WIND RIVER

Wind River® Workbench

MIGRATION GUIDE

2.5

Copyright © 2006 Wind River Systems, Inc.

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

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

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

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

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

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

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

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

http://www.windriver.com

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

http://www.windriver.com/support

Wind River Workbench Migration Guide, 2.5

25 Apr 06
Part #: DOC-15781-ZD-00

Contents

PART 2: MIGRATING WTX APPLICATIONS

PART 1

Migrating to Wind River Workbench

1 Overview

1.1 Introduction

NOTE: If you have already migrated from the Tornado tools to Workbench ,there should be no issues involved in opening your projects in Workbench 2.5. For information on using Workbench 2.5 with multiple versions of VxWorks, see *2.6 Using Workbench 2.5 With Multiple Versions of VxWorks*, p.14. For information on migrating your applications to VxWorks 6.3, see the *VxWorks Migration Guide*.

The *Wind River Workbench Migration Guide* is intended for several groups of developers:

- Application developers who used the Tornado 2.2 IDE and who will move to the Wind River Workbench development suite.
- Application developers who used the VxWorks command line and who will move to the Wind River Workbench graphical user interface (GUI).
- Developers responsible for configuring VxWorks 5.5 who will use Workbench to configure VxWorks 6.1.

- Tool developers migrating tools that worked with Tornado 2.2 to Workbench.

Part I of this Migration Guide contains information about how to carry out tasks that you used to do in the Tornado 2.2 IDE in Workbench. This includes configuring VxWorks; building VxWorks and applications; and downloading, running, and debugging applications. It also includes information on bringing existing Tornado, SNiFF+, and Workbench projects into Wind River Workbench 2.5.

Part II contains information about WTX that is required by developers moving Tornado 2.2.x tools to Workbench. Additional information about the Eclipse infrastructure is available from the Eclipse documentation included in Workbench, located in the Workbench GUI at
Help > Help Contents > Wind River Documentation > Eclipse Platform Documentation > Eclipse Workbench User Guide.

The *Wind River Workbench Migration Guide* does not contain information about new features of VxWorks unless they relate to features that have been replaced (such as the debugger). All new features are described fully in either the *VxWorks Application Programmer's Guide*, the *VxWorks Kernel Programmer's Guide*, or the *VxWorks Network Stack Programmer's Guide*.

The *Wind River Workbench Migration Guide* also does not contain information on non-Workbench components, including sets of real-time technologies such as Wind River USB peripheral stack or Wind River IPv6.

For information about these components, see your product *Getting Started*.

1.2 Summary of Changes

- Product name changes:
  - Wind River System Viewer is the new name for WindView.
  - Wind River Workbench is the new name for the Tornado.
  - Wind River Compiler is the new name for the Diab compiler.
- The Wind River Workbench user interface is based on Eclipse. For more information, see the *Wind River Workbench User's Guide*.

  - Workbench 2.2 can be migrated directly to Workbench 2.5. No changes are necessary.
  - Tornado 2.2 projects can be migrated directly to Workbench 2.5. For more information, see *2.2 Tornado 2.2 Projects*, p.9.
  - SNiFF+ projects (versions 4.1, 4.1.1, and 4.2) can be migrated to Workbench 2.3. For more information, see *2.3 SNiFF+ and SNiFF+ Pro Projects*, p.10.
  - A command table is provided for the new MultiX debugger, showing the old and new methods of performing common debugging functions. See *3.6 Debugger*, p.22.

- A new development shell is available for both GUI and command-line users. It is documented in the *Wind River Workbench Command-Line Developer's Guide* and can be started from the Windows **Start** menu by selecting **Start > Programs > Wind River > VxWorks 6.x and General Purpose Technologies > VxWorks 6.x Development Shell** or by typing the following on the command line:

  ```
  % installDir/wrenv.sh -p vxworks-6.x
  ```

- The build infrastructure has changed from Tornado. If you use the default build method, these changes will be transparent to you. If you have a customized build system, see your product *Getting Started: Compilers*. For more general build issues, see the *Wind River Workbench User's Guide* and the *Wind River Workbench Command-Line User's Guide*.

1.3 Terminology

VxWorks Terms

Real-time process (RTP):
: The user-mode application execution environment provided by VxWorks 6.x. Each process has its own address space, containing the executable program, the program's data, stacks for each task, the heap, and resources associated with the management of the process itself.

Project:
: A collection of source code, binary files, and build specifications.

Workspace:
A collection of projects.

Component:
A scalable VxWorks facility.

Error detection and reporting (ED&R)
The facility for preserving error information across a warm reboot.

Types of Projects

VxWorks image project:
The project type used to configure and build VxWorks images.

Downloadable kernel module project:
The project type used to manage and build dynamically linked application modules that run in kernel mode.

Real-time process project:
The project type used to manage and build statically and dynamically linked application modules into an executable that can be dynamically downloaded and run as a real time process (RTP).

Shared-library project:
The project type used to manage and build dynamically linked shared libraries.

Tool-related Terms

Toolchain:
A collection of development tools for a specific target CPU; includes compiler, linker, assembler, and so on.

Build specification:
A build context with toolchain- and architecture-specific settings.

Graphical User Interface (GUI)
The interactive, graphical face of Workbench. Workbench also offers a command-line interface.

Component Terms

Real-time technologies
An equivalent term for middleware components. A set of real-time technologies refers to a set of middleware components.

2

Workbench Projects

2.1 Introduction

The project concept has changed in Workbench from previous products. In Tornado, a project contained both the list of files included in the project and build specifications for various toolchains. In Workbench only the build specifications are stored; the list of files is no longer a part of the project. Instead, specifying the root directory in the workspace automatically causes all files and directories below the root to be part of the project.

A variety of project types can be migrated into Workbench 2.5. If you have existing Tornado 2.2, SNiFF+, SNiFF+ Pro, Workbench 2.x, or WIND POWER IDE 1.x projects, they can easily be brought into the Workbench 2.5 environment.

Tornado 2.2 Projects

Both kernel projects and application projects can be migrated from Tornado 2.2 to Workbench. For kernel projects, a command-line tool is available; for application projects an **Import** wizard is available in the GUI.

SNiFF+ and SNiFF+ Pro Projects

An **Import** wizard allows you to bring existing SNiFF+ projects into Wind River Workbench. You can bring the SNiFF+ project into Workbench in two different ways:

1. You can leave the sources at their origin, resulting in one Workbench project containing everything inside the SNiFF+ project root directory.
2. You can restructure the whole SNiFF+ project tree into a single Workbench project with absolute or workspace-relative paths, or create a Workbench project tree structure.

The projects can be converted either into downloadable kernel module, real-time process, or shared-library projects. For more information, see *2.3 SNiFF+ and SNiFF+ Pro Projects*, p.10.

WIND POWER IDE 1.x Projects

Another **Import** wizard imports WIND POWER IDE 1.x projects into Workbench. The migration process is similar to Tornado 2.2 application projects and allows you to convert the project into a downloadable kernel module, a real-time process, or a shared-library project.

For more information, see *2.4 WIND POWER IDE 1.x Projects*, p.12.

Workbench 2.0 and 2.1 to Workbench 2.5

Portable application projects written for Wind River Workbench for Linux can be imported to Workbench by sharing the project in a CVS repository or by copying the project from the old workspace to the new one.

For more information, see *2.5 Workbench 2.0/2.1 to Workbench 2.5 Projects*, p.12

Previous Workbench to the Current Version

Projects from any previous version of Workbench can be opened in the current version of Workbench without changes.

2.2 Tornado 2.2 Projects

Tornado 2.2 can be migrated to Wind River Workbench using one of two methods:

1. VxWorks image projects are migrated using the **tcMigrate** command-line tool and then imported to Workbench if desired.

- Projects where the application was not linked with VxWorks can be migrated using the appropriate **Import** wizard.

2.2.1 Migrating Kernel Projects

Migrate Tornado 2.2 VxWorks image projects to Workbench using the **tcMigrate** tool from the command line, then create a new Workbench project based on your migrated Tornado project.

1. **tcMigrate** *myProject*
2. Start Workbench.
3. Create a new VxWorks image project:
   a. On the first panel, name your project, and change the default location if you wish.
   b. On the second panel, select **An existing VxWorks Image Project** and browse to find and select the **.wpj** file you created. (Even if you cannot select the radio button, you will still be able to browse.)
   c. Click on **Finish**.

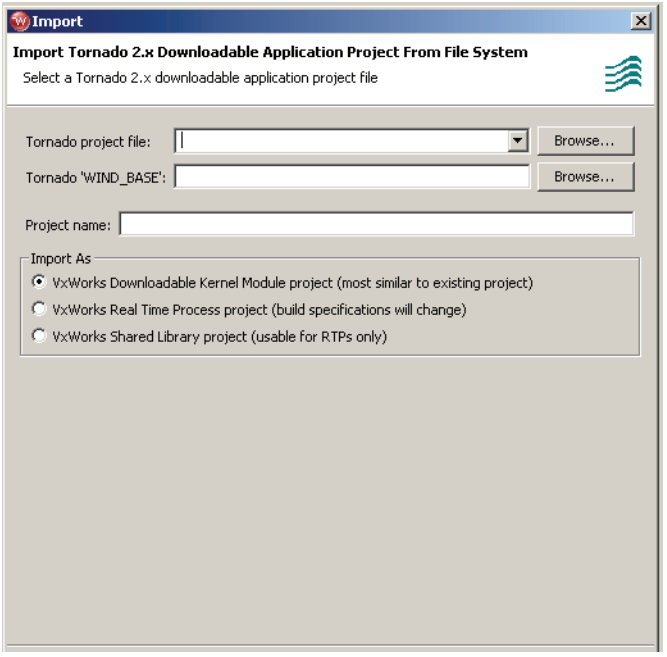
2.2.2 Migrating Application Projects

Import Tornado 2.2 application projects using the **Import** wizard.

1. Select **File > Import > Existing Tornado 2.x projects** to start the wizard.
2. Select one of the following project types:
   – downloadable kernel module project
   – real-time process project
   – shared-library project

Figure 2-1 shows the **Import** wizard.

Figure 2-1 **Application Import Wizard**



2.3 SNiFF+ and SNiFF+ Pro Projects

For application project development, you can select which type of project you want to import: a downloadable kernel module, a real-time process, or a shared library project. For SNiFF+ projects, you can decide how you want the project to be converted, in other words, whether you want to copy the source code into the project as is done with a Tornado project.

The wizard is accessible using **File > Import > Existing SNiFF+ Project into Workspace**. It allows you to import whole project tree structures as they are usually used within SNiFF+. To specify the root input project, you must declare the SNiFF+ **.proj** project file, the source root location (as the project file need not necessarily be stored along with the sources) plus, optionally, the project and

makefile location, which could be the common root directory for all project files and makefiles of a potential SNiFF+ project tree.

NOTE: There are two types of SNiFF+ import options: Migrate SNiFF+ Native Project and Migrate SNiFF+ VxWorks 5.x Project to VxWorks 6.x

Once this information is provided, click **Update** to populate a file-system-like tree viewer, from which you can select the contents of the project or projects you are generating. Filtering possibilities and selection help is provided within the wizard as well.

You can choose to leave sources at their origin and import everything beneath the source root location, which results in one Workbench project containing everything from the SNiFF+ project root directory. (For more information on creating projects, see the *Wind River Workbench User's Guide* or the *Wind River Workbench Command-Line User's Guide*.)

You can also choose to copy the resources into the workspace. In this case, the wizard offers options for overwriting existing resources and handling SNiFF+ project tree structures. You can create either a single Workbench project with absolute or workspace-relative paths, or a Workbench project tree structure with unified project names. Due to project tree structure handling, any project name you specify may be changed, but you are free to rename all created projects at a later time. Projects are converted into downloadable kernel module, real-time process, or shared-library projects.

During import, depending on the behavior you have selected, one or a set of projects, will be created and structured in one step. If sources are copied, any original SNiFF+ makefiles will be imported (using their original name appended with an additional **.sniff** extension) as well as all **.proj** files. For each created project, a default build target is created. Within this wizard, no build specification mapping is done, and the default build spec of the selected project type is set for all created projects. Also, any other build settings within an original project are ignored for a created project but can be specified manually in the same way as is described for Workbench 2.x application project import. (See *2.5.2 Build Property Update*, p.13.)

For particular use cases, it might make sense to restructure the SNiFF+ project's file system to correspond better to Eclipse workspace concepts. It might be necessary to move projects out of other projects to the same directory level if you want separate projects to be generated, as Eclipse does not allow projects to be created in any file system leaf where another project already exists. (For more information about creating projects, see the *Wind River Workbench User's Guide: Projects Overview*.)

2.4 WIND POWER IDE 1.x Projects

The wizard is accessible using
File > Import > Migrate WIND POWER IDE VxWorks 5.x Project to VxWorks 6.x.

The wizard offers the same functionality as the Tornado 2.2 application project wizard, except the ability to specify **WIND_BASE**, which is not needed for this type of project. The project file of a **WIND POWER IDE 1.x** project is the **.mproj** file rather than the **.wpj** file used by Tornado. For more information, see the *Wind River Workbench User's Guide*.

The import itself is very much like importing Tornado 2.2 application projects, but instead of the **.wpj** file from Tornado, the **.mk** file and the **.mproj** file from **WIND POWER IDE 1.x** are imported. In addition, the default build spec is determined by the CPU and TOOLSET information used in the original project.

2.5 Workbench 2.0/2.1 to Workbench 2.5 Projects

Workbench 2.0/2.1 supported only Linux, not VxWorks. However, you may have portable C or C++ applications that you wish to migrate to Workbench 2.3.

2.5.1 Migrating the Project

There are two ways of migrating a project:

1. *Share The Project*, p.12.
2. *Copy The Project*, p.13.

Share The Project

There are no compatibility issues here. Share your project using your configuration management (CM) system. The documentation of your CM plug-in to Eclipse should document how to do this for your particular CM system.

Copy The Project

Copy the projects from the old workspace into the new one and import them with the corresponding wizard (**File > Import > Existing project into Workspace**). If the new workspace resides in the same location as the old one, remove only the **.metadata** directory within the workspace root to eliminate the old IDE properties stored there, since they differ fundamentally due to the switch from Eclipse 2.1.1 to Eclipse 3.0. Be aware that this disables the ability to switch back from Workbench 2.5 to Workbench 2.0/1 (unless you rename the two **.metadata** directories each time you want to switch).

2.5.2 Build Property Update

It is possible to update the build properties of existing Workbench 2.0/1 projects to Workbench 2.5 build properties (which might make sense if you want to have correct build dependency handling and did not change many of the build properties).

Import the build properties with the corresponding build settings import wizard. This overwrites everything regarding build properties except defined build targets, so you do not lose them. If you changed only a few properties, you might be able to migrate them manually. Apply the previous settings manually, by entering them in the project build properties. (Right-click the project and select **Project > Properties > Build Properties**.) For safety reasons, you should back up the original **.wrproject** and **.wrmakefile** files.

2.5.3 Compatibility Issues

Workbench 2.5 is fully backward compatible with Workbench 2.x. Workbench 2.5 is able to handle all Workbench 2.x projects.

2.5.4 Enhancements

Old versions of the GNU compilers (used with the **PPC82xx-gnu-eldk2.1** build specification) are not able to produce a dependency file usable within the make process. This issue is now solved explicitly with the new **ADAPT_DEPS** macro.

2.6 Using Workbench 2.5 With Multiple Versions of VxWorks

This section discusses which combinations of Wind River Workbench and VxWorks 6.x are supported and highlights some limitations.

Workbench 2.5 is compatible withVxWorks 6.x. It can be used with any 6.x version, alone or simultaneously.

Similarly, existing projects that were under VxWorks 6.2 or earlier can be opened in Workbench 2.5.

NOTE: Users were previously required to update lines in the *installDir*/install.properties file manually, in order to enable a supported combination of Workbench and VxWorks. This is no longer necessary as Workbench will point to VxWorks 6.x automatically.

CAUTION: It is not possible to work with VxWorks 6.0 VIPs if you install VxWorks 6.1 in a different root directory than 6.0.

2.6.1 VxWorks Image Project Support

Workbench 2.5 is able to support existing VxWorks 6.0 image projects (VIPs) correctly as well as new VxWorks 6.3 VIPs.

2.6.2 Application Project Support (DKM, RTP and SL)

In Workbench 2.5, each application project type has a platform version associated with. You can change which VxWorks version to link against by changing this value in **Project > Properties > Target Operating System**; however, this cannot be done for VxWorks image projects.

NOTE: For versions of VxWorks earlier than 6.2, it is necessary to make the change of platform in DKM, RTP, and SL projects.

The build spec contents of a project do not change when changing the target operating system. If different build specs should be used for different OSs, we recommend that you either:

- create separate build specs manually

or

- use different workspaces, which means copying the projects.

Switching the platform version for any project can only be done if the project is not referenced by any other project and does not reference any other project itself. This is the same as saying that the project may not have subprojects and may not be a subproject. Otherwise, there would be obvious inconsistencies in how each project was linked.

If a whole tree of projects needs to be switched, you can change the **.wrproject** files manually; this requires restarting Workbench in order to pick up the changes. Add or change the **platform_name** value as follows:

```
platform="VxWorks"
platform_name="vxworks-6.3"
```

2.6.3 Migration From VxWorks 6.0/6.1 to 6.3

These considerations apply to installations with different root directories.

VIPs

The following steps are required to move your existing VIPs from VxWorks 6.0/6.1 to VxWorks 6.3:

1. First, delete the VIP from your workspace. (Do *not* delete the contents!)
2. Migrate the 6.0/6.1 VIP to VxWorks 6.3 using **tcMigrate** on the VxWorks 6.3 command line.
3. Import the migrated project into your workspace again using **File > Import > Existing VxWorks 6.x Image Project**.

DKMs, RPTs, and SLs

If only a VxWorks 6.3 environment is available, projects are migrated automatically. However, the original build spec contents will be kept (not overwritten).

If necessary, new build specs based on the VxWorks 6.3 installation can be created for each project by using **File > Import > Build Settings > Default Settings**. This

overwrites all existing build specs. Alternatively, you can add specific new build specs.

File System Projects

File system projects can be reused without any changes.

2.7 Migrating From Previous Versions of VxWorks 6.x

Workbench is fully backwards-compatible; thus any existing project created under Workbench 2.3 or 2.4 can be opened, without any changes, in the new Workbench environment.

3 Operation Equivalents

3.1 Introduction

This section provides information on how to perform common Tornado 2.2 operations using the Workbench GUI. The information is grouped into a number of functional sections.

3.2 General

Table 3-1 **General Parallels Between Tornado 2.2 and Wind River Workbench**

Use Case	Tornado 2.2	Wind River Workbench
Start GUI	Select **Start > Programs >** *installationName* **> Tornado**.	Select **Start > Programs > Wind River > Workbench 2.x > Wind River Workbench 2.x**

3.3 Project

Table 3-2 **Parallels Between Tornado 2.2 and Wind River Workbench Project Tools**

Use Case	Tornado 2.2	Wind River Workbench
Create kernel project	Select **File > New Project**. Click **New** tab. Select **Create a bootable VxWorks image** (custom configured). Click **OK**. Follow instructions on screen.	Select **File > New > VxWorks Image Project**. Follow instructions on screen.
Create downloadable project	Select **File > New Project**. Click **New** tab. Select **Create downloadable application modules for VxWorks**. Click **OK**. Follow instructions on screen.	Select **File > New > Downloadable Kernel Module Project**.

Table 3-2 **Parallels Between Tornado 2.2 and Wind River Workbench Project Tools** (cont'd)

Use Case	Tornado 2.2	Wind River Workbench
Add new file to project	**Select File > New**. Follow instructions on screen.	In **Project Navigator** view, right-click **Project**. Select **New > Other**. Expand **Simple** folder. Select **File**, click **Next**. Follow instructions on screen.
Add existing file to project	From **Workspace** right-click **Project**. Select **Add Files**. Navigate to file, click **OK**.	From **Project Navigator** view, right-click on **Project**. Select **Import > File** system, click **Next**. Click **Browse**. Navigate to the directory, click **OK**. Again, on the **Import** dialog, check the files to import. Click **Finish**.
Browse kernel components	From **Workspace**, click **VxWorks**. Expand the *projName* **VxWorks node**. Components are displayed with yellow "Lego" icon.	From **Project Navigator** view, expand kernel project node and double-click **Kernel Configuration**. The **Editor** *projName* view opens to the right of the **Project Navigator** view. Select the **Components** tab. Components are displayed with a blue cube icon.
Add/remove kernel components	Right-click **Component Name**. Select **Include** *componentName* or **Exclude** *componentName*.	No change from Tornado 2.2

Table 3-2 **Parallels Between Tornado 2.2 and Wind River Workbench Project Tools** (cont'd)

Use Case	Tornado 2.2	Wind River Workbench
Set component parameters	Right-click **Component**. Select **Properties**. Click **Params**. Click the *parameter name* and enter a new value in the input box above the list.	Double-click *Component* to expand the parameter list. Click the *parameter name* in the list and enter a new value in the *Value* column in the window below the tree.

3.4 Build

Table 3-3 **Parallels Between Tornado 2.2 and Wind River Workbench Build Systems**

Use Case	Tornado 2.2	Wind River Workbench
Set build parameters	Right-click **Component**. Select **Properties**. Click **Params**. Click the *parameter name* and enter new value in the input box above the list.	Right-click the project, file, or other build element in the **Project Navigator**. Select **Properties**, then set the build properties as appropriate for the item you chose. The options you see (build support, build specs) in the left-hand pane change depending on the type of build target.
Build project	From **Workspace** right-click on **Project Node**. Select **Build** *buildSpec*.	From **Project Navigator** view right-click **Project Name Node**. Select **Build Project**.
Rebuild project	From **Workspace** right-click on **Project Node**. Select **ReBuild All** *projType*.	From **Project Navigator** view right-click **Project Node**. Select **Rebuild Project**.

3.5 Target Server

Table 3-4 **Parallels Between Tornado 2.2 and Wind River Workbench Target Servers**

Use Case	Tornado 2.2	Wind River Workbench
Connect to new registry	Select **Tools > Options**. Click **Tornado Registry**. Follow instructions on screen.	Right-click **Target Manager** view. Select **New > Registry**.
Create target server	Select **Tools > Target Server > Configure**. Follow instructions on screen.	From **Target Manager** view right-click **Root Node**. Select **New > Connection**. Select **Wind River Target Server Connection**, click **Next**. Follow instructions on screen.
Connect target server	Select **Tools > Target Server > Configure**. Find target server, click **Launch**.	From **Target Manager** view expand the **Root Node**. Right-click on the target to connect, select **Connect**.
Kill target server	Select **Tools > Target Server > Manage**. From **Set Selection** drop-down list, select **Kill**. Click **Apply**.	From **Target Manager** view, right-click the chosen target server and select **Kill Target Server**.

3.6 Debugger

Table 3-5 **Parallels Between Tornado 2.2 and Wind River Workbench Debuggers**

Use Case	Tornado 2.2	Wind River Workbench
Debug a task	Click **Launch Debugger** button. Select **Debug > Run**. Make selections, click **OK**.	From the **Run menu**, select **Debug**. In the **Launch** dialog, highlight **Kernel Task** and click **New**. Enter the task parameters. When finished, click **Debug**.
Set/remove a breakpoint	Right-click on **Line to set breakpoint on**. Select **Toggle Breakpoint**. Breakpoints are shown as a red triangle pointing down.	Double-click the space next to the **Line to set breakpoint on**. Breakpoints are shown as a blue circle.
Enable/disable breakpoints	Right-click anywhere on **Source** view, select **Breakpoints**. **Check/uncheck** the breakpoint.	From **Breakpoints** view, **Check/uncheck** the breakpoint.
Remove all breakpoints	Right-click anywhere on the **Source** view and select **Breakpoints**. Select all **Breakpoints** (click on the first breakpoint, then hold **Shift** and click on the last breakpoint). Click **Remove** button.	Click **Remove All** on **Breakpoints** view.
Control execution with: ▪ Step into ▪ Step over ▪ Step return ▪ Continue ▪ Suspend	Click the appropriate button in **Debug** toolbar (to display **Debug** toolbar go to **View > Toolbar > Debug Toolbar**): ▪ Step into: **Step Into** ▪ Step over: **Step Over** ▪ Step return: **Step Out** ▪ Continue: **Continue** ▪ Suspend: **Interrupt Debugger**	Click the appropriate button in **Debug** view: ▪ Step into: **Step Into** ▪ Step over: **Step Over** ▪ Step return: **Step Return** ▪ Continue: **Resume** ▪ Suspend: **Suspend**

Table 3-5 **Parallels Between Tornado 2.2 and Wind River Workbench Debuggers** (cont'd)

Use Case	Tornado 2.2	Wind River Workbench
Display: ▪ Variables ▪ Watch ▪ Registers ▪ Back trace ▪ Memory	Click the appropriate button in **Debug** toolbar (to display **Debug** toolbar go to **View > Toolbar > Debug Toolbar**).	From menu select **Window > Show View**. Select the view of interest.
Set local/global variable values	Display **Variable**. Find the variable under the **Name** column. Double-click under the **Value** column. The cell becomes editable. Enter a new value, click **Enter**.	From the menu select **Window > Show View > Local Variables** or **Global Variables**. Find the variable under name column. Double-click on the **Value cell** next to the name. The cell becomes editable. Enter a new value, and click **Enter**.
Toggle between source and disassembly modes	From menu select **View > Disassembly**. To toggle back, select **View > Source**.	While debugging, select **Run > Disassembly** from **Main** menu.
Toggle between system and task mode debugging	Click the **Switch to system mode debugging** toolbar button.	In the **Debug** view, right-click any item, then select the **Debug Mode** submenu and select the desired mode. In the **Target Manager** view, click the **Toggle System Mode** toolbar icon. From the Run menu, select the **Debug Mode** submenu and select the desired mode.

Task Mode versus System Mode

When the target is in **Task Mode**, the debugger interacts only with the process or processes being debugged. If this process is suspended, other processes keep running. This mode is less intrusive, as it allows the user to control the selected process or thread while the rest of the system can continue to operate normally.

In **System Mode**, the debugger interacts with the entire system at once, so if one task is suspended, all processes and kernel tasks running on the system are suspended as well. This allows for increased control and visibility into what is happening on the system, but it is also very disruptive.

Switching Between Modes

When you switch to System Mode, by right-clicking a task in the **Debug** view and selecting **System Mode** or by clicking the **Toggle System Mode** icon on the **Target Manager** toolbar, breakpoints planted in Task Mode are saved to a buffer and then removed, and so are not active in System Mode.

When you switch back to Task Mode, any breakpoints planted while in System Mode (or breakpoints configured with the **System Mode** checkbox selected) are removed to a buffer, and Task Mode breakpoints are replaced.

Each debug connection works in either the Task Mode or System Mode, but not both. When you switch modes, the connections for the other mode become unavailable, and the only operation available in that mode is disconnecting the debugger.

Stepping In System Mode

You can step through the OS kernel and scheduler while in System Mode, and this may cause the current task to be replaced by another. This can sometimes lead to confusing results in other views such as the Memory view.

3.7 Run

Table 3-6 **Parallels Between Running Tornado 2.2 and Wind River Workbench**

Use Case	Tornado 2.2	Wind River Workbench
Start simulator	Click **Launch Simulator**. Follow instructions on-screen.	Right-click **Target Manager** and select **New > Connection**. Select **Wind River VxWorks Simulator Connection**. Follow instructions on screen.
Boot project	Enter parameters on target console. Project directory is *installDir*/**target/proj** (by default).	Same as Tornado 2.2, but project directory is *installDir*/*workspaceDirectory* (by default).
Reboot target	Select **Tools > Target Server > Manage**. From **Set Selection** drop-down list, select **Reboot**. Click **Apply**.	From **Target Manager** view, right-click the chosen **Target Server**. Select **Reset connected target**.
Launch host shell	Click **Launch Shell** button.	From **Target Manager**, right-click **Target Server**. Select **Target Tools > Host Shell**.
Download project	From **Workspace** right-click **Project**. Select **Download** *buildSpec*.	There are two ways to do this: From **Project Navigator** view, expand the project node. Right-click **Build spec folder**, select **Download/Load Symbols**. Check default selections, then click **OK**. From **Target Manager** view, expand the **Target server node**. Right-click **Tasks** and select **Download/Load Symbols**. Enter the path to the binaries in the **Download** box or browse to it. Click **OK**.
Display task list	Click **Launch Browser** button. Select **Tasks** from drop-down list.	From **Target Manager** view, expand **Target server node**, then expand the **Tasks** node.

Table 3-6 **Parallels Between Running Tornado 2.2 and Wind River Workbench** (cont'd)

Use Case	Tornado 2.2	Wind River Workbench
Start task	Click **Launch Debugger** button. Select **Debug > Run**. Make selections, click **OK**.	From **Target Manage**r view, expand target server node. Right-click on **Tasks** folder and select **Run Task**. Check selections and click **OK** when ready.

3.8 Task States

The VxWorks target server and the Eclipse **Debug** view both make a distinction between running, suspended, or stopped tasks, but their definitions are not identical. Table 3-7 compares the definitions.

Table 3-7 **Target Server and Debug View Task States**

VxWorks Target Server	Eclipse Debug View	Definition
Running	Running	Task is active, and has focus.
Suspended	Running	Task is waiting while another task runs.
Stopped	Suspended	Task stopped at a breakpoint or other event, or was stopped by user.

3.9 System Viewer

The product formerly known as WindView is now the Wind River System Viewer. For details about new features of System Viewer, see your product *Release Notes*.

Creation and use of System Viewer logs has been simplified. This means that creating, uploading, and deleting logs is easier and less error-prone. Starting, stopping, and controlling event collection is unchanged. For examples, see the **wvLib** reference entry.

Table 3-8 **Parallels Between Tornado 2.2 WindView and Wind River Workbench System Viewer**

Use Case	Tornado 2.2	Wind River Workbench
Start Wind River System Viewer	Click **Launch WindView Configuration** button.	From **Target Manager** view right-click **Target server**. Select **Target Tools > System Viewer Configuration**.

New VxWorks features have resulted in some new System Viewer events. The affected libraries and functional areas are shown in Table 3-9.

Table 3-9 **New System Viewer Events**

Library	Feature
rtpLib	instrumentation of RTP creation and deletion
system call interface	instrumentation of system calls, selectable per RTP, with decoding of parameters
edrLib	instrumentation of writes to the error log
isrLib	ISR object creation and deletion, also ISR handler invocation
sdLib	instrumentation of creation and deletion of shared data regions
USB stack	instrumentation of host and peripheral USB stack
IPv6 dual network stack	updated instrumentation of the network stack

For information about changes to System Viewer routines, see the *VxWorks Migration Guide*.

PART 2

Migrating WTX Applications

4

Migrating WTX Applications

4.1 Introduction

This chapter provides general guidance on WTX with respect to Wind River Workbench 2.x. This information will be useful to tool developers who will use the WTX Tcl and WTX C information in the following chapters to migrate WTX applications to Workbench. Most VxWorks application developers will not need the information contained in these chapters.

WTX, the Wind River Tool eXchange, is the protocol that was used in Tornado 1.x and Tornado 2.x to communicate between host applications and the target server. In addition, it could be used to connect a third-party application to the Tornado GUI.

Workbench is backward compatible with Tornado 2.2. By migrating your WTX applications as shown in Part 2, your applications will be compatible with Workbench.

4.2 WTX Compatibility

WTX 2 was the version of WTX shipped with Tornado 2.2. WTX 4 is the version associated with Wind River Workbench 2.5. For detailed information on changes in the Tcl and C APIs, see *5. WTX Tcl API Migration* and *6. WTX C API Migration*

Applications based on WTX 2 will require recompiling, which will also provide an opportunity to migrate to WTX 4.

In most areas WTX 4 is highly compatible with WTX 2; important compatibilities and incompatibilities are shown below.

Target Server Level

Tools built for Tornado 2.2 *are not compatible* with a Workbench 2.5 target server. The tools should be migrated to the new WTX APIs.

Tcl Layer

A Tcl script built using **wtxtcl** for WTX 2.2 *will still work* with VxWorks 6.3 **wtxtcl**.

Gopher Language

Gopher strings used for VxWorks 5.5 *will not work* with VxWorks 6.3 because some internal structures have changed.

4.3 Changes in Events

In order to provide the information required to debug VxWorks 6.3 applications, several WTX events have changed. Table 4-1 shows the three affected events and compares their descriptors (previously called their parameters). It also shows when each event is sent.

Table 4-1 **Event Changes Between WTX 2 and WTX 4**

WTX 2 Event and Parameters	WTX 4 Event and Descriptors	Sent When:
TEXT_ACCESS	**WTX_TEXT_ACCESS**	Breakpoint hit.
	statusOfStoppedContext	
	StoppedCtxType	
	StoppedID	
	StoppedSubId	
CtxType	HitCtxType	
ctxId	HitCtxId	
	HitCtxSubId	
pc	Pc	
fp	Fp	
sp	Sp	
DATA_ACCESS	**WTX_DATA_ACCESS**	Watchpoint hit.
	statusOfStoppedContext	
	StoppedCtxType	
	StoppedID	
	StoppedSubId	
CtxtType	HitCtxType	
CtxtId	HitCtxId	
	HitCtxSubId	
pc	Pc	
fp	Fp	
sp	Sp	
addr	Addr	

Table 4-1 **Event Changes Between WTX 2 and WTX 4** (cont'd)

WTX 2 Event and Parameters	WTX 4 Event and Descriptors	Sent When:
EXCEPTION	**WTX_EXCEPTION**	Hardware exception raised.
	statusOfStoppedContext	
	StoppedCtxType	
	StoppedID	
	StoppedSubID	
CtxType	ExcCtxType	
CtxId	ExcCtxId	
	ExcCtxSubId	
excVector	vec	
pEsf	pEsf	
	sigNum	

4.3.1 WTX_TEXT_ACCESS and WTX_DATA_ACCESS

The following tables show what values various descriptors can take under various circumstances for **WTX_TEXT_ACCESS** and **WTX_DATA_ACCESS**. Table 4-2 shows what status values can be returned.

Table 4-2 **Status Descriptor Values**

Event descriptor	Value	Comment
statusOfStoppedContext	**RUNNING**	Can happen if **ACTION** was **NOTIFY** rather than **STOP**.
statusOfStoppedContext	**STOPPED**	Typical when **ACTION** is **STOP**.

Table 4-3 shows the possible values of the group of descriptors consisting of StoppedCtxType, StoppedCtxId, and StoppedCtxSubId.

Table 4-3 **Stopped Context Descriptor Values**

Event descriptor	Value	Comment
StoppedCtxType	PROCESS	The RTP RtpId is stopped; that means all tasks within the RtpId are stopped.
StoppedCtxId	rtpId	
StoppedCtxSubId	0	
StoppedCtxType	TASK	The taskId belonging to rtpId is stopped.
StoppedCtxId	taskId	
StoppedCtxSubId	Rtp Id of task	
StoppedCtxType	SYSTEM	The whole system is stopped.
StoppedCtxId	0	If HitCtxType == TASK, it means that the breakpoint ACTION was ACTION_SYS_STOP.
StoppedCtxSubId	0	

Table 4-4 shows the possible values of the group of descriptors consisting of HitCtxType, HitCtxId, and HitCtxSubId.

NOTE: The distinction between the *stopped* context and the *hit* context is new for Workbench. The hit context is what has hit the breakpoint. The stopped context is what the breakpoint has stopped, which may or may not be the same.

Table 4-4 **Hit Context Descriptor Values**

Event descriptor	Value	Comment
HitCtxType	TASK	The taskId belonging to rtpId has hit the breakpoint.
HitCtxId	taskId	
HitCtxSubId	rtpId of task	
HitCtxType	SYSTEM	The target was in system mode before hitting the breakpoint.
HitCtxId	0	
HitCtxSubId	0	

4.3.2 WTX_EXCEPTION

The following tables show what values various descriptors can take under various circumstances for **WTX_EXCEPTION**. Table 4-5 shows what status values can be returned.

Table 4-5 **Status Descriptor Values**

Event descriptor	Value	Comment
statusOfStoppedContext	**RUNNING**	Can happen if **ACTION** was **NOTIFY** rather than **STOP**.
statusOfStoppedContext	**STOPPED**	Typical when **ACTION** is **STOP**.

Table 4-6 shows the possible values of the group of descriptors consisting of StoppedCtxType, StoppedCtxId, and StoppedCtxSubId.

Table 4-6 **Stopped Context Descriptor Values**

Event descriptor	Value	Comment
StoppedCtxType	**PROCESS**	The RTP RtpId is stopped; that means all tasks within the RtpId are stopped.
StoppedCtxId	rtpId	
StoppedCtxSubId	0	
StoppedCtxType	TASK	The taskId belonging to rtpId is stopped.
StoppedCtxId	taskId	
StoppedCtxSubId	Rtp Id of task	
StoppedCtxType	**SYSTEM**	The whole system is stopped.
StoppedCtxId	0	If HitCtxType == **TASK**, it means the breakpoint **ACTION** was **ACTION_SYS_STOP**.
StoppedCtxSubId	0	

Table 4-7 shows the possible values of the group of descriptors consisting of HitCtxType, HitCtxId, and HitCtxSubId.

Table 4-7 **Hit Context Descriptor Values**

Event descriptor	Value	Comment
HitCtxType	**TASK**	The taskId belonging to rtpId has hit the breakpoint.
HitCtxId	taskId	
HitCtxSubId	Rtp Id of task	
HitCtxType	**SYSTEM**	The target was in system mode before hitting the breakpoint.
HitCtxId	0	
HitCtxSubId	0	

4.4 Event Summary

WTX events have changed from both Tornado 2.2 and Tornado 3.x.

Table 4-8 **Comparison of Events for WTX 2/3/4**

Tor 2.x (WTX 2)	Tor 3.x (WTX 3)	WB2.5 (WTX 4)
CTX_START	**CTX_START**	**CTX_START**
contextType	createdCtxType	contextType
contextId	createdCtxtId	contextId
		contextSubid
	creationCtxtType	contextType
	creationCtxtId	contextId
		contextSubid

Table 4-8 **Comparison of Events for WTX 2/3/4** (cont'd)

Tor 2.x (WTX 2)	Tor 3.x (WTX 3)	WB2.5 (WTX 4)
CTX_UPDATE	**CTX_UPDATE**	**CTX_UPDATE**
	ctxtType	contextType
	ctxtId	contextId
	contextSubid	
CTX_EXIT	**CTX_EXIT**	**CTX_EXIT**
contextType	ctxtType	contextType
contextId	ctxtId	contextId
		contextSubid
returnVal	returnVal	returnVal
errnoVal	errnoVal	errnoVal
TEXT_ACCESS	**TEXT_ACCESS**	**WTX_TEXT_ACCESS, EVENT_TEXT_ACCESS, DATA_TEXT_ACCESS**
		statusOfStoppedContext
		StoppedCtxType
		StoppedID
		StoppedSubId
contextType	ctxtType	HitCtxType
contextId	ctxtId	HitCtxId
		HitCtxSubId
pc	pc	Pc
fp	fp	Fp
sp	sp	Sp

Table 4-8 **Comparison of Events for WTX 2/3/4** (cont'd)

Tor 2.x (WTX 2)	Tor 3.x (WTX 3)	WB2.5 (WTX 4)
DATA_ACCESS	**DATA_ACCESS**	**EVENT_DATA_ACCESS, WTX_DATA_ACCESS**
		statusOfStoppedContext
		StoppedCtxType
		StoppedID
		StoppedSubId
contextType	ctxtType	HitCtxType
contextId	ctxtId	HitCtxId
		HitCtxSubId
pc	pc	Pc
fp	fp	Fp
sp	sp	Sp
addr	addr	Addr
EXCEPTION	**EXCEPTION**	**WTX_EXCEPTION, EVT_EXC**
		statusOfStoppedContext
		StoppedCtxType
		StoppedID
		StoppedSubID
contextType	ctxtType	ExcCtxType
contextId	ctxtId	ExcCtxId
		ExcCtxSubId
excVector	excVector	vec
esfAddr	esfAddr	

Table 4-8 **Comparison of Events for WTX 2/3/4** (cont'd)

Tor 2.x (WTX 2)	Tor 3.x (WTX 3)	WB2.5 (WTX 4)
		pEsf
		sigNum
VIO_WRITE	**VIO_WRITE**	**EVENT_VIO_WRITE**
channelId	channelId	channelId
addlData	addlData	
addlDataLen	addlDataLen	datalen
		datas
EVTPT_ADDED	**EVTPT_ADDED**	**EVTPT_ADDED**
evtptNum	evtptNum	evtptNumCurrent
	ctxtType	contextType
	ctxtId	contextId
		contextSubid
	actionType	actionType
	actionArg	actionArg
	callRtn	callRtn
	callArg	callArg
	eventType	eventType
	numArgs	numArgs
	args ...	args ...
EVTPT_DELETED	**EVTPT_DELETED**	**WTX_EVENT_EVTPT_DELETED**
evtptNum	evtptNum	evtptNumCurrent
CALL_RETURN	**CALL_RETURN**	**CALL_RETURN**
callTaskId	callTaskId	callId

Table 4-8 **Comparison of Events for WTX 2/3/4** (cont'd)

Tor 2.x (WTX 2)	Tor 3.x (WTX 3)	WB2.5 (WTX 4)
returnVal	returnVal	returnVal
errnoVal	errnoVal	error
	toolId	
TGT_RESET	**TGT_RESET**	**TGT_RESET**
X	X	X
TGT_LOST	**TGT_LOST**	**TGT_LOST**
	X	X
TGT_RECOVERED	**TGT_RECOVERED**	**TGT_RECOVERED**
	X	X
TS_POSTMORTEM	**TS_POSTMORTEM**	**TS_POSTMORTEM**
	X	X
TS_KILLED	**TS_KILLED**	**TS_KILLED**
X	X	X
OBJ_LOADED	**OBJ_LOADED**	**EVT_OBJ_LOADED**
objModId	objModId	moduleId
objModName	objModNameWithPath	name
OBJ_UNLOADED	**OBJ_UNLOADED**	**EVT_OBJ_UNLOADED**
objModId	objModId	moduleId
objModName	objModNameWithPath	name
SYM_ADDED	**SYM_ADDED**	**EVT_SYM_ADD**
symbol	symName	symbolName
symVal	symVal	symbolValue

Table 4-8 **Comparison of Events for WTX 2/3/4** (cont'd)

Tor 2.x (WTX 2)	Tor 3.x (WTX 3)	WB2.5 (WTX 4)
SYM_REMOVED	**SYM_REMOVED**	**EVT_SYM_REMOVE**
symbol	symName	symbolName
symType	symType	symbolType
TOOL_ATTACH	**TOOL_ATTACH**	**TOOL_ATTACH**
toolName	toolName	toolName
TOOL_DETACH	**TOOL_DETACH**	**TOOL_DETACH**
toolName	toolName	toolName
TOOL_MSG	**TOOL_MSG**	**TOOL_MSG**
destination	destination	
sender	sender	
message	message	
anything	anything	
USER	**USER**	**USER**
message	message	Message
TRIGGER	**TRIGGER**	**EVT_TRIGGER**
	ctxtType	ctxType
	ctxtId	contextId
	eventId	eventId
	objId	objId
	taskStopped	
	args ...	
	WTXEVENT_PD_INITIALIZED	
	X	

Table 4-8 **Comparison of Events for WTX 2/3/4** (cont'd)

Tor 2.x (WTX 2)	Tor 3.x (WTX 3)	WB2.5 (WTX 4)
	OTHER	
	X	
UNKNOWN	**UNKNOWN**	**UNKNOWN**
X	X	X

5 WTX Tcl API Migration

5.1 Introduction

This chapter provides details on changes in the WTX Tcl API of Tornado 2.2 to assist tool developers in migrating their VxWorks-supporting tools to Wind River Workbench 2.5 (VxWorks 6.1 /WTX Tcl 4.0).

5.2 Tcl 8.0 to Tcl 8.4

For Workbench 2.5 the underlying version of Tcl is Tcl 8.4. There are only two significant changes between Tcl 8.0 (the version shipped with Tornado 2.2) and Tcl 8.4:

- The **[array startsearch ...]** syntax has one bug. This problem is limited to access to the **env** (environment variable) array. It works for any other array.
- Integer values are 64 bits in Tcl 8.4 instead of 32 bits in Tcl 8.0. This means that when a variable is set to 0xf0000000 (for example), it used to be considered a negative value with Tcl 8.0 but it is now considered as a positive value. Because of this, some computations can return a different result. Note that this case is rare.

It is currently more efficient and easier to use either the **array get** or **array names**, together with **foreach**, to iterate over all but very large arrays. For example:

```
if {[lsearch -exact [array names env] $readName] != -1}
    {
    # name exists in array
    return [lindex [array get env $readName] 1]
    }
```

5.3 Backward Compatibility Mode

The Tcl layer of WTX provides a way to switch from the new set of APIs to the old ones. This is done by calling **wtxV2CompatSet( )** and **wtxV2CompatUnset( )**. This allows to you to reuse old WTX scripts without needing to rewrite the API calls. This compatibility mode also handles old-style WTX events.

When starting the **wtxtcl** shell, compatibility mode is automatically activated. You can change this behavior by modifying the *wtxVersion* variable in the **wtxinit.tcl** file located in the *installDir*/**vxworks-6.***x*/**host/resource/tcl** directory. Set this variable to 4 to start the **wtxtcl** shell in WTX 4.0 mode. You can dynamically change this compatibility mode by calling the compatibility APIs.

While compatibility mode eases your transition, you will want to migrate to WTX 4 in order to take advantage of new features, including manipulating RTP memory.

Example 5-1 **WTX Compatibility Modes**

The default is WTX 2:

```
wtxtcl> wtxToolAttach TGTSVR
TGTSVR@cocyte
wtxtcl> wtxTsInfoGet
{WDB Agent across named pipe} {255 0} {61 0 0 8192 1234 0 0 0}  {0 {6.0 -
Technology Access Release 2}} {SunOS 5.8 [sun4u]} cocyte:/default/vxWorks
{0x60000000 16777216} {0x601b7f6c 935945} philb 1084957757 1085383651
available 0 4.0 4.0
{0x132308 windsh {} {} philb@cocyte}
{0x10af18 wtxtcl {} {} philb@raptor}
```

Turning off compatibility yields WTX 4:

```
wtxtcl> wtxV2CompatUnset
2
wtxtcl> wtxTsInfoGet
{WDB Agent across named pipe} {255 0} {61 16777216 0 8192 1234 {}}
{VxWorks {6.0 - Technology Access Release 2} diab {}} {SunOS 5.8 [sun4u]}
cocyte: /default/vxWorks {0x60000000 16777216} {0x601b7f6c 935945} philb
{Wed May 19 11:09:17 2004} {Mon May 24 09:27:31 2004} available 0 4.0 4.0
1 {{tgtsvr.ex} {-n} {TUTU} {-B} {wdbpipe} {-Wd} {/tmp/wtx} {-Bd}
{/tmp/wdb} {-V} {vxsim}}
{0x132308 windsh {} {} philb@cocyte}
{0x10af18 wtxtcl {} {} philb@raptor}
wtxtcl>
```

Resetting compatibility returns to WTX 2:

```
wtxtcl> wtxV2CompatSet
2
wtxtcl> wtxTsInfoGet
{WDB Agent across named pipe} {255 0} {61 0 0 8192 1234 0 0 0}  {0 {6.0 -
Technology Access Release 2}} {SunOS 5.8 [sun4u]} cocyte:/default/vxWorks
{0x60000000 16777216} {0x601b7f6c 935945} philb 1084957757 1085383651
available 0 4.0 4.0
{0x132308 windsh {} {} philb@cocyte}
{0x10af18 wtxtcl {} {} philb@raptor}
```

VxWorks 5.5 tools can be used with VxWorks 6.1 kernel-only images. However, this is not expected to be the case with the next VxWorks release.

5.4 Modified Routines

Two major changes appear in VxWorks 6.3 (WTX 4):

1. Because of the new process descriptors required to support real-time processes in VxWorks 6.3, some WTX Tcl calls have changed to support them.
2. The returns of some functions were modified in order to be more accurate or more explicit.

In addition, some functions were renamed and others were removed.

Table 5-1 lists functions that were changed for VxWorks 6.3; for details, see *5.4.1 Memory Related Routines*, p.50, *5.4.2 Module-related Routines*, p.52, *5.4.3 Symbol-related Routines*, p.55, and *5.4.4 Other functions*, p.56.

Table 5-1 **Functions Modified Between Tornado 2.2 and Wind River Workbench**

Changed Functions	Function Description
Memory-Related Functions	
Table 5-1 lists functions that were changed for VxWorks 6.3; for details, see 5.4.1 Memory Related Routines, p.50, 5.4.2 Module-related Routines, p.52, 5.4.3 Symbol-related Routines, p.55, and 5.4.4 Other functions, p.56., p.48	query the target server disassembled instructions
wtxMemMove, p.50	move a block of memory on the target
wtxMemRead, p.50	read target memory into a memory block
wtxMemScan, p.50	scan target memory for the presence or absence of a pattern
wtxMemSet, p.51	set a block of memory to a specified value
wtxMemWidthRead, p.51	read a memory block of specified width from the target
wtxMemWidthWrite, p.51	write a memory block of specified width to the target
wtxMemWrite, p.51	write a memory block to the target

Table 5-1 **Functions Modified Between Tornado 2.2 and Wind River Workbench** (cont'd)

Changed Functions	Function Description
Module-Related Functions	
wtxObjModuleChecksum, p.52	check the validity of an object module
wtxObjModuleInfoGet, p.52	return information about an object module
wtxObjModuleLoad, p.53	load a multiple section object file
wtxObjModuleLoadProgressReport, p.53	get the asynchronous load status
wtxObjModuleLoadStart, p.54	load a multiple section object file asynchronously
wtxObjModuleListGet, p.54 (old **wtxObjModuleList**)	get a list of IDs of object modules loaded on the target
Symbol-Related Functions	
wtxSymListGet, p.55	query the target server symbol table
Other Functions	
wtxContextCreate, p.56	create a new context on the target
wtxGopherEval, p.56	request the evaluation of a Gopher script by the agent
wtxRegsGet, p.56	read a block of register data from the target
wtxRegsSet, p.57	write a block of register data to the target
wtxTsInfoGet, p.57	get information about the target server
wtxVioFileListGet, p.57 (old **wtxVioFileList**)	list the files managed by the target server

5.4.1 Memory Related Routines

wtxMemDisassemble

New prototype

wtxMemDisassemble [**-pd** *pdId* | **-pdkernel**] [**-address**] [**-opcodes**] [**-hex**] *startAddr* [**nInst** [*endAddr*]]

Old prototype

wtxMemDisassemble [**-address**] [**-opcodes**] [**-hex**] *startAddr* [**nInst** [*endAddr*]]

wtxMemMove

New prototype

wtxMemMove [**-pd** *pdId* | **-pdkernel**] *source destination nBytes*

Old prototype

wtxMemMove *source destination nBytes*

wtxMemRead

New prototype

wtxMemRead [**-pd** *pdId* | **-pdkernel**] [**-cachebypass**] *address nBytes*

Old prototype

wtxMemRead *address nBytes*

wtxMemScan

New prototype

wtxMemScan [**-pd** *pdId* | **-pdkernel**] [**-notmatch**] *startAddr endAddr* {**-string** *string* | **-memblk** *mblk*}

Old prototype

wtxMemScan [**-notmatch**] *startAddr endAddr* {**-string** *string* | **-memblk** *mblk*}

wtxMemSet

New prototype

wtxMemSet [**-pd** *pdId* | **-pdkernel**] *startAddr nBytes value*

Old prototype

wtxMemSet *startAddr nBytes value*

wtxMemWidthRead

New prototype

wtxMemWidthRead [**-pd** *pdId* | **-pdkernel**] [**-cachebypass**] *address nBytes* [*width*]

Old prototype

wtxMemWidthRead *address nBytes* [*width*]

wtxMemWidthWrite

New prototype

wtxMemWidthWrite [**-pd** *pdId* | **-pdkernel**] *blockId address* [*width*] [*offset*] [*nBytes*]

Old prototype

wtxMemWidthWrite *blockId address* [*width*] [*offset*] [*nBytes*]

wtxMemWrite

New prototype

wtxMemWrite [**-pd** *pdId* | **-pdkernel**] *blockId address* [*offset*] [*nBytes*]

Old prototype

wtxMemWrite *blockId address* [*offset*] [*nBytes*]

5.4.2 Module-related Routines

wtxObjModuleChecksum

New prototype

wtxObjModuleChecksum {*nameOrId* | **-allmodules}**

Old prototype

wtxObjModuleChecksum *nameOrId*

wtxObjModuleInfoGet

New prototype

wtxObjModuleInfoGet *objModuleId*

Old prototype

wtxObjModuleInfoGet *objModuleId*

Function return changed

Before:

moduleId moduleName moduleFormat moduleGroup loadFlags { *textSectionFlags textAddr textSegmentLeng* } { *dataSectionFlags dataAddr dataSegmentLength* } { *bssSectionFlags bssAddr bssSegmentLength* }

Example:

0x63158 vxColor.o elf 0x4 0x4 {0 0xd6218 0x18f9} {0 0xffffffff 0} {0 0xffffffff 0}

Now:

moduleId moduleNameFullPath pdId formatCode loadFlag { {*sectionName sectionType loadFlagOfSection sectionBaseAddr sectionPartitionName sectionLength*} [{*sectionName sectionType loadFlagOfSection sectionBaseAddr sectionPartitionName sectionLength* }…] }

Example:
0x1386e8 /tmp/test/vxColor.o 0xf0934 0x6 LOAD_GLOBAL_SYMBOL | LOAD_NO_AUTORUN { {.text SECTION_TEXT 0x1 0x211420 {} 0x1714} {.rodata SECTION_RODATA 0x1 0x213010 {} 0x14} }

wtxObjModuleLoad

New prototype

wtxObjModuleLoad [**-ts**] [*flags*] [**-d** *directivesList*] *filename*

Old prototype

wtxObjModuleLoad [**-ts**] [*flags*] [*textAddr*] [*dataAddr*] [*bssAddr*] *filename*

Function return changed

Before:
moduleId textAddr dataAddr bssAddr

Example:
0x63158 0xd6218 oxffffffff 0xffffffff

Now:
moduleId {{*sectionName sectionBaseAddr*} [{*sectionName sectionBaseAddr*} …]}

Example:
0x1386e8 { {.text 0x211420} {.rodata 0x123010} }

This function is not backward-compatible if your call uses *textAddr*, *dataAddr*, or *bssAddr*. You will need to modify it using the **–d** *directivesList* argument.

This example illustrates the *directives_list* parameter

Before:
wtxObjModuleLoad 0xffffffff $dataAddress 0xffffffff $fileName

Now:
wtxObjModuleLoad –d {.data $dataAddress SECTION_DATA} $fileName

wtxObjModuleLoadProgressReport

New prototype

wtxObjModuleLoadProgressReport

Old prototype

wtxObjModuleLoadProgressReport

Function return changed

Before:
moduleId textAddr dataAddr bssAddr

Example:
0x63158 0xd6218 oxffffffff 0xffffffff

Now:
moduleId {{*sectionName sectionBaseAddr*} [{*sectionName sectionBaseAddr*} …]}

Example:
0x1386e8 { {.text 0x211420} {.rodata 0x123010} }

wtxObjModuleLoadStart

New prototype

wtxObjModuleLoadStart [**-ts**] [*flags*] [**-d** *directives_list*] *filename*

Old prototype

wtxObjModuleLoadStart [**-ts**] [*flags*] [*textAddr*] [*dataAddr*] [*bssAddr*] *filename*

This function is not backward-compatible if your call uses *textAddr, dataAddr,* or *bssAddr*. You will need to modify it using the **–d** *directivesList* argument.

This example illustrates the *directives_list* parameter

Before:
wtxObjModuleLoadStart 0xffffffff $dataAddress 0xffffffff $fileName

Now:
wtxObjModuleLoadStart –d {.data $dataAddress SECTION_DATA}
$fileName

wtxObjModuleListGet

New prototype

wtxObjModuleListGet [*name*]

Old prototype

wtxObjModuleList

The function name and prototype changed; for more information, see the reference entry.

5.4.3 Symbol-related Routines

wtxSymListGet

New prototype

wtxSymListGet [**-module** *nameOrId* | **-moduleId** *modId* | **-moduleName** *modName* [**-unknown**]] [**-n** *nSymbols*] {**-value** *value* | **–name** *name*} [**-ref** *symbol_reference*] [**-type** *symbol_type*]

Old prototype

wtxSymListGet [**-module** *nameOrId* | **-moduleId** *modId* | **-moduleName** *modName* [**-unknown**]] [**-n** *nSymbols*] {**-name** *name* | **-value** *value*}

Function return changed:

Before:
symName symAddr symTypeInteger symTblId group moduleName

Example:
taskDelay 0x5c840 0x5 0 1 VxWorks

Now:
symName symAddr symTypeAscii pdId reference moduleName

Example:
taskDelay 0x4399c SYMBOL_GLOBAL | SYMBOL_TEXT | SYMBOL_ENTRY 0xf0934 0 ""

5.4.4 Other functions

wtxContextCreate

New prototype

wtxContextCreate *CONTEXT_TYPE name priority options stackBase stackSize entry redirIn redirOut redirErr* [*a0*...*a9*]

Old prototype

wtxContextCreate *CONTEXT_TYPE name priority options stackBase stackSize entry redirIn redirOut* [*a0*...*a9*]

This function is not backward-compatible because the *redirErr* argument has been added.

wtxGopherEval

New prototype

wtxGopherEval [**-pd** *pdId* | **-pdkernel**] *script*

Old prototype

wtxGopherEval *script*

The script format has new commands. For more information, see the **wtxGopherEval** reference entry.

wtxRegsGet

New prototype

wtxRegsGet *CONTEXT_TYPE contextId regSetType offset nBytes blockId*

Old prototype

wtxRegsGet *CONTEXT_TYPE contextId REG_SET_TYPE offset nBytes blockId*

The *regSetType* parameter is now an **int**.

wtxRegsSet

New prototype

wtxRegsSet *CONTEXT_TYPE contextId regSetType offset nBytes blockId*

Old prototype

wtxRegsSet *CONTEXT_TYPE contextId REG_SET_TYPE offset nBytes blockId*

The *regSetType* parameter is now an **int**.

wtxTsInfoGet

New prototype

wtxTsInfoGet

Old prototype

wtxTsInfoGet

Function return changed

Old profile:

{WDB Agent across named pipe} {255 0} {61 0 0 8192 1234 0 0 0} {0 {6.0 - Technology Access Release 2}} {SunOS 5.8 [sun4u] *myHost*:/default/vxWorks {0x60000000 16777216} {0x601b7f6c 935945} *name* 1084957757 1085383651 available 0 4.0 4.0 {0x10af18 wtxtcl {} {} *name@myHost*}

New profile (target server arguments added):

{WDB Agent across named pipe} {255 0} {61 16777216 0 8192 1234 {}} {VxWorks {6.0 - Technology Access Release 2} diab {}} {SunOS 5.8 [sun4u]} *myHost*: /default/vxWorks {0x60000000 16777216} {0x601b7f6c 935945} *name* {Wed May 19 11:09:17 2004} {Mon May 24 09:27:31 2004} available 0 4.0 4.0 1 {{tgtsvr.ex} {-n} {TUTU} {-B} {wdbpipe} {-Wd} {/tmp/wtx} {-Bd} {/tmp/wdb} {-V} {vxsim}} {0x10af18 wtxtcl {} {} *name@myHost*}

wtxVioFileListGet

New prototype

wtxVioFileListGet

Old prototype

wtxVioFileList

5.5 Obsoleted Routines

wtxObjModuleInfoAndPathGet
This routine was used because **wtxObjModuleInfoGet** returned information on a module that included only the module name. **wtxObjModuleInfoAndPathGet** was introduced in order to have information on the module with its full load path. **wtxObjModuleInfoGet** now returns information including the full path of the module.

wtxConsoleCreate
This routine has been replaced by the **wtxConsole** tool, which is no longer linked to the target server.

wtxConsoleKill
This routine has been replaced by the **wtxConsole** tool, which is no longer linked to the target server.

wtxEventpointList
Use **wtxEventpointListGet** instead.

wtxServiceAdd
Obsolete. This functionality is no longer available.

5.6 New Routines

For more information on these functions, see the appropriate reference entries.

Table 5-2 **New Routines Provided in Workbench 2.5**

New Function	Description
wtxContextStop	stop a context
wtxProcessCreate	create a real-time process on the target
wtxProcessDelete	delete a real-time process on the target
wtxRegistryEntryAdd	add an entry to the Workbench registry
wtxRegistryEntryGet	get an entry description from the Workbench registry
wtxRegistryEntryListGet	return a list of existing registry entries
wtxRegistryEntryRemove	remove an entry from the Workbench registry
wtxRegistryEntryUpdate	add an entry to the Workbench registry
wtxRegistryTimeout	set or get the Workbench registry connection timeout
wtxV2CompatSet	set WTX version 2 compatibility mode
wtxV2CompatUnset	unset WTX version 2 compatibility mode

6 WTX C API Migration

6.1 Introduction

This chapter describes the changes to the WTX C API for Workbench 2.5. Many routines only need a change in the way the context parameter is passed. However, others have additional changes due to changes in the VxWorks operating system.

Workbench 2.x supports real-time processes (RTPs). The loader routines support the ability to load a task into an RTP. However, it is no longer possible to load an individual task into the kernel; only an object module can be loaded into the kernel.

6.2 Modified Routines

Table 6-1 lists all the WTX C routines that have been modified for VxWorks 6.3. For a complete discussion of the changes, see below.

Table 6-1 **Modified Routines for Wind River Workbench 2.5**

Modified Routine	Summary Description
wtxContextCont( ), p.64	continue a target context
wtxContextExitNotifyAdd ( ), p.65	add a context exit notification eventpoint
wtxContextKill( ), p.65	kill a target context
wtxContextResume( ), p.66	resume a target context
wtxContextStatusGet( ), p.66	get the status of a context
wtxContextStep( ), p.67	step a target context
wtxDirectCall( ), p.67	call a function on the target within the agent
wtxEventpointAdd( ), p.68	create a new WTX eventpoint
wtxEventpointListGet( ), p.69	list eventpoints on the target server
wtxFuncCall( ), p.70	call a function on the target
wtxGopherEval( ), p.70	evaluate a Gopher string on the target
wtxHwBreakpointAdd( ), p.71	create an eventpoint or hardware breakpoint
wtxMemAddToPool( ), p.72	add memory to the agent pool
wtxMemAlign( ), p.72	allocate aligned target memory
wtxMemAlloc( ), p.73	allocate a block of memory to the target server
wtxMemChecksum( ), p.73	perform a checksum on target memory
wtxMemDisassemble( ), p.74	get disassembled instructions matching the given address
wtxMemFree( ), p.74	free a block of target memory

Table 6-1 **Modified Routines for Wind River Workbench 2.5** (cont'd)

Modified Routine	Summary Description
wtxMemInfoGet( ), p.75	get information about the target
wtxMemMove( ), p.75	move a block of target memory
wtxMemRead( ), p.76	read memory from the target
wtxMemRealloc( ), p.76	reallocate a block of target memory
wtxMemScan( ), p.77	scan target memory for the presence or absence of a pattern
wtxMemSet( ), p.78	set target memory to a given value
wtxMemWidthRead( ), p.78	read a specified amount of memory from the target
wtxMemWidthWrite( ), p.79	write to a specified amount memory on the target
wtxMemWrite( ), p.80	write memory on the target
wtxObjModuleByNameUnload( ), p.80	unload an object module from the target
wtxObjModuleChecksum( ), p.81	check the validity of target memory
wtxObjModuleFindId( ), p.82	find the ID of an object module given its name
wtxObjModuleFindName( ), p.82	find an object module name given its ID
wtxObjModuleInfoAndPathGet( ), p.83	get path and other information on a module given its ID
wtxObjModuleInfoGet( ), p.83	get information on a module given its ID
wtxObjModuleLoad( ), p.84	load a multiple-section object file
wtxObjModuleLoadStart( ), p.85	load a multiple-section object file asynchronously
wtxObjModuleUnload( ), p.86	unload an object module from the target
wtxRegsGet( ), p.86	read register data from the target

Table 6-1 **Modified Routines for Wind River Workbench 2.5** (cont'd)

Modified Routine	Summary Description
wtxRegsSet( ), p.87	set register data for the target
wtxSymAdd( ), p.88	add a symbol with the given name, value, and type
wtxSymFind( ), p.88	find information on a symbol given its name or value
wtxSymListByModuleIdGet( ), p.90	get a list of symbols given a module ID
wtxSymListByModuleNameGet( ), p.90	get a list of symbols given a module name
wtxSymListGet( ), p.91	get a list of symbols matching the given search criteria
wtxSymRemove( ), p.93	remove a symbol from the default target server symbol table
wtxSymTblInfoGet( ), p.93	return information about the target server symbol table

wtxContextCont( )

Continue execution of a target context.

New Prototype

```
STATUS wtxContextCont
    (
    HWTX              hWtx,          /* WTX API handle          */
    WTX_CONTEXT *      pContext      /* WTX Context             */
    )
```

Old Prototype

```
STATUS wtxContextCont
    (
    HWTX               hWtx,          /* WTX API handle          */
    WTX_CONTEXT_TYPE   contextType,   /* type of context to cont */
    WTX_CONTEXT_ID_T   contextId      /* id of context to cont   */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass it.

wtxContextExitNotifyAdd ( )

Add a context exit notification eventpoint.

New Prototype

```
WTX_TGT_ID_T wtxContextExitNotifyAdd
    (
    HWTX           hWtx,          /* WTX API handle         */
    WTX_CONTEXT *  pContext       /* WTX Context            */
    )
```

Old Prototype

```
UINT32 wtxContextExitNotifyAdd
    (
    HWTX              hWtx,        /* WTX API handle        */
    WTX_CONTEXT_TYPE  contextType, /* type of context       */
    WTX_CONTEXT_ID_T  contextId    /* associated context    */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass it.

wtxContextKill( )

Kill a target context.

New Prototype

```
STATUS wtxContextKill
    (
    HWTX            hWtx,           /* WTX API handle           */
    WTX_CONTEXT *   pCtx,           /* WTX Context              */
    UINT32          options         /* options for kill         */
    )
```

Old Prototype

```
STATUS wtxContextKill
    (
    HWTX              hWtx,           /* WTX API handle */
    WTX_CONTEXT_TYPE  contextType,    /* type of context to kill */
    WTX_CONTEXT_ID_T  contextId       /* id of context to kill   */
    )
```

The *options* parameter adds new possibilities to the context kill facilities, for other operating systems. For VxWorks, it should be set to 0.

wtxContextResume( )

Resume execution of a target context.

New Prototype

```
STATUS wtxContextResume
    (
    HWTX              hWtx,         /* WTX API handle           */
    WTX_CONTEXT *     pContext      /* WTX Context              */
    )
```

Old Prototype

```
STATUS wtxContextResume
    (
    HWTX              hWtx,         /* WTX API handle           */
    WTX_CONTEXT_TYPE  contextType,  /* type of context to resume */
    WTX_CONTEXT_ID_T  contextId     /* id of context to resume   */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass it.

wtxContextStatusGet( )

Get the status of a context on the target.

New Prototype

```
WTX_CONTEXT_STATUS wtxContextStatusGet
    (
    HWTX              hWtx,          /* WTX API handle          */
    WTX_CONTEXT *     pContext       /* WTX Content             */
    )
```

Old Prototype

```
WTX_CONTEXT_STATUS wtxContextStatusGet
    (
    HWTX              hWtx,          /* WTX API handle          */
    WTX_CONTEXT_TYPE  contextType,   /* type of context         */
    WTX_CONTEXT_ID_T  contextId      /* id of context           */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass it.

wtxContextStep( )

Single step execution of a target context.

New Prototype

```
STATUS wtxContextStep
    (
    HWTX              hWtx,         /* WTX API handle            */
    WTX_CONTEXT *     pContext,     /* WTX Context               */
    WTX_TGT_ADDR_T    stepStart,    /* step Start PD value       */
    WTX_TGT_ADDR_T    stepEnd       /* step End PC value         */
    )
```

Old Prototype

```
STATUS wtxContextStep
    (
    HWTX                hWtx,        /* WTX API handle            */
    WTX_CONTEXT_TYPE    contextType, /* type of context to resume */
    WTX_CONTEXT_ID_T    contextId,   /* id of context to step     */
    TGT_ADDR_T          stepStart,   /* stepStart pc value        */
    TGT_ADDR_T          stepEnd      /* stepEnd PC vale           */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass it.

wtxDirectCall( )

Call a function on the target within the agent.

New Prototype

```
STATUS wtxDirectCall
    (
    HWTX              hWtx,           /* WTX API handle          */
    TGT_ADDR_T        entry,          /* function address        */
    void *            pRetVal,        /* pointer to return value */
    UINT32            argc,           /* arguments count         */
    ...                               /* list of arguments       */
    )
```

Old Prototype

```
STATUS wtxDirectCall
    (
    HWTX              hWtx,           /* WTX API handle          */
    TGT_ADDR_T        entry,          /* function address        */
    void *            pRetVal,        /* pointer to return value */
    TGT_ARG_T         arg0,           /* function arguments      */
    TGT_ARG_T         arg1,
```

```
    TGT_ARG_T         arg2,
    TGT_ARG_T         arg3,
    TGT_ARG_T         arg4,
    TGT_ARG_T         arg5,
    TGT_ARG_T         arg6,
    TGT_ARG_T         arg7,
    TGT_ARG_T         arg8,
    TGT_ARG_T         arg9
    )
```

The limitation of 10 arguments has been removed, and it is now possible to **wtxDirectCall( )** a list of variable arguments through *argc* and the list itself.

wtxEventpointAdd( )

Create a new WTX eventpoint.

New Prototype

```
UINT32 wtxEventpointAdd
    (
    HWTX         hWtx,              /* WTX API handle             */
    WTX_EVTPT *  pEvtpt             /* eventpoint descriptor      */
    )
```

Old Prototype

```
UINT32 wtxEventpointAdd
    (
    HWTX           hWtx,            /* WTX API handle */
    WTX_EVTPT_2 *  pEvtpt           /* eventpoint descriptor */
    )
```

The **WTX_EVTPT_2** structure has been changed into a **WTX_EVTPT** structure, where the old **WTX_EVTPT_2** was:

```
typedef struct wtx_evtpt_2          /* Eventpoint desc. version 2 */
    {
    WTX_EVENT_2         event;      /* Event to detect version 2  */
    WTX_CONTEXT         context;    /* Context descriptor         */
    WTX_ACTION          action;     /* Action to perform          */
    } WTX_EVTPT_2;
```

The new **WTX_EVTPT** structure is:

```
typedef struct wtx_evtpt            /* eventpoint descriptor      */
    {
    WTX_EVENT    event;             /* event to detect            */
    WTX_CONTEXT  context;           /* context descriptor         */
    WTX_ACTION   action;            /* action to perform          */
    } WTX_EVTPT;
```

The **WTX_EVENT_2** structure has been changed into a **WTX_EVENT** structure, where the old **WTX_EVENT_2** was:

```
typedef struct wtx_event_2          /* Target event version 2      */
    {
    WTX_EVENT_TYPE  eventType;      /* type of event               */
    UINT32          numArgs;        /* Number of arguments         */
    TGT_ARG_T *     args;           /* List of arguments           */
    } WTX_EVENT_2;
```

The new **WTX_EVENT** structure is:

```
typedef struct wtx_event            /* target event                */
    {
    WTX_EVENT_TYPE  eventType;      /* type of event               */
    UINT32          numArgs;        /* number of arguments         */
    TGT_ARG_T *     args;           /* list of arguments           */
    } WTX_EVENT;
```

This means that a WTX 2 application becomes WTX 4 compatible if the **WTX_EVTPT_2** and **WTX_EVENT_2** structures are replaced by **WTX_EVTPT** and **WTX_EVENT** structures.

wtxEventpointListGet( )

List eventpoints on the target server.

New Prototype

```
WTX_EVTPT_LIST * wtxEventpointListGet
    (
    HWTX  hWtx                      /* WTX API handle              */
    )
```

Old Prototype

```
WTX_EVTPT_LIST_2 *   wtxEventpointListGet
    (
    HWTX             hWtx           /* WTX API handle              */
    )
```

The routine now returns a **WTX_EVTPT_LIST** structure pointer instead of a **WTX_EVTPT_LIST_2** pointer structure, where a **WTX_EVTPT_LIST** is:

```
typedef struct wtx_evtpt_list/* eventpoint list message             */
    {
    UINT32           nEvtpt;        /* number of eventpoints in list    */
    WTX_EVTPT_INFO * pEvtptInfo;    /* eventpoint information list      */
    } WTX_EVTPT_LIST;
```

wtxFuncCall( )

Call a function on the target.

New Prototype

```
WTX_CONTEXT_ID_T wtxFuncCall
    (
    HWTX                    hWtx,           /* WTX API handle             */
    WTX_TASK_CONTEXT_DEF *  pTaskContextDef /* pointer to call descriptor */
    )
```

Old Prototype

```
WTX_CONTEXT_ID_T wtxFuncCall
    (
    HWTX                    hWtx,           /* WTX API handle */
    WTX_CONTEXT_DESC *      pContextDesc    /* pointer to call descriptor */
    )
```

Since the introduction of processes descriptors, a WTX context can either be a task, or an RTP. The **wtxFuncCall( )** routine now uses the context as a task. Thus, the **WTX_TASK_CONTEXT_DEF** pointer replaces the **WTX_CONTEXT_DESC** pointer, where **WTX_TASK_CONTEXT_DEF** is:

```
typedef struct wtx_task_context_def        /* task creation descriptor    */
    {
    TGT_ADDR_T              pdId;          /* Not used in VxWorks         */
    WTX_RETURN_TYPE         returnType;    /* integer or double           */
    char *                  name;          /* task name                   */
    UINT32                  priority;      /* priority                    */
    UINT32                  options;       /* options                     */
    TGT_ADDR_T              stackBase;     /* base of stack address       */
    UINT32                  stackSize;     /* stack size                  */
    TGT_ADDR_T              entry;         /* context entry point         */
    INT32                   redirIn;      /* redirection in file or NULL   */
    INT32                   redirOut;     /* redirection out file or NULL  */
    INT32                   redirErr;     /* redirection error file or NULL*/
    UINT32                  argc;          /* number of arguments in argv */
    TGT_ARG_T *             argv;          /* arguments array             */
    } WTX_TASK_CONTEXT_DEF;
```

Note that *pdId* is not used in VxWorks and should be set to 0.

wtxGopherEval( )

Evaluate a Gopher string on the target.

New Prototype

```
WTX_GOPHER_TAPE * wtxGopherEval
    (
    HWTX                hWtx,           /* WTX API handle              */
    TGT_ADDR_T          pdId,           /* targeted Process Descriptor */
    const char *        inputString     /* gopher program to evaluate  */
    )
```

Old Prototype

```
WTX_GOPHER_TAPE * wtxGopherEval
    (
    HWTX                hWtx,           /* WTX API handle */
    const char *        inputString     /* Gopher program to evaluate  */
    )
```

The *pdId* parameter allows you to specify in which process descriptor the call will take place.

pdId
Where *pdId* should be a valid RTP ID.

wtxHwBreakpointAdd( )

Create an eventpoint or a hardware breakpoint.

New Prototype

```
WTX_TGT_ID_T wtxHwBreakpointAdd
    (
    HWTX                hWtx,           /* WTX API handle               */
    WTX_CONTEXT *       pContext,
    WTX_TGT_ADDR_T      tgtAddr,        /* breakpoint address           */
    WTX_TGT_INT_T       type            /* access type (arch dependant) */
    )
```

Old Prototype

```
UINT32 wtxHwBreakpointAdd
    (
    HWTX                hWtx,           /* WTX API handle */
    WTX_CONTEXT_TYPE    contextType,    /* type of context to put bp in */
    WTX_CONTEXT_ID_T    contextId,      /* associated context */
    TGT_ADDR_T          tgtAddr,        /* breakpoint address */
    TGT_INT_T           type            /* access type (arch dependant) */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass the pointer to it.

wtxMemAddToPool( )

Add memory to the agent pool.

New Prototype

```
STATUS wtxMemAddToPool
    (
    HWTX                hWtx,           /* WTX API handle              */
    TGT_ADDR_T          pdId,           /* PD ID to look into          */
    TGT_ADDR_T          address,        /* base of added memory block  */
    UINT32              size            /* size of added memory block  */
    )
```

Old Prototype

```
STATUS wtxMemAddToPool
    (
    HWTX                hWtx,           /* WTX API handle */
    TGT_ADDR_T          address,        /* base of memory block to add */
    UINT32              size            /* size of memory block to add */
    )
```

In VxWorks, *pdId* should be set to NULL. Only memory belonging to the kernel can be added.

wtxMemAlign( )

Allocate aligned target memory.

New Prototype

```
TGT_ADDR_T wtxMemAlign
    (
    HWTX         hWtx,          /* WTX API handle                    */
    TGT_ADDR_T   pdId,          /* PD into which we want to operate  */
    TGT_ADDR_T   alignment,     /* alignment boundary                */
    UINT32       numBytes       /* size of block to allocate         */
    )
```

Old Prototype

```
TGT_ADDR_T wtxMemAlign
    (
    HWTX         hWtx,           /* WTX API handle                   */
    TGT_ADDR_T   alignment,      /* alignment boundary               */
    UINT32       numBytes        /* size of block to allocate        */
    )
```

In VxWorks, *pdId* should be set to NULL.

wtxMemAlloc( )

Allocate a block of memory to the target-server-managed target memory pool.

New Prototype

```
TGT_ADDR_T wtxMemAlloc
    (
    HWTX         hWtx,          /* WTX API handle                    */
    TGT_ADDR_T   pdId,          /* PD into which we want to allocate */
    UINT32       numBytes       /* size to allocate in bytes         */
    )
```

Old Prototype

```
TGT_ADDR_T wtxMemAlloc
    (
    HWTX         hWtx,           /* WTX API handle                   */
    UINT32       numBytes        /* size to allocate in bytes        */
    )
```

In VxWorks, *pdId* should be set to NULL.

wtxMemChecksum( )

Perform a checksum on target memory.

New Prototype

```
UINT32 wtxMemChecksum
    (
    HWTX          hWtx,         /* WTX API handle                    */
    TGT_ADDR_T    pdId,         /* PD ID to look into                */
    TGT_ADDR_T    startAddr,    /* remote addr to start checksum at  */
    UINT32        numBytes      /* number of bytes to checksum       */
    )
```

Old Prototype

```
UINT32 wtxMemChecksum
    (
    HWTX         hWtx,           /* WTX API handle                    */
    TGT_ADDR_T   startAddr,      /* remote addr to start checksum at  */
    UINT32       numBytes        /* number of bytes to checksum       */
    )
```

In VxWorks, *pdId* should be set to NULL.

wtxMemDisassemble( )

Get disassembled instructions matching the given address.

New Prototype

```
WTX_DASM_INST_LIST * wtxMemDisassemble
    (
    HWTX        hWtx,          /* WTX API handle                          */
    TGT_ADDR_T  pdId,          /* PD ID to look into                      */
    TGT_ADDR_T  startAddr,     /* instruction address to match            */
    UINT32      nInst,         /* number of instructions to get           */
    TGT_ADDR_T  endAddr,       /* last address to match                   */
    BOOL32      printAddr,     /* if instruction\xd5 s address appended */
    BOOL32      printOpcodes,  /* if instruction\xd5 s opcodes appended */
    BOOL32      hexAddr        /* for HEX address representation          */
    )
```

Old Prototype

```
WTX_DASM_INST_LIST * wtxMemDisassemble
    (
    HWTX        hWtx,          /* WTX API handle                          */
    TGT_ADDR_T  startAddr,     /* Inst address to match                   */
    UINT32      nInst,         /* number of instructions to get           */
    TGT_ADDR_T  endAddr,       /* Last address to match                   */
    BOOL32      printAddr,     /* if instruction\xd5 s address appended */
    BOOL32      printOpcodes,  /* if instruction\xd5 s opcodes appended */
    BOOL32      hexAddr        /* for hex  adress representation          */
    )
```

pdId

Where *pdValue* should be a valid RTP ID in the case of VxWorks.

wtxMemFree( )

Free a block of target memory.

New Prototype

```
STATUS wtxMemFree
    (
    HWTX        hWtx,          /* WTX API handle                        */
    TGT_ADDR_T  pdId,          /* targeted PD ID                        */
    TGT_ADDR_T  address        /* tgt memory block address to free      */
    )
```

Old Prototype

```
STATUS wtxMemFree
    (
    HWTX        hWtx,          /* WTX API handle */
    TGT_ADDR_T  address        /* target memory block address to free  */
    )
```

In VxWorks, *pdId* should be set to NULL.

wtxMemInfoGet( )

Get information about the target-server-managed memory pool.

New Prototype

```
WTX_MEM_INFO * wtxMemInfoGet
    (
    HWTX        hWtx,           /* WTX API handle                    */
    TGT_ADDR_T  pdId            /* targeted PD                       */
    )
```

Old Prototype

```
WTX_MEM_INFO * wtxMemInfoGet
    (
    HWTX         hWtx            /* WTX API handle           */
    )
```

In VxWorks, *pdId* should be set to NULL.

wtxMemMove( )

Move a block of target memory.

New Prototype

```
STATUS wtxMemMove
    (
    HWTX         hWtx,          /* WTX API handle           */
    TGT_ADDR_T   srcPdId,       /* source PD                */
    TGT_ADDR_T   srcAddr,       /* remote addr to move from */
    TGT_ADDR_T   dstPdId,       /* destination PD           */
    TGT_ADDR_T   destAddr,      /* remote addr to move to   */
    UINT32       numBytes       /* number of bytes to move  */
    )
```

Old Prototype

```
STATUS wtxMemMove
    (
    HWTX         hWtx,           /* WTX API handle           */
    TGT_ADDR_T   SrcAddr,        /* remote addr to move from */
    TGT_ADDR_T   destAddr,       /* remote addr to move to   */
    UINT32       numBytes        /* number of bytes to move  */
    )
```

In VxWorks, *srcPdId* can be set to 0 (kernel) or the valid RTP ID. *dstPdId* can also be set to 0 (the kernel) or to a valid RTP ID.

wtxMemRead( )

Read memory from the target.

New Prototype

```
UINT32 wtxMemRead
    (
    HWTX        hWtx,           /* WTX API handle                  */
    TGT_ADDR_T  pdId,           /* targeted PD ID                  */
    TGT_ADDR_T  fromAddr,       /* target addr to read from        */
    void *      toAddr,         /* local addr to read to           */
    UINT32      numBytes,       /* number of bytes to read         */
    UINT32      options         /* memory read options             */
    )
```

Old Prototype

```
UINT32 wtxMemRead
    (
    HWTX        hWtx,           /* WTX API handle                  */
    TGT_ADDR_T  fromAddr,       /* Target addr to read from        */
    void *      toAddr,         /* Local addr to read to           */
    UINT32      numBytes        /* Bytes to read                   */
    )
```

pdId

Where *pdId* should be a valid RTP ID in the case of VxWorks.

wtxMemRealloc( )

Reallocate a block of target memory.

New Prototype

```
TGT_ADDR_T wtxMemRealloc
    (
    HWTX        hWtx,           /* WTX API handle                    */
    TGT_ADDR_T  pdId,           /* PD into which we want to operate  */
    TGT_ADDR_T  address,        /* memory block to reallocate        */
    UINT32      numBytes        /* new size                          */
    )
```

Old Prototype

```
TGT_ADDR_T wtxMemRealloc
    (
    HWTX        hWtx,           /* WTX API handle                    */
    TGT_ADDR_T  address,        /* memory block to reallocate        */
    UINT32      numBytes        /* new size                          */
    )
```

pdId

Where *pdId* should be a valid RTP ID in the case of VxWorks.

wtxMemScan( )

Scan target memory for the presence or absence of a pattern.

New Prototype

```
STATUS wtxMemScan
    (
    HWTX         hWtx,          /* WTX API handle                    */
    TGT_ADDR_T   pdId,          /* PD ID to look into                */
    BOOL32       match,         /* match/Not-match pattern boolean   */
    TGT_ADDR_T   startAddr,     /* target address to start scan      */
    TGT_ADDR_T   endAddr,       /* target address to finish scan     */
    UINT32       numBytes,      /* number of bytes in pattern        */
    void *       pattern,       /* pointer to pattern to search for  */
    TGT_ADDR_T * pResult        /* pointer to result address         */
    )
```

Old Prototype

```
STATUS wtxMemScan
    (
    HWTX         hWtx,          /* WTX API handle */
    BOOL32       match,         /* Match/Not-match pattern boolean */
    TGT_ADDR_T   startAddr,     /* Target address to start scan */
    TGT_ADDR_T   endAddr,       /* Target address to finish scan */
    UINT32       numBytes,      /* Number of bytes in pattern */
    void *       pattern,       /* Pointer to pattern to search for */
    TGT_ADDR_T * pResult        /* Pointer to result address */
    )
```

pdId
Where *pdId* should be a valid RTP ID in the case of VxWorks.

wtxMemSet( )

Set target memory to a given value.

New Prototype

```
UINT32 wtxMemSet
    (
    HWTX         hWtx,          /* WTX API handle          */
    TGT_ADDR_T   pdId,          /* PD ID to look into      */
    TGT_ADDR_T   addr,          /* remote addr to write to */
    UINT32       numBytes,      /* number of bytes to set  */
    UINT32       val            /* value to set            */
    )
```

Old Prototype

```
UINT32 wtxMemSet
    (
    HWTX         hWtx,          /* WTX API handle           */
    TGT_ADDR_T   addr,          /* remote addr to write to  */
    UINT32       numBytes,      /* number of bytes to set   */
    UINT32       val            /* value to set             */
    )
```

pdId
Where *pdId* should be a valid RTP ID in the case of VxWorks.

wtxMemWidthRead( )

Read memory from the target.

New Prototype

```
UINT32 wtxMemWidthRead
    (
    HWTX          hWtx,          /* WTX API handle           */
    TGT_ADDR_T    pdId,          /* PD ID to look into       */
    TGT_ADDR_T    fromAddr,      /* target addr to read from */
    void *        toAddr,        /* local addr to read to    */
    UINT32        numBytes,      /* number of bytes to read  */
    UINT8         width,         /* value width in bytes     */
    UINT32        options        /* memory read options      */
    )
```

Old Prototype

```
UINT32 wtxMemWidthRead
    (
    HWTX        hWtx,              /* WTX API handle           */
    TGT_ADDR_T  fromAddr,          /* Target addr to read from */
    void *      toAddr,            /* Local addr to read to    */
    UINT32      numBytes,          /* Bytes to read            */
    UINT8       width              /* Value width in bytes     */
    )
```

pdId
Where *pdId* should be a valid RTP ID in the case of VxWorks.

The *options* parameter can be **WTX_MEM_CACHE_BYPASS**. This makes the target server look into the target memory rather than into the target server cache.

wtxMemWidthWrite( )

Write memory on the target.

New Prototype

```
UINT32 wtxMemWidthWrite
    (
    HWTX         hWtx,           /* WTX API handle             */
    TGT_ADDR_T   pdId,           /* PD ID to look into         */
    void *       fromAddr,       /* local addr to write from   */
    TGT_ADDR_T   toAddr,         /* remote addr to write to    */
    UINT32       numBytes,       /* number of bytes to read    */
    UINT8        width,          /* width value: 1, 2, 4 bytes */
    UINT32       options         /* memory write options       */
    )
```

Old Prototype

```
UINT32 wtxMemWidthWrite
    (
    HWTX        hWtx,             /* WTX API handle             */
    void *      fromAddr,         /* Local addr to write from   */
    TGT_ADDR_T  toAddr,           /* Remote addr to write to    */
    UINT32      numBytes,         /* Bytes to read              */
    UINT8       width             /* Width value: 1, 2, 4 bytes */
    )
```

pdId
Where *pdId* should be a valid RTP ID in the case of VxWorks.

The *options* parameter can be **WTX_MEM_CACHE_BYPASS**. This makes the target server look into the target memory rather than into the target server cache.

wtxMemWrite( )

Write memory on the target.

New Prototype

```
UINT32 wtxMemWrite
    (
    HWTX         hWtx,          /* WTX API handle           */
    TGT_ADDR_T   pdId,          /* targeted PD ID           */
    void *       fromAddr,      /* local addr to write from */
    TGT_ADDR_T   toAddr,        /* remote addr to write to  */
    UINT32       numBytes,      /* number of bytes to read  */
    UINT32       options        /* memory write options     */
    )
```

Old Prototype

```
UINT32 wtxMemWrite
    (
    HWTX        hWtx,           /* WTX API handle                    */
    void *      fromAddr,       /* Local addr to write from          */
    TGT_ADDR_T  toAddr,         /* Remote addr to write to           */
    UINT32      numBytes        /* Bytes to read                     */
    )
```

pdId

Where *pdId* should be a valid RTP ID in the case of VxWorks.

The *options* parameter can be **WTX_MEM_CACHE_BYPASS**. This makes the target server look into the target memory rather than into the target server cache.

wtxObjModuleByNameUnload( )

Unload an object module from the target.

New Prototype

```
STATUS wtxObjModuleByNameUnload
    (
    HWTX         hWtx,          /* WTX API handle                    */
    TGT_ADDR_T   pdId,          /* PD into which we want to operate  */
    UINT32       option,        /* unload option                     */
    const char * name           /* name of module to look for        */
    )
```

Old Prototype

```
STATUS wtxObjModuleByNameUnload
    (
    HWTX        hWtx,           /* WTX API handle                   */
    char *      name            /* Name of module to look for       */
    )
```

pdId
Where *pdId* should be to NULL.

The *option* field can be:

WTX_UNLOAD_CPLUS_XTOR_AUTO
Automatically call the destructor.

WTX_UNLOAD_CPLUS_XTOR_MANUAL
Manually call the destructor.

WTX_UNLOAD_KEEP_BREAKPOINTS
Do not remove breakpoints.

WTX_UNLOAD_FORCE
Unload the module even if it is used.

wtxObjModuleChecksum( )

Check the validity of target memory.

New Prototype

```
STATUS wtxObjModuleChecksum
    (
    HWTX        hWtx,           /* WTX API handler                  */
    TGT_ADDR_T  pdId,           /* protection domain ID to look into  */
    INT32       moduleId,       /* module ID to check               */
    char *      moduleName      /* module name to check             */
    )
```

Old Prototype

```
STATUS wtxObjModuleChecksum
    (
    HWTX        hWtx,           /* WTX API handle */
    INT32       moduleId,       /* Module Id      */
    char *      fileName        /* Module name    */
    )
```

pdId
Where *pdId* should be NULL.

wtxObjModuleFindId( )

Find the ID of an object module given its name.

New Prototype

```
UINT32        wtxObjModuleFindId
    (
    HWTX         hWtx,          /* WTX API handle                      */
    TGT_ADDR_T   pdId,          /* protection domain in which to look */
    const char * moduleName     /* module name to get ID from         */
    )
```

Old Prototype

```
UINT32        wtxObjModuleFindId
    (
    HWTX         hWtx,          /* WTX API handle                        */
    const char * moduleName     /* object module file name               */
    )
```

pdId
Where *pdId* should be NULL.

wtxObjModuleFindName( )

Find the object module name given its ID.

New Prototype

```
char * wtxObjModuleFindName
    (
    HWTX        hWtx,           /* WTX API handle                        */
    TGT_ADDR_T  pdId,           /* protection domain in which to look   */
    UINT32      moduleId        /* module ID to get name from           */
    )
```

Old Prototype

```
char * wtxObjModuleFindName
    (
    HWTX    hWtx,             /* WTX API handle                      */
    UINT32  moduleId          /* id of module to find object name of */
    )
```

pdId
Where *pdId* should be to NULL.

wtxObjModuleInfoAndPathGet( )

Get information on a module given its ID.

New Prototype

```
WTX_MODULE_INFO * wtxObjModuleInfoAndPathGet
    (
    HWTX        hWtx,           /* WTX API handler                      */
    TGT_ADDR_T  pdId,           /* protection domain to look into       */
    UINT32      moduleId        /* module ID to get name from           */
    )
```

Old Prototype

```
WTX_MODULE_INFO * wtxObjModuleInfoAndPathGet
    (
    HWTX                hWtx,         /* WTX API handle                */
    UINT32              modId         /* id of module to look for      */
    )
```

pdId
Where *pdId* should be NULL.

wtxObjModuleInfoGet( )

Get information on a module given its ID.

New Prototype

```
WTX_MODULE_INFO * wtxObjModuleInfoGet
    (
    HWTX                hWtx,       /* WTX API handler                  */
    TGT_ADDR_T          pdId,       /* protection domain to look into   */
    UINT32              moduleId    /* module ID to get name from       */
    )
```

Old Prototype

```
WTX_MODULE_INFO * wtxObjModuleInfoGet
    (
    HWTX                hWtx,       /* WTX API handle                   */
    UINT32              modId       /* id of module to look for         */
    )
```

pdId
Where *pdId* should be NULL.

wtxObjModuleLoad( )

Load a multiple section object file.

New Prototype

```
WTX_MODULE_INFO * wtxObjModuleLoad
    (
    HWTX                    hWtx,          /* WTX API handle                */
    TGT_ADDR_T              pdId,          /* load destination PD           */
    WTX_MODULE_FILE_DESC *  pFileDesc,     /* module descriptor             */
    UINT32                  loadOptions    /* load options                  */
    )
```

Old Prototype

```
WTX_LD_M_FILE_DESC * wtxObjModuleLoad
    (
    HWTX                    hWtx,            /* WTX API handle    */
    WTX_LD_M_FILE_DESC *    pFileDesc        /* Module descriptor */
    )
```

pdId
Where *pdId* should be NULL.

The pointer *pFileDesc* now points to a **WTX_MODULE_FILE_DESC** structure instead of to a **WTX_LD_M_FILE_DESC** structure. It has almost the same structure and names, except that the **LD_M_SECTIONSpointer** *sections* has been replaced with a **WTX_SECTION_DESC** pointer named *sections*.

The new *loadOptions* parameter allows you to use *moduleId* to specify if a file has to be loaded from the target server file system or from the tool file system. Set *loadOptions* to **WTX_LOAD_FROM_TOOL** if you wish to load the file from the tool file system. Set it to NULL if you wish to load the file from the target server file system.

The returned **WTX_MODULE_INFO** pointer should be freed using **wtxResultFree( )** and not **free( )**.

Table 6-2 compares former and new returned parameters:

Table 6-2 **Comparison of Tornado 2.2 and Workbench 2.5 Returned Parameters**

WTX_LD_M_FILE for Tornado 2.2	WTX_MODULE_INFO for Workbench 2.5	Comments
filename	*moduleName*	variable name has changed
loadFlag	*loadFlag*	

Table 6-2 **Comparison of Tornado 2.2 and Workbench 2.5 Returned Parameters** (cont'd)

WTX_LD_M_FILE for Tornado 2.2	WTX_MODULE_INFO for Workbench 2.5	Comments
moduleId	*moduleId*	
nSections	*nSections*	
section	*section*	variable type differs
pReserved	*pReserved*	
N/A	*pdId*	N/A
N/A	format	N/A

wtxObjModuleLoadStart( )

Load a multiple section object file asynchronously.

New Prototype

```
STATUS wtxObjModuleLoadStart
    (
    HWTX                    hWtx,           /* WTX API handle      */
    TGT_ADDR_T              pdId,           /* load destination PD */
    WTX_MODULE_FILE_DESC *  pFileDesc,      /* module descriptor   */
    UINT32                  loadOptions     /* load options        */
    )
```

Old Prototype

```
STATUS wtxObjModuleLoadStart
    (
    HWTX                  hWtx,             /* WTX API handle      */
    WTX_LD_M_FILE_DESC *  pFileDesc         /* Module descriptor   */
    )
```

pdId
Where *pdId* should be NULL.

The pointer *pFileDesc* now points to a **WTX_MODULE_FILE_DESC** structure instead of to a **WTX_LD_M_FILE_DESC** structure. It has almost the same structure and names, except that the **LD_M_SECTIONSpointer** *sections* has been replaced with a **WTX_SECTION_DESC** pointer named *sections*.

The new *loadOptions* parameter allows you to replace the use of *moduleId* to specify if a file has to be loaded from the target server file system or from the tool file

system. Set *loadOptions* to **WTX_LOAD_FROM_TOOL** if you wish to load the file from the tool file system. Set it to NULL if you wish to load the file from the target server file system.

wtxObjModuleUnload( )

Unload an object module from the target.

New Prototype

```
STATUS wtxObjModuleUnload
    (
    HWTX         hWtx,          /* WTX API handle                    */
    TGT_ADDR_T   pdId,          /* PD into which we want to operate  */
    UINT32       option,        /* unload option                     */
    UINT32       moduleId       /* ID of module to unload            */
    )
```

Old Prototype

```
STATUS wtxObjModuleUnload
    (
    HWTX         hWtx,          /* WTX API handle                    */
    UINT32       moduleId       /* id of module to unload            */
    )
```

pdId
Where *pdId* should be NULL.

The *option* field can be:

WTX_UNLOAD_CPLUS_XTOR_AUTO
Automatically call the destructor.

WTX_UNLOAD_CPLUS_XTOR_MANUAL
Manually call the destructor.

WTX_UNLOAD_KEEP_BREAKPOINTS
Do not remove breakpoints.

WTX_UNLOAD_FORCE
Unload the module even if it is used.

wtxRegsGet( )

Read register data from the target.

New Prototype

```
STATUS wtxRegsGet
    (
    HWTX              hWtx,      /* WTX API handle                  */
    WTX_CONTEXT *     pContext,  /* WTX Context                     */
    WTX_REG_SET_TYPE  regSet,    /* type of register set            */
    UINT32            firstByte, /* first byte of register set      */
    UINT32            numBytes,  /* number of bytes of register set */
    void *            regMemory  /* place holder for reg. values    */
    )
```

Old Prototype

```
STATUS wtxRegsGet
    (
    HWTX              hWtx,        /* WTX API handle                  */
    WTX_CONTEXT_TYPE  contextType, /* context type to get regs of     */
    WTX_CONTEXT_ID_T  contextId,   /* context id to get regs of       */
    WTX_REG_SET_TYPE  regSet,      /* type of register set            */
    UINT32            firstByte,   /* first byte of register set      */
    UINT32            numBytes,    /* number of bytes of register set */
    void *            regMemory    /* place holder for reg. values    */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass it.

wtxRegsSet( )

Write to registers on the target.

New Prototype

```
STATUS wtxRegsSet
    (
    HWTX              hWtx,        /* WTX API handle                  */
    WTX_CONTEXT       pContext,
    WTX_REG_SET_TYPE  regSet,      /* type of register set            */
    UINT32            firstByte,   /* first byte of reg. set          */
    UINT32            numBytes,    /* number of bytes in reg. set.    */
    void *            regMemory    /* register contents               */
    )
```

Old Prototype

```
STATUS wtxRegsSet
    (
    HWTX              hWtx,         /* WTX API handle                 */
    WTX_CONTEXT_TYPE  contextType,  /* context type to set regs of    */
    WTX_CONTEXT_ID_T  contextId,    /* context id to set regs of      */
    WTX_REG_SET_TYPE  regSet,       /* type of register set           */
    UINT32            firstByte,    /* first byte of reg. set         */
```

```
    UINT32            numBytes,     /* number of bytes in reg. set.   */
    void *            regMemory     /* register contents              */
    )
```

Create a structure **WTX_CONTEXT**, populate it, and pass it.

wtxSymAdd( )

Add a symbol with the given name, value, and type.

New Prototype

```
STATUS wtxSymAdd
    (
    HWTX          hWtx,           /* WTX API handle                   */
    TGT_ADDR_T    pdId,           /* PD into which we want to operate */
    const char *  name,           /* name of symbol to add            */
    TGT_ADDR_T    value,          /* value of symbol to add           */
    UINT32        type            /* type of symbol to add            */
    )
```

Old Prototype

```
STATUS wtxSymAdd
    (
    HWTX          hWtx,           /* WTX API handle           */
    const char *  name,           /* name of symbol to add    */
    TGT_ADDR_T    value,          /* value of symbol to add   */
    UINT8         type            /* type of symbol to add    */
    )
```

pdId

Where *pdId* should be NULL.

wtxSymFind( )

Find information on a symbol given its name or value.

New Prototype

```
WTX_SYMBOL * wtxSymFind
    (
    HWTX                        hWtx,      /* WTX API handle     */
    WTX_SYM_FIND_CRITERIA *     pCriteria  /* criteria structure */
    )
```

Old Prototype

```
WTX_SYMBOL * wtxSymFind
    (
    HWTX          hWtx,          /* WTX API handle                    */
    const char *  symName,       /* name of symbol                    */
    TGT_ADDR_T    symValue,      /* value of symbol                   */
    BOOL32        exactName,     /* must match name exactly           */
    UINT8         symType,       /* type of symbol                    */
    UINT8         typeMask       /* mask to select type bits          */
    )
```


The **wtxSymFind( )** routine now uses a single **WTX_SYM_FIND_CRITERIA** for input parameter, where a **WTX_SYM_FIND_CRITERIA** is the following:

```
typedef struct wtx_sym_find_criteria
    {
    UINT32        options;       /* option flag                       */
    TGT_ADDR_T    pdId;          /* PD ID or WTX_SYM_FIND_IN_ALL_PD   */
    char *        findName;      /* symbol name (regular expression)  */
    TGT_ADDR_T    findValue;     /* symbol value (address)            */
    UINT32        nSymbols;      /* number of symbols to get by value */
    UINT32        type;          /* type value                        */
    TGT_ADDR_T    ref;           /* reference value                   */
    UINT32        moduleId;      /* module ID value                   */
    char *        moduleName;    /* module name value                 */
    } WTX_SYM_FIND_CRITERIA;
```

pdId
: Where *pdId* should be NULL.

The translation table between former and new parameters usage is shown in Table 6-3

Table 6-3 **Parameter Translation Table**

Former Parameter	New Parameter	Comment
symName	pCriteria>findName	Turn on **WTX_SYM_FIND_BY_NAME** in *options*.
symValue	pCriteria>findValue	Turn on **WTX_SYM_FIND_BY_VALUE** in *options*.
exactName	pCriteria>options	Turn on **WTX_FIND_BY_EXACT_NAME** flag.
symType	pCriteria>type	Turn on **WTX_SYM_FILTER_ON_TYPE** in *options*.

Table 6-3 **Parameter Translation Table** (cont'd)

Former Parameter	New Parameter	Comment
typeMask	pCriteria>options	Create your *options* according to *typeMask*.
N/A	pCriteria>pdId	The symbol belongs to an RTP (new).
N/A	pCriteria>ref	
N/A	pCriteria>nSymbols	Specifies the maximum number of symbols to get (new).

wtxSymListByModuleIdGet( )

Get a list of symbols given a module ID.

New Prototype

```
WTX_SYM_LIST * wtxSymListByModuleIdGet
    (
    HWTX         hWtx,           /* WTX API handle                    */
    TGT_ADDR_T   pdId,           /* protection domain ID to look into */
    const char * substring,      /* symbol name substring to match    */
    UINT32       moduleId,       /* module ID to search in            */
    TGT_ADDR_T   value,          /* symbol value to match             */
    BOOL32       listUnknown     /* list unknown symbols only flag    */
    )
```

Old Prototype

```
WTX_SYM_LIST * wtxSymListByModuleIdGet
    (
    HWTX         hWtx,           /* WTX API handle */
    const char * substring,      /* Symbol name substring to match */
    UINT32       moduleId,       /* Module Id to search in */
    TGT_ADDR_T   value,          /* Symbol value to match */
    BOOL32       listUnknown     /* List unknown symbols only flag */
    )
```

pdId

Where *pdId* should be to NULL.

wtxSymListByModuleNameGet( )

Get a list of symbols given a module name.

New Prototype

```
WTX_SYM_LIST * wtxSymListByModuleNameGet
    (
    HWTX         hWtx,           /* WTX API handle                    */
    TGT_ADDR_T   pdId,           /* protection domain ID to look into */
    const char * substring,      /* symbol name substring to match    */
    const char * moduleName,     /* module name to search in          */
    TGT_ADDR_T   value,          /* symbol value to match             */
    BOOL32       listUnknown     /* list unknown symbols only flag    */
    )
```

Old Prototype

```
WTX_SYM_LIST * wtxSymListByModuleNameGet
    (
    HWTX         hWtx,           /* WTX API handle                    */
    const char * substring,      /* Symbol name substring to match    */
    const char * moduleName,     /* Module name to search in          */
    TGT_ADDR_T   value,          /* Symbol value to match             */
    BOOL32       listUnknown     /* List unknown symbols only flag    */
    )
```

pdId
Where *pdId* should be NULL.

wtxSymListGet( )

Get a list of symbols matching the given search criteria.

New Prototype

```
WTX_SYM_LIST * wtxSymListGet
    (
    HWTX                        hWtx,          /* WTX API handler     */
    WTX_SYM_FIND_CRITERIA *     pCriteria      /* criteria structure  */
    )
```

Old Prototype

```
WTX_SYM_LIST * wtxSymListGet
    (
    HWTX         hWtx,           /* WTX API handle                     */
    const char * substring,      /* Symbol name substring to match     */
    const char * moduleName,     /* Module name to search in           */
    TGT_ADDR_T   value,          /* Symbol value to match              */
    BOOL32       listUnknown     /* List unknown symbols only flag     */
    )
```

This routine now uses the **WTX_SYM_FIND_CRITERIA** structure to know what kind of symbols to get. A **WTX_SYM_FIND_CRITERIA** structure has the following description:

```
typedef struct wtx_sym_find_criteria
    {
    UINT32       options;        /* option flag                          */
    TGT_ADDR_T   pdId;           /* PD ID or WTX_SYM_FIND_IN_ALL_PD      */
    char *       findName;       /* symbol name (regular expression)     */
    TGT_ADDR_T   findValue;      /* symbol value (address)               */
    UINT32       nSymbols;       /* number of symbols to get by value    */
    UINT32       type;           /* type value                           */
    TGT_ADDR_T   ref;            /* reference value                      */
    UINT32       moduleId;       /* module ID value                      */
    char *       moduleName;     /* module name value                    */
    } WTX_SYM_FIND_CRITERIA;
```

pdId
Where *pdId* should be NULL.

The table of equivalence of parameters between former and new use of **wtxSymListGet( )** is shown in Table 6-4.

Table 6-4 **Table of Equivalency**

Former Parameter	New Parameter	Comment
substring	pCriteria>findName	Turn on the **WTX_SYM_FIND_BY_NAME** flag of the pCriteria>options field.
moduleName	pCriteria>moduleName	Turn on the **WTX_FILTER_ON_MODULE_NAME** flag of the pCriteria>options field.
value	pCriteria>findValue	Turn on the **WTX_SYM_FIND_BY_VALUE** flag of the pCriteria>options field.
listUnkown	N/A	not yet supported
N/A	pCriteria>pdId	A symbol belongs to an RTP.
N/A	pCriteria>nSymbols	Specify the maximum number of symbols to retrieve when looking up the symbol value.
N/A	pCriteria>type	Filter symbols by their type.

Table 6-4 **Table of Equivalency** (cont'd)

Former Parameter	New Parameter	Comment
N/A	pCriteria>ref	Filter symbols by their reference number.
N/A	pCriteria>moduleId	Retrieve symbols by their *moduleId*.

wtxSymRemove( )

Remove a symbol from the default target-server symbol table.

New Prototype

```
STATUS wtxSymRemove
    (
    HWTX          hWtx,          /* WTX API handle                    */
    TGT_ADDR_T    pdId,          /* PD into which we want to operate */
    const char *  symName,       /* name of symbol to remove          */
    UINT32        symType        /* type of symbol to remove          */
    )
```

Old Prototype

```
STATUS wtxSymRemove
    (
    HWTX         hWtx,           /* WTX API handle                    */
    const char * symName,        /* Name of symbol to remove          */
    UINT8        symType         /* Type of symbol to remove          */
    )
```

pdId
Where *pdId* should be NULL.

Note that the *symType* parameter type has changed, and is no longer a UINT8, but a UIN32.

wtxSymTblInfoGet( )

Return information about the target server symbol table.

New Prototype

```
WTX_SYM_TBL_INFO * wtxSymTblInfoGet
    (
    HWTX         hWtx,           /* WTX API handle                    */
    TGT_ADDR_T   pdId            /* PD into which we want to operate */
    )
```

Old Prototype

```
WTX_SYM_TBL_INFO * wtxSymTblInfoGet
    (
    HWTX         hWtx            /* WTX API handle */
    )
```

pdId
Where *pdId* should be NULL.

6.3 Obsoleted Routines

These routines are no longer supported.

Table 6-5 **Obsoleted Functions for Wind River Workbench 2.5**

Obsoleted Function	Replacement Where Applicable
wtxAsyncResultFree( )	**wtxFreeAdd( )**
wtxCommandSend( )	No longer supported as a public API.
wtxConsoleCreate( )	This API has been replaced by the **wtxConsole** tool; see *A. WTX Console*.
wtxConsoleKill( )	This API has been replaced by the **wtxConsole** tool; see *A. WTX Console*.
wtxEach( )	No longer supported as a public API.
wtxEventpointList( )	**wtxEventpointListGet( )**
wtxObjModuleList( )	**wtxObjModuleListGet( )**
wtxServiceAdd( )	No longer supported as a public API.

Table 6-5 **Obsoleted Functions for Wind River Workbench 2.5** (cont'd)

Obsoleted Function	Replacement Where Applicable
wtxSymAddWithGroup( )	No longer supported as a public API.
wtxSymListFree( )	**wtxFreeAdd( )**
wtxTargetRtTypeGet( )	*rtType* has been replaced by *rtName*, which can be retrieved by **wtxTsInfoGet( )**. For more information, see the **wtx** reference entry.
wtxVioFileList( )	**wtxVioFileListGet( )**

6.4 New Routines

This section lists all the new WTX C routines. For more information, see the related reference entries.

Table 6-6 **New Functions for Wind River Workbench 2.5**

New Function	Description
wtxContextStop( )	stop execution of a target context
wtxFreeAdd( )	add a pointer to the WTX free list
wtxObjModuleListGet( )	fetch a list of loaded object modules from the target
wtxProcessCreate( )	create a new RTP
wtxProcessDelete( )	delete the given RTP from target
wtxRegistryEntryAdd( )	add an entry on a given *host* registry
wtxRegistryEntryGet( )	get registry information on an item for the given *host*
wtxRegistryEntryListGet( )	ask the given *host* for a registry entry list
wtxRegistryEntryRemove( )	remove an entry from the given *host* registry

Table 6-6 **New Functions for Wind River Workbench 2.5** (cont'd)

New Function	Description
wtxRegistryEntryUpdate( )	update an entry key on a given *host* registry
wtxRegistryLogGet( )	get the log file content of a registry
wtxRegistryTimeoutGet( )	get the registry connection timeout
wtxRegistryTimeoutSet( )	set the registry connection timeout in milliseconds
wtxTargetBspShortNameGet( )	get the target board-support-package name string
wtxTargetRtNameGet( )	get the target run-time name
wtxTargetToolNameGet( )	get the tool used to build the target operating system
wtxTgtsvrStart( )	start a target server on a given *host*
wtxTgtsvrStop( )	stop a target server on a given *host*
wtxToolOnHostAttach( )	connect a WTX client to a remote target server
wtxTsLogGet( )	get the log file content of a target server
wtxVioFileListGet( )	get the list of files handled by the target server
wtxVioLink( )	get a VIO channel and redirect it on two socket file descriptors
wtxVioUnlink( )	release a channel and close any sockets allocated by **wtxVioLink( )**

PART 3

References

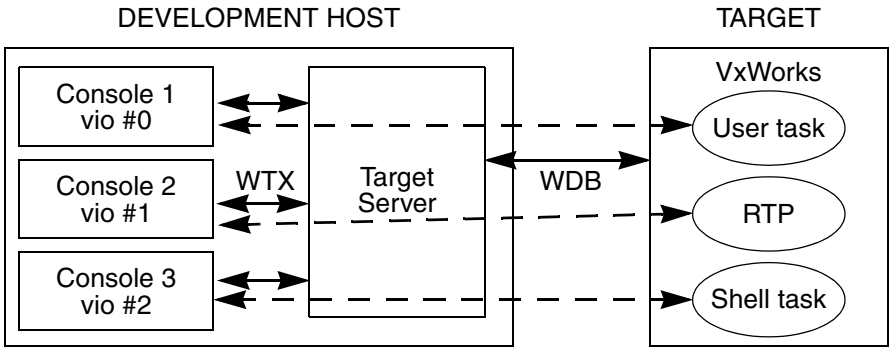
A

WTX Console

A.1 Introduction

Wind River Workbench provides a target server console tool, **wtxConsole**, that can be used for a variety of purposes. It is a standalone WTX tool, which attaches to the target server. Its main purpose is to redirect the I/O of target tasks, user tasks, RTPs, and such system tasks as the kernel shell. This tool can be helpful in certain circumstances. For example, when no serial port or network device is available for use by a target console, the WDB/WTX link can carry I/O to the WTX console. The purpose of the console is not to substitute for the host shell. The console tool does not provide shell functionality; it can only redirect I/O.

Figure A-1 **The Console Tool, Workbench, and the Target**



This chapter explains how to use the WTX console to:

- Redirect all target I/O.
- Redirect the I/O of a user task running on the target.
- Redirect kernel shell I/O (a system task) and access a file located on a network file system that is not visible from the host on which Workbench is running.
- Redirect the I/O of an RTP.

NOTE: Redirecting task I/O is useful for debugging tasks.

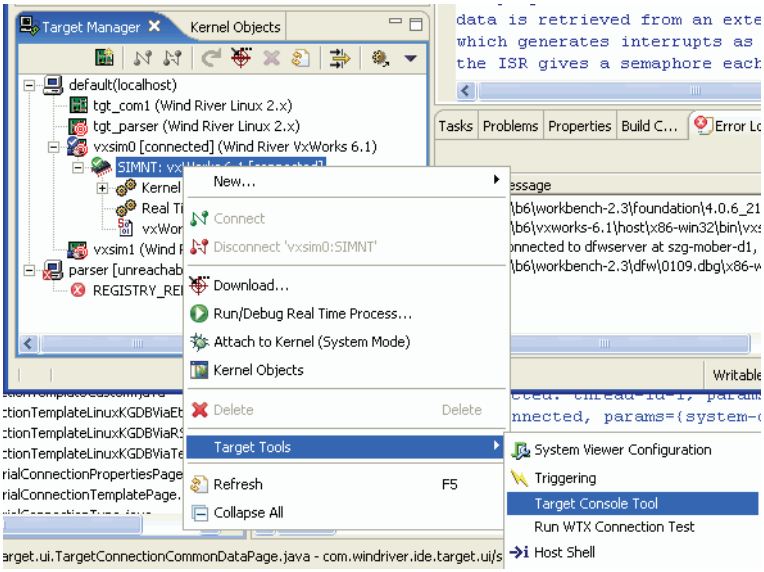
A.2 Using the Console

You can launch the console tool from the command line or from Workbench. You must have a target or simulator running before you can start a console session.

A.2.1 Starting the Console

To start **wtxConsole** from Workbench, right-click on your target server and select **Target Tools > Target Console Tool** to open the option dialog box (see Figure A-2).

Figure A-2 **Starting the Target Console from the Target Manager Menu**



To start **wtxConsole** from the command line, specify I/O redirection, a title, and which session to attach to, for example:

```
% start wtxConsole -io -t "my Console on menezbre" vxsim0-session
```

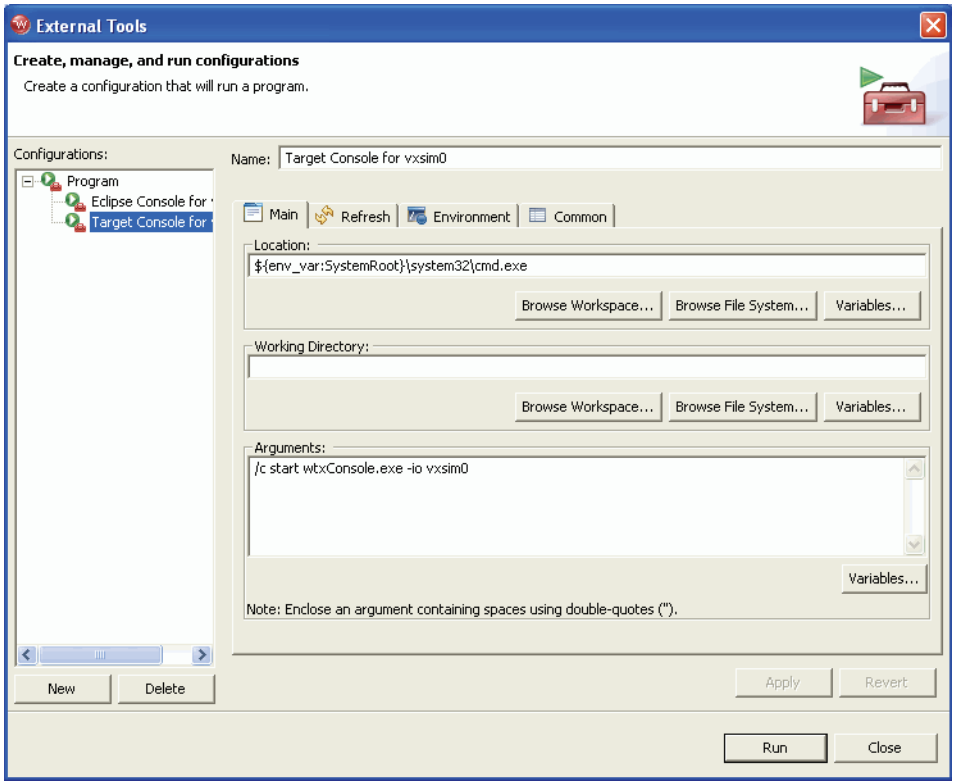
A.2.2 Automating Startup

Using a Windows Window

To add WTX console startup to the external tools menu, do the following:

1. Select **Run > External Tools > External Tools**.
2. Fill in the dialog as shown in Figure A-3.

Figure A-3 **Configuring an External Tool Entry To Start Console In a Windows Window**

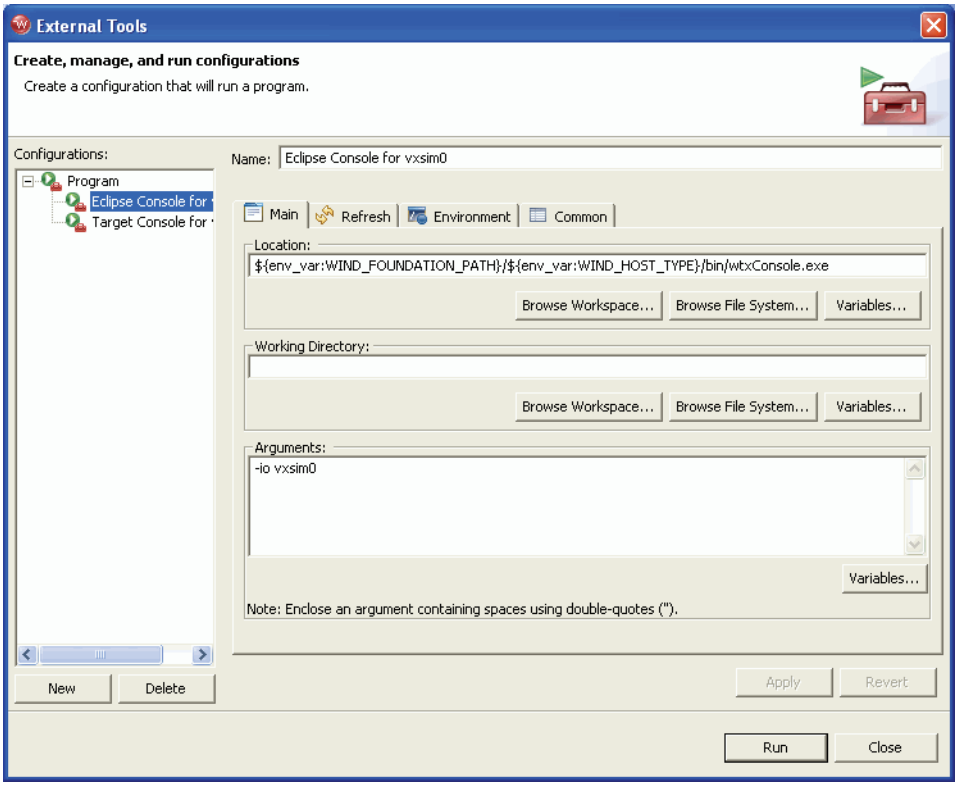


Using the Eclipse Console

To create a menu entry for a console that opens in the Eclipse console:

1. Select **Run > External Tools > External Tools**.
2. Fill in the dialog as shown in Figure A-3.
3. The target output will be in an Eclipse view named Console; open this view by selecting **Window > Show View > Other > Basic > Console**.
4. Terminate the console by choosing terminate in the debugger list.

Figure A-4 **Configuring an External Tool Entry To Start Console In an Eclipse View**



A.2.3 Redirecting the Target I/O

This section shows how to use the console tool to manage target I/O. Connect the console tool to the target server **mysession@myhost**, using the command line parameter **-io** or by checking the **Redirect target IOs in the console** box to enable target I/O redirection.

```
% wtxConsole -io mysession@myhost
Please press Control+ to close the console.
```

If you launch the tool from Workbench, you have to check to be sure that the host shell is not redirecting target I/O; set the **SH_GET_TASK_IO** shell environment variable to off. Then from the host shell, print a string using the **printf( )** target function:

```
% windsh mysession@myhost
[vxWorks]# ?shConfig SH_GET_TASK_IO off
[vxWorks]# printf "Hello from Host shell\n"
```

In the console you can see a new line printed, as indicated by the angle bracket (>):

```
    Please press Control+ to close the console.
>   Hello from Host shell
```

If you load an object module such as **funcCallTest.o** onto the target, you can call a function from the shell. A sample function, **funcCallIntTest( )**, is listed here:

```
/*******************************************************************************
*
* funcCallIntTest - This program adds ten integers and prints the result
*
* RETURN : it returns the total of the addition of its ten parameters.
*/

int funcCallIntTest
(
int arg1, /* 10 arguments */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8,
int arg9,
int arg10
)
{
int total;

total = arg1 + arg2 + arg3 + arg4 + arg5 + arg6 +
                      arg7 + arg8 + arg9 + arg10;

printf ("Total (int)    : %d\n", total);

return (total);
}
```

In the host shell, if you load the module and call the test function, you see the following:

```
-> ld<directory/objppc/funcCallTest.o
value = 1616114216 = 0x6053f228
-> funcCallIntTest (0,1,2,3,4,5,6,7,8,9)
value = 45 = 0x2d = '-'
->
```

Another line is added to the output in the console window:

```
    Please press Control+ to close the console.
    Hello from Host shell
>   Total (int)    : 45
```

A.2.4 **Redirecting I/O of a User Task**

This example demonstrates how to use the console tool to communicate with a user task running on the target. It is possible to launch as many task-console pairs as you wish; the only limitation is the number of virtual I/O numbers available (up to 255).

The program code below reads a number from standard input and checks whether it belongs to a valid range. If so, it prints its square to standard output. Otherwise it prints a warning to standard error:

```
#include "stdio.h"

/*****************************************************************************
*
* squareCompute.c - This sample reads a number and prints its square.
*
* Ensure INCLUDE_ANSI_STDIO_FSCANF is defined in your project config file,
* before building your target.
*
*/

#define MAX 256
#define MIN 16

int squareCompute ()
    {
    int d = 0;

    while (d != -1)
        {
    fprintf (stdout, "Enter a # between %d and %d (-1 to quit): ", MIN, MAX);
        fflush (stdout);
        fscanf (stdin, "%d", &d);

        if (d != -1)
            {
            if (d<MIN) fprintf (stderr, "Too small.\n");
            else if (d>MAX) fprintf (stderr, "Too big.\n");
                else fprintf (stdout, "%d ^ 2 = %d\n", d, d * d);
            fflush (stdout);
            fflush (stderr);
            }

        }
```

```
fprintf (stdout, "\nExiting squareCompute().\n");

return (0);
}
```

Once you have built the module and loaded it onto the target, you can create a target user task from the host shell using the **wtxContextCreate** Tcl API. Provide all the requested arguments to redirect all the I/O, **stdin**, **stdout**, and **stderr**, on a file descriptor opened on the target VIO number 5. (This number is totally arbitrary but if you wish to run another task-console pair, you can use a consecutive number. Execute the following commands in the host shell:

```
-> ld < /directory/objppc/squareCompute.o
Loading /directory/objppc/squareCompute.o
value = 134700584 = 0x8075e28
-> ?wtxSymFind -name squareCompute
squareCompute 0x60534000 0x5 0 0 ""
-> fd1=open("/vio/5", 2, 0)
fd1 = 0x60345080: value = 6 = 0x6
-> ?wtxV2CompatUnset
2
-> ?wtxContextCreate CONTEXT_TASK myTask5 60 0 0 0 0x60534000 0x6 0x6 0x6
0x60533380
-> ?wtxContextResume CONTEXT_TASK 0x60533380
```

Then launch the console tool with the **-vio 5** command line parameter or check the **Force the target server to use a vio number** box and select the vio number 5. Communication with task I/O is now enabled.

Enter the requested numbers, and check that the output is correct:

```
> wtxConsole -vio 5 mysession@myhost
Please press Control+\ to close the console.
Enter a number between 16 and 256 (-1 to quit) : 7
Too small.
Enter a number between 16 and 256 (-1 to quit) : 34
34 ^ 2 = 1156
Enter a number between 16 and 256 (-1 to quit) : 567
Too big.
Enter a number between 16 and 256 (-1 to quit) : -1

Exiting squareCompute().
^\
Termination signal received, safely exiting.
>
```

A.2.5 Redirecting the Kernel Shell I/O

Finally, you can use the console tool as a kernel shell console. To do so, specify the **-s** command line parameter or check the **Redirect kernel shell IOs in the console** checkbox in Workbench to redirect shell output to the console.

The kernel shell task redirects its I/O to a target file descriptor opened on the virtual I/O number #0 (by default). The console reads and writes on this VIO number and thus is able to communicate with the shell task.

Before trying this example, note that you must rebuild your target including the following components:


```
#define INCLUDE_SHELL           /* interactive c-expression interpreter */
#define INCLUDE_SHELL_VI_MODE   /* vi editing mode for the shell */
#define INCLUDE_SHELL_EMACS_MODE /* emacs editing mode for the shell */
#define INCLUDE_SHELL_INTERP_C  /* C interpreter */
#define INCLUDE_SHOW_ROUTINES   /* show routines for system facilities*/
#define INCLUDE_PASSFS_SYM_TBL  /* compiled-in symbol table */
#define INCLUDE_STAT_SYM_TBL    /* create user-readable error status */
#define INCLUDE_DISK_UTIL       /* ls, cd, mkdir, xcopy, etc */
#define INCLUDE_LOADER          /* object module loading */
#define INCLUDE_UNLOADER        /* object module unloading */
#define INCLUDE_DEBUG           /* native debugging */
#define INCLUDE_RTP             /* Real Time Process */
#define INCLUDE_RTP_APPL_INIT_BOOTLINE  /* RTP Startup bootline Facility */
```

For this example, we boot the target with a core file located on a UNIX host, **cocyte (147.11.80.31)**, accessible through a NFS path. Note that once booted, the target has an access to the **/net/coctyte/cocyte1** file system.

```
[VxWorks Boot]: c

'.' = clear field;  '-' = go to previous field;  ^D = quit

boot device          : lnPci0
processor number     : 0
host name            : cocyte
file name            : /folk/philb/ppcImage/vxWorks
inet on ethernet (e) : 147.11.80.206:ffffff00
inet on backplane (b):
host inet (h)        : 147.11.80.31
gateway inet (g)     :
user (u)             : philb
ftp password (pw) (blank = use rsh):
flags (f)            : 0x8
target name (tn)     : t80-210
startup script (s)   :
other (o)            :

[VxWorks Boot]:
```

Then launch the target server and the console on a Windows host.

```
c:\installDir\vxworks-6.x\host\x86-win32\bin> tgtsvr -n mysession -V
147.11.80.225
c:\installDir\vxworks-6.x\host\x86-win32\bin>
c:\installDir\vxworks-6.x\host\x86-win32\bin> wtxConsole -s mysession
Please press Control+\ to close the console.

->
-> i


  NAME      ENTRY        TID     PRI   STATUS      PC       SP      ERRNO   DELAY
---------- ----------- -------- --- --------- -------- -------- ------ -----
tJobTask   jobTask     6036c010   0 PEND      601137aa 60409f3c      0      0
tExcTask   excTask     6036c4f0   0 PEND      60111605 60417eec      0      0
tLogTask   logTask     6040fa28   0 PEND      60111605 60425f2c      0      0
tNbioLog   60092fa0    603700d8   0 PEND      601137aa 60433f54      0      0
tShell0    shellTask   60513080   1 PEND      601137aa 60529d64      0      0
wtxC15597  shellTask   6053f0e8   1 READY     60119c10 6055a454      0      0
tWdbTask   wdbTask     60371d10   3 READY     6011365c 60509d44      9      0
tAioIoTask 1aioIoTask  6043b8d0  50 PEND      60113e22 60459f88      0      0
tAioIoTask 0aioIoTask  6043bba0  50 PEND      60113e22 6046bf88      0      0
tNetTask   netTask     6044db90  50 PEND      601137aa 60479f94      0      0
tAioWait   aioWaitTask 6042b020  51 PEND      601137aa 60447f28      0      0
value = 0 = 0x0
```

Note the shell redirection, causing the shell prompt to be displayed in the console. Entering the shell command **i( )** checks whether the inputs are also working correctly.

Now from the console prompt, enter a target command that accesses a file **/net/***host*/*directory*/**objppc/test.o** located on a UNIX file system. This file system is not necessarily visible from the Windows host or directly accessible by the other Workbench tools, such as the host shell. Nevertheless we can load this file on the target using the **ml( )** target command.

```
Please press Control+ to close the console.

->
-> ld < /net/host/directory/objppc/test.o
value = 1634709952 = 0x616fb1c0 = mySemaphore + 0x710
```

A.2.6 Redirecting I/O of an RTP

The example illustrates how to redirect the I/O of a RTP in the WTX console. The steps are basically the same as for *A.2.4 Redirecting I/O of a User Task*, p.105, except that we create a RTP instead of a user task.

```
#include "stdio.h"

/************************************************************************
 *
 * squareRtp.c - This sample reads a number and prints its cube.
 *
 */

#define MAX 256
#define MIN 16

int main ()
    {
    int d = 0;

    while (d != -1)
        {
        fprintf (stdout, "Enter a number between %d and %d (-1 to quit)
:", MIN, MAX);
        fflush (stdout);
        fscanf (stdin, "%d", &d);

        if (d != -1)
            {
            if (d<MIN) fprintf (stderr, "Too small.

            else if (d>MAX) fprintf (stderr, "Too big.

                else fprintf (stdout, "%d ^ 3 = %d

            fflush (stdout);
            fflush (stderr);
            }

        }

   fprintf (stdout, "


   return (0);
   }

# Makefile - makefile for cube compute example RTP
#
# modification history
# --------------------
# 01a,03Sep04,nlg  written
#
# DESCRIPTION
# This file contains the makefile rules for building the cube compute RTP

EXE = cubeRtp.vxe

OBJS =  cubeRtp.o

include $(WIND_USR)/make/rules.rtp
```

From the host shell, launch RTP on vio 5:

```
-> fd1=open("/vio/5", 2, 0)
fd1 = 0x60345080: value = 6 = 0x6
-> ?wtxProcessCreate -name /directory/rtpppc/cubeRtp.vxe -rIn 0x6 -rOut 0x6
-rErr 0x6
0x605316d8
->
```

Then run a WTX console:

```
> wtxConsole -vio 5 mysession@host
Please press Control+\ to close the console.
Enter a number between 16 and 256 (-1 to quit) : 7
Too small.
Enter a number between 16 and 256 (-1 to quit) : 25
25 ^ 3 = 15625
^\
Termination signal received, safely exiting.
>
```

B

The WTX Protocol

B.1 Introduction

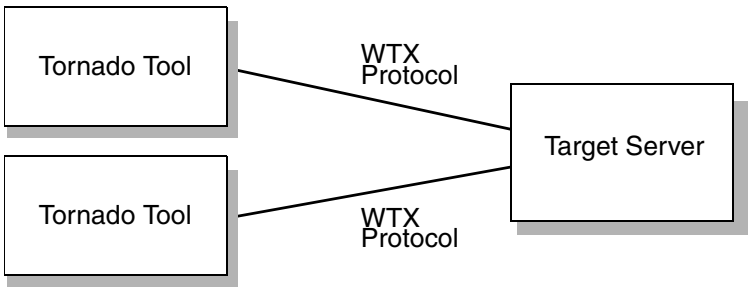
This chapter provides information about the WTX protocol and how to use it to enhance or build Workbench tools.

The Workbench tools communicate with the target server through a protocol known as the *Wind River Tool eXchange* protocol (WTX). The WTX protocol is a set of requests that are issued by Workbench tools running on the development host (for example, a shell or a debugger) to debug and monitor a real-time target system.

B.2 Protocol Overview

The WTX protocol permits a number of tools running on a host system to communicate with one or more host-based target servers, in order to operate on targets running VxWorks (see Figure B-1). Target servers can also connect to the VxWorks simulator. A target server can connect to only one target, but a tool can connect to more than one target server at a time, allowing multi-processor application development.

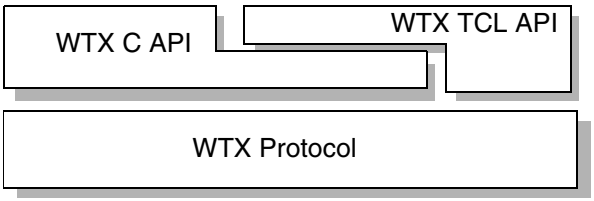
Figure B-1 **WTX Protocol Environment**



B.2.1 Language Support

The WTX protocol is usually accessed by Tcl commands or C functions rather than directly (see Figure B-2). Introductory material for protocol and language bindings is presented in *B.4 WTX Tcl API*, p.131 and in *B.5 WTX C API*, p.147. Workbench provides a Tcl interpreter and page 131 offers many interactive examples. You may want to work through the Tcl examples to become familiar with WTX even if you plan to program primarily in a compiled C environment.

Figure B-2 **WTX APIs**

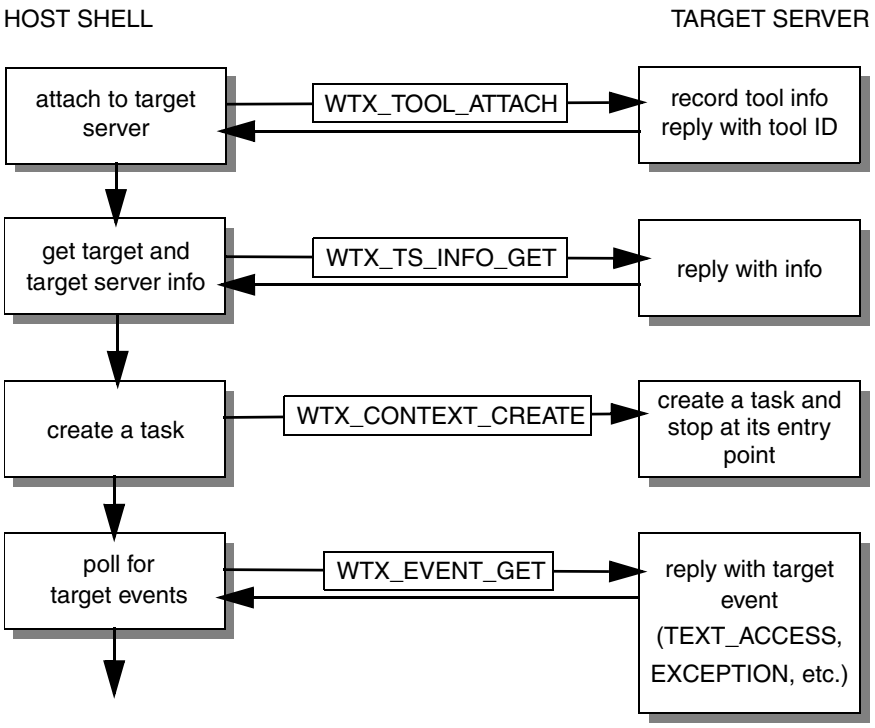


The reference entries give detailed descriptions of WTX services, error messages, and API parameters.

B.2.2 WTX Protocol Usage

The WTX protocol provides the interface between the tools and the target server. A tool requests information or an action of the target server, WTX conveys that request, and the target server makes an appropriate response. Figure B-3 is a schematic representation of a series of interactions between the Host Shell (a tool) and the target server, showing how WTX commands fit into the process.

Figure B-3 **WTX Protocol Links Tool to Target**



B.2.3 WTX Message Format

All WTX messages are defined in *installDir*/**workbench-2.***x*/**foundation/4.0.5/include/wtxmsg.h**. Their names are prefixed with "**WTX_MSG_**". They all contain a **WTX_CORE** structure (shown below) which carries the identifier of the tool making the request and an **errCode** field. The **errCode** contains **OK** (0) when the call succeeds or an error status code

(a value other than 0) when the call fails. The error status corresponds to the **errno** set by the server. The error status values are defined in *installDir*/**workbench-2.***x*/**foundation/4.0.5/include/wtxerr.h**.

```
typedef struct wtx_core                 /* WTX message core           */
    {
    UINT32           objId;             /* identifier                 */
    WTX_ERROR_T      errCode;           /* service error code         */
    UINT32           protVersion;       /* WTX protocol version       */
    WTX_TGT_ADDR_T   defaultPd;         /* derfault protection domain */
    UINT32           osErrCode;         /* OS specific error code     */
    } WTX_CORE;
```

B.2.4 **Debugging WTX Tools**

The target server option **-Wd** *filename* is designed to simplify debugging WTX tools. This option causes all the WTX messages exchanged between the connected tools and the target server to be written to the specified file. A sample file is as follows:

```
User Name                : chrisc
Started                  : Fri Oct  1 11:45:26 2004
Target Server Name       : vxsim@raptor
Target Name              : vxsim
Target Server Options    : tgtsvr.ex -V vxsim -Wd /tmp/wtx.log -B wdbpipe
Host                     : SunOS raptor 5.7 Generic_106541-23 sun4u

11:45:52.451       Request (Thread 0xd) WTX_TS_INFO_GET
        WTX_MSG_TOOL_ID
           WTX_CORE
              objId             0xf5a98
              errCode           0
              protVersion       4
              default PD        0x0
              osErrCode         0x0

11:45:52.452      Reply (Thread 0xd)    status          OK
        WTX_MSG_TS_INFO
           WTX_CORE
              objId             0xf5a98
              errCode           0
              protVersion       0
              default PD        0x0
              osErrCode         0x0
           WTX_TGT_LINK_DESC
              name              WDB Agent across named pipe
              type              255
              speed             0
           WTX_TGT_INFO
              WTX_AGENT_INFO
                 agentVersion   4.0
                 mtu            1372
```

```
      mode            AGENT_MODE_TASK|AGENT_MODE_EXTERN
   WTX_RT_INFO
      rtName          VxWorks
      rtVersion       6.0 - Beta
      cpuType         61
      hasCoprocessor  0x1000000
      hasWriteProtect         0x1
      pageSize        0x2000
      endian          1234
      bspName         SunOS 5.7 [sun4u]
      bspShortName    Unknown
      toolName        diab
      bootline        /wind/river/target/config/solaris/vxWorks
      memBase         0x60000000
      memSize         0x1f00000
      numRegions      0
      memRegion       Unknown
      hostPoolBase    0x601af468
      hostPoolSize    0x1d50b9
      numInts                 1
      WTX_VXWORKS_INFO
         PD list addr         0x0
WTX_TOOL_DESC
   id                 0xf5a98
   toolName           wtxtcl
   toolArgv           Unknown
   toolVersion        Unknown
   userName           chrisc@rance
   pReserved          Unknown
   next               Unknown
version               4.0
userName              chrisc
startTime             Fri Oct  1 11:45:26 2004

accessTime            Fri Oct  1 11:45:26 2004

lockMsg               available
pReserved             Unknown
```

B.3 **WTX Facilities**

The WTX protocol provides requests that serve the following functions:

- managing sessions and logging level
- supporting symbolic debugging in system or task mode
- attaching to a target server
- accessing target memory

- supporting disassembly requests
- managing object modules
- managing symbols
- managing contexts
- supporting virtual I/O
- managing events
- supporting Gopher

This section summarizes each of these functions.

B.3.1 Session Management

WTX provides a set of requests to attach tools to or detach tools from the target server. These requests start and terminate communication with the target server. Requests are also available to lock and unlock other users' access to the target server. WTX and WDB messages can also be logged to files on user demand.

B.3.2 Symbolic Debugging Support

The protocol provides a standard set of symbolic debugging facilities. WTX supports the following primitives:

- continue
- suspend
- single step
- set breakpoint
- set hardware breakpoint (if architecture supports)
- delete breakpoint
- get list of breakpoints

These primitives form the basis of more complicated debugging facilities such as "continue to return," or "step over." Such commands are provided by the Workbench debugger (see the *Wind River* Workbench *User's Guide: Debugger*).

Those commands can be used in task mode (debugging a task) or in system mode (the kernel itself is suspended).

B.3.3 Attaching to a Target Server

Binding to a target server is mediated by the Wind River registry, also known as the WTX registry (**wtxregd**), which provides and maintains a database of all

executing target servers and their remote procedure call (RPC) IDs. WTX service calls exist to look up target server information kept in this database.

There is also a WTX request to attach a tool to a target server. The act of attaching allows the tool to provide information about itself. In addition, attached tools can query the target server for a list of other attached tools.

B.3.4 Target Memory Access

WTX-based tools can read, write, move, and otherwise manipulate target memory. **WTX_MEM_ALLOC** and **WTX_MEM_FREE** operate on the portion of target memory managed by the target server. All other WTX memory requests operate on any valid memory location on the target.

WTX_MEM_READ can be used to transfer data from the target to the host. There are no restrictions on the size of a transfer, but be aware that the time required to fulfill a large request depends on the speed of the host-target link. When transferring complex data structures, use **WTX_GOPHER_EVAL**, which is discussed in *B.3.11 Gopher Support*, p.121.

B.3.5 Target Memory Disassembly

WTX-based tools can request the disassembly of a target memory area with **WTX_MEM_DISASSEMBLE**, which can operate on any valid memory location on the target. The output is a string representing the assembly instructions stored in the given area. The disassembly is done by the target server using the CPU vendor's instruction mnemonics. The output of this facility matches the CPU vendor's documentation, which the disassembly produced by GDB may not.

B.3.6 Object-Module Management

The object-module loader allows the target server to load and unload relocatable or fully linked Elf object modules to the target. Modules may be loaded from any tool. The only supported object-module format (OMF) is ELF.

The load operation can be synchronous (the tool waits for completion) or asynchronous (the tool checks for load completion, but can continue with other operations, including cancelling the submitted load request).

The loader adds all symbols defined by an object module to a symbol table managed by the target server. There is only one symbol table that is shared by all tools; thus a module loaded from one tool is visible from another.

The loader fully supports C++ modules and the symbol table records all symbols in their mangled form.

If some symbols are undefined during the link stage, the object module is not rejected. A partial link is performed, and the code can be utilized as long as all undefined references are avoided. **WTX_OBJ_MODULE_LOAD** returns a list of undefined symbols to inform the tool of all missing references.

The target server retains module information and WTX has service calls to get information on segment addresses, size, and so on.

B.3.7 Symbol Management

The target server provides symbol-table management for the target, but restricted to the kernel only. The target server does not handle RTP symbols. The symbol table holds all the target symbols, and service calls exist for adding, deleting, and looking up symbols. The symbol table is hashed for higher performance. There is only one symbol table; thus every tool sees all symbols.

B.3.8 Context Management

The protocol provides services to create and destroy tasks and RTPs on the target system. It also provides services to set and get the register values for a given task.

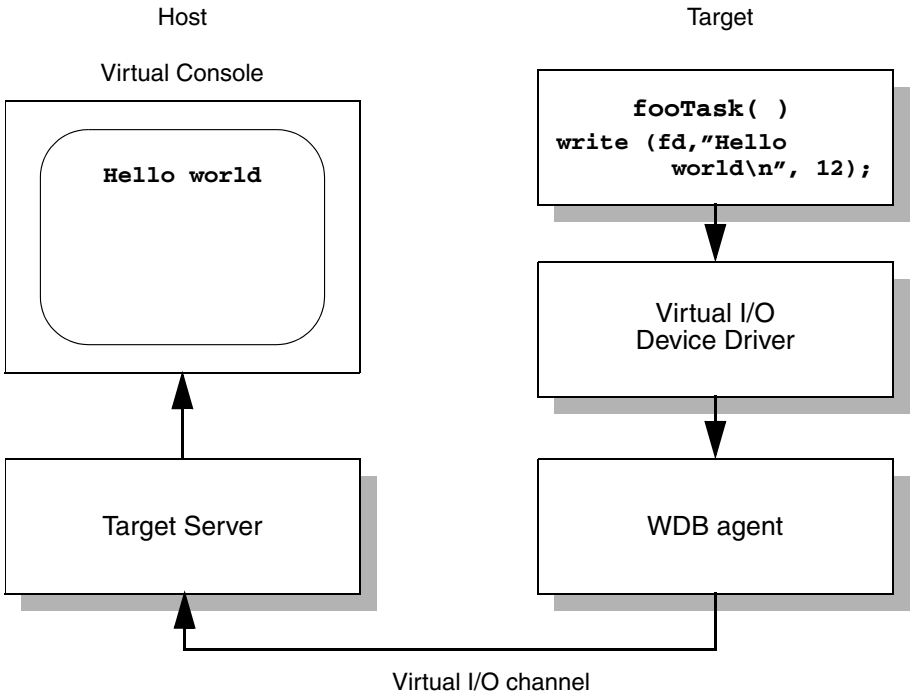
B.3.9 Virtual I/O

Because I/O requirements during development may exceed the facilities available to the target, WTX provides virtual input and output (VIO) for the target. On the host, channels can be associated with a variety of devices (displays, files, streams, and so on). These act as input and output devices associated with the target.

Virtual I/O requires that the target configuration include the WDB agent and a virtual I/O device driver. Once installed, the virtual I/O driver provides standard I/O system access to the virtual I/O facility.

In Figure B-4, a task on the target writes a buffer containing “Hello world” to a previously opened virtual I/O device. The buffer is transferred to the target server by the virtual I/O device driver and displayed in a host window.

Figure B-4 **Virtual I/O example**



A target task creates a virtual I/O channel by calling **open( )** on a virtual I/O device. A bi-directional virtual I/O channel referenced by a unique number is attached to the file descriptor returned by **open( )**. This unique VIO channel number is then used during all **read( )** and **write( )** transactions mediated by the target server. The target server manages redirection of the VIO data to the appropriate device or tool.

The Target Server File System (TSFS) is based on Virtual IO. For more information on TSFS, see the *VxWorks Kernel Programmer’s Guide: Local File Systems* or the *VxWorks Application Programmer’s Guide: Local File Systems*.

B.3.10 Event Management

Host tools must be notified of events occurring on the target and must know about certain actions performed by other tools. WTX meets this requirement by providing a register of events. Tools register to receive notification of particular events or event types. When an event occurs, it generates an event string which is sent to the target server. The target server forwards the string to the tools that have registered to receive notification of that event or event type.

Event strings have the following syntax:

evtName [*param1*] [*param2*] [*param3*] ... [*paramN*]

The parameters are optional and can be either a hexadecimal value (h) or a string (s). The maximum size of an event string is **MAX_TOOL_EVENT_SIZE** (300 characters).

For a comparison of Wind River Workbench 2.5 event parameters compared to earlier versions, see Table 4-8, "Comparison of Events for WTX 2/3/4," on page 37

When it registers for an event, a tool gives the target server a regular expression that matches the event string of the event the tool wants to receive. Tools can also send event strings; this allows tools attached to the same target to coordinate their activities and to exchange information or Tcl programs.

A tool can also choose not to receive any events or a subset of events. This is done by sending **WTX_UNREGISTER_FOR_EVENTS**. The parameter given to this command is a regular expression. Events matching this regular expression are not sent to the tool.

Event-String Examples

- For a breakpoint occurring at address 0x4002344 in task 0x4030090, when **WTX_CONTEXT_TASK** is defined as 0x3 in *installDir*/**vxworks-6.***x*/**host/include/wtxtypes.h**, the event string is:

```
TEXT_ACCESS 0x3 0x4030090 0x4002344 0x4008008 0x4008608
```

- For a watchpoint occurring at address 0x4040000 in task 0x4030090 with pc equal 0x4002344, the event string is:

```
DATA_ACCESS 0x4030090 0x3 0x4002344 0x4008008 0x4008608 0x4040000
```

- For a symbol **newSymbol** of type text, added with value 0x4002344, the event string is:

```
SYM_ADD newSymbol 0x4002344 text
```

- For a divide-by-zero exception in task context 0x4002344, with exception stack frame at address 0x212900 (assuming that 0x10 is the exception vector associated with zero divide), the event string is:

```
EXCEPTION 0x3 0x4002344 0x10 0x212900
```

Asynchronous notification (C API only)

The default method of handling events is for tools to poll the target server. Alternatively, your tool can register an event handler with the target server. In this case the target server calls the event handler when it receives an event. The following example shows the pieces of the event handler which are required to use the asynchronous event notification feature:

```
#include "wtx.h"
...
/* handle events */

LOCAL void eventHandler
    (
    WTX_EVENT_DESC *    event
    )
    {
    /* print the received event */

    if (event->event != NULL)
        printf ("Receeived the event : %s\n", event->event);
    }
...
    /* register to receive events */

    if (wtxAsyncNotifyEnable (wtxh, (FUNCPTR) eventHandler) != WTX_OK)
        {
        /* handle error here */
...
    }
...
```

B.3.11 Gopher Support

A unique facility of WTX is its support for a compact interpreted language called Gopher. Any tool can use small Gopher scripts to request that target data structures of arbitrary complexity be retrieved from the target.

The Gopher interpreter resides in the target agent, so the target does not need to store dedicated routines to perform each unique request. This greatly reduces the amount of target memory required, while giving users a powerful tool to access target data. Because of its power and efficiency, Gopher forms the basis of all Workbench show routines for system objects.

How the Gopher Language Works

The Gopher execution environment is an abstract machine consisting of two objects, the pointer and the tape. The pointer is an address in the target's address space, and the tape is a one-way write-only area where Gopher results are accumulated. When the Gopher script terminates, the contents of the tape are returned to the host tool.

Table B-1 shows the elements of the Gopher language and how they affect the pointer and the tape.

Table B-1 **Gopher Programming Language Elements**

Item	Action
integer	Sets the pointer to the value *integer*.
+ *integer*	Increments the pointer by *integer*.
- *integer*	Decrements the pointer by *integer*.
!	Writes the pointer to the tape.
|	Writes a separator character to the tape.
*	Replaces the pointer with the value pointed to by the pointer.
@	Writes the value pointed to by the pointer on the tape, and advances the pointer (if no other flag is given, the pointer is advanced by 32 bits).
@b	Writes the value pointed to by the pointer to the tape, and advances the pointer by 8 bits.
@w	Writes the value pointed to by the pointer to the tape, and advances the pointer by 16 bits.

Table B-1 **Gopher Programming Language Elements** (cont'd)

Item	Action
@l	Writes the value pointed to by the pointer to the tape, and advances the pointer by 32 bits.
$	Writes the string pointed to by the pointer on the tape.
{*script*}	Executes the Gopher script inside the braces repeatedly while the pointer is not **NULL**. The string inside the braces works with a local copy of the pointer, maintaining the original, external pointer value.
<*script*>	Executes the Gopher script inside the angle brackets once using a local copy of the pointer.
(*n script*)	Executes the Gopher script inside the parentheses *n* times, using a local copy of the pointer.
'*script*'	Executes the Gopher script inside the quotes repeatedly while the value pointed to by the pointer is not zero. The string inside the quotes uses a local copy of the pointer, maintaining the original, external pointer value. This is useful for scanning an array with a sentinel value at the end.
[*script*]	Executes the Gopher script inside square brackets repeatedly while the pointer is not equal to its initial value—before first execution of the script. The string inside the brackets uses a local copy of the pointer, maintaining the original, external pointer value. This is useful for scanning a circular linked list.
/!	Performs a **ntoh( )** operation before writing the pointer to the tape.
/*	Performs a **ntoh( )** operation before dereferencing the pointer. This is useful for dereferencing a pointer in shared memory (shared memory is always big endian).
/@	Performs a **ntoh( )** operation before writing data pointed to by the current pointer.
\!	Performs a **hton( )** operation before writing the pointer to the tape.

Table B-1 **Gopher Programming Language Elements** (cont'd)

Item	Action
@l	Writes the value pointed to by the pointer to the tape, and advances the pointer by 32 bits.
$	Writes the string pointed to by the pointer on the tape.
{*script*}	Executes the Gopher script inside the braces repeatedly while the pointer is not **NULL**. The string inside the braces works with a local copy of the pointer, maintaining the original, external pointer value.
<*script*>	Executes the Gopher script inside the angle brackets once using a local copy of the pointer.
(*n script***)**	Executes the Gopher script inside the parentheses *n* times, using a local copy of the pointer.
'*script*'	Executes the Gopher script inside the quotes repeatedly while the value pointed to by the pointer is not zero. The string inside the quotes uses a local copy of the pointer, maintaining the original, external pointer value. This is useful for scanning an array with a sentinel value at the end.
[*script*]	Executes the Gopher script inside square brackets repeatedly while the pointer is not equal to its initial value—before first execution of the script. The string inside the brackets uses a local copy of the pointer, maintaining the original, external pointer value. This is useful for scanning a circular linked list.
/!	Performs a **ntoh( )** operation before writing the pointer to the tape.
/*	Performs a **ntoh( )** operation before dereferencing the pointer. This is useful for dereferencing a pointer in shared memory (shared memory is always big endian).
/@	Performs a **ntoh( )** operation before writing data pointed to by the current pointer.
\!	Performs a **hton( )** operation before writing the pointer to the tape.

Table B-1 **Gopher Programming Language Elements** (cont'd)

Item	Action
*	Performs a **hton( )** operation before dereferencing the pointer.
\@	Performs a **hton( )** operation before writing data pointed to by the current pointer.
_	The underscore character is used to separate commands in the Gopher string. This character is ignored by the interpreter.

A Gopher Script for Sample Data Structures

This section describes a Gopher script that traverses and collects information from the following two sample data structures:

```
typedef struct _node
    {
    UINT32            number;         /* number of node */
    struct _node *    pNext;          /* single linked list */
    struct _other *   pOther;         /* pointer to an OTHER */
    char *            name;           /* name of node */
    UINT32            value;          /* value of node */
    } NODE;

typedef struct _other
    {
    UINT32            x1;             /* X Factor One */
    UINT32            x2;             /* X Factor Two */
    } OTHER;
```

Suppose you want a list of the numbers and names of each node in a list anchored at **nodeList**, whose elements are defined by **struct _node**. Initialize the Gopher pointer to the value of **nodeList**, *nodeAddr*, and dereference the pointer. Then, while the pointer is not **NULL**, reach into the **NODE** structure and write the number and name elements to the tape.

Referring to the table of Gopher elements, create the following script:

```
nodeAddr * { @ < +8 *$ > * }
```

First, initialize the pointer to the value of the symbol **nodeList** and then dereference it (*****). You now have a pointer to the first element. While the pointer is not **NULL** ({), write the **number** field to the tape and advance the pointer (**@**). Saving the pointer position, enter the subroutine (**<**). Add eight to the value of the pointer (**+8**), skipping over the **pNext** and **pOther** fields. Now the pointer points to the **name** field. Dereference the pointer at the first character of the **name** string

(*) and write the string to the tape (**$**). Exit the subroutine (**>**), which restores the previous value of the pointer (pointing to **pNext**). Dereference the pointer (*****) at the next element in the list, or at **NULL**. If the field is not **NULL**, the subprogram **{...}** repeats. At the end, the Gopher tape contains two entries for each node in the list, one numeric value and one string value.

There are many ways to write a Gopher script. Each of the following is equivalent to the one described above:

```
nodeAddr * { @ +8 <*$> -8 * }
nodeAddr * { <@> <+12 *$> +4 * }
```

Suppose you also want to find the values of **x1** and **x2** for a particular node. If the pointer points to the **pOther** member of a node and **pOther** is not **NULL**, this Gopher fragment writes the values of **x1** and **x2** to the tape:

```
* {@@ 0}
```

The first ***** replaces the pointer with the value of **pOther**; the fragment in braces executes only if the pointer is not **NULL**. The **@@** fragment writes **x1** and **x2** to the tape, and the final **0** sets the pointer to **NULL** so that the **{...}** loop exits. Guarding this fragment in angle brackets (so that it executes with a local copy of the pointer), adding **+4** to move the pointer to **pOther**, and inserting this fragment into the example string, you have:

```
nodeAddr * { @ < +8 *$ > < +4 * {@@ 0} > * }
```

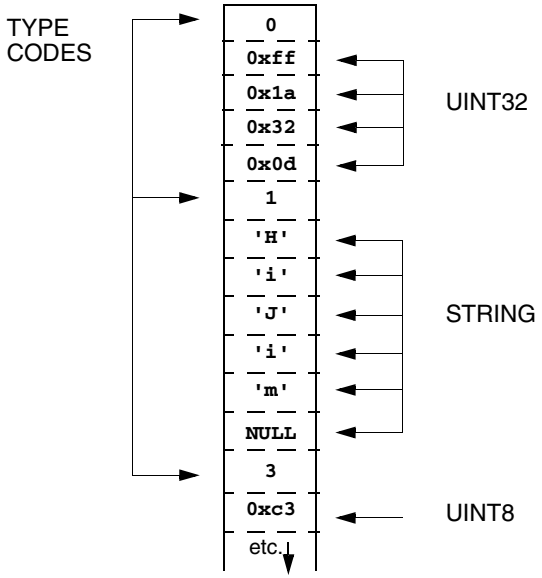
The Gopher Result Tape

The Gopher result tape is a byte-packed data stream. The data is a series of pairs, each consisting of a type code and its associated data. The type codes are defined in the file *installDir***/vxworks6.0/share/src/agents/wdb/wdb.h** as follows:

GOPHER_UINT32	0
GOPHER_STRING	1
GOPHER_UINT16	2
GOPHER_UINT8	3
GOPHER_SEPARATOR	7
GOPHER_UINT64	

The tape is formatted as shown in Figure B-5.

Figure B-5 **Gopher Result Tape**



Sending the Gopher Script to the Target Agent

The Gopher interpreter on the target agent interprets and executes the Gopher script. Any tool or application can send a Gopher script to the target agent by using a WTX command. Both Tcl and C language APIs provide appropriate commands.

Tcl

The Gopher script assumes that *nodeAddr* is a pointer to the first node. To establish this pointer using Tcl, query the symbol table for the value of **nodeList**, and then send the Gopher script to the target agent using the Tcl procedure **wtxGopherEval**:

```
set nodeList [lindex [wtxSymFind -name nodeList] 1]
wtxGopherEval $nodeList * { @ < +8 *$ > < +4 * {@@ 0} > * }
```

Tcl interprets the Gopher result tape and returns a Tcl list consisting of just the data values. For this example, the Tcl API returns the following list:

```
{ name1 value1 x11 x21 name2 value2 ... x1n x2n }
```

For further information, see the reference entries.

C Language

The procedure for sending the Gopher script using C involves a similar process. First use the C routine **wtxSymFind( )** to query the symbol table for the value of **nodeList**. Convert the value to a string, and then send this string, concatenated with the Gopher script, to the target agent using **wtxGopherEval( )**. C returns the byte-packed data stream without formatting it. Example B-1 shows how to submit the Gopher script and format the result tape. For further information, see the reference entries.

Example B-1 **Evaluating a Gopher Script Using the C API**

```
/* find the starting address */

WTX_SYM_FIND_CRITERIA Criteria = {0};    /* criteria structure  */
Criteria.findName     = "nodeList";
Criteria.options      = WTX_SYM_FIND_BY_NAME;
Criteria.nSymbols     = 1;

if ((pSymbol = wtxSymFind (hWtx, &symbolCriteria)) == NULL)
    {
    printf ("Could not find symbol\n");
    return;
    }

/* convert this value to a string */

sprintf (startAddress, "0x%p", pSymbol->value);

/* free the symbol room */

wtxResultFree (hWtx, pSymbol);

/* integrate this string into the Gopher script and execute it*/

sprintf (gopherCmd, "%s * { @ < +8 *$ > < +4 * {@@ 0} > * }",
        startAddress);

if ((pResult = wtxGopherEval (hWtx, 0, gopherCmd)) == NULL)
    {
    printf ("gopher error: %s\n", wtxErrToMsg (hWtx,
            wtxErrGet(hWtx)));
    return;
    }

/* format the result tape */

STATUS formatResult
    (
    WTX_GOPHER_TAPE   pTapeResult,
    char *            pFinalResult
    )
```

```
{
int    targetByteOrder = wtxTargetEndianGet (hWtx);
int    needSwap        = targetByteOrder != HOST_BYTE_ORDER;
int    bufIx;

bufIx = 0;
while (bufIx < pTapeResult->len)
    {
    char valbuf [20];
    UINT8 type          = pTapeResult->data [bufIx++];
    unsigned char *bufp = pTapeResult->data + bufIx;

    switch (type)
        {
        case GOPHER_UINT32:
            {
            UINT32 rawVal = UNPACK_32 (bufp);
            sprintf (valbuf, "%#x", SWAB_32_IF (needSwap, rawVal));
            strcat (pFinalResult, valbuf);
            bufIx += sizeof (UINT32);
            break;
            }
        case GOPHER_UINT16:
            {
            UINT16 rawVal = UNPACK_16 (bufp);
            sprintf (valbuf, "%#x", SWAB_16_IF (needSwap, rawVal));
            strcat (pFinalResult, valbuf);
            bufIx += sizeof (UINT16);
            break;
            }
        case GOPHER_UINT8:
            {
            UINT8 rawVal = *bufp;
            sprintf (valbuf, "%#x", rawVal);
            strcat (pFinalResult, valbuf);
            bufIx += sizeof (UINT8);
            break;
            }
        case GOPHER_FLOAT32:
            {
            union
                {
                UINT32      i;
                float       f;
                } u;

            UINT32 rawVal = UNPACK_32 (bufp);
            u.i = SWAB_32_IF (needSwap, rawVal);
            sprintf (valbuf, "%.8g", u.f);
            strcat (pFinalResult, valbuf);
            bufIx += sizeof (float);
            break;
            }
        case GOPHER_FLOAT80:
```

```
    /*
     * This one is trouble. We aren't able to represent
     * 80-bit floats on the host, so we just take the
     * upper 64 bits and convert to double.
     */

    if (targetByteOrder == LITTLE_ENDIAN)
        {
        /* skip over least significant 16 bits. */

        bufIx += 2;
        bufp += 2;
        }

    /* FALL THROUGH */

case GOPHER_FLOAT64:
    {
    union
        {
        UINT32      i [2];
        double      d;
        } u;

    if (needSwap)
        {
        u.i [1] = SWAB_32 (UNPACK_32 (bufp));
        u.i [0] = SWAB_32 (UNPACK_32 (bufp+4));
        }
    else
        {
        u.i [0] = UNPACK_32 (bufp);
        u.i [1] = UNPACK_32 (bufp+4);
        }

    sprintf (valbuf, "%.16g", u.d);
    strcat (pFinalResult, valbuf);
    bufIx += sizeof (double);

    /*
     * If target is big endian and we fell through the 80-bit
     * case, we need to advance over the least-significant
     * 16 bits which we have no way of converting.
     */

    if (type == GOPHER_FLOAT80 && targetByteOrder == BIG_ENDIAN)
        bufIx += 2;

    break;
    }
case GOPHER_STRING:
    {
    strcat (pFinalResult, pTapeResult->data + bufIx);
    bufIx += strlen (pTapeResult->data + bufIx) + 1;
```

```
                }
            }
        }
return OK;
  }
```

B.4 WTX Tcl API

This section introduces the WTX Tcl API and gives a series of interactive examples for using Tcl to communicate with a target server. These examples not only explain the WTX Tcl API, but also demonstrate how the WTX Protocol works. For this reason, users who plan to work primarily with the WTX C API (see *B.5 WTX C API*, p.147) may benefit from working through these Tcl examples.

B.4.1 Description

The WTX Tcl API provides a binding of the WTX protocol to the Tcl language. This allows you to write Tcl scripts that interact with the WTX environment. Every WTX protocol request is available to the Tcl interface. Workbench provides a standalone Tcl interpreter, as well as an object library that allows the WTX-Tcl binding to be included in other applications. The WTX Tcl API is based on Tcl version 8.4.

For each WTX protocol request (for example, **WTX_MEM_READ**), there is a corresponding Tcl command (**wtxMemRead**). The names of all WTX Tcl API commands are derived from the protocol request names according to the procedure described in the *VxWorks Hardware Interface Validation Guide: Coding Conventions*; in other words, underscores are removed and all words but the first are capitalized.

Each WTX Tcl command has an reference entry. For information on the online reference material, see your product *Getting Started*. Most WTX Tcl API commands also return a brief syntax message if they are invoked without arguments.

B.4.2 Starting a wtxtcl Session

One of the interactive tools in Workbench is called **wtxtcl**, a standalone Tcl interpreter for the WTX extensions. **wtxtcl** is an ideal environment for

experimenting with and testing the WTX interface because it contains only the WTX extensions.

When you call **wtxtcl**, Workbench responds with the **wtxtcl** prompt; you may begin typing Tcl commands immediately. Invoking **wtxtcl** with an argument identifies a file of Tcl commands to execute.

B.4.3 Attaching to a Target Server

Before the majority of **wtxtcl** commands can be used, you must attach to an active target server. The Wind River registry maintains a list of available target servers for a site or workgroup. Use **wtxInfoQ** to determine what servers are active by querying the contents of the Wind River registry.

```
wtxtcl> wtxInfoQ
{vxsim@van-chauve tgtsvr rpc/van-chauve/147.11.80.14/570425345/4/tcp/604}
{wtxregd@van-chauve registry rpc/van-chauve/147.11.80.14/570425344/4/tcp}
```

For example, to see only target servers running on host **van-chauve**, enter the following:

```
wtxtcl> wtxInfoQ van tgtsvr
{vxsim@van-chauve tgtsvr rpc/van-chauve/147.11.80.14/570425345/4/tcp/604}
wtxtcl>
```

Each of the three arguments is a regular expression that qualifies the corresponding field in the output, and an entry is returned only if it matches all regular expressions provided.

After learning the name of a target server, you must attach to it in order to carry out further WTX transactions:

```
wtxtcl> wtxToolAttach t80-202@aven myTool
t80-202@aven
```

The string **myTool** is a name you choose to represent your Tcl session. The target server records it and reports this name to any tool that requests a list of attached tools. You can see **myTool** in the list of attached tools on the Workbench target manager if your target server is selected.

The value returned by this command, **vxsim@van-chauve**, is called the WTX Tcl handle, and can be used to refer to this connection when multiple connections are made in a single **wtxtcl** session.

Once a target server is attached, you may use any WTX Tcl command. The examples in the remainder of this section assume that a target server is attached.

B.4.4 Obtaining Target Server Information

You can learn about the architecture and about other tools connected to the target server with **wtxTsInfoGet**:

```
wtxtcl> wtxTsInfoGet
{WDB Agent across named pipe} {255 0} {100 1 1 65536 4321 0 0 0}  {0 6.0}
{Windows 5.1 Service Pack 1}
{host:Z:\wind\river\target\config\simpc\vxWorks} {0x10000000 32505856}
{0x1016dc78 1937976} chrisc 1098435751 1098435751 available 0 4.0 4.0
{0xbddfa0 myTool {} {} chrisc@van-chauve}
wtxtcl>
```

For a detailed explanation of the fields in this structure, see the reference entry.

B.4.5 Working with Memory

To read a block of memory from the target, use **wtxMemRead**, specifying the base address and the size of the block:

```
wtxtcl> wtxMemRead 0x20000 256
mblk0
wtxtcl>
```

The command returns a *memory block handle*. This short string is used by a variety of Tcl commands that work with memory. These commands are provided to avoid the overhead of converting target memory to a string representation, and to make manipulating memory easier. It is often convenient to save memory block handles in variables:

```
wtxtcl> set mb [wtxMemRead 0x20000 256]
mblk1
wtxtcl>
```

To print the memory block, you can use **memBlockGet** with the handle as a parameter:

```
wtxtcl> memBlockGet $mb
0x00 0x10 0x00 0x00 0x80 0x08...
```

The target server manages a block of memory on the target. You can allocate such a block from the host using **wtxMemAlloc**. You can write to allocated memory, and read it back to verify that the write took place:

```
wtxtcl> set mem [wtxMemAlloc 256]
```

```
0x5897d0
wtxtcl> wtxMemWrite $mb $mem
0
wtxtcl> set ver [wtxMemRead $mem 256]
mblk2
wtxtcl> memBlockGet $ver
0x03 0x00 0x00 0x39 0x82 0x10 0x60...
```

You can also work with parts of memory blocks, free blocks, write memory blocks to binary files or pipes, fill and checksum blocks of memory, and so on. For more information, see the reference entries.

B.4.6 Disassembling Memory

To disassemble a block of memory from the target, use **wtxMemDisassemble**, specifying the base address and the number of instructions (default 10). The following example is from a PowerPC target:

```
"wtxtcl> wtxSymListGet -name "^printf$"
{printf 0x10031860 0x5 0 0 ""}
wtxtcl> wtxMemDisassemble -opcodes 0x10031860
{printf} 1 {} {55} {PUSH         EBP}
{} 2 {} {89 e5} {MOV          EBP, ESP}
{} 3 {} {83 ec 08} {SUB          ESP, 8}
{} 3 {} {8d 45 0c} {LEA          EAX, [EBP+12]}
{} 3 {} {89 45 f8} {MOV          [EBP-8], EAX}
{} 2 {} {6a 01} {PUSH         1}
{} 5 {} {68 10 19 03 10} {PUSH         0x10031910}
{} 3 {} {ff 75 f8} {PUSH         [EBP-8]}
{} 3 {} {ff 75 08} {PUSH         [EBP+8]}
{} 5 {} {e8 b2 ed ff ff} {CALL         fioFormatV}
wtxtcl>
```

B.4.7 Working with the Symbol Table

The target server symbol table can be queried and modified from **wtxtcl**. The reason for doing this is to learn the address of a routine to call, the entry point for a task, or the address of a variable in order to read its value. Symbols can be looked up by name or value:

```
wtxtcl> wtxSymFind -name taskDelay
taskDelay 0x10115d50 0x5 0 0 ""
wtxtcl>
wtxtcl> wtxSymFind -value 0x101143c0
taskSuspend 0x101143c0 0x5 0 0 ""
wtxtcl>
```

For convenience, you can define a procedure that returns the value member of the symbol list for a given symbol name. Further examples assume the presence of this procedure (**symbol**). In order to use it, define it by typing it as shown:

```
wtxtcl> proc symbol {name} \
{return [lindex [wtxSymFind -name $name] 1]}
wtxtcl> symbol taskDelay
0x10115d50
wtxtcl>
```

To query a list of symbols using a regular expression, use **wtxSymListGet**:

```
wtxtcl> wtxSymListGet -name ^taskD.*
{taskDestroy 0x10115730 0x5 0 0 ""}
{taskDeleteTable 0x1016ced0 0x9 0 0 ""}
{taskDeleteHookShow 0x100975f0 0x5 0 0 ""}
{taskDeleteHookDelete 0x10097240 0x5 0 0 ""}
{taskDeleteHookAdd 0x100971f0 0x5 0 0 ""} {taskDeleteForce
0x10115d10 0x5 0 0 ""} {taskDelete 0x10115d30 0x5 0 0 ""}
{taskDelaySc 0x1011aaf0 0x5 0 0 ""} {taskDelay 0x10115d50 0x5
0 0 ""} {taskDbgUnlock 0x10114ee0 0x5 0 0 ""}
wtxtcl>
```

You can also use **wtxSymListGet** to obtain a list of the symbols found near a given value:

```
wtxtcl> wtxSymListGet -value 0x101143d0 {taskSuspend 0x101143c0 0x5 0 0
""} {taskResume 0x10114720 0x5 0 0 ""}
{taskActivate 0x10114850 0x5 0 0 ""} {taskStop 0x10114a00 0x5
0 0 ""} {taskStopForce 0x10114a20 0x5 0 0 ""} {taskCont
0x10114a40 0x5 0 0 ""} {taskPrioritySet 0x10114ba0 0x5 0 0
""} {taskLock 0x10114d60 0x5 0 0 ""} {taskUnlock 0x10114dd0
0x5 0 0 ""} {taskDbgUnlock 0x10114ee0 0x5 0 0 ""} {taskSafe
0x10114fb0 0x5 0 0 ""} {taskUnsafe 0x101150c0 0x5 0 0 ""}
wtxtcl>
```

B.4.8 Working with Object Modules

The WTX Tcl API provides complete access to the target server's dynamic loading capabilities. The fundamental command is **wtxObjModuleLoad**, which works like the **ld( )** command in the Host Shell. It takes a path to an object module and downloads the module to the target. This command returns a module ID, followed by the addresses of the sections of the relocated module. You can load a module and save the ID in a variable:

```
wtxtcl> set modId [lindex [wtxObjModuleLoad /work/dir/mod.o] 0]
```

If the module contains unresolvable symbols, a list of them is returned with the module ID:

```
wtxtcl> wtxObjModuleLoad z:/wind/river/tutorial/test.o
0xba8748 0x108f0000 0x108f0000 0x10900008
test2
test3
wtxtcl>
```

There are also routines for obtaining information about object modules. **wtxObjModuleList** returns a list of loaded object modules. This list can be directed to input for **wtxObjModuleInfo**, which takes the list as an argument and returns information about all listed modules.

```
wtxtcl> foreach mod [wtxObjModuleList] {puts stdout [wtxObjModuleInfoGet
$mod]}
0xbb21c8 vxWorks 0x1 0 0x64 {0 0x10010000 0x13a848}  {0 0x10150000
0xc408}  {0 0 x1015c408 0x11870}
wtxtcl>
```

B.4.9 Working with Tasks

If the target server is connected to a target running VxWorks, you can create and manipulate VxWorks tasks interactively with Tcl commands. Creating a task interactively from **wtxtcl** is less efficient than using the Tornado shell. The Tornado shell uses Tcl procedures to automate many tasks, including task creation. This chapter treats only those features available directly from **wtxtcl**.

The first step is to spawn a task named **uMyTask** at priority 100 with VxWorks options 0x3 (**VX_UNBREAKABLE | VX_SUPERVISOR_MODE**). The entry point is the **taskDelay( )** routine, whose address comes from **symbol**, which you created in *B.4.7 Working with the Symbol Table*, p.134. The argument 10000 causes the task to delay for 10,000 ticks before exiting. (For more information on command parameters, see the reference entry.)

```
wtxtcl> wtxContextCreate CONTEXT_TASK uMyTask 100 0x3 5000 5000 [symbol
taskDela y] 0 0 10000
0x1047b6c8
wtxtcl>
```

The output, 0x1047b6c8, is the ID of the new task. Note that, on the simulator, running the **task** command gives:

```
uMyTask  10115d50  1047b6c8 100 SUSPEND  10023030 1092ffd8  0  0
```

If you are running a Tornado shell, you can use **i( )** to check that it has been created. The task has not begun running yet; it is created in the suspended state, and must be resumed before it executes:

```
wtxtcl> wtxContextResume CONTEXT_TASK 0x1047b6c8 0
> wtxtcl>
```

Now the task runs, and vanishes after the task delay has expired. To find out what other commands operate on contexts, see the reference entries. To see a list of these commands, type:

```
wtxtcl> info commands wtxContext*
wtxContextDetach wtxContextResume wtxContextKill wtxContextStep
wtxContextStatus Get wtxContextSuspend wtxContextCreate wtxContextCont
wtxContextAttach wtxContex tStop
wtxtcl>
```

B.4.10 Working with Events

The target server receives event strings from the target and from attached tools and directs them to appropriate queues. For a discussion of the WTX event facility, see *B.3.10 Event Management*, p.120. By default, when tools first attach, they receive no event strings. It is up to each tool to register for the events it is interested in and to check its queue periodically.

Registering for Events

Your **wtxtcl** session is not yet registered for any events. In order to receive event strings, you must call **wtxRegisterForEvent**. The argument to this command is a regular expression that serves as the target server filter for assigning events to your tool queue. The following example registers for all events:

```
wtxtcl> wtxRegisterForEvent .*
0
```

The return value 0 indicates that you have successfully registered for all events.

Unregistering for Events

Your **wtxtcl** session is registered for all events. If you don't want to receive some types of events, you must call **wtxUnregisterForEvent**. The argument to this command is a regular expression that serves the target server as a filter for assigning events to your tool queue. The following example unregisters for tool-related events (**TOOL_ATTACH** and **TOOL_DETACH)**:

```
wtxtcl> wtxUnregisterForEvent TOOL.*
0
```

The return value 0 indicates that you have successfully unregistered for tool related events.

Checking the Event Queue

One way to confirm that your **wtxtcl** session is registered for all events is to call **wtxEventGet** to query the target server for events. It returns an event string if any events are queued. The first word in the string is the event type, and the rest are event-specific parameters. For a detailed list of the events and their parameters, see Table 4-8, "Comparison of Events for WTX 2/3/4," on page 37. The reference entry for **wtxEventGet** details their Tcl representation. If the queue is empty, **wtxEventGet** returns the empty string.

Some environments generate many events. In other cases, you may have to query the queue many times before an event is returned. Within the Tornado shell Tcl interpreter, a routine **wtxEventPoll** is available to check the queue regularly. You can use the same building blocks in **wtxtcl** to create a polling routine. The WTX primitive **wtxEventGet**, which asks for one event if it is available, can be combined with the Tcl procedure **msleep**, which puts the invoking process to sleep for a specified number of milliseconds. The following Tcl procedure polls for events with the specified delay. Define this procedure (**eventPoll**) to the interpreter as follows:

```
wtxtcl> proc eventPoll {intvl} {
while {[set event [wtxEventGet]] == ""} {
msleep $intvl
}
return $event
}
```

Generating and Retrieving Events

The example in this section stimulates two events on the target and captures them. The first step is to create a task with entry point **tickGet( )**. Then set two eventpoints on **tickGet( )** and resume the task. When the task hits the first eventpoint it generates an event, which you can retrieve from the queue. When it completes, it generates a second event.

To create the task, use the technique from *B.4.9 Working with Tasks*, p.136, without setting the **VX_UNBREAKABLE** bit in the task options:

```
wtxtcl> set ctxId [wtxContextCreate CONTEXT_TASK tick 100 0x1 \
0 2000 [symbol tickGet] 0 0]
0x116f9588
wtxtcl>
```

You can use the shell to verify that the task has appeared. The task is in the suspended state, and its ID has been saved in the variable **ctxId**.

To add an eventpoint to **tickGet( )**, use **wtxEventpointAdd**. The eventpoint stops the task, which generates an event when it resumes. The parameters of **wtxEventpointAdd** include three WTX enumerated constant types: **EVENT_TYPE**, **CONTEXT_TYPE**, and **ACTION_TYPE**. (For information on specifying these arguments in numeric form, see the reference entry). In Tcl you can specify the parameters symbolically. Use **wtxEnumList** to obtain a list of constant types. Then give **wtxEnumInfo** any of these constants as a parameter to obtain a list of all events of a given type which can be passed to a Tcl command, as in the following example:

```
wtxtcl> wtxEnumList
ACTION_TYPE CONTEXT_TYPE EVENT_TYPE REG_SET_TYPE LOAD_FLAG UNLOAD_FLAG
VIO_CTL_R EQUEST AGENT_MODE TASK_OPTION OBJ_KILL_REQUEST OPEN_FLAG
CONTEXT_STATUS SECTION_ TYPE PD_TYPE SYMBOL_TYPE TGTFS_INODE_TYPE
PD_INFO_OPT
wtxtcl> wtxEnumInfo EVENT_TYPE
WTX_EVENT_CTX_START 0x1
WTX_EVENT_CTX_EXIT 0x2
WTX_EVENT_TEXT_ACCESS 0x3
WTX_EVENT_HW_BP 0x7
WTX_EVENT_TRIGGER 0x16
WTX_EVENT_CTX_DUMP 0x1e
```

Since it is difficult to type these long names at the keyboard, you can query for abbreviations by giving the **-abbrev** flag to **wtxEnumInfo** along with the constant:

```
wtxtcl> wtxEnumInfo -abbrev EVENT_TYPE
WTX_EVENT_CTX_START WTX_EVENT_CTX_START CTX_START cstart 0x1
WTX_EVENT_CTX_EXIT WTX_EVENT_CTX_EXIT CTX_EXIT cexit 0x2
WTX_EVENT_TEXT_ACCESS WTX_EVENT_TEXT_ACCESS TEXT_ACCESS tacc
0x3 WTX_EVENT_HW_BP WTX_EVENT_HW_BP HW_BP hwbp 0x7
WTX_EVENT_TRIGGER WTX_EVENT_TRIGGER TRIGGER trigger 0x16
WTX_EVENT_CTX_DUMP WTX_EVENT_CTX_DUMP CTX_DUMP cdump 0x1e
```

⚠ **CAUTION:** The abbreviations are handy for typing but, if you are writing scripts, it is better practice to use the complete name.

WTX_EVENT_TEXT_ACCESS is an appropriate event type for a traditional code breakpoint. In the list of event type abbreviations, **WTX_EVENT_TEXT_ACCESS** can be abbreviated **tacc**. **WTX_EVENT_CTX_EXIT** generates an event when the task exits, and can be abbreviated **cexit**.

Calling **wtxEnumInfo** with the **-abbrev** flag for **CONTEXT_TYPE** and **ACTION_TYPE** shows that you can use **task** and **notify** to abbreviate **CONTEXT_TASK** and **ACTION_NOTIFY**.

Now you have all the information you need to add two eventpoints to the **tickGet( )** routine, as shown in the following example:

```
wtxtcl> wtxEventpointAdd tacc [symbol tickGet] task $ctxId \
notify|stop 0 0 0
0x1
wtxtcl> wtxEventpointAdd cexit [symbol tickGet] task $ctxId \
notify 0 0 0
0x2
```

Before resuming the task, register to receive all events and flush the event queue. The following example creates a procedure, **listEvents**, which flushes the queue and writes all events:

```
wtxtcl> wtxRegisterForEvent .*
0
wtxtcl> proc listEvents {} {while {[set event [wtxEventGet]] != ""}\
{puts $event}}
wtxtcl> listEvents
TOOL_DETACH windsh 0x44ce50
TOOL_ATTACH windsh 0x44ff90
CALL_RETURN 0x4ae2720 0x4 0xd0003
...
```

Now start the task and check for events:

```
wtxtcl> wtxContextResume task $ctxId
0
wtxtcl> listEvents
TEXT_ACCESS 0x116f9588 0x3 0x10118260 0 0x109affd4
wtxtcl>
```

When you see the **TEXT_ACCESS** line of output, the breakpoint event has arrived. The following message appears in the simulator kernel shell:

```
->
Break at 0x10118260: _tickGet Task: 0x116f9588 (tick)
```

The task remains stopped at the breakpoint until you continue it:

```
wtxtcl> wtxContextCont task $ctxId
0
```

The task now completes, generating another event, which you can retrieve:

```
wtxtcl> listEvents
CTX_EXIT 0x3 0x116f9588 0x2b9f7 0
```

B.4.11 Working with Virtual I/O

Virtual input and output is comprised of strings that are placed into memory blocks by **wtxtcl.** WTX provides commands for manipulating the memory blocks. WTX uses an event to notify the tool that virtual output is available.

Virtual Output

This example assumes a host shell for configuring the virtual I/O and a target whose configuration includes virtual I/O (as is the case with the target images supplied with Workbench). For more information on configuring VxWorks and the target, see the *VxWorks Kernel Programmer's Guide: Kernel*.

First, open a virtual I/O channel and redirect the target's standard input, output, and error streams to this channel. To start the host shell on the target server, select the WindSh menu item in the Workbench target manager (right-click on a target server) or start the shell from the Windows command prompt or a UNIX shell prompt:

```
% windsh vxsim@van-chauve
...
```

In WindSh, unset the **SH_GET_TASK_IO** configuration parameter to avoid any conflict with the WindSh local redirection (see *Wind River Workbench Command-Line User's Guide: Host Shell* for more details). Open a virtual I/O device and reroute standard input, output, and error to it. Then, using the shell, print a string to standard output. The string is sent to the attached tools in the form of an event which you can examine in **wtxtcl**. The commands and output are shown in the following example:

```
-> ?shConfig SH_GET_TASK_IO off
-> vioFd = open ("/vio/1", 2)
new symbol "vioFd" added to symbol table.
vioFd = 0x39d1a0: value = 7 = 0x7
-> ioGlobalStdSet (0, vioFd)
value = 0 = 0x0
-> ioGlobalStdSet (1, vioFd)
value = 1 = 0x1
-> ioGlobalStdSet (2, vioFd)
value = 2 = 0x2
-> printf "hello\n"
value = 6 = 0x6
```

The string prints in the virtual console if it is enabled. It also goes to **vioFd**. Now you can return to your **wtxtcl** session and call the procedure **listEvents** (or repeatedly run **wtxEventGet**) to see what events have arrived:

```
wtxtcl> listEvents
SYM_ADDED vioFd 0x490efec
CALL_RETURN 0x4ae2720 0x4 0xd0003
CALL_RETURN 0x4ae2720 0x4 0xd0003
CALL_RETURN 0x4ae2720 0x4 0xd0003
VIO_WRITE 0 mblk0
```

Depending on your exact environment, you may see additional events. You should see at least the five events shown above, indicating that the symbol **vioFd** has been

added to the symbol table, that the three calls to **ioGlobalStdSet** have returned, and that virtual I/O is available. The event of interest is the one that indicates that virtual I/O has been received and stored in the memory block **mblk0**. The easiest way to see the data is to use the **memBlockGetString** command:

```
wtxtcl> memBlockGetString mblk0
hello
```

Another useful command for dealing with memory blocks that contain VIO data is **memBlockWriteFile**, which can be used to dump the contents of a block to **stdout** directly:

```
wtxtcl> memBlockWriteFile mblk0 -
hello
```

The **-** character indicates that the block should be written to **stdout**. Before continuing, free the memory block holding the VIO data:

```
wtxtcl> memBlockDelete mblk0
```

Virtual Input

For an example of virtual input, return to the shell and run a task that reads input. Create a new variable and initialize it to 0, then run a **scanf( )** task that blocks waiting for input. Since you redirected standard input to the virtual I/O channel, a virtual I/O write will satisfy the **scanf( )** call.

First, be sure that IO redirection is still turned off, or turn it off. Then create the variable and issue the **scanf( )** in the Host Shell:

```
-> ?shConfig
SH_GET_TASK_IO = off
...
-> myVioData = 0
new symbol "myVioData" added to symbol table.
myVioData = 0x39d188: value = 0 = 0x0
-> scanf ("%d", &myVioData)
```

At this point the shell prompt does not return, because **scanf( )** is waiting for input. To send the data for **scanf( )** to convert, return to **wtxtcl** and issue this command:

```
wtxtcl> wtxVioWrite 1 -string "99\n"
0x3
```

In the shell, you should see the routine return, and you can inspect the value of the variable that **scanf( )** filled in:

```
value = 1 = 0x1
-> myVioData
myVioData = 0x39d188: value = 99 = 0x63 = 'c'
```

NOTE: The host shell can also redirect the I/O of all the spawned tasks to its own window. For more informations on this feature, please see the *Wind River Workbench Command Line User's Guide: Host Shell.*

B.4.12 Calling Target Routines

Calling target routines is easier than spawning tasks from scratch, because the target server and agent provide support to simplify the process. The central command is **wtxFuncCall**, which spawns a task on the target and arranges for an event containing the return value to be generated when the task completes. For example, use **sysClkRateGet( )** to report the system clock rate:

```
wtxtcl> wtxFuncCall -int [symbol sysClkRateGet] 0x1047b738
```

The call returns a *call ID*, the ID of the task that is spawned to call the routine. In addition, the agent posts an event to the target server which can be viewed using **wtxEventGet**, **listEvents** (see *Generating and Retrieving Events*, p.138), or **eventPoll** (see *Checking the Event Queue*, p.138), as shown in the following example:

```
wtxtcl> wtxEventGet
CALL_RETURN 0x1047b738 0x3c 0
```

The event string includes both the call ID and the integer return value (0x3c = 60).

The next example calls **taskDelay( )** with a delay of 10 seconds (assuming the system clock rate is 60Hz) and then polls at the rate of 1 Hz (every 1000 milliseconds) waiting for the result.

```
wtxtcl> wtxFuncCall -int [symbol taskDelay] 600
0x1047b738
wtxtcl> eventPoll 1000
CALL_RETURN 0x1047b738 0 0
```

B.4.13 Working With Multiple Connections

It is possible to maintain more than one target server connection at once with **wtxtcl**. To make two connections, issue the **wtxToolAttach** command twice, each time saving the Tcl handle in a variable for convenience.

```
wtxtcl> set h1 [wtxToolAttach vxsim7]
wtxtcl> set h2 [wtxToolAttach mv147]
wtxtcl> set h1
vxsim7@aven
wtxtcl> set h2
mv147@aven
```

The WTX Tcl API maintains a stack of handles. The most recently opened connection is at the top of the stack, so any WTX commands issued after entering the list of commands in the previous example go to **mv147**. To examine and manipulate the handle stack, use the **wtxHandle** procedure. With no arguments, it displays the stack (top item first). With an argument, it places the named handle at the top of the stack. Notice that the second command in the following example places **vxsim7** at the top of the stack; further WTX commands are directed there.

```
wtxtcl> wtxHandle
mv147@aven vxsim7@aven
wtxtcl> wtxHandle $h1
vxsim7@aven mv147@aven
```

You can also direct any WTX Tcl command to any connected handle without manipulating the stack. The generic **-hwtx** option, which takes a handle name argument, directs the WTX command to the named connection. For example, the following example shows a script which prints the BSP name of each connected target. Executing **wtxHandle** without arguments provides the list of connected targets, and running **wtxTsInfoGet** with the **-hwtx** option queries the correct target. Output is directed to **stdout**.

```
wtxtcl> foreach handle [wtxHandle] {
set info [wtxTsInfoGet -hwtx $handle]
puts stdout [lindex $info 4]
}
SunOS 5.5.1 [sun4u]
Motorola MVME2600 - MPC 603p
```

B.4.14 Timeout Handling

Three methods are available for adjusting timeout values. They are listed from the most general to the most specific.

wtxTimeout

This command affects all subsequent WTX requests. The specified value serves as the default for all commands that do not have a different value assigned.

To get the current timeout setting, issue the following command:

```
wtxtcl> wtxTimeout
30
```

To set a new timeout value, issue the following command:

```
wtxtcl> wtxTimeout 120
120
```

wtxCmdTimeout

This array allows you to set a timeout for any WTX commands you choose. If the entry corresponding to a particular command is not set, the default timeout applies, unless the **-timeout** option is used when the command is issued.

To set up a timeout for all **wtxObjModuleLoad** commands, issue:

```
wtxtcl> set wtxCmdTimeout(wtxObjModuleLoad) 2400
240
```

All subsequent object module loads will use a timeout value of 240 seconds.

-timeout

All WTX Tcl commands that involve communicating with the target server can take a **-timeout** option. This option affects only the timeout setting for the specific command.

```
wtxtcl> wtxSymFind -timeout 1 -name errno
errno 0x96b3c 0x9 0 0 ""
```

B.4.15 Error Handling

WTX Tcl has an error handling facility that is designed to take advantage of Tcl's own exception handling features. When a **wtxtcl** request results in a WTX protocol error, the request generates a Tcl error. Tcl errors initiate a process called *unwinding*. The procedure that invoked the offending **wtxtcl** command stops executing, and immediately returns to its parent with the error code returned by the target server. If the parent was called by another procedure, it returns the error code to its parent, and so on. This process can continue indefinitely until the entire stack of procedure invocations is removed or the error is caught by an error handling routine. At each step, a descriptive message is appended to the global variable **errorInfo**. Printing this variable displays a backtrace showing the circumstances leading to the error.

The next example shows an error occurring in a nested context. A **wtxMemRead** call specifying an invalid address is embedded in a **foreach** and a **while** loop. When the error is returned, **errorInfo** prints the value of the error messages.

```
wtxtcl> foreach x {1 2 3} {
while 1 {
wtxMemRead 0xeeeeeeee 0x100
}
}
Error: WTX Error 0x100ca (AGENT_MEM_ACCES_ERROR)
```

```
wtxtcl> set errorInfo
WTX Error 0x100ca (AGENT_MEM_ACCES_ERROR)
while executing
"wtxMemRead 0xeeeeeeeee 0x100"
("while" body line 2)
invoked from within
"while 1 {
wtxMemRead 0xeeeeeeeee 0x100
}"
("foreach" body line 2)
invoked from within
"foreach x {1 2 3} {
while 1 {
wtxMemRead 0xeeeeeeeee 0x100
}
}"
```

Tcl's standard exception handling command, **catch**, can be used with WTX calls. Using **catch** interrupts the unwinding process, preventing errors from escaping to higher levels and allowing errors to be inspected and acted on.

While it is sometimes appropriate to examine each error individually, it is also common to take a specific action in response to a particular protocol error regardless of what command caused the error. In addition to the Tcl **catch** command, **wtxtcl** provides a means to establish an error handling procedure.

An error handler is a special Tcl procedure that is invoked when a protocol error occurs. An error handler procedure is invoked with four arguments. These are: the handle name, the command provoking the error, the error message itself, and a tag that can be supplied when the error handler is attached. (For more information see the reference entries.) An error handler procedure must accept these four arguments. Here is a simple error handler that prints its arguments and then resubmits the error to the Tcl interpreter. Except for the printing it has no effect on the application.

```
wtxtcl> proc myErrorHandler {handle cmd err tag} {
puts stdout "handle $handle\ncmd $cmd\nerr $err\ntag $tag"
error $err
}
```

Error handler procedures are associated with communication handles, so different connections may have different error handlers in one **wtxtcl** session. To attach the error handler *myErrorHandler* to the WTX handle at the top of the handle stack, use the following command:

```
wtxtcl> wtxErrorHandler [lindex [wtxHandle] 0] myErrorHandler
```

You can test the process by defining this procedure to the interpreter and entering a faulty memory read request as in the following example:

```
wtxtcl> wtxMemRead 0xeeeeeeee 0x1000
handle vxsim@van-chauve
cmd wtxMemRead 0xeeeeeeee 0x1000
err WTX Error 0x100ca (AGENT_MEM_ACCESS_ERROR) tag WTX Error
0x100ca (AGENT_MEM_ACCESS_ERROR)
```

The first three lines of output were printed by the handler. The fourth was printed by the **wtxtcl** main loop in response to the error it received. This is because the error handler resubmits the errors it receives.

There are several guidelines to observe when writing error handlers:

- Errors that are not specifically treated by the handler should be resubmitted intact with the error command. This is because many applications expect to receive errors in certain circumstances, and contain code that checks the result of WTX calls attempting to match the result against expected error codes.
- If an error handler resolves an error by retrying the operation, it should check the command argument to make sure the error was generated by the expected command.
- While it is possible to chain error handlers, this can be complicated. It is better if each application maintains one central error handling procedure.

B.5 WTX C API

This section introduces the WTX C API and provides an extensive code example. To experiment interactively with the WTX protocol, you may wish to work through the examples given in the *B.4 WTX Tcl API*, p.131. All WTX primitives are available to both APIs; thus a single detailed explanation of many items that are common to both the Tcl and C APIs is included in that section.

B.5.1 Description

The WTX C API is a binding of the WTX protocol to the ANSI C language. This allows C applications to call target server services. Every WTX protocol request is accessible through the WTX C interface.

There is a WTX C routine for each WTX protocol request. For example, the C function **wtxMemRead( )** corresponds to **WTX_MEM_READ**. The names of all WTX C routines are derived from the protocol request names according to the procedure

described in the *VxWorks Hardware Interface Validation Guide: Coding Conventions*; in other words, underscores are removed and all words but the first are capitalized.

Each WTX C interface has an reference entry. For information on the online reference material, see your product *Getting Started*. All WTX C routines provide access to error status through **wtxErrorGet( )**. Throughout the remainder of this section, please refer to the reference entries for more information on the WTX C API.

B.5.2 WTX C API Archive

The WTX C API is archived in *installDir*/**workbench-2.***x*/**foundation/4.0.9/***host*/**lib**. The library is called **wtxapi40.lib** on Windows hosts and **libwtxapi40.so** on UNIX and Linux hosts.

B.5.3 Initializing a Session

Before connecting an application to a particular target server you must initialize a session by calling **wtxInitialize( )**. The following code example initializes the specified session handle **wtxh**:

```
include "wtx.h"
...
{
HWTX                wtxh;
...
/* initialize session */

if (wtxInitialize (&wtxh) != OK)
    return (ERROR);
```

B.5.4 Obtaining Target Server Information

The Wind River registry maintains information about executing target servers. To find the list of registered target servers, you can call **wtxInfoQ( )**. To find information on a particular target server, you can call **wtxInfo( )**.

```
#include "wtx.h"
...
WTX_DESC * pWtxDesc; /* target server information */
...
/* get information about a target server using the Tornado registry*/

if ((pWtxTsInfo = wtxInfo (wtxh, "tJohn")) == NULL)
    return (ERROR);
...
```

B.5.5 Attaching to a Target Server

An application must bind an initialized handle to an executing target server before the majority of the WTX C interface routines can be called. This can be accomplished calling **wtxToolAttach( )**, which takes an initialized **HWTX** handle, a regular expression, and a tool name. The regular expression must uniquely match a registered server for the binding to succeed. The tool name is the application name. The target server records it and reports this name to any tool that requests a list of attached tools with **wtxTsInfoGet( )**.

```
#include "wtx.h"
...
/* attach to a target server */

if (wtxToolAttach (wtxh, "tJohn.*", "newapp") != OK)
    {
    wtxTerminate (wtxh); /* terminate session */
    return (ERROR);
    }
...
```

B.5.6 Application Example

The following code examples demonstrate a complete application of the WTX C API. This application attaches to the specified target server and allocates and de-allocates target memory in an endless cycle. Note that the number of blocks to allocate is specified in a separate header file to conform to the Wind River coding conventions. For more information about compiling this example see the *Wind River Workbench User's Guide: Building Projects and Applications* or the *Wind River Workbench Command-Line User's Guide: Building Applications and Libraries*.

To observe this application, start the application:

```
% wtxapp targetServerName &
```

If the Workbench is running, use the target manager and select your target server; you can observe **wtxapp** in the list of attached tools.

Example B-2 **WTX C API Application**

```
# Makefile - Simple Makefile to build a WTX C API application # Copyright
2004 Wind River Systems, Inc.
#
# modification history
# --------------------
# 01a,20oct04,c_c written
#
# This makefile is used to generate a WTX program sample. it assumes that
Unix utilities are available on Windows. If not, then it should be edited
to replace the SHELL and RM macros to fit customer's settings.

SHELL   = sh.exe
RM      = rm ""

LOCAL_INCLUDE   = $(subst \,/,$(WIND_FOUNDATION_PATH))/include

OBJS    = wtxApp.o

ifeq ($(HOST),x86-win32)
PROGRAM         = wtxApp.exe
LD              = link.exe
LD_FLAGS        = /out:$@ /nologo /subsystem:console /machine:I386 /fixed:no
\
        /libpath:$(WIND_FOUNDATION_PATH)/x86-win32/lib /out:$@
LOCAL_LIBS      = wtxapi40.lib
CC              = cl.exe
CFLAGS          = /I. /DHOST=$(HOST) /nologo /MD /W3 /Gm /GX /D "WIN32"\
          /D "_WINDOWS" /Od /ZI /I$(LOCAL_INCLUDE) /c $< /Fo$@
LOCAL_CLEAN     = *idb *pdb
else
PROGRAM         = wtxApp
LD              = ld
LD_FLAGS        =
LOCAL_LIBS      = -lwtxapi40
CC              = gcc
CFLAGS          = /I. /DHOST=$(HOST) -I$(LOCAL_INCLUDE) -c $< -o $@
LOCAL_CLEAN     =
endif

$(PROGRAM): $(OBJS)
        $(LD) $(LD_FLAGS) $(OBJS) $(LOCAL_LIBS)

.c.o:
        $(CC) $(CFLAGS)

clean:
        @- $(RM) $(wildcard $(OBJS) $(PROGRAM) $(LOCAL_CLEAN))
```

```
/* wtxapp.h - Simple WTX C API application header */

/* Copyright 1995-99 Wind River Systems, Inc. */

/*
modification history
--------------------
01a,08apr95-99,p_m  written
*/

/* defines */

#define NUM_BLOCKS 26           /* number of memory blocks to allocate */
```



```
/* wtxapp.c - Simple WTX C API application */

/* Copyright 1995-99 Wind River Systems, Inc. */

/*
modification history
--------------------
01c,05jan99,fle doc : made it documentable
01b,11jul95,p_m  implemented better error reporting and added comments
                                added signals handling

01a,08apr95,p_m  written
*/

#include <signal.h>
#include <stdio.h>
#include "host.h"
#include "wtx.h"
#include "wtxapp.h"

/*
DESCRIPTION

This is a simple program demonstrating the use of the WTX C API.  It
attaches to a target server and allocates and free memory from the target
memory pool managed by the target server.

SEE ALSO
tgtsvr, wtxtcl, wtx, WTX
*/

/* defines */

/* globals */

HWTX            wtxh;                   /* WTX API handle */
TGT_ADDR_T      blockTab[NUM_BLOCKS];   /* allocated blocks table */
```

```
/* forward declarations */

LOCAL STATUS        wtxAppSigInit (void);

#ifdef WIN32
    BOOL WINAPI     wtxAppSigHandler (DWORD dwCtrlType);
#else
    LOCAL void      wtxAppSigHandler (int signal, int code);
#endif
LOCAL void          wtxAppTerminate (void);

/*****************************************************************
*
* wtxapp - Simple WTX application example
*
* This is a simple WTX tool example showing the WTX C API.  It
* allocates blocks of memory from the target-server-managed target memory
* pool then frees them.  The effects of this program can be easily
* viewed by using a Tornado browser attached to the same target server.
*/

int main
    (
    int             argc,       /* number of arguments */
    char *          argv[]      /* table of arguments */
    )
    {
    UINT32          ix = 0;     /* useful counter */
    UINT32          blockSize;  /* size of a memory block */
    WTX_MEM_INFO *  pMemInfo;   /* target memory pool info */

    /* check input arguments */

    if (argc != 2)
        {
        printf ("Usage: wtxapp targetServerName\n");
        exit(0);
        }

    /*
    * Initialize signal handlers. This is necessary to avoid a tool
    * killed via Control-C or another signal remaining attached to a
    * target server.
    */


    if (wtxAppSigInit() != OK)
        {
        printf ("Error: cannot initialize signal handler\n");
        exit (0);
        }

    /* initialize WTX session */

    if (wtxInitialize (&wtxh) != OK)
```

```
    {
    printf ("Error: cannot initialize WTX API\n");
    exit (0);
    }

/* attach to target server */

if (wtxToolAttach (wtxh, argv[1], "wtxapp") != OK)
    {
    printf ("Error: cannot attach to target server: %s\n", argv[1]);
    wtxTerminate (wtxh);
    exit (0);
    }

printf ("Attaching to target server done\n");

/* register for events we want to hear about */

if (wtxRegisterForEvent (wtxh, ".*") != OK)
    {
    printf ("Error: cannot register for events\n");
    wtxAppTerminate ();
    }

/* get memory pool information */

if ((pMemInfo = wtxMemInfoGet (wtxh, 0)) == NULL)
    {
    printf ("Error: cannot get target memory information\n");
    wtxAppTerminate ();
    }

printf ("Starting target memory allocate/free demonstration\n");
printf ("Type Control-C to stop\n");

/* allocate approx. 1/NUM_BLOCKS of the biggest available blocks */

blockSize = pMemInfo->biggestBlockSize / NUM_BLOCKS - 200;

for (;;)
    {
    /* allocate blocks */

    for (ix = 0; ix < NUM_BLOCKS; ix++)
        {
        printf ("Allocating block #%02d\r", ix+1);
        blockTab [ix] = wtxMemAlloc (wtxh, 0, blockSize);

        if (blockTab[ix] == (TGT_ADDR_T) 0)
            {
            printf ("Error: cannot allocate target memory\n");
            wtxAppTerminate ();
            }

        #ifdef WIN32
            Sleep (1000);       /* wait 1 second */
```

```
            #else
                sleep (1);          /* wait 1 second */
            #endif
                fflush (stdout);
            }

        /* free blocks */

        for (ix = 0; ix < NUM_BLOCKS; ix++)
            {
            printf ("Freeing block #%02d   \r", ix+1);

            if (wtxMemFree (wtxh, 0, blockTab [ix]) != OK)
                {
                printf ("Error: cannot free target memory\n");
                wtxAppTerminate ();
                }
            blockTab [ix] = (TGT_ADDR_T) 0;
            #ifdef WIN32
                Sleep (1000);       /* wait 1 second */
            #else
                sleep (1);          /* wait 1 second */
            #endif
            fflush (stdout);
            }
        }
    return 0;
    }

/*******************************************************************
*
* wtxAppSigInit - initialize signal handlers
*
* wtxAppSigInit() installs the signal handlers needed by wtxapp.
*
* RETURNS: N/A
*/

STATUS wtxAppSigInit (void)
    {
    #ifndef WIN32
        int                 ix;
        struct sigaction    sv;
        struct sigaction    bb;


        /* setup default signal handler */

        sv.sa_handler = (VOIDFUNCPTR) wtxAppSigHandler;
        sigemptyset (&sv.sa_mask);
        sv.sa_flags   = 0;

        for (ix = SIGHUP; ix < SIGALRM; ix++)   /* capture errors */
            sigaction (ix, &sv, &bb);

        sigaction (SIGTERM, &sv, &bb);          /* terminate */
```

```
    #else
        SetConsoleCtrlHandler(wtxAppSigHandler, TRUE);  /* trap signals */
    #endif
    return OK;
    }

/*******************************************************************
*
* wtxAppTerminate - terminate wtxapp properly
*
* This routine performs all necessary cleanup required when an attached tool
* finishes a WTX session with a target server.
*
* RETURNS: N/A
*/

LOCAL void wtxAppTerminate (void)
    {
    int ix;

    printf ("\nFreeing all the memory blocks\n");
    for (ix = 0; ix < NUM_BLOCKS; ix++)
        {
        if (blockTab [ix] != (TGT_ADDR_T) 0)
            {
            wtxMemFree (wtxh, 0, blockTab [ix]);
            }
        }
    wtxToolDetach (wtxh);           /* detach from target server */
    wtxTerminate (wtxh);            /* terminate WTX session */

    exit (0);                               /* bye */
    }

/*******************************************************************
*
* wtxAppSigHandler - wtxapp signal handler
*
* This function is the signal handler for wtxapp.  It calls
* wtxAppTerminate() to perform the WTX session closing.
*
* RETURNS: N/A
*/

#ifdef WIN32
    BOOL WINAPI wtxAppSigHandler
    (
    DWORD                   dwCtrlType
    )
#else
    LOCAL void wtxAppSigHandler
    (
    int                             signal,
    int                             code
    )
#endif
```

```
{
wtxAppTerminate ();             /* terminate session cleanly */
#ifdef WIN32
return 0;
#endif
}
```

B.6 Integrating WTX with Applications

WTX Tcl can be used with WTX C or with non-WTX Tcl applications.

B.6.1 Using the Tcl and C APIs Together

Both the Tcl and C APIs provide access to all the WTX protocol commands. Which API to use for a particular application or module depends on whether an interactive or a compiled environment is more appropriate to the specific task. For example, you might write a development tool in C for which you wish to provide the same level of configurability as the rest of Workbench. The Tcl API is well suited to this application, and in fact is used to configure the various Tornado tools.

When using Tcl and C APIs together, an important question to address is: within which API will the target server connection be created? The API which creates the connection also creates the handle name. The handle must be passed to the other API in order for it to access the target server.

If connections are to be created by Tcl code using the **wtxToolAttach** command, the C code can obtain the **HWTX** structure associated with a Tcl handle name with the C routine **wtxTclHandle( )**:

```
HWTX wtxTclHandle (char * handleName)
```

If you call this routine with a **NULL** argument, it returns the handle at the top of the Tcl handle stack.

If connections are to be created by C code calling **wtxToolAttach( )**, the C program can pass the C API handle to the Tcl interpreter by calling **wtxTclHandleGrant( )**:

```
char * wtxTclHandleGrant (HWTX hWtx)
```

If you call this routine with a C API handle, a Tcl handle is created and placed on the top of the Tcl handle stack. The routine returns the handle name used by Tcl.

If the C program wishes to close one of these connections, it should first revoke the Tcl handle with **wtxTclHandleRevoke( )**:

```
void wtxTclHandleRevoke (HWTX hWtx)
```

B.6.2 Integrating WTX Tcl with Other Tcl Applications

You can integrate WTX Tcl with other Tcl interpreters. The shared library **wtxtcl4.0.lib** (Windows hosts) or **libwtxtcl40.so** (Linux/UNIX hosts) is in *installDir*/**workbench-2.x/foundation/4.0.5/***host*/**lib**; it can be linked with any Tcl 8.4 application. You must initialize the library with a call to **wtxTclInit( )**, supplying the Tcl interpreter handle as an argument:

```
int wtxTclInit (Tcl_Interp * pInterp)
```

Once you initialize the library, all the WTX Tcl commands are available to the Tcl interpreter. For example, to attach to a target, you can evaluate a **wtxToolAttach** Tcl expression.

Index

D

E

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W