr>
<tr>
<td>3. <code>usrInit()</code></td>
<td>Initial routine.</td>
<td><code>usrConfig.c</code></td>
</tr>
<tr>
<td>4. <code>usrKernelInit()</code></td>
<td>Routines invoked if the corresponding configuration constants are defined.</td>
<td><code>usrKernel.c</code></td>
</tr>
<tr>
<td>5. <code>kernelInit()</code></td>
<td>Initialize and start the kernel.</td>
<td><code>kernelLib.c</code></td>
</tr>
<tr>
<td>6. <code>usrRoot()</code></td>
<td>Initialize I/O system, install drivers, and create devices as configured in <code>configAll.h</code> and <code>config.h</code>.</td>
<td><code>usrConfig.c</code></td>
</tr>
<tr>
<td>Application routine</td>
<td>Application code.</td>
<td>Application source file</td>
</tr>
</tbody>
</table>
Symbols

? (C/Tcl interpreter toggle)  271
  quick access  273
@ command (booting)  47
@ prefix (WindSh)  259

A

About Tornado dialog box  65
add command (debugger)  354
Add New Build Specification window  136
Add Symbols command (Debug menu)  375
  code example  375
ADDED_C++FLAGS  190
ADDED_CFLAGS  444
  modifying run-time  190
ADDED_MODULES  444
  modifying run-time  189
address(es), memory
  current, setting  254
add-symbol-file command (debugger)  353
agent, see target agent
agentModeShow( )  237
  mode switching, under  243
ALT+F (open recently-used file)  67
ALT+F t (print)  72
ALT+F3 (find)  70

ALT+Fa (save as)  68
ANSI C
  function prototypes  153
  header files  154
-ansi compiler option  160
application files
  adding  99
  closing  100
  creating  99
  displaying information about  100
  linking files with  100
  modifying  100
  opening  100
  removing  100
  saving  100
VxWorks, for  113
application I/O (debugger)  355
application modules  152–167
  see also bootable applications; downloadable applications
  see online loadLib; unldLib
bootable  77
compiling with Diab
  C  163
  C++  164
compiling with GNU
  C  159
  C++  160
CPU type, defining
  Diab, using  162
GNU, using 158

cross-development 3

displaying information about
(browsers) 318–320
downloadable 76

group numbers 166

linking 164

loading 165–167

loading during debugging 354

option flags 358

make variables 186–190

makefiles

include files, using 190

module IDs 166

unloading

command line, from the 167

workspace, from the 336

application wizard

custom VxWorks images, creating 110
downloadable applications, creating 94

applications

see also bootable applications; downloadable applications; projects

adding source code 84

architecture-independent 86

creating new source code 85

custom build rules, using 85

linking to VxSim 203

VxSim, building 202

VxWorks for, configuring 87

architecture-independent development 86

architecture-specific development

CPU type

Diab, defining for 162

GNU, defining for 158

Diab compiler invocations 163

GNU compiler invocations 159

VxSim, simulated target 197–217

arp command 59

Arrange icons command 54

arrays (WindSh) 257

assignments (WindSh) 254

Attach command (ALT+F6) (Debug menu) 343

attach system command (debugger) 361

Attribute Pane checkbox (Options window) 403

attribute panes 66

authorizations, see security

auto scaling 128

AUTOREGISTER_COCLASS 395

B

-B option (target server) 295

b() 237

system mode, in 243

back ends, communications 286

Back Trace command (ALT+7) (Debug menu) 349

backplane

installing boards in 24

processor number 45

bd() 236

bdall() 236

bh() 237

BINHEX 446

board support packages (BSP) 168–171

boot media 23

building projects from 195

creating 82

creating projects 112

definition 78

documentation 170

header file (config.h) 172

initialization modules 170

make variables 186–190

parameter variables 188

pre-existing, using

project facility, outside the 83

project facility, with 82

simulator, using 83

system library 169

third-party 82

Tornado 1.0.1, using 82

Tornado 2.0, using 82

Tornado 2.2, using 82

virtual memory mapping 170

boards, see target board

boot images 146

boot media 23

512
boot programs

boot parameters, configuring 146
building 146
configuring 145–148
creating new 49
serial connection, configuring for 28
TSFS, for 147

boot ROMs
see also NetROM ROM emulator
compression 195
custom, building 148
emulators, substituting ROM 23
installing 23
makefile ROM-resident system images 177
system image (VxWorks), building a new 491

BOOT_EXTRA 446
modifying run-time 189

bootable applications 191–195
auto scaling 128
creating 128
definition 77
initialization routines, adding application 128
linking 191
size of 192

Bootable System icon 80

bootChange() 237

BOOTCONFIG 445
booting 41–50
@ command 47
commands 44
display, boot 47
networks, initializing 503
parameters 42–47
command-line format 49
displaying current, at boot time 43
nonvolatile RAM, effect of 49
setting 42
rebooting 50
startup scripts 505
TFTP, requiring 49
troubleshooting 58
TSFS, using 50

BOOTINIT 445

bootrom 180
bootrom_res 180

bootrom_res_high 177
bootrom_uncmp 180
bootrom_uncmp.hex 192
CTRL+BREAK (interrupt) 246

break thread command (debugger) 363–364
breakpoints
multiple, setting 337
setting 339
shell commands for 237
temporary, setting 339
threads, setting in 363–364

Breakpoints command (Debug menu) 337

browse() 238

browser (HTML)
specifying 424

browser (Tornado) 303–328
application module information,
displaying 318–320
behavior of, controlling 306
buttons 306
class information, displaying 317
CPU utilization, reporting 320
customizing 328
interrupt/exception vector table,
displaying 322
limitations 324
memory partition information, displaying 315
memory usage data, displaying 310
message queue information, displaying 314
object information, displaying 311–317
semaphore information, displaying 313
spy utility 320
stack checks, displaying 321
starting 304
target information, displaying 307
task information, displaying 312
tasks, listing running 309
troubleshooting with 324–328
updating displays 306
watchdog information, displaying 316
window display, choosing 304

Browser command (Tools menu) 304

Browser Configuration dialog box 306

browser.tcl 328

BSP, see board support packages
bsp2prj target 196
bss segment 497
Build dialog box 126
Build menu
  adding/removing custom commands 421–423
Build Output command (Window menu) 54
Build Output window 106
Build Specification icon 81
build specifications 129–137
  see also Properties: Build specification window
  assembler options, specifying 134
  changing 130
  compiler options, specifying 132
  creating new 136
  current build, selecting for 137
  downloadable application 103
  link order, specifying 134
  linker options, specifying 135
  makefile macros, specifying 132
  makefile rules, working with 131
  pre-set build macros 105
  property sheets, working with 130–135
Build toolbar
  buttons 108
  displaying 107
building
  boot programs 146
  boot ROM, custom 148
  build rules, custom 85
  COM/DCOM projects 393
  custom VxWorks 125
    command line, using a 127
    downloadable applications 101–109
    VxSim applications 202
built-in functions 160
bus, see VMEbus
buttons
  browser, Tornado 306
  Build toolbar 108
  debugger 331–333
  standard toolbar 65
  Tornado Launch toolbar 52
  WindView event triggering 53

C
  -c compiler option 159
  C interpreter (WindSh) 247–264
    address, setting current 254
    arrays 257
    assignments 254
    commands, built-in 259
    comments 255
    data types, handling 247
    function calls 251
    limitations 258
    literals 250
    operators 251
    pointers 257
    redirecting I/O 261–264
    startup scripts 264
    statements, handling 249
    strings 256
      declaring 249
      subroutines as commands 252
    tasks
      current, setting 254
      referencing 253
    Tcl interpreter
      quick access to 273
      toggling to 271
      variable references 250
  c() 236
    current task, using 254
    system mode, in 243
  C++ support 264–267
    compiling application modules
      Diab, using 164
      GNU, using 160
    demangling function names 266
    host and target strategies, using different 238
    initializing 504
    overloaded function names, reconciling 265
    shell commands for 238
  C++_COMPILER 187
  C++_WARNINGS 187
  C++FLAGS 187
  cables, connecting 25
Index

cache
  initializing 497
call command (debugger) 355
CAPS LOCK key 53
Cascade command 54
CC_COMPILER 187
CC_DEFINES 188
CC_INCLUDE 188
CC_OPTIM 188
CC_WARNINGS 187
cd() 233
CFLAGS 187
  Diab compiler 162
  GNU compiler 158
character arrays, see strings
characters, control, see control characters
checkStack() 231
classes (kernel objects)
  displaying information about (browser) 317
classShow() 238
clocks
  system 505
Close All command 54
Close command (File menu) 68
code
  see also application modules; code examples;
    object code; source code
  unsupported, directory for 438
code examples
debugger extensions 375–376
makefiles
  sample (mv147) 185
  skeleton for application modules 190
NetROM startup file 32
system mode debugging 244
coding conventions 455–488
C 457–475
code layout 463–466
documentation 471–475
header files 469
module headings 457
naming 466
programming style 468
routines 459
cache initializing 497
call command (debugger) 355
CAPS LOCK key 53
Cascade command 54
CC_COMPILER 187
CC_DEFINES 188
CC_INCLUDE 188
CC_OPTIM 188
CC_WARNINGS 187
cd() 233
CFLAGS 187
  Diab compiler 162
  GNU compiler 158
character arrays, see strings
characters, control, see control characters
checkStack() 231
classes (kernel objects)
  displaying information about (browser) 317
classShow() 238
clocks
  system 505
Close All command 54
Close command (File menu) 68
code
  see also application modules; code examples;
    object code; source code
  unsupported, directory for 438
code examples
debugger extensions 375–376
makefiles
  sample (mv147) 185
  skeleton for application modules 190
NetROM startup file 32
system mode debugging 244
coding conventions 455–488
C 457–475
code layout 463–466
documentation 471–475
header files 469
module headings 457
naming 466
programming style 468
routines 459
file heading 456
Tel 476–488
code layout 481–484
declarations 480
module headings 476
naming 484
outside procedures 479
procedures 477
programming style 485
colors, changing window 408–410
COM
  authenticating servers 396
  building projects 393
  creating applications 378
  proxy and stub code, linking 393
  registering
    proxy DLLs (Windows) 394
    servers 395
    type library 395
  support for, adding 379
  Win32 clients, building 394
COM 380
COM wizard 381–391
  automation datatypes 384
  clients, choosing 387
  generated files 389
COM_CORE 379
COM_SHOW 380
command-line parameters 49
commands, built-in (WindSh)
  C++ support 238
debugging 236
  network information, displaying 240
  system information, reporting 232
  system, modifying the 236
  target routines, differentiating from 259
  task information, reporting 230
  tasks, managing 229
  VxWorks objects, displaying 238
comments (WindSh) 255
compiler environment
  VxSim 202
compiler options
  see also specific compiler options
  build specifications, and 132
debugging and optimization, problems with 133
Diab 163
GNU 159
compiling
C modules
  GNU, using 159
C++ modules
  Diab, using 164
  GNU, using 160
make variables 187
Component folder icon 81
Component icon 81
component tree (VxWorks view) 117
components, VxWorks 77
configuring 116–123
conflicts among, identifying 122
dependencies, calculating 119
descriptions of 117
displaying 116
excluding 118
finding 117
including 118
parameters, changing 123
size of, estimating image 123
compress tool 195
COMPRESSION 446
compression, boot ROM 195
config target directory 430
config.h 172
config.h
  VxSim 205
CONFIG_ALL 189, 445
CONFIG_ALL_H 445
configAll.h 172
  see also configuration
configuration header files 172
configuration management, see version control
Configure Target Servers dialog box 283–292
  authorizations, setting 285
  Change Property drop down list 285
  options, specifying 285–292
  reserving target servers 300
configuring
  see also config.h; configAll.h; configuration header files
alternatives 174
command-line 151–196
directory and files 430
FTP server (Windows 95) 492
host-target communication 14–37
jumpers 24
module (usrConfig.c) 173
NetROM ROM emulator 31–34
network software, host 21–22
option dependencies 175
options (INCLUDE constants) 173
project facility for, using 152
standalone PCs 27
target 20
target hardware 22–26
target server 282–295
VxSim for networking 212
WFTPD (Windows 95) 492
console devices 501
CONSOLE_BAUD_RATE 501
CONSOLE_TTY 501
context-sensitive help
  workspace, in 79
Continue command (F5) (Debug menu) 341
control characters (WindSh) 260
  see also specific control characters; tyLib(1)
conventions
  coding 455–488
    C 457–475
      file heading 456
      Tcl 476–488
  Makefile macros 442
copy character (CTRL+C) 69
editor, from
  shell, in 261
Copy command (Edit menu) 69
cplusCtors() 238
cplusDtors() 238
cplusStratShow() 238
cplusXtorSet() 238
CPU type, defining
  Diab, for 162
  GNU, for 158
VxSim 203
Create Project window 94
cret() 236
cross-development 3–11, 152–196
commands
Diab 161
GNU 157
CrossWind 329–376
see also debugger; debugging; GDB User’s Guide
crosswind.tcl 372
CTRL+A (select all) 69
CTRL+BREAK (interrupt) 246
CTRL+C
  copy character
editor, from 69
  shell, in 261
  interrupt (console) 246
CTRL+D
  completing symbol and path names 224
  function synopsis, printing 225
  quitting 222
CTRL+F4 (close file) 68
CTRL+G (go to) 70
CTRL+H (delete) 261
CTRL+N (new file) 65
CTRL+O (open file) 67
CTRL+P (print) 72
CTRL+S (save file) 68
CTRL+SHIFT+X (reboot) 242
CTRL+U (delete line) 261
CTRL+V (paste)
editor, from 69
  shell, in 261
CTRL+W (function reference page) 225
CTRL+X (function reference page) 225
CTRL+X
  cut 69
  reboot 50
CTRL+X (reboot)
  VxSim 200
CTRL+Z (undo edit) 69
cursor, text 68
  navigating 70
Customize Tools dialog box 413–420
  macros, using 416
  customizing Tornado 399–424
see also initialization files
Build menu 421–423
colors, changing window 408–410
debugger attributes 411
download options, setting 400
tool editor 402–404
  alternate, specifying 403
fonts, changing screen 408–410
registry, Tornado 412
Tcl files, using 423
Tools menu 413–420
  alternate editor command, creating (example) 419
  binary utilities commands, creating (example) 420
  macros for custom commands 416
  version control command, creating (example) 418
  Web link to Wind River, creating (example) 420
Tornado 1.0.1 compatibility, setting 405
version control, configuring 406
customizing VxWorks 109–128
see also components
  building VxWorks 125
  command line, using a 127
  components, configuring 116–123
  image type, selecting 124
  project files, creating 113
  projects, creating 110
Cut command (Edit menu) 69

D

-D arch compiler option 163
d() 234
  current address, using 254
data types 247
data variables 250
DATASEGPAD 445
DCOM 380
DCOM (VxDCOM option)
  authenticating servers 396
  building projects 393
creating applications 378
demo, supporting the 380
proxy and stub code, linking 393
registering
  proxy DLLs (Windows) 394
  servers 395
type library 395
support for, adding 379
Win32 clients, building 394
DCOM wizard 381–391
  automation datatypes 384
generated files 389
DCOM_OPC 380
DCOM_PROXY 380
DCOM_SHOW 380
-DCPU compiler option
  Diab 164
  GNU 159
Debug menu 331
  Add Symbols command 375
  Attach command 343
  Back Trace command 349
  Breakpoints command 337
  Continue command 341
  Detach command 344
  Download command 358
  Interrupt Debugger command 337
  Memory command 350
  Registers command 348
  Run command 336
  Source Search Path command 335
  Step Into command 341
  Step Out command 341
  Step Over command 342
  Stop Debugging command 334
  Toggle Breakpoint command 339
  Toggle Temp. Breakpoint command 339
  Variables command 347
  Watch command 346
debugger (CrossWind) 329–376
  see also debugging; GDB User’s Guide
command-line interface 352–359
customizing 411
  Tcl, using 372–376
code examples 375–376
GDB Tcl interpreter 365–372
  commands, submitting 365–366
  expressions, evaluating 368
  GDB command line, using 366
  GDB facilities, invoking 368–370
  GUI Tcl interpreter, switching to 374
  I/O file descriptors, closing 369
  linked lists, traversing 370–372
  list of Tcl elements, returning 369
  redirecting I/O 369
  routine names, returning 370
  scripts, debugging GDB-based 367
  symbolic addresses 370
  symbols
    returning local 369
    testing for 370
  verbose error printing 367
.gdbinit, disabling 358
GUI 330–333
  extensions, creating 375–376
  code examples 375–376
  GDB Tcl interpreter, switching to 374
  hexadecimal, displaying 411
  initialization files 353
  customizing GUI 372–374
  internal parameters, changing 358–359
  menu commands 331–333
  quitting 334
  redirecting I/O 355
  source search path, setting 335
  starting 334
  toolbar 331–333
  unloading modules 336
debugging 333–350
  see also debugger; errors; GDB User’s Guide
application modules
  loading 354
    option flags 358
  unloading 354
breakpoints
  multiple, setting 337
  setting 339
  temporary, setting 339
  using 337–342
calling sequences, inspecting 349
command-line 352–359
  Tcl procedures, defining 366
examining data 344–350
exit status, reporting 358
expressions, evaluating 355
local variable values, displaying 347
naming session 359
object files, specifying 353
primitives for, shell 357
relative paths, translating 358
shell, from the 220
single-stepping 341
source code, displaying 351–352
source search path 335
stack frame summaries, displaying 356
stack size, setting 359
subroutines
  executing while debugging 355
symbol values
  monitoring 346
system mode 359–365
code example 244
  shell, from the 242–246
target memory, displaying 350
target register values, displaying 348
targets, connecting to 357
task mode 360
tasks
  attaching 343
  interrupting 337
  priority, setting 359
  releasing 344
  restarting 341
  running 336
threads, working with 360–365
timeout, download 359
WTX protocol requests, sending 357
defs.bsp include file 186
DELETE key
  projects, removing 109
delete-line character (CTRL+U) 261
demangling, see name demangling
demo host directory 436
depend.bspname file
  generating 441
dependencies
  component 119
  makefile 102
Dependencies dialog box 102
Detach command (Debug menu) 344
development environment 2–11, 171
devs( ) 232
Diab tools
  downloadable applications, and 106
directory tree 427–438
disassembling (l()) 234
Disassembly command (View menu) 351
docs directory 427
  see also online documentation
documentation
  online reference pages (on host) 170
documentation guidelines
  conventions
    C format 471–475
Domain Name Service (DNS) 21
dosFs file systems
  initializing 502
dosFsDevInit() 502
dosFsInit() 503
Download command (Debug menu)
  option flags 358
download options 400
Downloadable Application icon 80
downloadable applications 76
  building 101–108
    errors and messages, displaying 106
    specifications, providing 103
  creating 93–109
  Diab tools, and 106
Tornado 2.2
User’s Guide

- downloading 108
  project files, creating 97
  projects, creating new 94
  downloading 108

- project files, creating 97
- projects, creating new 94

- down% command (debugger) 374
  drivers
    - installing 502

- DSM_HEX_MOD 228
- -DTOOL compiler option
  Diab 164
  GNU 159
- -DTOOL_FAMILY compiler option
  Diab 164
  GNU 159
  dual mode 15

E

- -e_romInit entry point option 194

- Edit menu 69–71
  - Copy command 69
  - Cut command 69
  - Delete command 69
  - Find command 70
  - Find Next command 70
  - Find Previous command 70
  - Go To command 70
  - Next Error/Tag command 70
  - Paste command 69
  - Previous Error/Tag command 70
  - Replace command 70
  - Select All command 69
  - Undo command 69

- edit mode, shell (WindSh) 267–269
- input mode, toggling to 261

- editor (external)
  - alternate
    - Tools menu command for, creating
      (example) 419
    - specifying 403

- editor (Tornado) 63–73
  see also buttons; Edit menu; keyboard shortcuts
  - copying 69
  - customizing 73
  - deleting 69

- errors, going to 70
- insert mode 69
- navigating 70
- overtype mode 69
- pasting 69
- preferences, setting 402–404
- replacing text 70
- save options 403
- searching 70
- selecting all 69
- tabs, setting 403
- undo levels, setting 403
- undoing edits 69
- -ei compiler option 163

- Enhanced Network Driver (END) 27
- entry point 496
- ROM-based VxWorks 194, 508

- environment variables, host 171
  - HOME 438
  - shell behavior, controlling 227
  - Windows 171

- environment, see cross-development; development environment; host environment; target environment

- errorInfo global variable (debugger) 367

- errors
  - Tcl 454
  - unwinding 454

- ESC key (edit/input mode toggle) 261

- event triggering, starting 53
- -ew compiler option 163

- exception handling
  see also signals; excLib(1); sigLib(1)
  - initializing 501

- excInit() 502
- excTask() 502
- excVecInit() 497
- exit() 222

- expressions, C language (WindSh) 249–254
  see also data variables; function calls; literals; operators

- External Dependencies folder
  - makefile dependencies, listing 102
- External Dependencies folder 99

- external mode, see system mode
F

F3 (find next) 70
F4 (next error) 70

File menu 65
  Add Project to Workspace command 109
  Close command 68
  Open command 67
  Page Setup command 72
  Print command 72
  Save As command 68
  Save command 68

file tree 427–438

files
  see also editor; navigation
  attribute panes 66
  closing 68
  configuration header 172
  opening 67
  page, setting up 72
  printing 72
  saving 68
  saving as 68
  types 65

Find command (Edit menu) 70
Find dialog box 70
Find Next command (Edit menu) 70
Find Object dialog box 117
Find Previous command (Edit menu) 70

flags, see compiler options

floating-point support

  initializing 503

floatInit() 503

-fno-builtin compiler option 160

fonts, changing screen 408–410

frame command (debugger) 356

Free Software Foundation (FSF) 156
  see also GNU ToolKit User’s Guide

FTP (File Transfer Protocol)

password, user (boot parameter) 46

server (WFTPD) (Windows 95) 491–493
  configuring 492

starting 492

ftp password 492

function calls (WindSh) 251

  parentheses, omitting 253

  reference page, displaying HTML 225

  synopses, printing 225

G

-g compiler option

  Diab 163

  GNU 159

GDB (GNU debugger)

  see also debugger; debugging

  customizing with Tcl 353

  initialization files 353

gdb command (debugger) 368

gdb.tcl 353

gdbEvalScalar command (debugger) 368

gdbFileAddrInfo command (debugger) 369

gdbFileLineInfo command (debugger) 369

  .gdbinit 353

gdbIOClose command (debugger) 369

gdbIORedirect command (debugger) 369

gdbLocalsTags command (debugger) 369

gdbStackFrameTags command (debugger) 370

gdbSymbol command (debugger) 370

gdbSymbolExists command (debugger) 370

General Public License (GNU) 156

GNU binary utilities, working with 420

GNU ToolKit 6, 156

  General Public License 156

  Go To command (Edit menu) 70

H

h target directory 432
h() 233
| hardware       | initializing 497 |
|               | header files 153–156 |
|               | see also configuration header files; INCLUDE constants |
|               | ANSI 154 |
|               | function prototypes 153 |
| CPU type      | Diab, defining for 162 |
|               | GNU, defining for 158 |
|               | hiding internal details 156 |
|               | internal VxWorks 155 |
|               | nested 155 |
|               | private 156 |
|               | searching for 154 |
|               | Tornado tools 428 |
|               | VxWorks-supplied 432 |
| help          | context-sensitive |
|               | workspace, in 79 |
|               | online |
|               | reference documentation 117 |
| Help menu     | About Tornado command 65 |
|               | help set command (debugger) 358 |
|               | help() 233 |
|               | HEX_FLAGS 446 |
|               | modifying run-time 189 |
| history facility, shell | 267 |
| HOME          | host environment variable 438 |
| host          | communication modes 14 |
|               | directory and file tree 428–429 |
|               | network software |
|               | configuring 21–22 |
|               | initializing 21 |
|               | system names and addresses, establishing 22 |
|               | setting up 17–19 |
| host directory | 428–429 |
| host shell, see shell | |
| host-os host directory | 428 |
| hosts file (Windows) | 22 |
| hostShow()    | 240 |

- I compiler option 154
  - Diab 164
  - GNU 159
i() 231
  - system mode, in 243
icmpstatShow() 240
icons, Workspace 80
ifShow() 240
INCLUDE constants 173
  - see also specific constants
include files
  - see also header files
  - configuration headers 167
make facility 186
include host directory 428
INCLUDE_COM 380
INCLUDE_COM_SHOW 380
INCLUDE_CPLUS 504
INCLUDE_DOSFS 502
INCLUDE_EXC_HANDLING 502
INCLUDE_EXC_TASK 502
INCLUDE_FLOATING_POINT 503
INCLUDE_HW_FP 503
INCLUDE_INSTRUMENTATION 505
INCLUDE_IO_SYSTEM 500
INCLUDE_LOGGING 502
INCLUDE_MMU_BASIC 504
INCLUDE_MMU_FULL 504
INCLUDE_NET_INIT 503
INCLUDE_NTPASSFS 202
INCLUDE_PIPE 502
INCLUDE_PROTECT_TEXT 504
INCLUDE_PROTECT_VEC_TABLE 504
INCLUDE_RAWFS 502
INCLUDE_RT11FS 502
INCLUDE_SIGNALS 502
INCLUDE_SM_OBJ 504
INCLUDE_SPY 503
INCLUDE_STARTUP_SCRIPT 505
INCLUDE_STDIO 502
INCLUDE_SW_FP 503
INCLUDE_TIMEX 503
INCLUDE_TSFS_BOOT_VIO_CONSOLE 148
INDEX

INCLUDE_TTY_DEV  501
INCLUDE_USER_APPL  191
INCLUDE_WDB  505
INCLUDE_WDB_BANNER  142
INCLUDE_WDB_COMMPIPE  201
INCLUDE_WDB_START_NOTIFY  142
INCLUDE_WDB_TTY_TEST  143
INCLUDE_WDB_USER_EVENT  142
INCLUDE_WDB_VIO  142
info threads command (debugger) 362
inhibit-gdbinit (debugger) 358
initialization 495–508

see also usrcfg(1)
board support package 170
C++ support 504
cache 497
dosFs file systems 502
exception handling facilities 501
floating-point support 503
hardware 497
interrupt vectors 497
I/O system 500
kernel 498–499
logging 501
memory pool 499
MMU support 504
multitasking environment 498–499
network 503
network software, host 21
pipes 502
rawFs file systems 502
rt11Fs file systems 502
sequence of events, VxWorks 495
ROM-based 508–509
summary 506–508
shared-memory objects (VxMP option) 504
standard I/O 502
sysinit() 496
system clock 500
target agent 505
usrinith() 496–498
usrRoot() 499–505
virtual memory (VxVMI option) 504
WindView 505
initialization files 438

input mode (WindSh) 261
INSERT key 69
insertion point 68
installing

see also Tornado Getting Started Guide
boot ROMs 23
drivers 502
target boards 24
in backplane 24
VxSim optional product 208–211
Interface Definition Language (IDL) 389
editing files 389
Interrupt Debugger command (ALT+SHIFT+F5)
(Debug menu) 337
interrupt key
circle (CTRL+C) 246
Tornado (CTRL+BREAK) 246
interrupt/exception vector table 322
interrupts
thrashing 500
vectored
initializing 497
VxSim
Windows 205
intLockLevelSet() 498
intUnlock()
WindView, and 207
intVecBaseSet() 497
intVecShow() 240
I/O
redirecting
C interpreter 261–264
debugging, during 355
shconfig, using 223
virtual 11
I/O system
initializing 500
standard input/output/error 501
standard I/O
initializing 502
ioGlobalStdSet() 501
iosDevShow() 239
iosDrvShow() 239
iosFdShow() 239
iosInit() 500
ipstatShow()  240
iStrict()  230

J
jumpers  24

K
kernel  4
    excluding facilities  174
    execution, start of  496
    initializing  498–499
kernelInit()  498–499
keyboard settings  53
keyboard shortcuts
    ALT+3 (display symbol values)  346
    ALT+4 (display local variables)  347
    ALT+5 (display register values)  348
    ALT+6 (target memory)  350
    ALT+7 (back trace)  349
    ALT+D h (source search path)  335
    ALT+F (open recently-used file)  67
    ALT+F t (print)  72
    ALT+F3 (find)  70
    ALT+F6 (attach debugging)  343
    ALT+Fa (save as)  68
    ALT+SHIFT+F5 (interrupt)  337
    BACKSPACE (delete)  261
    CTRL+A (select all)  69
    CTRL+BREAK (interrupt)  246
    CTRL+C
        copy
text editor, from  69
        shell, in  261
        interrupt (console)  246
    CTRL+D
        completing symbol and path names  224
        function synopsis, printing  225
        quitting  222
    CTRL+F4 (close file)  68
    CTRL+F5 (step over)  342
CTRL+G (go to)  70
CTRL+H (delete)  261
CTRL+N (new file)  65
CTRL+O (open file)  67
CTRL+P (print)  72
CTRL+S (save file)  68
CTRL+SHIFT+X (reboot)  242
CTRL+U (delete line)  261
CTRL+V (paste)
        editor, from  69
        shell, in  261
CTRL+W (function reference page)  225
CTRL+X
    cut  69
    reboot  50
CTRL+Z (undo edit)  69
DEL (delete)  69
ESC key (input/edit mode toggle)  261
F10 (step into)  341
F3 (find next)  70
F4 (next error)  70
F5 (continue debugging)  341
F6 (run task)  336
F8 (temporary breakpoint)  339
F9 (toggle breakpoint)  339
line editing, shell (WindSh)  267–269
navigation, for file  70
SHIFT+F3 (find previous)  70
SHIFT+F4 (previous error)  70
SHIFT+F5 (step out)  341
kill, see stopping

L
L()  234
    current address, using  254
Launch Browser dialog box  304
Launch Shell dialog box  221
Id()  165–167, 223
    path name orthography  257
LD_CALL_XTORS  227
LD_COMMON_MATCH_ALL  228
LD_PATH  228
LD_SEND_MODULES  228
LD_SEND_MODULES facility 166
ldarch linker
   flags 182
LDFLAGS 189
lib target directory 433
LIB_EXTRA 189, 443
LIBS 105
LICENSE.TXT 428
linking
   application modules 164
      bootable 191
dynamic 223
   system image, VxWorks 182
      flags 183
      object modules, additional 184
             VxSim, and 203
linkSyms.c 114
list (Tcl) 449
literals (WindSh) 250
see also strings
lkAddr() 232
lkup() 233
load command (debugger) 165–167, 354
LOCAL_MEM_LOCAL_ADRS 178
Lock on Startup button (Configure Target Servers
   dialog box) 300
locking (target server) 300
Log Console window (target server) 295
logging facilities
   initializing 501
logInit() 502
logTask() 502
ls() 233

M
m() 237
MACH_EXTRA 442
   adding object modules 184
   modifying run-time 189
make facility (GNU) 180
Makefile 441–446
   customizing 441–446
   macros 442–446
bootConfig.c file, alternate 445
bootInit.c file, alternate 445
compile-time macros, specifying 443
compression program 446
configAll.h file, alternate 445
configuration files, modifying 445
conventions 442
dataSegPad.s file, alternate 445
defining without modifying source
   code 444
header directory locations, adding 443
host object-format-to-hex program 446
library archives, including 443
object modules, adding 442
   linked to compressed ROM
   images 446
   without touching source 444
preprocessor flags, adding 446
target directory tree 445
usrConfig.c file, alternate 445
Makefile (VxWorks) 180
   linking system images 182
   VxSim 205
Makefile file 115
makefile rules, custom 131
makefiles 185–191
   bootable applications, modifying for 191
code examples
      sample (mv147) 185
   skeleton for application modules 190
creating 185
dependencies, calculating 102
include files 186
   application modules, and 190
rebuilding VxWorks 180
ROM-resident system images 177
variables, include file 186–190
   BSP parameters, for 188
   compiling, for 187
   customizing run-time, for 189
makeSymTbl tool 193
malloc() 499
Manage command (Tools>Target Server
   menu) 298–301
Manage Target Servers dialog box  298–301
  authorizations, setting  300
  Select Action drop-down list  298
mangling, see name demangling
mathHardInit()  503
mathSoftInit()  503
memAddToPool()  499
memDesc.c  170
memory
  see also memory pool; shared-memory objects;
    virtual memory; memLib();
    memPartLib(1)
  allocation  499
  availability of, determining  499
  cache size, changing target server  289
  fragmentation, troubleshooting  325
  leaks, troubleshooting  324
  target, manipulating  237
  usage, displaying data about  310
Memory command (ALT+6) (Debug menu)  350
memory partitions
  displaying information about (browser)  315
memory pool
  adding to  499
  initializing  499
memPartLib  499
memPartShow()  239
memShow()  239
message queues
  displaying information about (browser)  314
migrating
  files between projects  86
MIPS
  ROM-resident images  177
Mixed Source and Disassembly command
  (View menu)  351
MMU
  initializing  504
moduleIdFigure()  240
modules
  see also application modules; tasks
    optional (INCLUDE constants)  173
moduleShow()  239
mqPxShow()  239
mRegs()  236
msgQShow()  239

N
name demangling (C++)  266
navigating  70
  Go To command  70
netLibInit()  503
NetROM ROM emulator  23
  back end connection, as  30–37
  code example  32
  configuring  31–34
  target agent for, configuring  31
  target server, starting  34
  troubleshooting  35–37
netstat command  59
netstatShow()  240
networks
  configuring simple  21–22
  excluding from VxWorks  175
  initializing  503
  target-server back end connection, as  26
  VxSim  207–217
New command (File menu)  65
Next Error/Tag command (Edit menu)  70
NUM LOCK key  53
NUM_TTY  501

O
-O compiler option  160
-O2 compiler option  160
Object folder icon  81
Object Modules folder  99
object modules, see application modules
online documentation
  function reference pages, displaying  225
  reference pages (on host)  170
OPC interfaces (DCOM)
  non-automation types  389
Open command (File menu)  67
operators (WindSh) 251
  using < and > 261
OPT_TERMINAL 501
optional VxWorks features (INCLUDE constants) 173
optional VxWorks products
  VxMP 4
  VxSim, target simulator 8, 197
  VxVMI 4
  WindView 8
Options command (Tools menu) 400
Options window 400–413

P

Page Setup command (File menu) 72
Page Setup dialog box 72
passFs (VxSim) 201
paste character (CTRL+V)
  editor, from 69
  shell, in 261
Paste command (Edit menu) 69
path names
  shell and target server 270
  typing conventions 256
performance monitoring
tools for, including 503
period() 229
PHYS_MEM_DESC 170
  see also sysPhysMemDesc]
ping command 59
pipeDevCreate() 502
pipeDrv() 502
pipes
  initializing 502
  pointers (WindSh) 257
  arithmetic, handling 258
POST_BUILD_RULE 105
PowerPC
  ROM-resident images 177
Previous Error/Tag command (Edit menu) 70
primitves (WindSh)
  debugging 357
Print command (File menu) 72
Print dialog box 72
printErrno() 232
printLogo() 233
priority inversion
  troubleshooting 326
prj_default target 195
prj_diab target 196
prj_diab_def target 196
prj_gnu target 196
prj_gnu_def target 196
PRJ_LIBS 100
prjComps.h 115
prjConfig.c 115
prjParams.h 115
processor number (boot parameter) 45
project facility 75–145
  bootable applications, creating 128
  BSPs with, using pre-existing 82
  customizing VxWorks 109–128
  downloadable applications, creating 93–109
  GUI 78
  makefile dependencies, calculating 102
  manual configuration, versus 76, 168
  target-host communication interface,
    configuring 138–145
  terminology 76
project files, see application files
projectName.wpj 115
projects 75–149
  see also applications; bootable applications;
    customizing VxWorks; downloadable
    applications; sub-projects; target agent
    adding 109
    application source code, adding 84
    architecture-independent 86
    architecture-independent applications 86
    boot programs, using new 145–148
    BSP, getting a functioning 81
    BSPs, building from 195
    build rules, custom 85
    creating 109
    based on existing project 111
    bootable applications, for 128
    BSP simulator, using 83
    downloadable applications, for 94
project wizard, using 84
customized VxWorks, for 110
definition 77
external build 87
linking to 135
organizing the build 87
planning 81–93
removing 109
sub-projects 88–93
Properties: Build specification window 103
Macros tab 105
makefile macros, viewing 132
Rules tab 103
pwd() 233

Q
quit() 222
quitting
see also stopping
debugger 334
shell sessions 222

R
-r linker option 165
RAM_HIGH_ADRS 188
RAM_LOW_ADRS 188
text point 496
rawFs file systems
initializing 502
rawFsDevInit() 502
rawFsInit() 503
README.TXT 428
reboot character (CTRL+X) 50
reboot character (CTRL+X)
VxSim 200
reboot() 242
rebooting 50
shell, from the 242
rebuilding VxWorks image 180
redirection, I/O (WindSh) 261–264
scripts, using 262
shConfig, using 223
reference entries
conventions, writing 471
external build 87
linking to 135
organizing the build 87
planning 81–93
removing 109
sub-projects 88–93
Properties: Build specification window 103
Macros tab 105
makefile macros, viewing 132
Rules tab 103

Q
quit() 222
quitting
see also stopping
debugger 334
shell sessions 222

R
-r linker option 165
RAM_HIGH_ADRS 188
RAM_LOW_ADRS 188
text point 496
rawFs file systems
initializing 502
rawFsDevInit() 502
rawFsInit() 503
README.TXT 428
reboot character (CTRL+X) 50
reboot character (CTRL+X)
VxSim 200
reboot() 242
rebooting 50
shell, from the 242
rebuilding VxWorks image 180
redirection, I/O (WindSh) 261–264
scripts, using 262
shConfig, using 223
reference entries
conventions, writing 471
external build 87
linking to 135
organizing the build 87
planning 81–93
removing 109
sub-projects 88–93
Properties: Build specification window 103
Macros tab 105
makefile macros, viewing 132
Rules tab 103

Q
quit() 222
quitting
see also stopping
debugger 334
shell sessions 222

R
-r linker option 165
RAM_HIGH_ADRS 188
RAM_LOW_ADRS 188
text point 496
rawFs file systems
initializing 502
rawFsDevInit() 502
rawFsInit() 503
README.TXT 428
reboot character (CTRL+X) 50
reboot character (CTRL+X)
VxSim 200
reboot() 242
rebooting 50
shell, from the 242
rebuilding VxWorks image 180
redirection, I/O (WindSh) 261–264
scripts, using 262
shConfig, using 223
reference entries
conventions, writing 471
external build 87
linking to 135
organizing the build 87
planning 81–93
removing 109
sub-projects 88–93
Properties: Build specification window 103
Macros tab 105
makefile macros, viewing 132
Rules tab 103

Q
quit() 222
quitting
see also stopping
debugger 334
shell sessions 222

R
-r linker option 165
RAM_HIGH_ADRS 188
RAM_LOW_ADRS 188
text point 496
rawFs file systems
initializing 502
rawFsDevInit() 502
rawFsInit() 503
README.TXT 428
reboot character (CTRL+X) 50
reboot character (CTRL+X)
VxSim 200
reboot() 242
rebooting 50
shell, from the 242
rebuilding VxWorks image 180
redirection, I/O (WindSh) 261–264
scripts, using 262
shConfig, using 223
reference entries
conventions, writing 471
external build 87
linking to 135
organizing the build 87
planning 81–93
removing 109
sub-projects 88–93
Properties: Build specification window 103
Macros tab 105
makefile macros, viewing 132
Rules tab 103

Q
quit() 222
quitting
see also stopping
debugger 334
shell sessions 222

R
-r linker option 165
RAM_HIGH_ADRS 188
RAM_LOW_ADRS 188
text point 496
rawFs file systems
initializing 502
rawFsDevInit() 502
rawFsInit() 503
README.TXT 428
reboot character (CTRL+X) 50
reboot character (CTRL+X)
VxSim 200
reboot() 242
rebooting 50
shell, from the 242
rebuilding VxWorks image 180
redirection, I/O (WindSh) 261–264
scripts, using 262
shConfig, using 223
reference entries
conventions, writing 471
external build 87
linking to 135
organizing the build 87
planning 81–93
removing 109
sub-projects 88–93
Properties: Build specification window 103
Macros tab 105
makefile macros, viewing 132
Rules tab 103

Q
quit() 222
quitting
see also stopping
debu...
S

s() 237

current task, using 254

system mode, in 244

Save As command (File menu) 68

Save command (File menu) 68

calability 174–176

VxWorks features 173

scripts, startup 505

SCROLL LOCK key 53

searching 70

security

target server authorizations 299

Select All command (Edit menu) 69

Selection folder icon 81

semaphores

displaying information about (browser) 313

semPxShow() 239

semShow() 239

serial lines

target-server back end connection, as 27–30

testing 28

servutil.exe service utility (Windows NT) 489–490

set command (debugger) 358–359

SetTimer() 206

SETUP directory 427

setup, see configuration

setup.log log file 428

SH_GET_TASK_IO 227

share directory 427

shared-memory objects (VxMP option)

initializing 504

shConfig (Tcl)

redirecting I/O 223

shell environment variables, setting 227

shell (WindSh) 219–279

see also C interpreter; C++ support; commands, built-in

C control structures 258

C interpreter 247–264

C++ support 264–267

calculating in 226

commands, built-in 229–241

differentiating from target routines 229

list of 259

components 277

control characters 260

data conversion 226

debugging from 220

debug mode 267–269

input mode, toggling to 261

evironment variables, setting 227

function synopsis, printing 225

history facility 267

input mode 261

interpretation, layers of 279

interrupting 246

line editing 267–269

linker, dynamic 223

path names

orthography 270

typing 224

preprocessor macros 259

rebooting 242

redirecting I/O 223

starting 221

startup scripts 264

stopping 222

system mode debugging 242–246

target from host, controlling 275

target routines from, running 241

target symbol names, typing 224

Tcl interpreter 271–275

initializing with 274

starting 271

types, compound and derived 258

Shell command (Tools menu) 221

shellHistory() 233

shellPromptSet() 233

SHIFT+F3 (find previous) 70

SHIFT+F4 (previous error) 70

show command (debugger) 358

show routines (shell) 238

show() 238

shParse utility (Tcl) 273

sigInit() 502

simulator, see VxSim

simulator, target (VxSim) 8

starting 53
single-stepping
  shell commands for 237
smMemPartShow() 239
smMemShow() 239
so() 236
source code (VxWorks)
  customizing 181
source code directories (VxWorks) 433
source control, see version control
Source command (View menu) 351
source search path 335
Source Search Path command (ALT+Dh)
  (Debug menu) 335
sp() 229
  system mode, in 244
sps() 229
spy utility 503
spy utility (browser) 320
  see also spyLib(1)
  reporting mode, selecting 320
src target directory 433
src/demo host directory 436
stacks
  overflow, troubleshooting 325
  usage, checking (browser) 321
STA N D A L O N E flag 193
standalone systems, using 27
standalone VxWorks systems 193
STA N D A L O N E_NET 193
standard file headings 456
standard I/O
  initializing 502
standard input/output/error 501
standard toolbar 64
starting
  browser, Tornado 53
debugger 53
  event triggering (WindView) 53
shell, Tornado 53
target servers 54
Tornado 51–54
VxSim target simulator 53
WindView 53
startup
  see also initialization
  entry point 496
  ROM-based 194
  scripts 505
  VxWorks, sequence of events 495–508
  ROM-based 508–509
state-information files 439
statements (WindSh) 249
status line (Tornado) 53
  _STD C_ 153
Step Into command (F10) (Debug menu) 341
Step Out command (SHIFT+F5)
  (Debug menu) 341
Step Over command (CTRL+F5)
  (Debug menu) 342
Stop Debugging command (Debug menu) 334
stopping
  shell (WindSh) 222
target servers 296
strings 256
  declaring 249
sub-projects 88–93
  absolute paths, avoiding 91
subroutines, see function calls; commands, built-in
subroutines, see routines
symbol table
  creating VxWorks system 184
group numbers, module 166
  standalone systems, in VxWorks 193
  synchronizing 289
target server, configuring 288
symbols
  VxSim 201
sysALib.s 115, 170
  entry point 496
sysClkConnect() 500
sysClkRateSet() 500
sysHwInit() 497
sysHwInit2() 500
sysInit() 496
sysLib.c 115, 169
  VxSim 205
sysMemTop() 499
sysPhysMemDesc[ ] 170
sysResume() 237
    mode switching, under 243
sysStatusShow() 237
    mode switching, under 243
sysSuspend() 237
    mode switching, under 243
system clock 505
    initializing 500
system controller 25
system image
downloading 496
    VxWorks
        boot ROMs, building with 491
system images 109
see also customizing VxWorks
boot ROM
    compressed 180
    ROM-resident 180
    uncompressed 180
building 178–184
creating
    ROM, in 194
    standalone 193
excluding facilities 174–176
linking 182
makefiles, ROM-resident in 177
size of, considering 84
Tornado
    basic 180
    ROM, in 180
    standalone 180
type, selecting 124
    VxWorks
        creating custom 109–128
system library 169
system mode 14
debugging 359–365
    code example 244
    initiating 361
    shell, from the 242–246
    threads, managing 361–365
    shell commands, using 243
    target agent 9
system mode debugging
    VxSim 201
networking, and 213
system modules, see system image

T
-T option (Tcl) 271

Target
see also browser (Tornado)
    communication modes 14
    configuring 20
    ROM emulation 31
directory and file tree 430–438
    information about, displaying (browser) 307
    monitoring state of (browser) 303–328
Target agent 9
    communication modes 14
    END drivers, configuring 139
    exception hooks, configuring for 143
    initializing 505
    kernel, starting before 143
    NetROM, configuring for 31
    macros, specifying 139
    networks, configuring for 141
    scaling 142
    serial connections, configuring for 141
    simulators, configuring integrated target 139
    system mode 9
    task mode 9
tyCoDrv connections, configuring 142
Target board
    backplane, installing boards in 24
    cables, connecting 25
    configuration header for 172
    configuring 22–26
    installing 24
    jumpers, setting 24
    processor number 45
    serial port 25
    terminal emulator 25
Target directory 430–438
target environment 4–5
Target Information window 307
target name (tn) (boot parameter) 46
target server 281–301

see also WTX protocol

authorizations 299

back ends 286

configuring 282–295

command line, using a 294

options, specifying 285–292

connections

END (Enhanced Network Driver) 27

network 26

ROM emulator 30–37

serial line 27–30

core file 288

icon 295

information about, displaying (browser) 307

locking 300

log output, displaying 295

memory cache size, changing 289

rebooting 299

reserving 300

security 299

selecting 296

sharing 299

starting 54

status information, displaying 56

stopping 296

Manage Target Servers dialog box 299

symbol table 288

synchronizing 289

troubleshooting 60

unregistering 299

unreserving 300

virtual consoles, creating 291

Target Server File System (TSFS) 290

boot program for, configuring 147

target wtx command (debugger) 357

target nr 170

TARGET_DIR 188

target-specific development

VxSim, simulated target 197–217

task mode 14

debugging 360

target agent 9

taskCreateHookShow() 239

taskDeleteHookShow() 239

taskIdDefault() 230

taskIdFigure() 231

taskRegsShow() 239

tasks

see also threads

CPU utilization, reporting 320

current, setting 254

debugging 361–365

displaying information about (browser) 312

IDs 253

interrupting 337

listing (browser) 309

names 253

referencing 253

shell commands for managing 229–232

stack checks, displaying (browser) 321

taskShow() 238

taskSwitchHookShow() 239

taskWaitShow() 239

Tcl (tool command language) 447–454

arithmetic expressions 451

arrays, associative 450

browser initialization files 328

C applications, integrating with 454

coding conventions 476–488

command substitution 451

customizing

browser, Tornado 328

debugger 372–376

GDB 353

initialization files 423

shell 274

debugger (CrossWind)

GDB interpreter 365–372

GUI interpreter 372–376

error handling 454

files 452

formatting 452

I/O 452

linked lists, traversing 370–372

lists 449

procedures, defining 452–453

shell (WindSh) 271–275

C interpreter
quick access from 273
toggling to 271
initialization files 274
target, controlling the 272
Tcl interpreter, starting 271
Tk graphics library 447
unwinding 454
variables 448–449
tcl command (debugger) 365–366
tcl host directory 429
Tcl_CreateCommand() 454
tcldebug command (debugger) 367
tclerror command (debugger) 367
-Tclmode option (Tcl) 271
tclproc command (debugger) 366
TCP/IP
Tornado tools, for 18
tcpstatShow() 240
td() 229
terminal characters, see control characters
terminating, see stopping
text cursor 68
navigating 70
tftpInfoShow() 240
TGT_DIR 445
tgtsvr.exe program 295
options, setting 295
this menu command (debugger: C++) 376
code example 376
thrashing 500
thread command (debugger) 363
thread IDs 361
- breakpoints and system context 364
threads 360
- breakpoints, setting 363–364
- managing 361–365
summary information, displaying 362
switching
explicity 363
implicitly 364–365
ti() 230
current task, using 254
tickAnnounce() 505
Tile command 54
timeout
NetROM 37
resetting
- target server 242
- wtx-load-timeout (CrossWind) 359
- wtxTimeout (WindSh) 223
- target server 287	timers
- subroutine execution 503
Tk graphics library (Tcl) 447
Toggle Breakpoint command (F9)
- (Debug menu) 339
Toggle Temp. Breakpoint command (F8)
- (Debug menu) 339
toolbars 51
see also buttons
Debug 331–333
standard 64
Tornado Launch 52
toolchain 78
Tools menu
- adding/removing custom
  commands 413–420
Browser command 304
Customize command 413
customizing 413–420
- alternate editor, for 419
- binary utilities, for 420
- version control, for 418
- Web link to Wind River, for 420
Debugger command 334
No Custom Tools placeholder 413
Options command 400
Shell command 221
Target Server submenu
- Manage command 298–301
target server, selecting a 297
tools, development 4–8
see also browser; debugger; editor; project
facility; shell; target server
adding/removing Tools menu
commands 413–420
communications, managing 281
Diab 161
GNU 157
TCP/IP, using 18
Tornado settings, changing 400–413
using, guidelines for 4
Tornado 1.0.1 compatibility 405
Tornado Launch toolbar 52
Browser button 304
buttons 53
debugger, starting 334
shell, starting a 221
Tornado Registry icon 17
Tornado Registry window 17
Tornado.tcl 423
TorVars.bat 294
-tPPC403FS:vxworks55 compiler option 163
trt() 229
troubleshooting 57–61
booting problems 58
hardware configuration 57
memory fragmentation 325
memory leaks 324
NetROM ROM emulator connection 35–37
priority inversion 326
stack overflow 325
target-server problems 60
tsf() 229
TSFS, see Target Server File System
tff() 231
tty command (debugger) 356
ttyDevCreate() 501
ttyDrv() 501
tw() 231

U

udpstatShow() 240
ULIP
installing
VxSim/Windows, for 208
#undef 174
Undo command (Edit menu) 69
uninstalling
VxSim optional product 211
unld() 167, 236
unload command (debugger) 354
unloading modules

see online unldLib
command line, from the 167
workspace, from the 336
unsupported target directory 438
unwinding (Tcl) 454
uptcl command (debugger) 374
USER_APPL_INIT 191
userAppInit.c 115
initialization calls, adding 128
usrClock() 505
USRCONFIG 445
usrConfig.c 173
usrDepend.c 175
usrInit() 496–498
usrKernelInit() 498
usrMmuInit() 504
usrNetInit() 503
usrRoot() 499–505
usrSmObjInit() (VxMP option) 504

V

-V option (target server) 295
variables
uninitialized 497
Variables command (ALT+4) (Debug menu) 347
variables, see data variables; environment variables, host; Tcl variables
version control 406
Uncheckout command, creating (example) 418
version number, displaying Tornado 65
version() 233
View menu
Disassembly command 351
Mixed Source and Disassembly command 351
Source command 351
virtual console 291
virtual I/O 11
consoles, creating dedicated 291
virtual memory
VxVMI option
initializing 504
vector table protection 504
VMEbus
   backplane, installing boards in 24
   system controller 25
VX_NO_STACK_FILL
   checkStack(), using 231
VxMP (option) 4
VxOPC 380
VxSim 8, 197–217
   BSP differences 205
   built-in product 198–202
   clocks 206
   compiler environment, configuring 203
   compiler options 203
   CPU type, defining 203
   debugging, system mode 201
   networking, and 213
   endianess 204
   exiting 200
   file systems 201
   interrupts
      Windows 205
   multiple simulators, running 213
   networking facilities 207–217
      configuring for 212
      installing 208–211
      ULIP (Windows) 208
      IP addressing 213
      processor numbers, choosing 215
      remote access 215
      uninstalling 211
   optional product 198
   rebooting 200
   starting 53, 199
   symbols, using 201
   timeouts 204
   unsupported features 204
vxsize command 192
VxVMI (option) 4
VxWorks 4–5
   booting 41–50
   configuring 20
   optional products 173
      VxMP 4
      VxSim, target simulator 8
      VxVMI 4
WindView 8
   project files 113
   rebooting 50
   rebuilding 180
   scalable features (INCLUDE constants) 174–176
   simulator (VxSim) 197–217
vxWorks 180
vxWorks.h 153
vxWorks.res_rom 180
vxWorks.res_rom_nosym 180
   creating 194
vxWorks.res_rom_nosym_res_low 177
vxWorks.res_rom_res_low 177
vxWorks.st 180
   creating 193
vxWorks.st_rom 180
   creating 194
vxWorks.sym symbol table 184
vxWorks_rom 180
W
w() 231
   -Wc++ compiler option 163
   -Wa compiler option 163
   -Wall compiler option 160
Watch command (ALT+3) (Debug menu) 346
watchdogs
   displaying information about (browser) 316
   WDB NetRom back end (target server) 287
   WDB Pipe back end (target server) 287
   WDB RPC back end (target server) 286
   WDB Serial back end (target server) 287
WDB_BP_MAX 143
WDB_NETROM_INDEX 139
WDB_NETROM_MTU 139
WDB_NETROM_NUM_ACCESS 139
WDB_NETROM_POLL_DELAY 140
WDB_NETROM_ROMSIZE 140
WDB_NETROM_TYPE 140
WDB_NETROM_WIDTH 140
wdbConfig() 505
wdShow() 239
World Wide Web
  Tools menu link to Wind River, creating (example) 420

writeDebugInput command (debugger) 374
WTX protocol 357
  see also target server requests, sending 357
wx-ignore-exit-status (debugger) 358
wx-load-flags (debugger) 358
wx-load-path-qualify (debugger) 358
wx-load-timeout (debugger) 359
wx-task-priority (debugger) 359
wx-task-stack-size (debugger) 359
wx tactic procedures 272
wx-tool-name (debugger) 359

X

-X compiler option 163
xsym tool 184