WIND RIVER

VxWorks®
6.2

COMMAND-LINE TOOLS USER'S GUIDE

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

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

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

http://www.windriver.com

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

http://www.windriver.com/support

VxWorks Command-Line Tools User's Guide, 6.2

6 Oct 05
Part #: DOC-15678-ZD-00

Contents

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

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 2-1 **Options for wrenv**

Option	Meaning	Example
-e	Do not redefine existing environment variables. (Variables identified with **addpath** in **install.properties** are still modified.)	`% wrenv.sh -p vxworks-6.0 -e`
-f *format*	Select *format* for **print_env** or **print_vars** (see **-o**): **plain** (the default) **sh** **csh** **bat** **tcl**	`C:\> wrenv -p vxworks-6.0 -o print_vars -f sh`
-i *path*	Specify the location of **install.properties**. (Overrides the default method of finding **install.properties**.)	`% wrenv.sh -p vxworks-6.0 -i` *directoryPath*`/install.properties`
-o *operation*	Select *operation*:	

	run
	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).
		`C:\> wrenv -p vxworks-6.0 -o run`
		
		`% wrenv.sh -p vxworks-6.0 -o run`

	print_vars[a] Show environment settings that would be made if **wrenv** were invoked with **-o run**.	`C:\> wrenv -o print_vars`

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

Option	Meaning	Example
	print_env[a] 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**.)	`% `**`wrenv.sh -o print_env`**
	print_packages[a] List the packages defined in **install.properties** and their attributes. (The displayed *name* of a package can later be specified with the **-p** option.)	`C:\> `**`wrenv -o print_packages`**
	print_compatible[a] 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.	`% `**`wrenv.sh -p vxworks-6.0 -o print_compatible`**
	print_package_name[a] Use with **-r**. Show the name of the package in the specified root directory.	`C:\> `**`wrenv -r`** *directory* **`-o print_package_name`**
-p *package*	Specify a *package* (a set of Workbench components) for environment initialization. The package must be defined in **install.properties**.	`% `**`wrenv.sh -p vxworks-6.0`**
-r *root*	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.	`C:\> `**`wrenv -p vxworks-6.0 -r`** *directory*
-v	Verbose mode. Show all altered environment settings.	`% `**`wrenv.sh -p vxworks-6.0 -v`**

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

Option	Meaning	Example
env=*value*	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).	`C:\> wrenv -p vxworks-6.0 PATH=`*directory*
command [*args*]	Execute the specified command in the new shell environment. (Does not open a new shell.) Must be the last item on the command line.	`% wrenv.sh -p vxworks-6.0 ls *.sh`

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.

2.4 install.properties and package.properties

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*] ...]

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
Working with Projects and Components

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[1]; 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.[2] 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 (**simpc**). 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:

```
% 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:

```
% 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:

```
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:

```
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:

```
% 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.

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:

vxprj file add [*projectFile*] *sourceFile*

If no project file is specified, **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:

vxprj file remove [*projectFile*] *sourceFile*

3.2.4 Generating Project and Component Diagnostics

Obtaining a List of Components

To see a list of components, type the following:

vxprj component list [*projectFile*] [*type*] [*pattern*]

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

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

For example:

```
% vxprj component list MyProject.wpj SHELL
```

This command returns all components in **MyProject.wpj** whose names contain "SHELL", such as **INCLUDE_SHELL_BANNER** and **INCLUDE_RTP_SHELL_C**.

```
% vxprj component list MyProject.wpj excluded VM
```

This command returns all available components with names containing "VM" that are *not* included in **MyProject.wpj**.

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".

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

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:

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

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

```
cmpProjOpen projectFile
cmpBuild
```

For example:

```
# 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:

```
% 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:

```
# 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:

```
# 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 listed 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 Programmer's Guide: Applications and Processes*.

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**.

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:

```
% tgtsvr 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 **tgtsrv** 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 *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

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*] [**;**]

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)
```

Call a function from your application.
```
-> someFunction (1,2,3)
```
Call an application function that has the same name as a shell command.
```
-> @i()
```

6.6 Tcl Mode

The Tcl interpreter provides access to the WTX Tcl API., which allows you to write complex scripts for testing an debugging. For a complete listing of WTX Tcl API commands, see the **wtxtcl** reference entries.

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
Host Shell Commands and Options

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 A-1 **Host Shell Startup Options**

Option	Description
-n, **-noinit**	Do not read home Tcl initialization file.
-T, **-Tclmode**	Start in Tcl mode.
-m[ode]	Indicates mode to start in: C (the default), Tcl (**Tcl** | **tcl** | **TCL**), GDB (**Gdb** | **gdb** | **GDB**), or Cmd (**Cmd** | **cmd** | **CMD**).
-v, **-version**	Display host shell version.
-h, **-help**	Print help.
-p, **-poll**	Sets event poll interval (msec, default=200).
-e, **-execute**	Executes Tcl expression after initialization.
-c, **-command**	Executes expression and exits shell (batch mode).
-r, **-root mappings**	Root pathname mappings.
-ds[DFW server Session]	Debugger Server session to use.
-dp[DFW server Port]	Debugger Server port to use.
-host	Retrieves target server information from host's registry.
-s, **-startup**	Specifies the startup file of shell commands to execute.
-q, **-quiet**	Turns off echo of script commands as they are executed.
-dt *target*	DFW target definition name.

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 A-2 **Host Shell Environment Variables**

Variable	Result
RTP_CREATE_STOP [on | off]	When RTP support is configured in the system, this option indicates whether RTPs launched via the host shell (using the host shell's command interpreter) should be launched in the stopped or running state.

Table A-2 **Host Shell Environment Variables** (cont'd)

Variable	Result
RTP_CREATE_ATTACH [on | off]	When RTP support is configured in the system, this option indicates whether the shell should automatically attach to any RTPs launched from the host shell (using the host shell's command interpreter).
VXE_PATH .	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.
ROOT_PATH_MAPPING	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).
LINE_LENGTH	This configuration variable indicates the maximum number of characters permitted in one line of the host shell's window.
STRING_FREE [manual | automatic]	.This configuration variable indicates whether strings allocated on the target by the host shell should be freed automatically by the shell, or whether they should be left for the user to free manually using the C interpreter API **strFree( ).**

Table A-2 **Host Shell Environment Variables** (cont'd)

Variable	Result
SEARCH_ALL_SYMBOLS [on | off]	This variable indicates whether symbol searches should be confined to global symbols or should search all symbols. If **SEARCH_ALL_SYMBOLS** is set to **on**, any request for a symbol searches the entire symbol table contents. This is equivalent to a symbol search performed on a target server launched with the **-A** option. Note that if the **SEARCH_ALL_SYMBOLS** flag is set to **on**, there is a considerable performance impact on commands performing symbol manipulation.
INTERPRETER [C | Tcl | Cmd | Gdb]	This variable indicates the host shell's current interpreter mode and permits the user to switch from one mode to another.
SH_GET_TASK_IO	Sets the I/O redirection mode for called functions. The default is **on**, which redirects input and output of called functions to WindSh. To have input and output of called functions appear in the target console, set **SH_GET_TASK_IO** to **off**.
LD_CALL_XTORS	Sets the C++ strategy related to constructors and destructors. The default is "target", which causes WindSh to use the value set on the target using **cplusXtorSet( )**. If **LD_CALL_XTORS** is set to **on**, the C++ strategy is set to automatic (for the current WindSh only). **Off** sets the C++ strategy to manual for the current shell.
LD_SEND_MODULES	Sets the load mode. The default **on** causes modules to be transferred to the target server. This means that any module WindSh can see can be loaded. If **LD_SEND_MODULES** if **off**, the target server must be able to see the module to load it.

Table A-2 **Host Shell Environment Variables** (cont'd)

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

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.

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*] [**;**]

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 -t -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 A-3 **General Command Interpreter Commands**

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

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**

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

A.3.3 Working with Memory

Table A-5 shows the commands related to memory.

Table A-5 **Command Interpreter Memory Commands**

Command	Description
help memory	Lists shell commands related to memory.
mem dump	Displays memory.
mem modify	Modifies memory values.
mem info	Displays memory information.
mem list	Disassembles and displays a specified number of instructions.

A.3.4 Displaying Object Information

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

Table A-6 **Command Interpreter Object Commands**

Command	Description
help objects	Lists shell commands related to objects.
object info	Displays information about one or more specified objects.
object class	Shows information about a class of objects.

A.3.5 Working with Symbols

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

Table A-7 **Command Interpreter Symbol Commands**

Command	Description
help symbols	Lists shell commands related to symbols.
echo	Displays a line of text or prints a symbol value.
printf	Writes formatted output.
set or **set symbol**	Sets the value of a symbol.
lookup	Looks up a symbol.

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 A-8 **Command Interpreter Task Commands**

Command	Description
help tasks	Lists the shell commands related to working with tasks.
task	Displays a summary of each tasks's TCB.
task info	Displays complete information from a task's TCB.
task spawn	Spawns a task with default parameters.
task stack	Displays a summary of each tasks's stack usage.
task delete	Deletes one or more tasks.
task default	Sets or displays the default task.
task trace	Displays a stack trace of a task.
task regs	Sets task register value.
show task regs	Displays task register values.
task suspend	Suspends a task or tasks.
task resume	Resumes a task or tasks.

Table A-8 **Command Interpreter Task Commands** (cont'd)

Command	Description
task hooks	Displays task hook functions.
task stepover	Single-steps a task or tasks.
task stepover	Single steps, but steps over a subroutine.
task continue	Continues from a breakpoint.
task stop	Stops a task.

A.3.7 Setting Shell Context Information

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

Table A-9 **Command Interpreter Shell Context Commands**

Command	Description
help set	Lists shell commands related to setting context information.
set or **set symbol**	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.
set bootline	Changes the boot line used in the boot ROMs.
set config	Sets or displays shell configuration variables.
set cwc	Sets the current working context of the shell session.
set history	Sets the size of shell history. If no argument is specified, displays shell history.
set prompt	Changes the shell prompt to the string specified. The following special characters are accepted: **%/** : current path **%n** : current user **%m** : target server name **%%** : display % character **%c** : current RTP name

Table A-9 **Command Interpreter Shell Context Commands** (cont'd)

Command	Description
unset config	Removes a shell configuration variable from the current shell session.

A.3.8 Displaying System Status

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

Table A-10 **Command Interpreter System Status Commands**

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

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 A-11 **Command Interpreter Aliases**

Alias	Definition
alias	List existing aliases. Add a new alias by typing **alias** *aliasname "command"*. For example, **alias ll "ls -l"**.
attach	**rtp attach**
b	**bp**

Table A-11 **Command Interpreter Aliases** (cont'd)

Alias	Definition
bd	bp -u
bdall	bp -u #*
bootChange	set bootline
c	task continue
checkStack	task stack
cret	task continue -r
d	mem dump
detach	rtp detach
devs	show devices
emacs	set config LINE_EDIT_MODE="emacs"
h	show history
i	task
jobs	rtp attach
kill	rtp detach
l	mem list
lkAddr	lookup -a
lkup	lookup
m	mem modify
memShow	mem info
ps	rtp
rtpc	rtp continue
rtpd	rtp delete
rtpi	rtp task

Table A-11 **Command Interpreter Aliases** (cont'd)

Alias	Definition
rtps	**rtp stop**
run	**rtp exec**
s	**task step**
so	**task stepover**
td	**task delete**
ti	**task info**
tr	**task resume**
ts	**task suspend**
tsp	**task spawn**
tt	**task trace**
vi	**set config LINE_EDIT_MODE="vi"**

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
------------ ---------- ----------- ---------- ---------- --------

[vxWorks]# /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  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 = 0x2fdfb0) launched.
```

Redirecting Output to the Host Shell

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

```
[vxWorks]# rtp exec myRTP.exe
```

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

```
[vxWorks]# 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 **rtp background**.

To move an RTP to the foreground, use the command **rtp foreground**.

To kill the RTP, press **Ctrl+C**.

To redirect output for all processes to the host shell, use the Tcl function **vioSet** as shown below:

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

    } else {
   shParse {printf ("Std I/O unchanged.

    }
}
```

Monitoring and Debugging RTPs

Table A-12 displays the commands related to RTPs.

Table A-12 **Command Interpreter RTP Commands**

Command	Description
help RTP	Displays a list of the shell commands related to RTPs.
help rtp	Displays shell commands related to RTPs, with synopses.

Table A-12 **Command Interpreter RTP Commands** (cont'd)

Command	Description
rtp	Displays a list of processes.
rtp stop	Stops a process.
rtp continue	Continues a process.
rtp delete	Deletes a process (or list of processes).
rtp info	Displays process information.
rtp exec	Executes a process.
rtp attach	Attaches the shell session to a process.
rtp detach	Detaches the shell session from a process.
set cwc	Sets the current working context of the shell session.
rtp task	Lists tasks running within a particular RTP.
rtp foreground	Brings the current or specified process to the shell foreground.
rtp background	Runs the current or specified process in the shell background.

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.

```
bp                  Display, set or unset a breakpoint
                    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,
-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 A-13 **C Interpreter Task Management Commands**

Command	Description
sp( )	Spawns a task with default parameters.
sps( )	Spawns a task, but leaves it suspended.
tr( )	Resumes a suspended task.
ts( )	Suspends a task.
td( )	Deletes a task.
period( )	Spawns a task to call a function periodically.

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

Command	Description
repeat( )	Spawns a task to call a function repeatedly.
taskIdDefault( )	Sets or reports the default (current) task ID.

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-14 **C Interpreter Task Information Reporting Commands**

Command	Description
i( )	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.
iStrict( )	Displays the same information as **i( )**, but queries target system information only once. At the expense of consuming more intermediate memory, this guarantees an accurate snapshot.
ti( )	Displays task information. This command gives all the information contained in a task's TCB. This includes everything shown for that task by an **i( )** command, plus all the task's registers, and the links in the TCB chain. If *task* is 0 (or the argument is omitted), the current task is reported on.
w( )	Prints a summary of each task's pending information, task by task. This routine calls **taskWaitShow( )** in quiet mode on all tasks in the system, or on a specified task if the argument is given.
tw( )	Prints information about the object the given task is pending on. This routine calls **taskWaitShow( )** on the given task in verbose mode.

Table A-14 **C Interpreter Task Information Reporting Commands** (cont'd)

Command	Description
checkStack( )	Shows a stack usage summary for a task, or for all tasks if no task is specified. The summary includes the total stack size (SIZE), the current number of stack bytes (CUR), the maximum number of stack bytes used (HIGH), and the number of bytes never used at the top of the stack (MARGIN = SIZE - HIGH). Use this routine to determine how much stack space to allocate, and to detect stack overflow. This routine does not work for tasks that use the **VX_NO_STACK_FILL** option.
tt( )	Displays a stack trace.
taskIdFigure( )	Reports a task ID, given its name.

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
  NAME        ENTRY       TID    PRI   STATUS     PC       SP     ERRNO  DELAY
--------- ----------- -------- --- -------- -------- -------- ------- -----
tExcTask  _excTask     3ad290   0 PEND       4df10   3ad0c0       0     0
tLogTask  _logTask     3aa918   0 PEND       4df10   3aa748       0     0
tWdbTask  0x41288      3870f0   3 READY      23ff4   386d78  3d0004     0
tNetTask  _netTask     3a59c0  50 READY      24200   3a5730       0     0
tFtpdTask _ftpdTask    3a2c18  55 PEND       23b28   3a2938       0     0
value = 0 = 0x0
```

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 A-15 **C Interpreter System Information Commands**

Command	Description
devs( )	Lists all devices known on the target system.
lkup( )	Lists symbols from symbol table.
lkAddr( )	Lists symbols whose values are near a specified value.

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

Command	Description
d( )	Displays target memory. You can specify a starting address, size of memory units, and number of units to display.
l( )	Disassembles and displays a specified number of instructions.
printErrno( )	Describes the most recent error status value.
version( )	Prints VxWorks version information.
cd( )	Changes the host working directory (no effect on target).
ls( )	Lists files in host working directory.
pwd( )	Displays the current host working directory.
help( )	Displays a summary of selected shell commands.
h( )	Displays up to 20 lines of command history.
shellHistory( )	Sets or displays shell history.
shellPromptSet( )	Changes the C-interpreter shell prompt.
printLogo( )	Displays the shell logo.

The lkup( ) command takes a regular expression as its argument, and looks up all symbols containing strings that match. In the simplest case, you can specify a substring to see any symbols containing that string. For example, to display a list containing routines and declared variables with names containing the string *dsm*, do the following:

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

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

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

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 A-16 **C Interpreter System Modification and Debugging Commands**

Command	Description
ld( )	Loads an object module into target memory and links it dynamically into the run-time.
unld( )	Removes a dynamically linked object module from target memory, and frees the storage it occupied.
m( )	Modifies memory in *width* (byte, short, or long) starting at *adr*. The **m( )** command displays successive words in memory on the terminal; you can change each word by typing a new hex value, leave the word unchanged and continue by typing **ENTER**, or return to the shell by typing a dot (**.**).
mRegs( )	Modifies register values for a particular task.
b( )	Sets or displays breakpoints, in a specified task or in all tasks.
bh( )	Sets a hardware breakpoint.
s( )	Steps a program to the next instruction.
so( )	Single-steps, but steps over a subroutine.
c( )	Continues from a breakpoint.
cret( )	Continues until the current subroutine returns.
bdall( )	Deletes all breakpoints.
bd( )	Deletes a breakpoint.
reboot( )	Returns target control to the target boot ROMs, then resets the target server and reattaches the shell.
bootChange( )	Modifies the saved values of boot parameters.

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

Command	Description
sysSuspend( )	If supported by the target agent configuration, enters system mode.
sysResume( )	If supported by the target agent (and if system mode is in effect), returns to task mode from system mode.
agentModeShow( )	Shows the agent mode (*system* or *task*).
sysStatusShow( )	Shows the system context status (*suspended* or *running).*
quit( ) or **exit( )**	Dismisses the shell.

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.

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 A-17 **General GDB Interpreter Commands**

Command	Description
help *command*	Prints a description of the command.

Table A-17 **General GDB Interpreter Commands** (cont'd)

Command	Description
cd *directory*	Changes the current directory.
pwd	Shows the current directory.
path *path*	Appends *path* to the **path** variable.
show path	Shows the **path** variable.
echo *string*	Echoes the string.
list *line* | *symbol* | *file:line*	Displays 10 lines of a source file, centered around a line number or symbol.
shell *command*	Runs a SHELL command (such as **ls** or **dir**).
source *scriptfile*	Runs a script of GDB commands.
directory *di*	Appends *dir* to the **directory** variable (for source file searches.)
q[uit]	Quits the GDB interpreter.

A.6.2 Working with Breakpoints

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

Table A-18 **GDB Interpreter Breakpoint Commands**

Command	Description
b[reak] *symbol* | *line* | *file:line* **[if** *expr***]**	Sets a breakpoint
t[break] *symbol* | *line* | *file:line* **[if** *expr***]**	Sets a temporary breakpoint.
enable *breakpointid*	Enables a breakpoint (+options **once** and **del**).
disable *breakpointid*	Disables a breakpoint.
delete *breakpointid*	Deletes a breakpoint.
clear *breakpointid*	Clears a breakpoint.

Table A-18 **GDB Interpreter Breakpoint Commands** (cont'd)

Command	Description
cond *breakpointid condition*	Changes a breakpoint condition (re-initializes the breakpoint).
ignore *breakpointid n*	Ignores a breakpoint *n* times (re-initializes the breakpoint).

A.6.3 Specifying Files to Debug

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

Table A-19 **GDB Interpreter File Context Commands**

Command	Description
file *filename*	Defines *filename* as the program to be debugged.
exec-file *filename*	Specifies that the program to be run is found in *filename*.
load *filename*	Loads a module.
unload *filename*	Unloads a module.
attach *processid*	Attaches to a process.
detach	Detaches from the debugged process.
thread *threadid*	Selects a thread as the current task to debug.
add-symbol-file *file addr*	Reads additional symbol table information from the *file* located at memory address *addr*.

A.6.4 Running and Stepping Through a File

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

Table A-20 **GDB Interpreter Running and Stepping Commands**

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

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**

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

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**

Command	Description
bt [*n*]	Displays back trace of *n* frames.
frame [*n*]	Selects frame number *n*.
up [*n*]	Move *n* frames up the stack.
down [*n*]	Moves *n* frames down the stack.

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**

Command	Description
info args	Shows function arguments.

Table A-23 **GDB Interpreter Information and Expression Commands** (cont'd)

Command	Description
info breakpoints	Shows breakpoints.
info extensions	Shows file extensions (c, c++, ...)
info functions	Shows all functions.
info locals	Shows local variables.
info registers	Shows contents of registers.
info source	Shows current source file.
info sources	Shows all source files of current process.
info target	Displays information about the target.
info threads	Shows all threads.
info warranty	Shows disclaimer information.
print /x *expression*	Evaluates and prints an *expression* in hexadecimal format.

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**

Command	Description
set args *arguments*	Specifies the *arguments* to be used the next time a debugged program is run.
set emacs	Sets display into **emacs** mode.
set environment *varname* = *value*	Sets environment variable *varname* to *value*. *value* may be any string interpreted by the program.
set tgt-path-mapping	Sets target to host pathname mappings.
set variable *expression*	Sets variable value to *expression*.

Table A-24 **GDB Interpreter Variable Display and Set Commands** (cont'd)

Command	Description
show args	Shows arguments of debugged program.
show environment	Shows environment of debugged program.

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.

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**

Command	Description
h [*size*]	Displays shell history if no argument is given; otherwise sets history buffer to *size*.
ESC	Switch to line editing mode from regular input mode.
ENTER	Give line to current interpreter and leave edit mode.
CTRL+D	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).
[tab]	Complete symbol or pathname (edit mode).
CTRL+H	Delete a character (backspace).
CTRL+U	Delete entire line (edit mode).
CTRL+L	Redraw line (edit mode).
CTRL+S	Suspend output.
CTRL+Q	Resume output.
CTRL+W	Display HTML reference entry for a routine.

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**

Command	Description
*n***G**	Go to command number *n*. The default value for *n* is 1.
/*s* or **?***s*	Search for string *s* backward or forward in history.
n	Repeat last search.

Table A-26 **vi-Style Movement and Search Commands** (cont'd)

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

Inserting and Changing Text

Table A-27 lists commands to insert and change text in the editor.

Table A-27 **vi-Style Insertion and Change Commands**

Command	**Description**
a or **A** **...ESC**	Append, or append at end of line (**ESC** ends input).
i or **I** **...ESC**	Insert, or insert at beginning of line (**ESC** ends input).
*n***s** **...ESC**	Change *n* characters (**ESC** ends input).
cw **...ESC**	Change word (**ESC** ends input).
cc or **S** **...ESC**	Change entire line (**ESC** ends input).
c$ or C **...ESC**	Change from cursor to end of line (**ESC** ends input).
c0 **...ESC**	Change from cursor to beginning of line (**ESC** ends input).

Table A-27 **vi-Style Insertion and Change Commands** (cont'd)

Command	**Description**
R ...ESC	Type over characters (**ESC** ends input).
*n***r***c*	Replace the following *n* characters with *c*.
~	Toggle between lower and upper case.

Deleting Text

Table A-28 shows commands for deleting text.

Table A-28 **vi-Style Commands for Deleting Text**

Command	**Description**
*n***x** or *n***X**	Delete next *n* characters or previous *n* characters, starting at cursor.
dw	Delete word.
dd	Delete entire line (also **CTRL+U**).
d$ or **D**	Delete from cursor to end of line.
d0	Delete from cursor to beginning of line.

Put and Undo Commands

Table A-29 shows put and undo commands.

Table A-29 **vi-Style Put and Undo Commands**

Command	**Description**
p or **P**	Put last deletion after cursor, or in front of cursor.
u	Undo last command.

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 A-30 **emacs-Style Cursor Motion Commands**

Command	Description
CTRL+B	Move cursor back (left) one character.
CTRL+F	Move cursor forward (right) one character.
ESC+b	Move cursor back one word.
ESC+f	Move cursor forward one word.
CTRL+A	Move cursor to beginning of line.
CTRL+E	Move cursor to end of line.

Deleting and Recalling Text

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

Table A-31 **emacs-Style Deletion and Recall Commands**

Command	Description
DEL or **CTRL+H**	Delete character to left of cursor.
CTRL+D	Delete character under cursor.

Table A-31 **emacs-Style Deletion and Recall Commands** (cont'd)

Command	Description
ESC+d	Delete word.
ESC+DEL	Delete previous word.
CTRL+K	Delete from cursor to end of line.
CTRL+U	Delete entire line.
CTRL+P	Get previous command in the history.
CTRL+N	Get next command in the history.
!*n*	Recall command *n* from the history.
!*substr*	Recall first command from the history matching *substr.*

Special Commands

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

Table A-32 **Special emacs-Style Commands**

Command	Description
CTRL+U	Delete line and leave edit mode.
CTRL+L	Redraw line.
CTRL+D	Complete symbol name.
ENTER	Give line to interpreter and leave edit mode.

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 **-m[ode]** 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:

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

Index