

WIND RIVER

Wind River® Workbench

HOST SHELL USER'S GUIDE

3.0

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, the Wind River logo, Tornado, and VxWorks are registered trademarks 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>

Contents

1	Overview	1
1.1	Introduction	1
1.1.1	Target Operating System Configuration	3
1.1.2	Reference Pages	3

PART 1: VXWORKS 6.X, LINUX, AND STANDALONE TARGETS

2	Using the Host Shell	7
2.1	Introduction	8
2.2	Starting the Host Shell	8
2.2.1	Starting the Host Shell from the Command Prompt	8
	Setting Your Environment	8
	Starting the Target Server	8
	Starting the Host Shell	9
	Starting the Host Shell from an Existing Connection	9
	Host Shell Startup Options	9
2.2.2	Starting the Host Shell from Workbench	11
2.2.3	Starting a Standalone Host Shell with an OCD Connection	11
	Starting the Host Shell	11
	Connecting to a Target	12

2.3	Switching Interpreters	13
2.3.1	Evaluating Statements in Different Modes	13
2.4	Setting Shell Environment Variables	14
2.5	Path Mapping	20
	The ROOT_PATH_MAPPING Variable	20
	The VXE_PATH Variable	21
2.6	Running the Host Shell in Batch Mode	22
2.7	Host Shell Logging	22
2.8	Host Shell Scripting	23
2.8.1	Single-Stepping Scripts	24
2.8.2	Stepping in the Tcl Interpreter	24
2.9	Host Shell Features	25
2.9.1	I/O Redirection	25
2.9.2	Symbol Matching	26
2.9.3	Directory and File Listing	26
2.9.4	Target Symbol and Path Completion	27
2.9.5	Synopsis Printing	27
2.9.6	Data Conversion	28
2.9.7	Data Calculation	28
	Calculations with Variables	29
2.10	Stopping the Host Shell	29
2.11	Host Shell Architecture	29
2.11.1	Layers of Interpretation	31
3	Using the C Interpreter with VxWorks 6.x	33
3.1	Introduction	35

3.2	C Interpreter Limitations	35
3.3	Host and Kernel Shell Differences	36
3.3.1	Function Calls in the Kernel	37
3.4	Running Target Routines From the Shell	38
3.4.1	Invocations of VxWorks Subroutines	38
3.4.2	Invocations of Application Subroutines	38
3.5	Rebooting from the Shell	38
3.6	Using the Host Shell for System-Mode Debugging	39
3.7	Interrupting a Shell Command	43
3.8	Task References	44
	The “Current” Task and Address	44
3.9	Data Types	45
3.10	Expressions	47
3.10.1	Literals	47
3.10.2	Variable References	47
3.10.3	Operators	48
3.10.4	Function Calls	48
3.10.5	Arguments to Commands	49
3.11	Assignments	50
3.11.1	Typing and Assignment	50
3.11.2	Automatic Creation of New Variables	51
3.12	Comments	51
3.13	Strings	52
3.13.1	Strings and Pathnames	52
3.14	Ambiguity of Arrays and Pointers	53

3.15	Pointer Arithmetic	54
3.16	Redirection in the C Interpreter	54
3.16.1	Ambiguity Between Redirection and C Operators	55
3.16.2	The Nature of Redirection	55
3.16.3	Scripts: Redirecting Shell I/O	56
	C Interpreter Startup Scripts	57
3.17	C++ Interpretation	57
3.17.1	Overloaded Function Names	58
3.17.2	Automatic Name Demangling	59
3.18	C Interpreter Primitives	60
3.18.1	Managing Tasks	60
3.18.2	Task Information	61
3.18.3	System Information	63
3.18.4	System Modification and Debugging	66
3.18.5	C++ Development	69
3.18.6	Object Display	69
3.18.7	Network Status Display	71
3.19	Resolving Name Conflicts Between Host and Target	72
3.20	Examples	73
4	Using the Command Interpreter with VxWorks 6.x	75
4.1	Introduction	76
4.2	General Commands	77
4.3	Displaying Target Agent Information	78
4.4	Working with Memory	79
4.5	Displaying Object Information	79

4.6	Working with Symbols	79
4.6.1	Accessing a Symbol's Contents and Address	80
4.6.2	Symbol Value Access	80
4.6.3	Symbol Address Access	81
4.6.4	Special Consideration of Text Symbols	81
4.7	Displaying, Controlling, and Stepping Through Tasks	82
4.8	Setting Shell Context Information	83
4.9	Displaying System Status	83
4.10	Using and Modifying Aliases	84
4.11	Launching RTPs	86
4.11.1	Redirecting Output to the Host Shell	86
4.11.2	Monitoring and Debugging RTPs	87
4.11.3	Setting Breakpoints	88
4.12	Event Scripting Commands	89
	handler add	89
	handler show	90
	handler remove	91
	handler enable	91
4.12.1	Limitations	91
4.12.2	Event Scripting Example	93
4.13	General Examples	94
5	Using the GDB Interpreter	97
5.1	Introduction	98
5.2	General GDB Commands	98
5.3	Working with Breakpoints	99

5.4	Specifying Files to Debug	100
5.5	Running and Stepping Through a File	100
5.6	Displaying Disassembly and Memory Information	101
5.7	Examining Stack Traces and Frames	102
5.8	Displaying Information and Expressions	102
5.8.1	info	103
5.8.2	print	103
5.9	Displaying and Setting Variables	104
5.10	Working with Signals	105
5.10.1	handle	105
5.10.2	info handle	106
5.10.3	signal	107
5.10.4	send signal	107
5.11	Event Scripting	108
5.11.1	Event Scripting Commands	108
	display	108
	undisplay	108
	info display	109
	enable display	109
	disable display	109
	commands	110
	info commands	110
	enable commands	110
	disable commands	111
5.11.2	Event Scripting Example	111
5.12	Wind River On-Chip Debugging GDB Commands	112
5.12.1	target ocd	112
5.12.2	wrsdeftarget	113
5.12.3	wrsregquery	115

5.12.4	Reset and Download Commands	115
6	Using the Tcl Interpreter	117
6.1	Introduction	117
6.2	Controlling the Target	118
6.3	Accessing the WTX Tcl API	120
6.4	Calling Target Routines	120
6.5	Passing Values to Target Routines	121
6.6	Calling Under C Control	121
6.6.1	Potential Problems	122
6.7	Shell Initialization	122
6.7.1	Shell Initialization File	123
6.8	Tcl Scripting	123
6.8.1	Event Scripting	124
	API Description	125
7	Executing an OCD Reset and Download	127
7.1	Introduction	127
7.2	Set Target Registers	128
7.3	Play Back Firmware Commands	129
7.4	Reset One or More Cores	130
7.5	Download Executables and Data and Program Flash	130
	Download Executables and Data	131
	Erase Flash Memory (Optional)	131
	Program Flash Memory (Optional)	131

7.6	Run the Target	132
7.7	Set a Hardware Breakpoint	132
7.8	Configure Target Memory Map	132
7.9	Pass Through Command to Firmware	134
7.10	Upload from Target Memory	134

PART II: VXWORKS 653 TARGETS

8	Overview for VxWorks 653	137
8.1	Introduction	137
8.2	Starting the Host Shell	138
8.2.1	Starting the Host Shell from the Command Prompt	138
	Setting Your Environment	138
	Starting the Target Server	138
	Starting the Shell	138
	Host Shell Startup Options	139
8.2.2	Starting the Host Shell from Workbench	140
8.3	Switching Interpreters	140
8.3.1	Evaluating Statements in Different Modes	140
8.4	Setting Shell Environment Variables	141
8.5	Path Mapping	144
	The ROOT_PATH_MAPPING Variable	144
8.6	Host Shell Features	145
8.6.1	Symbol Matching	145
8.6.2	Directory and File Listing	145
8.6.3	Target Symbol and Path Completion	146
8.6.4	Synopsis Printing	146

8.6.5	Data Conversion	147
8.6.6	Data Calculation	147
	Calculations with Variables	147
8.7	Stopping the Host Shell	148
8.8	Host Shell Architecture	148
8.8.1	Layers of Interpretation	151
9	Using the Host Shell with VxWorks 653	153
9.1	Introduction	154
9.2	Domain Selection and Identification	154
9.3	Running Target Routines From the Shell	156
9.3.1	Invocations of VxWorks 653 Subroutines	156
9.4	Function Calls from User Domains	156
9.5	Rebooting from the Host Shell	157
9.6	Task-Mode Debugging	158
9.6.1	Task Breakpoints	158
9.6.2	Protection Domain Breakpoints	159
9.7	Stack Tracing	161
9.8	Disassembler	161
9.9	Using the Host Shell for System-Mode Debugging	162
9.10	Interrupting a Shell Command	165
9.11	Working With Shared Library and Data Domains	167
9.12	Loading From the Shell	167
9.12.1	Incremental Loading	167

9.12.2	Dynamic Linking	167
9.12.3	Object Module Load Path	168
9.12.4	Loader Defaults	169
10	Using the C Interpreter with VxWorks 653	171
10.1	Introduction	172
10.2	Host and Target Shell Differences	172
10.2.1	Protection Domain Breakpoints	173
10.2.2	Function Calls in the Kernel Domain	174
10.3	Task References	174
	The "Current" Task and Address	175
10.4	Data Types	175
10.5	Expressions	177
10.5.1	Literals	177
10.5.2	Variable References	178
10.5.3	Operators	178
10.5.4	Function Calls	179
10.5.5	Arguments to Commands	180
10.6	Assignments	181
10.6.1	Typing and Assignment	181
10.6.2	Automatic Creation of New Variables	181
10.7	Comments	182
10.8	Strings	182
10.8.1	Strings and Pathnames	183
10.9	Ambiguity of Arrays and Pointers	183
10.10	Pointer Arithmetic	184

10.11	C Interpreter Limitations	185
10.12	Redirection in the C Interpreter	186
10.12.1	Ambiguity Between Redirection and C Operators	186
10.12.2	The Nature of Redirection	186
10.12.3	Scripts: Redirecting Shell I/O	187
	C Interpreter Startup Scripts	188
10.13	C++ Interpretation	189
10.13.1	Overloaded Function Names	189
10.13.2	Automatic Name Demangling	191
10.14	C Interpreter Primitives	192
10.14.1	Managing Tasks	192
10.14.2	Task Information	193
10.14.3	Displaying System Information	195
10.14.4	Modifying and Debugging the Target	199
10.14.5	Protection Domains	202
10.14.6	C++ Development	202
10.14.7	Object Display	203
10.14.8	Network Status Display	206
10.15	Resolving Name Conflicts Between Host and Target	207
10.16	Examples	207
11	Using the Tcl Interpreter with VxWorks 653	209
11.1	Introduction	209
11.2	Controlling the Target	210
11.3	Accessing the WTX Tcl API	211
11.4	Calling Target Routines	212

11.5	Passing Values to Target Routines	212
11.6	Calling Under C Control	213
11.7	Shell Initialization	213
11.7.1	Shell Initialization File	214

PART III: APPENDICES

A	Using the Host Shell Line Editor	217
A.1	Introduction	217
A.2	vi-Style Editing	218
A.2.1	Switching Modes and Controlling the Editor	218
A.2.2	Moving and Searching in the Editor	219
A.2.3	Inserting and Changing Text	220
A.2.4	Deleting Text	220
A.2.5	Put and Undo Commands	221
A.3	emacs-Style Editing	221
A.3.1	Moving the Cursor	221
A.3.2	Deleting and Recalling Text	222
A.3.3	Special Commands	222
A.4	Command Matching	223
A.4.1	Directory and File Matching	223
A.4.2	Command and Path Completion	223
B	Single Step Compatibility	225
B.1	Introduction	225
B.2	Scripting	226

B.3	SingleStep Command Equivalents	226
B.4	SingleStep read Command Compatibility	230
B.5	SingleStep write Command Compatibility	232
B.6	SingleStep Variable Compatibility	233

1

Overview

1.1 Introduction

The Wind River Workbench host shell is a command-line shell for the Wind River Workbench debugger. It is intended for scripting and for lightweight use of the debugger when the Eclipse-based Workbench graphical user interface (GUI) is not needed or not wanted. You can use it independently of the GUI to debug an application at source or symbol level.

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 VxWorks 6.x or VxWorks 653 kernel modules, VxWorks 6.x real-time processes (RTPs), and Linux executables. You can run application modules interactively by calling their entry points. You can also run the host shell in non interactive mode, as an engine for automated runs and tests.

The host shell behaves differently depending on what operating system image you have loaded on your target. Modes and features that are available for one OS may not be available for another.

This document divides target operating systems into two groups:

Group 1

- VxWorks 6.x
- Linux
- Standalone (no operating system)

Group 2

- VxWorks 653

The behavior of the shell differs so widely between these two groups that they need to be covered separately. In this document, Part I describes the host shell for operating systems in Group 1. Part II describes the host shell for VxWorks 653.

Modes in the host shell differ as follows:

Table 1-1 Differences Between Target Operating Systems

Target OS	Command Interpreter Mode	C interpreter Mode	GDB Interpreter Mode	Tcl Interpreter Mode	Default Mode
VxWorks 6.x	Yes	Yes	Yes	Yes	C
VxWorks 653	No	Yes	No	Yes	C
Linux	No	No	Yes	Yes	GDB
Standalone (no operating system)	No	No	Yes	Yes	GDB

- *Command Interpreter Mode* is a UNIX-style command interpreter for debugging and monitoring a VxWorks 6.x system, including real-time processes (RTPs.)
- *C Interpreter Mode* executes C-language expressions and allows prototyping and debugging in kernel space. The C interpreter is available to VxWorks and VxWorks 653 targets, but behaves differently for each OS. The C interpreter is described in Part I in [3. Using the C Interpreter with VxWorks 6.x](#) (for VxWorks 6.x targets) and in Part II in [10. Using the C Interpreter with VxWorks 653](#) (for VxWorks 653 targets.)

Note that C interpreter routine calls return 32-bit values only.

- *Tcl Interpreter Mode* allows you to access the Wind River Tool Exchange (WTX) Tcl API; it also allows scripting.
- *GDB Interpreter Mode* allows you to debug a target using GNU Debugger (GDB) commands.

Host shell operation involves three components:

- The *host shell* itself. The shell is where you directly exercise control; it receives your commands and executes them locally on the host, dispatching requests to the target server for any action involving the symbol table or target-resident programs or data.

- A *target server*, which manages the symbol table and handles all communications with the remote target, dispatching function calls and sending their results back as needed.
- 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 target-resident version of the shell is also available for VxWorks 6.x and VxWorks 653 targets. For VxWorks 6.x this is called the *kernel shell*. For VxWorks 653 it is called the *target shell*. See the *VxWorks Kernel Programmer's Guide: Target Tools* or the *Wind River Workbench User's Guide, VxWorks 653 Version: Tools*.

1.1.1 Target Operating System Configuration

Depending on your target operating system, you may need to configure your OS to use the host shell:

- VxWorks 6.x targets need to have the Wind River DeBug (WDB) agent running in order to use the host shell. Make sure the component `INCLUDE_WDB` is included when you configure your VxWorks image.
- Wind River Linux targets need to have the user mode agent running in order to use the host shell. Once your kernel is booted, call **usermode-agent** to start the user mode agent.

1.1.2 Reference Pages

For more information, see the host shell reference pages: **hostShell**, **cMode**, **cmdMode**, **gdbMode**, and **rtpCmdMode**. You can access these pages by opening Wind River Workbench and selecting **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference**.

PART 1

VxWorks 6.x, Linux, and Standalone Targets

2	Using the Host Shell	7
3	Using the C Interpreter with VxWorks 6.x	33
4	Using the Command Interpreter with VxWorks 6.x 75	
5	Using the GDB Interpreter	97
6	Using the Tcl Interpreter	117
7	Executing an OCD Reset and Download	127

2

Using the Host Shell

- 2.1 Introduction 8
- 2.2 Starting the Host Shell 8
- 2.3 Switching Interpreters 13
- 2.4 Setting Shell Environment Variables 14
- 2.5 Path Mapping 20
- 2.6 Running the Host Shell in Batch Mode 22
- 2.7 Host Shell Logging 22
- 2.8 Host Shell Scripting 23
- 2.9 Host Shell Features 25
- 2.10 Stopping the Host Shell 29
- 2.11 Host Shell Architecture 29

2.1 Introduction

This chapter describes features of the host shell common to VxWorks 6.x, Linux, and standalone (no operating system) targets.

2.2 Starting the Host Shell

You can start the host shell from a command prompt or from within the Workbench GUI.

2.2.1 Starting the Host Shell from the Command Prompt

Setting Your Environment

Before launching the host shell, you must use the command **wrenv** to set up your environment. If you do not set your environment, the prompt returns the following error:

```
WIND_FOUNDATION_PATH must be set to start the Host Shell
```

To set your environment, enter the following command from your installation directory:

```
%wrenv -p target_OS_version
```

For example, if you are using VxWorks 6.x, enter

```
%wrenv -p vxworks-6.x
```

If you are using Wind River Linux 1.x, enter

```
%wrenv -p wrlinux-1.x
```

Starting the Target Server

To start a target server, use the **tgtsvr** command. For example, to start a target server called **myTgtsvr** on a target with the IP address 123.456.78.90, enter the following command from your installation directory:


```
% tgtsvr 123.456.78.90 -n myTgtsvr
```

To see all available options for the **tgtsvr** command, enter **tgtsvr -h**.

Note that you must set up your environment with the **wrenv** command before using the **tgtsvr** command.

Starting the Host Shell

Once you have attached a target server to the target, start the host shell using the **hostShell** command.

To start the host shell, type the following:

```
%hostShell [options] target_server
```

For former users of Tornado, the **windsh** command is still supported:

```
%windsh [options] target_server
```

Starting the Host Shell from an Existing Connection

If you are using an OCD connection, or if you have already established a target connection using Workbench, then you can start the host shell with the **hostShell** command, indicating the name of the target connection to use and the backend server to use.

To start the host shell, type the following:

```
% hostShell [options] -dt target_connection -ds backend_server_session
```

If you do not know the names of your target connection and backend server session, type **hostShell** with no arguments and follow the options presented.

Host Shell Startup Options

[Table 2-1](#) summarizes startup options. For example, to connect to a running simulator, type the following:

```
%hostShell 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 [2.4 Setting Shell Environment Variables](#), p.14).

Table 2-1 **Host Shell Startup Options**

Option	Description
-N, -noconnection	Specifies that the host shell will not connect to the backend server on startup. This allows a Tcl script to control the host shell.
-n, -noinit	Do not read home Tcl initialization file.
-T, -Tclmode	Start in Tcl mode.
-m[ode]	Indicates mode to start in: C (C), 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 in milliseconds; the default is 200.
-e, -execute	Executes Tcl expression after initialization.
-c, -command	Executes expression and exits shell (batch mode).
-r, -root mappings	Root pathname mappings.
-ds[backend_server_session]	Debugger Server session to use.
-dp[backend_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	Backend target definition name.

2.2.2 Starting the Host Shell from Workbench

If you have established a target connection, you can start the host shell from the **Remote Systems** view in Workbench. For creating target connections, see the *Wind River Workbench User's Guide: Connecting to Targets*.

In the **Remote Systems** view, right-click on your target connection name and select **Target Tools > Host Shell**. The **Host Shell Properties** dialog appears. You can specify startup options from [Table 2-1](#) in this dialog, or leave them at their defaults. Click **OK** to start the host shell.

2.2.3 Starting a Standalone Host Shell with an OCD Connection

To start a standalone host shell independent of Workbench using an OCD connection, use the following steps.

Starting the Host Shell

Windows Hosts

1. Select **Start > All Programs > Wind River > Workbench version > Registry** to start the Wind River Registry manually.
2. Set up the Wind River environment variables.

Open a DOS window and navigate to your Workbench installation directory. Enter the command

```
wrenv -p workbench-version
```

3. Type **SET** to verify that the paths are set up correctly. The path **WIND_FOUNDATION_PATH** should be set to *installDir/workbench-3.x/foundation/version*.
4. Start the host shell with the **-N** flag so that it just brings up the prompt with no target connection:

```
hostShell -N
```

This opens the host shell showing the **(gdb)** prompt, which you can use to define the needed target connection.

Linux Hosts

1. Navigate to your Workbench install directory and enter the command

```
workbench-3.x/foundation/version/x86-linux2/bin/wtxregd start
```

This starts the Wind River Registry.

2. Set up the Wind River environment variables.

From your Workbench installation directory, login as root and enter the command

```
./wrenv.sh -p workbench-version
```

3. Exit root.
4. Type SET to verify that the paths are set up correctly. The path **WIND_FOUNDATION_PATH** should be set to **/home/username/installDir/workbench-3.x/foundation/version**.
5. Start the host shell with the **-N** flag so that it just brings up the prompt with no target connection:

```
workbench-3.x/foundation/version/x86-linux2/bin/hostShell -N
```

This opens the host shell showing the **(gdb)** prompt, which you can use to define the needed target connection.

Connecting to a Target

1. In the host shell, specify the target connection using the **wrsdeftarget** command.

For example, to connect to a PowerPC 8260 target using a Wind River ICE SX that has an IP address of 123.456.789.0, enter the following:

```
wrsdeftarget WRICE_8260 --core MPC8260 --cpuplugin 82xxcpuplugin  
DEVICE='Wind River ICE' STYLE=ETHERNET ADDR=123.456.789.0
```

In the above example, **WRICE_8260** is an arbitrary name. You can use any name for the connection; however, your connection name must not contain spaces. **--core** is set to **MPC8260**, because you are connecting to an 8260 target. **--cpuplugin** is set to **82xxcpuplugin**, which is the plugin common to all PowerPC 82xx processors.

For a full description of the **wrsdeftarget** command, see [5.12.2 wrsdeftarget](#), p.113.

2. Connect the host shell to the newly defined target connection using the **target ocd** command.

Use the syntax **target ocd** *target_id*. For example, to connect to the defined target connection WRICE_8260, enter

```
target ocd WRICE_8260
```

The host shell attaches to this target connection and opens a backend server for communication. This backend server manifests as a second terminal window. You can minimize this window, but do not close it; closing it severs the host shell's connection with the target.

2.3 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 **->**.
- **tcl** to switch to the Tcl interpreter. The prompt changes to **tcl>**.
- **gdb** to switch to the GDB interpreter. The prompt changes to **gdb>**.

2.3.1 Evaluating Statements in Different Modes

You can use the above commands to evaluate a statement native to a different interpreter for the one you are using.

To evaluate a statement native to another interpreter, use the routine **shEval** followed by the special character for the interpreter you want to invoke.

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

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

If you are using a command that is valid in more than one interpreter, another step is necessary. For example, the **set** command is valid in both the GDB interpreter and the Tcl interpreter, so the syntax

```
tcl> shEval gdb set $pc= address
```

will return an error:

```
can't read "pc": no such variable
```

To avoid this problem, precede the **set** command's argument with a backslash:

```
tcl> shEval gdb set \ $pc = 0x14200
```

2.4 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 2-2](#) 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 the Tcl special character, **tcl** (**tcl shConfig *variable option***).

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

```
-> shEval tcl 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 2-2](#). Not all of the listed environment variables are valid for all targets. For example, all variables dealing with real-time processes (RTPs) are specific to VxWorks 6.x.

Table 2-2 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 2-2 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 <i>pathname</i>	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 <i>value</i>	Indicates how host and target paths should be mapped to the host file system on which the backend 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 <i>/folk/user</i> is searched; no translation to another path is performed).
LINE_LENGTH <i>value</i>	Indicates the maximum number of characters permitted in one line of the host shell's window.
STRING_FREE [manual automatic]	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 <code>strFree()</code> .
SEARCH_ALL_SYMBOLS [ON OFF]	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]	Indicates the host shell's current interpreter mode and permits the user to switch from one mode to another.

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

Variable	Result
SH_GET_TASK_IO [ON OFF]	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 [ON OFF]	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 [ON OFF]	Sets the load mode. The default ON causes modules to be transferred to the target server. This means that any module the host shell can see can be loaded. If the variable is OFF , the target server must be able to see the module to load it.
LD_PATH <i>pathname</i>	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 [ON OFF]	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.

Table 2-2 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 <i>pathname</i>	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 <i>value</i>	Indicates how host and target paths should be mapped to the host file system on which the backend 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 <i>/folk/user</i> is searched; no translation to another path is performed).
LINE_LENGTH <i>value</i>	Indicates the maximum number of characters permitted in one line of the host shell's window.
STRING_FREE [manual automatic]	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 <code>strFree()</code> .
SEARCH_ALL_SYMBOLS [ON OFF]	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]	Indicates the host shell's current interpreter mode and permits the user to switch from one mode to another.

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

Variable	Result
SH_GET_TASK_IO [ON OFF]	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 [ON OFF]	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 [ON OFF]	Sets the load mode. The default ON causes modules to be transferred to the target server. This means that any module the host shell can see can be loaded. If the variable is OFF , the target server must be able to see the module to load it.
LD_PATH <i>pathname</i>	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 [ON OFF]	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.

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

Variable	Result
DSM_HEX_MOD [ON OFF]	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 [vi emacs]	Sets the line edit mode to use. Set to emacs or vi . Default is vi .
RECORD [ON OFF]	Enable input/output logging. See 2.7 Host Shell Logging , p.22.
RECORD_FILE <i>filename</i>	Specify a file for input/output logging. See 2.7 Host Shell Logging , p.22.
RECORD_TYPE [input output all]	Specify type of input/output logging: input (input only), output (output only), or all (both input and output.) See 2.7 Host Shell Logging , p.22.
SINGLE_STEP [ON OFF]	By default this variable is set to OFF . When set to ON , any script you call is executed one line at a time. After each single-step, you can resume the script by pressing any key on your keyboard.
C_OUTPUT_GET [ON OFF]	<p>This variable is for use when calling to the C interpreter from the Tcl interpreter.</p> <p>When set to ON, the output returned to the Tcl interpreter is the data displayed on the shell's standard output.</p> <p>When set to OFF, the output returned to the Tcl interpreter is the result of the call to the C interpreter.</p>
BP_PRINT [ON OFF]	When this variable is set to OFF , the shell will not display a message on standard output when a breakpoint is hit.

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

Variable	Result
EXC_PRINT [ON OFF]	When this variable is set to OFF , the shell will not display a message on standard output when an exception is encountered.
VIO_PRINT [ON OFF]	When this variable is set to OFF , any data written to VIO will not be displayed in the shell.
LD_UNLOAD_FIRST [ON OFF]	If this variable is set to ON , then when downloading any kernel module, if there is already a loaded kernel module with the same name, the shell unloads the existing module before downloading the new module.

2.5 Path Mapping

Since the host shell uses host paths to handle VxWorks 6.x RTPs and Linux processes in both the command and GDB interpreters, a path substitution mechanism operates to send the right target path to the debugger server.

This mechanism converts a host path passed on the command line to a target path understandable by both the debugger framework and the target, but you must provide the host shell with additional information before it can perform the conversion. Two shell environment variables are used to do this conversion: the **ROOT_PATH_MAPPING** and **VXE_PATH** variables.

The **ROOT_PATH_MAPPING** Variable

The **ROOT_PATH_MAPPING** environment variable is necessary for VxWorks 6.x and Linux targets. It defines path substitution pairs of the form `[tgtpath1,hostpath1][tgtpath2,hostpath2]...`

In an example where the host path is **C:/mydirectory/myrtp.vxe** and the target path is `hostname:/home/users/myName/mydirectory/myrtp.vxe`, the command is:

```
-> tcl
tcl> shConfig ROOT_PATH_MAPPING \[hostname:/home/users/myName/,C:/\]
```

Or in GDB mode,

```
(gdb) set tgtpathmapping [hostname:/home/users,/home/users]
```

(Note that in the GDB interpreter, square brackets do not have to be escaped.)

With this information, the host shell can compute the correct target path and send it to the debugger server. Note that the debugger server also needs this **ROOT_PATH_MAPPING** setting to retrieve the RTP or process file in order to parse the symbols, but the debugger server will send the path of this file directly to the target without any transformation by the host shell.

The VXE_PATH Variable

The **VXE_PATH** environment variable is necessary only for VxWorks 6.x targets. It is set in the command interpreter.

If, for example, the RTPs are located at **/folk/myName/rtp/bin/**, then you can set **VXE_PATH** to **/folk/myName/rtp/bin**:

```
[vxWorks *]# set config VXE_PATH=/folk/myName/rtp/bin
[vxWorks *]# helloworld.vxe
Hello World RTP!
```

This variable can contain several host paths separated by semi-colons, and is used as a **PATH** variable to indicate the locations in which the host shell should search for RTPs to launch.

If both **VXE_PATH** and **ROOT_PATH_MAPPING** are set, then the host shell reads successively each path in **VXE_PATH** and builds a full RTP path with the RTP passed to the command line. If this full host path matches one of the host paths stored in the **ROOT_PATH_MAPPING** variable, the host shell performs the corresponding path substitution on it to build a target path.

The result of this substitution is tested to discover if it is reachable from the target (by a **stat** performed on the target). If it is reachable, then this target path is sent to the debugger framework; if not, the host shell tries to apply another path substitution and when it reaches the end of **ROOT_PATH_MAPPING**, it retries other combinations with the next path stored in **VXE_PATH**.

2.6 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:

```
% hostShell -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:

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

Batch mode is useful for scripting automated tests, or automatically run applications, or system benchmarks, as part of a nightly build and test environment.

2.7 Host Shell Logging

The host shell uses three configuration variables to control input/output logging: **RECORD_TYPE**, **RECORD_FILE**, and **RECORD**. You can set these variables using the Tcl routine **shConfig**.

Since **shConfig** is a Tcl routine, you must either call it from within the host shell's Tcl interpreter, or, if you call it from within another interpreter, you must precede the **shConfig** command with the routine **shEval** and the Tcl special character (**shEval tcl**).

RECORD_TYPE can be set to any of the following:

- **input** - Only user commands are logged; shell output is not logged.

- **output** - Only shell output is logged; user commands are not logged.
- **all** - Both user commands and shell output are logged.

RECORD_FILE specifies the file to which data is logged, using the syntax

```
tcl> shConfig RECORD_FILE filename
```

If you enable logging without setting a value for **RECORD_FILE**, then the shell creates a file in the **temp** directory of the host upon which the host shell is running, and displays a message showing the location of the logging file. For example:

```
tcl> shConfig RECORD on
Started recording commands in '/tmp/shellRecordFile10406.cmds' (created).
```

Set the variable **RECORD** to **ON** to enable logging, and **OFF** to disable logging. If, within a shell session, you alternately set **RECORD** between **ON** and **OFF**, and the **RECORD_FILE** value remains the same (or is not specified), then the logging file is overwritten each time you set **RECORD** back to **ON**.

2.8 Host Shell Scripting

You can run a script from within any of the host shell interpreters.

Within the command interpreter or the C interpreter, you can run a script by using the redirection character “<” followed by the absolute path to the script file. For example, to run the script **myScript**, located at **C:/tmp**, in the C interpreter, enter the following:

```
-> < C:/tmp/myScript
```

In the command interpreter, use the same command.

In either the GDB interpreter or the Tcl interpreter, run a script using the **source** command, with the absolute path to the script as the argument.

```
tcl> source C:/tmp/myScript
```

or

```
(gdb) source C:/tmp/myScript
```

You can also invoke the host shell with a startup script using the command-line option **-s**. For example:

```
% hostShell -s C:/tmp/myScript %f%tsur%
```

2.8.1 Single-Stepping Scripts

The host shell allows you to single step scripts, allowing you to see the result of each command before the next one executes.

To enable single-stepping in the GDB interpreter, the C interpreter, or the command interpreter, enable the shell environment variable **SINGLE_STEP**, using one of the following commands:

For the GDB interpreter:

```
(gdb) set config SINGLE_STEP on
```

For the C interpreter:

```
-> shEval tcl shConfig SINGLE_STEP on
```

For the command interpreter:

```
[vxWorks*]# set config SINGLE_STEP=on
```

With the **SINGLE_STEP** variable enabled, the shell pauses after executing each line. Resume the script by pressing any key on your keyboard.

Note that you may not call any shell routine while stepping; that is, when the script is paused, your only available action is to resume it. You cannot call any shell routine while the script is paused.

2.8.2 Stepping in the Tcl Interpreter

The **SINGLE_STEP** variable does not work for the Tcl interpreter, because the host shell passes the Tcl **source** command to the Tcl interpreter and has no more interaction with the script.

You can single step a Tcl script by editing it to call the routine **::hostShell::tclShellStdinGet** at the end of each line you want to step.

If you write a Tcl script that contains a loop, you may wish to single step that loop. Using standard Tcl, you would do this by calling **get stdin** within the loop. However, in the host shell, if you execute the Tcl script from an input file, then the host shell's input is redirected to that file, and therefore the call to **get stdin** will be blocked.

To single step a Tcl loop executed from a script, call **::hostShell::tclShellStdinGet** instead of the standard **get stdin** routine. The **tclShellStdinGet** routine redirects the shell's input to **stdin**, enabling you to enter data using the keyboard, and therefore letting you resume the script when you wish.

For information on Tcl scripting, see [6.8 Tcl Scripting](#), p.123.

2.9 Host Shell Features

This section describes some of the features available in the host shell.

2.9.1 I/O Redirection

This feature is available only for VxWorks 6.x targets.

Developers often call routines that display data on standard output or accept data from standard input. By default the standard output and input streams are directed to the shell. For example, in a default configuration, invoking **printf()** from the shell gives the following display:

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

You can modify this using the Tcl procedure **shConfig** as follows:

```
-> shEval tcl shConfig SH_GET_TASK_IO off
->
-> printf("Hello World!\n")
value = 13 = 0xd
->
```

The shell now reports the **printf()** result, indicating that 13 characters have been printed. The output, however, goes to the target's standard output, not to the shell.

To determine the current configuration, use **shConfig**. If you issue the command without an argument, all parameters are listed. Use an argument to list only one parameter.

```
-> shEval tcl shConfig SH_GET_TASK_IO
SH_GET_TASK_IO = off
```

For more information on **shConfig**, see [2.4 Setting Shell Environment Variables](#), p.14.

The standard input and output are redirected for the function called from the shell. If the function called from the shell spawns other tasks, the input and output of the

spawned tasks are not redirected to the shell. To have all input and output redirected to the shell, use the following Tcl script:

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

2.9.2 Symbol Matching

Start to type any target symbol name and then type **CTRL+D**. The shell automatically lists all symbols matching the pattern:

```
[vxWorks] # sem[CTRL+D]
semPxShow          semShow
Symbol matching in vxKernel (PD ID 0x1efd40)
semTerminate       semTakeTbl      semTake            semSmTypeGetRtn
semSmShowRtn       semSmInfoRtn     semShowInit        semShow
semQPut            semQInit          semQGet            semQFlushDefer
semQFlush          semOTake          semMTake           semMPendQPut
semMLibInit        semMInit          semMGiveKernWork   semMGiveKern
semMGiveForce      semMGive          semMCreate         semMCoreInit
semLibInit         semInvalid        semIntRestrict     semInfo
semGiveTbl         semGiveDeferTbl   semGiveDefer       semGive
semFlushTbl        semFlushDeferTbl  semFlushDefer      semFlush
semDestroy         semDelete         semClear           semClassId
semClass           semCTake          semCLibInit        semCInit
semCGiveDefer      semCGive          semCCreate         semCCoreInit
semBTake           semBLibInit       semBInit           semBGiveDefer
semBGive           semBCreate        semBCoreInit
[vxWorks] # sem
```



NOTE: Symbol matching is not available for the GDB interpreter.

2.9.3 Directory and File Listing

You can also use **CTRL+D** to list all the files and directories that match a certain string. For example, to list all files and directories under **R:** that begin with **t**, type the following:

```
[vxWorks] # r:/t[CTRL+D]
```

```
t2cp2/
t3Keys/
taskSpawn
TDK-13671_001211_160045/
tornadoARMT2/
tornadoppC/
trgsh/
tsr152294src/
[vxWorks] # r:/t

t2i86config/
t3pen0107b/
TDK-13440_000504_104211_tar.gz
TORHELLO.WAV
tornadoi86t2/
torVars.bat
triggering/
tsr154738/
```

Directory and file listing is supported only in the host shell, not the kernel shell. Also, directory and file listing is not available for the GDB interpreter.

2.9.4 Target Symbol and Path Completion

Start to type any target symbol name or any existing directory name and then type **TAB**. The shell automatically completes the command or directory name for you. If there are multiple options, it prints them for you and then reprints your entry. You can add one or more letters and then type **TAB** again until the path or symbol is complete.

Symbol completion is supported in both the host shell and the kernel shell. Path completion is supported only in the host shell.

2.9.5 Synopsis Printing

Once you have typed the complete function name followed by a space, typing **CTRL+D** (not **TAB**) again prints the function synopsis, then reprints the function name ready for your input. (This function is not supported in the kernel shell.)

```
[vxWorks] # _taskIdDefault [CTRL+D]
taskIdDefault() - set the default task ID (WindSh)

int taskIdDefault
{
    int tid    /* user-supplied task ID; if 0, return default */
}

[vxWorks] # _taskIdDefault
```

If the routine exists on both host and target, the **hostShell** synopsis is printed. To print the target synopsis of a function, add the meta-character **@** before the function name.

You can extend the synopsis printing function to include your own routines. To do this, follow these steps:

1. Create the files that include the new routines following Wind River coding conventions.
2. Include these files in your project.
3. Add the filenames to the `DOC_FILES` macro in your makefile.
4. Go to the top of your project tree and run **make synopsis**:

```
[vxWorks] # cd installDir/vxworks-6.x/target/src/your_project
[vxWorks] # make synopsis
```

This adds your project file to the *installDir/vxworks-6.x/host/resource/synopsis* directory.



NOTE: Synopsis printing is not available for the GDB interpreter.

2.9.6 Data Conversion

Data conversion is available only in the C interpreter.

The shell prints all integers and characters in both decimal and hexadecimal, and if possible, as a character constant or a symbolic address and offset.

```
-> 68
value = 68 = 0x44 = 'D'
-> 0xf5de
value = 62942 = 0xf5de = _init + 0x52
-> 's'
value = 115 = 0x73 = 's'
```

2.9.7 Data Calculation

Data calculation is available only in the C interpreter.

Almost all C operators can be used for data calculation. Use “(” and “)” to force order of precedence.

```
-> (14 * 9) / 3
value = 42 = 0x2a = '*'
-> (0x1355 << 3) & 0x0f0f
value = 2568 = 0xa08
-> 4.3 * 5
value = 21.5
```

Calculations with Variables

```
[vxWorks] # (j + k) * 3
value = ...
[vxWorks] # *(j + 8 * k)
(address 0xn timer:: value = 0 = 0x0
[vxWorks] -> x = (val1 - val2) / val3
new symbol "x" added to symbol table
address = 0xn timer:: value = 0 = 0x0
[vxWorks] # f = 1.41 * 2
new symbol "f" added to symbol table
f = 0x7d4746f8: value = 2.82
```

Variable **f** gets an 8-byte floating point value.

2.10 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**. If the shell is not accepting input (for example, if it has lost connection to the target server) you can use the interrupt key (**CTRL+BREAK** on Windows; **CTRL+C** on Linux or Solaris.)

2.11 Host Shell Architecture

The host shell integrates host and target resources in such a way that it creates the illusion of executing entirely on the target itself. However, most interactions with the shell exploit the resources of both host and target. [Table 2-3](#) shows how the shell distributes the interpretation and execution of the following simple expression:

```
-> dir = opendir ("/myDev/myFile")
```

Parsing the expression is the activity that controls overall execution, and dispatches the other execution activities. This takes place on the host, in the shell's C interpreter, and continues until the entire expression is evaluated and the shell displays its result.

Table 2-3 **Interpreting: `dir = opendir("/myDev/myFile")`**

Host Shell (On Host)	Target Server and Symbol Table (On Host)	Target Agent (On Target)
Parse the string <code>"/myDev/myFile"</code> .	Allocate memory for the string; return address A .	Write <code>"/myDev/myFile"</code> ; return address A .
Parse the name <code>opendir</code> .	Look up <code>opendir</code> ; return address B .	
Parse the function call <code>B(A)</code> ; wait for the result.		Spawn a task to run <code>opendir()</code> at address A , passing address B as an argument, and signal result C when done.
	Retrieve C from target agent and pass it to host shell.	
Parse the symbol <code>dir</code> .	Look up <code>dir</code> (fails.)	
Request a new symbol table entry <code>dir</code> .	Define <code>dir</code> ; return symbol D .	
Parse the assignment <code>D=C</code> .	Allocate agent-pool memory for the value of <code>dir</code> .	Write the value of <code>dir</code> .

To avoid repetitive clutter, [Table 2-3](#) omits the following important steps, which must be carried out to link the activities in the three contexts (and two systems) shown in each column of the table:

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

The first access to server and agent is to allocate storage for the string **“/myDev/myFile”** on the target and store it there, so that subroutines such as **opendir()** have access to it. There is a pool of target memory reserved for host interactions. Because this pool is reserved, it can be managed from the host system. The server allocates the required memory, and informs the shell of its location; the shell then issues the requests to actually copy the string to that memory. This request reaches the agent on the target, and it writes the 14 bytes (including the terminating null) there.

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

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

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

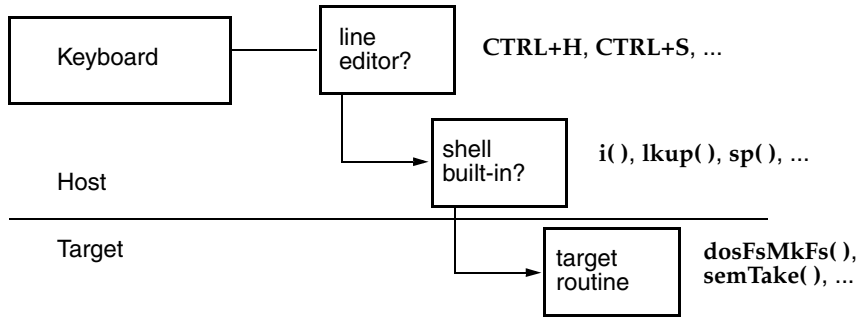
The C interpreter now has an address (in target memory) corresponding to **dir** on the left of the assignment statement; and it has the value returned by **opendir()** on the right of the assignment statement. It instructs the agent (again, through the server) to record the result at the **dir** address,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

2.11.1 Layers of Interpretation

To the user, the host shell seems to be a seamless environment; but in fact, the characters you type in the shell go through several layers of interpretation, as illustrated by [Figure 2-1](#). First, input is examined for special editing keystrokes (described in [A. Using the Host Shell Line Editor](#).) Then as much interpretation as possible is done in the host shell itself. In particular, execution of any subroutine is

first attempted in the shell itself; if a shell primitive with that name exists, it runs without any further checking. Only when a subroutine call does not match any shell primitives does the host shell call a target routine.

Figure 2-1 **Layers of Interpretation in the Host Shell**



For lists of host shell primitives, see the following chapters.

3

Using the C Interpreter with VxWorks 6.x

- 3.1 Introduction 35
- 3.2 C Interpreter Limitations 35
- 3.3 Host and Kernel Shell Differences 36
- 3.4 Running Target Routines From the Shell 38
- 3.5 Rebooting from the Shell 38
- 3.6 Using the Host Shell for System-Mode Debugging 39
- 3.7 Interrupting a Shell Command 43
- 3.8 Task References 44
- 3.9 Data Types 45
- 3.10 Expressions 47
- 3.11 Assignments 50
- 3.12 Comments 51
- 3.13 Strings 52
- 3.14 Ambiguity of Arrays and Pointers 53
- 3.15 Pointer Arithmetic 54
- 3.16 Redirection in the C Interpreter 54
- 3.17 C++ Interpretation 57
- 3.18 C Interpreter Primitives 60

3.19 Resolving Name Conflicts Between Host and Target 72

3.20 Examples 73

3.1 Introduction

This chapter describes the behavior of the C interpreter when used with a VxWorks 6.x target.

Note that C interpreter routine calls return 32-bit values only.

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 **Cmd** interpreter mode to debug VxWorks 6.x RTP applications, as described in [4. Using the Command Interpreter with VxWorks 6.x.](#))

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 the operating system itself. By using the host shell commands, you minimize the impact on both target memory and performance.

The shell parses and evaluates its input one line at a time. A line may consist of a single shell statement or several shell statements separated by semicolons. A semicolon is not required on a line containing only a single statement. A statement cannot continue on multiple lines.

Shell statements are either C expressions or assignment statements. Either kind of statement may call host shell commands or target routines.

3.2 C Interpreter Limitations

The C interpreter in the shell is not a complete interpreter for the C language. The following C features are not present in the host shell.

- Control structures

The shell interprets only C expressions (and comments). The shell does not support C control structures such as **if**, **goto**, and **switch** statements, or **do**, **while**, and **for** loops. Control structures are rarely needed during shell interaction. If you do come across a situation that requires a control structure, you can use the Tcl interface to the shell instead of using its C interpreter directly.

- Compound or derived types

No compound types (**struct** or **union** types) or derived types (**typedef**) are recognized in the shell C interpreter.

- Macros

No C preprocessor macros (or any other preprocessor facilities) are available in the shell. For constant macros, you can define variables in the shell with similar names to the macros. You can automate the effort of defining any variables you need repeatedly, by using an initialization script.

For control structures, or display and manipulation of types that are not supported in the shell, you might also consider writing auxiliary subroutines to provide these services during development; you can call such subroutines at will from the shell, and later omit them from your final application.

There are also certain limitations for C++ expressions: see [3.17 C++ Interpretation](#), p.57.

3.3 Host and Kernel Shell Differences

The host and kernel shells are almost identical. However, some of the commands (or routines) that you can execute from the shell are built into the host shell, rather than running as function calls on the target. These facilities parallel interactive utilities that can be linked into the target operating system itself. By using the host commands, you can minimize the impact on both target memory and performance.

Most of the shell commands correspond to similar routines that can be linked into the target operating system for use with the target-resident version of the shell. However, the target-resident routines differ in some details. For reference information on a shell command, be sure to consult the **windsh** reference entry.



CAUTION: Although there are usually entries with the same name in the VxWorks API references, these entries describe related target routines, not the shell commands.

Table 3-1 shows the differences between the host and kernel shells. For additional information on the kernel shell, see the *VxWorks Kernel Programmer's Guide: Target Tools*.

For information on shell commands, see the reference entries for the commands by opening Wind River Workbench and selecting **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > Routines Index**.

Table 3-1 Host Shell and Kernel Shell Differences

Features	Available in Mode	Host Shell	Kernel Shell
Symbol completion	C mode, cmd mode, Tcl mode	Yes	Yes
Path completion	All modes	Yes	No
Synopsis printing (CTRL+D)	C mode, cmd mode, gdb mode	Yes	No
HTML help (CTRL+W)	C mode, gdb mode	Yes	No

3.3.1 Function Calls in the Kernel

This section applies only to the C interpreter.

When using the kernel shell, function calls are executed by the shell task in the kernel and are therefore unbreakable.

```
-> b printf
value = 0 = 0x0
-> printf "Hello world\n"
Hello world
value = 12 = 0xc
```

In order to make the call break, you must use **sp()** to spawn a task to run the function.

```
-> sp printf, "Hello world\n"
task spawned: tid = 0x1f4008, name = t1
value = 0 = 0x0
->
Break at 0x00032cd8:printf Task: 0x001f4008
```

In the host shell, all function calls are breakable. This is because the host shell always creates a task to execute a function.

3.4 Running Target Routines From the Shell

All target routines are available from the host shell. This includes both VxWorks routines and your kernel application routines. This lets you test and debug your applications using all the host resources while having minimal impact on how the target performs and how the application behaves.

3.4.1 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 = ...
```

3.4.2 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 = ...
```

3.5 Rebooting from the Shell

In an interactive real-time development session, it is sometimes convenient to restart everything to make sure the target is in a known state. The host shell provides the **reboot()** command or **CTRL+X** for this purpose.

When you execute **reboot()** or type **CTRL+X**, the following reboot sequence occurs:

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

```
-> reboot
Rebooting...
```

2. The target reboots.

3. The original target server on the host detects the target reboot and restarts itself, with the same configuration as previously. The target-server configuration option **-Bt** (timeout) and **-Br** (retries) govern how long the new server waits for the target to reboot, and how many times the new server attempts to reconnect; see the **tgtsvr** reference entry.
4. The shell detects the target-server restart and begins an automatic-restart sequence (initiated any time it loses contact with the target server for any reason), indicated with the following message:

```
Target connection has been lost. Restarting shell...
```

followed by either:

```
Waiting to attach to target server
```

(indicates that the target server is restarting, the host shell is waiting for the attachment)

or

```
Waiting to attach to target agent
```

(indicates that the host shell is attached to the target server, but the target server is not yet attached to the target agent.)

When the host shell establishes contact with the new target server, it displays the prompt and awaits your input.



CAUTION: If the target server timeout (**-Bt**) and retry count (**-Br**) options are too low for your target and your connection method, the new target server may abandon execution before the target finishes rebooting. The default timeout is one second, and the default retry count is three; thus, by default the target server waits three seconds for the target to reboot. If the shell does not restart in a reasonably short time after a **reboot()**, try starting a new target server manually.

3.6 Using the Host Shell for System-Mode Debugging

The bulk of this chapter discusses the shell in its most frequent style of use: attached to a normally running VxWorks system, through a target agent running in task mode. However, you can also use the shell with a system-mode agent. Entering system mode stops the entire target system: all tasks, the kernel, and all

interrupt service requests (ISRs.) Similarly, breakpoints affect all tasks. One major shell feature is not available in system mode: you cannot execute expressions that call target-resident routines. You can still spawn tasks, but bear in mind that, because the entire system is stopped, a newly-spawned task can only execute when you allow the kernel to run long enough to schedule that task.

Depending on how the target agent is configured, you may be able to switch between system mode and task mode. When the agent supports mode switching, the following host shell commands control system mode:

- **sysSuspend()**
Enter system mode and stop the target system.
- **sysResume()**
Return to task mode and resume execution of the target system.

The following commands determine the state of the system and the agent:

- **agentModeShow()**
Show the agent mode (*system* or *task*).
- **sysStatusShow()**
Show the system context status (*suspended* or *running*).

The following shell commands behave differently in system mode:

- **b()**
Set a system-wide breakpoint; the system stops when this breakpoint is encountered by any task, or the kernel, or an ISR.
- **c()**
Resume execution of the entire system (but remain in system mode).
- **i()**
Display the state of the system context and the mode of the agent.
- **s()**
Single-step the entire system.
- **sp()**
Add a task to the execution queue. The task does not begin to execute until you continue the kernel or step through the task scheduler.

Example

This example uses system mode to debug a system interrupt.

In this case, **usrClock()** is attached to the system clock interrupt handler, which is called at each system clock tick when VxWorks is running. First suspend the system and confirm that it is suspended using either **i()** or **sysStatusShow()**.

```
-> sysSuspend
value = 0 = 0x0
->
-> i
NAME      ENTRY      TID      PRI      STATUS  PC      SP      ERRNO  DELAY
-----
tExcTask  _excTask  3e8f98   0       PEND    47982   3e8ef4   0       0
tLogTask  _logTask  3e6670   0       PEND    47982   3e65c8   0       0
tWdbTask  0x3f024   398e04   3       PEND    405ac   398d50   30067   0
tNetTask  _netTask  3b39e0   50      PEND    405ac   3b3988   0       0

Agent mode      : Extern
System context  : Suspended
value = 0 = 0x0
->
-> sysStatusShow
System context is suspended
value = 0 = 0x0
```

Next, set the system mode breakpoint on the entry point of the interrupt handler you want to debug. Since the target agent is running in system mode, the breakpoint will automatically be a system mode breakpoint, which you can confirm with the **b()** command. Resume the system using **c()** and wait for it to enter the interrupt handler and hit the breakpoint.

```
-> b usrClock
value = 0 = 0x0
-> b
0x00022d9a: _usrClock          Task:      SYSTEM Count:  0
value = 0 = 0x0
-> c
value = 0 = 0x0
->
Break at 0x00022d9a: _usrClock          Task: SYSTEM
```

You can now debug the interrupt handler. For example, you can determine which task was running when system mode was entered using **taskIdCurrent()** and **i()**.

```
-> taskIdCurrent
_taskIdCurrent = 0x838d0: value = 3880092 = 0x3b349c
-> i
NAME      ENTRY      TID      PRI      STATUS  PC      SP      ERRNO  DELAY
-----
tExcTask  _excTask  3e8a54   0       PEND    4eb8c   3e89b4   0       0
tLogTask  _logTask  3e612c   0       PEND    4eb8c   3e6088   0       0
tWdbTask  0x44d54   389774   3       PEND    46cb6   3896c0   0       0
```

```
tNetTask  _netTask  3b349c  50  READY  46cb6  3b3444  0  0
```

```
Agent mode      : Extern
System context  : Suspended
value = 0 = 0x0
```

You can trace all the tasks except the one that was running when you placed the system in system mode and you can step through the interrupt handler.

```
-> tt tLogTask
4da78  _vxTaskEntry  +10 : _logTask (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
3f2bc  _logTask      +18 : _msgQReceive (3e62e4, 3e60dc, 20, ffffffff)
27e64  _msgQReceive  +1ba: _qJobGet ([3e62e8, ffffffff, 0, 0, 0, 0])
value = 0 = 0x0
-> 1
_usrClock
00022d9a 4856                                PEA      (A6)
00022d9c 2c4f                                MOVEA .L  A7,A6
00022d9e 61ff 0002 3d8c                        BSR      _tickAnnounce
00022da4 4e5e                                UNLK     A6
00022da6 4e75                                RTS
00022da8 352e 3400                            MOVE .W  (0x3400,A6),-(A2)
00022dac 4a75 6c20                            TST .W  (0x20,A5,D6.L*4)
00022db0 3234 2031                            MOVE .W  (0x31,A4,D2.W*1),D1
00022db4 3939 382c 2031                        MOVE .W  0x382c2031,-(A4)
00022dba 343a 3337                            MOVE .W  (0x3337,PC),D2
value = 0 = 0x0
-> s
d0  =      3e  d1  =      3700  d2  =      3000  d3  =      3b09dc
d4  =      0  d5  =      0  d6  =      0  d7  =      0
a0  =      230b8  a1  =      3b3318  a2  =      3b3324  a3  =      7e094
a4  =      38a7c0  a5  =      0  a6/fp  =      bcb90  a7/sp  =      bcb84
sr  =      2604  pc  =      230ba
000230ba 2c4f                                MOVEA .L  A7,A6
value = 0 = 0x0
```

Return to task mode and confirm that return by calling **i0**.

```
-> sysResume
value = 0 = 0x0
-> i
NAME      ENTRY      TID      PRI      STATUS  PC      SP      ERRNO  DELAY
-----
tExcTask  _excTask  3e8f98  0      PEND    47982   3e8ef4  0      0
tLogTask  _logTask  3e6670  0      PEND    47982   3e65c8  0      0
tWdbTask  0x3f024  398e04  3      READY   405ac   398d50  30067  0
tNetTask  _netTask  3b39e0  50     PEND    405ac   3b3988  0      0
value = 0 = 0x0
```

If you want to debug an application you have loaded dynamically, set an appropriate breakpoint and spawn a task which runs when you continue the system:

```
-> sysSuspend
value = 0 = 0x0
-> ml < test.o
```

```
Loading /view/didier.temp/vobs/wpwr/target/lib/objMC68040gmutest//test.o
/
value = 400496 = 0x61c70 = _rn_addroute + 0x1d4
-> b address
value = 0 = 0x0
-> sp test
value = 0 = 0x0
-> c
```

The application breaks on address when the instruction at address is executed.

3.7 Interrupting a Shell Command

Occasionally it is desirable to abort the shell's evaluation of a statement. For example, an invoked routine may loop excessively, suspend, or wait on a semaphore. This may happen as the result of errors in arguments specified in the invocation, errors in the implementation of the routine itself, or simply oversight as to the consequences of calling the routine.

To regain control of the shell in such cases, press the interrupt character on the keyboard, usually **CTRL+BREAK** for Windows or **CTRL+C** for Linux and Solaris. This makes the shell stop waiting for a result and allows input of a new statement. Any remaining portions of the statement are discarded and the task that ran the function call is deleted.

Pressing **CTRL+BREAK** or **CTRL+C** is also necessary to regain control of the shell after calling a routine on the target that ends with **exit()** rather than **return()**.

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

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

In cases like this, you do not need to type **CTRL+BREAK** to recover control of the shell; it automatically returns to the prompt, just as if you had interrupted. Whether you interrupt or the shell does it for you, you can proceed to investigate the cause of the suspension.

An interrupted routine may have left things in a state which was not cleared when you interrupted it. For instance, a routine may have taken a semaphore, which

cannot be given automatically. Be sure to perform manual cleanup if you are going to continue the application from this point.

3.8 Task References

Most VxWorks routines that take an argument representing a task require a task ID. However, when invoking routines interactively, specifying a task ID can be cumbersome, since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, shell expressions can reference a task by either task ID or task name. The shell attempts to resolve a task argument to a task ID as follows: if no match is found in the symbol table for a task argument, the shell searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

When you enter any command, the shell attempts to match it in the following order: shell command, symbol, task name.

By convention, task names are prefixed by **s1**, **s2**, and so on: **s** to show that they were started by the shell, and **1**, **2**, and so on for the tool number of that shell. Task names are completed by **u0**, **u1**, and so on, where the integer indicates the task number. The integer is incremented by one each time the shell spawns a new task.

This avoids name conflicts with entries in the symbol table. The names of system tasks and the default task names assigned when tasks are spawned use this convention. Wind River recommends that you adopt a similar convention for tasks named in your applications.

The “Current” Task and Address

A number of commands, for example **c0**, **s0**, and **ti0**, take a task parameter that can be omitted. If omitted, the current task is used. The **l0** and **d0** commands use the current address if no address is specified. The current task and address are set when:

- A task hits a breakpoint or an exception trap. The current address is the address of the instruction that caused the break or exception.

- A task is single-stepped. The current address is the address of the next instruction to be executed.
- Any of the commands that use the current task or address are executed with a specific task parameter. The current address will be the address of the byte following the last byte that was displayed or disassembled.

3.9 Data Types

The most significant difference between the shell C-expression interpreter and a C compiler lies in the way that they handle data types. The shell does not accept any C declaration statements, and no data-type information is available in the symbol table. Instead, an expression's type is determined by the types of its terms.

Unless you use explicit type-casting, the shell makes the following assumptions about data types:

- In an assignment statement, the type of the left hand side is determined by the type of the right hand side.
- If floating-point numbers and integers both appear in an arithmetic expression, the resulting type is a floating-point number.

Data types are assigned to various elements, as shown in [Table 3-2](#).

Table 3-2 C Interpreter Data-Type Assumptions

Element	Data Type
variable	int
variable used as a floating-point	double
return value of subroutine	int
constant with no decimal point	int/long
constant with decimal point	double

A constant or variable can be treated as a different type than what the shell assumes by explicitly specifying the type with the syntax of C type-casting. Functions that return values other than integers require a slightly different

type-casting; see [3.10.4 Function Calls](#), p. 48. [Table 3-3](#) shows the various data types available in the shell C interpreter, with examples of how they can be set and referenced.

Table 3-3 Data Types in the C Interpreter

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

Strings, or character arrays, are not treated as separate types in the C interpreter. To declare a string, set a variable to a string value. (Memory allocated for string constants is never freed by the shell.) For example:

```
-> ss = "any string"
```

The variable `ss` is a pointer to the string `any string`. To display `ss`, enter

```
-> d ss
```

The `d()` command displays the memory where `ss` is pointing. You can also use `printf()` to display strings.

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

```
*(70000 + 3 * 16)
```

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

3.10 Expressions

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

3.10.1 Literals

The shell interprets the literals in [Table 3-4](#) in the same way as the C compiler, with one addition: the shell also allows hex numbers to be preceded by \$ instead of 0x.

Table 3-4 **Literals in the C Interpreter**

Literal	Example
decimal numbers	143967
octal numbers	017734
hex numbers	0xf3ba or \$f3ba
floating point numbers	555.555
character constants	'x' and '\$'
string constants	"This is a string."

3.10.2 Variable References

Shell expressions may contain references to variables whose names have been entered in the system symbol table. Unless a particular type is specified with a variable reference, the variable's value in an expression is the 4-byte value at the memory address obtained from the symbol table. It is an error if an identifier in an expression is not found in the symbol table, except in the case of assignment statements.

C compilers usually prefix all user-defined identifiers with an underscore, so that **myVar** is actually in the symbol table as **_myVar**. The identifier can be entered either way to the shell; the shell searches the symbol table for a match either with or without a prefixed underscore.

You can also access data in memory that does not have a symbolic name in the symbol table, as long as you know its address. To do this, apply the C indirection

operator `""` to a constant. For example, `*0x10000` refers to the 4-byte integer value at memory address 10000 hex.

3.10.3 Operators

The shell interprets the operators in [Table 3-5](#) in the same way as the C compiler.

Table 3-5 **Operators in the C Interpreter**

Operator Type	Operators					
arithmetic	+	-	*	/	unary-	
relational	==	!=	<	>	<=	>=
shift	<<	>>				
logical		&&	!			
bitwise		&	~	^		
address and indirection	&	*				

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

3.10.4 Function Calls

Shell expressions may contain calls to C functions (or C-compatible functions) whose names have been entered in the system symbol table; they may also contain function calls to host shell commands that execute on the host.

The shell executes such function calls in tasks spawned for the purpose, with the specified arguments and default task parameters; if the task parameters make a difference, you can call `taskSpawn()` instead of calling functions from the shell directly. The value of a function call is the 4-byte integer value returned by the function. The shell assumes that all functions return integers. If a function returns a value other than an integer, the shell must know the data type being returned before the function is invoked. This requires a slightly unusual syntax because you must cast the function, not its return value. For example:

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




NOTE: The examples in this book assume you are using the default shell prompts. However, you can change the C interpreter prompt to anything you like using the `shellPromptSet()` routine.

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

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

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

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

```
4 + myFunc2 ()
```

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

```
4 + myFunc2
```

An important exception to this occurs when the function name is the very first item encountered in a statement. See [3.10.5 Arguments to Commands](#), p.49.

Shell expressions can also contain calls to functions that do not have a symbolic name in the symbol table, but whose addresses are known to you. To do this, simply supply the address in place of the function name. Thus the following expression calls a parameter-less function whose entry point is at address 10000 hex:

```
0x10000 ()
```

3.10.5 Arguments to Commands

In practice, most statements input to the shell are function calls. To simplify this use of the shell, an important exception is allowed to the standard expression syntax required by C. When a function name is the very first item encountered in a shell statement, the parentheses surrounding the function's arguments may be omitted. Thus the following shell statements are synonymous:

```
-> rename ("oldname", "newname")  
-> rename "oldname", "newname"
```

as are:

```
->evtBufferAddress ( )  
->evtBufferAddress
```

However, note that if you wish to assign the result to a variable, the function call cannot be the first item in the shell statement—thus, the syntactic exception above does not apply. The following captures the address, not the return value, of `evtBufferAddress()`:

```
-> value = evtBufferAddress
```

3.11 Assignments

The shell C interpreter accepts assignment statements in the form:

addressExpression = expression

The left side of an expression must evaluate to an addressable entity; that is, a legal C value.

3.11.1 Typing and Assignment

The data type of the left side is determined by the type of the right side. If the right side does not contain any floating-point constants or non-integer type-casts, then the type of the left side will be an integer. The value of the right side of the assignment is put at the address provided by the left side. For example, the following assignment sets the 4-byte integer variable `x` to `0x1000`:

```
-> x = 0x1000
```

The following assignment sets the 4-byte integer value at memory address `0x1000` to the current value of `x`:

```
-> *0x1000 = x
```

The following compound assignment adds 300 to the 4-byte integer variable `x`:

```
-> x += 300
```

The following adds 300 to the 4-byte integer at address `0x1000`:

```
-> *0x1000 += 300
```

The following compound operators are available:

```
++ *= &=
-- /= |=
+= %= ^=
-=
```

3

3.11.2 Automatic Creation of New Variables

New variables can be created automatically by assigning a value to an undefined identifier (one not already in the symbol table) with an assignment statement.

When the shell encounters such an assignment, it allocates space for the variable and enters the new identifier in the symbol table along with the address of the newly allocated variable. The new variable is set to the value and type of the right-side expression of the assignment statement. The shell prints a message indicating that a new variable has been allocated and assigned the specified value.

For example, if the identifier **fd** is not currently in the symbol table, the following statement creates a new variable named **fd** and assigns to it the result of the function call:

```
-> fd = open ("file", 0)
```

3.12 Comments

The shell allows two kinds of comments.

First, comments of the form `/* ... */` can be included anywhere on a shell input line. These comments are simply discarded, and the rest of the input line evaluated as usual.

Second, any line whose first non-blank character is `#` is ignored completely.

3.13 Strings

When the shell encounters a string literal ("...") in an expression, it allocates space for the string including the null-byte string terminator. The value of the literal is the address of the string in the newly allocated storage. For instance, the following expression allocates 12 bytes from the target-agent memory pool, enters the string in those 12 bytes (including the null terminator), and assigns the address of the string to `x`:

```
-> x = "hello world"
```

Even when a string literal is not assigned to a symbol, memory is still permanently allocated for it. For example, the following uses 12 bytes of memory that are never freed:

```
-> printf ("hello world")
```

If strings were only temporarily allocated, and a string literal were passed to a routine being spawned as a task, then by the time the task executed and attempted to access the string, the shell would have already released, possibly even reused, the temporary storage where the string was held.

After extended development sessions, the cumulative memory used for strings may become noticeable. Use the routine **strFree()** to free up allocated strings. This routine presents a list of all allocated strings along with their addresses. To free one string, call **strFree()** with the address of the string as the argument. To free all allocated strings, call **strFree()** with **-1** as the argument.

3.13.1 Strings and Pathnames

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

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

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

You can use the **ld()** command with all of these variations of pathnames. The following **ld()** invocations are all correct and equivalent:

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

3.14 Ambiguity of Arrays and Pointers

In a C expression, a non-subscripted reference to an array has a special meaning, namely the address of the first element of the array. The shell, to be compatible, should use the address obtained from the symbol table as the value of such a reference, rather than the contents of memory at that address. Unfortunately, the information that the identifier is an array, like all data type information, is not available after compilation. For example, if a module contains the following:

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

you might be tempted to enter a shell expression as in Example 1.

Example 1

```
-> printf (string)
```

While this would be correct in C, the shell will pass the first 4 bytes of the string itself to **printf()**, instead of the address of the string. To correct this, the shell expression must explicitly take the address of the identifier, as in Example 2.

Example 2

```
-> printf (&string)
```

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

```
char *string = "hello";
```

then to a compiler, Example 1 would be correct and Example 2 would be wrong. This is especially confusing since C allows pointers to be subscripted exactly like arrays, so that the value of **string[0]** would be "h" in either of the above declarations.

Bear in mind that array references and pointer references in shell expressions are different from their C counterparts. In particular, array references require an explicit application of the address operator `&`.

3.15 Pointer Arithmetic

While the C language treats pointer arithmetic specially, the shell C interpreter does not, because it treats all non-type-cast variables as 4-byte integers.

In the shell, pointer arithmetic is no different than integer arithmetic. Pointer arithmetic is valid, but it does not take into account the size of the data pointed to. Consider the following example:

```
-> *myPtr + 4 = 5
```

Assume that the value of **myPtr** is 0x1000. In C, if **myPtr** is a pointer to a type char, this would put the value 5 in the byte at address at 0x1004. If **myPtr** is a pointer to a 4-byte integer, the 4-byte value 0x00000005 would go into bytes 0x1010–0x1013. The shell, on the other hand, treats variables as integers, and therefore would put the 4-byte value 0x00000005 in bytes 0x1004–0x1007.

3.16 Redirection in the C Interpreter

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

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

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

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

3.16.1 Ambiguity Between Redirection and C Operators

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

```
-> x > y
```

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

```
-> x; > y  
-> x > y;
```

The first line prints the value of the variable *x* to the output stream *y*. The second line prints on standard output the value of the expression “*x* greater than *y*.”

3.16.2 The Nature of Redirection

The redirection mechanism of the host shell is fundamentally different from that of the Windows command shell, although the syntax and terminology are similar.

In the host shell, redirecting input or output affects only a command executed from the shell. In particular, this redirection is not inherited by any tasks started while output is redirected.

For example, you might be tempted to specify redirection streams when spawning a routine as a task, intending to send the output of **printf()** calls in the new task to an output stream, while leaving the shell’s I/O directed at the virtual console. This stratagem does not work. For example, the shell input line:

```
-> taskSpawn (...myFunc...) > output
```

momentarily redirects the shell standard output during the brief execution of the spawn routine, but does not affect the I/O of the resulting task.

To redirect the input or output streams of a particular task, call **ioTaskStdSet()** once the task exists.

3.16.3 Scripts: Redirecting Shell I/O

A special case of I/O redirection concerns the I/O of the shell itself; that is, redirection of the streams the shell's input is read from, and its output is written to. The syntax for this is simply the usual redirection specification, on a line that contains no other expressions.

The typical use of this mechanism is to have the shell read and execute lines from a file. For example, the input lines:

```
-> <startup
```

or

```
-> < c:\fred\startup
```

cause the shell to read and execute the commands in the file **startup**, either on the current working directory (in the first example) or explicitly on the complete pathname (in the second example.) If your working directory is **\fred**, then the two examples are equivalent.

Such command files are called *scripts*. Scripts are processed exactly like input from an interactive terminal. After reaching the end of the script file, the shell returns to processing I/O from the original streams.

During execution of a script, the shell displays each command as well as any output from that command. You can change this by invoking the shell with the **-q** option (see [2Using the Host Shell](#), p.7.)

An easy way to create a shell script is from a list of commands you have just executed in the shell. The history command **h()** prints a list of the last 20 shell commands. The following creates the file **c:\tmp\script** with the current shell history:

```
-> h > c:\tmp\script
```

The command numbers must be deleted from this file before using it as a shell script.

Scripts can also be nested. That is, scripts can contain shell input redirections that cause the shell to process other scripts.



CAUTION: Input and output redirection must refer to files on a host file system. If you have a local file system on your target, files that reside there are available to target-resident subroutines, but not to the shell (unless you export them from the target using NFS, and mount them on your host).



CAUTION: Wind River recommends that you set the shell environment variable `SH_GET_TASK_IO` to **OFF** before you use redirection of input from scripts, or before you copy and paste blocks of commands to the shell command line. Otherwise commands might be taken as input for a command that precedes them, and thus get lost.

C Interpreter Startup Scripts

Host shell scripts can be useful for setting up your working environment. You can run a startup script through the shell C interpreter by specifying its name with the `-s` option. For example:

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

You can also use the `-e` option to run a Tcl expression at startup, or place Tcl initialization in `windsh.tcl` under your home directory.

You can use startup scripts for setting system parameters to personal preferences: defining variables, specifying the target's working directory, and so forth. They can also be useful for tailoring the configuration of your system without having to rebuild the image. For example:

- creating additional devices
- loading and starting up application modules
- adding a complete set of network host names and routes
- setting NFS parameters and mounting NFS partitions

3.17 C++ Interpretation

Workbench supports both C and C++ as development languages. For information about C++ development, see the *VxWorks Kernel Programmer's Guide: C++ Development*.

Because C and C++ expressions are so similar, the host shell C-expression interpreter supports many C++ expressions. The facilities explained in this chapter are all available regardless of whether your source language is C or C++. In addition, there are a few special facilities for C++ extensions. This section describes those extensions.

The host shell is not a complete interpreter for C++ expressions. In particular:

- The shell has no information about user-defined types.
- There is no support for the `::` operator.
- Constructors, destructors, and operator functions cannot be called directly from the shell.
- Member functions cannot be called with the `.` or `->` operators.

To exercise C++ facilities that are missing from the C interpreter, you can compile and download routines that encapsulate the special C++ syntax.

3.17.1 Overloaded Function Names

If you have several C++ functions with the same name, distinguished by their argument lists, call any of them as usual with the name they share. When the shell detects the fact that several functions exist with the specified name, it lists them in an interactive dialog, printing the matching functions' signatures so that you can recall the different versions and make a choice among them.

You make your choice by entering the number of the desired function. If you make an invalid choice, the list is repeated and you are prompted to choose again. If you enter 0 (zero), the shell stops evaluating the current command and prints a message like the following:

```
undefined symbol: your_function_name
```

This can be useful, for example, if you misspelled the function name and you want to abandon the interactive dialog. However, because the shell is an interpreter, not a compiler, portions of the expression may already have executed (perhaps with side effects) before you abandon execution in this way.

The following example shows how the support for overloaded names works. In this example, there are four versions of a function called `xmin()`. Each version of `xmin()` returns at least two arguments, but each version takes arguments of different types.

```
-> 1 xmin
"xmin" is overloaded - Please select:
  1: _xmin(double,double)
  2: _xmin(long,long)
  3: _xmin(int,int)
  4: _xmin(float,float)
Enter <number> to select, anything else to stop: 1
      _xmin(double,double):
3fe710 4e56 0000      LINK      .W      A6,#0
3fe714 f22e 5400 0008      FMOVE      .D      (0x8,A6),F0
3fe71a f22e 5438 0010      FCMP      .D      (0x10,A6),F0
```

```

3fe720 f295 0008      FB      .W      #0x8f22e
3fe724 f22e 5400 0010  FMOVE   .D      (0x10,A6),F0
3fe72a f227 7400      FMOVE   .D      F0,-(A7)
3fe72e 201f          MOVE    .L      (A7)+,D0
3fe730 221f          MOVE    .L      (A7)+,D1
3fe732 6000 0002      BRA      0x003fe736
3fe736 4e5e          UNLK     A6
value = 4187960 = 0x3fe738 = _xmin(double,double) + 0x28

-> 1 xmin
"xmin" is overloaded - Please select:
  1: _xmin(double,double)
  2: _xmin(long,long)
  3: _xmin(int,int)
  4: _xmin(float,float)
Enter <number> to select, anything else to stop: 3
      _xmin(int,int):
3fe73a 4e56 0000      LINK     .W      A6,#0
3fe73e 202e 0008      MOVE    .L      (0x8,A6),D0
3fe742 b0ae 000c      CMP     .L      (0xc,A6),D0
3fe746 6f04          BLE     0x003fe74c
3fe748 202e 000c      MOVE    .L      (0xc,A6),D0
3fe74c 6000 0002      BRA     0x003fe750
3fe750 4e5e          UNLK     A6
3fe752 4e75          RTS
      _xmin(long,long):
3fe7544e560000      LINK     .W      A6,#0
3fe758202e0008      MOVE    .L      (0x8,A6),D0
value = 4187996 = 0x3fe75c = _xmin(long,long) + 0x8

```

In this example, the user calls the disassembler to list the instructions for `xmin()`, then selects the version that computes the minimum of two **double** values. Next, the user invokes the disassembler again, this time selecting the version that computes the minimum of two **int** values. Note that a different routine is disassembled in each case.

3.17.2 Automatic Name Demangling

Many shell debugging and system information functions display addresses symbolically (for example, the `!0` routine). This might be confusing for C++, because compilers encode a function's class membership (if any) and the type and number of the function's arguments in the function's linkage name. The encoding is meant to be efficient for development tools, but not necessarily convenient for human comprehension. This technique is commonly known as *name mangling* and can be a source of frustration when the mangled names are exposed to the developer.

To avoid this confusion, the debugging and system information routines in the host shell print C++ function names in a demangled representation. Whenever the

shell prints an address symbolically, it checks whether the name has been mangled. If it has, the name is demangled (complete with the function's class name, if any, and the type of each of the function's arguments) and printed.

The following example shows the demangled output when **lkup()** displays the addresses of the **xmin()** functions mentioned in the previous section.

```
-> lkup "xmin"
_xmin(double,double)    0x003fe710 text      (templex.out)
_xmin(long,long)        0x003fe754 text      (templex.out)
_xmin(int,int)           0x003fe73a text      (templex.out)
_xmin(float,float)       0x003fe6ee text      (templex.out)
value = 0 = 0x0
```

3.18 C Interpreter Primitives

3.18.1 Managing Tasks

[Table 3-6](#) summarizes the host shell commands that manage VxWorks tasks. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Table 3-6 Task Management Commands

Call	Description
sp()	Spawn a task with default parameters.
sps()	Spawn a task, but leave it suspended.
tr()	Resume a suspended task.
ts()	Suspend a task.
td()	Delete a task.
period()	Spawn a task with entry point periodHost to call a function periodically.

Table 3-6 Task Management Commands

Call	Description
repeat()	Spawn a task with entry point repeatHost to call a function repeatedly.
taskIdDefault()	Set or report the default (current) task ID. (For information on how the current task is established and used, see The “Current” Task and Address , p.44.)

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()**. (This download is not always reliable with remote target servers.) 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.

3.18.2 Task Information

[Table 3-7](#) summarizes the host shell commands that report task information. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh**.)

Table 3-7 Task Information Commands

Call	Description
i()	Display system information. This command gives a snapshot of what tasks are in the system, and some information about each of them, such as state, PC, SP, and task control block (TCB) address. To save memory, this command queries the target repeatedly; thus, it may occasionally give an inconsistent snapshot.
iStrict()	Display the same information as i() , but query target system information only once. At the expense of consuming more intermediate memory, this guarantees an accurate snapshot.

Table 3-7 Task Information Commands

Call	Description
ti()	Display task information. This command gives all the information contained in a task's task control block (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 <i>task</i> is 0 (or the argument is omitted), the current task is reported.
w()	Print 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 a specified task if the argument is given.
tw()	Print information about the object the given task is pending on. This routine calls taskWaitShow() on the given task in verbose mode.
checkStack()	Show 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()	Display a stack trace.
taskIdFigure()	Report 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:

```
-> 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
```

The **w()** and **tw()** commands allow you to see what object a task is pending on. **w()** displays summary information for all tasks, while **tw()** displays object information

for a specific task. Note that the **OBJ_NAME** field is used only for objects that have a symbolic name associated with the address of their structure.

```
-> w
```

NAME	ENTRY	TID	STATUS	DELAY	OBJ_TYPE	OBJ_ID	OBJ_NAME
tExcTask	_excTask	3d9e3c	PEND	0	MSG_Q (R)	3d9ff4	N/A
tLogTask	_logTask	3d7510	PEND	0	MSG_Q (R)	3d76c8	N/A
tWdbTask	_wdbCmdLoo	36dde4	READY	0		0	
tNetTask	_netTask	3a43d0	READY	0		0	
u0	_smtask1	36cc2c	PEND	0	MSG_Q_S (S)	370b61	N/A
u1	_smtask3	367c54	PEND	0	MSG_Q_S (S)	370b61	N/A
u3	_taskB	362c7c	PEND	0	SEM_B	8d378	_mySem2
u6	_smtask1	35dca4	PEND	0	MSG_Q_S (S)	370ae1	N/A
u9	_task3B	358ccc	PEND	0	MSG_Q (S)	8cf1c	_myMsgQ

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

NAME	ENTRY	TID	STATUS	DELAY	OBJ_TYPE	OBJ_ID	OBJ_NAME
u1	_smtask3	367c54	PEND	0	MSG_Q_S (S)	370b61	N/A

```
Message Queue Id      : 0x370b61
Task Queueing         : SHARED_FIFO
Message Byte Len      : 100
Messages Max          : 0
Messages Queued       : 0
Senders Blocked       : 2
Send Timeouts         : 0
Receive Timeouts      : 0

Senders Blocked:
TID      CPU Number  Shared TCB
-----
0x36cc2c    0        0x36e464
0x367c54    0        0x36e47c

value = 0 = 0x0
->
```

3.18.3 System Information

[Table 3-8](#) summarizes the host shell commands that display information from the symbol table, from the target system, and from the shell itself. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Table 3-8 **System Information Commands**

Call	Description
devs(0)	List all devices known on the target system.
lkup(0)	List symbols from the symbol table.
lkAddr(0)	List symbols whose values are near a specified value.
d(0)	Display target memory. You can specify a starting address, size of memory units, and number of units to display.
l(0)	Disassemble and display a specified number of instructions.
printErrno(0)	Describe the most recent error status value.
version(0)	Print operating system version information.
cd(0)	Change the working directory on the host (does not affect target.)
ls(0)	List files in the host working directory.
pwd(0)	Display the current host working directory.
help(0)	Display a summary of shell commands.
h(0)	Display or set the size of shell history.
shellHistory(0)	Display or set the size of shell history.
shellPromptSet(0)	Change the C interpreter shell prompt.
printLogo(0)	Display the shell logo.

The **lkup(0)** 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 can use a regular expression, as in the following:

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

Regular-expression searches of the symbol table can be as simple or elaborate as required. For example, the following simple regular expression displays the names of three internal VxWorks semaphore functions:

```
-> lkup "sem.Take"
 semBTake          0x0002aeec text      (vxWorks)
 semCTake          0x0002b268 text      (vxWorks)
 semMTake          0x0002bc48 text      (vxWorks)
 value = 0 = 0x0
->
```

Another information command is a symbolic disassembler, **l0**. The command syntax is:

```
l [addr[, n]]
```

This command lists *n* disassembled instructions, starting at *addr*. If *n* is 0 or not given, the command uses the *n* from a previous **l0**, or if there is none, the default value (10). If *addr* is 0, **l0** starts from where the previous **l0** stopped, or from where an exception occurred (if there was an exception trap or a breakpoint since the last **l0** command).

The disassembler uses any symbols that are in the symbol table. If an instruction whose address corresponds to a symbol is disassembled (the beginning of a routine, for instance), the symbol is shown as a label in the address field. Symbols are also used in the operand field. The following is an example of disassembled code for an MC680x0 target:

```
-> l printf
_printf
 00033bce 4856          PEA          (A6)
 00033bd0 2c4f          MOVEA .L    A7,A6
 00033bd2 4878 0001      PEA          0x1
 00033bd6 4879 0003 460e PEA          _fioFormatV + 0x780
 00033bdc 486e 000c      PEA          (0xc,A6)
 00033be0 2f2e 0008      MOVE .L     (0x8,A6),-(A7)
 00033be4 6100 02a8      BSR          _fioFormatV
 00033be8 4e5e          UNLK          A6
 00033bea 4e75          RTS
```

This example shows the **printf()** routine. The routine does a **LINK**, then pushes the value of **std_out** onto the stack and calls the routine **fioFormatV()**. Notice that symbols defined in C (routine and variable names) are prefixed with an underscore (**_**) by the compiler.

Perhaps the most frequently used system information command is **dd()**, which displays a block of memory starting at the address that is passed to it as a parameter. As with any other routine that requires an address, the starting address can be a number, the name of a variable or routine, or the result of an expression.

Several examples of variations on **dd()** appear below.

Display starting at address 1000 decimal:

```
-> dd (1000)
```

Display starting at 1000 hex:

```
-> dd 0x1000
```

Display starting at the address contained in the variable **foo**:

```
-> dd foo
```

The above is different from a display starting at the address of **foo**. For example, if **foo** is a variable at location 0x1234, and that memory location contains the value 10000, **dd()** displays starting at 10000 in the previous example and at 0x1234 in the following:

```
-> dd &foo
```

Display starting at an offset from the value of **foo**:

```
-> dd foo + 100
```

Display starting at the result of a function call:

```
-> dd func (foo)
```

Display the code of **func()** as a simple hex memory dump:

```
-> dd func
```



CAUTION: Remember that the effect of a command may be different in the host and kernel shells. If you mount a drive on the target at **/ata0/**, you will be unable to **cd()** to it from the host shell, which has no concept of a target working directory. However, if you use **@cd**, the kernel shell will recognize the device.

3.18.4 System Modification and Debugging

Developers often need to change the state of the target, whether to run a new version of some software module, to patch memory, or simply to single-step a program. [Table 3-9](#) summarizes the shell commands of this type. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River**

Documentation > References > Host Tools > Wind River Host Shell API
Reference > windsh.)

Table 3-9 System Modification and Debugging Commands

Call	Description
ld()	Load an object module into memory and link it dynamically into the runtime.
unld()	Remove a dynamically-linked object module from target memory, and free the storage it occupied.
m()	Modify memory in <i>width</i> (byte, short, or long) starting at <i>addr</i> . The m() command displays successive words in memory on the terminal; you can change each word by typing a new hex value, leave the word unchanged and continue by typing ENTER , or return to the shell by typing a dot (".").
mRegs()	Modify register values for a specific task.
b()	Set or display breakpoints, in a specified task or in all tasks.
bh()	Set a hardware breakpoint.
s()	Step a program to the next instruction.
so()	Single-step, but step over a subroutine.
c()	Continue from a breakpoint.
cret()	Continue until the current subroutine returns.
bdall()	Delete all breakpoints.
bd()	Delete a breakpoint.
reboot()	Return target control to the boot loader, then reset the target server and reattach the shell.
bootChange()	Modify the saved values of boot parameters.
sysSuspend()	Enter system mode (if supported by the target-agent configuration.)
sysResume()	Return from system mode to task mode.

Table 3-9 **System Modification and Debugging Commands**

Call	Description
agentModeShow()	Show the agent mode (<i>system</i> or <i>task</i> .)
sysStatusShow()	Show the system context status (<i>suspended</i> or <i>running</i> .)
quit() or exit()	Close the shell.

One of the most useful shell features for interactive development is the dynamic linker. With the shell command **ld()**, you can download and link new portions of the application. Because the linking is dynamic, you only have to rebuild the particular piece you are working on, not the entire application. Download can be cancelled with **CTRL+C** or by clicking **Cancel** in the load progress indicator window.

The **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 an eligible task executes that instruction (as specified with the **b()** command), the task that was executing on the target suspends, and a message appears at the shell. At this point, you can examine the task's registers, do a task trace, and so on. The task can then be deleted, continued, or single-stepped.

If a routine called from the shell encounters a breakpoint, it suspends just as any other routine would, but in order to allow you to regain control of the shell, such suspended routines are treated in the shell as though they had returned 0. The suspended routine is nevertheless available for your inspection.

When you use **s()** to single-step a task, the task executes one machine instruction, then suspends again. The shell display shows all the task registers and the next instruction to be executed by the task.

You can use the **bh()** command to set hardware breakpoints at any instruction or data element. Instruction hardware breakpoints can be useful to debug code running in ROM or flash EPROM. Data hardware breakpoints (watchpoints) are useful if you want to stop when your program accesses a specific address. Hardware breakpoints are available on Intel x86, MIPS, and some PowerPC processors. The arguments of the **bh()** command are architecture-specific. For more information, run the **help()** command. The number of hardware breakpoints you can set is limited by the hardware; if you exceed the maximum number, you will receive an error.

3.18.5 C++ Development

[Table 3-10](#) describes commands that are intended specifically for C++ applications.

For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Also see the *VxWorks Kernel Programmer's Guide: C++ Development*.

Table 3-10 C++ Development Commands

Call	Description
cplusCtors()	Call static constructors manually.
cplusDtors()	Call static destructors manually.
cplusStratShow()	Report on whether current constructor/destructor strategy is manual or automatic.
cplusXtorSet()	Set constructor/destructor strategy.

In addition, you can use the Tcl routine **shConfig** to set the environment variable **LD_CALL_XTORS** within a particular shell. This allows you to use a different C++ strategy in a shell than is used on the target. For more information on **shConfig**, see [2.4 Setting Shell Environment Variables](#), p.14.

3.18.6 Object Display

[Table 3-11](#) describes commands that display VxWorks objects. The browser provides displays that are analogous to the output of many of these routines, except that browser windows can update their contents periodically.

For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Table 3-11 Object Display Commands

Call	Description
show()	Print information on a specified object in the shell window.
classShow()	Show information about a class of VxWorks kernel objects. List available classes with lkup "ClassId" .
taskShow()	Display information from a task's task control block (TCB.)
taskCreateHookShow()	Show the list of task create routines.
taskDeleteHookShow()	Show the list of task delete routines.
taskRegsShow()	Display the contents of a task's registers.
taskSwitchHookShow()	Show the list of task switch routines.
taskWaitShow()	Show information about the object a task is pended on. Note that taskWaitShow() cannot give object IDs for POSIX semaphores or message queues.
semShow()	Show information about a semaphore.
semPxShow()	Show information about a POSIX semaphore.
wdShow()	Show information about a watchdog timer.
msgQShow()	Show information about a message queue.
mqPxShow()	Show information about a POSIX message queue.
iosDrvShow()	Display a list of system drivers.
iosDevShow	Display the list of devices in the system.
iosFdShow()	Display a list of file descriptor names in the system.
memPartShow()	Show partition blocks and statistics at specified level of verbosity.

Table 3-11 Object Display Commands

Call	Description
memShow()	Display the total amount of free and allocated space in the system partition, the number of free and allocated fragments, the average free and allocated fragment sizes, and the maximum free fragment size. Show current as well as cumulative values. With an argument of 1 , also display the free list of the system partition; with an argument of 2 , display the address of each free block.
smMemShow()	Display the amount of free space and statistics on memory-block allocation for the shared-memory system partition.
smMemPartShow()	Display the amount of free space and statistics on memory-block allocation for a specified shared-memory partition.
moduleShow()	Show the current status for all loaded modules.
moduleIdFigure()	Report a loaded module's module ID, given its name.
intVecShow()	Display the interrupt vector table. This routine displays information about the given vector or the whole interrupt vector table if vector is equal to -1. Note that intVecShow() is not supported on architectures that do not use interrupt vectors.

3.18.7 Network Status Display

Table 3-12 describes commands that display information about the operating system network. In order for a protocol-specific command to work, the appropriate protocol must be included in your operating system configuration.

For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Table 3-12 Network Status Display Commands

Call	Description
hostShow()	Display the host table.
icmpstatShow()	Display statistics for Internet Control Message Protocol (ICMP).
ifShow()	Display the attached network interfaces.
inetstatShow()	Display all active connections for Internet protocol sockets.
ipstatShow()	Display IP statistics.
routeStatShow()	Display routing statistics.
tcpstatShow()	Display all statistics for the TCP protocol.
tftpInfoShow()	Get TFTP status information.
udpstatShow()	Display statistics for the UDP protocol.

3.19 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, **i0** 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 **i0**, invoke it with the name **@i0**.

3.20 Examples

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)
```

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 example below.

Call an application function that has the same name as a shell command.

-> `@i()`

4

Using the Command Interpreter with VxWorks 6.x

- 4.1 Introduction 76
- 4.2 General Commands 77
- 4.3 Displaying Target Agent Information 78
- 4.4 Working with Memory 79
- 4.5 Displaying Object Information 79
- 4.6 Working with Symbols 79
- 4.7 Displaying, Controlling, and Stepping Through Tasks 82
- 4.8 Setting Shell Context Information 83
- 4.9 Displaying System Status 83
- 4.10 Using and Modifying Aliases 84
- 4.11 Launching RTPs 86
- 4.12 Event Scripting Commands 89
- 4.13 General Examples 94

4.1 Introduction

This chapter describes the behavior of the command interpreter, which can be used only with a VxWorks 6.x target.

The host shell running in command interpreter mode allows debugging for VxWorks 6.x RTP applications.

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 the operating system itself. By using the host shell commands, you minimize the impact on both target memory and performance.

The shell parses and evaluates its input one line at a time. A line may consist of a single shell statement or several shell statements separated by semicolons. A semicolon is not required on a line containing only a single statement. A statement cannot continue on multiple lines.

To switch to command interpreter mode from any other mode, enter **cmd** at the prompt and press **ENTER**. For example, the default mode of the host shell when connected to a VxWorks 6.x target is the C interpreter mode. To switch to command interpreter mode, enter the following:

```
-> cmd  
[vxWorks]#
```

The command interpreter is command-oriented and does not understand C language syntax. (For C syntax, use the C interpreter, as described in [3. Using the C Interpreter with VxWorks 6.x.](#))

A command name is composed of one or more strings followed by option 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] [i]
```

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
```

4.2 General Commands

Table 4-1 summarizes general command-interpreter commands.

Table 4-1 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 <i>errno</i>	Displays the symbol value of an <i>errno</i> .
pwd	Displays the current working directory.

Table 4-1 **General Command Interpreter Commands** (cont'd)

Command	Description
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.

4.3 Displaying Target Agent Information

For information about the WDB target agent, see the *VxWorks Kernel Programmer's Guide: Target Tools*.

[Table 4-2](#) lists the commands related to the target agent.

Table 4-2 **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.

4.4 Working with Memory

Table 4-3 shows commands related to memory.

Table 4-3 **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.

4.5 Displaying Object Information

Table 4-4 shows commands that display information about objects.

Table 4-4 **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.

4.6 Working with Symbols

Table 4-5 lists commands for displaying and setting values of symbols.

Table 4-5 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.

4.6.1 Accessing a Symbol's Contents and Address

The host shell command interpreter is a string-oriented interpreter, but you may want to distinguish between symbol names, regular strings, and numerical values.

When a symbol name is passed as an argument to a command, you 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.

4.6.2 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.

4.6.3 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

```

4.6.4 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

```

4.7 Displaying, Controlling, and Stepping Through Tasks

Table 4-6 displays commands for working with tasks.

Table 4-6 **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.
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.

4.8 Setting Shell Context Information

Table 4-7 displays commands for displaying and setting context information.

Table 4-7 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
unset config	Removes a shell configuration variable from the current shell session.

4.9 Displaying System Status

Table 4-8 lists commands for showing system status information.

Table 4-8 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.

4.10 Using and Modifying Aliases

The command interpreter accepts aliases to speed up access to shell commands. [Table 4-9](#) lists the aliases that already exist; they can be modified, and you can add new aliases. Aliases are visible from all shell sessions.

Table 4-9 Command Interpreter Aliases

Alias	Definition
alias	List existing aliases. Add a new alias by typing alias <i>aliasname</i> " <i>command</i> ". For example, alias ll "ls -l" .
attach	rtp attach
b	bp
bd	bp -u
bdall	bp -u #*
bootChange	set bootline
c	task continue
checkStack	task stack

Table 4-9 Command Interpreter Aliases (cont'd)

Alias	Definition
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
rtps	rtp stop
run	rtp exec
s	task step
so	task stepover
td	task delete

Table 4-9 **Command Interpreter Aliases** (cont'd)

Alias	Definition
ti	task info
tr	task resume
ts	task suspend
tsp	task spawn
tt	task trace
vi	set config LINE_EDIT_MODE="vi"

4.11 Launching RTPs

From the command interpreter, type the RTP executable pathname as a regular command, adding any command arguments after the RTP executable 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.
```

4.11.1 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.

    }
}
```

4.11.2 Monitoring and Debugging RTPs

[Table 4-10](#) displays the commands related to RTPs.

Table 4-10 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.
rtp	Displays a list of processes.
rtp stop	Stops a process.

Table 4-10 **Command Interpreter RTP Commands** (cont'd)

Command	Description
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.

4.11.3 Setting Breakpoints

The **bp** command displays, sets, or unsets a breakpoint in the kernel, in an RTP, for any task, for a particular task, or for a particular context. A breakpoint number is assigned to each breakpoint, which can be used to remove that breakpoint.

Enter **bp** with no arguments to display breakpoints currently set.

Syntax

```
bp [-p rtpIdNameNumber] [-t taskId] [[-u {bp_number | bp_addr} ...] | [-n count] [-h  
type] [-q] [-a] [expr]]
```

Use the special character **#*** as the breakpoint number, or the special character ***** as the breakpoint address, to unset all breakpoints.

Table 4-11 **bp Command Options**

Option	Description
-a	Stop all tasks in a context.
-n	Specify <i>count</i> passes before breakpoint is hit.
-h	Specify a hardware breakpoint.
-p	Apply the breakpoint to a specific RTP.
-q	Do not send notification when the breakpoint is hit.
-t	Apply the breakpoint to a specific task.
-u	Unset breakpoint.

You can set breakpoints in a memory context only if the *current working* memory context is set to that memory context.

4.12 Event Scripting Commands

This section describes commands used with Tcl event scripting (see [6.8 Tcl Scripting](#), p.123.)

handler add

Add an event handler to the shell.

Syntax

```
handler add [-e event_type] [-b breakpoint_number] [-d] [-n] [tcl_script | tcl_routine_name]
```

The host shell calls the handler when the specified event is encountered. You can specify the following options:

-e *event_type*

This is the event that triggers the handler. By default, this is the stopped event. *event_type* is any of the events sent by the backend server. [Table 4-12](#) shows a list of the most important event types.

Table 4-12 **Event Types**

Event Name	Meaning	Reason
stopped	Target has stopped.	One of the following: breakpoint-hit signal-received end-stepping-range exited watchpoint-trigger user-stopped
context-start	Context has started.	A kernel task or real-time process has started.
context-exit	Context has exited.	A kernel task or real-time process has exited.

-b *breakpoint_number*

If you want the host shell to call the handler when a breakpoint is hit, you can specify the breakpoint number with this option. If this option is not specified, all breakpoints will trigger the handler.

-d

Disable the handler. By default, the handler is enabled.

-n

Do not run the default handler for this event. By default, the default handler will run after the user handler.

You can specify the handler to be executed when the event is encountered by using this command as a Tcl routine, giving a full path to a Tcl script, or entering the script manually at the **tcl>** prompt in the Tcl interpreter.

handler show

Show any event handlers you have registered.

Syntax

```
handler show
```

handler remove

Remove a specified event handler.

Syntax

```
handler remove [-a]
```

This command removes the user event handler specified by the handler ID.

If you specify the flag **-a**, all handlers are removed.

handler enable

Enable the user event handler.

Syntax

```
handler enable [-a] [-d]
```

This command enables the user event handler specified by the handler ID.

If you specify the flag **-a**, all handlers are enabled.

If you specify the flag **-d**, the handler is disabled.

4.12.1 Limitations

Handlers have the following limitations:

- One handler per event type or event ID.
- If you register a handler for an event that already has a handler, the original handler is overwritten.
- If the handler is specified to be triggered by an event ID, then when the event is removed, the handler is also removed.
- If your handler calls one of the shell's interpreters, it must use the shell function **shEval** followed by the interpreter name and then the command. The interpreter name alone followed by the command will not succeed.

- When using eventpoint scripting on slow or remote target connections, the performance of the host shell is significantly affected. This is principally due to the length of time required by the backend to communicate with the target. Therefore Wind River recommends that you make use of the eventpoint scripting capability on local targets, and limit as far as possible the amount of event exchange between the host shell and the target when a script is called. (For example, launch RTPs without VIO redirection; make event handlers very short with as few calls to backend or shell APIs as possible; where possible, limit the use of recurrent event handlers.)
- If you specify a handler that listens for the context-start event, you must be careful when calling any of the shell APIs to create a context (for example, the C interpreter's **sp0** command or the Cmd interpreter's **rtp exec** command.) These shell APIs also listen for the context-start event and act upon that event. If your handler uses a **while** loop, the shell is likely to hang and the call to one of the APIs to create a context will not succeed. As a workaround, you can create a context using the GDB/MI commands directly, through the GDB interpreter's **mi** commands, and not rely on the shell's APIs, as shown in the following example:

The user wishes to add a user handler listening for the context-start event. When the event is received, the handler calls a GDB/MI command to resume the context, and then loop until the context exits.

```
proc taskWatchHandler {evt} {  
    regexp {thread-id=\"([^\"]+)\"} $evt dummy threadId  
    shEval gdb mi "-wrs-tos-object-modify -t $threadId KernelTask  
taskResume --"  
    set taskId [expr int([taskIdGet $threadId])]  
    puts "taskWatchHandler Task $taskId"  
    while {1} {  
        set taskList [shEval cmd task]  
        set idx 0  
        set taskFound 0  
        foreach task [split $taskList "\n"] {  
            set id [lindex $task 2]  
            if {[catch {expr int($id)} err]} {  
                if {$err == $taskId} {  
                    set taskFound 1  
                }  
            }  
        }  
    }  
    if {!$taskFound} {  
        puts "TASK EXITED"  
        return  
    }  
    after 1000  
}  
}
```

In the shell, the user registers the handler and then creates a task calling taskDelay(500). The handler is called and it reports when the task has exited.

```
[vxWorks *]# lkup taskDelay
taskDelay          0x6012be20 text      (ctdt.c)
taskDelaySc        0x60130c20 text      (ctdt.c)
[vxWorks *]# handler add -e context-start taskWatchHandler
Added user handler id: 2
[vxWorks *]# gdb mi -wrs-tos-object-create KernelTask taskSpawn -- s2u1
100 83886097 0 20000 0x6012be20 true false 0x64 0x0 0x0 0x0 0x0 0x0
0x0 0x0 0
^done,thread-id="39"
[vxWorks *]# taskWatchHandler Task 1617033120
TASK EXITED
```

4.12.2 Event Scripting Example

In this example, a handler has been added that overrides the default breakpoint handler. The handler calls the command interpreter's **task** command, and then calls **continue**. The call to **handler show** describes the handler that has just been added. A breakpoint is then set on **printf** and a task spawned to call **printf**; when the breakpoint is hit, the handler is called.

```
[vxWorks *]# handler add -n
Type Tcl script to be executed when event is encountered.
End with a line saying just "end".
puts "Breakpoint hit!!"
shEval cmd task
shEval cmd c
end
User Handler Added: 1
[vxWorks *]# handler show
```

Id	Event Type	Handler	Enabled	BreakpointId
1	stopped	puts "Breakpoint hit" shEval cmd task shEval cmd c	yes	ALL

```
-----
[vxWorks *]# bp &printf
[vxWorks *]# C sp printf, "coucou"
task spawned: id = 0x616ffd08, name = s2u0
value = 1634729224 = 0x616ffd08
[vxWorks *]# Breakpoint hit!!
```

NAME	ENTRY	TID	PRI	STATUS	PC	ERRNO	DELAY
tJobTask	jobTask	0x6038e2a0	0	Pend	0x60126df4	0	0
tExcTask	excTask	0x6018e1d0	0	Pend	0x60126df4	0	0
tLogTask	logTask	0x6039bc20	0	Pend	0x60124dab	0	0
tNbioLog	nbioLogServe	0x60392010	0	Pend	0x60126df4	0	0
tShell0	shellTask	0x6052d190	1	Pend	0x60126df4	0	0

```
tWdbTask    wdbTask      0x6038bbd8    3 Ready    0x60126df4      0      0
tErfTask    erfServiceTa 0x60447c40   10 Pend    0x60127414      0      0
tAioIoTask  aioIoTask    0x60457c00   50 Pend    0x60127414      0      0
tAioIoTask  aioIoTask    0x604399a8   50 Pend    0x60127414      0      0
tNetTask    netTask      0x603a3020   50 Pend    0x60126df4      0      0
tAioWait    aioWaitTask  0x604396d0   51 Pend    0x60126df4      0      0
s2u0        printf       0x616ffd08  100 Stop    0x60034c90      0      0
```

```
[vxWorks *]# task
NAME          ENTRY          TID    PRI    STATUS    PC          ERRNO    DELAY
-----
tJobTask      jobTask      0x6038e2a0    0 Pend    0x60126df4      0      0
tExcTask      excTask      0x6018e1d0    0 Pend    0x60126df4      0      0
tLogTask      logTask      0x6039bc20    0 Pend    0x60124dab      0      0
tNbioLog      nbioLogServe 0x60392010    0 Pend    0x60126df4      0      0
tShell0       shellTask    0x6052d190    1 Pend    0x60126df4      0      0
tWdbTask      wdbTask      0x6038bbd8    3 Ready    0x60126df4      0      0
tErfTask      erfServiceTa 0x60447c40   10 Pend    0x60127414      0      0
tAioIoTask    aioIoTask    0x60457c00   50 Pend    0x60127414      0      0
tAioIoTask    aioIoTask    0x604399a8   50 Pend    0x60127414      0      0
tNetTask      netTask      0x603a3020   50 Pend    0x60126df4      0      0
tAioWait      aioWaitTask  0x604396d0   51 Pend    0x60126df4      0      0
[vxWorks *]#
```

4.13 General Examples

List the contents of a directory.

```
[vxWorks]# ls -l /folk/usr
```

Create an alias.

```
[vxWorks]# alias ls "ls -l"
```

Summarize task control blocks (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
```


5

Using the GDB Interpreter

- 5.1 Introduction 98
- 5.2 General GDB Commands 98
- 5.3 Working with Breakpoints 99
- 5.4 Specifying Files to Debug 100
- 5.5 Running and Stepping Through a File 100
- 5.6 Displaying Disassembly and Memory Information 101
- 5.7 Examining Stack Traces and Frames 102
- 5.8 Displaying Information and Expressions 102
- 5.9 Displaying and Setting Variables 104
- 5.10 Working with Signals 105
- 5.11 Event Scripting 108
- 5.12 Wind River On-Chip Debugging GDB Commands 112

5.1 Introduction

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

For Linux and standalone (no operating system) targets, the GDB interpreter is the default mode. For VxWorks 6.x targets, change to the GDB interpreter from any other interpreter by entering **gdb** at the prompt. For example, to change to GDB mode from C mode, enter the following:

```
-> gdb  
(gdb)
```

The GDB interpreter includes several Wind River-specific commands; these commands are prefaced with the prefix **wrs-** to prevent confusion with existing or future GDB commands. These commands are listed in [5.12 Wind River On-Chip Debugging GDB Commands](#), p. 112.

5.2 General GDB Commands

This section lists general commands available within the GDB interpreter.

Table 5-1 General GDB Commands

Command	Syntax	Description
help	help <i>command</i>	Print a description of <i>command</i> .
cd	cd <i>directory</i>	Change the working directory.
pwd	pwd	Print the working directory.
path	path <i>pathname</i>	Append <i>pathname</i> to the PATH variable.
show path	show path	Show the PATH variable.
echo	echo <i>string</i>	Echo a string.
list	list [<i>line</i> <i>symbol</i> <i>filename</i>]	Display ten lines of a source file, centered around a line number or symbol.
shell	shell <i>command</i>	Run a shell command.

Table 5-1 General GDB Commands

Command	Syntax	Description
source	source <i>filename</i>	Run a script of GDB commands.
directory	directory <i>dir</i>	Append a directory to the DIRECTORY variable (for source file searches.)
quit	q or quit	Quit the host shell.

5.3 Working with Breakpoints

This section lists commands available for setting and manipulating breakpoints.

Table 5-2 Breakpoint Commands

Command	Syntax	Description
break	break [<i>line</i> <i>symbol</i>] <i>filename</i> [if expr] or b [<i>line</i> <i>symbol</i>] <i>filename</i> [if expr]	Set a breakpoint.
tbreak	tbreak [<i>line</i> <i>symbol</i>] <i>filename</i> [if expr] or t [<i>line</i> <i>symbol</i>] <i>filename</i> [if expr]	Set a temporary breakpoint.
enable	enable <i>breakpoint_id</i>	Enable a breakpoint.
disable	disable <i>breakpoint_id</i>	Disable a breakpoint.
delete	delete <i>breakpoint_id</i>	Delete a breakpoint.
clear	clear <i>breakpoint_id</i>	Clear a breakpoint.
cond	cond <i>breakpoint_id</i> <i>condition</i>	Change a breakpoint condition (re-initializes the breakpoint).
ignore	ignore <i>breakpoint_id</i> <i>n</i>	Ignore a breakpoint <i>n</i> times (re-initializes the breakpoint).

5.4 Specifying Files to Debug

This section lists commands that specify the file(s) to be debugged.

Table 5-3 File Specification Commands

Command	Syntax	Description
file	file <i>filename</i>	Define <i>filename</i> as the program to be debugged.
exec-file	exec-file <i>filename</i>	Specify that the program to be run is located in <i>filename</i> .
load	load <i>filename</i>	Load a module.
unload	unload <i>filename</i>	Unload a module.
attach	attach <i>process_id</i>	Attach to a process.
detach	detach	Detach from the attached process.
thread	thread <i>thread_id</i>	Select a thread as the current task to debug.
add-symbol-file	add-symbol-file <i>filename addr</i>	Read additional symbol table information from the file located at memory address <i>addr</i> .

5.5 Running and Stepping Through a File

This section lists commands to run and step through programs.

Table 5-4 Run/Step Commands

Command	Syntax	Description
run	run	Run a process for debugging (use set arguments and set environment if your program needs them).
kill	kill <i>process_id</i>	Kill a process.

Table 5-4 Run/Step Commands

Command	Syntax	Description
interrupt	interrupt	Interrupt a running task or process.
continue	continue	Continue an interrupted task or process.
step	step <i>[n]</i>	Step one instruction. If <i>n</i> is used, step <i>n</i> times.
stepi	stepi <i>[n]</i>	Step one assembly-language instruction. If <i>n</i> is used, step <i>n</i> times.
next	next <i>[n]</i>	Continue to the next source line in the current stack frame. If <i>n</i> is used, continue through <i>n</i> lines.
nexti	nexti <i>[n]</i>	Execute one assembly-language instruction. If the instruction is a function call, proceed until the function returns. If <i>n</i> is used, execute <i>n</i> instructions.
until	until	Continue running until a source line past the current line in the current stack frame is reached.
jump	jump <i>addr</i>	Move the instruction pointer to <i>addr</i> .
finish	finish	Finish execution of current block.

5.6 Displaying Disassembly and Memory Information

This section lists commands for disassembling code and displaying contents of memory.

Table 5-5 Disassembly Commands

Command	Syntax	Description
disassemble	disassemble <i>addr</i>	Disassemble code at a specified address.
x	x [<i>lformat</i>] <i>addr</i>	Display memory starting at <i>addr</i> . <i>format</i> is one of the formats used by print : either s for a null-terminated string, or i for a machine instruction. Initially, the default is x for hexadecimal; but the default changes each time you use either x or print .

5.7 Examining Stack Traces and Frames

This section lists commands for selecting and displaying stack frames.

Table 5-6 Stack Trace Commands

Command	Syntax	Description
bt	bt [<i>n</i>]	Display back trace of <i>n</i> frames.
frame	frame [<i>n</i>]	Select frame number <i>n</i>
up	up [<i>n</i>]	Move <i>n</i> frames up the stack.
down	down [<i>n</i>]	Move <i>n</i> frames down the stack.

5.8 Displaying Information and Expressions

This section lists commands that display functions, registers, expressions, and other debugging information.

5.8.1 info

Display information on a specified option.

Syntax

info *option*

The **info** command takes the following options:

- **args** - Shows function arguments.
- **breakpoints** - Shows breakpoints.
- **commands** - Shows commands to be executed when a breakpoint is hit.
- **display** - Shows expressions to display when the program stops.
- **extensions** - Shows file extensions (**c**, **cpp**, and so on.)
- **functions** - Shows all functions.
- **locals** - Shows local variables.
- **proc** - Show **/proc** process information about any running process.
- **registers** - Shows contents of registers.
- **source** - Shows current source file.
- **sources** - Shows all source files of current process.
- **symbol** *addr* - Shows the symbol at address *addr*.
- **system** - Shows which debug mode is running on the target.
- **target** - Displays information about the target.
- **threads** - Shows all threads.
- **variables** - Shows all global and static variable names.
- **warranty** - Shows disclaimer information.
- **watchpoints** - Same as **breakpoints**.

5.8.2 print

Evaluate and print an expression.

The accessible variables are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.

The command prints a specified number of objects of a specified size according to a specified format.

Syntax

print */count format size expression*

count is a repeat count.

format can be any of the following:

- **o** (octal)
- **x** (hex)
- **d** (decimal)
- **u** (unsigned decimal),
- **t** (binary)
- **a** (address)
- **i** (instruction)
- **s** (string)



NOTE: The options **f** (float) and **c** (char) are not supported. The debugger returns the natural value for the given expression by default.

size can be any of the following:

- **b** (byte)
- **h** (half word)
- **w** (word)
- **g** (giant, 8 bytes)

5.9 Displaying and Setting Variables

This section lists commands for displaying and setting variables.

Table 5-7 Variable Commands

Command	Syntax	Description
set args	set args <i>arguments</i>	Specify the arguments to be used the next time a debugged program is run.
set emacs	set emacs	Set display to emacs mode.
set environment	set environment <i>varname=value</i>	Set environment variable <i>varname</i> to <i>value</i> . <i>value</i> may be any string interpreted by the program.
set tgtpathmapping	set tgtpathmapping	Set target to host pathname mappings.
set variable	set variable <i>expression</i>	Set variable value to <i>expression</i> .
show args	show args	Show arguments of the debugged program.
show environment	show environment	Show environment of the debugged program.

5.10 Working with Signals



NOTE: This section applies only to Linux targets.

This section lists commands for handling, sending, and killing POSIX-style process signals on the target.

5.10.1 handle

Specify how to handle a given signal.

Syntax

handle [*signal_name* | **all**] *action*

signal_name is the symbolic name of the signal, for example **SIGSEGV**. Setting this argument to **ALL** specifies all signals except those used by the debugger, typically **SIGTRAP** and **SIGINT**. To find available signals for your target operating system, use the command **INFO HANDLE**.

action can be any of the following:

- **stop** - Re-enter the debugger if this signal occurs.
- **nostop** - Do not re-enter the debugger if this signal occurs.
- **pass** - Allow the program to see this signal.
- **nopass** - Do not allow the program to see this signal.

pass and **stop** may be combined.

Example

```
(gdb) handle SIGINT stop
Signal      Stop      Print      Pass to program  Description
SIGINT      true      NotSupported true             Interrupt
```

5.10.2 info handle

Display available signals for the target operating system.

Syntax

info handle

Example

```
(gdb) info handle
Signal      Stop      Print      Pass to program  Description
SIGHUP      false     NotSupported true             Hangup
SIGINT      true      NotSupported true             Interrupt
SIGQUIT     false     NotSupported true             Quit
SIGILL      true      NotSupported false            Illegal Instruction
SIGTRAP     false     NotSupported true             Trap
SIGABRT     false     NotSupported true             Abort
SIGBUS      true      NotSupported false            Bus Error
SIGFPE      false     NotSupported true             Floating Point
                        Exception
SIGKILL     false     NotSupported true             Kill
SIGUSR1     false     NotSupported true             User
SIGSEGV     true      NotSupported false            Segmentation
                        Violation
SIGUSR2     false     NotSupported true             User
SIGPIPE     false     NotSupported true             Broken Pipe
```

SIGALRM	false	NotSupported	true	Alarm Clock
SIGTERM	false	NotSupported	true	Expiration
SIGSTKFLT	false	NotSupported	true	Software Termination
SIGCHLD	false	NotSupported	true	Stack overflow
SIGCONT	false	NotSupported	true	Child Exited
SIGSTOP	false	NotSupported	true	Continuation
SIGTSTP	false	NotSupported	true	Stop
SIGTTIN	false	NotSupported	true	Stop from tty
SIGTTOU	false	NotSupported	true	Background tty read
SIGURG	false	NotSupported	true	Background tty output
				Urgent Condition on
				I/O Channel
SIGXCPU	false	NotSupported	true	CPU Time Limit
				Exceeded
SIGXFSZ	false	NotSupported	true	File Size Limit
				Exceeded
SIGVTALARM	false	NotSupported	true	Virtual Time Alarm
SIGPROF	false	NotSupported	true	Profiling Time Alarm
SIGWINCH	false	NotSupported	true	Window Changed
SIGIO	false	NotSupported	true	I/O Ready

5.10.3 signal

Continue a running program, while giving it a specified signal.

Syntax

signal [*signal_name* | 0]

Specify 0 as the argument to continue the program without giving it a signal.

5.10.4 send signal

Send a specified signal to a specified program.

Syntax

send signal *signal_name* [*program*]

If you do not specify a program, the signal is sent to the current program.

Example

In this example, a breakpoint is set at the symbol **main**. Then the signal **SIGINT** is sent to the running program.

```
(gdb) b main
Breakpoint 5 at 0x80484DC: file signalTest.c, line 115.
```

```
(gdb) run
Starting program: ../signalTest

Breakpoint 5, main (argc=1, argv=0xBFFFD144) at signalTest.c:115

(gdb) send signal SIGINT
Sending signal SIGINT to program.

(gdb) c
Continuing.
Program received signal SIGINT, Interrupt at 0x80484DC

(gdb) signal SIGINT
Continuing with signal SIGINT.
(gdb) Program exited normally.
```

5.11 Event Scripting

This section lists commands for use with Tcl event scripting (see [6.8 Tcl Scripting](#), p.123.)

5.11.1 Event Scripting Commands

display

Print the value of an expression each time the program stops.

Syntax

```
display [/FMT i | s] expression
```

You can use the option /FMT to set the format: either **s** for a null-terminated string, or **i** for a machine instruction.

The command **display** with no arguments displays all currently requested auto-display expressions. Use **undisplay** to cancel a display request.

undisplay

Cancel display of expressions when the program stops.

Syntax

undisplay *args*

args are the code numbers of the expressions to stop displaying. For a current list of code numbers, use the command **info display**.

The command **undisplay** with no arguments cancels all automatic-display expressions.

This command is equivalent to the command **delete display**.

info display

Lists expressions currently specified to display when the program stops, with code numbers.

Syntax

info display

enable display

Enable expressions to be displayed when the program stops.

Syntax

enable display *args*

args are the code numbers of the expressions to resume displaying. For a current list of code numbers, use the command **info display**.

The command **enable display** with no arguments enables all automatic-display expressions.

disable display

Disable display of expressions when the program stops.

Syntax

disable display *args*

args are the code numbers of the expressions to stop displaying. For a current list of code numbers, use the command **info display**.

The command **disable display** with no arguments disables all automatic-display expressions.

commands

Set commands to be executed when a breakpoint is hit.

Syntax

```
commands breakpoint_number
```

If you do not enter a breakpoint number, the shell targets the breakpoint that was most recently set.

The commands themselves follow starting on the next line. To indicate the end of the commands, use the line **end**.

Example

```
commands 123  
silent  
command_1  
command_2  
command_3  
end
```

If you use **silent** as the first command, no output is printed when the breakpoint is hit, except any output specified by the subsequent commands.

info commands

Lists commands to be executed when a breakpoint is hit.

Syntax

```
info commands
```

enable commands

Enable commands to be executed when a breakpoint is hit.

Syntax

enable commands *args*

args are the code numbers of the commands to enable. For a list of code numbers, use the command **info commands**.

The command **enable commands** with no arguments enables all automatic-execution commands.

disable commands

Disable the ability to execute commands when a breakpoint is hit.

Syntax

disable commands *args*

args are the code numbers of the commands to stop executing. For a list of code numbers, use the command **info commands**.

The command **disable commands** with no arguments disables all automatic-execution commands.

5.11.2 Event Scripting Example

This example downloads a real-time process (RTP) and sets a breakpoint within that RTP. The user then calls **commands**, indicating that when the breakpoint is hit, the shell should call the GDB command **info proc**. The user then runs the RTP, the breakpoint is hit and the shell calls the command **info proc**. The user then calls **display**, indicating two variables to watch each time the program stops. The user calls **step** several times, and each time the step completes, the shell displays the value of the auto-watch variables.

```
(gdb) file /usr/bin/SIMPENTIUMdiab/printTest.vxe
Reading symbols from /usr/bin/SIMPENTIUMdiab/printTest.vxe...done
(gdb) b 46
Breakpoint 2 at 0x63000316: file printTest.c, line 46.
(gdb) commands
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
info proc
end
(gdb) run
Starting program: /usr/bin/SIMPENTIUMdiab/printTest.vxe

Breakpoint 2, func (val=3.14000000000000, val0=12345) at printTest.c:46
```

```
46                dummy1 = val0;
0x60556010 16 /usr/bin/SIMPENTIUMdiab/printTest.vxe 0x630002B1
RTP_GLOBAL_SYMBOLS|RTP_DEBUG RTP_NORMAL

(gdb) display dummy1
0: dummy1 = 48
(gdb) display dummy2
1: dummy2 = 8.60716350995449E+168
(gdb) step
0x6300031C      47                dummy2 = val;
0: dummy1 = 12345
1: dummy2 = 8.60716350995449E+168
(gdb) info display
Auto-display expressions now in effect:
Num  End Expression
0    y  dummy1
1    y  dummy2
(gdb)
```

5.12 Wind River On-Chip Debugging GDB Commands

This section lists GDB mode commands that are specific to Wind River On-Chip Debugging (OCD.)

5.12.1 target ocd

Spawn a backend server, connect to it, and connect to a target. If the host shell is already connected to a backend server, this command simply connects to a target using that backend server.

Syntax

target ocd *target-id*

target-id is one of the target IDs from the output of the **wrsregquery** command, or the *target-id* given to an earlier **wrsdeftarget** command.

There is no corresponding command to disconnect from the target or backend server. Once connected to a backend server and a target, the host shell remains connected until the user terminates the host shell.

If the host shell is not connected to a backend server when this command is issued, the host shell spawns a backend server and connects to it before sending GDB/MI messages.

5.12.2 **wrsdeftarget**

Create a new target definition.

Syntax

```
wrsdeftarget target-id --core core-name --cpuplugin cpu-plugin [ --targetplugin  
target-plugin ] param=value [ param=value ... ]
```

target-id is a user-supplied name for this target definition.

core-name is the type of the target CPU.

cpu-plugin is the name of the CPU plugin.

target-plugin is the name of the target plugin.



NOTE: If you omit the **--targetplugin** option, the host shell uses **ocdtargetplugin** by default.

param is one of the parameter names shown in [Table 5-8](#).

Table 5-8 **wrsdeftarget Parameter Names**

Parameter Name	Description
DEVICE	Specifies the device that is being connected to. Its value is one of the following strings, enclosed in double quotes: <ul style="list-style-type: none">▪ Wind River ICE - Connects to a Wind River ICE SX tool. For this DEVICE type, the BFNAME parameter is required; for other DEVICE types, the BFNAME parameter is optional.▪ Wind River Probe - Connects to a Wind River Probe tool.▪ Wind River ISS - Connects to the Wind River Instruction Set Simulator. For this DEVICE type, STYLE and ADDR are unnecessary.
STYLE	Specifies the style of the connection and how the ADDR parameter is interpreted. The value of this parameter can be either of the keywords ETHERNET or USBDEVICE .
ADDR	Specifies the connection address. When the parameter STYLE is set to ETHERNET , the value of ADDR is either an IP address or a hostname. When STYLE is set to USBDEVICE , the value of ADDR is the serial number printed on the back of your Wind River Probe.
BFNAME	Specifies the host pathname of the board descriptor file.

Example 1

```
wrsdeftarget mytarget --core MPC8260 --cpuplugin 82xxcpuplugin DEVICE='Wind
River ICE' STYLE=ETHERNET ADDR=123.456.789.012 BFNAME=WindRiverSBC8260.brd
```

Example 2

```
wrsdeftarget mytarget --core MPC8260 --cpuplugin 82xxcpuplugin DEVICE='Wind
River Probe' STYLE=USBDEVICE ADDR=PRO12345
```

If the host shell is not connected to a backend server when you issue this command, the host shell spawns a backend server and connects to it before sending GDB/MI messages.

The new target definition is transient; it does not persist beyond the lifetime of the backend server session in which it was created.

This command does not modify the contents of the Wind River Registry.

5.12.3 **wrsregquery**

wrsregquery queries the Wind River registry to obtain target definition information. Target definitions can later be given to the **wrsdeftarget** command (see 5.12.2 *wrsdeftarget*, p.113) to connect to a specific target.

Syntax

wrsregquery

The output is a list of target definitions having the format *target-id*, *target-name*.

target-id is a unique identifier specifying a target definition.

target-name is a non-unique human-readable version of the target definition.

Example

```
(gdb) wrsregquery
jsmith_1136574941992, WRISS_MPC8260
jsmith_1136836847022, vxsim0
jsmith_1140032123849, WRICE_MPC8260
```

This command does not display backend servers, even though the Wind River registry contains a list of backend servers running on the same host, because the host shell will only connect to the backend server specified by the **-ds** command-line option, or to a newly spawned backend server.

5.12.4 Reset and Download Commands

Wind River has created several GDB mode commands to allow you to perform the equivalent of a Wind River Workbench On-Chip Debugging reset and download operation. Table 5-9 lists these commands.

For full syntax and examples for these commands, see 7. *Executing an OCD Reset and Download*.

Table 5-9 **Reset and Download Commands**

Command	Description
wrsdownload	<p>This command has three separate syntaxes:</p> <ul style="list-style-type: none">– Download executables and raw data to the target.– Erase flash memory on the target.– Program flash memory on the target. <p>These three syntaxes cannot be used at the same time. (That is, you cannot specify more than one kind of operation in the arguments for one wrsdownload command.)</p>
wrsmemmap	<p>Specify whether the debugger backend has read/write access to target memory, and where in target memory such access is allowed. This command only affects the backend. It does not affect memory map registers on the target, and does not cause a state change.</p>
wrspassthru	<p>Pass commands directly to the firmware without interpretation.</p>
wrsplayback	<p>Play a file of commands directly to the firmware.</p>
wrsreset	<p>Reset one or more target cores.</p>
wrsupload	<p>Upload data from target memory.</p>

6

Using the Tcl Interpreter

- 6.1 Introduction 117
- 6.2 Controlling the Target 118
- 6.3 Accessing the WTX Tcl API 120
- 6.4 Calling Target Routines 120
- 6.5 Passing Values to Target Routines 121
- 6.6 Calling Under C Control 121
- 6.7 Shell Initialization 122
- 6.8 Tcl Scripting 123

6.1 Introduction

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.

The Tcl interpreter is available for all target operating systems. To switch to the Tcl interpreter from another mode, type the Tcl special character **tcl** 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 **shEval tcl** before your line of Tcl code.



CAUTION: You may not embed Tcl evaluation inside a C expression; the **tcl** prefix works only as the first non-blank character on a line, and passes the entire line following it to the Tcl interpreter.

The following example uses the C interpreter to define a variable in the symbol table, then switch to the Tcl interpreter to define a similar Tcl variable in the shell itself, and then switch back to the C interpreter:

```
-> foo="bar"
new symbol "foo" added to symbol table.
foo = 0x3616e8: value = 3544824 = 0x3616f8 = foo + 0x10
-> tcl
tcl> set foo {bar}
bar
tcl> C
->
```

On startup, you can use the option **-Tclmode** (or **-T**) to start with the Tcl interpreter.

Using the shell's Tcl interface allows you to extend the shell with your own procedures, and also provides a set of control structures which you can use interactively. The Tcl interpreter also gives you access to command-line utilities on your development host.

6.2 Controlling the Target

In the Tcl interpreter, you can create custom commands, or use Tcl control structures for repetitive tasks, while using the building blocks that allow the C interpreter and the host shell commands to control the target remotely. These building blocks as a whole are called the **wtxtcl** procedures.

For example, **wtxMemRead** returns the contents of a block of target memory (given its starting address and length). That command in turn uses a special memory-block data type designed to permit memory transfers without unnecessary Tcl data conversions. The following example uses **wtxMemRead**, together with the memory-block routine **memBlockWriteFile**, to write a Tcl procedure that dumps target memory to a host file. Because almost all the work is done on the host, this procedure works whether or not the target run-time environment contains I/O libraries or any networked access to the host file system.

```
# tgtMemDump - copy target memory to host file
#
```

```
# SYNOPSIS:
#  tgtMemDump hostfile start nbytes

proc tgtMemDump {fname start nbytes} {
    set memHandle [wtxMemRead $start $nbytes]
    memBlockWriteFile $memHandle $fname
}
```

For reference information on the **wtxtcl** routines available in the host shell, see the online help: in Workbench, select **Help > Help Contents > Wind River Documentation > References > Host Tools > WTX Tcl Library Reference**.

All of the commands defined for any other interpreter are also available from the Tcl level. To use another interpreter's commands, use the **shEval** command followed by the special character for the interpreter you want and then the command.

For example, to call the C interpreter command **i()** from the Tcl interpreter, use the following command:

```
tcl> shEval C i
```

The output of a call to another interpreter can be written to a Tcl variable. For example, to write the list of real-time processes returned by the command **rtp** into the Tcl variable **rtpList**, enter the following:

```
tcl> set rtpList [shEval cmd rtp]
```

The behaviour for the C interpreter is slightly different. Since you can call target functions with the C interpreter, you may wish to recuperate the result of that target function call. To do so, enter the following:

```
tcl> set sysClk [shEval C sysClkRateGet]
```

The Tcl variable **sysClk** now contains the value returned by **sysClkRateGet()**.

If you want to set a Tcl variable to the value that is displayed on standard output when a C interpreter routine is called, set the shell configuration variable **C_OUTPUT_GET** to **ON**. With this variable set to **ON**, the following call writes the list of tasks as displayed on standard output to the Tcl variable **taskList**.

```
tcl> shConfig C_OUTPUT_GET on
tcl> set taskList [shEval C i]
```

In some cases, it is more convenient to call a **wtxtcl** routine instead.

For example, you can call the dynamic linker using **ld** from the Tcl interpreter, but the argument that names the object module may not seem intuitive: it is the address of a string stored on the target. It is more convenient to call the underlying **wtxtcl** command. In the case of the dynamic linker, the underlying **wtxtcl**

command is **wtxObjModuleLoad**, which takes an ordinary Tcl string as its argument.

6.3 Accessing the WTX Tcl API

The Wind River Tool Exchange (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 reference of WTX Tcl API commands, see the online help: in Workbench, select **Help > Help Contents > Wind River Documentation > References > Host Tools > WTX Tcl Library Reference**.

6.4 Calling Target Routines

The **shParse** utility allows you to embed calls to the C interpreter in Tcl expressions; the most frequent application is to call a single target routine, with the arguments specified (and perhaps capture the result). For example, the following sends a logging message to your target console:

```
tcl> shParse {logMsg("foobar\n")}  
32
```

You can also use **shParse** to call host shell commands more conveniently from the Tcl interpreter, rather than using their **wtxtcl** building blocks. For example, the following is a convenient way to spawn a task from Tcl, using the C interpreter command **sp0**, if you do not remember the underlying **wtxtcl** command:


```
tcl> shParse {sp appTaskBegin}
task spawned: id = 25e388, name = u1
0
```

6.5 Passing Values to Target Routines

6

Because **shParse** accepts a single, ordinary Tcl string as its argument, you can pass values from the Tcl interpreter to C subroutine calls by using Tcl facilities to concatenate the appropriate values into a C expression.

For example, a more realistic way of calling **logMsg()** from the Tcl interpreter would be to pass, as its argument, the value of a Tcl variable rather than a literal string. The following example evaluates the Tcl variable **tclLog** and inserts its value (with a newline appended) as the **logMsg()** argument:

```
tcl> shParse "logMsg(\"$tclLog\\n\")"
32
```

6.6 Calling Under C Control

To use a Tcl command and return immediately to the C interpreter, you can type a single line of Tcl prefixed with the **shEval** command and the Tcl special character **tcl** (rather than using **tcl** by itself to toggle into Tcl mode). For example:

```
-> shEval tcl set test foobar; puts "This is $test."
This is foobar.
->
```

Notice that the **->** prompt indicates that you are still in the C interpreter, even though you just executed a line of Tcl.



CAUTION: You may not embed Tcl evaluation inside a C expression; the **shEval tcl** prefix works only as the first nonblank character on a line, and passes the entire line following it to the Tcl interpreter.

For example, you may want to use Tcl control structures to supplement the facilities of the C interpreter. Suppose you have an application under development

that involves several collaborating tasks; in an interactive development session, you may need to restart the whole group of tasks repeatedly. You can define a Tcl variable with a list of all the task entry points, as follows:

```
-> shEval tcl set appTasks {appFrobStart appGetStart appPutStart ...}  
appFrobStart appGetStart appPutStart ...
```

Then whenever you need to restart the whole list of tasks, you can use something like the following:

```
-> shEval tcl foreach it $appTasks {shParse "sp($it)"}  
task spawned: id = 25e388, name = u0  
task spawned: id = 259368, name = u1  
task spawned: id = 254348, name = u2  
task spawned: id = 24f328, name = u3
```

6.6.1 Potential Problems

The **HOME** environment variable must be accessible and writeable in order to call to another interpreter from the Tcl interpreter. If this variable is not accessible, attempting to call to another interpreter from the Tcl interpreter returns the following error:

```
""
```

When a call to an external interpreter from the Tcl interpreter occurs, the host shell writes a temporary file to the directory indicated by the **HOME** variable. Therefore if that directory is not accessible and writable, the call to the external interpreter fails.

On Windows hosts, the shell first attempts to access the directory indicated by **HOME**, then **HOMEDRIVE/HOMEPATH**, and finally your installation directory. If any of these directories cannot be accessed, the call fails.

On UNIX hosts, the shell first attempts to access the directory indicated by **HOME**, and then your installation directory. If either directory cannot be accessed, the call fails.

6.7 Shell Initialization

When you execute an instance of the host shell, it begins by looking for a file called **windsh.tcl** in two places: first under

installDir/**workbench-3.x/foundation/build/resource/windsh**, and then in the directory specified by the **HOME** environment variable (if that environment variable is defined). In each of these directories, if the file exists, the shell reads and executes its contents as Tcl expressions before beginning to interact. You can use this file to automate any initialization steps you perform repeatedly.

You can also specify a Tcl expression to execute initially on the host shell command line, with the option **-e tcl_expression**. For example, you can test an initialization file before saving it as **windsh.tcl** using this option, as follows:

```
% windsh phobos -e "source c:\\fred\\tcltest"
```

6.7.1 Shell Initialization File

This file causes I/O for target routines called in the host shell to be directed to the target's standard I/O rather than to the host shell. It changes the default C++ strategy to automatic for this shell, sets a path for locating load modules, and causes modules not to be copied to the target server.

```
# Redirect Task I/O to WindSh
shConfig SH_GET_TASK_IO off
# Set C++ strategy
shConfig LD_CALL_XTORS on
# Set Load Path
shConfig LD_PATH "/home/username/project/app;/home/username/project/test"
# Let the Target Server directly access the module
shConfig LD_SEND_MODULES off
```

6.8 Tcl Scripting

From any of the host shell interpreters, a single command can be executed by any other interpreter by prefixing it with the appropriate command prefix. For example, when the host shell is in Tcl mode, the following command executes the single GDB mode command **continue**, leaving the host shell in Tcl mode:

```
tcl> shEval gdb continue
```

In this way, Tcl scripts executed by the host shell can issue GDB mode commands to perform OCD debugging operations, such as a reset and download operation.

The host shell can be made to execute a Tcl script by invoking it as follows:

```
%hostShell -m Tcl -q -s script-pathname
```

The `-q` option is optional; it tells the host shell not to echo script commands as they are executed.

6.8.1 Event Scripting

The host shell can also execute Tcl scripts when an event is encountered. The user indicates the script to execute and the event type that will trigger the script or the breakpoint ID that will trigger the script.

You must provide one of the following:

- The name of the procedure to execute.
- The name and location of the Tcl script to execute.
- The Tcl script to enter, typed interactively at the `tcl>` prompt in the host shell.

You may also enter the following optional information:

- The event that will trigger the execution. (By default, this is the stopped event.)
- Whether the handler is enabled or disabled. (By default, it is enabled.)
- For breakpoint events, the ID of the breakpoint that will trigger the handler. (If no ID is indicated, all breakpoints will trigger the handler.)
- Whether the default handler for the event should run after this new handler. (By default, the default handler will run.)

When the event is hit, you have two choices:

- Execute a script (in which case you should indicate the path to the script to execute.)
- Execute a Tcl routine (in which case you should have previously sourced the file containing the Tcl routine, either by using Tcl's source code or by adding some code to the shell's startup procedures.)

If no argument is specified, you may enter Tcl code to execute at the `tcl>` prompt in the host shell. The line **end** indicates the end of the script.

Your script should be written in Tcl. You have access to the target through the Gnu Debugger/Machine Interface (GDB/MI) synchronous commands and the API **gdb mi**. You can call the other interpreters by prefixing a command with the interpreter you wish to call for that command. For example, to call the C interpreter's **i()** command, you would write

shEval C i

You can copy the output from calls to other interpreters into Tcl variables, and manipulate them using standard Tcl.

If you wish to process the event that triggered the user handler, then your handler should take the form of a Tcl procedure having one argument. The argument sent to that procedure when the event type is encountered will be the triggering event itself. You may then process the event to extract the various data fields using standard Tcl string parsing procedures.

An example user handler:

```
proc breakpointHandler {evt} {
    puts "Breakpoint Hit event received $evt"
}
```

When registering the script, you may indicate whether the script is enabled (that is, whether it should be executed upon the next occurrence of the event specified) or you may register the script in disabled mode and enable it later, using an API.

When writing your script, Wind River recommends that you pay close attention to re-entrancy issues. If the script enters an infinite loop, you can exit the loop by typing **Ctrl+C**.

API Description

The host shell uses both command interpreter and GDB interpreter APIs for eventpoint scripting.

The command interpreter commands are:

- **handler add**
- **handler show**
- **handler remove**
- **handler enable**

For descriptions of these commands, see [4.12 Event Scripting Commands](#), p.89.

The GDB interpreter commands are:

- **display**
- **undisplay**
- **info display**
- **enable display**
- **disable display**
- **commands**

- **info commands**
- **enable commands**
- **disable commands**

For descriptions of these commands, see [5.11 Event Scripting](#), p.108.

7

Executing an OCD Reset and Download

- 7.1 Introduction 127
- 7.2 Set Target Registers 128
- 7.3 Play Back Firmware Commands 129
- 7.4 Reset One or More Cores 130
- 7.5 Download Executables and Data and Program Flash 130
- 7.6 Run the Target 132
- 7.7 Set a Hardware Breakpoint 132
- 7.8 Configure Target Memory Map 132
- 7.9 Pass Through Command to Firmware 134
- 7.10 Upload from Target Memory 134

7.1 Introduction

The host shell uses several commands to perform the equivalent of a Workbench on-chip debugging (OCD) reset and download operation. Rather than implement a single monolithic command having many options and optional arguments, several simpler commands are provided that can be used together to achieve a variety of goals.

If you need to invoke multiple commands repeatedly, you can create Tcl procedures.

The OCD reset and download workflow has the following steps:

1. Optionally play firmware commands to configure target registers.
2. Reset one or more cores, optionally initializing registers.
3. Optionally download one or more executables (optionally verifying the correctness of the download).
4. Optionally set the instruction pointer to an absolute address, the start address specified in the downloaded file, the address of a symbol (for example, **main**), or the address of a source line number (for example, **foo.c:123**).
5. Optionally play back firmware commands for post-reset target configuration.
6. Optionally set a breakpoint.
7. Optionally run the target.

All of these steps can be performed using GDB mode host shell commands, as described in this chapter.

7.2 Set Target Registers

Use the GDB mode **set** command to set target registers.

Syntax

set *\$register_name* = *option*

option can take any of the following four forms:

- *filename:line_number*

set *\$pc* = **foo.c:113**

Set the Program Counter to line 113 of the file **foo.c**.



NOTE: If the specified line number does not correspond to executable code, the host shell returns an error.

- *address*

```
set $pc = 0xffff000f0
```

Set the Program Counter to address 0xffff000f0.
- *program_symbol* (typically a function name)

```
set $pc = main
```

Set the Program Counter to the beginning of the function **main**.
- *program_symbol + constant*

```
set $pc = main + 0x60
```



NOTE: In most cases you can use a GDB mode command from a Tcl prompt by preceding it with the command **gdb**. However, because the **set** command is valid in both GDB mode and Tcl mode, the syntax

```
tcl> gdb set $pc= address
```

will return an error:

```
can't read "pc": no such variable
```

To avoid this problem, precede the **set** command's argument with a backslash:

```
tcl> gdb set \$pc= address
```

7.3 Play Back Firmware Commands

Use the GDB mode **wrsplayback** command to play a file of commands directly to the firmware.

Syntax

```
wrsplayback [ --quiet | --q ] pathname
```

pathname identifies the full path to an object file suitable for downloading to the target. This file must be accessible by the backend.

By default, the **wrsplayback** command returns human-readable status messages as they are received from the backend.

Use the option **--quiet** or **-q** to set the **wrsplayback** command not to return status messages.

With either option (that is, whether status messages are displayed to the user or not) the **wrsplayback** command waits for the playback to complete.

7.4 Reset One or More Cores

Use the GDB mode **wrsreset** command to reset one or more target cores.

Syntax

```
wrsreset [ --tied | -t ] [ --noinitregs | -n ] corename_1 [ corename_2 ... ]
```

The **-tied** option performs a tied reset of all specified cores.

The **-noinitregs** option specifies that target registers will not be initialized. If the **-noinitregs** option is omitted, target registers will be initialized by default.

7.5 Download Executables and Data and Program Flash

The GDB mode **wrsdownload** command has three separate syntaxes: one for downloading executables and raw data to the target; one for erasing flash memory on the target; and one for programming flash memory on the target. These three syntaxes cannot be used at the same time. (That is, you cannot specify more than one kind of operation in the arguments for one **wrsdownload** command.)

Erasing and programming flash are optional steps in a reset and download operation. However, if you use the erase and program syntaxes, you must issue the **wrsdownload** command three times: once to download code and data; once to erase flash; and once to program flash.

Download Executables and Data

First, use the **wrsdownload** command to download executables and data, using the following syntax.

Syntax 1

```
wrsdownload [ {--offset | -o} byte_offset ] [ {--modulename | -m} modulename ]  
            [--symbolonly | -s ] [ --nosymbols | n ] pathname
```

byte_offset is the byte offset to apply to the download.

modulename is the logical name for the object file.

The **--symbolonly** or **-s** option suppresses transfer of any data to target memory (but still loads symbols from *pathname*.)

The **--nosymbols** or **-n** option suppresses loading symbols from *pathname* (but still downloads the file to the target.)

pathname is the full path to the file to download.

Erase Flash Memory (Optional)

Erase a specified area of flash memory on the target, using the following syntax.

Syntax 2

```
wrsdownload {--eraseFlash | -e} start_address end_address
```

This command erases the content of flash memory from *start_address* to *end_address*.

Program Flash Memory (Optional)

Program flash using the following syntax:

Syntax 3

```
wrsdownload {--flash | -f} address pathname
```

This command loads the file at *pathname* to flash memory, beginning at *address*.

7.6 Run the Target

First, use the GDB mode **attach** command to attach to a specific thread or to system mode.

Example

```
attach system
```

Next, use the GDB mode **continue** command to make an OCD target begin execution at the current instruction pointer.

Syntax

```
continue
```

7.7 Set a Hardware Breakpoint

Set hardware breakpoints using the GDB mode **hbreak** command.

Syntax

```
hbreak [ address | file:line | symbol ] [ if condition ] [ --hx param=value ... ]  
        [ --sx param=expr ... ]
```

The **--hx** and **--sx** options correspond to the equivalent options to the GDB/MI command **-wrs-break-insert**, and *param* is any target-specific parameter that is valid in that GDB/MI command. The **hbreak** command will not validate target-specific parameters.

7.8 Configure Target Memory Map

Use the GDB mode **wrsmemmap** command to specify whether the debugger backend has read/write access to target memory, and where in target memory such access is allowed. This command only affects the backend. It does not affect memory map registers on the target, and does not cause a state change.

Syntax

```
wrsmemmap { --access | --noaccess }
           { offset size { --inv |
                           [ -r bitsize[ | bitsize]... ]
                           [ -w bitsize[ | bitsize]... ] | -rw bitsize[ | bitsize]... }
           } ...
```

This command will not check the validity of the numeric arguments, but it will communicate any errors reported by the backend.

Example 1

With the **--access** option set, all of memory is accessible to reads or writes; but within the range specified, there are modifications to the access privileges. So, for example, the command

```
wrsmemmap --access 0x14000 4000 -rw 8|16|32
```

allows you to read only the 4000-byte block of memory starting at address 0x14000, with accesses of 8, 16, or 32 bits.

Example 2

With the **--noaccess** option set, all of memory is inaccessible to reads or writes; but within the range specified there are modifications to the access privileges. The command

```
wrsmemmap --noaccess 0x14000 4000 -rw 16
```

allows you to read and write the 4000-byte block of memory starting at address 0x14000 with 16-bit accesses.

Example 3

The **--inv** option will invert the defined access setting without changing the undefined settings, as in the following example:

First, enter the command

```
wrsmemmap --access 0x14000 4000 -r 8|16|32
```

This allows read-only access starting at address 0x14000. You can now read that memory location but you cannot write to it. A read at 0x14000 provides data, but a write returns the error

```
GDB/MI Error: Invalid 'write' access for address '0x00014000'
```

Next, enter the command

```
wrsmemmap --access 0x14000 4000 --inv
```

This inverts the previously defined setting. The previously defined setting allowed a read, so the inverted setting does not. Now a read or a write at 0x14000 returns an access error.

7.9 Pass Through Command to Firmware

Use the GDB mode **wrspassthru** command to pass commands directly to the firmware without interpretation.

Syntax

wrspassthru *command*

command is an arbitrary sequence of space-separated strings. These strings are concatenated with a single space between each, and passed as a single command to the firmware.

Use this command to configure a target's flash memory by issuing configuration (CF) commands to the firmware. For information on the CF command, see the *Wind River Workbench On-Chip Debugging Command Reference*.

7.10 Upload from Target Memory

Use the GDB mode **wrsupload** command to upload data from target memory.

Syntax

wrsupload [{ **--style** | **-s** } *file_style*] [**--append** | **-a**] *start_address* *byte_count* *filename*

--style specifies the type of file to create. Currently the only supported *file_style* is RAWBIN. If you do not specify a style, the shell uses RAWBIN by default.

--append appends the uploaded data to *filename* instead of overwriting it.

PART II

VxWorks 653 Targets

8	Overview for VxWorks 653	137
9	Using the Host Shell with VxWorks 653	153
10	Using the C Interpreter with VxWorks 653	171
11	Using the Tcl Interpreter with VxWorks 653	209

8

Overview for VxWorks 653

- 8.1 Introduction 137
- 8.2 Starting the Host Shell 138
- 8.3 Switching Interpreters 140
- 8.4 Setting Shell Environment Variables 141
- 8.5 Path Mapping 144
- 8.6 Host Shell Features 145
- 8.7 Stopping the Host Shell 148
- 8.8 Host Shell Architecture 148

8.1 Introduction

This part of the document describes host shell support for VxWorks 653 targets.

You can use the host shell to invoke operating-system and application subroutines, and to monitor and debug applications in the VxWorks 653 kernel domain.

Because VxWorks 653 uses the protection domain paradigm, loading a module and calling a routine in a partition from the host shell (or from the target shell) is prohibited.

8.2 Starting the Host Shell

You can start the host shell from a command prompt or from within the Workbench GUI.

8.2.1 Starting the Host Shell from the Command Prompt

Setting Your Environment

Before launching the host shell, you must use the command **wrenv** to set up your environment. If you do not set your environment, the prompt returns the following error:

```
WIND_FOUNDATION_PATH must be set to start the Host Shell
```

To set your environment, enter the following command from your installation directory:

```
%wrenv -p target_OS_version
```

For example, to connect to a VxWorks 653 2.x target, enter

```
%wrenv -p vxworks653-2.x
```

Starting the Target Server

To start a target server, use the **tgtsvr** command. For example, to start a target server called **myTgtsvr** on a target with the IP address 123.456.78.90, enter the following command from your installation directory:

```
% tgtsvr 123.456.78.90 -n myTgtsvr
```

To see all available options for the **tgtsvr** command, enter **tgtsvr -h**.

Note that you must set up your environment with the **wrenv** command before using the **tgtsvr** command.

Starting the Shell

Once you have attached a target server to the target, start the shell using the **windsh** command.

```
%windsh [options] target_server
```

Host Shell Startup Options

[Table 8-1](#) summarizes startup options. For example, to connect to a running simulator, type the following:

```
%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 [8.4 Setting Shell Environment Variables](#), p.141).

Table 8-1 Host Shell Startup Options

Option	Description
-N, -noconnection	Specifies that the host shell will not connect to the backend server on startup. This allows a Tcl script to control the host shell.
-n, -noinit	Do not read home Tcl initialization file.
-T, -Tclmode	Start in Tcl mode.
-m[ode]	Indicates mode to start in: C (C) or Tcl (Tcl tcl TCL).
-v, -version	Display host shell version.
-h, -help	Print help.
-p, -poll	Sets event poll interval in milliseconds; the default is 200.
-e, -execute	Executes Tcl expression after initialization.
-c, -command	Executes expression and exits shell (batch mode).
-r, -root mappings	Root pathname mappings.
-ds[backend_server_session]	Debugger Server session to use.
-dp[backend_server_port]	Debugger Server port to use.

Table 8-1 **Host Shell Startup Options** (cont'd)

Option	Description
-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 <i>target</i>	Backend target definition name.

8.2.2 Starting the Host Shell from Workbench

If you have established a target connection, you can start the host shell from the **Remote Systems** view in Workbench. For creating target connections, see the *Wind River Workbench User's Guide: Connecting to Targets*.

In the **Remote Systems** view, right-click on your target connection name and select **Target Tools > Host Shell**. The **Host Shell Properties** dialog appears. You can specify startup options from [Table 8-1](#) in this dialog, or leave them at their defaults. Click **OK** to start the host shell.

8.3 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**:

- **C** to switch to the C interpreter. The prompt changes to **->**.
- **?** to switch to the Tcl interpreter. The prompt changes to **tcl>**.

8.3.1 Evaluating Statements in Different Modes

You can use the above commands to evaluate a statement native to a different interpreter for the one you are using.

To evaluate a statement native to another interpreter when using a VxWorks 653 target, precede the command with the special character from the list above. For example, to evaluate a C interpreter command from within the Tcl interpreter, type the following:

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

To evaluate a Tcl interpreter command from within the C interpreter, type the following:

```
-> ? set $pc = 0x14200
```

8.4 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 8-2](#) 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 the C interpreter if you precede the **shConfig** command with a question mark (**? shConfig variable option**).

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 8-2](#). Not all of the listed environment variables are valid for all targets.

Table 8-2 **Host Shell Environment Variables**

Variable	Result
ROOT_PATH_MAPPING	Indicates how host and target paths should be mapped to the host file system on which the backend 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 <i>/folk/user</i> is searched; no translation to another path is performed).
LINE_LENGTH	Indicates the maximum number of characters permitted in one line of the host shell's window.
STRING_FREE [manual automatic]	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 <code>strFree()</code> .
SEARCH_ALL_SYMBOLS [on off]	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]	Indicates the host shell's current interpreter mode and permits the user to switch from one mode to another.

Table 8-2 Host Shell Environment Variables (cont'd)

Variable	Result
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.
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.

Table 8-2 **Host Shell Environment Variables** (cont'd)

Variable	Result
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 .

8.5 Path Mapping

Since the host shell uses host paths to handle processes, a path substitution mechanism operates to send the right target path to the debugger server.

This mechanism converts a host path passed on the command line to a target path understandable by both the debugger framework and the target, but you must provide the host shell with additional information before it can perform the conversion. To perform this conversion, use the **ROOT_PATH_MAPPING** variable.

The **ROOT_PATH_MAPPING** Variable

The **ROOT_PATH_MAPPING** shell environment variable defines path substitution pairs of the form `[tgtpath1,hostpath1][tgtpath2,hostpath2]...`

In an example where the host path is **C:/mydirectory/myrt.p.vxe** and the target path is *hostname:/home/users/myName/mydirectory/myrt.p.vxe*, the command is:

```
-> ?
tcl> shConfig ROOT_PATH_MAPPING \[hostname:/home/users/myName/,C:/\]
```

(Note that square brackets must be escaped with a backslash.)

With this information, the host shell can compute the correct target path and send it to the debugger server. Note that the debugger server also needs this **ROOT_PATH_MAPPING** setting to retrieve the process file in order to parse the symbols, but the debugger server will send the path of this file directly to the target without any transformation by the host shell.

8.6 Host Shell Features

This section describes some of the features available in the host shell.

8.6.1 Symbol Matching

Start to type any target symbol name and then type **CTRL+D**. The shell automatically lists all symbols matching the pattern:

```
[coreOS] -> sem[CTRL+D]
semPxShow          semShow
Symbol matching in coreOS (PD ID 0x1efd40)
semTerminate       semTakeTbl          semTake          semSmTypeGetRtn
semSmShowRtn       semSmInfoRtn          semShowInit      semShow
semQPut            semQInit              semQGet          semQFlushDefer
semQFlush          semOTake              semMTake         semMPendQPut
semMLibInit        semMInit              semMGiveKernWork semMGiveKern
semMGiveForce      semMGive              semMCreate       semMCoreInit
semLibInit         semInvalid            semIntRestrict   semInfo
semGiveTbl         semGiveDeferTbl       semGiveDefer     semGive
semFlushTbl        semFlushDeferTbl      semFlushDefer    semFlush
semDestroy         semDelete             semClear         semClassId
semClass           semCTake              semCLibInit      semCInit
semCGiveDefer      semCGive              semCCreate       semCCoreInit
semBTake          semBLibInit           semBInit         semBGiveDefer
semBGive           semBCreate            semBCoreInit
[coreOS] -> sem
```

8.6.2 Directory and File Listing

You can also use **CTRL+D** to list all the files and directories that match a certain string. For example, to list all files and directories under **R:** that begin with **t**, type the following:

```
[coreOS] -> r:/t[CTRL+D]
```

```
t2cp2/
t3Keys/
taskSpawn
TDK-13671_001211_160045/
tornadoARMT2/
tornadoppc/
trgsh/
tsr152294src/
[coreOS] -> r:/t

t2i86config/
t3pen0107b/
TDK-13440_000504_104211_tar.gz
TORHELLO.WAV
tornadoi86t2/
torVars.bat
triggering/
tsr154738/
```

Directory and file listing is supported only in the host shell, not the target shell.

8.6.3 Target Symbol and Path Completion

Start to type any target symbol name or any existing directory name and then type **TAB**. The shell automatically completes the command or directory name for you. If there are multiple options, it prints them for you and then reprints your entry. You can add one or more letters and then type **TAB** again until the path or symbol is complete.

Symbol completion is supported in both the host shell and the target shell. Path completion is supported only in the host shell.

8.6.4 Synopsis Printing

Once you have typed the complete function name followed by a space, typing **CTRL+D** (not **TAB**) again prints the function synopsis, then reprints the function name ready for your input. (This function is not supported in the target shell.)

```
[coreOS] -> _taskIdDefault [CTRL+D]
taskIdDefault() - set the default task ID (WindSh)

int taskIdDefault
{
    int tid    /* user-supplied task ID; if 0, return default */
}

[coreOS] -> _taskIdDefault
```

If the routine exists on both host and target, the **hostShell** synopsis is printed. To print the target synopsis of a function, add the meta-character **@** before the function name.

You can extend the synopsis printing function to include your own routines. To do this, follow these steps:

1. Create the files that include the new routines following Wind River coding conventions.
2. Include these files in your project.
3. Add the filenames to the `DOC_FILES` macro in your makefile.
4. Go to the top of your project tree and run **make synopsis**:

```
[coreOS] -> cd installDir/vxworks-version/target/src/your_project
[coreOS] -> make synopsis
```

This adds your project file to the `installDir/vxworks-version/host/resource/synopsis` directory.

8.6.5 Data Conversion

Data conversion is available only in the C interpreter.

The shell prints all integers and characters in both decimal and hexadecimal, and if possible, as a character constant or a symbolic address and offset.

```
[coreOS] -> 68
value = 68 = 0x44 = 'D'
-> 0xf5de
value = 62942 = 0xf5de = _init + 0x52
[coreOS] -> 's'
value = 115 = 0x73 = 's'
```

8.6.6 Data Calculation

Data calculation is available only in the C interpreter.

Almost all C operators can be used for data calculation. Use “(” and “)” to force order of precedence.

```
[coreOS] -> (14 * 9) / 3
value = 42 = 0x2a = '*'
[coreOS] -> (0x1355 << 3) & 0x0f0f
value = 2568 = 0xa08
[coreOS] -> 4.3 * 5
value = 21.5
```

Calculations with Variables

```
[coreOS] -> (j + k) * 3
value = ...
```

```
[coreOS] -> *(j + 8 * k)
(address 0xn timer:: value = 0 = 0x0 (PD NAME: coreOS)
[coreOS] -> x = (val1 - val2) / val3
new symbol "x" added to symbol table
address = 0xn timer:: value = 0 = 0x0 (PD NAME: coreOS)
[coreOS] -># f = 1.41 * 2
new symbol "f" added to symbol table
f = 0x7d4746f8: value = 2.82 (PD NAME: coreOS)
```

Variable **f** gets an 8-byte floating point value.

8.7 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**. If the shell is not accepting input (for example, if it has lost connection to the target server) you can use the interrupt key (**CTRL+BREAK**.)

For more information, see the host shell reference pages. You can access these pages by opening Wind River Workbench and selecting **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference** and **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River VxWorks 653 Shell**.

8.8 Host Shell Architecture

The host shell integrates host and target resources in such a way that it creates the illusion of executing entirely on the target itself. However, most interactions with the shell exploit the resources of both host and target. [Table 8-3](#) shows how the shell distributes the interpretation and execution of the following simple expression:

```
-> dir = opendir ("/myDev/myFile")
```

Parsing the expression is the activity that controls overall execution, and dispatches the other execution activities. This takes place on the host, in the shell's

C interpreter, and continues until the entire expression is evaluated and the shell displays its result.

Table 8-3 Interpreting: `dir = opendir("/myDev/myFile")`

Host Shell (On Host)	Target Server and Symbol Table (On Host)	Target Agent (On Target)
Parse the string <code>"/myDev/myFile"</code> .	Allocate memory for the string; return address A .	Write <code>"/myDev/myFile"</code> ; return address A .
Parse the name <code>opendir</code> .	Look up <code>opendir</code> ; return address B .	
Parse the function call <code>B(A)</code> ; wait for the result.		Spawn a task to run <code>opendir()</code> at address A , passing address B as an argument, and signal result C when done.
	Retrieve C from target agent and pass it to host shell.	
Parse the symbol <code>dir</code> .	Look up <code>dir</code> (fails.)	
Request a new symbol table entry <code>dir</code> .	Define <code>dir</code> ; return symbol D .	
Parse the assignment <code>D=C</code> .	Allocate agent-pool memory for the value of <code>dir</code> .	Write the value of <code>dir</code> .

To avoid repetitive clutter, [Table 8-3](#) omits the following important steps, which must be carried out to link the activities in the three contexts (and two systems) shown in each column of the table:

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

The first access to server and agent is to allocate storage for the string **"/myDev/myFile"** on the target and store it there, so that subroutines such as **opendir()** have access to it. There is a pool of target memory reserved for host interactions. Because this pool is reserved, it can be managed from the host system. The server allocates the required memory, and informs the shell of its location; the shell then issues the requests to actually copy the string to that memory. This request reaches the agent on the target, and it writes the 14 bytes (including the terminating null) there.

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

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

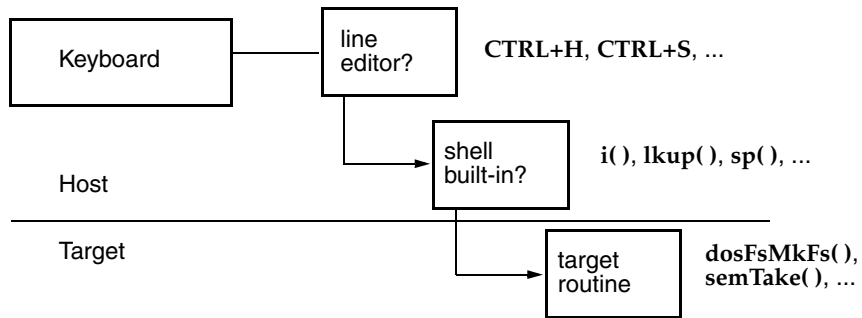
As before, the C interpreter looks up a symbol name (**dir**) through the target server; when the name turns out to be undefined, it instructs the target server to allocate storage for a new **int** and to make an entry pointing to it with the name **dir** in the symbol table. Again these symbol-table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

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

8.8.1 Layers of Interpretation

To the user, the host shell seems to be a seamless environment; but in fact, the characters you type in the shell go through several layers of interpretation, as illustrated by [Figure 8-1](#). First, input is examined for special editing keystrokes (described in [A. Using the Host Shell Line Editor](#).) Then as much interpretation as possible is done in the host shell itself. In particular, execution of any subroutine is first attempted in the shell itself; if a shell primitive with that name exists, it runs without any further checking. Only when a subroutine call does not match any shell primitives does the host shell call a target routine.

Figure 8-1 Layers of Interpretation in the Host Shell



For lists of host shell primitives, see the chapters in this book for each interpreter.

9

Using the Host Shell with VxWorks 653

- 9.1 Introduction 154
- 9.2 Domain Selection and Identification 154
- 9.3 Running Target Routines From the Shell 156
- 9.4 Function Calls from User Domains 156
- 9.5 Rebooting from the Host Shell 157
- 9.6 Task-Mode Debugging 158
- 9.7 Stack Tracing 161
- 9.8 Disassembler 161
- 9.9 Using the Host Shell for System-Mode Debugging 162
- 9.10 Interrupting a Shell Command 165
- 9.11 Working With Shared Library and Data Domains 167
- 9.12 Loading From the Shell 167

9.1 Introduction

This chapter describes some of the uses of the host shell when using a VxWorks 653 target.

9.2 Domain Selection and Identification

The domain in which shell commands are issued must be selected before the commands are issued. At startup, the default current working protection domain for the shell is the kernel domain, which by default is named **coreOS**. Use the colon operator to switch from one protection domain to another; that is, to set the “current working protection domain” for subsequent shell commands.

The command syntax for setting the protection domain is

:pdNameOrId

where the variable *pdNameOrId* can be the protection domain ID, the domain name, or a string matching the beginning of the domain name. It can also be a symbol name, which can be useful for scripting.

Example

In this example, use the **pdShow** command to display a list of available protection domains, then use the colon operator to change from the default kernel domain **coreOS** to the domain **part1**. In the domain **part1**, use the **pi** command to display a list of internal sub-tasks for that domain. Then use the colon operator to return to the kernel domain.

Note that the prompt includes the name of the current working protection domain.

[coreOS] -> **pdShow**

NAME	ID	TYPE	START ADRS	SIZE	L	PRI	H	PRI	TASK	CNT
coreOS	0x2026f100	KERNEL	0x20000000	0xb10000	255	0			15	
pos	0x2037a820	SYSTEM LIB	0x50000000	0x80000	0		0		0	
apexPartit >	0x2037d1e0	APPLICATION	0x28000000	0x300000	255		100		1	
posixParti >	0x203987a0	APPLICATION	0x28000000	0x300000	255		100		1	
part1	0x203c6d80	APPLICATION	0x28000000	0x300000	255		100		1	
part2	0x203f3418	APPLICATION	0x28000000	0x300000	255		100		1	
value = 0 = 0x0										

```
[coreOS] -> :part1
[part1] -> pi
```

NAME	ENTRY	TID	PRI	STATUS	PC	ERRNO
tSelGblF	0x50010930	0x282b4388	1	PEND	0x50020a62	0
tExcTask	0x50006c70	0x28294f40	0	PEND	0x50020a62	0
tLogTask	0x5000ba90	0x28274388	0	PEND	0x50020a62	0
tp3_1	0x2800020d	0x2825ad00	100	DELAY	0x50020a62	0
tp3_2	0x2800020d	0x2823ad00	100	DELAY	0x50020a62	0

```
value = 0 = 0x0
[part1] -> :coreOS
[coreOS] ->
```

Using the colon operator without an argument returns you to the kernel domain.

You can also use the colon operator with shell commands to identify the domain in which a symbol resides (symbol names can be duplicated across domains). The syntax is

symbol:pdNameOrId

To reference the symbol **task1** in protection domain **dExPd**, use **task1:dExPd** or **task1:0xcb430**.

```
[coreOS] -> 1 task1:dExPd
Disassembly for dExPd (PD ID 0xcb430)
task1:
0x20014078 9421ffe0 stwu    r1,-32(r1)
0x2001407c 7c0802a6 mfspr   r0,LR
0x20014080 93e1001c stw     r31,28(r1)
0x20014084 90010024 stw     r0,36(r1)
0x20014088 7c3f0b78 or      r31,r1,r1
0x2001408c 907f0008 stw     r3,8(r31)
0x20014090 3d202001 lis     r9,8193
0x20014094 3d602001 lis     r11,8193
0x20014098 3d402001 lis     r10,8193
0x2001409c 386a6040 addi    r3,r10,24640

value = 536952888 = 0x20014038
1
```



NOTE: The shell does not recognize the syntax *address:pdNameOrId*.

The following syntax allows you to switch temporarily to another domain to run a command, in this case **d**:

```
[coreOS] -> :pd1 d 0x20000000
```

This is the equivalent of running the following group of commands:

```
[coreOS] -> :pd1
[pd1] -> d 0x20000000
[pd1] -> :
[coreOS] ->
```

9.3 Running Target Routines From the Shell

All target routines are available from the host shell. This lets you test and debug your applications using all the host resources while having minimal impact on how the target performs and how the application behaves.

9.3.1 Invocations of VxWorks 653 Subroutines

```
[coreOS] -> taskSpawn ("tmyTask", 10, 0, 1000, myTask, fd1, 300)
value = ...
[coreOS] -> fd = open ("file", 0, 0)
new symbol "fd" added to "vsKernel" symbol table
fd = (...address_of_fd...): value = ...
```

9.4 Function Calls from User Domains

From a user protection domain, several different types of calls are possible.

- Call functions in the application that has been loaded in the user protection domain.
- Call links created by the loader.
- Call any entry points that can be found in the link path of the user domain. If the given function is not found in the user domain, the shell searches for the function in the link path. If the symbol is found in the link path, the function is called. For example:

```
[test] -> lkup ""
value = 0 = 0x0
[test] -> printf "Hello\n"
Hello
value = 6 = 0x6
[test] -> lkup ""
Symbol Table for test (PD ID 0x1f2198)
printf          0x200000a0 text link ---> vxWorks
value = 0 = 0x0
[test] ->
```

There are three different ways to call a function in a user domain. The only way to make breakable calls is to use **sp0**. One way to call a function in a user domain is to switch to that domain and call the function directly:

```
[coreOS] -> :appl
[appl] -> applStart
value = 0 = 0x0
```

Another method is to use the `:pdNameOrId` command syntax. This syntax changes the current working protection domain of the shell only for the execution of the given command. This example executes the function `applStart` in the domain `appl`:

```
[coreOS] -> :appl applStart
value = 0 = 0x0
[coreOS] ->
```

9.5 Rebooting from the Host Shell

In an interactive real-time development session, it is sometimes convenient to restart everything to make sure the target is in a known state. The host shell provides the `reboot()` command or **CTRL+X** for this purpose.

When you execute `reboot()` or type **CTRL+X**, the following reboot sequence occurs:

1. The shell displays a message to confirm rebooting has begun.
2. The target reboots.
3. The original target server on the host detects the target reboot and restarts itself, with the same configuration as previously. The target-server configuration option `-k[eePAlive]` (delay) governs the frequency of target pings; see the `tgtsvr` reference entry.
4. The shell detects the target-server restart and begins an automatic-restart sequence (initiated any time it loses contact with the target server for any reason), indicated with the following messages:

```
[coreOS] -> Rebooting...
Waiting to attach to target server          (press CTRL+C to stop) |
Waiting to attach to target agent          (press CTRL+C to stop)
|||||  |||||  |||||  |||||  |||||  |
```

- “Waiting to attach to target server”

The target server is restarting, the host shell is waiting for the attachment.

- “Waiting to attach to target agent”

The host shell is attached to the target server, but the target server is not yet attached to the target agent.

When the host shell establishes contact with the new target server, it displays the prompt and awaits your input.

9.6 Task-Mode Debugging

9.6.1 Task Breakpoints

The host shell allows you to set breakpoints in your code using the shell command **b()**. When setting a breakpoint it is important to remember that there may be several locations with the same symbol name (if the same module has been loaded into multiple protection domains) and multiple instances of the same address (if multiple user protection domains exist). For this reason you must select the protection domain in which you want to set the breakpoint before using **b()**.

- You can set a breakpoint on the kernel function **printf()** for all tasks and all protection domains. All breakable tasks referencing **printf()** will hit this breakpoint:

```
[coreOS] -> b printf
```

- If you specify a particular task, for example **task1**, only that task can hit the breakpoint:

```
[coreOS] -> b printf,task1
```

- If you specify a particular protection domain, only tasks owned by that domain can hit the breakpoint. In this example, only tasks owned by **pd1** will hit the breakpoint on the kernel function **printf()**:

```
[coreOS] -> b printf,pd1
```

- Finally, you can always restrict the breakpoint to a specific task as well as to a specific version of the routine:

```
[coreOS] -> b pd1App1Start:pd1,task1
```

- If you specify a particular shared library, only tasks owned by domains attached to that shared library will hit the breakpoint:

```
[coreOS] -> b printf:s1, s1
```

Once the breakpoint has been set, use the assembler level step and continue functions (**s()** and **c()**) to step through the code. When you step into a system call (from a user protection domain into a kernel function) you see two things:

- Whether the breakpoint has been set on the link using the colon separator:

```
[pd1] -> b printf:pd1
```

the linkage code that transfers control to the kernel is stepped through.

- If the breakpoint has been set on the link using the colon separator, the first instruction of the kernel function is displayed twice. This is normal. Closer examination of the register dump shows that the context has changed, causing the double display. The debugging sub-system has skipped over the exception handling code and gone directly to the kernel function. For more information on this, see the reference entry for **dbgLib**.

Invoking **b()** with no arguments lists all breakpoints and shows the protection domain in addition to other information.

```
[coreOS] -> b
ADDRESS      SYMBOL      TASK      PD      CNT  TYPE
-----
0x200000a0 printf      all      0x000fb0a8  0
value = 0 = 0x0
```

9.6.2 Protection Domain Breakpoints

A protection domain breakpoint stops all breakable tasks in the protection domain owning the task that encounters the breakpoint. The shell command **pdb()** sets or displays protection domain breakpoints.

```
pdb addr [context [count]]
```

- addr*

This argument can be specified numerically or symbolically with an optional offset.

- context*

A task or a protection domain. If context is a task, the breakpoint applies to the specified task. If context is a protection domain, the breakpoint applies to all breakable tasks owned by the specified protection domain. If context is zero or omitted, the breakpoint applies to all breakable tasks.

- count*

If *count* is zero or omitted, the breakpoint occurs every time it is hit. If *count* is specified, the break does not occur until an eligible task hits the breakpoint *count* + one times. (In other words, the breakpoint is ignored the first *count* times it is hit.)

There is also a hardware breakpoint routine, **pdbh()**. This routine stops all breakable tasks in the protection domain that owns the task that encounters the breakpoint.

Example

This example stops all tasks owned by the domain **tel** on the first **printf()** call performed by the application. The application is frequently calling **printf()**, and therefore the breakpoint will be hit shortly after it is set.

Using **i()** shows that all tasks in the user domain are in the **BREAK** state. The **BREAK** state is a new task status that distinguishes a task stopped by the debugger (marked **BREAK**.)

```
[tel] -> pdb printf
value = 0 = 0x0

[tel] -> b

ADDRESS      SYMBOL      TASK      PD      CNT TYPE
-----
0x20000100 printf      all      0x001f2ea8  0 PD break
value = 0 = 0x0

[tel] ->
PD Break at 0x20000100:printf      Task: 0x001f63f8 (tPlayer1)
                                   PD : 0x001f2ea8 (tel)

[tel] -> i

NAME      ENTRY      TID      PRI  STATUS      PC      ERRNO      PD ID
-----
tMgrTask  mgrTask      0x100008  0  PEND      0xaafe8  0  0xed7cc
tExcTask  excTask      0x1004c0  0  PEND      0xaafe8  0  0xed7cc
tLogTask  logTask      0x104940  0  PEND      0xaafe8  0  0xed7cc
tShell    shell        0x1dfa10  1  READY     0x77aac  0  0xed7cc
tWdbTask  0x880b0      0x1e0fb8  3  PEND      0x66040  0  0xed7cc
tNetTask  netTask      0x145628  50  READY     0x66040  0  0xed7cc
tOperator operator >    0x1ff6a8  150  PEND+BRK   0x66040  0  0x1f2ea8
tPlayer1  playerTask   0x1f63f8  200  BREAK     0x20000100  0  0x1f2ea8
tPlayer2  playerTask   0x1f66e0  200  BREAK     0x20014420  0  0x1f2ea8
tPlayer3  playerTask   0x1f4830  200  BREAK     0x20014420  0  0x1f2ea8
value = 0 = 0x0
[tel] ->
```


The **c()** command supports both tasks or protection domains, so it is possible to continue all tasks of a given protection domain. The following command continues all tasks of the domain **tel**:

```
[tel] -> c tel
```

9.7 Stack Tracing

The following example sets a breakpoint on the kernel function **fioFormatV()**. Then it calls **func()** in a user domain; **func()** makes a call to **printf()**, which uses **fioFormatV()**. The application is frequently calling **printf()**, and therefore the breakpoint will be hit shortly after it is set. Once the breakpoint is hit, you can trace the call using **tt()**.

```
[pd1] -> b fioFormatV:vxWorks  
value = 0 = 0x0
```

```
[pd1] -> b
```

ADDRESS	SYMBOL	TASK	PD	CNT	TYPE
0x00033324	fioFormatV	all	0x000fb0a8	0	

```
Break at 0x00033324:fioFormatV Task: 0x001fa5d0 (pdt2)  
PD : 0x000fb0a8 (pd1)  
Called function encountered a breakpoint (returning 0).  
value = 0 = 0x0
```

```
[pd1] -> ptt  
0x200140b0 func +38 : printf ()  
0x00032d54 printf +7c : fioFormatV ()  
value = 0 = 0x0
```

9.8 Disassembler

The routine **l()** allows you to specify the protection domain of the code to be disassembled.

```
-> 1 startPDs:myPd
Disassembly for myPd (PD ID 0xbef88)
      startPDs:
0x814a20  9421ff68  stwu      r1, -152(r1)
0x814a24  7c0802a6  mfspr     r0, LR
0x814a28  93410080  stw       r26, 128(r1)
0x814a2c  93610084  stw       r27, 132(r1)
0x814a30  93810088  stw       r28, 136(r1)
0x814a34  93a1008c  stw       r29, 140(r1)
0x814a38  93c10090  stw       r30, 144(r1)
0x814a3c  93e10094  stw       r31, 148(r1)
0x814a40  9001009c  stw       r0, 156(r1)
0x814a44  7c7b1b78  or        r27, r3, r3
value = 8473160 = 0x814a48 = startPDs + 0x28
```

10) now displays the ID of the protection domain where the text resides.

9.9 Using the Host Shell for System-Mode Debugging

The bulk of this chapter discusses the shell in its most frequent style of use: attached to a normally running VxWorks 653 system, through a target agent running in task mode. However, you can also use the shell with a system-mode agent. Entering system mode stops the entire target system: all tasks, the kernel, and all interrupt service requests (ISRs.) Similarly, breakpoints affect all tasks. One major shell feature is not available in system mode: you cannot execute expressions that call target-resident routines. You can still spawn tasks, but bear in mind that, because the entire system is stopped, a newly-spawned task can only execute when you allow the kernel to run long enough to schedule that task.

Depending on how the target agent is configured, you may be able to switch between system mode and task mode. When the agent supports mode switching, the following host shell commands control system mode:

- **sysSuspend()**
Enter system mode and stop the target system.
- **sysResume()**
Return to task mode and resume execution of the target system.

The following commands determine the state of the system and the agent:

- **agentModeShow()**

Show the agent mode (system or task).

- **sysStatusShow()**

Show the system context status (suspended or running).

The following shell commands behave differently in system mode:

- **b()**

Set a system-wide breakpoint; the system stops when this breakpoint is encountered by any task, or the kernel, or an ISR.

- **c()**

Resume execution of the entire system (but remain in system mode).

- **i()**

Display the state of the system context and the mode of the agent.

- **s()**

Single-step the entire system.

- **sp()**

Add a task to the execution queue. The task does not begin to execute until you continue the kernel or step through the task scheduler.

Example

This example uses system mode to debug a system interrupt.

In this case, **usrClock()** is attached to the system clock interrupt handler, which is called at each system clock tick when VxWorks is running. First suspend the system and confirm that it is suspended using either **i()** or **sysStatusShow()**.

```
[coreOS] -> sysSuspend
value = 0 = 0x0
[coreOS] ->
[coreOS] -> i
```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	3e8f98	0	PEND	47982	3e8ef4	0	0
tLogTask	_logTask	3e6670	0	PEND	47982	3e65c8	0	0
tWdbTask	0x3f024	398e04	3	PEND	405ac	398d50	30067	0
tNetTask	_netTask	3b39e0	50	PEND	405ac	3b3988	0	0

```
Agent mode      : Extern
System context  : Suspended
value = 0 = 0x0
[coreOS] ->
[coreOS] -> sysStatusShow
```

```
System context is suspended
value = 0 = 0x0
```

Next, set the system mode breakpoint on the entry point of the interrupt handler you want to debug. Since the target agent is running in system mode, the breakpoint will automatically be a system mode breakpoint, which you can confirm with the **b()** command. Resume the system using **c()** and wait for it to enter the interrupt handler and hit the breakpoint.

```
[coreOS] -> b usrClock
value = 0 = 0x0
[coreOS] -> b
0x00022d9a: _usrClock          Task:      SYSTEM Count:  0
value = 0 = 0x0
[coreOS] -> c
value = 0 = 0x0
[coreOS] ->
Break at 0x00022d9a: _usrClock          Task: SYSTEM
```

You can now debug the interrupt handler. For example, you can determine which task was running when system mode was entered using **taskIdCurrent()** and **i()**.

```
[coreOS] -> taskIdCurrent
_taskIdCurrent = 0x838d0: value = 3880092 = 0x3b349c
[coreOS] -> i
```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	3e8a54	0	PEND	4eb8c	3e89b4	0	0
tLogTask	_logTask	3e612c	0	PEND	4eb8c	3e6088	0	0
tWdbTask	0x44d54	389774	3	PEND	46cb6	3896c0	0	0
tNetTask	_netTask	3b349c	50	READY	46cb6	3b3444	0	0

```
Agent mode      : Extern
System context  : Suspended
value = 0 = 0x0
```

You can trace all the tasks except the one that was running when you placed the system in system mode and you can step through the interrupt handler.

```
[coreOS] -> tt tLogTask
4da78  _vxTaskEntry  +10 : _logTask (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
3f2bc  _logTask       +18 : _msgQReceive (3e62e4, 3e60dc, 20, ffffffff)
27e64  _msgQReceive    +1ba: _qJobGet ([3e62e8, ffffffff, 0, 0, 0, 0])
value = 0 = 0x0
[coreOS] -> l
_usrClock
00022d9a 4856          PEA          (A6)
00022d9c 2c4f          MOVEA .L     A7,A6
00022d9e 61ff 0002 3d8c BSR          _tickAnnounce
00022da4 4e5e          UNLK         A6
00022da6 4e75          RTS
00022da8 352e 3400      MOVE .W      (0x3400,A6),-(A2)
00022dac 4a75 6c20      TST .W       (0x20,A5,D6.L*4)
00022db0 3234 2031      MOVE .W      (0x31,A4,D2.W*1),D1
00022db4 3939 382c 2031 MOVE .W      0x382c2031,-(A4)
```

```

00022dba 343a 3337          MOVE .W      (0x3337, PC), D2
value = 0 = 0x0
[coreOS] -> s
d0  =      3e  d1  =      3700  d2  =      3000  d3  =      3b09dc
d4  =          0  d5  =          0  d6  =          0  d7  =          0
a0  =      230b8  a1  =      3b3318  a2  =      3b3324  a3  =      7e094
a4  =      38a7c0  a5  =          0  a6/fp =      bcb90  a7/sp =      bcb84
sr  =      2604  pc  =      230ba
          000230ba 2c4f          MOVEA .L      A7, A6
value = 0 = 0x0

```

Return to task mode and confirm that return by calling **i0**.

```

[coreOS] -> sysResume
value = 0 = 0x0
[coreOS] -> i

```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	3e8f98	0	PEND	47982	3e8ef4	0	0
tLogTask	_logTask	3e6670	0	PEND	47982	3e65c8	0	0
tWdbTask	0x3f024	398e04	3	READY	405ac	398d50	30067	0
tNetTask	_netTask	3b39e0	50	PEND	405ac	3b3988	0	0

```

value = 0 = 0x0

```

If you want to debug an application you have loaded dynamically, set an appropriate breakpoint and spawn a task which runs when you continue the system:

```

[coreOS] -> sysSuspend
value = 0 = 0x0
[coreOS] -> ml < test.o
Loading /view/didier.temp/vobs/wpwr/target/lib/objMC68040gntest//test.o
/
value = 400496 = 0x61c70 = _rn_addroute + 0x1d4
[coreOS] -> b address
value = 0 = 0x0
[coreOS] -> sp test
value = 0 = 0x0
[coreOS] -> c

```

The application breaks on address when the instruction at address is executed.

9.10 Interrupting a Shell Command

Occasionally it is desirable to abort the shell's evaluation of a statement. For example, an invoked routine may loop excessively, suspend, or wait on a semaphore. This may happen as the result of errors in arguments specified in the

invocation, errors in the implementation of the routine itself, or simply oversight as to the consequences of calling the routine.

To regain control of the shell in such cases, press the interrupt character on the keyboard, usually **CTRL+BREAK** or **CTRL+C**. This makes the shell stop waiting for a result and allows input of a new statement. Any remaining portions of the statement are discarded and the task that ran the function call is deleted.

Pressing **CTRL+BREAK** or **CTRL+C** is also necessary to regain control of the shell after calling a routine on the target that ends with **exit()** rather than **return()**.

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

```
[coreOS] -> sp taskSpawn,-4
task spawned: id = 0x20153b50, name = s1u0
value = 538262352 = 0x20153b50

Exception number 0xb: Task: 0x20153b50 (s1u0)
Segmentation violation
program counter:          0x20008740
next program counter:     0x20008744
processor status:         0xfe801001
access address:          0xffffffff
0x2002e038 vxTaskEntry    +c : taskSpawn (ffffffffff, 0, 0, 0, 0, 0)
0x20028fa8 taskSpawn      +7c : taskCreate (ffffffffff, 0, 0, 0, 0, 0)
0x2002909c taskCreate     +a8 : pdTaskCreate (0, fffffffffff, 0, 0, 0, 0)
0x200607fc pdTaskCreate   +864: taskInit (20155650, fffffffffff, 0, 805,
2046400 0, c000)
0x20029a78 taskInit       +7f8: objNameSet (20155650, fffffffffff, 0, 5, 8,
0)
0x2004e958 objNameSet     +24 : strlen (ffffffffff, 0, 0, 0, 0, 0)
[coreOS] ->
```

In cases like this, you do not need to type **CTRL+BREAK** to recover control of the shell; it automatically returns to the prompt, just as if you had interrupted. Whether you interrupt or the shell does it for you, you can proceed to investigate the cause of the suspension.

An interrupted routine may have left things in a state which was not cleared when you interrupted it. For instance, a routine may have taken a semaphore, which cannot be given automatically. Be sure to perform manual cleanup if you are going to continue the application from this point.

9.11 Working With Shared Library and Data Domains

When the current working protection domain of the shell is set to a shared library or shared data domain, it is possible to use the shell commands, but not to call functions. For example:

```
[sl] -> printf "Hello"
Error: Cannot call a function in a shared library.
[sl] -> :sd
[sd] -> printf "Hello"
Error: Cannot call a function from a shared memory region.
[sd] ->
```

Modifying data of the shared library has no impact on user protection domains that are already attached to the shared library.

9

9.12 Loading From the Shell



CAUTION: Dynamic loading and unloading is contrary to the static nature of the VxWorks 653 operating system. The feature described in this section is still supported for backwards compatibility with previous versions of VxWorks 653, but Wind River does not recommend its use.

9.12.1 Incremental Loading

During the development process, the shell allows you to test partially developed applications. An application need not be stored in one big object module. Instead, it can be broken into several smaller object modules corresponding to the various logical parts of the application. The loader allows you to load these smaller object modules independently for testing.

Dynamic loading and unloading is supported only in the VxWorks 653 kernel domain.

9.12.2 Dynamic Linking

Loading an object module is done in four steps:

1. Unload the existing version of the module, if any (host shell only).
2. Copy the file content (the sections) into target memory.
3. Relocate the sections at their installation addresses.
4. Link the new code with the code already in place.

The linking process uses symbols to establish the relation between the newly-loaded code and the code pre-existing in the system. An object module holds unresolved symbols for all external references to functions or data. The goal of the loader is to find matching symbols in the system and link each of the module's external references to the code that these symbols refer to.

The loader uses the link path to find the requested symbols. The link path specifies where and in which order of preference the loader should look for symbols. Each protection domain is created with the following default link path `..coreOS`. The leading dot represents the current domain, and **coreOS** is, by default, the name of the kernel protection domain. The loader interprets this link path as follows: look first in the current domain's symbol table; then, if the symbol is not found there, look in the kernel domain's symbol table.

If required, the link path of each domain can be changed by using **linkPathSet()**. For instance, if a shared library domain exists and must be considered when looking for symbols, an application domain's link path would become `..sharedCode:coreOS`, where *sharedCode* is the name of the shared library domain.



NOTE: It is not possible to look for symbols in another application protection domain. If this is necessary, the required code must be moved into a shared library domain.

If the loader cannot find the requested symbols, it issues an error message. If the out of order load feature is in use, it is possible to load additional code holding the missing symbols, and to have the previously loaded code re-linked accordingly.

9.12.3 Object Module Load Path

In order to download an object module dynamically to the target, both the host shell and the target server must be able to locate the file. If path naming conventions are different between the host shell and the target server, the two systems may both have access to the file, but it may be mounted with different pathnames. This situation arises often in environments where UNIX and Windows systems are networked together, because the path naming convention is different: the UNIX `/usr/fred/applic.o` may well correspond to the Windows

n:\fred\applic.o. If you encounter this problem, check to be sure the **LD_SEND_MODULES** variable of **shConfig** is set to **ON** or use the **LD_PATH** facility to tell the target server about the path known to the shell.

Example: Alternate Path Names

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

For more information on using **LD_PATH** and other **shConfig** facilities, see [8.4 Setting Shell Environment Variables](#), p.141.



NOTE: If you call **ml()** with an explicit argument list, any instances of the backslash character in Windows paths must be either be changed to forward slashes (**n:/fred/applic.o**) or else doubled (**n:\\fred\\applic.o**). If you supply the module name with the redirection symbol instead, no double backslashes are necessary.

Certain host shell commands and browser utilities imply dynamic downloads of auxiliary target-resident code. These subroutines fail in situations where the shell and target-server view of the file system is incompatible. To get around this problem, download the required routines explicitly from the host where the target server is running (or configure the routines statically into the VxWorks image). Once the supporting routines are on the target, any host can use the corresponding shell and browser utilities. [Table 9-1](#) lists the affected utilities.

Table 9-1 **Shell Utilities with Target-Resident Components**

Utility	Supporting Module
repeat()	repeatHost.o
period()	periodHost.o

9.12.4 Loader Defaults

Default loader behavior differs between the host shell and the target shell. In the host shell, **LD_COMMON_MATCH_ALL** is set to **ON** by default. Thus if you load two modules containing the same common symbols, the result is the creation of

one occurrence of the symbol related to the first module. The second module is linked to the first one through a common symbol.

In the target shell, **LD_COMMON_MATCH_ALL** is set to **OFF** by default. Thus if you load two modules containing the same common symbols, the result is the creation of two separate symbols of the same name, one for each module.

10

Using the C Interpreter with VxWorks 653

10.1 Introduction	172
10.2 Host and Target Shell Differences	172
10.3 Task References	174
10.4 Data Types	175
10.5 Expressions	177
10.6 Assignments	181
10.7 Comments	182
10.8 Strings	182
10.9 Ambiguity of Arrays and Pointers	183
10.10 Pointer Arithmetic	184
10.11 C Interpreter Limitations	185
10.12 Redirection in the C Interpreter	186
10.13 C++ Interpretation	189
10.14 C Interpreter Primitives	192
10.15 Resolving Name Conflicts Between Host and Target	207
10.16 Examples	207

10.1 Introduction

This chapter describes the behavior of the C interpreter when used with a VxWorks 653 target.

Note that C interpreter routine calls return 32-bit values only.

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.

The shell parses and evaluates its input one line at a time. A line may consist of a single shell statement or several shell statements separated by semicolons. A semicolon is not required on a line containing only a single statement. A statement cannot continue on multiple lines.

Shell statements are either C expressions or assignment statements. Either kind of statement may call host shell commands or target routines.

10.2 Host and Target Shell Differences

The host and target shells are almost identical. However, some of the commands (or routines) that you can execute from the shell are built into the host shell, rather than running as function calls on the target. These facilities parallel interactive utilities that can be linked into the target operating system itself. By using the host commands, you can minimize the impact on both target memory and performance.

Most of the shell commands correspond to similar routines that can be linked into the target operating system for use with the target-resident version of the shell. However, the target-resident routines differ in some details. For reference information on a shell command, be sure to consult the **windsh** reference entry.



CAUTION: Although there are usually entries with the same name in the VxWorks 653 API references, these entries describe related target routines, not the shell commands.

Table 10-1 shows the differences between the host and target shells. For additional information on the target shell, open Wind River Workbench and select **Help** >

Help Contents > Wind River Documentation > References > Host Tools > Wind River VxWorks 653 Shell > Routines Index.

For information on shell commands, see the reference entries for the commands by opening Wind River Workbench and selecting **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > Routines Index.**

Table 10-1 **Host Shell and Target Shell Differences – VxWorks 653**

Features	Available in Mode	Host Shell	Target Shell
Symbol completion	C mode, Tcl mode	Yes	Yes
Path completion	C mode, Tcl mode	Yes	No
Synopsis printing (CTRL+D)	C mode	Yes	No
HTML help (CTRL+W)	C mode	Yes	No

10.2.1 Protection Domain Breakpoints

pdb *addr* [*,context* [*,count* [*,quiet*]]]

addr

This argument can be specified numerically or symbolically with an optional offset.

context

A task or a protection domain. If context is a task, the breakpoint applies to the specified task. If context is a protection domain, the breakpoint applies to all breakable tasks owned by the specified protection domain. If context is zero or omitted, the breakpoint applies to all breakable tasks.

count

If *count* is zero or omitted, the breakpoint occurs every time it is hit. If *count* is specified, the break does not occur until an eligible task hits the breakpoint *count* plus one times. (In other words, the breakpoint is ignored the first *count* times it is hit.)

quiet

A target-shell-only argument. When set, it suppresses debugging information destined for the console when the breakpoint is hit. It is included to support

external source code debuggers that handle the breakpoint user interface themselves.

10.2.2 Function Calls in the Kernel Domain

This section applies only to the C interpreter.

When using the target shell, function calls are executed by the shell task in the kernel and are therefore unbreakable.

```
[coreOS] -> b printf
value = 0 = 0x0
[coreOS] -> printf "Hello world\n"
Hello world
value = 12 = 0xc
```

In order to make the call break, you must use **sp()** to spawn a task to run the function.

```
[coreOS] -> sp printf, "Hello world\n"
task spawned: tid = 0x1f4008, name = pdt1
value = 0 = 0x0
[coreOS] ->
Break at 0x00032cd8:printf Task: 0x001f4008 (pdt1)
PD : 0x000ed7cc (coreOS)
```

In the host shell, all function calls are breakable. This is because the host shell always creates a task to execute a function.

10.3 Task References

Most VxWorks 653 routines that take an argument representing a task require a task ID. However, when invoking routines interactively, specifying a task ID can be cumbersome, since the ID is an arbitrary and possibly lengthy number.

To accommodate interactive use, shell expressions can reference a task by either task ID or task name. The shell attempts to resolve a task argument to a task ID as follows: if no match is found in the symbol table for a task argument, the shell searches for the argument in the list of active tasks. When it finds a match, it substitutes the task name with its matching task ID. In symbol lookup, symbol names take precedence over task names.

When you enter any command, the shell attempts to match it in the following order: shell command, symbol, task name, and protection domain name.

By convention, task names are prefixed with a **u** for tasks started from the shell, and with a **pdt** for tasks started from the target itself. In addition, tasks started from a shell are prefixed by **s1**, **s2**, and so on to indicate which shell they were started from. This avoids name conflicts with entries in the symbol table. The names of system tasks and the default task names assigned when tasks are spawned use this convention. For example, tasks spawned with the shell command **sp0** in the first shell opened are given names such as **s1u0** and **s1u1**. Tasks spawned with the second shell opened have names such as **s2u0** and **s2u1**. Wind River recommends that you adopt a similar convention for tasks named in your applications.

The “Current” Task and Address

10

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

- A task hits a breakpoint or an exception trap. The current address is the address of the instruction that caused the break or exception.
- A task is single-stepped. The current address is the address of the next instruction to be executed.
- Any of the commands that use the current task or address are executed with a specific task parameter. The current address will be the address of the byte following the last byte that was displayed or disassembled.

10.4 Data Types

The most significant difference between the shell C-expression interpreter and a C compiler lies in the way that they handle data types. The shell does not accept any C declaration statements, and no data-type information is available in the symbol table. Instead, an expression’s type is determined by the types of its terms.

Unless you use explicit type-casting, the shell makes the following assumptions about data types:

- In an assignment statement, the type of the left hand side is determined by the type of the right hand side.
- If floating-point numbers and integers both appear in an arithmetic expression, the resulting type is a floating-point number.

Data types are assigned to various elements, as shown in [Table 10-2](#).

Table 10-2 **C Interpreter Data-Type Assumptions**

Element	Data Type
variable	int
variable used as a floating-point	double
return value of subroutine	int
constant with no decimal point	int/long
constant with decimal point	double

A constant or variable can be treated as a different type than what the shell assumes by explicitly specifying the type with the syntax of C type-casting. Functions that return values other than integers require a slightly different type-casting; see [10.5.4 Function Calls](#), p.179. [Table 10-3](#) shows the various data types available in the shell C interpreter, with examples of how they can be set and referenced.

Table 10-3 **Data Types in the C Interpreter**

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

Strings, or character arrays, are not treated as separate types in the C interpreter. To declare a string, set a variable to a string value. (Memory allocated for string constants is never freed by the shell.) For example:

```
-> ss = "any string"
```

The variable `ss` is a pointer to the string `any string`. To display `ss`, enter

```
-> d ss
```

The `d0` command displays the memory where `ss` is pointing. You can also use `printf()` to display strings.

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

```
*(70000 + 3 * 16)
```

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

10.5 Expressions

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

10.5.1 Literals

The shell interprets the literals in [Table 10-4](#) in the same way as the C compiler, with one addition: the shell also allows hex numbers to be preceded by `$` instead of `0x`.

Table 10-4 **Literals in the C Interpreter**

Literal	Example
decimal numbers	143967
octal numbers	017734
hex numbers	0xf3ba or \$f3ba
floating point numbers	555.555

Table 10-4 **Literals in the C Interpreter**

Literal	Example
character constants	'x' and '\$'
string constants	"This is a string."

10.5.2 **Variable References**

Shell expressions may contain references to variables whose names have been entered in the system symbol table. Unless a particular type is specified with a variable reference, the variable's value in an expression is the 4-byte value at the memory address obtained from the symbol table. It is an error if an identifier in an expression is not found in the symbol table, except in the case of assignment statements.

C compilers usually prefix all user-defined identifiers with an underscore, so that **myVar** is actually in the symbol table as **_myVar**. The identifier can be entered either way to the shell; the shell searches the symbol table for a match either with or without a prefixed underscore.

You can also access data in memory that does not have a symbolic name in the symbol table, as long as you know its address. To do this, apply the C indirection operator ****** to a constant. For example, ***0x10000** refers to the 4-byte integer value at memory address 10000 hex.

10.5.3 **Operators**

The shell interprets the operators in [Table 10-5](#) in the same way as the C compiler.

Table 10-5 **Operators in the C Interpreter**

Operator Type	Operators					
arithmetic	+	-	*	/	unary-	
relational	==	!=	<	>	<=	>=
shift	<<	>>				
logical		&&	!			

Table 10-5 Operators in the C Interpreter

Operator Type	Operators			
bitwise		&	~	^
address and indirection	&	*		

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

10.5.4 Function Calls

Shell expressions may contain calls to C functions (or C-compatible functions) whose names have been entered in the system symbol table; they may also contain function calls to host shell commands that execute on the host.

The shell executes such function calls in tasks spawned for the purpose, with the specified arguments and default task parameters; if the task parameters make a difference, you can call `taskSpawn()` instead of calling functions from the shell directly. The value of a function call is the 4-byte integer value returned by the function. The shell assumes that all functions return integers. If a function returns a value other than an integer, the shell must know the data type being returned before the function is invoked. This requires a slightly unusual syntax because you must cast the function, not its return value. For example:

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



NOTE: The examples in this book assume you are using the default shell prompts. However, you can change the C interpreter prompt to anything you like using the `shellPromptSet()` routine.

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

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

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

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

```
4 + myFunc2 ()
```

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

```
4 + myFunc2
```

An important exception to this occurs when the function name is the very first item encountered in a statement. See [10.5.5 Arguments to Commands](#), p.180.

Shell expressions can also contain calls to functions that do not have a symbolic name in the symbol table, but whose addresses are known to you. To do this, simply supply the address in place of the function name. Thus the following expression calls a parameter-less function whose entry point is at address 10000 hex:

```
0x10000 ()
```

10.5.5 Arguments to Commands

In practice, most statements input to the shell are function calls. To simplify this use of the shell, an important exception is allowed to the standard expression syntax required by C. When a function name is the very first item encountered in a shell statement, the parentheses surrounding the function's arguments may be omitted. Thus the following shell statements are synonymous:

```
-> rename ("oldname", "newname")  
-> rename "oldname", "newname"
```

as are:

```
->evtBufferAddress ( )  
->evtBufferAddress
```

However, note that if you wish to assign the result to a variable, the function call cannot be the first item in the shell statement—thus, the syntactic exception above does not apply. The following captures the address, not the return value, of **evtBufferAddress()**:

```
-> value = evtBufferAddress
```

10.6 Assignments

The shell C interpreter accepts assignment statements in the form:

addressExpression = expression

The left side of an expression must evaluate to an addressable entity; that is, a legal C value.

10.6.1 Typing and Assignment

The data type of the left side is determined by the type of the right side. If the right side does not contain any floating-point constants or non-integer type-casts, then the type of the left side will be an integer. The value of the right side of the assignment is put at the address provided by the left side. For example, the following assignment sets the 4-byte integer variable x to 0x1000:

```
-> x = 0x1000
```

The following assignment sets the 4-byte integer value at memory address 0x1000 to the current value of x:

```
-> *0x1000 = x
```

The following compound assignment adds 300 to the 4-byte integer variable x:

```
-> x += 300
```

The following adds 300 to the 4-byte integer at address 0x1000:

```
-> *0x1000 += 300
```

The following compound operators are available:

```
++  *=  &=
--  /=  |=
+=  %=  ^=
-=
```

10.6.2 Automatic Creation of New Variables

New variables can be created automatically by assigning a value to an undefined identifier (one not already in the symbol table) with an assignment statement.

When the shell encounters such an assignment, it allocates space for the variable and enters the new identifier in the symbol table along with the address of the newly allocated variable. The new variable is set to the value and type of the

right-side expression of the assignment statement. The shell prints a message indicating that a new variable has been allocated and assigned the specified value.

For example, if the identifier **fd** is not currently in the symbol table, the following statement creates a new variable named **fd** and assigns to it the result of the function call:

```
-> fd = open ("file", 0)
```

10.7 Comments

The shell allows two kinds of comments.

First, comments of the form `/* ... */` can be included anywhere on a shell input line. These comments are simply discarded, and the rest of the input line evaluated as usual.

Second, any line whose first non-blank character is `#` is ignored completely.

10.8 Strings

When the shell encounters a string literal (`"..."`) in an expression, it allocates space for the string including the null-byte string terminator. The value of the literal is the address of the string in the newly allocated storage. For instance, the following expression allocates 12 bytes from the target-agent memory pool, enters the string in those 12 bytes (including the null terminator), and assigns the address of the string to `x`:

```
-> x = "hello world"
```

Even when a string literal is not assigned to a symbol, memory is still permanently allocated for it. For example, the following uses 12 bytes of memory that are never freed:

```
-> printf ("hello world")
```

If strings were only temporarily allocated, and a string literal were passed to a routine being spawned as a task, then by the time the task executed and attempted to access the string, the shell would have already released, possibly even reused, the temporary storage where the string was held.

After extended development sessions, the cumulative memory used for strings may be noticeable. If this becomes a problem, restart your target server.

10.8.1 Strings and Pathnames

In VxWorks 653, the directory and file segments of pathnames (for target-resident files and devices) are separated with the slash character (/). This presents no difficulty when subroutines require a pathname argument, because the / character has no special meaning in C strings.

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

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

You can use the shell's **ml()** command with all of these variations of pathnames. The following **ml()** invocations are all correct and equivalent:

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

10.9 Ambiguity of Arrays and Pointers

In a C expression, a non-subscripted reference to an array has a special meaning, namely the address of the first element of the array. The shell, to be compatible, should use the address obtained from the symbol table as the value of such a reference, rather than the contents of memory at that address. Unfortunately, the information that the identifier is an array, like all data type information, is not available after compilation. For example, if a module contains the following:

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

you might be tempted to enter a shell expression as in Example 1.

Example 1

```
-> printf (string)
```

While this would be correct in C, the shell will pass the first 4 bytes of the string itself to **printf()**, instead of the address of the string. To correct this, the shell expression must explicitly take the address of the identifier, as in Example 2.

Example 2

```
-> printf (&string)
```

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

```
char *string = "hello";
```

then to a compiler, Example 1 would be correct and Example 2 would be wrong. This is especially confusing since C allows pointers to be subscripted exactly like arrays, so that the value of **string[0]** would be “h” in either of the above declarations.

Bear in mind that array references and pointer references in shell expressions are different from their C counterparts. In particular, array references require an explicit application of the address operator **&**.

10.10 Pointer Arithmetic

While the C language treats pointer arithmetic specially, the shell C interpreter does not, because it treats all non-type-cast variables as 4-byte integers.

In the shell, pointer arithmetic is no different than integer arithmetic. Pointer arithmetic is valid, but it does not take into account the size of the data pointed to. Consider the following example:

```
-> *(myPtr + 4) = 5
```

Assume that the value of **myPtr** is 0x1000. In C, if **myPtr** is a pointer to a type char, this would put the value 5 in the byte at address at 0x1004. If **myPtr** is a pointer to a 4-byte integer, the 4-byte value 0x00000005 would go into bytes 0x1010–0x1013.

The shell, on the other hand, treats variables as integers, and therefore would put the 4-byte value 0x00000005 in bytes 0x1004–0x1007.

10.11 C Interpreter Limitations

The C interpreter in the shell is not a complete interpreter for the C language. The following C features are not present in the host shell.

- Control structures

The shell interprets only C expressions (and comments). The shell does not support C control structures such as **if**, **goto**, and **switch** statements, or **do**, **while**, and **for** loops. Control structures are rarely needed during shell interaction. If you do come across a situation that requires a control structure, you can use the Tcl interface to the shell instead of using its C interpreter directly.

- Compound or derived types

No compound types (**struct** or **union** types) or derived types (**typedef**) are recognized in the shell C interpreter.

- Macros

No C preprocessor macros (or any other preprocessor facilities) are available in the shell. For constant macros, you can define variables in the shell with similar names to the macros. You can automate the effort of defining any variables you need repeatedly, by using an initialization script.

For control structures, or display and manipulation of types that are not supported in the shell, you might also consider writing auxiliary subroutines to provide these services during development; you can call such subroutines at will from the shell, and later omit them from your final application.

10.12 Redirection in the C Interpreter

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

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

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

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

10.12.1 Ambiguity Between Redirection and C Operators

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

```
-> x > y
```

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

```
-> x; > y  
-> x > y;
```

The first line prints the value of the variable **x** to the output stream **y**. The second line prints on standard output the value of the expression “**x** greater than **y**.”

10.12.2 The Nature of Redirection

The redirection mechanism of the host shell is fundamentally different from that of the Windows command shell, although the syntax and terminology are similar.

In the host shell, redirecting input or output affects only a command executed from the shell. In particular, this redirection is not inherited by any tasks started while output is redirected.

For example, you might be tempted to specify redirection streams when spawning a routine as a task, intending to send the output of **printf()** calls in the new task to an output stream, while leaving the shell's I/O directed at the virtual console. This stratagem does not work. For example, the shell input line:

```
-> taskSpawn (...myFunc...) > output
```

momentarily redirects the shell standard output during the brief execution of the spawn routine, but does not affect the I/O of the resulting task.

To redirect the input or output streams of a particular task, call **ioTaskStdSet()** once the task exists.

10.12.3 Scripts: Redirecting Shell I/O

A special case of I/O redirection concerns the I/O of the shell itself; that is, redirection of the streams the shell's input is read from, and its output is written to. The syntax for this is simply the usual redirection specification, on a line that contains no other expressions.

The typical use of this mechanism is to have the shell read and execute lines from a file. For example, the input lines:

```
-> <startup
```

or

```
-> < c:\fred\startup
```

cause the shell to read and execute the commands in the file **startup**, either on the current working directory (in the first example) or explicitly on the complete pathname (in the second example.) If your working directory is **\fred**, then the two examples are equivalent.

Such command files are called *scripts*. Scripts are processed exactly like input from an interactive terminal. After reaching the end of the script file, the shell returns to processing I/O from the original streams.

During execution of a script, the shell displays each command as well as any output from that command. You can change this by invoking the shell with the **-q** option (see [Host Shell Startup Options](#), p.139.)

An easy way to create a shell script is from a list of commands you have just executed in the shell. The history command **h()** prints a list of the last 20 shell commands. The following creates the file **c:\tmp\script** with the current shell history:

```
-> h > c:\tmp\script
```

The command numbers must be deleted from this file before using it as a shell script.

Scripts can also be nested. That is, scripts can contain shell input redirections that cause the shell to process other scripts.



CAUTION: Input and output redirection must refer to files on a host file system. If you have a local file system on your target, files that reside there are available to target-resident subroutines, but not to the shell (unless you export them from the target using NFS, and mount them on your host).



CAUTION: Wind River recommends that you set the shell environment variable **SH_GET_TASK_IO** to **OFF** before you use redirection of input from scripts, or before you copy and paste blocks of commands to the shell command line. Otherwise commands might be taken as input for a command that precedes them, and thus get lost.

C Interpreter Startup Scripts

Host shell scripts can be useful for setting up your working environment. You can run a startup script through the shell C interpreter by specifying its name with the **-s** option. For example:

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

You can also use the **-e** option to run a Tcl expression at startup, or place Tcl initialization in **windsh.tcl** under your home directory.

You can use startup scripts for setting system parameters to personal preferences: defining variables, specifying the target's working directory, and so forth. They can also be useful for tailoring the configuration of your system without having to rebuild the image. For example:

- creating additional devices
- loading and starting up application modules

- adding a complete set of network host names and routes
- setting NFS parameters and mounting NFS partitions

10.13 C++ Interpretation

Workbench supports both C and C++ as development languages. For information about C++ development, see the *VxWorks 653 Programmer's Guide: Developing C++ Applications*.

Because C and C++ expressions are so similar, the host shell C-expression interpreter supports many C++ expressions. The facilities explained in this chapter are all available regardless of whether your source language is C or C++. In addition, there are a few special facilities for C++ extensions. This section describes those extensions.

The host shell is not a complete interpreter for C++ expressions. In particular:

- The shell has no information about user-defined types.
- There is no support for the `::` operator.
- Constructors, destructors, and operator functions cannot be called directly from the shell.
- Member functions cannot be called with the `.` or `->` operators.

To exercise C++ facilities that are missing from the C interpreter, you can compile and download routines that encapsulate the special C++ syntax.

10.13.1 Overloaded Function Names

If you have several C++ functions with the same name, distinguished by their argument lists, call any of them as usual with the name they share. When the shell detects the fact that several functions exist with the specified name, it lists them in an interactive dialog, printing the matching functions' signatures so that you can recall the different versions and make a choice among them.

You make your choice by entering the number of the desired function. If you make an invalid choice, the list is repeated and you are prompted to choose again. If you enter 0 (zero), the shell stops evaluating the current command and prints a message like the following:

undefined symbol: *your_function_name*

This can be useful, for example, if you misspelled the function name and you want to abandon the interactive dialog. However, because the shell is an interpreter, not a compiler, portions of the expression may already have executed (perhaps with side effects) before you abandon execution in this way.

The following example shows how the support for overloaded names works. In this example, there are four versions of a function called **xmin()**. Each version of **xmin()** returns at least two arguments, but each version takes arguments of different types.

```
-> 1 xmin
"xmin" is overloaded - Please select:
  1: _xmin(double,double)
  2: _xmin(long,long)
  3: _xmin(int,int)
  4: _xmin(float,float)
Enter <number> to select, anything else to stop: 1
      _xmin(double,double):
3fe710 4e56 0000      LINK      .W      A6,#0
3fe714 f22e 5400 0008  FMOVE     .D      (0x8,A6),F0
3fe71a f22e 5438 0010  FCMP      .D      (0x10,A6),F0
3fe720 f295 0008      FB        .W      #0x8f22e
3fe724 f22e 5400 0010  FMOVE     .D      (0x10,A6),F0
3fe72a f227 7400      FMOVE     .D      F0,-(A7)
3fe72e 201f           MOVE      .L      (A7)+,D0
3fe730 221f           MOVE      .L      (A7)+,D1
3fe732 6000 0002      BRA        0x003fe736
3fe736 4e5e           UNLK      A6
value = 4187960 = 0x3fe738 = _xmin(double,double) + 0x28

-> 1 xmin
"xmin" is overloaded - Please select:
  1: _xmin(double,double)
  2: _xmin(long,long)
  3: _xmin(int,int)
  4: _xmin(float,float)
Enter <number> to select, anything else to stop: 3
      _xmin(int,int):
3fe73a 4e56 0000      LINK      .W      A6,#0
3fe73e 202e 0008      MOVE      .L      (0x8,A6),D0
3fe742 b0ae 000c      CMP       .L      (0xc,A6),D0
3fe746 6f04           BLE        0x003fe74c
3fe748 202e 000c      MOVE      .L      (0xc,A6),D0
3fe74c 6000 0002      BRA        0x003fe750
3fe750 4e5e           UNLK      A6
3fe752 4e75           RTS
      _xmin(long,long):
3fe7544e560000      LINK      .W      A6,#0
3fe758202e0008      MOVE      .L      (0x8,A6),D0
value = 4187996 = 0x3fe75c = _xmin(long,long) + 0x8
```

In this example, the user calls the disassembler to list the instructions for `xmin0`, then selects the version that computes the minimum of two **double** values. Next, the user invokes the disassembler again, this time selecting the version that computes the minimum of two **int** values. Note that a different routine is disassembled in each case.

10.13.2 Automatic Name Demangling

Many shell debugging and system information functions display addresses symbolically (for example, the `10` routine). This might be confusing for C++, because compilers encode a function's class membership (if any) and the type and number of the function's arguments in the function's linkage name. The encoding is meant to be efficient for development tools, but not necessarily convenient for human comprehension. This technique is commonly known as *name mangling* and can be a source of frustration when the mangled names are exposed to the developer.

To avoid this confusion, the debugging and system information routines in the host shell print C++ function names in a demangled representation. Whenever the shell prints an address symbolically, it checks whether the name has been mangled. If it has, the name is demangled (complete with the function's class name, if any, and the type of each of the function's arguments) and printed.

The following example shows the demangled output when `lkup0` displays the addresses of the `xmin0` functions mentioned in the previous section.

```
-> lkup "xmin"
_xmin(double,double) 0x003fe710 text (templex.out)
_xmin(long,long)     0x003fe754 text (templex.out)
_xmin(int,int)        0x003fe73a text (templex.out)
_xmin(float,float)    0x003fe6ee text (templex.out)
value = 0 = 0x0
```

10.14 C Interpreter Primitives

10.14.1 Managing Tasks

[Table 10-6](#) summarizes the commands that manage tasks. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Table 10-6 Task Management Commands

Call	Description
<code>sp()</code>	Spawn a task with default parameters.
<code>sps()</code>	Spawn a task, but leave it suspended.
<code>tr()</code>	Resume a suspended task.
<code>ts()</code>	Suspend a task.
<code>td()</code>	Delete a task.
<code>period()</code>	Spawn a task with entry point periodHost to call a function periodically.
<code>repeat()</code>	Spawn a task with entry point repeatHost to call a function repeatedly.
<code>taskIdDefault()</code>	Set or report the default (current) task ID. (For information on how the current task is established and used, see The “Current” Task and Address , p.175.)
<code>trace()</code>	Trace the execution of a task or object).

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()**. (This download is not always reliable with remote target servers.) 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.

10.14.2 Task Information

Table 10-7 summarizes the host shell commands that report task information. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh**.)

Table 10-7 Task Information Commands

Call	Description
i()	Display system information. This command gives a snapshot of what tasks are in the system, and some information about each of them, such as state, PC, and protection domain ID. To save memory, this command queries the target repeatedly; thus, it may occasionally give an inconsistent snapshot.
iStrict()	Display the same information as i() , but query target system information only once. At the expense of consuming more intermediate memory, this guarantees an accurate snapshot.
ti()	Display task information. This command gives all the information contained in a task's task control block (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 <i>task</i> is 0 (or the argument is omitted), the current task is reported.
w()	Print 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 a specified task if the argument is given.
tw()	Print information about the object the given task is pending on. This routine calls taskWaitShow() on the given task in verbose mode.
checkStack()	Show 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.

Table 10-7 Task Information Commands

Call	Description
tt()	Display a stack trace.
taskIdFigure()	Report 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:

```
[coreOS] -> i
NAME      ENTRY      TID      PRI STATUS      PC      ERRNO      PD ID
-----
tMgrTask  mgrTask     0xeabfc  0  PEND      0xa09f0  0      0xd7cf8
tExcTask  excTask     0xe8a30  0  PEND      0xa09f0  0      0xd7cf8
tLogTask  logTask     0xee018  0  PEND      0xa09f0  0      0xd7cf8
tShell    shell       0x14c018 1  PEND      0xb26bc  0      0xd7cf8
tWdbTask  wdbTask     0x19b478 3  READY     0xb2a24  0      0xd7cf8
tNetTask  netTask     0xf0a70  50 READY     0x561b4  0      0xd7cf8
value = 0 = 0x0
```

The **w()** and **tw()** commands allow you to see what object a task is pending on. **w()** displays summary information for all tasks, while **tw()** displays object information for a specific task. Note that the **OBJ_NAME** field is used only for objects that have a symbolic name associated with the address of their structure.

```
[coreOS] -> w
NAME      ENTRY      TID      STATUS  DELAY  OBJ_TYPE  OBJ_ID  OBJ_NAME
-----
tMgrTask  mgrTask     0xeabfc  PEND    0  MSG_Q(R)  0xeab08 N/A
tExcTask  excTask     0xe8a30  PEND    0  MSG_Q(R)  0xe8824 N/A
tLogTask  logTask     0xee018  PEND    0  MSG_Q(R)  0xec63c N/A
tShell    shell       0x14c018  PEND    0  SEM_B     0xec01c N/A
tWdbTask  wdbTask     0x19b478  READY   0
tNetTask  netTask     0xf0a70  READY   0
u0        smtask1     0x36cc2c  PEND    0  MSG_Q_S(S)  0xf0b61 N/A
u1        smtask3     0x367c54  PEND    0  MSG_Q_S(S)  0xf0b61 N/A
u3        taskB       0x362c7c  PEND    0  SEM_B     0xfd378 _mySem2
u6        smtask1     0x35dca4  PEND    0  MSG_Q_S(S)  0xf0ae1 N/A
u9        task3B      0x358ccc  PEND    0  MSG_Q(S)    0xfcf1c _myMsgQ
value = 0 = 0x0
[coreOS] -> tw tLogTask
NAME      ENTRY      TID      STATUS  DELAY  OBJ_TYPE  OBJ_ID  OBJ_NAME
-----
tLogTask  logTask     0xee018  PEND    0      MSG_Q(R)  0xec63c  N/A

Message Queue Id      : 0xec63c
Task Queueing         : FIFO
Message Byte Len      : 32
Messages Max          : 50
Messages Queued       : 0
```

```
Messages Queued High : 0
Receivers Blocked    : 1
Send Timeouts        : 0
Receive Timeouts     : 0

Receivers Blocked:

NAME      TID      PRI  TIMEOUT
-----
tLogTask  0xee018  0      0

value = 0 = 0x0
```

10.14.3 Displaying System Information

Table 10-8 summarizes the host shell commands that display information from the symbol table, from the target system, and from the shell itself. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

10

Table 10-8 System Information Commands

Call	Description
devs()	List all devices known on the target system.
lkup()	List symbols from the symbol table.
lkAddr()	List symbols whose values are near a specified value.
d()	Display target memory. You can specify a starting address, size of memory units, and number of units to display.
l()	Disassemble and display a specified number of instructions; optionally specify protection domain.
printErrno()	Describe the most recent error status value.
version()	Print operating system version information.
cd()	Change the working directory on the host (does not affect target.)
ls()	List files in the host working directory.
pwd()	Display the current host working directory.

Table 10-8 **System Information Commands**

Call	Description
help()	Display a summary of shell commands.
h()	Display or set the size of shell history.
shellHistory()	Display or set the size of shell history.
shellPromptSet()	Change the C interpreter shell prompt.
printLogo()	Display 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 that are already loaded in the current protection domain. For example, to display a list containing routines and declared variables with names containing the string *dsm*, do the following:

```
-> lkup "dsm"
Symbol Table for coreOS (PD ID 0xd7cf8)
dsmNbytes          0x0001eed0 text
dsmInst            0x0001ee78 text
value = 0 = 0x0
```

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 can use a regular expression, as in the following:

```
-> lkup "[dD]sm"
Symbol Table for coreOS (PD ID 0xd7cf8)
dsmNbytes          0x0001eed0 text
dsmInst            0x0001ee78 text
_dbgDsmInstRtn     0x000cfc3c data
value = 0 = 0x0
->
```

Regular-expression searches of the symbol table can be as simple or elaborate as required. For example, the following simple regular expression displays the names of three internal VxWorks semaphore functions:

```
-> lkup "sem.Take"
Symbol Table for coreOS (PD ID 0xd7cf8)
semOTake           0x000b2794 text  entry
semMTake           0x000b2960 text  entry
semCTake           0x000b27ec text  entry
semBTake           0x000b2480 text  entry
value = 0 = 0x0
->
```

To see a symbol in another protection domain, it is necessary to specify the protection domain. The following command displays all the symbols in the protection domain **ldTest**. It also uses the option **0x8** to add pending symbols to the displayed output. For more information on pending symbols, see the *Wind River Workbench User's Guide, VxWorks 653 Version: Tools*.

```
-> lkup "",0x8,ldTest
Symbol Table for ldTest (PD ID 0xca430)
gooFunc      0x02414fc8 text pend!      (module_3.o)
fooVar       0x02415fe0 data           (module_1.o)
printf       0x02400080 text link      ---> kernel
looFunc      0x02414f50 text           (module_3.o)
fooFunc      0x02414ed8 text           (module_1.o)
[coreOS] ->
```

You can achieve the same result by switching to the **ldTest** protection domain and then not specifying the domain in the **lkup()** command.

```
-> :ldTest
[ldTest] -> lkup "",0x8
gooFunc      0x02414fc8 text pend!      (module_3.o)
fooVar       0x02415fe0 data           (module_1.o)
printf       0x02400080 text link      ---> kernel
looFunc      0x02414f50 text           (module_3.o)
fooFunc      0x02414ed8 text           (module_1.o)
->
```

Table 10-9 lists options for the **lkup()** command.

Table 10-9 **lkup()** Options

Option	Value	Description
LKUP_ALL	0x0	Print everything registered in the symbol tables. (This is the default.)
LKUP_DETAILS	0x1	Print additional information about the symbols, including the name of the modules that use a given symbol.
LKUP_LINKS	0x2	Restrict output to the links to symbols in other protection domains.
LKUP_ENTRY	0x4	Restrict output to the protection domain's entry points.

Table 10-9 **lkup() Options**

Option	Value	Description
LKUP_PENDING	0x8	Restrict output to the pending symbols required by the modules within the protection domain.
LKUP_SYSMS	0x10	Restrict output to the symbols declared within the protection domain. Do not show links, entry points, or pending symbols.

Another information command is a symbolic disassembler, **l0**. The command syntax is:

l [*addr* [, *n*]]

This command lists *n* disassembled instructions, starting at *addr*. If *n* is 0 or not given, the command uses the *n* from a previous **l0**, or if there is none, the default value (10). If *addr* is 0, **l0** starts from where the previous **l0** stopped, or from where an exception occurred (if there was an exception trap or a breakpoint since the last **l0** command).

The disassembler uses any symbols that are in the symbol table. If an instruction whose address corresponds to a symbol is disassembled (the beginning of a routine, for instance), the symbol is shown as a label in the address field. Symbols are also used in the operand field. The following is an example of disassembled code for a PowerPC target:

```
[coreOS] -> l printf
Disassembly for coreOS (PD ID 0x3577f4)
printf:
0x0013edd4 9421ff80      stwu      r1,-128(r1)
0x0013edd8 7c0802a6      mfspr     r0,LR
0x0013eddc 90a10010      stw       r5,16(r1)
0x0013ede0 3ca00014      lis       r5,0x14 # 20
0x0013ede4 9081000c      stw       r4,12(r1)
0x0013ede8 38a501d0      addi      r5,r5,0x1d0 # 464
0x0013edec 90c10014      stw       r6,20(r1)
0x0013edf0 38810070      addi      r4,r1,0x70 # 112
0x0013edf4 90010084      stw       r0,132(r1)
0x0013edf8 38c00001      li        r6,0x1 # 1
value = 0 = 0x0
[coreOS] ->
```

This example shows the **printf()** routine. The routine does a **LINK**, then pushes the value of **std_out** onto the stack and calls the routine **fioFormatV()**. Notice that symbols defined in C (routine and variable names) are prefixed with an underscore (**_**) by the compiler.

Perhaps the most frequently used system information command is **d0**, which displays a block of memory starting at the address that is passed to it as a parameter. As with any other routine that requires an address, the starting address can be a number, the name of a variable or routine, or the result of an expression.

Several examples of variations on **d0** appear below.

Display starting at address 1000 decimal:

```
-> d (1000)
```

Display starting at 1000 hex:

```
-> d 0x1000
```

Display starting at the address contained in the variable **foo**:

```
-> d foo
```

The above is different from a display starting at the address of **foo**. For example, if **foo** is a variable at location 0x1234, and that memory location contains the value 10000, **d0** displays starting at 10000 in the previous example and at 0x1234 in the following:

```
-> d &foo
```

Display starting at an offset from the value of **foo**:

```
-> d foo + 100
```

Display starting at the result of a function call:

```
-> d func (foo)
```

Display the code of **func()** as a simple hex memory dump:

```
-> d func
```



CAUTION: Remember that the effect of a command may be different in the host and target shells. If you mount a drive on the target at **/ata0/**, you will be unable to **cd()** to it from the host shell, which has no concept of a target working directory. However, if you use **@cd**, the target shell will recognize the device.

10.14.4 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. This section summarizes the shell commands of this type. For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River**

Documentation > References > Host Tools > Wind River Host Shell API
Reference > windsh.)

Table 10-10 System Modification and Debugging Commands

Call	Description
ml()	Load an object module into memory and link it dynamically into the runtime.
mlr()	Load an object module into memory, link it dynamically into the runtime, and run it.
ld()	Obsolete; replaced by ml() . Available for backward compatibility.
mu()	Remove a dynamically-linked object module from target memory, and free the storage it occupied.
unld()	Obsolete; replaced by mu() . Available for backward compatibility.
m()	Modify memory in <i>width</i> (byte, short, or long) starting at <i>addr</i> . The m() command displays successive words in memory on the terminal; you can change each word by typing a new hex value, leave the word unchanged and continue by typing ENTER , or return to the shell by typing a dot (".").
mRegs()	Modify register values for a specific task.
b()	Set or display breakpoints, in a specified task or in all tasks.
pdb()	Set or display protection domain breakpoints.
bh()	Set a hardware breakpoint.
pdbh()	Set a hardware protection domain breakpoint.
s()	Step a program to the next instruction.
so()	Single-step, but step over a subroutine.
c()	Continue from a breakpoint.
cret()	Continue until the current subroutine returns.
bdall()	Delete all breakpoints.
bd()	Delete a breakpoint.

Table 10-10 System Modification and Debugging Commands

Call	Description
reboot()	Return target control to the boot loader, then reset the target server and reattach the shell.
bootChange()	Modify the saved values of boot parameters.
sysSuspend()	Enter system mode (if supported by the target-agent configuration.)
sysResume()	Return from system mode to task mode.
agentModeShow()	Show the agent mode (<i>system</i> or <i>task</i> .)
sysStatusShow()	Show the system context status (<i>suspended</i> or <i>running</i> .)
quit() or exit()	Close the shell.

One of the most useful shell features for interactive development is the dynamic linker. With the shell command **ml()**, you can download and link new portions of the application. Because the linking is dynamic, you only have to rebuild the particular piece you are working on, not the entire application. Download can be cancelled with **CTRL+C** or by clicking **Cancel** in the load progress indicator window.

The **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 an eligible task executes that instruction (as specified with the **b0** command), the task that was executing on the target suspends, and a message appears at the shell. At this point, you can examine the task's registers, do a task trace, and so on. The task can then be deleted, continued, or single-stepped.

If a routine called from the shell encounters a breakpoint, it suspends just as any other routine would, but in order to allow you to regain control of the shell, such suspended routines are treated in the shell as though they had returned 0. The suspended routine is nevertheless available for your inspection.

When you use **s()** to single-step a task, the task executes one machine instruction, then suspends again. The shell display shows all the task registers and the next instruction to be executed by the task.

You can use the **bh()** command to set hardware breakpoints at any instruction or data element. Instruction hardware breakpoints can be useful to debug code

running in ROM or flash EPROM. Data hardware breakpoints (watchpoints) are useful if you want to stop when your program accesses a specific address. Hardware breakpoints are available on Intel x86, MIPS, and some PowerPC processors. The arguments of the **bh0** command are architecture-specific. For more information, run the **help0** command. The number of hardware breakpoints you can set is limited by the hardware; if you exceed the maximum number, you will receive an error.

10.14.5 Protection Domains

This section describes routines that relate exclusively to protection domains. To view task information on a per-domain basis, or to spawn a task in a specific protection domain, use **pdi0**. You can also spawn a task in a user domain with **sp0** if it is the current working domain of the shell. There are show routines for domains in general, as well as specific routines for shared data and shared library domains. The **pdHelp0** routine displays a summary of the protection domain related utility functions with a short description of each.

For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Table 10-11 **Protection Domain Commands**

Call	Description
pdi0	Display tasks on a per-protection-domain basis.
pdShow0	Display information for protection domains.
sdShow0	Display information about a shared data protection domain.
slShow0	Display information about a shared library protection domain.
pdHelp0	Display protection domain shell function synopsis.

10.14.6 C++ Development

This section describes commands that are intended specifically for C++ applications.

For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh**.)

Also see the *VxWorks 653 Programmer's Guide: Developing C++ Applications*.

Table 10-12 C++ Development Commands

Call	Description
cplusCtors()	Call static constructors manually.
cplusDtors()	Call static destructors manually.
cplusStratShow()	Report on whether current constructor/destructor strategy is manual or automatic.
cplusXtorSet()	Set constructor/destructor strategy.

In addition, you can use the Tcl routine **shConfig** to set the environment variable **LD_CALL_XTORS** within a particular shell. This allows you to use a different C++ strategy in a shell than is used on the target. For more information on **shConfig**, see [8.4 Setting Shell Environment Variables](#), p.141.

10.14.7 Object Display

This section describes commands that display operating system objects. The browser provides displays that are analogous to the output of many of these routines, except that browser windows can update their contents periodically.

For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh**.)

Table 10-13 Object Display Commands

Call	Description
show(0)	Print information on a specified object in the shell window.
classShow()	Show information about a class of kernel objects. List available classes with lkup "ClassId" .
taskShow()	Display information from a task's task control block (TCB), including protection domain information.
taskCreateHookShow()	Show the list of task create routines.
taskDeleteHookShow()	Show the list of task delete routines.
taskRegsShow()	Display the contents of a task's registers.
taskSwitchHookShow()	Show the list of task switch routines.
taskWaitShow()	Show information about the object a task is pended on. Note that taskWaitShow() cannot give object IDs for POSIX semaphores or message queues.
semShow()	Show information about a semaphore.
semPxShow()	Show information about a POSIX semaphore.
wdShow()	Show information about a watchdog timer.
msgQShow()	Show information about a message queue.
mqPxShow()	Show information about a POSIX message queue.
iosDrvShow()	Display a list of system drivers.
iosDevShow	Display the list of devices in the system.
iosFdShow()	Display a list of file descriptor names in the system.
memPartShow()	Show partition blocks and statistics at specified level of verbosity.

Table 10-13 Object Display Commands

Call	Description
memShow()	Display the total amount of free and allocated space in the system partition, the number of free and allocated fragments, the average free and allocated fragment sizes, and the maximum free fragment size. Show current as well as cumulative values. With an argument of 1 , also display the free list of the system partition; with an argument of 2 , display the address of each free block.
smMemShow()	Display the amount of free space and statistics on memory-block allocation for the shared-memory system partition.
smMemPartShow()	Display the amount of free space and statistics on memory-block allocation for a specified shared-memory partition.
moduleShow()	Show the current status for all loaded modules.
moduleIdFigure()	Report a loaded module's module ID, given its name.
intVecShow()	Display the interrupt vector table. This routine displays information about the given vector or the whole interrupt vector table if vector is equal to -1. Note that intVecShow() is not supported on architectures that do not use interrupt vectors.
memAttrShow()	Display information about all the typed partitions for a given protection domain.
memAttrPartShow()	Display statistics of a given typed partition.
objShowAll()	Show all information on an object.
objNameShow()	Display information about named objects.
pdShow()	Display information for protection domains.
sdShow()	Display information about a shared data protection domain.
slShow()	Display information about a shared library protection domain.
pgPoolLstShow()	Show data of a list of page pools.

Table 10-13 **Object Display Commands**

Call	Description
<code>pgPoolShow()</code>	Show data of a page pool.
<code>pgMgrShow()</code>	Display information about a page manager.

10.14.8 Network Status Display

This section describes commands that display information about the operating system network. In order for a protocol-specific command to work, the appropriate protocol must be included in your operating system configuration.

For more detailed reference information, see the **windsh** reference entry (open Wind River Workbench and select **Help > Help Contents > Wind River Documentation > References > Host Tools > Wind River Host Shell API Reference > windsh.**)

Table 10-14 **Network Status Display Commands**

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

10.15 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()`.

10

10.16 Examples

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)
```

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 example below.

Call an application function that has the same name as a shell command.

```
-> @i()
```


11

Using the Tcl Interpreter with VxWorks 653

11.1 Introduction	209
11.2 Controlling the Target	210
11.3 Accessing the WTX Tcl API	211
11.4 Calling Target Routines	212
11.5 Passing Values to Target Routines	212
11.6 Calling Under C Control	213
11.7 Shell Initialization	213

11.1 Introduction

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 non-blank character on a line, and passes the entire line following it to the Tcl interpreter.

The following example uses the C interpreter to define a variable in the symbol table, then switch to the Tcl interpreter to define a similar Tcl variable in the shell itself, and then switch back to the C interpreter:

```
-> foo="bar"
new symbol "foo" added to symbol table.
foo = 0x3616e8: value = 3544824 = 0x3616f8 = foo + 0x10
-> ?
tcl> set foo {bar}
bar
tcl> C
->
```

On startup, you can use the option **-Tclmode** (or **-T**) to start with the Tcl interpreter.

Using the shell's Tcl interface allows you to extend the shell with your own procedures, and also provides a set of control structures which you can use interactively. The Tcl interpreter also gives you access to command-line utilities on your development host.

11.2 Controlling the Target

In the Tcl interpreter, you can create custom commands, or use Tcl control structures for repetitive tasks, while using the building blocks that allow the C interpreter and the host shell commands to control the target remotely. These building blocks as a whole are called the **wtxtcl** procedures.

For example, **wtxMemRead** returns the contents of a block of target memory (given its starting address and length). That command in turn uses a special memory-block data type designed to permit memory transfers without unnecessary Tcl data conversions. The following example uses **wtxMemRead**, together with the memory-block routine **memBlockWriteFile**, to write a Tcl procedure that dumps target memory to a host file. Because almost all the work is done on the host, this procedure works whether or not the target run-time environment contains I/O libraries or any networked access to the host file system.

```
# tgtMemDump - copy target memory to host file
#
```

```
# SYNOPSIS:
#  tgtMemDump hostfile start nbytes

proc tgtMemDump {fname start nbytes} {
    set memHandle [wtxMemRead $start $nbytes]
    memBlockWriteFile $memHandle $fname
}
```

For reference information on the **wtxtcl** routines available in the host shell, see the online help: in Workbench, select **Help > Help Contents > Wind River Documentation > References > Host Tools > WTX Tcl Library Reference**.

All of the commands defined for the C interpreter (see [10. Using the C Interpreter with VxWorks 653](#)) are also available, with a double-underscore prefix, from the Tcl level; for example, to call **i()** from the Tcl interpreter, run the Tcl procedure **_i**. However, in many cases, it is more convenient to call a **wtxtcl** routine instead, because the host shell commands are designed to operate in the C interpreter context.

For example, you can call the dynamic linker using **ld** from the Tcl interpreter, but the argument that names the object module may not seem intuitive: it is the address of a string stored on the target. It is more convenient to call the underlying **wtxtcl** command. In the case of the dynamic linker, the underlying **wtxtcl** command is **wtxObjModuleLoad**, which takes an ordinary Tcl string as its argument.

11.3 Accessing the WTX Tcl API

The Wind River Tool Exchange (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 reference of WTX Tcl API commands, see the online help: in Workbench, select **Help > Help Contents > Wind River Documentation > References > Host Tools > WTX Tcl Library Reference**.

11.4 Calling Target Routines

The **shParse** utility allows you to embed calls to the C interpreter in Tcl expressions; the most frequent application is to call a single target routine, with the arguments specified (and perhaps capture the result). For example, the following sends a logging message to your target console:

```
tcl> shParse {logMsg("foobar\n")}
32
```

You can also use **shParse** to call host shell commands more conveniently from the Tcl interpreter, rather than using their **wtxtcl** building blocks. For example, the following is a convenient way to spawn a task from Tcl, using the C interpreter command **sp()**, if you do not remember the underlying **wtxtcl** command:

```
tcl> shParse {sp appTaskBegin}
task spawned: id = 25e388, name = u1
0
```

11.5 Passing Values to Target Routines

Because **shParse** accepts a single, ordinary Tcl string as its argument, you can pass values from the Tcl interpreter to C subroutine calls by using Tcl facilities to concatenate the appropriate values into a C expression.

For example, a more realistic way of calling **logMsg()** from the Tcl interpreter would be to pass, as its argument, the value of a Tcl variable rather than a literal string. The following example evaluates the Tcl variable **tclLog** and inserts its value (with a newline appended) as the **logMsg()** argument:

```
tcl> shParse "logMsg(\"$tclLog\n\")"
32
```

11.6 Calling Under C Control

To use a Tcl command and return immediately to the C interpreter, you can type a single line of Tcl prefixed with the `?` character (rather than using `?` by itself to toggle into Tcl mode). For example:

```
-> ?set test foobar; puts "This is $test."
This is foobar.
->
```

Notice that the `->` prompt indicates that you are still in the C interpreter, even though you just executed a line of Tcl.



CAUTION: You may not embed Tcl evaluation inside a C expression; the `?` prefix works only as the first nonblank character on a line, and passes the entire line following it to the Tcl interpreter.

For example, you may want to use Tcl control structures to supplement the facilities of the C interpreter. Suppose you have an application under development that involves several collaborating tasks; in an interactive development session, you may need to restart the whole group of tasks repeatedly. You can define a Tcl variable with a list of all the task entry points, as follows:

```
-> ? set appTasks {appFrobStart appGetStart appPutStart ...}
appFrobStart appGetStart appPutStart ...
```

Then whenever you need to restart the whole list of tasks, you can use something like the following:

```
-> ? foreach it $appTasks {shParse "sp($it)"}
task spawned: id = 25e388, name = u0
task spawned: id = 259368, name = u1
task spawned: id = 254348, name = u2
task spawned: id = 24f328, name = u3
```

11.7 Shell Initialization

When you execute an instance of the host shell, it begins by looking for a file called **windsh.tcl** in two places: first under *installDir/workbench-3.x/foundation/build/resource/windsh*, and then in the directory specified by the **HOME** environment variable (if that environment variable is defined). In each of these directories, if the file exists, the shell reads and

executes its contents as Tcl expressions before beginning to interact. You can use this file to automate any initialization steps you perform repeatedly.

You can also specify a Tcl expression to execute initially on the host shell command line, with the option **-e *tcl_expression***. For example, you can test an initialization file before saving it as **windsh.tcl** using this option, as follows:

```
% windsh phobos -e "source c:\\fred\\tcltest"
```

11.7.1 Shell Initialization File

This file causes I/O for target routines called in the host shell to be directed to the target's standard I/O rather than to the host shell. It changes the default C++ strategy to automatic for this shell, sets a path for locating load modules, and causes modules not to be copied to the target server.

```
# Redirect Task I/O to WindSh
shConfig SH_GET_TASK_IO off
# Set C++ strategy
shConfig LD_CALL_XTORS on
# Set Load Path
shConfig LD_PATH "/home/username/project/app;/home/username/project/test"
# Let the Target Server directly access the module
shConfig LD_SEND_MODULES off
```

PART III

Appendices

A	Using the Host Shell Line Editor	217
B	Single Step Compatibility	225

A

Using the Host Shell Line Editor

- [A.1 Introduction 217](#)
- [A.2 vi-Style Editing 218](#)
- [A.3 emacs-Style Editing 221](#)
- [A.4 Command Matching 223](#)

A.1 Introduction

This appendix applies to all target operating systems.

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.2 vi-Style Editing

The **ESC** key switches the shell from normal input mode to edit mode. The history and editing commands in [Table A-1](#) and [Table A-3](#) 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-2](#); you can then edit and execute the line with one of the commands from the table.

A.2.1 Switching Modes and Controlling the Editor

[Table A-1](#) lists commands that give you basic control over the editor.

Table A-1 **vi-Style Basic Control Commands**

Command	Description
h [<i>size</i>]	Displays shell history if no argument is given; otherwise sets history buffer to <i>size</i> .
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).

Table A-1 **vi-Style Basic Control Commands** (cont'd)

Command	Description
CTRL+S	Suspend output.
CTRL+Q	Resume output.
CTRL+W	Display HTML reference entry for a routine.

A.2.2 Moving and Searching in the Editor

[Table A-2](#) lists commands for moving and searching in input mode.

Table A-2 **vi-Style Movement and Search Commands**

Command	Description
<i>n</i> G	Go to command number <i>n</i> . The default value for <i>n</i> is 1.
<i>/s</i> or <i>?s</i>	Search for string <i>s</i> backward or forward in history.
n	Repeat last search.
<i>n</i> k or <i>n</i> -	Get <i>n</i> th previous shell command.
<i>n</i> j or <i>n</i> +	Get <i>n</i> th next shell command.
<i>n</i> h	Go left <i>n</i> characters (also CTRL+H).
<i>n</i> l or SPACE	Go right <i>n</i> characters.
<i>n</i> w or <i>n</i> W	Go <i>n</i> words forward, or <i>n</i> large words. <i>Words</i> are separated by spaces or punctuation; <i>large words</i> are separated by spaces only.
<i>n</i> e or <i>n</i> E	Go to end of the <i>n</i> th next word, or <i>n</i> th next large word.
<i>n</i> b or <i>n</i> B	Go back <i>n</i> words, or <i>n</i> large words.
\$	Go to end of line.
0 or ^	Go to beginning of line, or to first nonblank character.
f <i>c</i> or F <i>c</i>	Find character <i>c</i> , searching forward or backward.

A.2.3 Inserting and Changing Text

Table A-3 lists commands to insert and change text in the editor.

Table A-3 **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).
ns ...ESC	Change <i>n</i> 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).
R ...ESC	Type over characters (ESC ends input).
nrc	Replace the following <i>n</i> characters with <i>c</i> .
~	Toggle between lower and upper case.

A.2.4 Deleting Text

Table A-4 shows commands for deleting text.

Table A-4 **vi-Style Commands for Deleting Text**

Command	Description
nx or nX	Delete next <i>n</i> characters or previous <i>n</i> 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.

A.2.5 Put and Undo Commands

Table A-5 shows put and undo commands.

Table A-5 **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.3 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**.

A.3.1 Moving the Cursor

Table A-6 lists commands for moving the cursor in emacs mode.

Table A-6 **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.

Table A-6 **emacs-Style Cursor Motion Commands** (cont'd)

Command	Description
CTRL+A	Move cursor to beginning of line.
CTRL+E	Move cursor to end of line.

A.3.2 Deleting and Recalling Text

Table A-7 shows commands for deleting and recalling text.

Table A-7 **emacs-Style Deletion and Recall Commands**

Command	Description
DEL or CTRL+H	Delete character to left of cursor.
CTRL+D	Delete character under cursor.
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.
! <i>n</i>	Recall command <i>n</i> from the history.
! <i>substr</i>	Recall first command from the history matching <i>substr</i> .

A.3.3 Special Commands

Table A-8 shows some special emacs-mode commands.

Table A-8 **Special emacs-Style Commands**

Command	Description
CTRL+U	Delete line and leave edit mode.

Table A-8 **Special emacs-Style Commands** (cont'd)

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

A.4 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.

A.4.1 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.4.2 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.

B

Single Step Compatibility

B.1 Introduction	225
B.2 Scripting	226
B.3 SingleStep Command Equivalents	226
B.4 SingleStep read Command Compatibility	230
B.5 SingleStep write Command Compatibility	232
B.6 SingleStep Variable Compatibility	233

B.1 Introduction

This chapter describes backward compatibility for previous users of Wind River SingleStep.

In this release, Wind River has used the host shell to implement a replacement for SingleStep scripting functionality.

The host shell provides a Tcl interpreter in place of SingleStep's C shell. Tcl offers superior control constructs (i.e., arrays, namespaces, exceptions, and so on) and the ability to bind to native code libraries.

In the host shell, Tcl variables take the place of a subset of SingleStep's debugger and shell variables. These variables are given default values by the host shell's startup Tcl code.

When the host shell starts, it sources the file *value/.wind/wb/windsh.tcl*, where *value* is the value of the environment variable **HOME**; or, if that variable is not defined, the value of the environment variable **WIND_FOUNDATION_PATH**. You can edit this file to contain arbitrary Tcl commands to execute every time the host shell starts. In particular, commands in this file can modify the default value of Tcl variables used to provide SingleStep compatibility.

B.2 Scripting

The host shell will not execute SingleStep scripts. Existing SingleStep scripts must be manually converted, using the equivalents described in this chapter.

The host shell does not have all of the scripting functionality of SingleStep; in particular, pROBE+ and pRISM+ debugger variables are not supported. See [B.6 SingleStep Variable Compatibility](#), p.233.

B.3 SingleStep Command Equivalents

[Table B-1](#) enumerates each SingleStep command, along with its description and the equivalent host shell command (if any). There are 72 SingleStep commands. Some have equivalent host shell commands, some have no equivalent host shell commands, and some have similar but not exactly equivalent host shell commands.

Table B-1 **SingleStep Command Equivalents**

SingleStep Command	Description	Host Shell Equivalent
?	Print value of expression	print (GDB mode)
@	Set shell variable to expression	set (Tcl mode)
alias	Create command aliases	proc (Tcl mode)

Table B-1 SingleStep Command Equivalents

SingleStep Command	Description	Host Shell Equivalent
args	Display own arguments	None.
asm	Assemble into memory	None.
break	Set a breakpoint	break or hbreak (GDB mode). These commands are not as functional as the SingleStep break command.
cache	Display instruction/data cache	None.
call	Call function or subroutine	None.
cd	Change directory	cd (cmd and Tcl modes)
cflush	Flush cache memory	None.
continue	Continue loop	continue (Tcl mode)
control	Enable diagnostics	None.
copymem	Copy memory	None.
curtask	Set current task	attach (GDB mode)
debug	Select program to debug	No single equivalent. This command maps to the wrsreset and wrsdownload commands.
echo	Display arguments	puts (Tcl mode)
exit	Exit debugger or script	exit (Tcl mode)
false	No-op that always fails	false (Tcl procedure defined in host shell startup script)
flash	Flash programmer commands	wrspassthru (GDB mode)
foreach	Loop through a list	foreach (Tcl mode)
glob	Display arguments	None.

Table B-1 SingleStep Command Equivalents

SingleStep Command	Description	Host Shell Equivalent
go	Run the target	Similar command: continue (GDB mode). The continue command does not support the -n and -i options from the go command.
goto	Execute to a location	Similar commands: set and continue (GDB mode). This is equivalent to setting the instruction pointer and issuing a continue command.
help		
help	Display help on commands	help (GDB mode)
history	Display command history	None.
if	Conditional execution	if (Tcl mode)
jobs	Report background jobs	None.
kernel	Display kernel objects	None.
load	Load memory	None. (Downloads Block Binary files, which the host shell does not support.)
loadi	Load a memory image	wrsdownload (GDB mode)
loop	Execute until here again	Similar commands: tbreak and continue (GDB mode). This is equivalent to setting a temporary breakpoint and issuing a continue command.
loopbreak	Break a loop	break (Tcl mode)
mem	Specify a memory map	wrsmemmap (GDB mode)
module	Load or unload symbols	Similar command: wrsdownload (GDB mode)
nop	No operation	; (Tcl mode)
offset	For position independence	None.
osboot	Boot probe+	None.

Table B-1 SingleStep Command Equivalents

SingleStep Command	Description	Host Shell Equivalent
probe	Pass command to probe+	None.
pwd	Print working directory	pwd (Tcl mode)
read	Read a variable or memory	Similar commands: print and x (GDB mode).
regs	Display registers	print and info registers (GDB mode)
repeat	Repeat a command	Similar commands: for or while (Tcl mode).
reset	Reset the target	Similar command: wrsreset (GDB mode).
see	See contents of files	None.
set	Set debugger variable	Similar command: set (Tcl mode). (In the host shell, all variables are Tcl variables.)
setenv	Set an environment variable	set env (<i>varname</i>) <i>value</i> (Tcl mode)
shift	Shift a variable	set argv [lreplace \$argv 0 0] (Tcl mode)
sizeof	Display size of variables	None.
sleep	Simulate sleep mode	after (Tcl mode)
source	Execute from a file	source (Tcl mode)
stack	Display the call stack	bt (GDB mode)
status	Get target status	None.
step	Step one statement	step (GDB mode)
stop	Stop the target	None.
targetio	Share target i/o spaces	None.
true	Generate success status	true (Tcl procedure defined in host shell startup script.)
typeof	Display variable types	None.
umask	Get/set creation mask	None.

Table B-1 SingleStep Command Equivalents

SingleStep Command	Description	Host Shell Equivalent
unalias	Remove an alias	Similar command: proc (Tcl mode). The closest thing the host shell can do to emulate unalias is to redefine the Tcl procedure to do nothing.
unset	Remove a shell variable	unset (Tcl mode)
unsetenv	Remove an environment variable	array unset env <i>varname</i> (Tcl mode).
update	Control view updates (graphical)	None.
upload	Upload memory	Similar command: wrsupload (GDB mode)
visible	Execute DOS command	exec (Tcl mode). The exec command works on every platform, not just Windows.
wait	Wait for child processes	None.
watch	Watch a variable	None.
wedit	Edit source code (graphical)	None.
where	Display context	list (GDB mode)
whereis	Find files in the path	None.
while	Command loop	while (GDB mode)
write	Write variables or memory	Similar commands: print (GDB mode) or mem modify (Cmd mode).

B.4 SingleStep read Command Compatibility

The SingleStep **read** command has a complex syntax that has no exact equivalent in the host shell. Existing host shell GDB mode commands provide most of the

same functionality as the **read** command. Table B-2 shows how various SingleStep **read** commands map to host shell GDB mode commands.

Table B-2 **SingleStep read Command Compatibility**

SingleStep read Command	Description	Host Shell Equivalent
read <i>x</i>	Display value of variable <i>x</i> .	print <i>x</i>
read <i>x y z</i>	Display values of three variables.	GDB mode: print <i>x</i> ; print <i>y</i> ; print <i>z</i> Tcl mode: foreach var { <i>x y z</i> } {eval "puts \\$\$var"}
read -l# <i>arg</i>	Display variable <i>arg</i> from first function on stack.	None.
read file.c# <i>var</i>	Display static variable <i>var</i> from file.c.	print file.c: <i>var</i>
read main	Disassemble starting at main.	disassemble main
read 0x4000	Dump starting at address 0x4000.	<i>x</i> /32xw 0x4000
read -ux CPU:0x3FF00=long	Read the MBAR register of a 68360.	None.
read -Rux 0x7E02=char	Read one byte at address 0x7E02.	<i>x</i> /1xb 0x7e02
read -F 0x4000	Disassemble starting at 0x4000.	disassemble 0x4000
read <i>var</i> =long	Display variable <i>var</i> as if it were a long.	print (long) <i>var</i>
read 0x120=(<i>sym</i>)	Display 0x120 using type from variable <i>sym</i> .	None.
read * <i>p</i>	Display whatever <i>p</i> points to.	print * <i>p</i>
read <i>a</i> [5]	Display the fifth element of array <i>a</i> .	print [<i>a</i>]5
read str.mem	Display member <i>mem</i> .	print str.mem

Table B-2 SingleStep read Command Compatibility

SingleStep read Command	Description	Host Shell Equivalent
read p->mem	Display member mem .	print p->mem
read	Continue previous read.	None.

B.5 SingleStep write Command Compatibility

The SingleStep **write** command has a complex syntax that has no exact equivalent in the host shell. Existing host shell GDB mode commands provide most of the same functionality as the **write** command. [Table B-3](#) shows how various SingleStep **read** commands map to host shell GDB mode commands.

Table B-3 SingleStep write Command Compatibility

SingleStep write Command	Description	Host Shell Equivalent
write <i>var</i> =99	Write value 99 to variable <i>var</i> .	set <i>var</i> =99
write x=1 y=2 z=3	Write values to multiple variables.	set x=1; set y=2; set z=3
write *ptr=88	Write value to destination of a pointer.	set *ptr = 88
write obj.member=77	Write value to member of structure or class.	set obj.member = 77
write -b 0x1000=99	Write the value 99 to the byte at 0x1000.	set *(char *)0x1000 = 99
write -w 0x1000=999	Write the value 999 to the word at 0x1000.	set *(short *)0x1000 = 999
write -l 0x1000=99999	Write the value 99999 to the longword at 0x1000.	set *(long *)0x1000 = 99999

Table B-3 SingleStep write Command Compatibility

SingleStep write Command	Description	Host Shell Equivalent
write -s 0x1000=3.14	Write the value 3.14 to the single-precision float at 0x1000.	set *(float *)0x1000 = 3.14
write -d 0x1000=3.14	Write value 3.14 to the double-precision float at 0x1000.	set *(double *)0x1000 = 3.14
write -e 0x1000=3.14	Write value 3.14 to the extended-precision float at 0x1000.	None.
write -f 99 x y z	Write value 99 to variables x, y, and z.	No GDB mode equivalent. In Tcl mode: foreach var {x y z} {set \$var 99}

The following SingleStep **write** command options are not implemented in the host shell:

- -c *count*
- -q
- -r
- -u
- -x
- -H
- -W

B.6 SingleStep Variable Compatibility

Table B-4 enumerates each SingleStep debugger and shell variable along with its description and the equivalent host shell variable (if any). There are 44 SingleStep variables. Some have equivalent host shell variables, some have no equivalent host shell variables, and some have similar but not exactly equivalent host shell variables.

SingleStep had two variable namespaces: debugger variables and shell variables. The host shell only has the Tcl variable namespace.

Table B-4 **SingleStep Variable Equivalents**

SingleStep Variable	Description	Host Shell Equivalent
altsep	Word separator	None
altshell	Alternate shell	None
argv	List of arguments	None
backtick	Command substitution character	None
breaknums	Breakpoint numbers	breaknums
cdpath	Directory search path	None
child	Background process id	None
debugblk	Download data file	None
debugchip	Processor name	None
debugdb	Symbol database file	None
debugdb2	Symbol database file	None
debugout	Linker output file	None
echo	Echo commands	None
hexreplace	Floats in hex	None
histchars	History substitution characters	None
history	Size of history list	None
home	Home directory	Equivalent expression: \$env(HOME)
ignoreeof	Ignore eof characters	None
kanji_code	Kanji codes	None
litebold	Highlight sequence	None
liteoff	No source highlight	None
liteon	Turn on highlighting	None
mail	Files for mail	None

Table B-4 **SingleStep Variable Equivalents**

SingleStep Variable	Description	Host Shell Equivalent
morelines	Lines for display	None
no_binary_msg	ASCII download	None
noclobber	Do not overwrite files	None
noglob	No file name substitution	None
nonomatch	No match complaint	None
ovlflags	Overlay flags	None
path	Executable search path	\$env(PATH)
product	Version of debugger	Equivalent expression: [tclShellVersionGet]
prompt	Command line prompt	None
random	Random seed value	Equivalent expressions: expr srand(N) or expr rand()
root	Root directory name	root
shell	Primary shell name	None
srclines	Lines for source window	None
srclist	Source file path list	None
srcpath	Source file path list	None
status	Command return status	None
stkberr	Stack error checking	None
unixwild	Wild card style	None
vectaddr	Vector address	None
vectskip	Exception vector list	None
verbose	Verbose information	None