Contents

1 Overview .......................................................................................................................... 1
  1.1 Introduction .................................................................................................................. 1
  1.2 What’s in This Book ..................................................................................................... 2

2 Creating a Development Shell with wrenv ............................................................. 5
  2.1 Introduction ................................................................................................................ 5
  2.2 Invoking wrenv .......................................................................................................... 6
  2.3 Options to wrenv ....................................................................................................... 7
  2.4 install.properties and package.properties ............................................................ 9

3 Working with Projects and Components ............................................................. 11
  3.1 Introduction ............................................................................................................... 11
  3.2 Using vxprj ............................................................................................................... 12
    3.2.1 Creating Kernel Configuration Projects ....................................................... 12
      Copying Projects .................................................................................................... 14
      Using Profiles ...................................................................................................... 14
    3.2.2 Deleting Projects .......................................................................................... 15
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3</td>
<td>Modifying Projects</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Adding Components</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>Removing Components</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Setting Configuration Parameter Values</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Changing the Project Makefile Name</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>Adding and Removing Individual Files</td>
<td>18</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Generating Project and Component Diagnostics</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Obtaining a List of Components</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Checking a Component</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Checking Component Dependencies</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Listing Configuration Parameters and Values</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Examining the Source Files in a Project</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Using cmpScriptLib and Other Libraries</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>RTP and Library Projects</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>Building Kernel and Application Projects</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Building Kernel Configuration Projects with vxprj</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>4.2.1 Examining Build Specifications and Rules</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>4.2.2 Changing Build Settings</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>Adding and Changing Build Rules</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Building Projects with cmpScriptLib</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Other VxWorks Project Types</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Calling make Directly</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>RTP Applications and Libraries</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>RTP Applications</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>RTP Libraries</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>Makefile and Directory Structure</td>
<td>31</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Rebuilding VxWorks RTP (User-Mode) Libraries</td>
<td>32</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Make Variables</td>
<td>32</td>
</tr>
</tbody>
</table>
A.6.6 Examining Stack Traces and Frames ..................................................... 77
A.6.7 Displaying Information and Expressions ............................................. 77
A.6.8 Displaying and Setting Variables ........................................................... 78

A.7 Using the Built-in Line Editor ............................................................................. 79
A.7.1 vi-Style Editing .......................................................................................... 79
Switching Modes and Controlling the Editor .................................................. 80
Moving and Searching in the Editor ............................................................... 80
Inserting and Changing Text ............................................................................. 81
Deleting Text ........................................................................................................ 82
Put and Undo Commands .................................................................................. 82
A.7.2 emacs-Style Editing .................................................................................. 83
Moving the Cursor ............................................................................................... 83
Deleting and Recalling Text .............................................................................. 83
Special Commands .............................................................................................. 84
A.7.3 Command Matching ................................................................................ 84
Directory and File Matching ............................................................................ 85
A.7.4 Command and Path Completion ............................................................... 85

A.8 Running the Host Shell in Batch Mode ............................................................. 85

Index ...................................................................................................................... 87
1 Overview

1.1 Introduction

This guide describes the command-line host tools provided with Wind River Workbench. It is a supplement to the Wind River Workbench User’s Guide for programmers who prefer to do most development tasks outside of the Workbench IDE’s graphical interface or who need to create scripted build environments.

The Workbench IDE includes many features, such as code browsing and multicore debugging, that are not available from the command line. For a complete description of the Workbench environment, see the Wind River Workbench User’s Guide. For information about the VxWorks operating system and developing applications that run under it, see the VxWorks Kernel Programmer’s Guide and VxWorks Application Programmer’s Guide.

Workbench ships with two compilers and associated toolkits: The Wind River Compiler (sometimes called the Diab compiler) and GCC (part of the GNU project). Both toolkits use the GNU make utility for VxWorks application development. To the largest extent possible, these toolkits have been designed for interoperability, but it is not always safe to assume that code written for one set of tools will work perfectly with the other. While examples in this guide may use
either or both compilers, it is best to select one compiler and toolkit before starting a project.

The Wind River Compiler and tools are documented in a separate user’s guide for each supported target architecture. For example, information about the PowerPC compiler is in the *Wind River Compiler for PowerPC User’s Guide*. GCC and the other GNU tools are documented in a series of manuals from the Free Software Foundation, including *Using the GNU Compiler Collection*, that are provided with this release.

**NOTE:** Throughout this guide, *UNIX* refers to both Solaris and Linux host development environments. *Windows* refers to all supported versions of Microsoft Windows.

Examples in this guide include both UNIX and Windows command lines. It is assumed that the reader can make appropriate modifications—such as changing forward slashes (/) to backward slashes (\)—depending on the environment.

Workbench includes a variety of other tools whose use is illustrated in this guide. In many cases, readers are referred to another document for detailed information about a tool. This guide is a roadmap for command-line development, but it is not self-contained.

Before starting to work with the tools, read 2. Creating a Development Shell with *wrenv*.

### 1.2 What’s in This Book

This guide contains information about the following topics:

- This chapter introduces the command-line tools and reviews the contents of the guide.
- Chapter 2 shows how to use the environment utility (*wrenv*) to set up a host development shell.
- Chapter 3 explains how to create and modify projects with *vxprj* and other Tcl scripts. It discusses VxWorks components and kernel configuration.
- Chapter 4 explains how to build applications with *vxprj* and other tools, and describes the VxWorks build model.
• Chapter 5 explains how to run compiled applications on targets and simulators.

• Chapter 6 explains how to use the VxWorks host shell to run and debug applications.

• Appendix A contains a list of options and commands for the VxWorks host shell.
2

Creating a Development Shell with wrenv

2.1 Introduction 5

2.2 Invoking wrenv 6

2.3 Options to wrenv 7

2.4 install.properties and package.properties 9

2.1 Introduction

To use the tools efficiently from the command line, you need to configure some environment variables and other settings. The best way to do this is with the wrenv environment utility, which sets up a development shell based on information in the install.properties file.

When using the Workbench tools from the command line, always begin by invoking the environment utility as shown in the next section. The wrenv utility, which is also run by the IDE on startup, guarantees a consistent, portable execution environment that works from the IDE, from the command line, and in automated build systems. Throughout this guide, whenever host operating system commands are shown or described, it is assumed that you are working from a properly configured development shell created by wrenv.
2.2 Invoking wrenv

Assuming a standard installation of Workbench, you can invoke wrenv as follows.

**UNIX**

At your operating system’s shell prompt, type the following:

```
% installDir/wrenv.sh -p vxworks-6.0
```

However, if your shell configuration file overwrites the environment every time a new shell is created, this may not work. If you find that you still cannot invoke the Workbench tools after executing the command above, try this instead:

```
% eval `installDir/wrenv.sh -p vxworks-6.0 -o print_env -f shell`
```

where *shell* is *sh* or *csh*, depending on the current shell program. For example:

```
% eval `./wrenv.sh -p vxworks-6.0 -o print_env -f sh`
```

**Windows**

You can invoke wrenv from the command prompt by typing the following:

```
C:\> installDir\wrenv.exe -p vxworks-6.0
```

An easier method is to use the shortcut installed under **Start > Programs > Wind River > VxWorks 6.0 > VxWorks 6.0 Development Shell**. This shortcut invokes wrenv to open a Windows command prompt configured for Workbench development.

Workbench also supplies a fully configured Windows version of the Z shell (*sh.exe*). The Z shell, sometimes called *zsh*, gives Windows users a UNIX-like command-line interface.
2.3 Options to wrenv

The wrenv utility accepts several options, summarized in Table 2-1, that can be useful in complex build environments. For most purposes, -p is the only option you need.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e</td>
<td>Do not redefine existing environment variables.</td>
<td>% wrenv.sh -p vxworks-6.0 -e</td>
</tr>
<tr>
<td></td>
<td>(Variables identified with addpath in install.properties are still modified.)</td>
<td></td>
</tr>
<tr>
<td>-f format</td>
<td>Select format for print_env or print_vars (see -o):</td>
<td>C:&gt; wrenv -p vxworks-6.0 -o print_vars -f sh</td>
</tr>
<tr>
<td></td>
<td>plain (the default)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>csh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bat</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tcl</td>
<td></td>
</tr>
<tr>
<td>-i path</td>
<td>Specify the location of install.properties.</td>
<td>% wrenv.sh -p vxworks-6.0 -i directoryPath/install.properties</td>
</tr>
<tr>
<td></td>
<td>(Override the default method of finding install.properties.)</td>
<td></td>
</tr>
<tr>
<td>-o operation</td>
<td>Select operation:</td>
<td></td>
</tr>
<tr>
<td>run</td>
<td>The default operation. Configures the environment and creates a command shell in which subsequent commands are executed. Checks the value of SHELL (usually defined under UNIX) to determine which shell program to invoke; if SHELL is not defined, it checks ComSpec (Windows).</td>
<td>C:&gt; wrenv -p vxworks-6.0 -o run % wrenv.sh -p vxworks-6.0 -o run</td>
</tr>
<tr>
<td>print_vars</td>
<td>Show environment settings that would be made if wrenv were invoked with -o run.</td>
<td>C:&gt; wrenv -o print_vars</td>
</tr>
</tbody>
</table>
VxWorks

Table 2-1  Options for wrenv (cont'd)

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>print_env</strong></td>
<td>Like print_vars, but shows only variables that are exported to the system environment. (Such variables are identified with export or addpath, rather than define, in install.properties.)</td>
<td><code>% wrenv.sh -o print_env</code></td>
</tr>
<tr>
<td><strong>print_packages</strong></td>
<td>List the packages defined in install.properties and their attributes. (The displayed name of a package can later be specified with the -p option.)</td>
<td><code>C:\&gt; wrenv -o print_packages</code></td>
</tr>
<tr>
<td><strong>print_compatible</strong></td>
<td>Use with -p. Show the list of packages defined in installation.properties as compatible with the specified package. Helpful for determining which IDE version works with a given target OS platform.</td>
<td><code>% wrenv.sh -p vxworks-6.0 -o print_compatible</code></td>
</tr>
<tr>
<td><strong>print_package_name</strong></td>
<td>Use with -r. Show the name of the package in the specified root directory.</td>
<td><code>C:\&gt; wrenv -r directory -o print_package_name</code></td>
</tr>
<tr>
<td><strong>-p package</strong></td>
<td>Specify a package (a set of Workbench components) for environment initialization. The package must be defined in install.properties.</td>
<td><code>% wrenv.sh -p vxworks-6.0</code></td>
</tr>
<tr>
<td><strong>-r root</strong></td>
<td>Specify the root path for a package. (Overrides the default method of finding packages.) Usually, the root path is a directory under installDir that has the same name as the package.</td>
<td><code>C:\&gt; wrenv -p vxworks-6.0 -r directory</code></td>
</tr>
<tr>
<td><strong>-v</strong></td>
<td>Verbose mode. Show all altered environment settings.</td>
<td><code>% wrenv.sh -p vxworks-6.0 -v</code></td>
</tr>
</tbody>
</table>
The install.properties file is a hierarchical registry of package components. It aggregates information from the package.properties files that accompany each installed package. An entry in a properties file has the following form:

```
rootkey.subkey=value[.subkey=value] ...
```

### Table 2-1 Options for wrenv (cont'd)

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>env=value</td>
<td>Set the specified variable in addition to other environment settings. Overrides install.properties.(^b) env=value must be the last item on the command line, except for command [args] (see below).</td>
<td>C:&gt; wrenv -p vxworks-6.0 PATH=directory</td>
</tr>
<tr>
<td>command</td>
<td>Execute the specified command in the new shell environment. (Does not open a new shell.) Must be the last item on the command line.</td>
<td>% wrenv.sh -p vxworks-6.0 ls *.sh</td>
</tr>
</tbody>
</table>

\(^a\) These operations are primarily for internal use and may change in later releases.
\(^b\) Once a setting has been overridden with this option, wrenv maintains the override on subsequent executions—even if the option is omitted from the command line—as long as the new shell is running. A list of overridden settings is stored in a temporary variable called WIND_PROTECTED_ENV that disappears when the shell terminates.
A root key (for example, vxworks60) identifies each package. Subkeys include name, version, and so forth. The entries for a typical package look like this:

```
vxworks60.name=vxworks-6.0
vxworks60.version=6.0
vxworks60.type=platform
vxworks60.subtype=vxworks
vxworks60.label=Wind River VxWorks 6.0

# this entry shows which version of Workbench works with vxworks-6.0
vxworks60.compatible=workbench-2.2
```

# eval entries tell wrenv to make environment settings; see below
vxworks60.eval.01=export WIND_HOME=$(builtin:InstallHome)
vxworks60.eval.02=export WIND_BASE=$(WIND_HOME)$/vxworks-6.0
vxworks60.eval.03=require workbench-2.2
vxworks60.eval.04=addpath PATH $(WIND_BASE)$/host$/$(WIND_HOST_TYPE)$/bin

... An eval subkey specifies an environment processing command, such as defining an environment variable. Each eval subkey has a unique integer subkey appended to it that determines the order in which wrenv executes commands. However, if there are multiple definitions for the same variable, the first (lowest-numbered) definition takes precedence and subsequent definitions are ignored.

A define value for an eval key defines a variable for internal use, but does not export it to the system environment. An export value defines a variable and exports it to the system environment. An addpath value prefixes an element to an existing path variable.

A require value for an eval key specifies the name of another package in which to continue eval processing. An optional value is similar to a require value, except that the command is treated as a no-op if the referenced package does not exist.

Comment lines in a properties file begin with a # symbol.

Properties files are created during installation and should not ordinarily be edited.
3.1 Introduction

This chapter explains how to create and modify projects with the vxprj utility and related Tcl libraries. It contains information about VxWorks components and kernel configuration.

A VxWorks component is a functional unit of code and configuration parameters. A project is a collection of components and build options used to create an application or kernel image. Components are defined with the Component Description Language (CDL) and stored in .cdf files; for more information about components and CDL, see the VxWorks Kernel Programmer’s Guide. Projects are defined in .wpj or .wrproject files; for more information about projects, see the Wind River Workbench User’s Guide.

1. .wrproject files are created and used by the IDE.
Workbench provides a variety of tools for managing projects:

- For most purposes, `vxprj` is the easiest way to manage projects from the command line. However, `vxprj` manages kernel configuration projects only. (A kernel configuration project, also called a VxWorks image project, is a complete VxWorks kernel, possibly with additional application code, that can be downloaded and run on a target.)

- Other Tcl libraries supplied with VxWorks—especially `cmpScriptLib`—provide direct access to scripts called by `vxprj`.

- The IDE’s project management facility handles additional types of project, such as downloadable kernel modules, shared libraries, RTPs, and file system projects, and offers features of its own that are not available from command-line tools.

It is always possible to create and modify projects by directly editing VxWorks component and source code files. However, if you plan to continue using the Workbench project management tools, direct editing of generated project files or makefiles is not recommended.

### 3.2 Using `vxprj`

This guide covers only the most important `vxprj` functionality. For a complete description of `vxprj` and all of its options, see the `vxprj` reference entry. For information about using `vxprj` to invoke the compiler, see 4. Building Kernel and Application Projects.

#### 3.2.1 Creating Kernel Configuration Projects

The command-line `vxprj` utility can create a project based on an existing board support package (BSP)—that is, a collection of files needed to run VxWorks on a specific piece of hardware. To create a kernel configuration project, type the following:

```
vxprj create [-source] [-profile profile] BSP tool [projectFile | projectDirectory]
```

2. For information about BSPs, see the VxWorks BSP Developer’s Guide.
The command sequence above uses the following arguments:

- **-source**
  
  An optional argument to enable source build. A project can be built from source if both of the following are true:
  
  - The -source option is used.
  - All components in your project support the -source option.

  If either of these conditions is not true, the project is created from pre-built libraries (the default). After the project has been created, you can change the build mode.

  Note that only certain kernel profiles support the -source option. If you modify a pre-defined kernel profile by including additional components, building from source may not be possible.

  For complete details on this option, see the *vxprj* reference entry.

- **-profile** *profile*
  
  An optional argument to specify the kernel profile type to be built. If you do not specify a profile, the project is built according to the default profile that is defined for your BSP. For information on profiles, see *Using Profiles*, p.14, and the *VxWorks Kernel Programmer’s Guide: Kernel*.

- **BSP**
  
  The name or location of a BSP.

- **tool**
  
  A recognized compiler (usually diab or gnu).

- **projectFile|projectDirectory**
  
  An optional argument to specify the name of the project file to create, or the name of its parent directory. If no filename or directory is specified, *vxprj* generates a name based on the other input arguments and stores the resulting project in a new directory under installDir/target/proj.

If you do not have a target board for your application, the easiest way to experiment with *vxprj* is to create projects for the VxWorks Simulator. A standard Workbench installation includes a “BSP” for the simulator. For example:

```
C:\> vxprj create simpc diab C:\myProj\myproj.wpj
```
This command creates a kernel configuration project to be built with the Wind River compiler (diab) and run on the VxWorks Simulator for Windows (simp). The project file (myproj.wpj), source files, and makefile are stored in C:\myProj.

To generate a similar project for Solaris or Linux hosts, specify solaris or linux as the BSP, type the following:

```bash
% vxprj create solaris diab /myProj/myproj.wpj
$ vxprj create linux diab /myProj/myproj.wpj
```

(For more information about the VxWorks Simulator, see 5.3 Using the VxWorks Simulator, p.38, and the Wind River VxWorks Simulator User’s Guide.)

The following command creates a project in the default location using the wrSbcPowerQuiccII (PowerPC) board support package:

```bash
% vxprj create wrSbcPowerQuiccII diab
```

When this command is executed, vxprj creates a subdirectory for the new project and reports the location of directory.

The following command creates a GCC project in C:\myProjects\mips64\ using the rh5500_mips64 BSP:

```bash
C:\> vxprj create rh5500_mips64 gnu C:\myProjects\mips64\mipsProject.wpj
```

The project file created by this command is called mipsProject.wpj.

**Copying Projects**

To copy an existing project to a new location, type the following:

```bash
vxprj copy [sourceFile] destinationFile | destinationDirectory
```

If no source file is specified, vxprj looks for a .wpj file in the current directory. If a destination directory—but no destination filename—is specified, vxprj creates a project file with the same name as the directory it resides in. For example:

```bash
% vxprj copy myproj.wpj /New
```

This command copies myproj.wpj and associated files to /New/New.wpj.

**Using Profiles**

A profile is a preconfigured set of components that provides a starting point for custom kernel configuration. For example, PROFILE_DEVELOPMENT includes a variety of standard development and debugging tools. The following command
creates a PROFILE DEVELOPMENT-based project in the default location using the wrSbcPowerQuiccII BSP:

```
% vxprj create -profile PROFILE DEVELOPMENT wrSbcPowerQuiccII diab
```

For a list of profile names and information on components included in each profile, see the VxWorks Kernel Programmer’s Guide: Kernel.

### 3.2.2 Deleting Projects

To delete a project, type the following:

```
vxprj delete projectFile
```

where `projectFile` is the `.wpj` file associated with the project. The `delete` command permanently deletes the directory in which `projectFile` resides and all of its contents and subdirectories. (Do not run the command from the project directory you are trying to remove.) For example:

```
% vxprj delete /myProj/myproj.wpj
```

This command deletes the entire `myProj` directory.

⚠️ **CAUTION:** `vxprj delete` removes any file passed to it, regardless of the file’s name or extension, along with the entire directory in which the file resides. It does not verify that the specified file is a Workbench project file, nor does it attempt to save user-generated files.

### 3.2.3 Modifying Projects

#### Adding Components

To add components to a kernel configuration project, type the following:

```
vxprj component add [projectFile] component [component ...]
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory and adds the specified components to that file. Components are identified by the names used in `.wpj` and `.cdf` files, which have the form `INCLUDE_xxx`. For example:

```
% vxprj component add MyProject.wpj INCLUDE_MMU_BASIC INCLUDE_ROMFS
```

This command adds support for a memory management unit and the ROMFS target file system to `MyProject.wpj`. 
Adding Bundles

Some components are grouped into *bundles* that provide related or complementary functionality. Adding components in bundles is convenient and avoids unresolved dependencies.

To add a bundle to a project, type the following:

```
vxprj bundle add [projectFile] bundle [bundle ...]
```

For example:

```
% vxprj bundle add BUNDLE_RTP_DEVELOP
```

This command adds process (RTP) support to the kernel configuration project in the current working directory.

```
% vxprj bundle add MyProject.wpj BUNDLE_RTP_DEVELOP
  BUNDLE_STANDALONE_SHELL BUNDLE_POSIX BUNDLE_EDR
```

This command adds support for processes, the kernel shell, POSIX, and error detection and reporting to *MyProject.wpj*.

Removing Components

To remove components from a kernel configuration project, type the following:

```
vxprj component remove [projectFile] component [component ...]
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. For example:

```
% vxprj component remove MyProject.wpj INCLUDE_MMU_BASIC INCLUDE_DEBUG
```

This command removes the specified components as well as any components that are dependent on them.

Removing Bundles

To remove a bundle, type the following:

```
vxprj bundle remove [projectFile] bundle [bundle ...]
```

Removing a bundle removes all the components it contains (regardless of how they were added), so it is best to check the contents of a bundle before using `remove`. See *Examining Bundles*, p.19.
3 Working with Projects and Components

3.2 Using vxprj

Setting Configuration Parameter Values

To set the value of a configuration parameter, type the following:

```
vxprj parameter set [projectFile] parameter value
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. For example:

```
% vxprj parameter set MyProject.wpj VM_PAGE_SIZE 0x10000
```

This command sets `VM_PAGE_SIZE` to 0x10000. (To list a project’s configuration parameters, see Listing Configuration Parameters and Values, p.20.)

Parameter values that contain spaces should be enclosed in quotation marks. If a parameter value itself contains quotation marks, they can be escaped with `\` (Windows) or the entire value surrounded with `’...’` (UNIX). An easier way to set parameter values that contain quotation marks is to use `setstring`, which tells `vxprj` to treat everything from the space after the `parameter` argument to the end of the command line as a single string. For example:

```
% vxprj parameter setstring SHELL_DEFAULT_CONFIG "LINE_LENGTH=128"
```

This command sets `SHELL_DEFAULT_CONFIG` to “LINE_LENGTH=128”, including the quotation marks.

To reset a parameter to its default value, type the following:

```
vxprj parameter reset [projectFile] parameter [parameter ... ]
```

Changing the Project Makefile Name

To change the name of a project’s makefile, type the following:

```
vxprj makefile [projectFile] newMakefileName
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. For example:

```
% vxprj makefile make.rules
```

This command changes the name of the makefile (for the project in the current working directory) from the default `Makefile` to `make.rules`. 
Adding and Removing Individual Files

To add a specific source code file to a kernel configuration project, type the following:

\[\text{vxprj file add } \text{[projectFile]} \text{ sourceFile}\]

If no project file is specified, \text{vxprj} looks for a .wpj file in the current directory. When the project is built, the specified source file is compiled and linked into the resulting kernel image.

To remove a file from a project, type the following:

\[\text{vxprj file remove } \text{[projectFile]} \text{ sourceFile}\]

3.2.4 Generating Project and Component Diagnostics

Obtaining a List of Components

To see a list of components, type the following:

\[\text{vxprj component list } \text{[projectFile]} \text{ [type]} \text{ [pattern]}\]

If no project file is specified, \text{vxprj} looks for a .wpj file in the current directory. If \text{pattern} is specified, \text{vxprj} lists only components whose names contain \text{pattern} as a substring; if \text{pattern} is omitted, all components are listed.

The \text{type} argument can be \text{all}, \text{included}, \text{excluded}, or \text{unavailable}. The default is \text{included}, which lists components included in the project. Specify \text{excluded} to list installed components that are \text{not} included in the project; \text{all} to list all installed components; or \text{unavailable} to list components that are installed but not available for the project. (An \text{available} component is one that is installed, with all its dependent components, under the VxWorks directory.)

For example:

\% \text{vxprj component list MyProject.wpj SHELL}\n
This command returns all components in \text{MyProject.wpj} whose names contain “SHELL”, such as \text{INCLUDE\_SHELL\_BANNER} and \text{INCLUDE\_RTP\_SHELL\_C}.

\% \text{vxprj component list MyProject.wpj excluded VM}\n
This command returns all available components with names containing “VM” that are \text{not} included in \text{MyProject.wpj}. 

18
Examining Bundles

To see a list of bundles, type the following:

```
vxprj bundle list [projectFile] [type] [pattern]
```

For `type` and `pattern`, see *Obtaining a List of Components*, p.18.

To see the components and other properties of a bundle, type the following:

```
vxprj bundle get [projectFile] bundle
```

Examining Profiles

To see a list of profiles, type the following:

```
vxprj profile list [projectFile] [pattern]
```

For `pattern`, see *Obtaining a List of Components*, p.18.

To see the components and other properties of a profile, type the following:

```
vxprj profile get [projectFile] profile
```

Comparing the Components in Different Projects

To compare the components in two projects, type the following:

```
vxprj component diff [projectFile] projectFile \ directory
```

If only one project file or directory is specified, `vxprj` looks for a `.wpj` file in the current directory and compares it to the specified project. For example:

```
% vxprj component diff /Apps/SomeProject.wpj
```

This command compares the components included in `/Apps/SomeProject.wpj` to those included in the project in the current working directory. It returns a list of the unique components in each project.

Checking a Component

To verify that components are correctly defined, type the following:

```
vxprj component check [projectFile] [component ...]
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. If no component is specified, `vxprj` checks every component in the project. For example:

```
% vxprj component check MyProject.wpj
```
This command invokes the `cmpTest` routine, which tests for syntactical and semantic errors.

**Checking Component Dependencies**

To generate a list of component dependencies, type the following:

```
vxprj component dependencies [projectFile] component [component ...]
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. For example:

```
% vxprj component dependencies INCLUDE_OBJ_LIB
```

This command displays a list of components required by `INCLUDE_OBJ_LIB`.

**Listing Configuration Parameters and Values**

To list a project’s configuration parameters, type the following:

```
vxprj parameter list [projectFile] [pattern]
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. If `pattern` is specified, `vxprj` lists only parameters whose names contain `pattern` as a substring; if `pattern` is omitted, all parameters are listed. For example:

```
% vxprj parameter list MyProject.wpj TCP
```

This command lists all parameters defined in `MyProject.wpj` whose names contain “TCP”, such as `TCP_MSL_CFG`.

To list a project’s parameters and their values, type the following:

```
vxprj parameter value [projectFile] [NamePattern] [valuePattern]
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. If `namePattern` is specified, `vxprj` lists only parameters whose names contain `namePattern` as a substring; if `valuePattern` is specified, `vxprj` lists only parameters whose values contain `valuePattern` as a substring. For example:

```
% vxprj parameter value
% vxprj parameter value USER TRUE
```

The first command lists all parameters and values for the project in the current directory. The second lists only parameters whose names contain “USER” and whose values contain “TRUE”.

20
Comparing Parameters in Different Projects

To compare the configuration parameters of two projects, type the following:

```
vxprj parameter diff [projectFile] projectFile \ directory
```

If only one project file or directory is specified, **vxprj** looks for a .wpj file in the current directory and compares it to the specified project. For example:

```
% vxprj parameter diff /MyProject/MyProject.wpj /Apps/SomeProject.wpj
```

This command compares the parameters in **MyProject.wpj** to those in **SomeProject.wpj** and returns a list of unique parameter-value pairs for each project.

Examining the Source Files in a Project

To list a project’s source code files, type the following:

```
vxprj file list [projectFile] [pattern]
```

If no project file is specified, **vxprj** looks for a .wpj file in the current directory. If **pattern** is specified, **vxprj** lists only files whose names contain **pattern** as a substring; otherwise, all files are listed.

To see build information for a source code file, type the following:

```
vxprj file get [projectFile] sourceFile
```

### 3.3 Using cmpScriptLib and Other Libraries

The VxWorks tools include several Tcl libraries, such as cmpScriptLib, that provide routines for project and component management. Additional information about these routines is available in the reference pages for each library and in the VxWorks Kernel Programmer’s Guide.

To access cmpScriptLib, you need a correctly configured Tcl shell. Start by entering the following:

```
C:\> tclsh
# package require OsConfig
```
From the new command prompt you can use routines in `cmpScriptLib`, including `cmpProjCreate`, which creates kernel configuration projects. For example:

**UNIX**

```
# cmpProjCreate pcPentium4 /MyProject/project1.wpj
```

**Windows**

```
# cmpProjCreate pcPentium4 C:\MyProject\project1.wpj
```

This command creates a kernel configuration project using the `pcPentium4` BSP. Notice the double backslashes (`\`) in the Windows directory path.

To manipulate an existing project, first identify the project by “opening” it with `cmpProjOpen`. (A project that has just been created with `cmpProjCreate` or `cmpProjCopy` is already open. Only one project at a time can be open within a Tcl shell.) When you are finished, you can close the project with `cmpProjClose`:

```
# cmpProjOpen  ProjectFile
# ... additional commands ...
# cmpProjClose
```

Another useful routine in `cmpScriptLib` is `autoscale`, which analyzes projects and generates a list of unused components. This can help to produce the most compact executable. For example:

```
# cmpProjOpen myProj.wpj
# autoscale
# ... output ...
# cmpProjClose
```

These commands generate a list of components that can be removed from `myProj.wpj` (because their code is never called) as well as a list of components that should be added to the project (because of dependencies). The `autoscale` facility can also be invoked through `vxprj`; see the `vxprj` reference entry for details.

The `vxprj` utility gives you access to most `cmpScriptLib` functionality without having to start a Tcl shell or write Tcl scripts. For example, consider the following `vxprj` command:

```
% vxprj component remove myProj.wpj INCLUDE_WDB
```

This command is equivalent to the following:

```
% tclsh
# package require OsConfig
# cmpProjOpen myProj.wpj
# cmpRemove INCLUDE_WDB
# cmpProjClose
```
3.4 RTP and Library Projects

User-mode (RTP) application and library projects can be created by following the models in `target/usr/apps/samples/` and `target/usr/src/usr/` under the VxWorks installation directory. For more information, see 4.6 RTP Applications and Libraries, p.30.
4

Building Kernel and Application Projects

4.1 Introduction 25
4.2 Building Kernel Configuration Projects with vxprj 26
4.3 Building Projects with cmpScriptLib 29
4.4 Other VxWorks Project Types 29
4.5 Calling make Directly 30
4.6 RTP Applications and Libraries 30

4.1 Introduction

This chapter explains how to build projects using the vxprj facility, cmpScriptLib routines, and make, and how to generate makefiles for building VxWorks kernel libraries.

The tools and methods available for build management depend on the type of project under development. While the IDE supports most project types, the command-line facilities are less uniform:

- **Kernel configuration (VxWorks image) projects** can be built with vxprj or cmpScriptLib, or by calling make directly. The preferred method is to use vxprj. The output of a build is a group of object (.o) files and a VxWorks kernel image (vxWorks).
RTP (user-mode) applications and libraries can be built by calling make directly. A standard series of make rules is available for this purpose. The output of an RTP application build is a group of object (.o) files and an executable (.vxe) file that runs under VxWorks. The output of an RTP library build is a group of object (.o) files and an archive (.a) file that can be linked into RTP applications.

Custom Boot Loaders can be built by running make bootrom in the BSP directory. For details, see the VxWorks Kernel Programmer's Guide: Kernel.

Shared libraries, downloadable kernel modules, and file system projects are most easily handled from the IDE, but they can be built by calling make directly.

Regardless of how build management is approached, Workbench supports two toolkits—GCC and the Wind River Compiler—both of which use the GNU make utility. When you create a kernel configuration project with vxprj or the IDE, you must select a toolkit. When you build other projects from the command line, you can (assuming that your application code is portable) select a toolkit at the time of compilation.

For information on building VxWorks kernel libraries, see the getting started guide for your platform product.

### 4.2 Building Kernel Configuration Projects with vxprj

A kernel configuration (VxWorks image) project includes build rules based on the format used in makefiles. Projects also include build specifications, which organize and configure build rules. Build specifications depend on the type of project and BSP, but a typical project might have four build specifications: default, default_rom, default_romCompress, and default_romResident; for information about these build specifications, see the VxWorks User's Guide. A build specification defines variables passed to make and flags passed to the compiler. Each project has a current build specification, initially defined as default.

To build a kernel configuration project with vxprj, type the following:

```
vxprj build [projectFile] [buildSpecification | buildRule]
```

If no project file is specified, vxprj looks for a .wpj file in the current directory. If the second argument is omitted, vxprj uses the project’s current build
specification. Output from the compiler is saved in a subdirectory—with the same name as the build specification—under the project’s source directory. For example:

```
% vxprj build
% vxprj build myproj.wpj default_rom
```

The first command builds the project found in the current directory using the project’s current build specification. The second command builds the project defined in `myproj.wpj` using the `default_rom` build specification.

### 4.2.1 Examining Build Specifications and Rules

To see the name of the current build specification, type the following:

```
vxprj build get [projectFile]
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. To see all available build specifications for a project, type the following:

```
vxprj build list [projectFile]
```

To see all the build rules in a project’s current build specification, type the following:

```
vxprj buildrule list [projectFile]
```

To examine a build rule in a project’s current build specification, type the following:

```
vxprj buildrule get [projectFile] buildRule
```

For example:

```
% vxprj buildrule get prjConfig.o
```

This command displays the `prjConfig.o` build rule.

### 4.2.2 Changing Build Settings

To change a project’s current build specification, type the following:

```
vxprj build set [projectFile] buildSpecification
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. For example:

```
% vxprj build set myproj.wpj default_romCompress
```
This command changes the current build specification of `myproj.wpj` to `default_romCompress`.

To reset a project’s current build specification to its default, type the following:

```
vxprj build reset [projectFile]
```

The `set` and `reset` commands update a project’s makefile as well as its `.wpj` file.

### Adding and Changing Build Rules

The commands documented below edit project makefiles and `.wpj` files.

To add a build rule to a project’s current build specification, type the following:

```
vxprj buildrule add [projectFile] buildRule value
```

If no project file is specified, `vxprj` looks for a `.wpj` file in the current directory. For example:

```
% vxprj buildrule add default_new "$(CC) $(CFLAGS) ./prjConfig.c -o $@
```

This command creates a build rule (if it doesn’t already exist) called `default_new`, adds it to the current build specification, and sets its value to `$(CC) $(CFLAGS) ./prjConfig.c -o $@`.

To create or edit a build rule without including it in the project’s current build specification, type the following:

```
vxprj buildrule set [projectFile] buildRule value
```

Rules created with `set` are added to the list of available build rules for the current build specification.

To remove a build rule from a project, type the following:

```
vxprj buildrule remove [projectFile] buildRule
```

To set the default build rule for the current build specification, type the following:

```
vxprj buildrule [projectFile] buildRule
```

For example:

```
% vxprj buildrule default_new
```
4.3 Building Projects with cmpScriptLib

The cmpScriptLib Tcl library includes routines that build and manipulate kernel configuration projects. To access cmpScriptLib, you need a correctly configured Tcl shell. Start by entering the following:

```bash
% tclsh
# package require OsConfig
```

From the new command prompt, you can build a project by typing the following:

```bash
cmpProjOpen  projectFile
cmpBuild
```

For example:

```bash
# cmpProjOpen /Apps/SomeProject.wpj
# cmpBuild
```

This command builds the project defined in SomeProject.wpj. If you specify a Windows directory path with `projectFile`, be sure to use double backslashes (`\`).

cmpScriptLib contains several routines that allow you to view and set build rules and build specifications, including `cmpBuildRule`, `cmpBuildRuleSet`, `cmpBuildSpecSet`, `cmpBuildRuleListGet`, `cmpBuildRuleAdd`, `cmpBuildRuleRemove`, and `cmpBuildRuleDefault`. Information about these routines is available in the reference entry for cmpScriptLib. For additional information about cmpScriptLib, see also 3.3 Using cmpScriptLib and Other Libraries, p. 21.

4.4 Other VxWorks Project Types

Real-time process (RTP), shared library, downloadable kernel module, and file system projects are most easily created and built from the IDE. For information about managing these projects with the IDE, see the VxWorks User’s Guide. For detailed information about RTPs, shared libraries, and file systems, see the VxWorks Application Programmer’s Guide. For information about downloadable kernel modules—essentially object (.o) files that can be dynamically linked to a VxWorks kernel—see the VxWorks Kernel Programmer’s Guide.
4.5 Calling make Directly

The recommended way of building kernel configuration projects is to use vxprj, cmpScriptLib, or the IDE. VxWorks projects of other types can be built from the IDE. If you decide to build a project by invoking the make utility directly, and especially if you edit the makefile, you should discard any generated project (.wpj or .wrproject) files and no longer use the Workbench project management tools.

The wrenv environment utility automatically configures the correct version of make, and Workbench-generated makefiles contain information about build tools, target CPUs, and compiler options. Hence you should be able to build a project created by vxprj, cmpScriptLib, or the IDE simply by moving to the project’s parent directory and entering make from the command prompt. If the project is set up to support multiple compilers or target CPUs, you may need to specify values for make variables on the command line; for example:

```bash
% make TOOL=diab
```

On Windows, the make utility, as configured by wrenv, executes the Z shell (zsh, installed with the Workbench tools as sh.exe). On UNIX, the make utility executes whatever shell program is invoked by the command sh.

**NOTE:** Your compiler installation may include a copy of dmake, an alternative open-source make utility. This make utility is used only for building libraries shipped with the standalone Wind River Compiler toolkit. VxWorks projects, whether compiled with GCC or the Wind River Compiler, should be managed with make.

For complete information about make, see the GNU Make manual.

4.6 RTP Applications and Libraries

To build a user-mode (RTP) application or library from the command line, you can invoke the make utility directly. Wind River supplies a general build model implemented by a series of make rules, with standard make variables that control aspects of the build process such as target CPU and toolkit.
For example, the following command could be used to build an application like the 
helloworld RTP sample included with Workbench:

```
% make CPU=CPU_Name TOOL=diab
```

This command builds helloworld for PowerPC targets, using the Wind River 
(Diab) compiler. For more information about CPU and TOOL, see the VxWorks 
Architecture Supplement.

### RTP Applications

The makefile for the helloworld RTP application looks like this:

```make
# This file contains make rules for building the hello world RTP 
EXE = helloworld.vxe
OBJS = helloworld.o
include $(WIND_USR)/make/rules.rtp
```

This makefile is simple because most of the make rules are included by indirection 
from rules.rtp and other files referenced in rules.rtp. When make processes this 
file, helloworld.c is compiled to helloworld.o, which is then linked with other 
VxWorks user-mode (RTP) libraries to produce the application executable 
helloworld.vxe.

### RTP Libraries

The makefile for a static archive looks like this:

```make
# This file contains make rules for building the foobar library 
LIB_BASE_NAME = foobar
OBJS = 
        foo1.o foo2.o \ 
        bar1.o bar2.o 
include $(WIND_USR)/make/rules.library
```

This makefile could be used to build a library called libfoobar.a. It includes rules 
defined in rules.library.

### Makefile and Directory Structure

Typically, each application or library project is stored in a separate directory with 
its own makefile. For examples, see target/usr/apps/samples/ under the VxWorks 
installation directory.

rules.rtp, rules.library, and other make rule files (some of them host-specific) are 
found in target/usr/make/ under the VxWorks installation directory. Except for the
LOCAL_CLEAN and LOCAL_RCLEAN variables (see 4.6.2 Make Variables, p.32), rules.rtp or rules.library should usually be included as the last line of the makefile; user-defined build rules must precede rules.rtp or rules.library.

4.6.1 Rebuilding VxWorks RTP (User-Mode) Libraries

The VxWorks user-mode source base, which provides supporting routines for applications, is installed under target/usr/src/. These files can be used to build both statically linked (.a) and dynamically linked (.so) libraries by setting the value of LIB_FORMAT (see 4.6.2 Make Variables, p.32). From the target/usr/src/ directory, you can rebuild the user-mode libraries by typing the following:

```
% make CPU=target TOOL=toolkit
```

Unless overridden with SUBDIRS or EXCLUDE_SUBDIRS (see 4.6.2 Make Variables, p.32), the build system descends recursively through the directory tree, executing make in each subdirectory.

When RTP libraries are compiled, the preprocessor macro __RTP__ is defined and the preprocessor macro _WRS_KERNEL is not defined.

4.6.2 Make Variables

The VxWorks build system utilizes a number of make variables (also called macros), some of which are described below. The files in target/usr/make/ include additional variables, but only the ones documented here are intended for redefinition by the user.

**ADDED_C++FLAGS**
- Additional C++ compiler options.

**ADDED_CFLAGS**
- Additional C compiler options.

**ADDED_CLEAN_LIBS**
- A list of libraries (static and dynamic) deleted when make clean is executed in the application directory. See LOCAL_CLEAN and LOCAL_RCLEAN below.

**ADDED_DYN_EXE_FLAGS**
- Additional compiler flags specific to the generation of dynamic executables.

**ADDED_LIBS**
- A list of static libraries (in addition to the standard VxWorks RTP libraries) linked into the application.
ADDED_LIB_DEPS
Dependencies between the application and the application’s static libraries (with the + operator). For static linkage, this variable forces a rebuild of the application if the static library has been changed since the last build.

ADDED_SHARED_LIBS
A list of shared libraries dynamically linked to the application. Items in the list are prefixed with lib and have the .so file extension added. For example, ADDED_SHARED_LIBS="foo bar" causes the build system to try to link libfoo.so and libbar.so.

CPU
The target instruction-set architecture to compile for. This is not necessarily the exact microprocessor model.

DOC_FILES
A list of C and C++ source files that are specially parsed during the build to generate online API documentation. Should be an empty list if there are no files to generate documentation from.

EXCLUDE_SUBDIRS
A list of subdirectories excluded from the build. Generally, when make is executed, the system tries to build the default target for every subdirectory of the current directory that has a makefile in it. Use EXCLUDE_SUBDIRS to override this behavior. See also SUBDIRS.

EXE
The output executable filename. Specify only one executable per makefile. Do not include a directory path. See VXE_DIR.

EXE_FORMAT
The format of the output executable (static or dynamic). The default is static.

EXTRA_INCLUDE
Additional search paths for the include files. Uses the += operator.

LIBNAME
A non-default directory for the static library.

LIB_BASE_NAME
The name of the archive that objects built in the directory are collected into. For example, LIB_BASE_NAME=foo causes the build system to create a static library called libfoo.a or a dynamic library called libfoo.so. See LIB_FORMAT. (Library builds only.)
LIB_DIR
A local make variable that can be used conveniently to identify where a library
is located (if it is not in the default location) in the ADDED_LIBS and
ADDED_LIB_DEPS lines without repeating the literal path information.

LIB_FORMAT
The type of library to build. Can be static, shared (dynamic), or both; defaults
to both. (Library builds only.)

LOCAL_CLEAN
Additional files deleted when make clean is executed. By default, the clean
target deletes files in OBJS. Use LOCAL_CLEAN to specify additional
files to be deleted. Must be defined after rules.rtp or rules.library.

LOCAL_RCLEAN
Additional files deleted when make rclean is executed. The rclean target
recursively descends the directory tree starting from the current directory,
executing make clean in every subdirectory; if SUBDIRS is defined, only
directories listed in SUBDIRS are affected; directories listed in
EXCLUDE_SUBDIRS are not affected. Use LOCAL_RCLEAN to specify
additional files in the current directory to be deleted. Must be defined after
rules.rtp or rules.library.

OBJ_DIR
The output subdirectory for object files.

OBJS
A list of object files built; should include object files to be linked into the final
executable. Each item must end with the .o extension. If you specify an object
file that does not have a corresponding source file (.c for C, .cpp for C++, or .s
for assembly), there must be a build rule that determines how the object file is
generated. Do not include a directory path. See OBJ_DIR and LIBDIRBASE.

SL_INSTALL_DIR
A non-default location for the library file. It is often useful to keep project work
outside of the installation directory.

SL_VERSION
A version number for a shared library. By default, the shared library version
number is one (libName.so.1), so this variable is not needed unless you want to
build an alternate version of the shared library.

SUBDIRS
A list of subdirectories of the current directory in which the build system looks
for a makefile and tries to build the default target. If SUBDIRS is not defined,
the system tries to build the default target for every subdirectory of the current
directory that has a makefile in it. EXCLUDE_SUBDIRS overrides SUBDIRS.

**TOOL**
The compiler and toolkit used. The Wind River (Diab) and GCC (GNU)
compilers are supported. TOOL can also specify compilation options for
endianness or floating-point support.

**VXE_DIR**
The output subdirectory for executable files and shared libraries. Defaults to
target/usr/root/CPUtool/bin/ (executables) or target/usr/root/CPUtool/lib/
(shared libraries); for example, target/usr/root/PPC32diab/bin/.

For further information on these make variables, see the VxWorks Application

For information on the supported values for the CPU and TOOL make variables,
see the VxWorks Architecture Supplement: Building Applications.
5

Connecting to a Target

5.1 Introduction

This chapter outlines procedures for running compiled VxWorks applications on targets and simulators.

5.2 Connecting to a Target Board

Downloading VxWorks to a physical target involves the following steps:

1. Launch the Wind River registry:
   - On Windows, type `wtxregd`.
   - On UNIX, type `wtxregd start`.

5.3 Using the VxWorks Simulator
The registry maintains a database of target servers, boards, ports, and other items used by the development tools to communicate with targets. For more information, see the *wtxregd* and *wtxreg* reference entries.

2. Connect the target to a serial terminal.

3. Switch on the target.

4. Edit the boot loader parameters to tell the boot loader the IP address of the target and the location of the VxWorks image. See the *VxWorks Kernel Programmer’s Guide: Kernel* for details.

5. Start the target server by typing the following:

   ```
   % tgsrv targetIPaddress -n target -c pathToVxWorksImage
   ```

   The target server allows development tools, such as the host shell or debugger, to communicate with a remote target. For more information, see the *tgsrv* and *wtxConsole* reference entries.

6. Start the host shell by typing the following:

   ```
   % windsh targetServer
   ```

   The host shell allows command-line interaction with a VxWorks target. For an overview of the host shell, see *Debugging Applications with the Host Shell*, p.41.

7. From the host shell, you can load and run applications. For example:

   ```
   % ld < filename.out
   % rtpSp "filename.vxe"
   ```

   For detailed information, see *Host Shell Commands and Options*, p.47.

### 5.3 Using the VxWorks Simulator

To run under the VxWorks Simulator, an application must be specially compiled. For VxWorks images, the best way to do this is to set up and build a simulator-enabled version of the project. This can be done with *vxprj*:

```
vxprj create simpc|solaris|linux diab|gnu [projectFile|directory]
vxprj build projectFile
```

For more information about these commands, see 3.2.1 Creating Kernel Configuration Projects, p.12 and 4.2 Building Kernel Configuration Projects with *vxprj*, p.26.
RTP applications that use the standard Wind River build model can be compiled for the simulator by specifying an appropriate value for the `CPU make` variable:

```
cd projectDirectory
make CPU=SIMPENTIUM SIMSPARCSOLARIS TOOL=diab|gnu
```

For more information, see Section 4.6 RTP Applications and Libraries, p.30.

To run the simulator, type the following:

```
vxsim [-f pathToVxWorksImage | filename.vxe] [otherOptions]
```

For more information about the simulator, see the Wind River VxWorks Simulator User’s Guide.
6

Debugging Applications with the Host Shell

6.1 Introduction 41
6.2 Starting and Stopping the Host Shell 42
6.3 Switching Modes 42
6.4 Command Mode 42
6.5 C Interpreter 44
6.6 Tcl Mode 45
6.7 GDB Mode 45

6.1 Introduction

This chapter explains how to run and debug programs from the host shell. For a list of host shell commands and options, see Host Shell Commands and Options, p.47 and the host shell reference pages: hostShell, cMode, cmdMode, gdbMode, and rtpCmdMode.

You can use the host shell in four interactive modes: C interpreter, which executes C-language expressions; Command (Cmd), a UNIX-style command interpreter; Tcl, to access the WTX Tcl API and for scripting; and GDB, for debugging with GNU Debugger commands.

In any mode, entering help displays a summary of available commands.
6.2 Starting and Stopping the Host Shell

To start the host shell, type the following:

```
windsh [options] targetServer
```

For example, to connect to a running simulator, type the following:

```
C:\> windsh vxsim0@hostname
```

You can run multiple host shells attached to the same target.

To terminate a host shell session—from any mode—type `exit` or `quit`, or press `Ctrl+D`.

6.3 Switching Modes

By default, the host shell starts in C interpreter mode. To switch to another mode, type the following:

- `cmd` for command mode. The prompt changes to `[vxWorks] #.`
- `?` for Tcl mode. The prompt changes to `tcl>`.  
- `gdb` for GDB mode. The prompt changes to `gdb>`.  
- `C` to return to the C interpreter. The prompt changes to `->`.

These commands can also be used to evaluate a statement native to another interpreter. For example, to execute a C statement from Command mode, you could type the following:

```
[vxWorks]# C test = malloc(100)
```

6.4 Command Mode

The command interpreter uses a mixture of GDB and UNIX syntax. To use the command interpreter, type the following:

```
command [subcommand [subcommand ...]] [options] [arguments] [;]
```
6 Debugging Applications with the Host Shell
6.4 Command Mode

Spaces within an argument must be preceded by a backslash (\) unless the entire argument is surrounded by single or double quotation marks.

For example:

List the contents of a directory.
[vxWorks]# ls -l /folk/usr

Create an alias.
[vxWorks]# alias ls "ls -l"

Summarize task TCBs.
[vxWorks]# task

Suspend a task, then resume it.
[vxWorks]# task suspend t1
[vxWorks]# task resume t1

Set a breakpoint for a task at a specified address.
[vxWorks]# bp -t t1 0x12345678

Set a breakpoint on a function.
[vxWorks]# bp &printf

Show the address of someInt.
[vxWorks]# echo &someInt

Step over a task from a breakpoint.
[vxWorks]# task stepover t1

Continue a task.
[vxWorks]# task continue t1

Delete a task.
[vxWorks]# task delete t1

Run an RTP application.
[vxWorks]# /folk/user/TMP/helloworld.vxe

Run an RTP application, passing parameters to the executable.
[vxWorks]# cal.vxe -j 2002

Run an RTP application, passing options to the executable and to the RTP loader (in this case, setting the stack size to 8K).
[vxWorks]# rtp exec -u 8096 /folk/user/TMP/foo.vxe -q

List RTPs or show brief information about a specific RTP.
[vxWorks]# rtp [rtpID]

Show details about an RTP.
[vxWorks]# rtp info [rtpID]

Stop an RTP, then continue it.
[vxWorks]# rtp stop 0x43210
[vxWorks]# rtp continue 0x43210
6.5 C Interpreter

The C interpreter, in addition to executing most C-language expressions, provides a variety of native commands for managing tasks and programs. It does not provide access to processes; use command mode to debug processes (RTPs).

Sometimes a routine in your application code will have the same name as a host shell command. If such a conflict arises, you can direct the C interpreter to execute the target routine, rather than the host shell command, by prefixing the routine name with @, as shown in the final example below.

For example:

Execute C statements.
- » test = malloc(100); test[0] = 10; test[1] = test[0] + 2
- » printf("Hello!"")

Download and dynamically link a new module.
- » ld < /usr/apps/someProject/file1.o

Create new symbols.
- » MyInt = 100; MyName = "Bob"

Show system information (task summary).
- » i

Show information about a specific task.
- » ti(s1u0)

Suspend a task, then resume it.
- » ts(s1u0)
- » tr(s1u0)

Show stack trace.
- » tt

Show current working directory; list contents of directory.
- » pwd
- » ls

Set a breakpoint.
- » b(0x12345678)

Step program to the next routine.
- » s

Call a VxWorks function; create a new symbol (my_fd).
- » my_fd = open("file", 0, 0)
6 Debugging Applications with the Host Shell

6.6 Tcl Mode

Call a function from your application.
  -> someFunction (1,2,3)

Call an application function that has the same name as a shell command.
  -> @i()

6.7 GDB Mode

In GDB mode, the host shell interprets GNU Debugger commands.

For example:

- Set a breakpoint on foo.
  gdb> break foo

- Set a breakpoint on line 100 of main.c.
  gdb> break main.c:100

- Show breakpoints.
  gdb> info breakpoints

- Delete a breakpoint.
  gdb> delete 4

- Run a program.
  gdb> exec-file main.out
  gdb> run

- Step through a program.
  gdb> step

- Continue.
  gdb> c
Display back trace of ten frames.
gdb> bt 10

Display variables.
gdb> p var

Disconnect from the target.
gdb> disconnect

Quit GDB.
gdb> quit
A.1 Introduction

This appendix provides details about operating the host shell. For more information, see the host shell reference pages: hostShell, cMode, cmdMode, gdbMode, and rtpCmdMode.

The host shell is a host-resident command shell that allows you to download application modules, invoke operating-system and application subroutines, and monitor and debug applications. A target-resident version of the shell (called the kernel shell) is also available; see the VxWorks Kernel Programmer's Guide: Target Tools.
Host shell operation involves a target server, which handles communication with the remote target, dispatching function calls and returning their results; and a target agent, a small monitor program that mediates access to target memory and other facilities. The target agent is the only component that runs on the target. The symbol table, managed by the target server, resides on the host, although the addresses it contains refer to the target system.

### A.2 Host Shell Basics

You can use the host shell in one of four modes:

- **C interpreter**, which executes C-language expressions and allows prototyping and debugging in kernel space. The host shell starts in this mode by default. (See A.4 Using the C Interpreter, p.66.)

- **Command (Cmd)**, a UNIX-style command interpreter for debugging and monitoring a system, including RTPs. (See A.3 Using the Command Interpreter, p.54.)

- **Tcl**, to access the WTX TCL API and for scripting. (See A.5 Using the Tcl Interpreter, p.72.)

- **GDB**, for debugging a target using GDB commands. (See A.6 Using the GDB Interpreter, p.73.)

#### A.2.1 Starting the Host Shell

To start the host shell, type the following:

```
windsh [options] targetServer
```

Table A-1 summarizes startup options. For example, to connect to a running simulator, type the following:

```
C:\> windsh vxsim0@hostname
```

**NOTE:** When you start the host shell, a second shell window appears, running the Debug server. You can minimize this second window to reclaim screen space, but do not close it.
You may run as many different host shells attached to the same target as you wish. The output from a function called in a particular shell appears in the window from which it was called, unless you change the shell defaults using `shConfig` (see A.2.3 Setting Shell Environment Variables, p.50).

### Host Shell Startup Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-n</code>, <code>-noinit</code></td>
<td>Do not read home Tcl initialization file.</td>
</tr>
<tr>
<td><code>-T</code>, <code>-Tclmode</code></td>
<td>Start in Tcl mode.</td>
</tr>
<tr>
<td><code>-m[ode]</code></td>
<td>Indicates mode to start in: C (the default), Tcl (Tcl</td>
</tr>
<tr>
<td><code>-v</code>, <code>-version</code></td>
<td>Display host shell version.</td>
</tr>
<tr>
<td><code>-h</code>, <code>-help</code></td>
<td>Print help.</td>
</tr>
<tr>
<td><code>-p</code>, <code>-poll</code></td>
<td>Sets event poll interval (msec, default=200).</td>
</tr>
<tr>
<td><code>-e</code>, <code>-execute</code></td>
<td>Executes Tcl expression after initialization.</td>
</tr>
<tr>
<td><code>-c</code>, <code>-command</code></td>
<td>Executes expression and exits shell (batch mode).</td>
</tr>
<tr>
<td><code>-r</code>, <code>-root mappings</code></td>
<td>Root pathname mappings.</td>
</tr>
<tr>
<td><code>-ds[DFW server Session]</code></td>
<td>Debugger Server session to use.</td>
</tr>
<tr>
<td><code>-dp[DFW server Port]</code></td>
<td>Debugger Server port to use.</td>
</tr>
<tr>
<td><code>-host</code></td>
<td>Retrieves target server information from host’s registry.</td>
</tr>
<tr>
<td><code>-s</code>, <code>-startup</code></td>
<td>Specifies the startup file of shell commands to execute.</td>
</tr>
<tr>
<td><code>-q</code>, <code>-quiet</code></td>
<td>Turns off echo of script commands as they are executed.</td>
</tr>
<tr>
<td><code>-dt target</code></td>
<td>DFW target definition name.</td>
</tr>
</tbody>
</table>
A.2.2 Switching Interpreters

At times you may want to switch from one interpreter to another. From a prompt, type these special commands and then press Enter:

- `cmd` to switch to the command interpreter. The prompt changes to `[vxWorks] #`.
- `C` to switch to the C interpreter. The prompt changes to `->`.
- `?` to switch to the Tcl interpreter. The prompt changes to `tcl>`.
- `gdb` to switch to the GDB interpreter. The prompt changes to `gdb>`.

These commands can also be used to evaluate a statement native to another interpreter. Simply precede the command you want to execute with the appropriate interpreter’s special command.

For example, to evaluate a C interpreter command from within the command interpreter, type the following:

```
[vxWorks] # C test = malloc(100); test[0] = 10; test[1] = test[0] + 2
```

A.2.3 Setting Shell Environment Variables

The host shell has a set of environment variables that configure different aspects of the shell’s interaction with the target and with the user. These environment variables can be displayed and modified using the Tcl routine `shConfig`. Table A-2 provides a list of the host shell’s environment variables and their significance.

Since `shConfig` is a Tcl routine, it should be called from within the shell’s Tcl interpreter; it can also be called from within another interpreter if you precede the `shConfig` command with a question mark (`?shConfig variable option`).

<table>
<thead>
<tr>
<th>Variable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTP_CREATE_STOP [on</td>
<td>off]</td>
</tr>
</tbody>
</table>
### A Host Shell Commands and Options

#### A.2 Host Shell Basics

<table>
<thead>
<tr>
<th>Variable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTP_CREATE_ATTACH [on</td>
<td>off]</td>
</tr>
<tr>
<td>VXE_PATH .</td>
<td>When RTP support is configured in the system, this option indicates the path in which the host shell should search for RTPs to launch. If this is set to “.”, the full pathname of an RTP should be supplied to the command to launch an RTP.</td>
</tr>
<tr>
<td>ROOT_PATH_MAPPING</td>
<td>This variable indicates how host and target paths should be mapped to the host file system on which the DFW server used by the host shell is running. If this value is not set, a direct path mapping is assumed (for example, a pathname given by /folk/user is searched; no translation to another path is performed).</td>
</tr>
<tr>
<td>LINE_LENGTH</td>
<td>This configuration variable indicates the maximum number of characters permitted in one line of the host shell’s window.</td>
</tr>
<tr>
<td>STRING_FREE [manual</td>
<td>automatic]</td>
</tr>
</tbody>
</table>
Table A-2  Host Shell Environment Variables (cont'd)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH_ALL_SYMBOLS [on</td>
<td>off]</td>
</tr>
<tr>
<td>INTERPRETER [C</td>
<td>Tcl</td>
</tr>
<tr>
<td>SH_GET_TASK_IO</td>
<td>Sets the I/O redirection mode for called functions. The default is on, which redirects input and output of called functions to WindSh. To have input and output of called functions appear in the target console, set SH_GET_TASK_IO to off.</td>
</tr>
<tr>
<td>LD_CALL_XTORS</td>
<td>Sets the C++ strategy related to constructors and destructors. The default is “target”, which causes WindSh to use the value set on the target using cplusXtorSet( ). If LD_CALL_XTORS is set to on, the C++ strategy is set to automatic (for the current WindSh only). Off sets the C++ strategy to manual for the current shell.</td>
</tr>
<tr>
<td>LD_SEND_MODULES</td>
<td>Sets the load mode. The default on causes modules to be transferred to the target server. This means that any module WindSh can see can be loaded. If LD_SEND_MODULES if off, the target server must be able to see the module to load it.</td>
</tr>
</tbody>
</table>


For example, to switch from vi mode to emacs mode when using the C interpreter, type the following:

```
-> ?shConfig LINE_EDIT_MODE emacs
```

When in command interpreter mode, you can use the commands set config and show config to set and display the environment variables listed in Table A-2.

---

<table>
<thead>
<tr>
<th>Variable</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD_PATH</td>
<td>Sets the search path for modules using the separator &quot;;&quot;. When a ld() command is issued, WindSh first searches the current directory and loads the module if it finds it. If not, WindSh searches the directory path for the module.</td>
</tr>
<tr>
<td>LD_COMMON_MATCH_ALL</td>
<td>Sets the loader behavior for common symbols. If it is set to on, the loader tries to match a common symbol with an existing one. If a symbol with the same name is already defined, the loader take its address. Otherwise, the loader creates a new entry. If set to off, the loader does not try to find an existing symbol. It creates an entry for each common symbol.</td>
</tr>
<tr>
<td>DSM_HEX_MOD</td>
<td>Sets the disassembling “symbolic + offset” mode. When set to off the “symbolic + offset” address representation is turned on and addresses inside the disassembled instructions are given in terms of “symbol name + offset.” When set to on these addresses are given in hexadecimal.</td>
</tr>
<tr>
<td>LINE_EDIT_MODE</td>
<td>Sets the line edit mode to use. Set to emacs or vi. Default is vi.</td>
</tr>
</tbody>
</table>
A.2.4 Stopping the Host Shell

Regardless of how you start it, you can terminate a host shell session by typing `exit` or `quit` at the prompt or pressing `Ctrl+D`.

A.3 Using the Command Interpreter

The command interpreter is command-oriented and does not understand C language syntax. (For C syntax, use the C interpreter as described in A.4 Using the C Interpreter, p.66.)

A command name is composed of one or more strings followed by options flags and parameters. The command interpreter syntax is a mix of GDB and UNIX syntax.

The syntax of a command is as follows:

```
command [subcommand [... subcommand]] [options] [arguments] [1]
```

*command* and *subcommand* are alphanumeric strings that do not contain spaces. *arguments* can be any string.

For example:

```
[vxWorks]# ls -l /folk/user
[vxWorks]# task delete t1
[vxWorks]# bp -t t1 0x12345678
```

The *options* and *arguments* strings may be processed differently by each command and so can follow any format. Most of the commands follow the UNIX standard. In that case, each argument and each option are separated by at least one space.

An option is composed of the dash character (-) plus one character (-o for example). Several options can be gathered in the same string (-oats is identical to -o -a -l -s).

An option may have an extra argument (-f filename). The -- option is a special option that indicates the end of the options string.

Arguments are separated by spaces. Therefore, if an argument contains a space, the space has to be escaped by a backslash ("\") character or surrounded by single or double quotes. For example:

```
[vxWorks]# ls -l "/folk/user with space characters"
[vxWorks]# ls -l /folk/user/ with\ space\ characters
```
## A.3.1 General Host Shell Commands

Table A-3 summarizes general host shell commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>Adds an alias or displays list of aliases.</td>
</tr>
<tr>
<td>bp</td>
<td>Displays, sets, or unsets a breakpoint.</td>
</tr>
<tr>
<td>cat</td>
<td>Concatenates and displays files.</td>
</tr>
<tr>
<td>cd</td>
<td>Changes current directory.</td>
</tr>
<tr>
<td>expr</td>
<td>Evaluates an expression.</td>
</tr>
<tr>
<td>help</td>
<td>Displays the list of shell commands.</td>
</tr>
<tr>
<td>ls</td>
<td>Lists the files in a directory.</td>
</tr>
<tr>
<td>more</td>
<td>Browses and pages through a text file.</td>
</tr>
<tr>
<td>print</td>
<td>Displays the symbol value of an errno.</td>
</tr>
<tr>
<td>pwd</td>
<td>Displays the current working directory.</td>
</tr>
<tr>
<td>quit</td>
<td>Shuts down the shell.</td>
</tr>
<tr>
<td>reboot</td>
<td>Reboots the system.</td>
</tr>
<tr>
<td>string free</td>
<td>Frees a string allocated by the shell on the target.</td>
</tr>
<tr>
<td>unalias</td>
<td>Removes an alias.</td>
</tr>
<tr>
<td>version</td>
<td>Displays VxWorks version information.</td>
</tr>
</tbody>
</table>
A.3.2 Displaying Target Agent Information

Table A-4 lists the commands related to the target agent.

Table A-4 Command Interpreter Target Agent Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help agent</td>
<td>Displays a list of shell commands related to the target agent.</td>
</tr>
<tr>
<td>agent info</td>
<td>Displays the agent mode: system or task.</td>
</tr>
<tr>
<td>agent status</td>
<td>Displays the system context status: suspended or running. This command can</td>
</tr>
<tr>
<td></td>
<td>be completed successfully only if the agent is running in system (external)</td>
</tr>
<tr>
<td></td>
<td>mode.</td>
</tr>
<tr>
<td>agent system</td>
<td>Sets the agent to system (external) mode then suspends the system,</td>
</tr>
<tr>
<td></td>
<td>halting all tasks. When the agent is in external mode, certain commands</td>
</tr>
<tr>
<td></td>
<td>(bp, task step, task continue) work with the system context instead of a</td>
</tr>
<tr>
<td></td>
<td>particular task context.</td>
</tr>
<tr>
<td>agent task</td>
<td>Resets the agent to tasking mode and resumes the system.</td>
</tr>
</tbody>
</table>

A.3.3 Working with Memory

Table A-5 shows the commands related to memory.

Table A-5 Command Interpreter Memory Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help memory</td>
<td>Lists shell commands related to memory.</td>
</tr>
<tr>
<td>mem dump</td>
<td>Displays memory.</td>
</tr>
<tr>
<td>mem modify</td>
<td>Modifies memory values.</td>
</tr>
<tr>
<td>mem info</td>
<td>Displays memory information.</td>
</tr>
<tr>
<td>mem list</td>
<td>Disassembles and displays a specified number of instructions.</td>
</tr>
</tbody>
</table>
A.3.4 **Displaying Object Information**

Table A-6 shows commands that display information about objects.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help objects</td>
<td>Lists shell commands related to objects.</td>
</tr>
<tr>
<td>object info</td>
<td>Displays information about one or more specified objects.</td>
</tr>
<tr>
<td>object class</td>
<td>Shows information about a class of objects.</td>
</tr>
</tbody>
</table>

A.3.5 **Working with Symbols**

Table A-7 lists commands for displaying and setting values of symbols.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help symbols</td>
<td>Lists shell commands related to symbols.</td>
</tr>
<tr>
<td>echo</td>
<td>Displays a line of text or prints a symbol value.</td>
</tr>
<tr>
<td>printf</td>
<td>Writes formatted output.</td>
</tr>
<tr>
<td>set or set symbol</td>
<td>Sets the value of a symbol.</td>
</tr>
<tr>
<td>lookup</td>
<td>Looks up a symbol.</td>
</tr>
</tbody>
</table>

**Accessing a Symbol’s Contents and Address**

The host shell command interpreter is a string-oriented interpreter, but a user may want to distinguish between symbol names, regular strings, and numerical values. When a symbol name is passed as an argument to a command, the user may want to specify either the symbol address (for example, to set a hardware breakpoint on that address) or the symbol value (to display it).
To do this, a symbol should be preceded by the character & to access the symbol’s address, and $ to access a symbol’s contents. Any commands that specify a symbol should now also specify the access type for that symbol. For example:

```
[vxWorks]# task spawn &printf %c $toto.r
```

In this case, the command interpreter sends the address of the text symbol `printf` to the `task spawn` command. It accesses the contents of the data symbol `toto` and, due to the `.r` suffix, it accesses the data symbol as a character.

The commands `printf` and `echo` are available in the shell for easy display of symbol values.

**Symbol Value Access**

When specifying that a symbol is of a particular numerical value type, use the following:

```
$symName [.type]
```

The special characters accepted for `type` are as follows:

- r = chaR
- h = sHort
- i = Integer (default)
- l = Long
- ll = Long Long
- f = Float
- d = Double

For example, if the value of the symbol name `value` is 0x10, type the following:

```
[vxWorks]# echo $value
0x10
```

But:

```
[vxWorks]# echo value
value
```

By default, the command interpreter considers a numerical value to be a 32-bit integer. If a numerical string contains a “.” character, or the `E` or `e` characters (such as 2.0, 2.1e1, or 3.5E2), the command interpreter considers the numerical value to be a double value.

**Symbol Address Access**

When specifying that a symbol should be replaced by a string representing the address of the symbol, precede the symbol name by a `&` character.
For example, if the address of the symbol name `value` is 0x12345678, type the following:

```
[vxWorks]$ echo &value
0x12345678
```

**Special Consideration of Text Symbols**

The “value” of a text symbol is meaningless, but the symbol address of a text symbol is the address of the function. So to specify the address of a function as a command argument, use a `&` character.

For example, to set a breakpoint on the `printf()` function, type the following:

```
[vxWorks]$ bp &printf
```

### A.3.6 Displaying, Controlling, and Stepping Through Tasks

Table A-8 displays commands for working with tasks.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>help tasks</code></td>
<td>Lists the shell commands related to working with tasks.</td>
</tr>
<tr>
<td><code>task</code></td>
<td>Displays a summary of each task’s TCB.</td>
</tr>
<tr>
<td><code>task info</code></td>
<td>Displays complete information from a task’s TCB.</td>
</tr>
<tr>
<td><code>task spawn</code></td>
<td>Spawns a task with default parameters.</td>
</tr>
<tr>
<td><code>task stack</code></td>
<td>Displays a summary of each task’s stack usage.</td>
</tr>
<tr>
<td><code>task delete</code></td>
<td>Deletes one or more tasks.</td>
</tr>
<tr>
<td><code>task default</code></td>
<td>Sets or displays the default task.</td>
</tr>
<tr>
<td><code>task trace</code></td>
<td>Displays a stack trace of a task.</td>
</tr>
<tr>
<td><code>task regs</code></td>
<td>Sets task register value.</td>
</tr>
<tr>
<td><code>show task regs</code></td>
<td>Displays task register values.</td>
</tr>
<tr>
<td><code>task suspend</code></td>
<td>Suspends a task or tasks.</td>
</tr>
<tr>
<td><code>task resume</code></td>
<td>Resumes a task or tasks.</td>
</tr>
</tbody>
</table>
A.3.7 Setting Shell Context Information

Table A-9 displays commands for displaying and setting context information.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help set</td>
<td>Lists shell commands related to setting context information.</td>
</tr>
<tr>
<td>set or set symbol</td>
<td>Sets the value of an existing symbol. If the symbol does not exist, and if the current working context is the kernel, a new symbol is created and registered in the kernel symbol table.</td>
</tr>
<tr>
<td>set bootline</td>
<td>Changes the boot line used in the boot ROMs.</td>
</tr>
<tr>
<td>set config</td>
<td>Sets or displays shell configuration variables.</td>
</tr>
<tr>
<td>set cwc</td>
<td>Sets the current working context of the shell session.</td>
</tr>
<tr>
<td>set history</td>
<td>Sets the size of shell history. If no argument is specified, displays shell history.</td>
</tr>
<tr>
<td>set prompt</td>
<td>Changes the shell prompt to the string specified. The following special characters are accepted:</td>
</tr>
<tr>
<td></td>
<td>%/ : current path</td>
</tr>
<tr>
<td></td>
<td>%n : current user</td>
</tr>
<tr>
<td></td>
<td>%m : target server name</td>
</tr>
<tr>
<td></td>
<td>%% : display % character</td>
</tr>
<tr>
<td></td>
<td>%c : current RTP name</td>
</tr>
</tbody>
</table>
A.3.8 Displaying System Status

Table A-10 lists commands for showing system status information.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>show bootline</td>
<td>Displays the current boot line of the kernel.</td>
</tr>
<tr>
<td>show devices</td>
<td>Displays all devices known to the I/O system.</td>
</tr>
<tr>
<td>show drivers</td>
<td>Displays all system drivers in the driver list.</td>
</tr>
<tr>
<td>show fds</td>
<td>Displays all opened file descriptors in the system.</td>
</tr>
<tr>
<td>show history</td>
<td>Displays the history events of the current interpreter.</td>
</tr>
<tr>
<td>show lasterror</td>
<td>Displays the last error value set by a command.</td>
</tr>
</tbody>
</table>

A.3.9 Using and Modifying Aliases

The command interpreter accepts aliases to speed up access to shell commands. Table A-11 lists the aliases that already exist; they can be modified, and you can add new aliases. Aliases are visible from all shell sessions.

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>List existing aliases. Add a new alias by typing alias aliasname &quot;command&quot;. For example, alias ll &quot;ls -l&quot;.</td>
</tr>
<tr>
<td>attach</td>
<td>rtp attach</td>
</tr>
<tr>
<td>b</td>
<td>bp</td>
</tr>
</tbody>
</table>
Table A-11  Command Interpreter Aliases (cont'd)

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>bd</td>
<td>bp -u</td>
</tr>
<tr>
<td>bdall</td>
<td>bp -u #*</td>
</tr>
<tr>
<td>bootChange</td>
<td>set bootline</td>
</tr>
<tr>
<td>c</td>
<td>task continue</td>
</tr>
<tr>
<td>checkStack</td>
<td>task stack</td>
</tr>
<tr>
<td>creset</td>
<td>task continue -r</td>
</tr>
<tr>
<td>d</td>
<td>mem dump</td>
</tr>
<tr>
<td>detach</td>
<td>rtp detach</td>
</tr>
<tr>
<td>devs</td>
<td>show devices</td>
</tr>
<tr>
<td>emacs</td>
<td>set config LINE_EDIT_MODE=&quot;emacs&quot;</td>
</tr>
<tr>
<td>h</td>
<td>show history</td>
</tr>
<tr>
<td>i</td>
<td>task</td>
</tr>
<tr>
<td>jobs</td>
<td>rtp attach</td>
</tr>
<tr>
<td>kill</td>
<td>rtp detach</td>
</tr>
<tr>
<td>l</td>
<td>mem list</td>
</tr>
<tr>
<td>lkAddr</td>
<td>lookup -a</td>
</tr>
<tr>
<td>lkup</td>
<td>lookup</td>
</tr>
<tr>
<td>m</td>
<td>mem modify</td>
</tr>
<tr>
<td>memShow</td>
<td>mem info</td>
</tr>
<tr>
<td>ps</td>
<td>rtp</td>
</tr>
<tr>
<td>rpc</td>
<td>rtp continue</td>
</tr>
<tr>
<td>rtpd</td>
<td>rtp delete</td>
</tr>
<tr>
<td>rtpi</td>
<td>rtp task</td>
</tr>
</tbody>
</table>
A.3.10 Launching RTPs

From the command interpreter, type the RTP pathname as a regular command, adding any command arguments after the RTP pathname (as in a UNIX shell).

[vxWorks]# /folk/user/TMP/helloworld.vxe
Launching process '/folk/user/TMP/helloworld.vxe' ...
Process '/folk/user/TMP/helloworld.vxe' (process Id = 0x471630) launched.

[vxWorks]# rtp
NAME ID STATUS ENTRY ADDR SIZE TASK CNT
----------------- ----- -------- -------- ------ -----
[folk/user/TMP/cal 12 2004
Launching process '/folk/user/TMP/cal' ...
December 2004
S M Tu W Th F S
1 2 3 4
5 6 7 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
Process '/folk/user/TMP/cal' (process Id = 0x2fd60) launched.

Table A-11 Command Interpreter Aliases (cont’d)

<table>
<thead>
<tr>
<th>Alias</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>rtps</td>
<td>rtp stop</td>
</tr>
<tr>
<td>run</td>
<td>rtp exec</td>
</tr>
<tr>
<td>s</td>
<td>task step</td>
</tr>
<tr>
<td>so</td>
<td>task stepover</td>
</tr>
<tr>
<td>td</td>
<td>task delete</td>
</tr>
<tr>
<td>ti</td>
<td>task info</td>
</tr>
<tr>
<td>tr</td>
<td>task resume</td>
</tr>
<tr>
<td>ts</td>
<td>task suspend</td>
</tr>
<tr>
<td>tsp</td>
<td>task spawn</td>
</tr>
<tr>
<td>tt</td>
<td>task trace</td>
</tr>
<tr>
<td>vi</td>
<td>set config LINE_EDIT_MODE=&quot;vi&quot;</td>
</tr>
</tbody>
</table>
Redirecting Output to the Host Shell

To launch an RTP in the foreground, simply launch it as usual:

\[
\text{[vxWorks]} \# \text{rtp exec myRTP.exe}
\]

To launch an RTP in the background but redirect its output to the host shell, include the -i option:

\[
\text{[vxWorks]} \# \text{rtp exec -i myRTP.exe}
\]

To move the RTP to the background and stop it, press Ctrl+W. To resume an RTP in the background that is stopped, use the command \text{rtp background}.

To move an RTP to the foreground, use the command \text{rtp foreground}.

To kill the RTP, press Ctrl+C.

To redirect output for all processes to the host shell, use the Tcl function \text{vioSet} as shown below:

\[
\text{proc vioSet {}} {\
\hspace{1em}#Set stdin, stdout, and stderr to /vio/0 if not already in use\
\hspace{1em}# puts stdout "set stdin stdout stderr here (y/n)?"\
\hspace{1em}if { [shParse {tstz = open (/vio/0,2,0)}] != -1 } {\
\hspace{2em}shParse {vf0 = tstz};\
\hspace{2em}shParse {ioGlobalStdSet (0,vf0)} ;\
\hspace{2em}shParse {ioGlobalStdSet (1,vf0)} ;\
\hspace{2em}shParse {ioGlobalStdSet (2,vf0)} ;\
\hspace{2em}shParse {logFdSet (vf0)} ;\
\hspace{2em}shParse {printf ("Std I/O set here!");\
\hspace{2em}} else {\
\hspace{3em}shParse {printf ("Std I/O unchanged.");\
\hspace{2em}}\
\hspace{1em}}}
\]

Monitoring and Debugging RTPs

Table A-12 displays the commands related to RTPs.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help RTP</td>
<td>Displays a list of the shell commands related to RTPs.</td>
</tr>
<tr>
<td>help rtp</td>
<td>Displays shell commands related to RTPs, with synopses.</td>
</tr>
</tbody>
</table>
Setting Breakpoints

The `bp` command has been designed to set breakpoint in the kernel, in a RTP, for any task, for a particular task or a particular context. A breakpoint number is assigned to each breakpoint, which can be used to remove that breakpoint.

```bash
bp [-p <rtpIdNameNumber>] [-t <taskId>] [[-u {<#bp number> | <bp address>} ...] | [-n <count>] [-h <type>] [-q] [-a] [expr]]
```

This command is used to set or unset (if the option `-u` is specified) a breakpoint. The breakpoint is a hardware breakpoint if the option `-h` is specified. Without any arguments, this command displays the breakpoints currently set.

The special breakpoint number `'*'` or breakpoint address `'*'` is used to unset all the breakpoints.

- `-a`: stop all tasks in a context,
- `-n`: number of passes before hit,
- `-h`: specify a hardware breakpoint type value,
- `-p`: breakpoint applies to specify RTP.
VxWorks

- `q`: no notification when the breakpoint is hit,
- `t`: breakpoint applies to specify task,
- `u`: unset breakpoint

Breakpoints can be set in a memory context only if the current working memory context is set to that memory context.

A.4 Using the C Interpreter

The host shell running in C interpreter mode interprets and executes almost all C-language expressions and allows prototyping and debugging in kernel space (it does not provide access to processes; use the command interpreter mode described on page 54 to debug processes and RTPs).

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 commands parallel interactive utilities that can be linked into VxWorks itself. By using the host shell commands, you minimize the impact on both target memory and performance.

A.4.1 Managing Tasks

Table A-13 summarizes the commands that manage tasks.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sp()</code></td>
<td>Spawns a task with default parameters.</td>
</tr>
<tr>
<td><code>sps()</code></td>
<td>Spawns a task, but leaves it suspended.</td>
</tr>
<tr>
<td><code>tr()</code></td>
<td>Resumes a suspended task.</td>
</tr>
<tr>
<td><code>ts()</code></td>
<td>Suspends a task.</td>
</tr>
<tr>
<td><code>td()</code></td>
<td>Deletes a task.</td>
</tr>
<tr>
<td><code>period()</code></td>
<td>Spawns a task to call a function periodically.</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()`. 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.

Table A-13 summarizes the commands that report task information.

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()`. 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.

Table A-14 summarizes the commands that report task information.

Table A-13 C Interpreter Task Management Commands (cont'd)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>repeat()</code></td>
<td>Spawns a task to call a function repeatedly.</td>
</tr>
<tr>
<td><code>taskIdDefault()</code></td>
<td>Sets or reports the default (current) task ID.</td>
</tr>
</tbody>
</table>

Table A-14 C Interpreter Task Information Reporting Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i()</code></td>
<td>Displays 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><code>iStrict()</code></td>
<td>Displays the same information as <code>i()</code>, but queries target system information only once. At the expense of consuming more intermediate memory, this guarantees an accurate snapshot.</td>
</tr>
<tr>
<td><code>ti()</code></td>
<td>Displays task information. This command gives all the information contained in a task’s TCB. This includes everything shown for that task by an <code>i()</code> 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><code>w()</code></td>
<td>Prints 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 on a specified task if the argument is given.</td>
</tr>
<tr>
<td><code>tw()</code></td>
<td>Prints information about the object the given task is pending on. This routine calls <code>taskWaitShow()</code> on the given task in verbose mode.</td>
</tr>
</tbody>
</table>
The `i()` command is commonly used to get a quick report on target activity. If nothing seems to be happening, `i()` is often a good place to start investigating. To display summary information about all running tasks, type the following:

```
-> i
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Entry</th>
<th>TID</th>
<th>Pri</th>
<th>Status</th>
<th>PC</th>
<th>SP</th>
<th>Errno</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>tExcTask</td>
<td>_excTask</td>
<td>3ad290</td>
<td>0</td>
<td>PEND</td>
<td>4df10</td>
<td>3ad0c0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tLogTask</td>
<td>_logTask</td>
<td>3aa918</td>
<td>0</td>
<td>PEND</td>
<td>4df10</td>
<td>3aa748</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tWdbTask</td>
<td>0x41288</td>
<td>387060</td>
<td>0</td>
<td>READY</td>
<td>23ff4</td>
<td>386d78</td>
<td>3d0004</td>
<td>0</td>
</tr>
<tr>
<td>tNetTask</td>
<td>_netTask</td>
<td>3a59c0</td>
<td>50</td>
<td>READY</td>
<td>24200</td>
<td>3a5730</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tFtpdTask</td>
<td>_ftpdTask</td>
<td>3a2c18</td>
<td>55</td>
<td>PEND</td>
<td>23b28</td>
<td>3a2938</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Value = 0 = 0x0

A.4.2 Displaying System Information

Table A-15 shows the commands that display information from the symbol table, from the target system, and from the shell itself.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devs()</td>
<td>Lists all devices known on the target system.</td>
</tr>
<tr>
<td>lkup()</td>
<td>Lists symbols from symbol table.</td>
</tr>
<tr>
<td>lkAddr()</td>
<td>Lists symbols whose values are near a specified 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:

```bash
-> lkup "dsm"
_dsmData 0x00049d08 text (vxWorks)
_dsmNbytes 0x00049d76 text (vxWorks)
_dsmInst 0x00049d28 text (vxWorks)
mydsm 0x003c6510 bss  (vxWorks)
```

Case is significant, but position is not (`mydsm` is shown, but `myDsm` would not be). To explicitly write a search that would match either `mydsm` or `myDsm`, you could write the following:

```bash
-> lkup "[dD]sm"
```

---

**Table A-15  C Interpreter System Information Commands (cont'd)**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d()</td>
<td>Displays 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>Disassembles and displays a specified number of instructions.</td>
</tr>
<tr>
<td>printErrno()</td>
<td>Describes the most recent error status value.</td>
</tr>
<tr>
<td>version()</td>
<td>Prints VxWorks version information.</td>
</tr>
<tr>
<td>cd()</td>
<td>Changes the host working directory (no effect on target).</td>
</tr>
<tr>
<td>ls()</td>
<td>Lists files in host working directory.</td>
</tr>
<tr>
<td>pwd()</td>
<td>Displays the current host working directory.</td>
</tr>
<tr>
<td>help()</td>
<td>Displays a summary of selected shell commands.</td>
</tr>
<tr>
<td>h()</td>
<td>Displays up to 20 lines of command history.</td>
</tr>
<tr>
<td>shellHistory</td>
<td>Sets or displays shell history.</td>
</tr>
<tr>
<td>shellPromptSet</td>
<td>Changes the C-interpreter shell prompt.</td>
</tr>
<tr>
<td>printLogo()</td>
<td>Displays the shell logo.</td>
</tr>
</tbody>
</table>
A.4.3 **Modifying and Debugging the Target**

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 A-16 summarizes the commands of this type.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld()</td>
<td>Loads an object module into target memory and links it dynamically into the run-time.</td>
</tr>
<tr>
<td>unld()</td>
<td>Removes a dynamically linked object module from target memory, and frees the storage it occupied.</td>
</tr>
<tr>
<td>m()</td>
<td>Modifies memory in \textit{width} (byte, short, or long) starting at \textit{adr}. The \textit{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 \texttt{ENTER}, or return to the shell by typing a dot (.).</td>
</tr>
<tr>
<td>mRegs()</td>
<td>Modifies register values for a particular task.</td>
</tr>
<tr>
<td>b()</td>
<td>Sets or displays breakpoints, in a specified task or in all tasks.</td>
</tr>
<tr>
<td>bh()</td>
<td>Sets a hardware breakpoint.</td>
</tr>
<tr>
<td>s()</td>
<td>Steps a program to the next instruction.</td>
</tr>
<tr>
<td>so()</td>
<td>Single-steps, but steps over a subroutine.</td>
</tr>
<tr>
<td>c()</td>
<td>Continues from a breakpoint.</td>
</tr>
<tr>
<td>cret()</td>
<td>Continues until the current subroutine returns.</td>
</tr>
<tr>
<td>bdall()</td>
<td>Deletes all breakpoints.</td>
</tr>
<tr>
<td>bd()</td>
<td>Deletes a breakpoint.</td>
</tr>
<tr>
<td>reboot()</td>
<td>Returns target control to the target boot ROMs, then resets the target server and reattaches the shell.</td>
</tr>
<tr>
<td>bootChange()</td>
<td>Modifies the saved values of boot parameters.</td>
</tr>
</tbody>
</table>
A Host Shell Commands and Options
A.4 Using the C Interpreter

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.

A.4.4 Running Target Routines from the Host Shell

All target routines are available from the host shell. 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.

---

Table A-16 C Interpreter System Modification and Debugging Commands (cont’d)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysSuspend()</td>
<td>If supported by the target agent configuration, enters system mode.</td>
</tr>
<tr>
<td>sysResume()</td>
<td>If supported by the target agent (and if system mode is in effect), returns to task mode from system mode.</td>
</tr>
<tr>
<td>agentModeShow()</td>
<td>Shows the agent mode (system or task).</td>
</tr>
<tr>
<td>sysStatusShow()</td>
<td>Shows the system context status (suspended or running).</td>
</tr>
<tr>
<td>quit() or exit()</td>
<td>Dismisses the shell.</td>
</tr>
</tbody>
</table>

The m() command provides an interactive way of manipulating target memory.
Invocations of VxWorks Subroutines

-> taskSpawn ("tmyTask", 10, 0, 1000, myTask, fd1, 300)
value = ...

-> fd = open ("file", 0, 0)
new symbol "fd" added to symbol table
fd = (...address of fd...): value = ...

Invocations of Application Subroutines

-> testFunc (123)
value = ...

-> myValue = myFunc (1, &val, testFunc (123))
myValue = (...address of myValue...): value = ...

-> myDouble = (double ()) myFuncWhichReturnsADouble (x)
myDouble = (...address of myDouble...): value = ...

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().

A.5 Using the Tcl Interpreter

The Tcl interpreter allows you to access the WTX Tcl API, and to exploit Tcl’s
sophisticated scripting capabilities to write complex scripts to help you debug and
monitor your target.
To switch to the Tcl interpreter from another mode, type a question mark (?) at the prompt; the prompt changes to `tcl>` to remind you of the shell’s new mode. If you are in another interpreter mode and want to use a Tcl command without changing to Tcl mode, type a ? before your line of Tcl code.

⚠️ **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.

### A.5.1 Accessing the WTX Tcl API

The WTX Tcl API allows you to launch and kill a process, and to apply several actions to it such as debugging actions (continue, stop, step), memory access (read, write, set), perform gopher string evaluation, and redirect I/O at launch time.

A real time process (RTP) can be seen as a protected memory area. One or more tasks can run in an RTP or in the kernel memory context as well. It is not possible to launch a task or perform load actions in an RTP, therefore an RTP is seen by the target server only as a memory context.

For a complete listing of WTX Tcl API commands, consult the `wtxtcl` reference entries.

### A.6 Using the GDB Interpreter

The GDB interpreter provides a command-line GDB interface to the host shell, and permits the use of GDB commands to debug a target.

#### A.6.1 General GDB Commands

Table A-17 lists general commands available within the GDB interpreter.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>help command</code></td>
<td>Prints a description of the command.</td>
</tr>
</tbody>
</table>
A.6.2 Working with Breakpoints

Table A-18 shows commands available for setting and manipulating breakpoints.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b[reak]</code></td>
<td>Sets a breakpoint</td>
</tr>
<tr>
<td><code>t[break]</code></td>
<td>Sets a temporary breakpoint.</td>
</tr>
<tr>
<td><code>enable</code></td>
<td>Enables a breakpoint (+options <code>once</code> and <code>del</code>).</td>
</tr>
<tr>
<td><code>disable</code></td>
<td>Disables a breakpoint.</td>
</tr>
<tr>
<td><code>delete</code></td>
<td>Deletes a breakpoint.</td>
</tr>
<tr>
<td><code>clear</code></td>
<td>Clears a breakpoint.</td>
</tr>
</tbody>
</table>
A.6 Using the GDB Interpreter

### A.6.3 Specifying Files to Debug

Table A-19 lists commands that specify the file(s) to be debugged.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>file</strong></td>
<td><em>filename</em> Defines <em>filename</em> as the program to be debugged.</td>
</tr>
<tr>
<td><strong>exec-file</strong></td>
<td><em>filename</em> Specifies that the program to be run is found in <em>filename</em>.</td>
</tr>
<tr>
<td><strong>load</strong></td>
<td><em>filename</em> Loads a module.</td>
</tr>
<tr>
<td><strong>unload</strong></td>
<td><em>filename</em> Unloads a module.</td>
</tr>
<tr>
<td><strong>attach</strong></td>
<td><em>processid</em> Attaches to a process.</td>
</tr>
<tr>
<td><strong>detach</strong></td>
<td></td>
</tr>
<tr>
<td><strong>thread</strong></td>
<td><em>threadid</em> Selects a thread as the current task to debug.</td>
</tr>
<tr>
<td><strong>add-symbol-file</strong></td>
<td><em>file</em> <em>addr</em> Reads additional symbol table information from the <em>file</em> located at memory address <em>addr</em>.</td>
</tr>
</tbody>
</table>
A.6.4 Running and Stepping Through a File

Table A-20 contains commands to run and step through programs.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run</td>
<td>Runs a process for debugging (use <code>set arguments</code> and <code>set environment</code> if program needs them).</td>
</tr>
<tr>
<td>kill <code>processid</code></td>
<td>Kills a process.</td>
</tr>
<tr>
<td>interrupt</td>
<td>Interrupts a running task or process.</td>
</tr>
<tr>
<td>continue</td>
<td>Continues an interrupted task or process.</td>
</tr>
<tr>
<td>step <code>[n]</code></td>
<td>Steps through a program (if <code>n</code> is used, step <code>n</code> times).</td>
</tr>
<tr>
<td>steipi <code>[n]</code></td>
<td>Steps through one machine instruction (if <code>n</code> is used, step through <code>n</code> instructions).</td>
</tr>
<tr>
<td>next <code>[n]</code></td>
<td>Continues to the next source line in the current stack frame (if <code>n</code> is used, continue through <code>n</code> lines).</td>
</tr>
<tr>
<td>nexti <code>[n]</code></td>
<td>Execute one machine instruction, but if it is a function call, proceed until the function returns (if <code>n</code> is used, execute <code>n</code> instructions).</td>
</tr>
<tr>
<td>until</td>
<td>Continue running until a source line past the current line, in the current stack frame, is reached.</td>
</tr>
<tr>
<td>jump <code>address</code></td>
<td>Moves the instruction pointer to <code>address</code>.</td>
</tr>
<tr>
<td>finish</td>
<td>Finishes execution of current block.</td>
</tr>
</tbody>
</table>
A.6.5 Displaying Disassembler and Memory Information

Table A-21 lists commands for disassembling code and displaying contents of memory.

Table A-21  GDB Interpreter Disassembly and Memory Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disassemble</td>
<td>Disassembles code at a specified address.</td>
</tr>
<tr>
<td>x [format] address</td>
<td>Displays memory starting at address. format is one of the</td>
</tr>
<tr>
<td></td>
<td>formats used by print: s for null-terminated string, or i for</td>
</tr>
<tr>
<td></td>
<td>machine instruction. Default is x for hexadecimal initially,</td>
</tr>
<tr>
<td></td>
<td>but the default changes each time you use either x or print.</td>
</tr>
</tbody>
</table>

A.6.6 Examining Stack Traces and Frames

Table A-22 shows commands for selecting and displaying stack frames.

Table A-22  GDB Interpreter Stack Trace and Frame Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt [n]</td>
<td>Displays back trace of n frames.</td>
</tr>
<tr>
<td>frame [n]</td>
<td>Selects frame number n.</td>
</tr>
<tr>
<td>up [n]</td>
<td>Move n frames up the stack.</td>
</tr>
<tr>
<td>down [n]</td>
<td>Moves n frames down the stack.</td>
</tr>
</tbody>
</table>

A.6.7 Displaying Information and Expressions

Table A-23 lists commands that display functions, registers, expressions, and other debugging information.

Table A-23  GDB Interpreter Information and Expression Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info args</td>
<td>Shows function arguments.</td>
</tr>
</tbody>
</table>
A.6.8 Displaying and Setting Variables

Table A-24 lists commands for displaying and setting variables.

### Table A-24 GDB Interpreter Variable Display and Set Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set args arguments</td>
<td>Specifies the arguments to be used the next time a debugged program is run.</td>
</tr>
<tr>
<td>set emacs</td>
<td>Sets display into emacs mode.</td>
</tr>
<tr>
<td>set environment varname = value</td>
<td>Sets environment variable varname to value. value may be any string interpreted by the program.</td>
</tr>
<tr>
<td>set tgt-path-mapping</td>
<td>Sets target to host pathname mappings.</td>
</tr>
<tr>
<td>set variable expression</td>
<td>Sets variable value to expression.</td>
</tr>
</tbody>
</table>
A 7 Using the Built-in Line Editor

The host shell provides various line editing facilities available from the library **ledLib** (Line Editing Library). **ledLib** serves as an interface between the user input and the underlying command-line interpreters, and facilitates the user’s interactive shell session by providing a history mechanism and the ability to scroll, search, and edit previously typed commands. Any input is treated by **ledLib** until the user presses the **ENTER** key, at which point the command typed is sent on to the appropriate interpreter.

The line editing library also provides command completion, path completion, command matching, and synopsis printing functionality.

### A.7.1 vi-Style Editing

The **ESC** key switches the shell from normal input mode to edit mode. The history and editing commands in Table A-25 and Table A-27 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** 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 previously typed line, press **ESC** followed by one of the search commands listed in Table A-26; you can then edit and execute the line with one of the commands from the table.

### Table A-24  GDB Interpreter Variable Display and Set Commands (cont’d)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>show args</td>
<td>Shows arguments of debugged program.</td>
</tr>
<tr>
<td>show environment</td>
<td>Shows environment of debugged program.</td>
</tr>
</tbody>
</table>
Switching Modes and Controlling the Editor

Table A-25 lists commands that give you basic control over the editor.

Table A-25  vi-Style Basic Control Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h [size]</td>
<td>Displays shell history if no argument is given; otherwise sets 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 current interpreter and leave edit mode.</td>
</tr>
<tr>
<td>CTRL+D</td>
<td>Complete symbol or pathname (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 pathname (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 (edit mode).</td>
</tr>
<tr>
<td>CTRL+S</td>
<td>Suspend output.</td>
</tr>
<tr>
<td>CTRL+Q</td>
<td>Resume output.</td>
</tr>
<tr>
<td>CTRL+W</td>
<td>Display HTML reference entry for a routine.</td>
</tr>
</tbody>
</table>

Moving and Searching in the Editor

Table A-26 lists commands for moving and searching in input mode.

Table A-26  vi-Style 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. The default value for n is 1.</td>
</tr>
<tr>
<td>/s or ?s</td>
<td>Search for string s backward or forward in history.</td>
</tr>
<tr>
<td>n</td>
<td>Repeat last search.</td>
</tr>
</tbody>
</table>
A Host Shell Commands and Options

A.7 Using the Built-in Line Editor

Inserting and Changing Text

Table A-26 lists commands to insert and change text in the editor.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n\text{k}) or (n-)</td>
<td>Get (n)th previous shell command.</td>
</tr>
<tr>
<td>(n\text{j}) or (n+)</td>
<td>Get (n)th next shell command.</td>
</tr>
<tr>
<td>(n\text{h})</td>
<td>Go left (n) characters (also CTRL+H).</td>
</tr>
<tr>
<td>(n\text{l}) or SPACE</td>
<td>Go right (n) characters.</td>
</tr>
<tr>
<td>(n\text{w}) or (n\text{W})</td>
<td>Go (n) words forward, or (n) large words. Words are separated by spaces or punctuation; large words are separated by spaces only.</td>
</tr>
<tr>
<td>(n\text{e}) or (n\text{E})</td>
<td>Go to end of the (n)th next word, or (n)th next large word.</td>
</tr>
<tr>
<td>(n\text{b}) or (n\text{B})</td>
<td>Go back (n) words, or (n) large words.</td>
</tr>
<tr>
<td>$</td>
<td>Go to end of line.</td>
</tr>
<tr>
<td>0 or ^</td>
<td>Go to beginning of line, or to first nonblank character.</td>
</tr>
<tr>
<td>(f\text{c}) or (F\text{c})</td>
<td>Find character (c), searching forward or backward.</td>
</tr>
</tbody>
</table>

Table A-27 vi-Style Insertion and Change Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a or A ...ESC</td>
<td>Append, or append at end of line (ESC ends input).</td>
</tr>
<tr>
<td>i or I ...ESC</td>
<td>Insert, or insert at beginning of line (ESC ends input).</td>
</tr>
<tr>
<td>ns ...ESC</td>
<td>Change (n) characters (ESC ends input).</td>
</tr>
<tr>
<td>cw ...ESC</td>
<td>Change word (ESC ends input).</td>
</tr>
<tr>
<td>cc or S ...ESC</td>
<td>Change entire line (ESC ends input).</td>
</tr>
<tr>
<td>c$ or C ...ESC</td>
<td>Change from cursor to end of line (ESC ends input).</td>
</tr>
<tr>
<td>c0 ...ESC</td>
<td>Change from cursor to beginning of line (ESC ends input).</td>
</tr>
</tbody>
</table>
### vi-Style Insertion and Change Commands (cont’d)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R ...ESC</td>
<td>Type over characters (ESC ends input).</td>
</tr>
<tr>
<td>nrc</td>
<td>Replace the following n characters with c.</td>
</tr>
<tr>
<td>~</td>
<td>Toggle between lower and upper case.</td>
</tr>
</tbody>
</table>

### Deleting Text

Table A-28 shows commands for deleting text.

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

### Put and Undo Commands

Table A-29 shows put and undo commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p or P</td>
<td>Put last deletion after cursor, or in front of cursor.</td>
</tr>
<tr>
<td>u</td>
<td>Undo last command.</td>
</tr>
</tbody>
</table>
A.7.2 emacs-Style Editing

The shell history mechanism is similar to the UNIX Tcsh shell history facility, with a built-in line editor similar to emacs that allows previously typed commands to be edited. The command `h()` displays the 20 most recent commands typed into the shell; old commands fall off the top as new ones are entered.

To edit a command, the arrow keys can be used on most of the terminals. Up arrow and down arrow move up and down through the history list, like `CTRL+P` and `CTRL+N`. Left arrow and right arrow move the cursor left and right one character, like `CTRL+B` and `CTRL+F`.

### Moving the Cursor

Table A-30 lists commands for moving the cursor in emacs mode.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL+B</td>
<td>Move cursor back (left) one character.</td>
</tr>
<tr>
<td>CTRL+F</td>
<td>Move cursor forward (right) one character.</td>
</tr>
<tr>
<td>ESC+b</td>
<td>Move cursor back one word.</td>
</tr>
<tr>
<td>ESC+f</td>
<td>Move cursor forward one word.</td>
</tr>
<tr>
<td>CTRL+A</td>
<td>Move cursor to beginning of line.</td>
</tr>
<tr>
<td>CTRL+E</td>
<td>Move cursor to end of line.</td>
</tr>
</tbody>
</table>

### Deleting and Recalling Text

Table A-31 shows commands for deleting and recalling text.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEL or CTRL+H</td>
<td>Delete character to left of cursor.</td>
</tr>
<tr>
<td>CTRL+D</td>
<td>Delete character under cursor.</td>
</tr>
</tbody>
</table>
Table A-31 emacs-Style Deletion and Recall Commands (cont’d)

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESC+d</td>
<td>Delete word.</td>
</tr>
<tr>
<td>ESC+DEL</td>
<td>Delete previous word.</td>
</tr>
<tr>
<td>CTRL+K</td>
<td>Delete from cursor to end of line.</td>
</tr>
<tr>
<td>CTRL+U</td>
<td>Delete entire line.</td>
</tr>
<tr>
<td>CTRL+P</td>
<td>Get previous command in the history.</td>
</tr>
<tr>
<td>CTRL+N</td>
<td>Get next command in the history.</td>
</tr>
<tr>
<td>!n</td>
<td>Recall command n from the history.</td>
</tr>
<tr>
<td>!substr</td>
<td>Recall first command from the history matching substr.</td>
</tr>
</tbody>
</table>

Special Commands

Table A-32 shows some special emacs-mode commands.

Table A-32 Special emacs-Style Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL+U</td>
<td>Delete line and leave edit mode.</td>
</tr>
<tr>
<td>CTRL+L</td>
<td>Redraw line.</td>
</tr>
<tr>
<td>CTRL+D</td>
<td>Complete symbol name.</td>
</tr>
<tr>
<td>ENTER</td>
<td>Give line to interpreter and leave edit mode.</td>
</tr>
</tbody>
</table>

A.7.3 Command Matching

Whenever the beginning of a command is followed by CTRL+D, ledLib lists any commands that begin with the string entered.
To avoid ambiguity, the commands displayed depend upon the current interpreter mode. For example, if a command string is followed by CTRL+D from within the C interpreter, `ledLib` attempts to list any VxWorks symbols matching the pattern. If the same is performed from within the command interpreter, `ledLib` attempts to list any commands available from within command mode that begin with that string.

Directory and File Matching

You can also use CTRL+D to list all the files and directories that match a certain string. This functionality is available from all interpreter modes.

A.7.4 Command and Path Completion

`ledLib` attempts to complete any string typed by the user that is followed by the TAB character (for commands, the command completion is specific to the currently active interpreter).

Path completion attempts to complete a directory name when the TAB key is pressed. This functionality is available from all interpreter modes.

A.8 Running the Host Shell in Batch Mode

The host shell can also be run in batch mode, with commands passed to the host shell using the -c option followed by the command(s) to execute.

The commands must be delimited with double quote characters. The default interpreter mode used to execute the commands is the C interpreter; to execute commands in a different mode, specify the mode with the -mode option. It is not possible to execute a mixed mode command with the -c option.

For example:

1. To launch the host shell in batch mode, executing the Command interpreter commands `task` and `rtp task`, type the following:

```% windsh -m cmd -c "task ; rtp task" tgtsvr@host```
The -m option indicates that the commands should be executed by the Command interpreter.

2. To launch the host shell in batch mode, executing the tcl mode commands puts and expr, type the following:

```bash
% windsh -m tcl -c "puts helloworld; expr 33 + 22" tgsvr@host
```
Symbols

symbols - see assignment operators, asterisk, at, backslashes, quotation marks, slashes, spaces

A

add (vxprj) 15
ADDED_C++FLAGS 32
ADDED_CFLAGS 32
ADDED_CLEAN_LIBS 32
ADDED_DYN_EXE_FLAGS 32
ADDED_LIB_DEPS 33
ADDED_LIBS 32
ADDED_SHARED_LIBS 33
aliases, host shell 61
archives
   makefile 31
at symbol (@) 44
autoscale 22

B

backslashes 2

host-shell commands 43
Tel 22
batch mode, host shell 85
board support package
   defined 12
   documentation 12
boot-loader 38
breakpoints, setting in host shell
   C interpreter 44
   command mode 43
   GDB mode 45
   list of commands 74
BSPs - see board support package
build rules
   changing 28
   defined 26
   examining 27
build specifications
   defined 26
   examining 27
   setting 27
bundles
   adding to projects 16
   defined 16
   listing and examining 19
   removing from projects 16
C

C interpreter (host shell) 44
.cdf files 11
  components 15
CDL 11
check (vxprj component) 19
cmpProjClose 22
cmpProjCopy 22
cmpProjCreate 22
cmpProjOpen 22
cmpScriptLib
  and vxprj 22
  building projects 29
documentation 21
project management 12
  using 21
cmpTest 20
code-browsing 1
compiler
  and make utility 26
documentation 2
GNU 1
option flags (C) 32
option flags (C++) 32
selecting 26
  TOOL make variable 35
  vxprj 13
Wind River (Diab) 1
component (vxprj) 15
component dependencies 20
Component Description Language (CDL) 11
components
  adding to projects 15
  CDL and .cdf files 11
  checking syntax 19
defined 11
  listing (projects) 18
  removing from projects 16
configuration parameters
  listing and examining 20
  setting 17
CPU (make variable) 33

D

documentation
  BSPs 12
compiler 2
generating 33
host shell 41
make utility 30
projects 11
vxprj 12
VxWorks 1
VxWorks Simulator 14
Workbench 1
downloadable kernel modules
  building 29
IDE project management 12
dynamic libraries, building 34

e

-e 7
editor
  host shell 79
eemacs-style editing, host shell 83
environment variables
  displaying with wrenv 8
  host shell 50
  setting with wrenv 5
error detection 16
eval subkeys (properties files) 10
EXCLUDE_SUBDIRS 33
EXE (make variable) 33
EXE_FORMAT 33
EXTRA_INCLUDE 33

F

-f 7
files
  adding and removing in projects 18
  examining source 21
file-system projects 12
  building 29

G

GCC (GNU Compiler) 1
documentation 2
GDB (GNU Debugger) mode, host shell 45
gnu
  specifying compiler with vxprj 13

H

host shell
  aliases 61
  batch mode 85
  breakpoints
    C interpreter 44
    command mode 43
    GDB mode 45
    list of GDB-mode commands 74
  C interpreter 44
  command mode 42
  debugging 41
  detailed commands 47
  documentation 41
  editor 79
  environment variables 50
  GDB mode 45
  help 41
  memory 56
  modes, switching 42
  starting 42
  stopping 42
  symbols 57
  tasks 59
    C interpreter 66
    Tcl interpreter 45

I

-i 7
IDE
  exclusive features 1
  project files 11
  project management 12
  wrenv 5
install.properties
  syntax 9
  wrenv 5
IP address 38

K

kernel configuration
  vxprj 11
kernel configuration projects
  building 25
    cmpScriptLib 29
    vxprj 26
  creating
    cmpProjCreate 22
    vxprj 12
  defined 12
kernel shell 16
ledLib 79
LIB_BASE_NAME 33
LIB_DIR 34
LIB_FORMAT 34
LIBNAME 33
libraries
- linking to RTP applications 32
- makefiles 31
- shared (dynamic), building 34
- static, building 34
- user-mode (RTP), building 30
linux (simulator BSP) 14
LOCAL_CLEAN 34
LOCAL_RCLEAN 34

make utility 1
- calling directly 30
- dmake 30
- documentation 30
- sh command 30
- wrenv, configuring environment 30
- Z shell 30
make variables 32
makefiles
- archives (libraries) 31
- changing name 17
- editing generated files 12
- editing, and project files 30
- RTPs (applications) 31
- RTPs (libraries) 31
- vxprj-generated 14
make-rule files 31
memory, host-shell commands 56
multicore debugging 1

OBJ_DIR 34
object files
- specifying build output 34
OBJS 34

packages
- displaying with wrenv 8
- install.properties 9
- Tcl 21
- parameter (vxprj) 17
- POSIX 16
- print_compatible (wrenv -o) 8
- print_env (wrenv -o) 8
- print_package_name (wrenv -o) 8
- print_packages (wrenv -o) 8
- print_vars (wrenv -o) 7
- profile 15
- PROFILE_DEVELOPMENT 14
profiles
- creating projects 14
- defined 14
- listing and examining 19
project files
- .wpj 11
- .wrproject 11
- editing generated files 12
- IDE 11
- makefiles
- editing 30
projects
- building 25
- cmpScriptLib 29
- vxprj 26
- comparing
- components 19
- parameters 21
- copying 14
- defined 11
- deleting 15
- directory structure 31
- documentation 11
- managing 11
with Tcl scripts 21
see also kernel configuration projects, VxWorks
image project, downloadable kernel
modules, shared libraries, RTPs, file-
system projects
types of project
building 25
managing 12
vxprj 12

Q
quotation marks
on command line 17

R
registry 37
remove (vxprj) 16
—RTP
—RTP
building RTP libraries 32
RTPs (real time processes)
building 30
bundle 16
creating projects 23
makefiles for applications 31
makefiles for libraries 31
project management 12
running in host shell 43
VxWorks libraries, building 32
VxWorks simulator 39
rules.library 32
and LOCAL_CLEAN 34
and LOCAL_RCLEAN 34
rules.rtp 32
and LOCAL_CLEAN 34
and LOCAL_RCLEAN 34
example of use 31
run (wrenv -o) 7

S
serial terminal 38
sh command
make utility 30
sh.exe
Z shell 6
called by make 30
shared libraries 12
building 34
linking to RTP applications 33
shell
development 5
host
ddebugging 41
detailed commands 47
see also host shell
dkernel (target) 47
dZ (zsh) 6
called by make 30
dimpc 14
dimulator (VxWorks)
BSPs 13, 14
documentation 14
dkernel configuration projects 14
drunning 38
SL_INSTALL_DIR 34
SL_VERSION 34
dslashes
tcl 22
dUNIX and Windows 2
d.so files 33
dsolaris (simulator BSP) 14
dspaces
host shell 43
don command line 17
dstatic libraries, building 34
dSUBDIRS 34
dsymbols - see assignment operators, asterisk, at,
dbacksslashes, quotation marks, slashes,
dspaces
dsymbols, in host shell 57
T

T

T

T

T

T

T

T

Target connecting 37

Target agent 48

Target server 48

Tasks, in host shell 59

C interpreter 66

Tcl interpreter (host shell) 45

Tcl libraries

documentation 21

Project management 12, 21

See also cmpScriptLib

tgtsrv 38

TOOL (make variable) 35

V

V

V

V

V

V

V

V

V

Variables

environment 5

displaying with wrenv 7

host shell 50

VxWorks

documentation 1

VxWorks image project

defined 12

See also kernel configuration projects

VxWorks Simulator

See simulator

VxWorks60 package 10

W

Workbench
documentation 1

Project management 12

.WPj files 11

Components 15

Wrenv

IDE startup 5

Options 7

Overview 5

.WrProject files 11

_WRS_KERNEL

Building RTP libraries 32

WtxConsole 38

Wtcl, WTX 45

Z

Z shell 6

called by make 30