WIND RIVER

Wind River® Workbench

HOST SHELL USER'S GUIDE

2.6

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

Wind River Workbench Host Shell User's Guide, 2.6

28 Sep 06
Part #: DOC-15898-ND-00

Contents

1 Introduction

1.1 Introduction

The host shell is a host-resident command shell that allows you to download application modules, invoke operating-system and application subroutines, and monitor and debug applications. A target-resident version of the shell (called the *kernel shell*) is also available; see the *VxWorks Kernel Programmer's Guide: Target Tools*.

Host shell operation involves a *target server*, which handles communication with the remote target, dispatching function calls and returning their results; and a *target agent*, a small monitor program that mediates access to target memory and other facilities. The target agent is the only component that runs on the target. The

symbol table, managed by the target server, resides on the host, although the addresses it contains refer to the target system.

1.2 Host Shell Modes

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

- *C interpreter*, which executes C-language expressions and allows prototyping and debugging in kernel space.
- *Command* (*Cmd*), a UNIX-style command interpreter for debugging and monitoring a system, including RTPs.
- *Tcl*, to access the WTX TCL API and for scripting.
- *GDB*, for debugging a target using GNU Debugger (GDB) commands.

Which mode the shell opens in by default depends on what operating system it detects. If it does not detect an operating system it opens in **GDB** mode. You can also specify a mode when invoking the shell (see *1.3 Starting the Host Shell*, p.2.)

1.3 Starting the Host Shell

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

1.3.1 Starting the Host Shell from the Command Prompt

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 will return the following error:

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

To set your environment, enter the following command at the prompt:

```
wrenv -p workbench-2.x
```

where *x* is the currently installed version of Workbench.

To start the host shell, type the following:

```
windsh [options] targetServer
```

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

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

NOTE: When you start the host shell, a second shell window appears, running the Debug server. You can minimize this second window to reclaim screen space, but do not close it.

You may run as many different host shells attached to the same target as you wish. The output from a function called in a particular shell appears in the window from which it was called, unless you change the shell defaults using **shConfig** (see *1.5 Setting Shell Environment Variables*, p.5).

Host Shell Startup Options

Table 1-1 **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.

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

Option	Description
-e, **-execute**	Executes Tcl expression after initialization.
-c, **-command**	Executes expression and exits shell (batch mode).
-r, **-root mappings**	Root pathname mappings.
-ds[DFW server Session]	Debugger Server session to use.
-dp[DFW server Port]	Debugger Server port to use.
-host	Retrieves target server information from host's registry.
-s, **-startup**	Specifies the startup file of shell commands to execute.
-q, **-quiet**	Turns off echo of script commands as they are executed.
-dt *target*	Backend target definition name.

1.3.2 Starting the Host Shell from Workbench

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

In the **Target Manager**, right-click on your target connection name and select **Target Tools > Host Shell**. The **Start Host Shell** dialog appears. You can specify startup options from Table 1-1 in this dialog, or leave them at their defaults. Click **OK** to start the host shell.

1.4 Switching Interpreters

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

- **cmd** to switch to the command interpreter. The prompt changes to **[vxWorks] #**.

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

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

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

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

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> 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 = 0x14200
```

1.5 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 1-2 provides a list of the host shell's environment variables and their significance.

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

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 1-2.

Table 1-2 **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.
RTP_CREATE_ATTACH [on | off]	When RTP support is configured in the system, this option indicates whether the shell should automatically attach to any RTPs launched from the host shell (using the host shell's command interpreter).
VXE_PATH .	When RTP support is configured in the system, this option indicates the path in which the host shell should search for RTPs to launch. If this is set to "." the full pathname of an RTP should be supplied to the command to launch an RTP.
ROOT_PATH_MAPPING	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 **/folk/***user* 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 **strFree( ).**

Table 1-2 **Host Shell Environment Variables** (cont'd)

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

Table 1-2 **Host Shell Environment Variables** (cont'd)

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

Path Mapping

Since the host shell uses host paths to handle RTPs 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.

Use the **ROOT_PATH_MAPPING** shell environment variable to define 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 **cocyte:/home/users/philb/mydirectory/myrtp.vxe**, the command is:

```
-> ?
tcl> shConfig ROOT_PATH_MAPPING \[cocyte:/home/users/philb/,C:/\]
```

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 file in order to parse the RTP symbols, but the debugger server will send the path of this RTP directly to the target without any transformation by the host shell.

You can also define the **VXE_PATH** shell environment variable.

```
-> ?
tcl> shConfig VXE_PATH "C:/dir1/;C:/dir2/;E:/"
```

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**.

1.6 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:

   ```
   % windsh -m cmd -c "task ; rtp task" tgtsvr@host
   ```

   The **-m** option indicates that the commands should be executed by the Command interpreter.

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

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

1.7 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**.

For more information, see the host shell reference pages: *hostShell*, *cMode*, *cmdMode*, *gdbMode*, and *rtpCmdMode*.

2 Using the Command Interpreter

2.1 Overview

Command interpreter mode applies only to VxWorks targets.

To switch to command interpreter mode from another mode, enter **cmd** at the prompt and press **ENTER**.

The command interpreter is command-oriented and does not understand C language syntax. (For C syntax, use the C interpreter.)

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*] [**;**]

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
```

2.2 General Commands

Table 2-1 summarizes general command-interpreter commands.

Table 2-1 **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 *errno*	Displays the symbol value of an *errno*.
pwd	Displays the current working directory.
quit	Shuts down the shell.
reboot	Reboots the system.
string free	Frees a string allocated by the shell on the target.
unalias	Removes an alias.
version	Displays VxWorks version information.

2.3 Displaying Target Agent Information

Table 2-2 lists the commands related to the target agent.

Table 2-2 **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.

2.4 Working with Memory

Table 2-3 shows the commands related to memory.

Table 2-3 **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.

2.5 Displaying Object Information

Table 2-4 shows commands that display information about objects.

Table 2-4 **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.

2.6 Working with Symbols

Table 2-5 lists commands for displaying and setting values of symbols.

Table 2-5 **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.

2.6.1 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.

2.6.2 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.

2.6.3 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
```

2.6.4 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
```

2.7 Displaying, Controlling, and Stepping Through Tasks

Table 2-6 displays commands for working with tasks.

Table 2-6 **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.

Table 2-6 **Command Interpreter Task Commands** (cont'd)

Command	Description
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.

2.8 Setting Shell Context Information

Table 2-7 displays commands for displaying and setting context information.

Table 2-7 **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.

Table 2-7 **Command Interpreter Shell Context Commands** (cont'd)

Command	Description
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.

2.9 Displaying System Status

Table 2-8 lists commands for showing system status information.

Table 2-8 **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.

2.10 Using and Modifying Aliases

The command interpreter accepts aliases to speed up access to shell commands. Table 2-9 lists the aliases that already exist; they can be modified, and you can add new aliases. Aliases are visible from all shell sessions.

Table 2-9 **Command Interpreter Aliases**

Alias	Definition
alias	List existing aliases. Add a new alias by typing **alias** *aliasname "command"*. For example, **alias ll "ls -l"**.
attach	**rtp attach**
b	**bp**
bd	**bp -u**
bdall	**bp -u #***
bootChange	**set bootline**
c	**task continue**
checkStack	**task stack**
cret	**task continue -r**
d	**mem dump**
detach	**rtp detach**
devs	**show devices**
emacs	**set config LINE_EDIT_MODE="emacs"**
h	**show history**
i	**task**
jobs	**rtp attach**
kill	**rtp detach**
l	**mem list**
lkAddr	**lookup -a**

Table 2-9 **Command Interpreter Aliases** (cont'd)

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

2.11 Launching RTPs

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

```
[vxWorks]# /folk/user/TMP/helloworld.vxe
Launching process '/folk/user/TMP/helloworld.vxe' ...
Process '/folk/user/TMP/helloworld.vxe' (process Id = 0x471630) launched.
[vxWorks]# rtp
    NAME         ID       STATUS    ENTRY ADDR    SIZE    TASK CNT
------------ ---------- ----------- ---------- ---------- --------

[vxWorks]# /folk/user/TMP/cal 12 2004
Launching process '/folk/user/TMP/cal' ...
December 2004
 S  M Tu  W Th  F  S
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
Process '/folk/user/TMP/cal' (process Id = 0x2fdfb0) launched.
```

2.11.1 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.

     }
}
```

2.11.2 Monitoring and Debugging RTPs

Table 2-10 displays the commands related to RTPs.

Table 2-10 **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.
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.

2.11.3 Setting Breakpoints

The **bp** command is designed to set a breakpoint in the kernel, in a 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.

```
bp                  Display, set or unset a breakpoint
                    bp [-p <rtpIdNameNumber>] [-t <taskId>] [[-u {<#bp number>
                    | <bp address>} ...] | [-n <count>] [-h <type>] [-q] [-a]
                    [expr]]
                    This command is used to set or unset (if the option -u is
                    specified) a breakpoint. The breakpoint is a hardware
                    breakpoint if the option -h is specified. Without any
                    arguments, this command displays the breakpoints currently
                    set.
                    The special breakpoint number '#*' or breakpoint address
                    '*' is used to unset all the breakpoints.
                    -a : stop all tasks in a context,
                    -n : number of passes before hit,
                    -h : specify a hardware breakpoint type value,
                    -p : breakpoint applies to specify RTP,
                    -q : no notification when the breakpoint is hit,
                    -t : breakpoint applies to specify task,
                    -u : unset breakpoint
```

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

2.12 Examples

List the contents of a directory.

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

Create an alias.

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

Summarize task TCBs.

```
[vxWorks]# task
```

Suspend a task, then resume it.

```
[vxWorks]# task suspend t1
[vxWorks]# task resume t1
```

Set a breakpoint for a task at a specified address.

```
[vxWorks]# bp -t t1 0x12345678
```

Set a breakpoint on a function.

```
[vxWorks]# bp &printf
```

Show the address of **someInt**.

```
[vxWorks]# echo &someInt
```

Step over a task from a breakpoint.

```
[vxWorks]# task stepover t1
```

Continue a task.

```
[vxWorks]# task continue t1
```

Delete a task.

```
[vxWorks]# task delete t1
```

Run an RTP application.

```
[vxWorks]# /folk/user/TMP/helloworld.vxe
```

Run an RTP application, passing parameters to the executable.

```
[vxWorks]# cal.vxe -j 2002
```

Run an RTP application, passing options to the executable and to the RTP loader (in this case, setting the stack size to 8K).

```
[vxWorks]# rtp exec -u 8096 /folk/user/TMP/foo.vxe -q
```

List RTPs or show brief information about a specific RTP.

```
[vxWorks]# rtp [rtpID]
```

Show details about an RTP.

```
[vxWorks]# rtp info [rtpID]
```

Stop an RTP, then continue it.

```
[vxWorks]# rtp stop 0x43210
[vxWorks]# rtp continue 0x43210
```

3 Using the C Interpreter

3.1 Overview

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 processes and RTPs).

Some of the commands (or routines) that you can execute from the shell are built into the host shell, rather than running as function calls on the target. These commands parallel interactive utilities that can be linked into 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.

3.2 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 3-1.

Table 3-1 **C Interpreter Data-Type Assumptions**

Element	Data Type
variable	int
variable used as a floating-point	double
return value of subroutine	int

Table 3-1 **C Interpreter Data-Type Assumptions**

Element	Data Type
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 *3.3.4 Function Calls*, p.31. Table 3-2 shows the various data types available in the shell C interpreter, with examples of how they can be set and referenced.

Table 3-2 **Data Types in the C Interpreter**

Type	Bytes	Set Variable	Display Variable
int	4	`x = 99`	`x` `(int) x`
long	4	`x = 33` `x = (long)33`	`x` `(long_ x`
short	2	`x = (short)20`	`(short) x`

char	1	`x = 'A'`
		`x = (char)65`
		`x = (char)0x41`
			`(char)x`

double	8	`x = 11.2` `x = (double)11.2`	`(double) x`
float	4	`x = (float)5.42`	`(float) x`

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.

3.3 Expressions

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

3.3.1 Literals

The shell interprets the literals in Table 3-3 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 3-3 **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."

3.3.2 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.

3.3.3 Operators

The shell interprets the operators in Table 3-4 in the same way as the C compiler.

Table 3-4 **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 &&.

3.3.4 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 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:

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

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

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

```
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 *3.3.5 Arguments to Commands*, p.33.

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

```
0x10000 ()
```

3.3.5 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:

```
[vxKernel] -> rename ("oldname", "newname")
[vxKernel] -> rename "oldname", "newname"
```

as are:

```
[vxKernel] -> evtBufferAddress ( )
[vxKernel] -> 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( )**:

```
[vxKernel] -> value = evtBufferAddress
```

3.4 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.

3.4.1 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:

```
[myDomain] -> x = 0x1000
```

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

```
[myDomain] -> *0x1000 = x
```

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

```
[myDomain] -> x += 300
```

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

```
[myDomain] -> *0x1000 += 300
```

The following compound operators are available:

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

3.4.2 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:

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

3.5 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.

3.6 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**:

```
[myDomain] -> x = "hello there"
```

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:

```
[myDomain] -> printf ("hello there")
```

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

After extended development sessions, the cumulative memory used for strings may be noticeable. If this becomes a problem, restart your target server.

3.7 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

```
[myDomain] -> 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

```
[myDomain] -> 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 **&**.

3.8 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:

```
[myDomain] -> *(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.

3.9 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.

3.10 C Interpreter Primitives

3.10.1 Managing Tasks

Table 3-5 summarizes the commands that manage tasks.

Table 3-5 **C Interpreter Task Management Commands**

Command	Description
sp( )	Spawns a task with default parameters.
sps( )	Spawns a task, but leaves it suspended.
tr( )	Resumes a suspended task.
ts( )	Suspends a task.
td( )	Deletes a task.
period( )	Spawns a task to call a function periodically.
repeat( )	Spawns a task to call a function repeatedly.
taskIdDefault( )	Sets or reports the default (current) task ID.

The **repeat( )** and **period( )** commands spawn tasks whose entry points are **_repeatHost** and **_periodHost**. The shell downloads these support routines when you call **repeat()** or **period()**. These tasks may be controlled like any other tasks on the target; for example, you can suspend or delete them with **ts( )** or **td( )** respectively.

Table 3-6 summarizes the commands that report task information.

Table 3-6 **C Interpreter Task Information Reporting Commands**

Command	Description
i( )	Displays system information. This command gives a snapshot of what tasks are in the system, and some information about each of them, such as state, PC, SP, and TCB address. To save memory, this command queries the target repeatedly; thus, it may occasionally give an inconsistent snapshot.

Table 3-6 **C Interpreter Task Information Reporting Commands** (cont'd)

Command	Description
iStrict( )	Displays the same information as **i( )**, but queries target system information only once. At the expense of consuming more intermediate memory, this guarantees an accurate snapshot.
ti( )	Displays task information. This command gives all the information contained in a task's TCB. This includes everything shown for that task by an **i( )** command, plus all the task's registers, and the links in the TCB chain. If *task* is 0 (or the argument is omitted), the current task is reported on.
w( )	Prints a summary of each task's pending information, task by task. This routine calls **taskWaitShow( )** in quiet mode on all tasks in the system, or on a specified task if the argument is given.
tw( )	Prints information about the object the given task is pending on. This routine calls **taskWaitShow( )** on the given task in verbose mode.
checkStack( )	Shows a stack usage summary for a task, or for all tasks if no task is specified. The summary includes the total stack size (SIZE), the current number of stack bytes (CUR), the maximum number of stack bytes used (HIGH), and the number of bytes never used at the top of the stack (MARGIN = SIZE - HIGH). Use this routine to determine how much stack space to allocate, and to detect stack overflow. This routine does not work for tasks that use the **VX_NO_STACK_FILL** option.
tt( )	Displays a stack trace.
taskIdFigure( )	Reports a task ID, given its name.

The **i( )** command is commonly used to get a quick report on target activity. If nothing seems to be happening, **i( )** is often a good place to start investigating. To display summary information about all running tasks, type the following:

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

3.10.2 Displaying System Information

Table 3-7 shows the commands that display information from the symbol table, from the target system, and from the shell itself.

Table 3-7 **C Interpreter System Information Commands**

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

Table 3-7 **C Interpreter System Information Commands** (cont'd)

Command	Description
printLogo( )	Displays the shell logo.

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

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

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

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

3.10.3 Modifying and Debugging the Target

Developers often need to change the state of the target, whether to run a new version of some software module, to patch memory, or simply to single-step a program. Table 3-8 summarizes the commands of this type.

Table 3-8 **C Interpreter System Modification and Debugging Commands**

Command	Description
ld( )	Loads an object module into target memory and links it dynamically into the run-time.
unld( )	Removes a dynamically linked object module from target memory, and frees the storage it occupied.

Table 3-8 **C Interpreter System Modification and Debugging Commands** (cont'd)

Command	Description
m( )	Modifies memory in *width* (byte, short, or long) starting at *addr*. The **m( )** command displays successive words in memory on the terminal; you can change each word by typing a new hex value, leave the word unchanged and continue by typing **ENTER**, or return to the shell by typing a dot (**.**).
mRegs( )	Modifies register values for a particular task.
b( )	Sets or displays breakpoints, in a specified task or in all tasks.
bh( )	Sets a hardware breakpoint.
s( )	Steps a program to the next instruction.
so( )	Single-steps, but steps over a subroutine.
c( )	Continues from a breakpoint.
cret( )	Continues until the current subroutine returns.
bdall( )	Deletes all breakpoints.
bd( )	Deletes a breakpoint.
reboot( )	Returns target control to the target boot ROMs, then resets the target server and reattaches the shell.
bootChange( )	Modifies the saved values of boot parameters.
sysSuspend( )	If supported by the target agent configuration, enters system mode.
sysResume( )	If supported by the target agent (and if system mode is in effect), returns to task mode from system mode.
agentModeShow( )	Shows the agent mode (*system* or *task*).
sysStatusShow( )	Shows the system context status (*suspended* or *running).*
quit( ) or **exit( )**	Dismisses the shell.

The **m( )** command provides an interactive way of manipulating target memory.

The remaining commands in this group are for breakpoints and single-stepping. You can set a breakpoint at any instruction. When that instruction is executed by an eligible task (as specified with the **b( )** command), the task that was executing on the target suspends, and a message appears at the shell. At this point, you can examine the task's registers, do a task trace, and so on. The task can then be deleted, continued, or single-stepped.

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

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

3.11 Running Target Routines from the Host Shell

All target routines are available from the host shell. This includes both VxWorks routines and your application routines. Thus the shell provides a tool for testing and debugging your applications using all the host resources while having minimal impact on how the target performs and how the application behaves.

Invocations of VxWorks Subroutines

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

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

Invocations of Application Subroutines

```
-> testFunc (123)
value = …

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

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

Resolving Name Conflicts Between Host and Target

If you invoke a name that stands for a host shell command, the shell always invokes that command, even if there is also a target routine with the same name. Thus, for example, **i( )** always runs on the host, regardless of whether you have the VxWorks routine of the same name linked into your target.

However, you may occasionally need to call a target routine that has the same name as a host shell command. The shell supports a convention allowing you to make this choice: use the single-character prefix **@** to identify the target version of any routine. For example, to run a target routine named **i( )**, invoke it with the name **@i( )**.

3.12 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 Tcl Interpreter

4.1 Using the Tcl Interpreter

The Tcl interpreter allows you to access the WTX Tcl API, and to exploit Tcl's sophisticated scripting capabilities to write complex scripts to help you debug and monitor your target.

To switch to the Tcl interpreter from another mode, type a question mark (**?**) at the prompt; the prompt changes to **tcl>** to remind you of the shell's new mode. If you are in another interpreter mode and want to use a Tcl command without changing to Tcl mode, type a **?** before your line of Tcl code.

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

4.1.1 Accessing the WTX Tcl API

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

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

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

4.2 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:

```
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.

The Host Shell can also execute Tcl scripts when events are encountered; see *8. Eventpoint Scripting*.

5 Using the GDB Interpreter

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

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 *5.9 Wind River-Specific GDB Commands*, p.63.

5.1 General GDB Commands

This section lists general commands available within the GDB interpreter.

5.1.1 HELP

Syntax

`help` *command*

Prints a description of *command*.

5.1.2 CD

Syntax

`cd` *directory*

Changes the working directory.

5.1.3 PWD

Syntax

`pwd`

Prints the working directory.

5.1.4 PATH

Syntax

`path` *pathname*

Appends *pathname* to the **path** variable.

5.1.5 SHOW PATH

Syntax

`show path`

Shows the **path** variable.

5.1.6 ECHO

Syntax

`echo` *string*

Echoes the string.

5.1.7 LIST

Syntax

`list` *line* | *symbol* | *file*:*line*

Displays 10 lines of a source file, centered around a line number or symbol.

5.1.8 SHELL

Syntax

`shell` *command*

Runs a shell command (such as **ls** or **dir**).

5.1.9 SOURCE

Syntax

`source` *scriptfile*

Runs a script of GDB commands.

5.1.10 DIRECTORY

Syntax

`directory` *dir*

Appends *dir* to the **directory** variable (for source file searches.)

5.1.11 QUIT

Syntax

`q` `or` **`quit`**

Quits the Host Shell.

5.2 Working with Breakpoints

This section lists commands available for setting and manipulating breakpoints.

5.2.1 BREAK

Syntax

`break` *symbol* | *line* | *file*:*line* [`if expr`]

or

`b` *symbol* | *line* | *file*:*line* [`if expr`]

Sets a breakpoint.

5.2.2 TBREAK

Syntax

tbreak *symbol* | *line* | *file*:*line* [`if expr`]

or

t *symbol* | *line* | *file*:*line* [`if expr`]

Sets a temporary breakpoint.

5.2.3 ENABLE

Syntax

enable *breakpoint_id*

Enables a breakpoint. Takes optional arguments **once** and **del**.

5.2.4 DISABLE

Syntax

disable *breakpoint_id*

Disables a breakpoint.

5.2.5 DELETE

Syntax

delete *breakpoint_id*

Deletes a breakpoint.

5.2.6 CLEAR

Syntax

clear *breakpoint_id*

Clears a breakpoint.

5.2.7 COND

Syntax

cond *breakpoint_id condition*

Changes a breakpoint condition (re-initializes the breakpoint).

5.2.8 IGNORE

Syntax

ignore *breakpoint_id n*

Ignores a breakpoint *n* times (re-initializes the breakpoint).

5.3 Specifying Files to Debug

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

5.3.1 FILE

Syntax

file *filename*

Defines *filename* as the program to be debugged.

5.3.2 EXEC-FILE

Syntax

exec-file *filename*

Specifies that the program to be run is found in *filename*.

5.3.3 LOAD

Syntax

load *filename*

Loads a module.

5.3.4 UNLOAD

Syntax

unload *filename*

Unloads a module.

5.3.5 ATTACH

Syntax

attach *process_id*

Attaches to a process.

5.3.6 DETACH

Syntax

detach

Detaches from the debugged process.

5.3.7 THREAD

Syntax

thread *thread_id*

Selects a thread as the current task to debug.

5.3.8 ADD-SYMBOL-FILE

Syntax

`add-symbol-file` *filename addr*

Reads additional symbol table information from the file located at memory address *addr*.

5.4 Running and Stepping Through a File

This section lists commands to run and step through programs.

5.4.1 RUN

Syntax

`run`

Runs a process for debugging (use **set arguments** and **set environment** if program needs them).

5.4.2 KILL

Syntax

`kill` *process_id*

Kills a process.

5.4.3 INTERRUPT

Syntax

`interrupt`

Interrupts a running task or process.

5.4.4 CONTINUE

Syntax

`continue`

Continue an interrupted task or process.

5.4.5 STEP

Syntax

`step` [*n*]

Step one instruction. If *n* is used, step *n* times.

5.4.6 STEPI

Syntax

`stepi` [*n*]

Step one assembly-language instruction. If *n* is used, step *n* times.

5.4.7 NEXT

Syntax

`next` [*n*]

Continue to the next source line in the current stack frame. If *n* is used, continue through *n* lines.

5.4.8 NEXTI

Syntax

`nexti` [*n*]

Execute one assembly-language instruction. If the instruction is a function call, proceed until the function returns. If *n* is used, execute *n* instructions.

5.4.9 UNTIL

Syntax

`until`

Continue running until a source line past the current line in the current stack frame is reached.

5.4.10 JUMP

Syntax

`jump` *address*

Move the instruction pointer to *address*.

5.4.11 FINISH

Syntax

`finish`

Finish execution of current block.

5.5 Displaying Disassembly and Memory Information

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

5.5.1 DISASSEMBLE

Syntax

`disassemble` *address*

Disassemble code at a specified address.

5.5.2 X

Syntax

`x` [/*format*] *address*

Display memory starting at *address*.

format 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**.

5.6 Examining Stack Traces and Frames

This section lists commands for selecting and displaying stack frames.

5.6.1 BT

Syntax

`bt` [*n*]

Display back trace of *n* frames.

5.6.2 FRAME

Syntax

`frame` [*n*]

Select frame number *n*.

5.6.3 UP

Syntax

`up` [*n*]

Move *n* frames up the stack.

5.6.4 DOWN

Syntax

`down` [*n*]

Move *n* frames down the stack.

5.7 Displaying Information and Expressions

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

5.7.1 INFO

Syntax

`info` *option*

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

- **args** - Shows function arguments.
- **breakpoints** - Shows breakpoints.
- **extensions** - Shows file extensions (c, c++, ...)
- **functions** - Shows all functions.
- **locals** - Shows local variables.
- **registers** - Shows contents of registers.
- **source** - Shows current source file.

- **sources** - Shows all source files of current process.
- **target** - Displays information about the target.
- **threads** - Shows all threads.
- **warranty** - Shows disclaimer information.

5.7.2 PRINT /X

Syntax

`print /x` *expression*

Evaluate and print an expression in hexadecimal format.

5.8 Displaying and Setting Variables

This section lists commands for displaying and setting variables.

5.8.1 SET ARGS

Syntax

`set args` *arguments*

Specify the arguments to be used the next time a debugged program is run.

5.8.2 SET EMACS

Syntax

`set emacs`

Set display to emacs mode.

5.8.3 SET ENVIRONMENT

Syntax

`set environment` *varname =value*

Set environment variable *varname* to *value*. *value* may be any string interpreted by the program.

5.8.4 SET TGT-PATH-MAPPING

Syntax

`set tgt-path-mapping`

Set target to host pathname mappings.

5.8.5 SET VARIABLE

Syntax

`set variable` *expression*

Set variable value to *expression*.

5.8.6 SHOW ARGS

Syntax

`show args`

Show arguments of debugged program.

5.8.7 SHOW ENVIROMENT

Syntax

`show environment`

Show environment of debugged program.

5.9 Wind River-Specific GDB Commands

5.9.1 TARGET OCD

target ocd spawns a backend server, connects to it, and connects 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.

5.9.2 WRSDEFTARGET

wrsdeftarget creates 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 5-1.

Table 5-1 **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: ▪ **visionICE II** - Connects to a visionICE II tool. ▪ **Wind River ICE** - Connects to a Wind River ICE SX tool (also called WINDPOWER ICE.) For this **DEVICE** type, the **BFNAME** parameter is required; for other **DEVICE** types, the **BFNAME** parameter is optional. ▪ **visionPROBE II** - Connects to a visionPROBE II tool. ▪ **Wind River ISS** - Connects to a 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 **PARALLEL**.
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 **PARALLEL**, the value of **ADDR** is either of the keywords LPT1 or LPT2.
BFNAME	Specifies the host pathname of the board descriptor file.

Example

```
wrsdeftarget mytarget --core MPC8260 --cpuplugin 82xxcpuplugin DEVICE="Wind
River ICE" STYLE=ETHERNET ADDR=128.224.50.236
```

If the Host Shell is not connected to a backend server when you issue this command, the Host Shell will spawn a backend server and connect 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.

5.9.3 WRSDOWNLOAD

The **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.)

Download Executables and Data

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

Syntax 1

wrsdownload [{**--offset** | **-o**} *byte_offset*] [{**--modulename** | **-m**} *modulename*]
[**--symbolsonly** | **-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 **--symbolsonly** 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 file to download.

Erase Flash Memory

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

Program flash memory on the target using the following syntax:

Syntax 3

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

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

5.9.4 WRSMEMMAP

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 specifed, 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.

5.9.5 WRSPASSTHRU

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*.

5.9.6 WRSPLAYBACK

Play a file of commands directly to the firmware.

Syntax

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

pathname identifies 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.

5.9.7 WRSREGQUERY

wrsregquery queries the Wind River registry to obtain target definition information. Target definitions can later be given to the **wrsdeftarget** command (see *5.9.2 WRSDEFTARGET*, p.63) 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.

5.9.8 WRSRESET

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.

5.9.9 WRSUPLOAD

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.

6 Single Step Compatibility

6.1 Overview

This chapter describes backward compatibility for Wind River SingleStep scripting.

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, etc.) 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.

6.2 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 *6.6 SingleStep Variable Compatibility*, p.79.

6.3 SingleStep Command Equivalents

Table 6-1 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 6-1 **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 6-1 **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 6-1 **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 6-1 **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**(*varname*) *value* (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** (GBD 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 6-1 **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** *varname* (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).

6.4 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 6-2 shows how various SingleStep **read** commands map to Host Shell GDB mode commands.

Table 6-2 **SingleStep read Command Compatibility**

SingleStep read Command	Description	Host Shell Equivalent
read *x*	Display value of variable *x*.	print *x*
read *x y z*	Display values of three variables.	GDB mode: print *x*; print *y*; print *z* Tcl mode: foreach var {*x y z*} {eval "puts \$$var"}
read -1#*arg*	Display variable *arg* from first function on stack.	None.
read file.c#*var*	Display static variable *var* from file.c.	print file.c:*var*
read main	Disassemble starting at main.	disassemble main
read 0x4000	Dump starting at address 0x4000.	x /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.	x /1xb 0x7e02
read -F 0x4000	Disassemble starting at 0x4000.	disassemble 0x4000
read *var*=long	Display variable *var* as if it were a long.	print (long)*var*
read 0x120=(*sym*)	Display 0x120 using type from variable *sym*.	None.
read *p	Display whatever **p** points to.	print *p
read a[5]	Display the fifth element of array **a**.	print [a]5
read str.mem	Display member **mem**.	print str.mem

Table 6-2 **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.

6.5 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 6-3 shows how various SingleStep **read** commands map to Host Shell GDB mode commands.

Table 6-3 **SingleStep write Command Compatibility**

SingleStep write Command	Description	Host Shell Equivalent
write *var*=99	Write value 99 to variable *var*.	set *var*=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 6-3 **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

6.6 SingleStep Variable Compatibility

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 6-4 **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 6-4 **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

7 Executing an OCD Reset and Download

7.1 Overview

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 = 0xfff000f0

  Set the Program Counter to address 0xfff000f0.

- *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 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*]
[**--symbolsonly** | **-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 **--symbolsonly** 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 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 tarrget-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 specifed, 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.

8 Eventpoint Scripting

8.1 Overview

The Host Shell has the ability to execute a Tcl script 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 not 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

```
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**.

8.2 Detailed API Description

The APIs to register and enable/disable handlers and to list all registered handlers are accessible from the Cmd and GDB interpreters.

8.2.1 Cmd Interpreter

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 will trigger the handler. By default, this is the stopped event.

-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 Host Shell.

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.

8.2.2 GDB Interpreter

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.

8.3 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 **sp()** 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 0x0 0
^done,thread-id="39"
[vxWorks *]# taskWatchHandler Task 1617033120
TASK EXITED
```

8.4 Example Cmd Session

```
[vxWorks *]# handler add -n
Type Tcl script to be executed when event is encoutered.
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        yes            ALL
                             shEval cmd task
                                shEval cmd c

--- -------------------- -------------------- ---------- -------------
[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 *]#
```

In this example, a handler has been added that overrides the default breakpoint handler. The handler calls the Cmd 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.

8.5 Example GDB Session

```
(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 Enb Expression
0    y  dummy1
1    y  dummy2
(gdb)
```

This example downloads an RTP and sets a breakpoint within that RTP. The user then calls the **commands** API, 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 the **display** API, 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.

9 Using the Host Shell Line Editor

9.1 Overview

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.

9.2 vi-Style Editing

The **ESC** key switches the shell from normal input mode to edit mode. The history and editing commands in Table 9-1 and Table 9-3 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 9-2; you can then edit and execute the line with one of the commands from the table.

9.2.1 Switching Modes and Controlling the Editor

Table 9-1 lists commands that give you basic control over the editor.

Table 9-1 **vi-Style Basic Control Commands**

Command	Description
h [*size*]	Displays shell history if no argument is given; otherwise sets history buffer to *size*.
ESC	Switch to line editing mode from regular input mode.
ENTER	Give line to current interpreter and leave edit mode.
CTRL+D	Complete symbol or pathname (edit mode), display synopsis of current symbol (symbol must be complete, followed by a space), or end shell session (if the command line is empty).
[tab]	Complete symbol or pathname (edit mode).
CTRL+H	Delete a character (backspace).
CTRL+U	Delete entire line (edit mode).
CTRL+L	Redraw line (edit mode).

Table 9-1 **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.

9.2.2 Moving and Searching in the Editor

Table 9-2 lists commands for moving and searching in input mode.

Table 9-2 **vi-Style Movement and Search Commands**

Command	Description
*n***G**	Go to command number *n*. The default value for *n* is 1.
/*s* or **?***s*	Search for string *s* backward or forward in history.
n	Repeat last search.
*n***k** or *n***-**	Get *n*th previous shell command.
*n***j** or *n***+**	Get *n*th next shell command.
*n***h**	Go left *n* characters (also **CTRL+H**).
*n***l** or **SPACE**	Go right *n* characters.
*n***w** or *n***W**	Go *n* words forward, or *n* large words. *Words* are separated by spaces or punctuation; *large words* are separated by spaces only.
*n***e** or *n***E**	Go to end of the *n*th next word, or *n*th next large word.
*n***b** or *n***B**	Go back *n* words, or *n* large words.
$	Go to end of line.
0 or **^**	Go to beginning of line, or to first nonblank character.
f*c* or **F***c*	Find character *c*, searching forward or backward.

9.2.3 Inserting and Changing Text

Table 9-3 lists commands to insert and change text in the editor.

Table 9-3 **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).
*n***s** ...**ESC**	Change *n* characters (**ESC** ends input).
cw ...**ESC**	Change word (**ESC** ends input).
cc or **S** ...**ESC**	Change entire line (**ESC** ends input).
c$ or C ...**ESC**	Change from cursor to end of line (**ESC** ends input).
c0 ...**ESC**	Change from cursor to beginning of line (**ESC** ends input).
R ...**ESC**	Type over characters (**ESC** ends input).
*n***r***c*	Replace the following *n* characters with *c*.
~	Toggle between lower and upper case.

9.2.4 Deleting Text

Table 9-4 shows commands for deleting text.

Table 9-4 **vi-Style Commands for Deleting Text**

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

9.2.5 Put and Undo Commands

Table 9-5 shows put and undo commands.

Table 9-5 **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.

9.3 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**.

9.3.1 Moving the Cursor

Table 9-6 lists commands for moving the cursor in emacs mode.

Table 9-6 **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 9-6 **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.

9.3.2 Deleting and Recalling Text

Table 9-7 shows commands for deleting and recalling text.

Table 9-7 **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.
!*n*	Recall command *n* from the history.
!*substr*	Recall first command from the history matching *substr*.

9.3.3 Special Commands

Table 9-8 shows some special emacs-mode commands.

Table 9-8 **Special emacs-Style Commands**

Command	Description
CTRL+U	Delete line and leave edit mode.

Table 9-8 **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.

9.4 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.

9.4.1 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.

9.4.2 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.

Index

E

G

H

I

L

M

O

R

S

T

U

V