WIND RIVER

VxWorks®

COMMAND-LINE TOOLS USER'S GUIDE

6.6

Copyright 2007 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

http://www.windriver.com/company/terms/trademark.html

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location: *installDir*/*product_name*/**3rd_party_licensor_notice.pdf**.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

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, 6.6

6 Nov 07
Part #: DOC-16079-ND-00

Contents

1 Overview

1.1 Introduction

This guide describes the command-line host tools provided with VxWorks. It complements the *Wind River Workbench User's Guide* for programmers who prefer to do 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, host-target communication, 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. Setting Environment Variables*.

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 environment variables in 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.

1.3 Related Documents

Wind River Workbench Host Shell User's Guide
Wind River Workbench User's Guide
Wind River VxWorks Kernel Programmer's Guide
Wind River VxWorks Application Programmer's Guide
Wind River VxWorks Simulator User's Guide

2 Setting Environment Variables

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-6.6
```

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-6.6 -o print_env -f shell`
```

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

```
% eval `./wrenv.sh -p vxworks-6.6 -o print_env -f sh`
```

Windows

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

```
C:\> installDir\wrenv.exe -p vxworks-6.6
```

You can also invoke **wrenv** and start a development shell window using the shortcut installed under **Start > Programs > Wind River > VxWorks 6.6 & General Purpose Technologies > VxWorks Development Shell**. This shortcut invokes **wrenv** to open a Windows command prompt configured with the proper environment variables.

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-6.6 -e`
-f *format*	Select *format* for **print_env** or **print_vars** (see **-o**): **plain** (the default) **sh** **csh** **bat** **tcl**	`C:\> wrenv -p vxworks-6.6 -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-6.6 -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-6.6 -o run`
		
		`% wrenv.sh -p vxworks-6.6 -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-6.6 -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-6.6`
-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-6.6 -r` *directory*
-v	Verbose mode. Show all altered environment settings.	`% wrenv.sh -p vxworks-6.6 -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-6.6 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-6.6 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, **vxworks66**) identifies each package. Subkeys include **name**, **version**, and so forth. The entries for a typical package look like this:

```
vxworks66.name=vxworks-6.6
vxworks66.version=6.6
vxworks66.type=platform
vxworks66.subtype=vxworks
vxworks66.label=Wind River VxWorks 6.6

# this entry shows which version of Workbench works with vxworks-6.6
vxworks66.compatible=workbench-2.2

# eval entries tell wrenv to make environment settings; see below
vxworks66.eval.01=export WIND_HOME=$(builtin:InstallHome)
vxworks66.eval.02=export WIND_BASE=$(WIND_HOME)$/vxworks-6.6
vxworks66.eval.03=require workbench-3.0
vxworks66.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
Configuring and Building VxWorks

3.1 Introduction

This chapter explains how to create and modify VxWorks projects with the **vxprj** utility and related Tcl libraries. It contains information about VxWorks components and kernel configuration. It also describes the use of profiles and bundles to provide higher-level logical groupings of components.

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.

Wind River provides a variety of tools for managing projects:

- For most purposes, **vxprj** is the easiest way to manage kernel configuration 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.
- The default make system that uses Makefiles and the config.h file provide BSP and driver developers a way to build kernels. In general, this method of build management should only be used for early-stage development.

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 or **vxprj** project management tools, you should not directly edit generated project files or makefiles

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. Configuring and Building 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**] [**-smp**] [**-profile** *profile*] *BSP* *tool* [*projectFile* | *projectDirectory*]

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 (the **PROFILE_BASIC_KERNEL** and **PROFILE_MINIMAL_KERNEL**) support the **-source** option. If you modify a pre-defined kernel profile by including additional components, building from source may not be possible. If you specify the **-source** option, and building from source is not possible, **vxprj** will notify you of that fact and use the libraries to build.

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

-smp

An optional argument that enables an SMP kernel. **Vxprj** will now verify that BSPs support SMP and will give an error when the feature is not available. For example,

```
vxprj create -smp sb1480_0_mips64 diab
Exception while creating project : "option SMP not available for BSP
sb1480_0_mips64"
```

-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.21, and the *VxWorks Kernel Programmer's Guide: Kernel*.

BSP

The name or location of a BSP.

tool

A recognized compiler (usually **diab** or **gnu**). See *Checking the Toolchain*, p.20 for information about determining valid compilers for a given BSP.

2. For information about BSPs, see the *VxWorks BSP Developer's Guide*.

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.44, 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**.

Checking the Toolchain

You can use vxprj to check the valid toolchains for a BSP:

```
% vxprj tccheck list BSP
```

For example:

```
% vxprj tccheck list sb1250a_0_mipsi64
```

```
diab sfdiab gnu sfgnu
```

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
```

The profile affects only the components use to create the project: after creation, the project does not retain any information about the profile used to create it.

For a list of profiles, 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* ...]

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.26.)

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 link a kernel application with the VxWorks kernel image, the source file must be added to the project. 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
```

For information about configuring VxWorks to start kernel applications at boot time, see the *VxWorks Kernel Programmer's Guide: Kernel.*

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.24.

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.24.

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.

Listing 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
```

3.2.5 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 Kernel Programmer's Guide: Kernel Configuration*. 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.

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.

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
```

3.2.6 Example vxprj Session

The following sample **vxprj** session creates an SMP kernel for the **pcPentium4** BSP using the diab compiler in the directory **pcPentium4_diab** (relative to the current directory). In addition to the basic operations of creating and building the project, this example demonstrates the addition of bundles, components, and build macros.

```
% vxprj create -smp pcPentium4 diab pcPentium4_diab
% cd pcPentium4_diab
% vxprj bundle add pcPentium4_diab.wpj BUNDLE_STANDALONE_SHELL
% vxprj bundle add pcPentium4_diab.wpj BUNDLE_RTP_DEVELOP
% vxprj bundle add pcPentium4_diab.wpj BUNDLE_POSIX
% vxprj component add pcPentium4_diab.wpj INCLUDE_HRFS
% vxprj component add pcPentium4_diab.wpj INCLUDE_HRFS_FORMAT
% vxprj component add pcPentium4_diab.wpj INCLUDE_RAM_DISK
% vxprj parameter setstring RAM_DISK_DEV_NAME /hrfs0
% vxprj buildmacro add ROMFS_DIR
% vxprj buildmacro set ROMFS_DIR ./romfs
% vxprj build pcPentium4_diab.wpj
```

3.2.7 Creating a Standalone Kernel Image

A kernel image is a non-relocatable object file that gets loaded and runs on the target board. For development , when you will be communicating with the target board from a host machine, the symbol table can reside on the host. For a standalone VxWorks image (**vxWorks.st**), the OS image includes the symbol table. To get a standalone kernel image, add the **INCLUDE_SHELL** and **INCLUDE_STANDALONE_SYM_TBL** components to your build:

```
% vxprj component add INCLUDE_SHELL INCLUDE_STANDALONE_SYM_TBL
```

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.

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
```

3.3.1 Building Kernel Configuration Projects with cmpScriptLib

The **cmpScriptLib** Tcl library includes routines that build and manipulate downloadable kernel modules as well as 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.31.

4
Configuring and Building Application Projects

4.1 Introduction

This chapter explains how to configure and build application 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 the IDE, **vxprj**, **cmpScriptLib**, or by calling **make** directly. The preferred command-line 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 with the IDE, or 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.
- **Shared libraries, downloadable kernel modules, and file system projects** are most easily handled from the IDE, but they can also be built by calling **make** directly.

Regardless of how build management is approached, Workbench supports two toolkits—GCC and the Wind River Compiler (diab)—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 any project that is created from the IDE (including shared libraries, downloadable kernel modules, and real-time process (RTP) applications), you can examine the generated **Makefile** and create a separate copy of the project if you want to hand-modify or script the build.

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

4.2 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 more 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*.

4.3 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 *4.4 RTP Applications and Libraries*, p.35.

4.4 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*.

4.4.1 RTP Application Makefiles

You can make a simple 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**.

You can also make a simple, but more explicit makefile that builds an RTP application in the current directory:

```
# basic build file for an RTP application

# for Windows hosts, fix slashes
WIND_HOME := $(subst \,/,$(WIND_HOME))
WIND_BASE := $(subst \,/,$(WIND_BASE))
WIND_USR := $(subst \,/,$(WIND_USR))

# project-specific macros
RTP = Hello_rtp.vxe
OBJ = Hello_rtp.o
SRC = Hello_rtp.c

# libraries, includes
LIBS = -lstlstd
INCLUDES = \
           -I$(WIND_BASE)/target/usr/h \
           -I$(WIND_BASE)/target/usr/h/wrn/coreip

# define CPU and tools
VX_CPU_FAMILY = simpentium
CPU = SIMPENTIUM
TOOL_FAMILY = diab
TOOL = diab

# compiler, linker (dcc or dplus)
CC = dcc
LL = dcc
LFLAGS =

# See Wind River compiler guide for information on target selection.
TARGET_SPEC = -tX86LH:rtp


##############################
# generic build targets, rules
$(RTP) : $(OBJ)
    $(LL) $(LFLAGS) \
          -L$(WIND_BASE)/target/usr/lib/$(VX_CPU_FAMILY)/$(CPU)/common \
          -o $@ $(OBJ) $(LIBS)

$(OBJ) : $(SRC)
    $(CC) $(TARGET_SPEC) $(INCLUDES) \
            -D_VX_CPU=_VX_$(CPU) \
            -D_VX_TOOL=$(TOOL) \
            -D_VX_TOOL_FAMILY=$(TOOL_FAMILY) \
            -o $@ -c $<

# clean up
clean :
    rm $(RTP) $(OBJ)

# See the Wind River compiler documentation for more details on
# compiler and linker options.
```

Note that the options for compiling an RTP include the following options: **-D_VX_CPU**, **-D_VX_TOOL**, and **-D_VX_TOOL_FAMILY**. You need to use these options when compiling an RTP, instead of the compiler options **-DCPU**, **-DTOOL**, and **-DTOOL_FAMILY** that you use when compiling a downloadable kernel module.

4.4.2 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**.

4.4.3 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 *4.4.5 Make Variables*, p.38), **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**.

4.4.4 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 *4.4.5 Make Variables*, p.38). 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 *4.4.5 Make Variables*, p.38), 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.

4.4.5 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*.

4.5 Downloadable Kernel Modules

The easiest way to create downloadable kernel modules is to use Workbench. However, you can build downloadable kernel modules from the command line using **make**. For example, the following simple makefile builds a downloadable kernel module in the current directory:

```
# basic build file for a DKM application

# for Windows hosts, fix slashes
WIND_HOME := $(subst \,/,$(WIND_HOME))
WIND_BASE := $(subst \,/,$(WIND_BASE))
WIND_USR := $(subst \,/,$(WIND_USR))

# project-specific macros
EXE = Hello_dkm.out
OBJ = Hello_dkm.o
SRC = Hello_dkm.c

# libraries, includes
LIBS =
INCLUDES = -I$(WIND_BASE)/target/h \
           -I$(WIND_BASE)/target/h/wrn/coreip

# define CPU and tools
CPU = SIMPENTIUM
TOOL_FAMILY = diab
TOOL = diab

# compiler, linker (dcc, dld)
CC = dcc
LL = dld

# See Wind River compiler guide for complete details on linker options
# -r retains relocation entries, so the file can be re-input to the
linker
LFLAGS = $(TARGET_SPEC) -r

# See Wind River compiler guide for information on target selection.
TARGET_SPEC = -tX86LH:vxworks66


##############################
# generic build targets, rules

# executable constructed using linker from object files
$(EXE) : $(OBJ)
    $(LL) $(LFLAGS) -o $@ $(OBJ) $(LIBS)

# objects compiled from source
$(OBJ) : $(SRC)
    $(CC) $(TARGET_SPEC) $(INCLUDES) \
            -DCPU=$(CPU) \
                -DTOOL=$(TOOL) \
                -DTOOL_FAMILY=$(TOOL_FAMILY) \
                -o $@ -c $<

# clean up
clean :
    rm $(EXE) $(OBJ)

# See the Wind River compiler documentation for more details on
# compiler and linker options.
```

Note that the compiler options for a downloadable kernel module include **-DCPU**, **-DTOOL**, and **-DTOOL_FAMILY**.

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:

   ▪ Run wrenv: **wrenv -p vxworks-6.***x*.
   ▪ 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 the *Wind River Workbench Host Shell User's Guide.*

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

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

For detailed information, see the *Wind River Workbench Host Shell User's Guide*.

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.18 and *3.2.5 Building Kernel Configuration Projects with vxprj*, p.28.

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 *4.4 RTP Applications and Libraries*, p.35.

To run the simulator, type the following:

```
wrenv -p vxworks-6.x
vxsim [-f pathToVxWorksImage|filename.vxe] [otherOptions]
```

From the kernel shell on the simulator, you can run downloadable kernel modules (DKMs) or real time process applications (RTPs).

For example, to download and run a DKM on the target simulator:

```
-> ld < host:dkm_filepath.out      # load the DKM
-> main                             # run main() routine
```

To spawn an RTP from a binary file on the host machine:

```
-> rtpSp "host:rtp_filepath.vxe"      # spawn an RTP
```

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

A
Configuring and Building VxWorks Using config.h

A.1 Introduction

Wind River recommends using either Workbench or the vxprj command-line facility to configure and build VxWorks. The legacy method using *bspDir***/config.h** and *bspDir***/make** has been deprecated for most purposes, and you cannot use it for multiprocessor (SMP and AMP) development.

However, you must still use the **config.h** method for the following:

- Some middleware products (see the VxWorks 6.6 Release Notes for further information).
- Boot loaders, when the BSP does not support the **PROFILE_BOOTAPP** configuration profile.

You can also use the **config.h** method for uniprocessor BSP development prior to the CDF (configuration description file) development to support vxprj and Workbench builds.

A.2 Configuring an Image Using config.h

In the absence of CDFs (configuration description files), you can configure a boot loader or VxWorks image using the **config.h** file. For example, you can select a different network driver, or networking components can be removed if the system does not require networking to boot or to operate.

In general, you should only configure a build manually if it is not already set up to use **vxprj** or Workbench.

To add or remove components from a boot loader or VxWorks image, you must use two BSP configuration files: **configAll.h** (for reference purposes only) and **config.h**:

```
installDir/vxworks-6.6/target/config/all/configAll.h
installDir/vxworks-6.6/target/config/bspName/config.h
```

The **configAll.h** file provides the default VxWorks configuration for all BSPs. The **INCLUDED SOFTWARE FACILITIES** section of this header file lists all components that are included in the default configuration. The **EXCLUDED FACILITIES** section lists those that are not. To add or remove components from a kernel configuration, examine the **configAll.h** file to determine the default, then change the default for a specific BSP by editing the **config.h** file for that BSP, defining or undefining the appropriate components.

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

- Default boot parameter string for boot ROMs.
- Interrupt vectors for system clock and parity errors.
- Device controller I/O addresses, interrupt vectors, and interrupt levels.
- Shared memory network parameters.
- Miscellaneous memory addresses and constants.

The **config.h** file overrides any setting in the **configAll.h** file. You should never edit the **configAll.h** file.

CAUTION: Do not edit the **configAll.h** file. Any changes that you make will affect all BSPs. You should only edit the **config.h** files in the individual BSP directories.

A.3 **Configuration Examples**

The following examples draw from **configAll.h** and from specific BSPs.

A.3.1 **Removing Network Support**

The **INCLUDED SOFTWARE FACILITIES** section of **configAll.h** has the following entry for the core networking component:

```
#define INCLUDE_NETWORK         /* network subsystem code *
```

If you are creating a boot loader for a **wrSbc8560** board that does not require networking, you would add the following line to the **config.h** file in *installDir*/**vxworks-6.6/target/config/wrSbc8560**:

```
#undef INCLUDE_NETWORK
```

A.3.2 **Adding RTP Support**

RTP (real-time process) support is not included by default in BSP builds. Even though the line

```
#define INCLUDE_RTP
```

appears in **configAll.h**, it is in the **EXCLUDED FACILITIES** section (wrapped in an **#if FALSE** block).

If you want to include RTP support, you need to define the appropriate macros in config.h. For example (from the **simpc** BSP):

```
#define INCLUDE_RTP                          /* Real Time Process */
#define INCLUDE_RTP_APPL_INIT_BOOTLINE       /* RTP bootline Facility */

#ifdef INCLUDE_RTP
#define INCLUDE_SC_POSIX        /* POSIX system calls */
#endif                          /* INCLUDE_RTP */
```

A.4 Using make

The recommended way of building kernel configuration projects is to use **vxprj** or the IDE. You can build VxWorks projects of other types 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 CPU=PPC32 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.

A.4.1 Makefile Details

Each BSP has a makefile for building VxWorks. This makefile, called **Makefile**, is abbreviated to declare only the basic information needed to build VxWorks with the BSP. The makefile includes other files to provide target and VxWorks specific rules and dependencies. In particular, a file of the form **depend.***bspname* is included. The first time that **make** is run in the BSP directory, it creates the **depend.***bspname* file.

The **Makefile** in the BSP directory is used only when building from the traditional command line. It is not used when building projects from the project tool. Each build option for a project has its own makefile that the tool uses to build the project modules.

There are a number of different VxWorks image targets that can be created using the **Makefile**, including **vxWorks** (a downloadable image that executes from RAM), and **vxWorks.st** (a standalone image that includes a symbol table). In addition, you can build VxWorks images that can be stored in ROM as well as images that can be executed from ROM. In addition to VxWorks images, you can build bootrom images that are VxWorks images configured with a boot loader as the application. For details on VxWorks image types, see the *Boot Sequence* section of the *Wind River VxWorks BSP Developer's Guide: Overview of a BSP*.

When projects are created from a BSP, the BSP makefile is scanned once and the make parameters are captured into the project. Any changes made to the BSP makefile after a project has been created do not affect that project. Only projects built from the BSP after the change is made are affected.

Customizing the VxWorks Makefile

The BSP makefile provides several mechanisms for configuring the VxWorks build. Although VxWorks configuration is more commonly controlled at compile-time by macros in **configAll.h** and *bspname*/**config.h**.

Most of the makefile macros fall into two categories: macros intended for use by the BSP developer, and macros intended for use by the end user. The needs of these two audiences differ considerably. Maintaining two separate compile-time macro sets lets the **make** separate the BSP-specific requirements from user-specific requirements.

Macros containing **EXTRA** in their name are intended for use by the BSP developer to specify additional object modules that must be compiled and linked with all VxWorks images.

Macros containing **ADDED** in their name are intended for use by the end-user on the **make** command line. This allows for easy compile time options to be specified by the user, without having to repeat any BSP-specific options in the same macro declaration.

Commonly Used Makefile Macros

Of the many makefile macros, this document discusses only the most commonly used.

MACH_EXTRA

You can add an object module to VxWorks by adding its name to the skeletal makefile. To include **fooLib.o**, for example, add it to the **MACH_EXTRA** definition in **Makefile**. This macro causes the linker to link it into the output object.

Finally, regenerate VxWorks with **make**. The module will now be included in all future VxWorks builds. If necessary, the module will be made from **fooLib.c** or **fooLib.s** using the **.c.o** or **.s.o** makefile rule.

MACH_EXTRA can be used for drivers that are not included in the VxWorks driver library. BSPs do not usually include source code for device drivers; thus, when preparing your system for distribution, omit the driver source file and change the object file's name from **.o** to **.obj** (update the makefiles, too). Now the end user can build VxWorks without the driver source, and **rm *.o** will not inadvertently remove the driver's object file.

LIB_EXTRA

The **LIB_EXTRA** makefile variable makes it possible to include library archives in the VxWorks build without altering the standard VxWorks archive or the driver library archive. Define **LIB_EXTRA** in **Makefile** to indicate the location of the extra libraries.

The libraries specified by **LIB_EXTRA** are provided to the link editor when building any VxWorks or boot ROM images. This is useful for third-party developers who want to supply end users with network or SCSI drivers, or other modules in object form, and find that the **MACH_EXTRA** mechanism described earlier in this chapter does not suit their needs.

The extra libraries are searched first, before Wind River libraries, and any references to VxWorks symbols are resolved properly.

EXTRA_INCLUDE

The makefile variable **EXTRA_INCLUDE** is available for specifying additional header directory locations. This is useful when the user or BSP provider has a separate directory of header files to be used in addition to the normal directory locations.

```
EXTRA_INCLUDE = -I../myHdrs
```

The normal order of directory searching for **#include** directives is:

```
$(INCLUDE_CC) (reserved for compiler specific uses)
$(EXTRA_INCLUDE)
.
$(CONFIG_ALL)
```

```
$(TGT_DIR)/h
$(TGT_DIR)/src/config
$(TGT_DIR)/src/drv
```

EXTRA_DEFINE

The makefile variable **EXTRA_DEFINE** is available for specifying compile time macros required for building a specific BSP. In the following example the macro **BRD_TYPE** is given the value **MB934**. This macro is defined on the command line for all compiler operations.

```
EXTRA_DEFINE = -DBRD_TYPE=MB934
```

By default a minimum set of macro names are defined on the compiler command line. This is primarily used to pass the same memory addresses used in both the compiling and linking operations.

These default macro definitions include:

```
-DCPU=$(CPU)
```

ADDED_CFLAGS

Sometimes it is inconvenient to modify **config.h** to control VxWorks configuration. **ADDED_CFLAGS** is useful for defining macros without modifying any source code.

Consider the hypothetical Acme XYZ-75 BSP that supports two hardware configurations. The XYZ-75 has a daughter board interface, and in this interface either a Galaxy-A or a Galaxy-B module is installed. The drivers for the modules are found in the directory **src/drv/multi**.

The macro **GALAXY_C_FILE** determines which driver to include at compile-time. The file named by **GALAXY_C_FILE** is **#included** by **sysLib.c**.

The default configuration (Galaxy-A module installed) is established in **config.h**:

```
#ifndef GALAXY_C_FILE
#define GALAXY_C_FILE "multi/acmeGalaxyA.c"
#endif /* GALAXY_C_FILE */
```

When **make** is called normally, VxWorks supports the XYZ-75 with the Galaxy-A module installed. To override the default and build VxWorks for the XYZ-75/Galaxy-B configuration, do the following:

```
% make ADDED_CFLAGS='-DGALAXY_C_FILE=\"multi\/acmeGalaxy02.c\"' vxWorks
```

For ease of use, you can encapsulate the lengthy command line within a shell script or independent makefile.

To ensure that a module is incorporated in **vxWorks**, remove the module's object file and **vxWorks** before running **make**.

ADDED_MODULES

The **ADDED_MODULES** makefile variable makes it possible to add modules to **vxWorks** without modifying any source code.

While **MACH_EXTRA** requires the makefile to be modified, **ADDED_MODULES** allows one or more extra modules to be specified on the **make** command line. For example, to build **vxWorks** with the BSP VTS support library included, copy **pkLib.c** to the target directory and enter the following:

```
% make ADDED_MODULES=pkLib.o vxWorks
```

One disadvantage of using **ADDED_MODULES** is that makefile dependencies are not generated for the module(s). In the above example, if **pkLib.c**, **pkLib.o**, and VxWorks already exist, you must remove **pkLib.o** and **vxWorks** before running **make** to force the latest **pkLib.c** to be incorporated into **vxWorks**.

CONFIG_ALL

Under extreme circumstances, the files in the **config/all** directory might not be flexible enough to support a complex BSP. In this case, copy all the **config/all** files to the BSP directory (**config/***bspname*) and edit the files as necessary. Then redefine the **CONFIG_ALL** makefile variable in **Makefile** to direct the build to the altered files. To do this, define **CONFIG_ALL** to equal the absolute path to the BSP's **config/***bspname* directory as shown in the following example:

```
CONFIG_ALL     =     $(TGT_DIR)/config/bspname/
```

The procedure described above works well if you must modify all or nearly all the files in **config/all**. However, if you know that only one or two files from **config/all** need modification, you can copy just those files to the BSP's **config/***bspname* directory. Then, instead of changing the **CONFIG_ALL** makefile macro, change one or more of the following (which ever are appropriate).

USRCONFIG
The path to an alternate **config/all/usrConfig.c** file.

BOOTCONFIG
The path to an alternate **config/all/bootConfig.c** file.

BOOTINIT
The path to an alternate **config/all/bootInit.c** file.

DATASEGPAD
The path to an alternate **config/all/dataSegPad.s** file.

CONFIG_ALL_H
The path to an alternate **config/all/configAll.h** file.

TGT_DIR
The path to the target directory tree, normally **$(WIND_BASE)/target**.

COMPRESS
The path to the host's compression program. This is the program that compresses an executable image. The binary image is input through **stdin**, and the output is placed on the **stdout** device. This macro can contain command-line flags for the program if necessary.

BINHEX
The path to the host's object-format-to-hex program. This program is called using **HEX_FLAGS** as command line flags. See **target/h/make/rules.bsp** for actual calling sequence.

HEX_FLAGS
Command line flags for the **$(BINHEX)** program.

BOOT_EXTRA
Additional modules to be linked with compressed ROM images. These modules are not linked with uncompressed or ROM-resident images, just compressed images.

EXTRA_DOC_FLAGS
Additional preprocessor flags for making man pages. The default documentation flags are **-DDOC -DINCLUDE_SCSI**. If **EXTRA_DOC_FLAGS** is defined, these flags are passed to the man page generation routines in addition to the default flags.

A.4.2 **Building Custom Boot Loaders**

Once a boot loader is configured, you can build the custom boot loader by running **make** *target* in the BSP directory. There are a variety of boot loader image targets, including the compressed **bootrom** that executes from RAM, the uncompressed **bootrom_uncmp** (which also executes from RAM), and the **bootrom_res** (which executes from ROM). For details, see the *Boot Loader Image Types* section of the *VxWorks Kernel Programmer's Guide: Boot Loader.*

Index