## Contents

1 Tornado Architecture ............................................................................................................... 1

1.1 Introduction ....................................................................................................................... 1

1.2 The WDB Agent and the WDB Protocol ........................................................................... 2

1.3 The Target Server and the WTX Protocol ....................................................................... 3

1.4 Target Server Threads ..................................................................................................... 7

1.5 Overview of Tornado APIs .......................................................................................... 8

1.6 Tornado Tree Organization ........................................................................................... 11

1.6.1 Path Name Conventions ......................................................................................... 11

1.6.2 Tornado Libraries .................................................................................................. 11

1.6.3 Resources ................................................................................................................ 12

1.7 Coding Conventions ...................................................................................................... 14
2 Target Server Back End ........................................................................................................ 15

2.1 Introduction ...................................................................................................................... 15
  2.1.1 Target Server Overview ....................................................................................... 16
  2.1.2 Back-End Overview .............................................................................................. 17
    WDB Back Ends ........................................................................................................... 17
    Non-WDB Back Ends ................................................................................................. 18

2.2 Back-End Implementation ................................................................................................. 19
  2.2.1 Attachment and Initialization .............................................................................. 20
    The Flag Parsing Routine ....................................................................................... 21
    The Initialization Routine ....................................................................................... 23
    The TGT_OPS Structure ......................................................................................... 24
    The BKEND_INFO Structure ................................................................................... 25
  2.2.2 Back-End Functions ............................................................................................... 26
  2.2.3 Event Notification ................................................................................................. 27
  2.2.4 Designing a select-like routine ........................................................................... 28
  2.2.5 Handling Target-Board Reboots ....................................................................... 28
  2.2.6 Cleaning Up the Tornado Registry ..................................................................... 29
  2.2.7 Message Logging ................................................................................................. 30
  2.2.8 Informing Windows About a New Back End ..................................................... 31

2.3 Writing a New WDB Back End ......................................................................................... 32
  2.3.1 Overview of Writing a WDB Back End ............................................................... 32
  2.3.2 WDB Protocol ........................................................................................................ 33
    Protocol Overview .................................................................................................... 33
    Requests Sent to the Agent ....................................................................................... 33
    Replies Sent by the Agent ......................................................................................... 35
    Reply Errors ............................................................................................................. 36
    Asynchronous Notification Sent By the Agent ....................................................... 38
  2.3.3 Host-Side Code ....................................................................................................... 38
    Back-End Initialization: wdbserial.c ....................................................................... 40
    Client-Side RPC Implementation ............................................................................. 43
3.2 Terminology ........................................................................................................ 90
3.3 Loader Overview ................................................................................................. 91
  Linking and Loading ............................................................................................. 91
  Bootstrapping the Target Server Symbol Table ............................................. 92
3.4 Using the Target Server Loader ....................................................................... 92
  3.4.1 Loader Usage from the Tornado Shell ............................................ 92
  3.4.2 Loader Usage From \texttt{wtxcl} ............................................................ 93
  3.4.3 Loader Usage From the WTX C API ............................................. 94
  3.4.4 Asynchronous Load Operation ..................................................... 94
3.5 Loader Architecture .......................................................................................... 95
  3.5.1 Object-Module Configuration in Target Memory ............................. 96
    Relocatable Files ............................................................................................. 96
    Fully Linked Files ......................................................................................... 97
  3.5.2 Module Management ........................................................................ 97
  3.5.3 Symbol Management ........................................................................ 98
  3.5.4 Loader Options .................................................................................. 101
    Symbol Scope ............................................................................................... 101
    Module Visibility ......................................................................................... 102
    Module Type ............................................................................................... 102
    Common Symbols ....................................................................................... 102
    Special Options ......................................................................................... 103
3.6 Memory Management ....................................................................................... 103
  3.6.1 Memory Alignment ........................................................................... 104
  3.6.2 Target Server Memory Cache ............................................................ 105
  3.6.3 Target Virtual Memory and Cache Management .......................... 106
  3.6.4 Type Abstractions and Address Manipulation ............................. 107
3.7 Generic Loader-Library Interface ..................................................................... 107
3.8 Target Server Loader Thread ........................................................................... 107
3.9 Writing an OMF Manager ................................................................. 108
  3.9.1 Dynamically Linked Library Implementation .......................... 109
  3.9.2 Installing a Shared-Library Manager ....................................... 109
  3.9.3 Naming Conventions ............................................................... 110
    Interface Routines ................................................................. 110
    File Names .............................................................................. 111
  3.9.4 The RU Interface ................................................................. 112
  3.9.5 The OMF Interface ............................................................... 114
  3.9.6 Byte Order .......................................................................... 122
  3.9.7 OMF Header Processing ........................................................ 124
  3.9.8 Determining the Size of the Segments ................................... 125
  3.9.9 Reading the Object-Module Symbol Table ............................. 128
  3.9.10 Determining the Compiler Signature .................................... 129
  3.9.11 Allocating Memory For the Segments if Required ................ 131
  3.9.12 Reading in the Segments ..................................................... 131
  3.9.13 Target Server Symbol-Table Processing ............................... 136
  3.9.14 Relocating the Object Modules ........................................... 142
    Reading in Relocation Entries ................................................ 142
    Relocation Process .................................................................. 143
  3.9.15 Downloading Relocated Modules to Target Memory ........... 145
  3.9.16 Managing Memory ............................................................. 147
  3.9.17 Notes on Asynchronous Load Operation ............................. 147
  3.9.18 Utility Functions and Error Logging ................................. 147

4 The WTX Protocol ........................................................................... 149
  4.1 Introduction ............................................................................... 149
  4.2 Protocol Overview ................................................................. 149
    4.2.1 Language Support ........................................................... 150
4.4.8 Working with Object Modules ......................................................... 172
4.4.9 Working with Tasks ................................................................. 173
4.4.10 Working with Events ............................................................... 173
    Registering for Events .............................................................. 174
    Unregistering for Events ........................................................... 174
    Checking the Event Queue .......................................................... 174
    Generating and Retrieving Events ............................................. 175
4.4.11 Working with Virtual I/O ....................................................... 177
    Virtual Output ............................................................................ 177
    Virtual Input ............................................................................. 179
4.4.12 Calling Target Routines ......................................................... 180
4.4.13 Working With Multiple Connections ....................................... 181
4.4.14 Timeout Handling ................................................................. 181
    wtxTimeout ................................................................................. 182
    wtxCmdTimeout ......................................................................... 182
    -timeout .................................................................................. 182
4.4.15 Error Handling ................................................................. 182
4.5 WTX C API ....................................................................................... 185
    4.5.1 Description ........................................................................... 185
    4.5.2 WTX C API Archive ............................................................. 185
    4.5.3 Initializing a Session ............................................................... 185
    4.5.4 Obtaining Target Server Information .................................... 186
    4.5.5 Attaching to a Target Server .................................................. 186
    4.5.6 Application Example ............................................................. 187
4.6 Integrating WTX with Applications ............................................... 192
    4.6.1 Using the Tcl and C APIs Together ........................................ 192
    4.6.2 Integrating WTX Tcl with Other Tcl Applications ................. 193
5 Extending Tornado Tools ................................................................. 195

5.1 Introduction ................................................................................. 195

5.2 Outlining the Example Problem ................................................ 195

5.3 Extracting Information from a Structure ..................................... 197
  5.3.1 Definitions and Structures ..................................................... 198
  5.3.2 Locating the Start of the First Structure ................................. 199
  5.3.3 Listing All Network Interfaces ............................................. 200
  5.3.4 Listing the Contents of Each Structure ................................. 201

5.4 The Shell Presentation ............................................................... 204

5.5 The Browser Presentation .......................................................... 207
  5.5.1 UNIX Implementation .......................................................... 208
     Creating the Hierarchy Window Display .................................. 208
     Installing the New Browser Panel .......................................... 211
  5.5.2 Windows Implementation .................................................... 214

6 Adding Tools to Tornado ............................................................... 219

6.1 Introduction ................................................................................ 219

6.2 Creating a Tornado Application ................................................ 219
  6.2.1 Example Specifications ......................................................... 220
  6.2.2 Directories and File Naming ............................................... 220
  6.2.3 Building the Application ..................................................... 220
  6.2.4 Compiling the Application .................................................. 222
     UNIX Hosts .............................................................................. 222
     Windows Hosts ....................................................................... 222
  6.2.5 Testing the Application ....................................................... 223
     UNIX Hosts .............................................................................. 223
     Windows Hosts ....................................................................... 224
1 Tornado Architecture

1.1 Introduction

The central principles of the Tornado architecture are the following:

- Minimal intrusion on the target, by exploiting the development host as extensively as possible for interactive development tools.
- Centralized communications with the target, so that multiple development tools can share access to the target even when communications channels are limited.
- Published protocols and application program interfaces (APIs) to make it simple to customize or extend Tornado.

The following two protocols are particularly important, because they mediate all contact between a Tornado tool and the target:

**WTX** The WTX (Wind River Tool eXchange) protocol specifies how the target server communicates with host-resident development tools. The protocol is extensible, to permit adding new services as required for new tool designs.

**WDB** The WDB (Wind DeBug) protocol specifies how the target server (on the host) communicates with the target agent (on the target). The protocol includes a compact programming language called Gopher, which permits on-the-fly extension by supporting programmable investigation of the target system.

Figure 1-1 illustrates the role of these two protocols in Tornado.
1.2 The WDB Agent and the WDB Protocol

The WDB agent carries out requests transmitted from the target server, and replies with the results. The WDB agent contains a compact implementation of UDP/IP, which supports an RPC messaging protocol called WDB. The WDB protocol is a core minimum of the services necessary to respond to requests from the Tornado tools. These protocol requests include memory transactions, breakpoint/event notification services, virtual I/O support, and tasking control. The WDB protocol uses the Sun Microsystems specification for External Data Representation (XDR) for data transfer.

The WDB agent synthesizes the target-control strategies of task-level and system-wide debugging. The agent can execute in either mode and switch dynamically between them, provided the appropriate drivers are present in the Board Support Package (BSP). This permits debugging of any aspect of an embedded application whether it is a task, an interrupt service routine, or the kernel itself.

A significant feature provided by the WDB agent is its ability to interpret a simple language called Gopher (for a description of the Gopher language, see 4.3.11 Gopher Support, p.160). Gopher scripts are very compact strings that specify a target data
structure (or a list of structures) of arbitrary complexity to be gathered and transferred to the host. Because this technique is interpretive, the target is not burdened with specialized routines to gather state information for every system object.

By virtue of a Gopher script, application developers can likewise gather application-object state information without burdening the application with state-information routines. The Gopher interpreter itself is very small (smaller than the amount of the memory that would otherwise be required by target routines to gather only the most essential system objects), and the language is compact to reduce communication overhead. The Gopher language is accessible at higher-level Tornado APIs.

The WDB agent’s interface to communications drivers avoids the run-time I/O system, so that the WDB agent remains independent of the run-time OS. Drivers for the WDB agent are low-level drivers that provide both interrupt-driven and polling-mode operation. Polling mode is required to support system-level control of the target.

The WDB agent itself is independent of the target operating system: it attaches to run-time OS services through a virtual-function run-time interface. The WDB agent can execute before VxWorks is running (as in the early stages of porting a BSP to a new board) or even stand-alone.

Should your application require different function from that provided by the WDB agent, the target server can communicate with alternative agents by means of an alternative back end. For more information on non-WDB agents and back ends, see 2. Target Server Back End.

1.3 The Target Server and the WTX Protocol

Target servers run on Tornado host systems. Each target server represents one development target. All tools access a target through its server, whose responsibility is to satisfy each tool’s requests for target actions or target information. The target server manages all of the details of communicating with the target, so that tools are not concerned themselves with host-target transport details. Tornado tools use the WTX protocol to communicate with a target server.

In some cases, a request from a Tornado tool is passed directly to the WDB agent for execution on the target system. In other cases, requests can be managed entirely
within the target server. For example, the target server caches recently-used regions of memory. This enables the target server to respond directly when a target-memory read hits a memory region that is already cached in the target server, thereby avoiding the need for an actual host-target transaction.

The target server architecture is shown schematically in Figure 1-2, and outlined in the following paragraphs.

Figure 1-2  Target Server Architecture

- **WTX Protocol Dispatch.** The “front end” of the target server, this layer connects to Tornado tools, receives protocol requests and prepares replies, in the meantime calling target server subroutines to provide the services required.
- **Target Server Core.** The target server core contains the interpreter for the WTX protocol. It is here that the target server determines what services to perform locally on the host, and what services to pass along to the back end to transmit to the target. WTX services that can be carried out entirely in the target server are implemented in the target server core; general-purpose callbacks used by other parts of the target server also reside here.

- **Target Symbol Table.** The target server uses the symbol list contained in every object module to build a host-resident symbol table representing all loaded functions and variable names on the target system. Names from system and application modules alike are included in the target symbol table. The object module loader uses the symbol table to resolve external references and relocate symbolic references. Tornado tools can also manipulate the symbol table through WTX requests.

- **Object Loader/Unloader.** Central to the Tornado architecture is its ability to incrementally load object modules into the target system. Because modules can be loaded dynamically, development cycles are much shorter: when a developer changes a particular feature, only the affected module needs to be recompiled and reloaded, not the entire application. The loader uses the system symbol table to resolve undefined references in modules being loaded, dynamically linking newly loaded modules to previously loaded modules. In addition, the target server provides the ability to unload modules from the system. When a module is removed, all the associated symbols are removed from the symbol table and memory is reclaimed.

- **OMF Readers.** Object modules come in many formats. The target server utilizes shared libraries on UNIX hosts and dynamically linked libraries (DLLs) on Windows hosts to isolate the OMF reader from the target server core. This makes it simple to add new object module formats without requiring a new release of the target server.

- **Target Memory Manager.** The loader requires memory from the target pool when allocating space for the linked module, and host-resident Tornado tools, such as the shell or debugger, also use some target memory as a working area. Rather than call upon the target to manage a pool of memory on the host tools’ behalf, the target server manages a pool of target memory for all allocation requests originating on the host. The target memory manager maintains all bookkeeping structures (such as a free block list and block headers) on the host, and thus provides zero-overhead target memory allocation. When a tool asks for 24 bytes, it receives exactly 24 bytes of target memory (subject to alignment). The size of the pool managed by the target server is configured at target start-up time, but it can increase dynamically.
- **Target Memory Cache.** Some regions of target memory are good candidates for caching. The target memory cache, by default, caches the program text sections of all target-resident modules. Because it is highly unorthodox to modify the text of the program (beyond adding breakpoints), this use of the cache provides a considerable boost in performance for cross-debugging.

- **Target Server disassembler.** Your WTX application can call on the target server to disassemble a region of target memory. The target server disassembly facility differs from GDB disassembly in that it produces chip-manufacturer standard output instead of FSF standard output. The disassembler for each chip resides in its own shared library which is dynamically linked when the target server attaches to the target.

- **Target Server File System.** The Target Server File System (TSFS) is a full-featured VxWorks file system, but the files operated on by using the file system are actually located on the host. TSFS uses a WDB driver to transfer requests from the I/O system to the target server. The target server reads the request and executes it using the host file system and the host-target communication path. It is possible to access host files randomly without copying the entire file to the target, to load an object module from a virtual file source, to supply the file name to routines such as `ld()` and `copy()`, and even to open a socket without installing the TCP/IP stack on the target.

- **Back-End Manager.** Perhaps one of the most convenient aspects of the target server is its handling of the drudgery associated with managing a host-target connection mechanism. The back-end manager maps a transport-independent layer of subroutine calls, used by the rest of the target server, into the appropriate transport-dependent calls for a particular back end.

  In some cases, certain back-end requests have no direct mapping to a particular back end. For instance, it is unlikely that an emulator can support the parsing of a Gopher script. In this case the back-end manager decomposes the request into more easily mapped target-memory reads. Likewise, the back-end manager behaves correctly in response to the inability of a back end to perform some types of breakpoint strategies. By virtue of Tornado’s back-end management, tool providers can concentrate on enhanced functionality instead of porting to yet another connection strategy.

- **Target Server Back End.** Whether it is as simple as an Ethernet connection, or as complicated as an emulator connection, the target server back end does whatever is required to communicate a WDB request to the target agent. As with the OMF readers, the target server back ends are organized as shared libraries on UNIX hosts, and as DLLs on Windows hosts. This allows introducing a new connection mechanism at any time.
1.4 Target Server Threads

The target server uses multiple threads to service WTX requests, target events, load requests, and VIO events efficiently. Figure 1-3 shows the various threads.

Figure 1-3 Target Server Threads

- **WTX thread.** Each thread services a unique WTX client. When a tool connects to the target server, the target server creates a thread for it which processes all tool requests and exists until the tool disconnects.
- **Loader thread.** This thread waits for and performs load requests. This allows the load to be performed asynchronously, so clients are free to do other operations while the load is in progress. It also means a load can be cancelled in process.
Virtual I/O thread. This thread waits for inputs from a target server virtual console and sends data to the associated virtual I/O channel. On Windows it is created when the target server is started with the `-C` (virtual console) option. On UNIX it is always available because UNIX permits consoles to be created at any time using `wtxConsoleCreate` and this thread is in place to support all consoles.

Asynchronous events thread. This thread waits for and handles target events. (For more information, see 2. Target Server Back End).

1.5 Overview of Tornado APIs

A central feature of Tornado is the rich set of APIs that allow access to each level of the technology. These APIs are outlined here to provide a broad view of what options are available to developers who wish to extend the Tornado environment.

The easiest APIs to learn about are those based on Tcl, in part because of the interactive nature of Tcl, but also because extensive examples are readily available. (If you are not familiar with Tcl, see the Tornado User’s Guide: Tcl for an overview of the language and for references to more detailed expositions.) The Tcl source for all standard Tornado tools is included with every Tornado installation: for example, procedures for examining semaphores, dumping memory, spawning tasks, and so on are available for customization, extension, or simply for their educational value. On UNIX hosts, every aspect of the user interface is also under user control, from dialogs to menu items. (The graphical interface on Windows hosts is somewhat less configurable, in part due to the existence of interface standards on that platform).

The Tornado APIs are shown graphically in Figure 1-4. The following paragraphs discuss each interface:

- **Graphical User Interface.** This API provides a set of Tcl commands to create and configure graphical user interfaces (such as menus, buttons, dialog boxes, or charts). You can use this interface to create new Tornado tools or to customize existing tools like the launcher or browser.

- **WTX C API.** This API provides a complete set of C routines covering all the WTX protocol services. Use this interface to create tools that access the target server or the Tornado registry, when you use C as the implementation language. The CrossWind debugger is implemented using this API.
- **WTX Tcl API.** The WTX Tcl API provides a complete set of Tcl commands covering all the WTX protocol services. Use this interface to create tools that access the target server or the Tornado registry, when you use Tcl as the implementation language. The Tornado shell and browser (and, on UNIX hosts, the launcher) are implemented through this API.

- **WTX Java API.** The WTX Java API provides a set of Java classes permitting access to the WTX protocol services. It is built on top of the WTX C API using the Java Native Interface; its principal class implementing methods call the WTX C protocol routines. Use this interface to create tools that access the target server or the Tornado registry using Java. It is available free of charge through Wind River System WEB server (www.wrs.com).
• **WTX Protocol.** The WTX protocol defines a set of RPC requests interpreted by
the target server or the Tornado registry. You can use RPC calls to send WTX
protocol requests directly, but in order to simplify application development,
we recommend the use of the WTX C or Tcl APIs instead.

• **Target Server Core API.** Several C subroutine libraries provide callbacks to
target server core services. These callbacks are required to write support for
either a new object module format (OMF) reader, or a new target
communication back end. The libraries listed in Table 1-1 collectively
constitute the target server core API. The associated DLLs are shown in
Table 1-2 and Table 1-3 for UNIX and Windows.

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bkendlib</td>
<td>Back-end communications interface.</td>
</tr>
<tr>
<td>bkendlog</td>
<td>Back-end logging facility.</td>
</tr>
<tr>
<td>loadlib</td>
<td>Common object module loader routines.</td>
</tr>
<tr>
<td>symlib</td>
<td>Symbol management facility.</td>
</tr>
<tr>
<td>tgtlib</td>
<td>Target operation support routines.</td>
</tr>
<tr>
<td>tgtmem</td>
<td>Target server memory management facility.</td>
</tr>
<tr>
<td>wpwrlog</td>
<td>Error and warning logging utilities.</td>
</tr>
</tbody>
</table>

• **OMF Reader Interface.** The OMF (object module format) reader interface
specifies a set of format-independent subroutine calls to manage object code.
A new shared library or DLL to support a new object-code format must define
the subroutines specified by this interface.

• **Target Server Back-End Interface.** The back-end interface specifies a set of
transport-independent subroutine calls to manage communication with the
target agent. A new shared library or DLL to support a new transport
mechanism must define the subroutines specified by this interface.

• **Target Agent Interface.** The WDB agent relies on low-level device drivers to
connect with the target server. The agent interface specifies what routines a
device driver must provide to the WDB agent.

The remaining chapters of this manual discuss these APIs in more detail. Reference
entries for all subroutines, protocols, and interfaces appear in the online reference
material under Tornado API Reference.
1.6 Tornado Tree Organization

The Tornado tree unifies all Wind River Systems products under a single directory with a common root. This greatly facilitates installation, and reduces the administrative overhead of managing path and environment variables.

The tree design makes it possible for Wind River Systems or third parties to add new products to the tree, and even update previously installed products to reflect the existence of the new product. For instance, on UNIX hosts the launcher can be extended to include an additional button to launch the new product, without requiring a rebuild or re-release of the base Tornado product.

1.6.1 Path Name Conventions

The top-level Tornado directory structure includes three major directories (see the Tornado User's Guide: Directories and Files). Most references in this manual will be to files in the installDir/host directory, although this is not always the case. For example, if you install Tornado in /group/wind on a UNIX host or in C:\Tornado on a Windows host, the full pathname for the file shown as installDir/host/resource/tcl/app-config/WindSh is /group/wind/host/resource/tcl/app-config/WindSh (which is also $WIND_BASE/host/resource/tcl/app-config/WindSh) on UNIX or C:\Tornado\host\resource\tcl\app-config\WindSh on Windows.

NOTE: In this manual, forward slashes are used as pathname delimiters for both UNIX and Windows file names since this is the default for VxWorks.

1.6.2 Tornado Libraries

The libraries in the installDir/host/hostType/lib directory (which is installDir/host/$WIND_HOST_TYPE/lib in UNIX and installDir/host/x86-win32/lib in Windows) provide support routines for Tornado applications. The routines in these libraries are documented in the appendices of this manual.

NOTE: Some of the Windows libraries are static libraries, while others are DLL export libraries.
In addition to the formal Tornado APIs, you can customize or enhance Tornado by overriding, changing, or adding to the files under the installDir/host/resource subdirectory. This directory contains all components of the Tornado implementation that do not require compilation, notably the Tcl procedures that implement the user interface (in installDir/host/resource/tcl).
Aside from Tcl source, the resource directory collects such things as mappings between target architectures and OMF libraries, bitmaps for screen objects, definitions for services that extend the WDB protocol, help files, and (on UNIX hosts) X Window System resources.

The `installDir/host/resource/tcl/app-config` directory includes a subdirectory for each Tornado tool. Every Tornado tool begins execution by sourcing all Tcl procedures contained in its resource subdirectory. For example, the shell (whose main Tcl source is in the file `installDir/host/resource/tcl/shell.tcl`) sources all of the files in the directory `installDir/host/resource/tcl/app-config/WindSh`, in alphabetical order. Thus, new features (such as buttons or menu items) can be added to Tornado tools by adding a new file to the appropriate subdirectory. (For local or personal customization, however, it is usually better to save customizations in a private `.wind` directory; see the Tornado User’s Guide: Directories and Files for more information.)

Files are read in shell collating order (the same order that the files would be displayed by the `ls` or `dir` commands). This is why these files typically start with two numbers: so that new additions to this directory can be interleaved with existing files easily. In addition, to add a later modification that further extends an existing NetShow extension, you could provide a file called `01NetShow2.tcl`, which would be read immediately after `01NetShow.tcl`.

Fully harnessing the interpretive nature of the Tornado environment is the key to a successful Tornado integration. Wind River Systems encourages the enhancement of the Tornado environment using this technique. Unlike many other development frameworks, Tornado remains open to the extent that third parties can not only introduce new components, but override the look, feel, and even functionality of previously existing Tornado components.

To the extent that your organization permits it, you can also benefit other Tornado users by sharing your Tornado resource extensions. The `comp.os.vxworks` newsgroup is one appropriate vehicle, as is the archive maintained by the Wind River Users Group. See the Tornado User’s Guide: Customer Service for information about finding these resources on the World Wide Web.
1.7 Coding Conventions

Tcl and C programs added to Tornado should follow the Wind River Systems coding conventions. Tcl coding conventions are described in A. Tcl Coding Conventions. C coding conventions are described in the VxWorks Programmer’s Guide: C Coding Conventions.
2

Target Server Back End

2.1 Introduction

The target server acts as a broker for the communication path to the target and provides services commonly required by the Tornado tools. In order to support new CPU architectures, new object-module formats, and new communications back ends with minimum programming effort, the corresponding parts of the target server are structured as shared libraries on UNIX or dynamically linked libraries (DLLs) on Windows. The remainder of the target server, known as the target server core, is independent of the target architecture, the target operating system, and of the communications transport layer.

This chapter examines the communications back end. It shows how a back end fits into the architecture of the target server, how the back end is implemented in Tornado, and how to implement new back ends, either for the WDB agent or for non-WDB agents such as emulators. For an overview of the target server and its components, see 1.3 The Target Server and the WTX Protocol, p. 3. To add a new object-module-format loader, see 3. Object-Module Loader. This chapter gives an overall orientation; for details, see the online reference material under Tornado API Reference>Target Server Back End Interface.

1. The Tornado implementation is designed to work in both UNIX and Windows environments; thus, in most cases the terms can be used interchangeably. In this chapter, DLL refers to both Windows DLLs and UNIX shared libraries unless an explicit distinction is made.
2.1.1 Target Server Overview

A target server session typically involves the following activities:

**Initializing the Target Server**

1. The target server is invoked with the following syntax:

   **UNIX:**
   ```shell
   % tgtsvr -V -B mybkend ... &
   ```

   **Windows:**
   ```shell
   c:> tgtsvr -V -B mybkend ... 
   ```

   where `-V` enables verbose diagnostic messages and `-B` specifies the back end.

2. The target server loads the back-end DLL specified with the `-B` parameter. (In the example, the back end is named `mybkend`; in the discussion, replace `mybkend` with the name of the back end you are developing.) The target server asks the back end if it can be customized through special command-line arguments.

3. If so, the target server parses its command-line argument and feeds the information to the back end.

4. Next, the target server registers itself with the registry, `wtxregd`, so that tools can connect to it.

5. The target server initializes the back end by calling a function named `mybkendInitialize()` in the DLL. Every back end must provide an initialization routine which has the name of the back end followed by `Initialize`. The routine `mybkendInitialize()` initializes the back end, fills in a `TGT_OPS` structure with the address of each back end function, and a `BKEND_INFO` structure with the method chosen to alert the target server when events come to the back end. The `TGT_OPS` structure is defined in `installDir/host/include/host.h` and the `BKEND_INFO` structure is defined in `installDir/host/include/bkendlib.h`.

6. The target server connects the back end to the target by calling the back-end target-connect function.

7. Finally, the target server performs other initialization, such as loading the OMF reader and reading the symbols in the OS core file.
Servicing Tool Requests

1. The target server launches a thread to wait for WTX protocol requests sent by tools and another to wait for target events. Whenever a target event occurs, the back end notifies the target server using the method specified in BKEND_INFO. The target server then queries the back end for information about the event and notifies the tools that have registered to receive notification of the event.

2. If a tool makes a request, the target server either satisfies the request itself (for example, by looking up a symbol in the host-resident symbol table) or forwards the request to the back end (for example, by requesting that the back end set a breakpoint or read memory).

Shutting Down the Target Server

1. When the target server exits, it calls the back-end disconnect routine to clean up all resources that have been allocated by the back end.

2. Next, the target server disconnects all tools and removes its name from the Tornado registry (wtxregd).

3. Finally, the target server exits.

2.1.2 Back-End Overview

The portion of the Tornado target server that communicates with the target is called the back end. Many back ends are supported, each dedicated to an alternative host-target connection. All back ends share a single purpose: to provide a set of primitive services for the target server core. Depending on the back end, executing these routines can be as simple as forwarding a request to the target agent (the WDB RPC back end) or as complex as translating the request to an entirely different protocol (an emulator back end).

WDB Back Ends

Some back ends (WDB back ends) are designed to communicate with the target agent supplied with VxWorks, which is called the WDB agent. (See Figure 2-1.) The WDB agent uses the Wind DeBug (WDB) protocol to communicate between the host and target operating system. The key advantage of this protocol, from a programming perspective, is that it corresponds one-to-one with the target server back-end API. There is a back-end function corresponding to each WDB protocol
request. When using the WDB agent, the back end manages the mechanics of transmitting the request to the target over whatever hardware medium provides host-target communication. Creating a new back end is a matter of writing the necessary host and target code to transport WDB protocol messages.2

Figure 2-1  WDB Back End

Non-WDB Back Ends

Other back ends (non-WDB back ends) are designed to communicate with alternative agents, such as an emulator that serves as a hardware debug agent. (See Figure 2-2). If a back end does not use the WDB agent, the back end itself must provide the requested service. The underlying service could be provided by another software agent (for example, a ROM monitor) or by a hardware agent (for example, an in-circuit emulator or ICE). Virtually any cross-development framework can be supported in a back end because the back-end API is general. It is also possible, if necessary, to return a “service not available” error or to fill in the TGT_OPS table with a NULL pointer if it is impossible to implement the service as requested by the target server.

This chapter provides background information on the structure of a target server back end, followed by descriptions of how to write new back ends for the WDB agent and for a non-WDB agent. Writing a new back end allows you to use a different communication protocol or to use a new method for providing services to the Tornado tools; none of the tools needs to be modified because the tools

2. The current implementation of this protocol is based on Sun Microsystems RPC 4.0, which can be implemented on virtually any transport layer.
communicate only with the target server. Our discussion provides an overall orientation; throughout this chapter, refer to the reference entries in Part 2 for details.

## 2.2 Back-End Implementation

This section discusses several aspects of back-end implementation that are similar for both WDB and non-WDB back ends.

- It provides a description of the **FLAG_DESC** data structure and the `bkendFlagsGet()` routine which is used to retrieve those flags so they can be sent to the back end.

- It also provides a prototype of the back-end initialization routine, with an explanation of the parameters passed to the back end by the target server and a description of the **TGT_OPS** and **BKEND_INFO** data structures (filled by the initialization routine).
This section also covers event notification requirements and methods, as well as issues relating to target-board rebooting.

In addition, message logging facilities provided for WDB back ends are discussed. These facilities may be of interest to writers of non-WDB back ends as a source of sample target-server target-agent interactions.

### 2.2.1 Attachment and Initialization

A target server can only use one back end at a time. The back end is attached when the target server is started. The `-B` option specifies which back end to attach. For example, to use the back end named `wdbserial`, select `wdbserial` in the target server initialization window, which invokes the target server with the following command:

```
% tgtsvr ... -B wdbserial ...
```

Back ends are linked with the target server dynamically using DLLs. At run time, the target server searches `installDir/host/hostType/lib/backend`, which contains a DLL for each available back end, and loads the specified back end. The back end can have any name, but we recommend you choose names that are short, lower case, and suggest the connection strategy to which they refer. If the requested back end is not found, the attachment fails and the target server abandons its start-up sequence, exiting with an error condition.

Once the correct DLL is found, the target server asks if the back end handles special flags with the `bkendFlagsGet()` routine. If the back end exports this routine, the target server calls the routine and appends the returned flags array to its own array. When it finishes parsing the command line, the target server calls an initialization routine that attaches the back end. Besides initializing the back end, this routine fills in a table of function pointers (the `TGT_OPS` structure) used by the target server to call the back-end functions and a structure (the `BKEND_INFO` structure) which describes how the back end notifies the target server of asynchronous events. Finally, the initialization routine calls `bkendTgtConnect()` to attach the target server to a specified target.

⚠️ **CAUTION:** The target server symbol table is not available until you attach the target.

---

3. On UNIX hosts and on Windows host when you use the command line, the environment variable `WIND_HOST_TYPE` records the host type. (For example, on a Sun-4 running Solaris 2.5.1, `WIND_HOST_TYPE` is `sun4-solaris2`; see the *Tornado User’s Guide: Getting Started.*)
The Flag Parsing Routine

A back end can export a flag-parsing routine to handle any back-end specific command line parameters. If the loaded back end exports such a routine, the target server runs the routine to get a list of the parameters the back end needs to access. The parameters are contained in the `FLAG_DESC` structure (defined in `installDir/host/include/flagutil.h`). The target server adds the flags to its own array of recognized flags and parses its entire command line.

⚠️ **CAUTION:** The target server overrides its own flag description if a back end defines the same flag. Back ends should use flags that are different from the target server core flags.

The name of the flag-parsing routine is standardized so that the target server can find it. It is composed of the back-end file name (without extension) plus the word `FlagsGet`. For our example back end, called `mybkend`, the flag-parsing routine is `mybkendFlagsGet()`, which is defined in `mybkend.c`.

The back-end flag-parsing routine must match the following prototype:

```c
FLAG_DESC * mybkendFlagsGet (void)
```

The returned value is a null-terminated `FLAG_DESC` structure array. This structure describes the arguments handled by the back end and is composed of five fields:

```c
typedef struct flag_desc /* target server flag descriptions */
{
    char * flagName, /* verbose flag name */
    char * flagTerseName, /* abbreviated name of flag (or NULL) */
    PARSE_RTN parseRoutine; /* flag processing routine */
    int outputPtr; /* where to store the output result */
    char * flagHelp; /* flag help string */
} FLAG_DESC;
```

When the target server parses its command line, it tries to match the current argument to the `flagname` or `flagtersename` field of each `FLAG_DESC` structure array element. If an element matches the command line argument, the target server executes the `parseRoutine` field of the matching element, giving three arguments: the remaining arguments number, the command line argument vector starting with the argument which caused the call to the routine, and the value stored in the `outputPtr` field. The provided routine interprets as many arguments as necessary, fills the provided location with the parsed value(s), and returns the number of parsed command line arguments. The parsing routine must match the following prototype:
/* flag processing routine definition */

typedef int (*PARSE_RTN) (int argc, char **argv, void * outputPtr)

Store your flagInt() routine in the libwpwr shared library and declared it in installDir/host/include/flagutil.h.

⚠️ CAUTION: The target server drops the number of arguments returned by the parseRoutine routine. So, if an error occurs during the routine parsing phase, the routine should return 0, so the target server can resynchronize itself on the next command line argument.

Example 2-1 Flag-parsing Routine

Suppose you want to define a back end which can take a serial line speed argument. This argument will be coded with -speed value on the command line. Your code includes declarations and mybkendFlagsGet() as follows:

```c
static int mySpeed = 9600; /* mybackend default speed */
static FLAG_DESC mybkendFlags /* my flags descriptions */
{
    "-speed", "-sp", flagInt, (void *) &mySpeed,
    "-sp[eed] myBkend Speed definition (default 9600)."},
    {NULL, NULL, NULL, 0, NULL} /* End Of Record delimiter */
};
```

```c
...
FLAG_DESC * mybkendFlagsGet (void)
{
    return (FLAG_DESC *) mybkendFlags;
}
```

Now, suppose you launch the target server with the following command line:

tgtsvr -B mybackend -speed 19200 targetName

When the target server loads mybkend, it calls mybkendFlagsGet(), appends the mybkendFlags array to its own FLAG_DESC array, and parses the command line. When it parses the -speed argument, it calls flagInt() with the first argument set to 3, the second argument pointing to the array "-speed 19200 targetName", and the last argument set to &mySpeed.

This routine interprets the 19200 string as an integer and stores that value at the &mySpeed location. Thus, the back end speed is modified by the command line parameter.
The Initialization Routine

When the target server attaches a back end, it calls the initialization routine of the named back end. The main purpose of this routine is to fill in the TGT_OPS structure defined in `installDir/host/include/tgtlib.h`, and the BKEND_INFO structure defined in `installDir/host/include/bkendlib.h`. It also performs whatever back-end initialization is appropriate.

The TGT_OPS structure, whose location is passed as an argument to the back-end initialization routine, is a collection of function pointers. The initialization routine fills this structure with pointers to functions that provide the back-end services. The implementation of the services is entirely back-end dependent. By virtue of the coupling, it is possible to support many back ends, each providing the same service in its own way. Once the TGT_OPS structure is filled in and returned, the target server is ready to start communicating with the target.

The BKEND_INFO structure, whose location is also passed as an argument to the back-end initialization routine, contains information about the newly loaded back end. This information consists of the back-end version and the asynchronous events notification. Once the BKEND_INFO structure is filled, the target server launches a thread which handles the asynchronous events according to the given method.

The name of the initialization routine is standardized so that the target server can find and attach the back end chosen by the user. It is composed of the back-end file name (without extension) plus the word Initialize. For our example back end, called mybkend, the initialization routine is `mybkendInitialize()`, which is defined in `mybkend.c`. This name indicates the DLL that the target server loads as well as the back-end initialization routine to call. The back-end name is passed to the target server by the -B option when the target server is started.

The back-end initialization routine must match the following prototype:

```c
STATUS mybkendInitialize
{
    char * tgtName, /* network unique target name to connect */
    TGT_OPS * pTgtOps /* vector of back end functions to fill */
    BKEND_INFO * pBkendInfo /* Backend notification method */
}
```

The first parameter, tgtName, passed to the back-end initialization routine from the target server specifies the target name. The meaning of the parameter is completely back-end dependent. For example, the wdbrpc back end uses the tgtName as the target’s IP address while the wdbserial back end doesn’t use tgtName. The only restriction is that tgtName must be unique for each target server.
The TGT_OPS Structure

The second parameter passed to the back-end initialization routine, pTgtOps, is a pointer to the TGT_OPS structure (defined in installDir/host/include/tgtlib.h), which declares the target server back-end interface. The initialization routine must fill this structure with the appropriate information and function pointers so that the target server understands what the back end can do. When the target server needs to perform a service, it calls the appropriate back-end function.

typedef struct tgt_ops /* target server back end operations */
{
  BOOL            tgtConnected;     /* TRUE if connected to Target */
  TGT_LINK_DESC   tgtLink;          /* link descriptor */
  u_int           tgtSupInfo;       /* additional information */
  int             tgtEventFd;       /* not used. Use BKEND_INFO structure*/
  /* back end routines pointers */
  UINT32         (*tgtPingRtn)          (void);
  UINT32         (*tgtConnectRtn)       (WDB_TGT_INFO *);
  UINT32         (*tgtDisconnectRtn)    (void);
  UINT32         (*tgtModeSetRtn)       (UINT32 *);
  ...
} TGT_OPS;

The tgtConnected field is a boolean variable that is TRUE if the target board is connected to the target server by the back end. This boolean must be set to FALSE by the initialization routine because the board is not yet connected. It is set to TRUE by the target server if the target-board connection succeeds.

The tgtLink field is a TGT_LINK_DESC structure also defined in installDir/host/include/tgtlib.h. This structure describes the communication link with the board and is composed of three fields:

typedef struct
{
  char *     name;            /* target/host link name */
  u_int      type;            /* target/host link type */
  u_int      speed;           /* target/host link speed in bps */
} TGT_LINK_DESC;

The tgtSupInfo field is a target server spare field.

The tgtEventFd field obsolete. The file descriptor that the back end uses to wake up the target server when an asynchronous event arrives from the target system is now stored in the BKEND_INFO structure. The back end should set tgtEventFd to NONE.
The `tgtPingRtn` field is the first of many back-end function pointers. Each function is a service provided by the back end. The complete description of each function is given in the online reference material under Tornado API Reference>Target Server Back End Interface. (Note that the function names are similar to the pointer names, but are prefixed with `bkend`. All function pointers of this structure must be filled in. If a back-end service is not provided, set the pointer to `NULL`.

The **BKEND_INFO Structure**

The last parameter passed to the back-end initialization routine, `pbkendInfo`, is a pointer to a `BKEND_INFO` structure (defined in `installDir/host/include/bkendlib.h`). This structure describes information related to the currently attached back end. The key item is the asynchronous notification method provided by the back end.

```c
typedef struct bkEndInfo {
    int tgtBkInfoSize; /* size of this structure */
    int tgtBkVersion; /* Version of BackEnd Hiword = Major;
                      Loword = Minor*/
    union {
        struct _pollingMethod {
            int bkEndPollingMethod, /* polling method used :
                * POLL_NONE_MODE
                * POLL_SELECT_MODE
                * POLL_BKROUTINE_MODE
            */
        } POLLING_METHOD;
        union _bkEndNotifMethod {
            int pollingFd; /* polling file desc for POLL_SELECT_MODE */
            int (*bkEndSelectRtn)(int timeout) /* bkEnd select routine */
        } BKEND_NOTIF_METHOD;
    } INFO;
} BKEND_INFO;
```

The target server initializes `BKEND_INFO` with the following values:

```c
bkendinfo.tgtBkInfoSize = sizeof (bkend_info);
bkendinfo.tgtBkVersion = BKEND_VERSION_2
```
The *tgtBkInfoSize* field is set by the target server to the size of its *BKEND_INFO* structure. The *tgtBkVersion* field is set to *BKEND_VERSION_2* (defined in *installDir/host/include/bkendlib.h*) for the current version. The back end should check these values and reset *tgtBkVersion* if necessary.

During the initialization phase, the back end fills in the structure with its method of notifying the target server of asynchronous events. An excerpt from the WDB serial back-end initialization routine is as follows:

```c
... 

pbkInfo->tgtBkVersion = BKEND_VERSION_2; /* Backend version 2 */

/* we'll provide our polling method ( on NT ) */

#ifdef WIN32
pbkInfo->INFO.POLLING_METHOD.bkEndPollingMethod = POLL_BKROUTINE_MODE;
pbkInfo->INFO.POLLING_METHOD.BKEND_NOTIF_METHOD.bkEndSelectRtn = win32SerialSelect;
#else
/* on UNIX we give the tty file desc */

pbkInfo->INFO.POLLING_METHOD.bkEndPollingMethod = POLL_SELECT_MODE;
pbkInfo->INFO.POLLING_METHOD.BKEND_NOTIF_METHOD.pollingFd = EventFd;
#endif

return (OK);
```

### 2.2.2 Back-End Functions

After attaching and initializing the back end, the target server is able to perform actions on the connected board by calling back-end functions. The syntax of all back-end functions is similar. Functions usually take two pointers as arguments: the first points to a structure that contains parameters specifying the service input data; the second points to a structure that is filled in with the data returned by the service. One or both structure pointers may be omitted when a service requires no input or output data. Each function must return a WDB status code: either *WDB_OK* on success, or an appropriate WDB error code describing the error encountered during the service call. The complete error code list is located in the file *installDir/share/src/agents/wdb/wdb.h*.

Because the connection to the back-end interface is indirect, through a table of function pointers initialized when the back end is attached, there are no naming restrictions on back-end functions. Their scope is local to the back end, as all other modules access the back end through the function table. A complete description of
the function interface is provided in the online reference material under Tornado API Reference>Target Server Back End Interface.

2.2.3 Event Notification

The target server not only sends requests and information to the target, it also needs feedback from the target about what is occurring there. When a target event (such as hitting a breakpoint or an exception) occurs, the target server must be notified. The back-end interface provides two notification methods, synchronous and asynchronous. The notification method is provided to the target server by the BKEND_INFO structure pointer.

Three methods can be provided:

POLL_NONE_MODE:
No asynchronous mechanism is provided. The back end cannot handle asynchronous events. When this method is specified, no asynchronous events are sent to the target server. The target-server back-end thread uses synchronous requests to get events from the target. After each WTX request, it issues a bkendEventGet() call to get any events from the target.

POLL_SELECT_MODE:
The back end provides a file descriptor which can be put into a select() routine. When this method is specified, the thread handling the events waits on this descriptor using select(). When this file descriptor becomes active because the emulator or target agent has written to it, the back-end thread gets the events.

POLL_BKROUTINE_MODE:
The back end provides a routine which behaves like the select() routine. When this method is specified, the back-end thread pends on the provided routine until the routine returns something positive. When this occurs, the back-end thread receives the pending events.

With the two asynchronous methods, the target server calls the back end bkendEventGet() function only to get the event information. Those methods are more efficient because no polling for events is required.

After issuing a bkendEventGet() command, the target server calls bkendEvtPending() to see if other events are pending, and calls bkendEventGet() again if bkendEvtPending() indicates that other events are available to be read. This method allows the target server to upload multiple target events without having to enter the select loop each time.
2.2.4 Designing a select-like routine

The target server is a multi-threaded application. This allows a back end to export its own select-like routine instead of being limited to a selectable file descriptor. This is particularly useful on WIN32 hosts where select() addresses only sockets and not other handles such as serial lines or files. The select-like routine can be expanded to address such devices. It allows the target server back-end thread to sleep (thus not consuming CPU cycles) until events arrive. This routine must match the following prototype:

```
int bkendSelect (int timeout);
```

The timeout given by the target server is always WAIT_FOREVER (-1).

The return value should follow the following convention:

**Positive value**

The target server has something to read. It calls the bkendEventGet() and bkendEvtPending() routines to purge all the pending events.

0

Timeout, or another thread is taking the back end. The target server back-end thread waits until the actual thread dealing with the back end finishes its transaction, and pick up the resulting events afterward by calling bkendEvtPending() and bkendEventGet() if events are pending.

**Negative value**

An error occurred. The back-end thread terminates.

It is possible that two threads may use the back ends routines at the same time: a synchronous thread (the one servicing a WTX request), which can call all the routines given in the TGT_OPS structure, and the asynchronous thread (the one which is pending on the given select-like routine). You will have to handle the race condition here. The simplest way is to make the back-end thread return from the select-like routine with a 0 status. This causes the target server to wait until the other thread completes its transaction before asking if there are pending events.

For a discussion of how to inform the target server that a select-like routine is used, see 2.2.1 Attachment and Initialization, p.20.

2.2.5 Handling Target-Board Reboots

To keep the target server synchronized with the target agent during a target reboot, the back end must call the routine targetServerRestart(). A reboot detection
mechanism is not optional, and the target server does not function properly if the back end fails to call this routine.

2.2.6 Cleaning Up the Tornado Registry

If the target server does not exit gracefully, which usually occurs because there is an error in the back end, it leaves its name in the registry. This prevents you from reusing the same name when you restart the target server. On UNIX, clean up the registry using the “unregister” button of the Tornado launcher. On Windows, select Target Server>Manage on the Tools menu and select Unregister. On either host you can create a \texttt{wtxtcl} script, \texttt{unreg}, like the following:

```bash
#!/bin/sh
#
# unreg - remove a name from the registry
#
# Syntax: unreg <targetServerName>
#
# echo -n Unregistering $1 ...

wtxtcl << EOF
set seekName $1
set allNames [wtxInfo]

set pos [lsearch -glob \$allNames \*$seekName* ]
if { \$pos < 0} {
    puts "invalid name"
    exit
}

set fullName [lindex [lindex \$allNames \$pos ] 0 ]
wtxUnregister \$fullName
EOF

echo ' done.'
```

This is an example of the helpful utilities you can create with the \texttt{wtxtcl} shell. For more information on WTX Tcl, see 4. The WTX Protocol, the Tornado User’s Guide: Tcl Appendix, and the online reference material under Tornado API Reference>WTX TCL Library.
2.2.7 Message Logging

A message logging facility exists for all WDB back ends. This facility logs all transactions between the back end and the target agent to a file or terminal. Message logging simplifies diagnosing connection problems between the host and target because one can look at the sequence of transactions that led to the problem. This facility can also be used to help new back-end designers understand requests exchanged between existing back ends and the target.

All back ends that connect to the WDB agent send requests to the target using the library `installDir/host/src/tgtsvr/backend/share/rpccore.c`, which contains the logging calls. This logging mechanism is back-end-specific, and is only provided for WDB back ends.

Message logging is enabled when the target server is started with the `-Bd fileName` option. For example, to save log information in the file named `/tmp/WDB.log`, invoke the target server with a command like the following:

```
% tgtsvr ... -Bd /tmp/tgtDebug.log ...
```

The file name must be specified; otherwise message logging is not enabled. If the file already exists, log information is added at the end of the file. Each time a request is logged to the file, output is flushed to assure that the last request written in the log file is actually the last request sent even if the target server hangs.

The length of the debug file can be controlled by the `-Bm logMaxSize` option. With this flag, a file is created, or reset if it already exists, and is written as a circular file: when the file length reaches `logMaxSize`, the file is rewritten from the beginning, overriding the existing data. If this flag is not set or set with a 0 value, a file is created, or opened in append mode if it already exists, and is truncated.

⚠️ CAUTION: Log files can become huge. An average of 100 bytes is used for each request logged; it is not uncommon to have a log file larger than a megabyte. Be sure to allocate enough disk space before starting the target server without the `-Bm` option. Also be aware that logging affects the performance of the back end.

Each log file starts with a header, followed by records of transactions with the back end. The header provides:

- the target server user name
- the time and date the target server was launched
- the names of the target server and the target
- the request timeout value
- the number of times the server retransmits a request when no response is received.
The following is an example of this header:

```
User Name : wrs
Target Server Name : target@couesnon
Target Name : target
Target Server Options : tgtsvr -V target -Bd /tmp/WDB.log
Timeout value : 1 second(s)
Request re-send Max : 3
```

Each request log is made of two parts: the service requested and the reply. The service-requested information includes the request number, the service name, and the input-structure name with the name and value of each field. The request number is a 16-bit integer assigned to distinguish each request. When the upper limit is reached, the request number restarts from zero. The input-structure name and values are omitted when the service does not require input arguments. A service input structure is signaled by the word **In** placed before its name.

The reply log consists of three parts: the number of times the request was resent, the service status, and the reply-structure name with the name and value of each field. The service status value is one of the WDB error codes. The file **wdb.h** located in the `installDir/share/src/agents/wdb` directory provides the complete error code list. As in the input case, the output structure is signaled by the word **Out** before its name and it is omitted when the reply has no return value.

An example of a request log is given below. In this example the target server performs a checksum on a block of 49788 bytes of target memory starting at address 0x2000. The return status is OK and the checksum value is 0xffff34de.

```
2 2       WDB_MEM_CHECKSUM Wed Jun 17 16:38:19 1998
In      WDB_MEM_REGION
    baseAddr        0x20000
    numBytes        49788
    param 0
3       Out status: 0k
    UINT32 0xffff34de
```

### 2.2.8 Informing Windows About a New Back End

Windows does not automatically recognize that a given back end is available. To make a new back end accessible to the Configure pop-up window under the Target Server option of the Tools menu, it is necessary to edit the file:

```
installDir/host/resource/tcl/app-config/Tornado/01TargetServer.win32.tcl
```
Change the line:

```plaintext
set tsCfg_backendList {wdbrpc wdbserial netrom}
```

to

```plaintext
set tsCfg_backendList {wdbrpc wdbserial netrom mybackend}
```

Back-end-specific flags must be supplied in the Target Server Launch command edit box, since Tornado can not automatically know how many and what kind of arguments the back end supports.

### 2.3 Writing a New WDB Back End

The back ends provided by Wind River Systems support a variety of communication alternatives, but not all cases are supported. Customers interested in using an unsupported communication link must write a new back end.

#### 2.3.1 Overview of Writing a WDB Back End

Developing a back end to communicate with the WDB agent is simple: because all the services are already implemented in the WDB agent, the new back end only needs to transport WDB RPC messages.

The key development steps are:

- Write a back end which utilizes RPC core functions, if necessary. For a new machine, you must write `clnt_xxx.c`.
- Write a device driver on the target, if necessary.
2.3.2 WDB Protocol

This section describes the Wind DeBug (WDB) version 1.0 protocol. It is the protocol used by the Tornado target server back ends that communicate with the WDB agent (WDB RPC, WDB Serial, and NetROM). See Figure 2-1 for a diagram of how these back ends connect to the WDB agent.

NOTE: This section is provided for completeness only. If you want to create a new back end for Tornado, knowledge of the WDB protocol is not needed because the support libraries handle all aspects of the protocol not related to transport. In fact, in order to assure that your back end is not affected by changes in WDB, you should gain access to WDB facilities through the support libraries as shown in the example back ends. To create a non-WDB back end that communicates with a device such as an ICE or a ROM monitor, you also need to implement all the target services required. For details, refer to 2.4 Writing a Non-WDB Back End, p.47.

Protocol Overview

WDB is an RPC-based protocol which uses UDP. The WDB agent acts as a server for requests sent by the target server. For more information, see one of the many generally available references on UDP and Sun RPC.

See installDir/share/src/agents/wdb/wdbP.h for some key definitions:
- The UDP port that the agent accepts requests from, WDBPORT, is 0x4321.
- The agent’s RPC program number, WDBPROG, is 0x55555555.
- The agent’s RPC version number, WDBVERS, is 1.

Requests Sent to the Agent

A WDB request packet sent to the agent contains the following parts, as shown in Figure 2-3:
- A 20-byte IP header.
- An 8-byte UDP header. It consists of four 16-bit words. WDBPORT is word 2.
- A 40-byte RPC request header. It consists of ten 32-bit words. WDBPROG is word 3. WDBVERS is word 4. The RPC procedure number is word 5.
An XDR stream. The first 12 bytes of this stream is WDB-parameter-wrapper information. The rest of the stream is the XDR-encoded parameters for the procedure. (See `installDir/share/src/agents/wdb` for more information on the `WDB_XDR` functions.)

The WDB parameter wrapper contains three 4-byte words. The first word is a checksum over the whole RPC packet (RPC header plus XDR stream). The second word is the packet size. These two words enable the agent to determine if a corrupted packet has arrived. The third word is a sequence number. The low order two bytes of the sequence number are used to allow the agent to ignore old or duplicated requests (which can occur with UDP transport). The high order two bytes are the `host ID`. When a host issues a
The Target Server Back End

**WDB_TARGET_CONNECT** request, the host ID portion of the sequence number is recorded. If a request arrives from a non-connected host, the RPC fails with the RPC error status **PROG_UNA VAIL** or **SYSTEM_ERR**, depending on whether the agent is already connected to another target server or not. The **WDB_TARGET_CONNECT** request always establishes a new connection, if necessary by dropping the old one. If the host wants to test whether or not the agent is already connected before trying to establish a connection, it should first issue a **WDB_TARGET_PING** request and see if the RPC fails with error status **PROG_UNA VAIL**. If so the connection is busy.

The routine **xdr_WDB_PARAM_WRAPPER** is used to encode or decode the entire XDR stream (the procedure parameters plus the 12-byte parameter wrapper). The following example is a code stub from the host routine **rpcCoreClntCall()**:

```c
seqNumber++;
...
wdbParamWrapper.xdr = inProc;   /* xdr func for proc params */
wdbParamWrapper.pParams = in;   /* pointer to proc params */
wdbParamWrapper.seqNum = processId | seqNumber;   /* seq nb */
...
clintStatus = clnt_call (pWdbClnt, procNum, xdr_WDB_PARAM_WRAPPER,
&wdbParamWrapper, ...);
```

**Replies Sent by the Agent**

A WDB reply packet sent by the agent contains the following parts, as shown in Figure 2-4:

- A 20-byte IP header.
- An 8-byte UDP header.
- A 24-byte RPC reply header.
- An XDR stream. The first 12 bytes of this stream is WDB-reply-wrapper information. The rest of the stream is the XDR-encoded reply from the procedure.

Like the WDB parameter wrapper, the WDB reply wrapper contains three 4-byte words. The first word is a checksum over the whole RPC packet (RPC header plus XDR stream). The second word is the packet size. These two words enable the host to determine if a corrupted reply was returned (and, if so, to reissue the request). The third word is the WDB error status. The high order bit is set if there are events pending on the target, in which case the host
can issue a `WDB_EVENT_GET` request to upload the event. The rest of the word is the actual error status.

**Figure 2-4  WDB Reply Packet**

<table>
<thead>
<tr>
<th>IP HEADER</th>
<th>20 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP Header</td>
<td>8 bytes</td>
</tr>
<tr>
<td>RPC Reply Header</td>
<td>24 bytes (6 32-bit words)</td>
</tr>
<tr>
<td>XDR Stream</td>
<td></td>
</tr>
<tr>
<td>XDR encoded reply</td>
<td></td>
</tr>
</tbody>
</table>

**Reply Errors**

After a remote procedure is called, the program should perform error checking. Error status can be communicated in one of two ways: in the RPC header or in the reply wrapper. If the failure is reported in the third word of the RPC header, then the host’s `clnt_call` returns an RPC error status. These have conventional meanings according to the RPC specification. In addition, the WDB agent uses a couple of special codes:
RPC_PROCUNAVAIL
   The RPC procedure requested is not configured into the agent. The agent
   is scalable, so this usually means the agent was built without that service.

RPC_PROGUNAVAIL
   The agent refused to execute the procedure because it is connected to
   another host.

RPC_SYSTEMERROR
   The agent refused to execute the procedure because it is not connected to
   any host. If you were previously connected, the target has rebooted.

Even if the RPC call succeeds (meaning that the agent tries to execute your
command), the command may still fail. The errCode field in the reply wrapper can
be checked; if the lower 31 bits are zero, the command succeeded. (Remember that
the high order bit is set if there are events pending on the target.) If the value is
non-zero, then the procedure failed and the word contains the error code. Error
codes have the format WDB_ERR_XXX. The error-code definitions are located in
installDir/share/src/agents/wdb/wdb.h.

The routine xdr_WDB_REPLY_WRAPPER is used to encode or decode the entire
XDR stream (the reply data plus the 12-byte reply wrapper). The following pseudo
code shows how it works:

```c
wdbReplyWrapper.xdr = outProc; /* reply xdr function */
wdbReplyWrapper.pReply = out; /* where to decode reply */
wdbReplyWrapper.errCode = OK; /* just to clear this field */
```

```
... clntStatus = clnt_call (pWdbClnt, procNum, xdr_WDB_PARAM_WRAPPER,
&wdbParamWrapper, xdr_WDB_REPLY_WRAPPER,
&wdbReplyWrapper, timeout); check (clntStatus)
{
   if (RPC_TIMEDOUT or RPC_CANTDECODERES or RPC_CANTDECODEARGS)
      try again
   if (RPC_SYSTEMERROR)
      if we were previously connected, then target must have rebooted so resync
      and reconnect.
   if (RPC_PROCUNAVAIL)
      procedure not configured into agent. Try to rebuild the agent with that
      facility included (e.g., virtual I/O is an optional agent facility).
   if (RPC_SUCCESS)
      agent tried to execute the routine.
      check high order bit of wdbReplyWrapper.errCode to see if events are
      pending on the target. If so, execute a WDB_EVENT_GET request after
      finishing processing this reply.
      mask off the high order bit of wdbReplyWrapper.errCode.
      if (wdbReplyWrapper.errCode == 0)
         success! In this case wdbReplyWrapper.xdr decoded the reply and put
         it in wdbReplyWrapper.pReply.
      else
```
Asynchronous Notification Sent By the Agent

Asynchronous events can be generated on the target. These include exceptions, breakpoints, and task exiting. These events are queued on the target until the host uploads them with the **WDB_EVENT_GET** service. In order to prevent the host from polling for events, the agent has two ways to notify the host that events are pending: (1) by setting the high order bit in the **errCode** status of the reply wrapper; (2) by sending a *notify packet*.

Normally the agent only sends data to the host in response to RPC requests. The convention is that if the host receives data when it is not waiting for a reply, it means that an event is pending on the target. This allows the target server to *select()* on file descriptors associated with the host tools which are connected as well as with the target. If the target file descriptor becomes active, the host issues a **WDB_EVENT_GET** request to upload the event (and keeps uploading events until the high order bit in the **errCode** field is clear). The actual notify packet sent by the agent is a packet that can not be confused with an RPC reply (in case it sends the notify packet just as the host issues an RPC request). In fact, it sends a bogus RPC request.

2.3.3 Host-Side Code

To provide the necessary host-side support for a new communication pathway, you must write a new back-end DLL to transport WDB protocol messages. Fortunately, most of the back-end code is generic RPC code to transport WDB protocol messages; thus you can reuse Wind River’s **rpccore** library. Consequently, you can only need to write the back end’s initialization code and, if necessary, the client-side RPC implementation.

A WDB back end consists of three parts: (1) the initialization routine, which initializes the back end; (2) the RPC core which manages WDB RPC requests, including XDR; and (3) the client-side RPC implementation, which sends and receives the RPC messages over the network medium. (See Figure 2-5.)

---

4. For more information on RPC, please see the O’Reilley book, *Power Programming with RPC*.  
5. The **rpccore** library can be obtained when you request the Back-End Developer’s Kit.
The `bkendInit()` routine is the back-end DLL’s entry point and must initialize the back end. To do this, it performs the following services:

1. It creates the `CLIENT` data structure, which the RPC core uses to communicate with the WDB agent. The `CLIENT` structure is a standard RPC data structure that contains information needed by the RPC core, such as pointers to buffers and to the functions for sending and receiving data. It is important to create a `CLIENT` structure appropriate to the network medium your back end is using: for example, Wind River’s `wdbrpc` back end calls `clnt_udpccreate()` so that the back end can send UDP messages over the network; the `wdbserial` back end calls `clnt_ttccreate()` so that the back end can communicate over a serial device.

2. It initializes the `TGT_LINK_DESC` information in the `TGT_OPS` structure.
3. It calls \texttt{rpcCoreInit()} to handle the rest of the back-end initialization. This routine is part of the \texttt{installDir/host/src/tgtsvr/backend/share/rpccore.c} support library, which handles all the generic parts of the protocol including:
   - translating back-end requests into WDB-agent requests, including XDR encoding and decoding
   - packet sequencing and checksum
   - timeout and retransmission
   - asynchronous event notification
   - back-end logging

In the following sections, we examine the \texttt{wdbserial} back end to demonstrate how to write a new WDB back end. The \texttt{wdbserial} back end consists of two main modules, the \texttt{wdbserial.c} module, which implements the back-end initialization routine, and \texttt{clnt_tty.c}, which provides the client-side RPC implementation for a serial device (see \textit{Client-Side RPC Implementation}, p.43).

\textbf{Back-End Initialization: wdbserial.c}

The \texttt{wdbserial.c} module consists of one routine, \texttt{wdbserialInit()}. The target server invokes \texttt{wdbserialInit()} after it loads the back end in order to initialize it. The \texttt{wdbserialInit()} call creates an RPC \texttt{CLIENT} structure for communicating over a serial device and establishes a link, initializes the generic RPC core library to operate the target agent using the WDB (Wind DeBug) protocol, and initializes a data structure describing the back-end link with the agent. The \texttt{wdbserial.c} code shows how these steps are carried out.

```c
/* wdbserial.c - Remote Procedure Call (RPC) backend library */

After the usual preamble of comments, copyright notice, and inclusion of system header files, \texttt{wdbserial.c} includes the Tornado host header files from the \texttt{installDir/host/include} directory:

```...```
Target Server Back End

```c
#include "wdb.h"
#include "wdbp.h"
#include "wpwrutil.h"
#include "bkendlib.h"
#include "bkendlog.h"

Windows '95 and NT hosts must include `backend.h` by means of the following #ifdef:

```c
#ifdef WIN32
#include "backend.h"
extern int dbg_on;
#endif /* WIN32 */
```

Next, `wdbserial.c` imports the prototypes to `rpcCoreInit()`, which initializes the RPC core library, and `clnttty_create()`, which creates an RPC connection over a serial link.

```c
extern STATUS rpcCoreInit (CLIENT *, u_int, u_int, TGT_OPS *);
extern CLIENT * clnttty_create (char *, int, u_long, u_long, struct timeval);
```

When the target server calls `wdbserialInitialize()`, it passes pointers to the `TGT_OPS` and `BKEND_INFO` structures. (For more information on these structures, see `The TGT_OPS Structure`, p.24 and `The BKEND_INFO Structure`, p.25.)

```c
STATUS wdbserialInitialize

char * tgtName, /* target name to connect to (unused) */
TGT_OPS * pTgtOps /* back-end function */
BKEND_INFO * pBkendInfo /* Backend notification method */
}
```

Windows hosts optionally enable logging of debugging messages from the lower-level serial support:

```c
#ifdef WIN32
dbg_on = GetDebugFlag();
#endif /* WIN32 */
```

Next, the back end must enable RPC communications over a serial link by creating the `CLIENT` data structure. If your back end uses an unsupported link type you will need to implement a `clntXXX_create()` call for your communication medium. Wind River implemented `clnttty_create()` for the `wdbserial` back end.

The back end initializes the RPC client-side transport to the WDB agent: first, `struct timeval` is initialized with the user’s specified timeout; then `clnttty_create()` is called to create the RPC link to the target. Note that the `clnttty_create()` call is retried until it succeeds or exceeds the user-specified number of retries.
/* set the connection timeout to the current value */

    tv.tv_sec = timeout;
    tv.tv_usec = 0;
    ...
    resendCnt = recallNum;

do
{
    /* create the backend client and connect the target daemon */

    pClnt = clntttty_create (pTtyDevName, baudRate, WDBPROG, WDBVERS, tv);
}
    while ((pClnt == NULL) && (--resendCnt > 0));
    ...

If the RPC client-side initialization fails, the back end sets errno to the appropriate error number, logs an error message, and returns ERROR to indicate that the called routine failed.

    if (pClnt == NULL)
    {
        ...
        errno = WTX_ERR_SVR_INVALID_DEVICE;

        WPWR_LOG_ERR ("%s\n",
                       clnt_spcreateerror ("wdbserial backend client create");
        return (ERROR);
    }

If the RPC client initialization succeeds, the back end calls rpcCoreInit() to initialize the rpccore library for use over the serial device. The rpccore library provides all the support necessary to operate the WDB target agent in response to target server requests. In particular, rpcCoreInit() initializes the TGT_OPS structure, specifying that the target server should call rpccore routines to perform the various back-end operations.

    rpcCoreInit (pClnt, timeout, recallNum, pTgtOps);
    ...

Finally, the back end describes the target link type. If your back end uses a new link type, define a TGT_LINK_XXX macro in installDir/host/h/tgtlib.h.

    pTgtOps->tgtLink.name = "WDB Agent across serial line";
    pTgtOps->tgtLink.type = TGT_LINK_SERIAL_RPC;
    pTgtOps->tgtLink.speed = baudRate;

    return (OK);
}
Client-Side RPC Implementation

If your back end must communicate over an unsupported network medium, you must provide a client-side RPC implementation. As a starting point, you should use the clnt_udp.c which is part of Sun Microsystem’s public domain RPC distribution and is provided in installDir/target/unsupported/rpc4.0/rpc.

Your implementation must provide the following services:

- A clntXXX_create() routine (where XXX describes the device type you are using): this routine must open the device for communication and return a pointer to a CLIENT data structure. The CLIENT structure should contain the information needed to perform RPC, including pointers to the clnt_ops functions for managing the transmission of data and data such as file descriptors.
- The clnt_ops functions: these functions perform the basic RPC operations over a specific network medium.

NOTE: The client-side RPC implementation must transmit datagrams in a UDP-like manner. In other words, the client-side RPC must reliably transmit an entire datagram. If data is lost, the back end can drop the datagram and RPC repeats the request. Consequently, if the network medium is character-oriented, like a serial device, the back end must packetize datagrams on both the host and target sides (see 2.3.4 Target-Side Code, p.45). One way of doing this is to use the SLIP protocol, as Wind River does in the wdbserial back end.

Building a Back End for UNIX Hosts

By including the generic makefile templates provided with Tornado, it is easy to write a makefile for a back end.

# Makefile - for WDB serial backend

After the modification history, specify the suffixes of the shared library that will be built:

.SUFFIXES: .so .sl

Next, include the makefile templates provided by the Tornado development environment. These fragments make it easy to build a portable makefile.

```bash
include $(WIND_BASE)/host/include/make/generic.mh
include $(WIND_BASE)/host/include/make/$(HOST).mh
```
Set the `INCLUDES` macro to specify that the compiler can find header files in the `installDir/host/include` and the `installDir/share/src/agents/wdb` directories.

Specify any other directories that the compiler needs to search to find your back-end header files:

```
INCLUDES = $(WIND_INC) $(WIND_SHARE_INC)
```

Specify which modules should be linked to create the back end. List both the back end modules and the `rpccore` modules (the modules in `./share`):

```
BKEND_OBJS = clnt_tty.o wdbserial.o

BKEND_XDR_OBJS = ../share/ctx.o ../share/ctxcreat.o \\
                 ../share/ctxstep.o ../share/evtdata.o \\
                 ../share/evtpoint.o ../share/memory.o \\
                 ../share/regs.o ../share/rpccksum.o \\
                 ../share/rpccore.o ../share/tgtinfo.o \\
                 ../share/wrapper.o ../share/xdrcore.o
```

State the name of the back end:

```
SH_BKEND_OBJS = wdbserial.$(SHLIB_EXT)
```

Next, specify any extra compiler flags you need:

```
LOCAL_CFLAGS = -DPORTABLE -DHOST $(DYN_LK_FLAGS)
```

Finally, an inference rule states that the back end is built from C source modules that are linked into the shared library. If you need to link the back end with other libraries or are using C++, you must modify this rule.

```
.c.$(SHLIB_EXT) : $(BKEND_OBJS) $(BKEND_XDR_OBJS) \\
$(SHARED_LD) $(SHARED_LDFLAGS) -o $(SH_BKEND_LIB)/$*.$(SHLIB_EXT) \\
$(BKEND_OBJS) $(BKEND_XDR_OBJS)
```

```
include $(WIND_BASE)/host/include/make/generic2.mh
```

**Building a Back End for Windows Hosts**

**Using Visual C++ 5.0**

To build a back end for Windows hosts, you need to create a project for the back end using Microsoft’s Visual C++. Because the target server was built with Visual C++ 5.0, we recommend that developers also use Visual C++ 5.0 to avoid incompatibilities between different versions of the standard libraries. Remember to address the following build issues:
- Define the macros `HOST` and `WIN32` so that Wind River header files have the proper configuration for the Windows host.
- Add `installDir/host/include` and `installDir/share/src/agents/wdb` to the list of directories to search for include files.
- Link with:
  - `wsck32.lib` to provide the necessary socket support for RPC.
  - `installDir/host/x86-win32/lib/backend.lib` to pull in back-end support facilities.

### 2.3.4 Target-Side Code

The WDB target agent needs a means to send and receive UDP/IP datagrams over the physical network connection. There are two protocol stacks that can be used, the full VxWorks network protocol stack and a lightweight UDP/IP interpreter. The full network protocol stack provides a rich set of functionality, while the lightweight UDP/IP interpreter requires much less target memory. The choice of protocol stacks affects the type of driver that must be written for the physical device.

![Driver Type for Different Protocol Stacks](image)
Writing a Network Interface Driver

Drivers that interface with the VxWorks TCP/IP network stack are called network interface drivers. Details of how to write a network interface driver are covered in The BSP Porting Kit (an optional product). The advantage of creating such a driver is that, in addition to being used as a debug communication path, it can also be used for application network communication. To use a network interface driver, configure the target agent for network communication (the default configuration).

Writing a WDB Packet Driver

Drivers that interface with the target agent’s UDP/IP interpreter are called WDB packet drivers. Such drivers have the advantage that they do not require the TCP/IP stack to be present on the target. This can save space on resource-constrained targets. The agent’s UDP/IP interpreter has the advantage of small size (only 800 bytes) but it also has limited functionality.

To create a WDB packet driver, start with the template driver in installDir/target/src/drv/wdb/wdbTemplatePktDrv.c. Use of the template is documented in the source file.

You must also modify the agent’s startup code to initialize the new communication pathway. The target agent configuration code is provided in installDir/target/src/config/usrWdb.c. To initialize your custom packet driver, add initialization code to usrWdb.c similar to the section bracketed with 

Build Issues

To build the packet driver, copy it into your BSP directory and use the standard techniques described in the Tornado User’s Guide: Projects or the VxWorks Programmer’s Guide: Configuration and Build. The makefile templates provided in installDir/target/h/make assist in developing a portable makefile. Modify the makefile in your BSP directory so that these modules are built and linked into your VxWorks image.
2.3.5 Testing

Use the wtxtest test suite to verify the new back-end connection. This facility exercises all back-end functions over your new link. For more information about wtxtest, see the online reference material under Tornado Reference>Tornado Tools.

NOTE: wtxtest is available in the Tornado distribution but it is not installed by default. If you want to use it, you must install it using SETUP. In the Select Products dialog box, highlight the appropriate Tornado object and click Details. Check Tornado WTX Test Suite in the Select Parts dialog box and complete your installation.

2.4 Writing a Non-WDB Back End

This section addresses the task of writing a back end that communicates with an agent (such as an emulator) other than the WDB agent. In this case, your back end methods must service back-end API requests directly, in other words, by way of a proprietary emulator API. For a general approach highlighting some of the issues involved in this process, see 2.4.1 Overview of Writing a Non-WDB Back End, p.48. The remainder of this chapter presents an example implementation. The complete source code for the acecpu32 example back end plus a back-end development class library, which implements the back-end framework as a C++ abstract base class, are provided in the Back-End Developer’s Kit.

2.4.1 Overview of Writing a Non-WDB Back End

The following tasks are involved in writing a non-WDB back end:

- writing the initialization routine
- writing the back-end methods to provide basic run control
- adding advanced features (such as target function invocations)
- testing the back end

The initialization routine is the entry point of the back end. It performs the back-end initialization and attachment. Even before the rest of the back-end functions are implemented, this routine can be called to test correct attachment between the back end and the target server.
Once the attachment and initialization procedures have been carried out successfully, you are ready to begin the major task of writing, documenting, and testing the back-end functions. The remainder of this section outlines the key development issues and suggests one approach to implementing a back end.

**Design**

The first step in your implementation is to determine what functions your back end must provide. The easiest way to establish this is by using the *Back-End Developer’s Kit* as a basis. It contains a sample back end, the *acecpu32*, which is an implementation for the fictitious ACE C API. It operates an emulator supporting Motorola CPU32 microprocessors through BDM. It provides a structure and an implementation of common back-end services, including Gopher and checksum, which are emulator independent. You will need to support memory and register accesses, event handling on the target, communication with the emulator or other agent, state handling, and other emulator-dependent services.

Figure 2-8 shows the structure of the *acecpu32* implementation.

![Figure 2-7 acecpu32 Implementation](image)

Key:  
- Indicates that *Ace_T* inherits from *Backend_T*
- Indicates that *RWSlistCollectablesQueue* uses *Ace_T* in its implementation

Figure 2-9 shows how the *Event_T* class, which manages target event information, is implemented to queue *WDB_EVT_DATA*. 
Implementation

The recommended plan for implementing the back end is to add support incrementally for the desired services, exercising them with the `wtxtl` shell at each stage. (For more information on `wtxtl`, see 4.4 WTX Tcl API, p. 167.) Initially, you should write enough of the framework so that the target server can load your skeleton back end. Then you can add the key functions, testing them to make sure that you are on the right track. If you plan to support both UNIX and Windows hosts, we recommend that you build and test on both hosts in parallel.

1. First, create a directory in `installDir/host/src/tgtsvr/backend` for the back end. By convention, the name of the back end and its directory should be a short, descriptive phrase which is all lower-case, such as `acecpu32`, for the ACE CPU32 emulator back end.

2. Next, implement a skeleton back-end module which initializes the back end. The module must contain the back-end initialization routine and any other necessary routines. The target server will call the initialization routine when it loads the back end. You may find it helpful to provide stubs for other functions which log an error message to the console. Then you can start the target server with your back end and use the `wtxtl` shell to test that it invokes functions in response to WTX requests.

3. Implement the functions to read and write memory and registers.

4. Implement the functions to set breakpoints, stop execution, resume execution, and step a context.

5. Add event handling. At this point you should be able to get CrossWind, the source-level debugger, to operate successfully.
6. Implement desired optional functions, such as the ability to invoke functions on the target.

7. Tune performance. If you can optimize memory and register reads, you can improve performance significantly.

By implementing the back-end functions in this order, you will be able to verify quickly that different layers of the Tornado development environment interface correctly with your back end and the emulator you are supporting.

**Wind River Conventions**

The interface between the target server and the back end is documented in 2.2 Back-End Implementation, p. 19. In addition, 2.2.1 Attachment and Initialization, p. 20 describes the WDB data structures used to pass information into and out of the back end. Your back end methods must also use the WDB error codes. Both the error codes and the data structures are declared in `installDir/share/src/agents/wdb/wdb.h`.

The Wind River C coding conventions are documented in the *Tornado User’s Guide: C and C++ Coding Conventions*. The following additional conventions have been adopted for C++ and the Back-End Developer’s Kit:

- Class names end with `_T`.
- Member function names end with `_m`.
- Member data names end with `_`.
- Static member function names end with `_s`.
- Wind River C conventions for naming and indentation.

Finally, we recommend that you do not use C++ static objects in the back end because you will have no control over when their constructors and destructors are called. Instead, manage the objects with `new` and `delete`.

**Testing**

Back-end testing relies on the `wtxcl` shell provided with Tornado. This shell allows you to send WTX protocol requests to the target server, the same kinds of requests the tools send. Using `wtxcl`, you can interactively invoke the back-end methods to verify that they work correctly. Program `wtxcl` in Tcl, which Wind River has extended to support the WTX protocol.
2.4.2 Setting Up the Back-End Developer’s Kit

This section and the two sections that follow it use the libraries and methods of the Back-End Developer’s Kit as an example implementation for the CPU32 emulator. You may be able to use significant portions of this framework for your back end.

Design Considerations

The Back-End Developer’s Kit consists of a class library that provides the basic back-end framework, an example back end for acecpu32, sample test scripts, and this document. The library also implements common back-end methods, including Gopher and checksum, which are emulator-independent. By integrating this back-end framework into a new back end, you will not have to spend time developing and debugging these back-end services. To take advantage of this framework, derive a vendor-specific back-end class from the abstract base class Backend_T, declared in backend.h as shown in the following example:

```cpp
#include "backend.h"

// Declaration of Vendor-specific back end class
class Ace_T : public Backend_T
{
    // Backend_T methods which you are overriding go here
    ...  
    // Your vendor-specific methods and data go here
};
```

Once you have derived your vendor-specific class, determine what other classes and objects you need in your system. If you are integrating an existing C API for operating the emulator, you may find that the acecpu32 example back end can be modified.

Some of the classes and objects you may need include the following:

- an event class for handling events on the target
- a communication object to communicate with the emulator
- an object for handling state
- a parsing object to operate an emulator’s ASCII interface if there is no binary interface

Figure 2-10 shows the architecture of the acecpu32 back end.
Implementation

After installing the libraries supplied with the Back-End Developer’s Kit, follow the implementation procedure laid out in 2.4.1 Overview of Writing a Non-WDB Back End, p.48. When you begin tuning performance, if your emulator can perform an operation more efficiently than the default implementation (for example, a memory scan) override Backend_T’s methods.

For an example of how to use WDB data structures and error codes, look at the acecpu32 example code in installDir/host/src/tgtsvr/backend/acecpu32.

Porting

The Back-End Developer’s Kit’s class library is supported only on Solaris 2.5.1 and 2.6, Windows NT, and Windows 95 hosts. The source code for this library is provided and can be readily ported to other hosts because of the limited use of operating system calls. Furthermore, the Tornado development environment provides generic makefile stubs which abstract most of the host-dependent build issues.
Testing

The Back-End Developer’s Kit provides example test scripts which are located in \texttt{installDir/host/src/tgtsvr/backend/bedk/tests}. You can use these scripts as a basis for your own test scripts.

2.4.3 Getting Started

The first task is to get the basic skeleton of the back end operating. This means the target server should be able to start up, load the back-end DLL, and initialize, connect, and disconnect the back end.

Creating a Framework

In order to create the skeleton back end, carry out the following steps:

- Write \texttt{acecpu32.cpp}, which contains the routine \texttt{acecpu32Init()}. This file is the “main” of the back-end DLL.
- Declare your vendor-specific back-end class, \texttt{Ace_T}, writing stubs for the mandatory methods. This code belongs in the modules that implement the back end, \texttt{acecpu32Backend.h} and \texttt{acecpu32Backend.cpp}. The methods are not yet supported, but it is convenient to have them log a message to the console and return a value indicating that they were successfully invoked.
- Modify the example makefile to compile your back end.
- Use \texttt{wtxtcl} to test that the skeleton back end loads, and that you can execute back-end methods.

Implementation

To implement the skeleton back end, write the “main” back-end module, \texttt{acecpu32.cpp}, which initializes the back end by creating a vendor-specific back-end object, an instance of \texttt{Ace_T}.

Writing \texttt{acecpu32Initialize()} 

First, create \texttt{acecpu32.cpp} and define the routine \texttt{acecpu32Initialize()}, which initializes the back end. Excerpts from \texttt{acecpu32.cpp} are annotated throughout this section.
Tornado provides header files for the development of host tools in installDir/host/include. The first Tornado header file included must be host.h. Declare the interface for the vendor-specific back end, Ace_T, by including acecpu32Backend.h as shown:

```c
/* acecpu32.cpp - back end for ACE's CPU32 BDM emulator */
/* includes */

#ifdef WIN32
/* fix clash between Wind River's ERROR and Microsoft's ERROR macros */
#ifdef ERROR
#undef ERROR
#endif
#include <windows.h>
#endif /* #ifdef WIN32 */
#include "host.h"
#include "tgtlib.h"
#include "wpwrutil.h"
#include "windll.h"
#include "acecpu32Backend.h"

NOTE: Always include windows.h first to avoid clashes with Wind River definitions. Do not use ERROR, which Microsoft defines to be (0); always undefine ERROR, include system header files next, and include Wind River header files last.

Next, define the back-end initialization routine, acecpu32Initialize(). For a complete discussion, see 2.2.1 Attachment and Initialization, p.20.

The back end must initialize the TGT_OPS and BKEND_INFO structures with the information needed by the target server to operate the back end; in particular, the addresses of the back-end methods must be stored in TGT_OPS and the asynchronous notification method must be stored in BKEND_INFO. In order for the target server, which is a C application, to call the back-end methods, the Backend_T class uses static member functions to provide an interface that supports C calling conventions; these static functions then invoke normal member functions on the actual back end. For more information on the static wrapper and Backend_T architecture, see the documentation in provided in installDir/host/src/tgtsvr/backend/bedk/backend.h.

```c
STATUS acecpu32Initialize
{
    char * tgtName, /* network unique target name to connect */
    TGT_OPS * pTgtOps /* vector of back end functions to fill */
    BKEND_INFO * pBkendInfo /* Backend notification method */
}
```
The final step is to allocate the actual back end by calling `new` and to perform error logging. The constructor of `Backend_T`, the back-end abstract base class, will initialize the back-end function pointers in the `TGT_OPS` structure. `Ace_T`'s constructor initializes the back-end-specific part of the `TGT_OPS` structure.

The initialization routine must return `OK` on success or `ERROR` on failure, as defined in `host.h`. The Tornado' `wpwrutil` library provides several functions for error logging, including the function `WPWR_LOG_MSG` used in our example. Message logging is enabled by starting the target server with the option `-V`. For more information on the `wpwrutil` library, see the online reference material under Tornado API Reference>Target Server Back End Interface.

```c
// Create Backend
pTheBkEnd = new Ace_T(tgtName, timeout, recallNum, pTtyDevName,
                        baudRate, pTgtOps);
if (pTheBkEnd == NULL)
  {
    WPWR_LOG_MSG ("acecpu32Init(): new() failed.\n");
    return (ERROR);
  }
if (! pTheBkEnd->isValid_m())
  {
    WPWR_LOG_MSG ("acecpu32Init(): back end initialization failed.\n")
    return (ERROR);
  }
return (OK);
```

NOTE: For Windows hosts, it is important to use default calling conventions and not `WINAPI`.

Add any other routines to `acecpu32.cpp` that are needed to support the initialization of the back end.

NOTE: Constructors can fail, but most compilers lack exception support. For this reason, the `Backend_T` class provides a pair of protected members. `Backend_T::isNotValid_m()` should be called if vendor-specific constructors fail. Call `Backend_T::isValid_m()` to see if the back end has been successfully initialized.
Declaring the Vendor-Specific Back-End Class Skeleton

Next, declare the skeleton of the vendor-specific back-end class, Ace_T. Derive Ace_T from Backend_T and declare all the mandatory methods in acecpu32Backend.h. The examples in this section show the declaration of the Ace_T class in acecpu32Backend.h.

After the usual preamble of comments and modification history, give the header file the standard macro #ifndef/#define construction to prevent multiple inclusions of the header file. Next, provide the #include statements for the other necessary header files.

```c
/* acecpu32Backend.h - header file for ACE SuperBDM BDM back end */
#define __INCacecpu32Backendh

/* includes */

#include "rw/queuecol.h"
#endif
#define WIN32
#define ERROR (-1)
#endif

#include "ace/api.h" /* ACE's API header file */
#endif
#endif

#include "host.h"
#include "tgtlib.h"
#include "wpwrutil.h"
#include "wdb.h"
```

First, include the Rogue Wave header file.

```
#include "rw/queuecol.h"
```

Next, include the header file for the ACE C API, ace/api.h. Include it at this point rather than later, with the other BEDK header files, because it includes Windows header files, and they must be included before the system headers. The undefining and redefining of ERROR is required when Windows header files are included because Windows uses a non-standard ERROR definition.

```
#include "ace/api.h" // ACE's API header file
```

Next, include Tornado header files; wdb.h defines the WDB data structures used to pass information between the back end and the target server.

```
#include "host.h"
#include "tgtlib.h"
#include "wpwrutil.h"
#include "wdb.h"
```
Finally, include `backend.h`, which pulls in the declaration of `Backend_T` (the back-end abstract base class) and other back-end-specific header files.

```c
#include "backend.h"
#include "event.h"
#include "bdmExcLib.h"
```

Next, create a new link ID to identify your back end as a unique target-link type. This should be done by defining a macro `TGT_LINK_BDM_ACE` in `tgtlib.h`. You should choose an unused number in the 0x200 range.

```c
#ifndef TGT_LINK_BDM_ACE
#define TGT_LINK_BDM_ACE (0x201) /* ACE BDM support */
#endif
```

Now, declare the `Ace_T` back-end class as shown below. To reuse the back-end framework and services implemented in `Backend_T`, you must derive your vendor-specific back end from it. It is declared in `installDir/host/src/tgtsvr/backend/bedk/backend.h`. The `Ace_T` constructor has the same arguments as the `acecpu32Init()` routine. It is automatically called when the actual `Ace_T` back-end object is allocated. The destructor must be declared `virtual` because `Ace_T` has virtual functions.

```c
class Ace_T : public Backend_T
{
  public:

    // Constructors and Destructors.
    Ace_T (char * tgtName, u_int timeout, u_int retryNum, char * devName,
           u_int param, TGT_OPS * pTgtOps, BKEND_INFO * pBkendInfo);

    virtual ~Ace_T () ;

    Once the constructor and destructor have been declared, declare the mandatory back-end methods. These methods are listed in `backend.h`.

    // Declaration of mandatory member functions.
    UINT32 tgtPing_m (void);

    UINT32 tgtConnect_m (WDB_TGT_INFO * pTgtInfo);

    UINT32 tgtDisconnect_m (void);
    ...
```
For the present, ignore the optional member functions. If your emulator supports a faster way of performing these operations than the generic implementation provided by the Backend_T class, you may decide to implement them later. Typically, Backend_T implements these methods by reading or writing a block of target memory. If your emulator supports an operation (for example, memory fill) directly, you will gain performance by using your emulator’s primitive.

    // optional member functions
    ...

Now declare the mandatory helper methods. Ace_T::fdGet_m() should return the event file descriptor, whose activity indicates that an event has occurred on the target. Ace_T::halt_m() and Ace_T::unhalt_m() are used by Backend_T whenever it needs to suspend the target’s system context or return the target to the state it was in before suspension. Typically, these primitives are used before performing a memory operation like Gopher or checksum.

    // mandatory helper methods
    virtual int       fdGet_m();
    // State management
    virtual UINT32   halt_m();
    virtual UINT32   unhalt_m();

Eventually, you will provide whatever vendor-specific helper methods and data are needed to implement your back-end class, such as Ace_T::eventCallBack(), which is used to handle target events.

    ....
    // handles asynchronous events in the ACE API
    static void eventCallBack (ACE_ConnectionHandle, ACE_Event *, void *);
    ....

    // End of class Ace_T
    //////////////////////////////////////////////////////////////////////
    #endif /* #ifndef __INCacecpu32Backendh */

Implementing the Vendor-Specific Back-End Class Skeleton

The back-end methods are implemented in acecpu32Backend.cpp. Initially, provide stubs for these methods that log a message to the console when they return successfully. This allows you to validate that the Tornado framework can successfully invoke methods in the new back end before you have coded all the methods. For a discussion of how to implement the mandatory and optional
methods, see 2.4.4 Implementing Mandatory Member Functions, p.67 and
2.4.5 Implementing Optional Member Functions, p.87.

/* acecpu32Backend.cpp - implements back end for ACE’s SuperBDM */
/* includes */
...

The constructor call has the same prototype as the acecpu32Init() routine and the Backend_T constructor.

Ace_T::Ace_T
{
    char *        tgtName,
    TGT_OPS *     pTgtOps,
    BKEND_INFO * pBkendInfo
}

Because Ace_T is derived from Backend_T, whenever an Ace_T object is allocated, Backend_T’s constructor is called first. It is a good idea to include the call to Backend_T’s constructor in an initializer list as a reminder of this fact, to improve performance, and to make sure the correct Backend_T constructor is called. Also, include any aggregate objects used in Ace_T, such as event queues and flags.

: Backend_T (tgtName, pTgtOps, pBkendInfo)
{

Backend_T’s constructor initializes the generic information in the TGT_OPS structure. It also initializes BKEND_INFO. In other words, it defines what functions the target server should invoke to perform the various back-end operations. (See The BKEND_INFO Structure, p.25 and 2.2.2 Back-End Functions, p.26.) See installDir/host/src/tgtsvr/backend/bedk/backend.cpp for functions Backend_T initializes.

Ace_T must initialize the other information in the TGT_OPS structure. Note that when event handling is implemented, the constructor calls Ace_T::fdGet_m() to determine which event file descriptor the target server should monitor for events on the target. Initially, since event handling has not yet been implemented, you should store NONE in tgtEventFd. A new Tornado back-end type was specified by defining TGT_LINK_BDM_ACE in tgtlib.h.

... pTgtOps->tgtConnected = FALSE;
pTgtOps->tgtEventFd = NONE; // Record event fd.
Log a diagnostic message on completion of the constructor:

```c
// Log diagnostic message
WPWR_LOG_MSG ("Ace_T::Ace_T() : succeeded!\n");
```

Similarly, implement a stub for the destructor. (C++ automatically calls `Backend_T`'s destructor.)

```c
Ace_T::~Ace_T ()
{
    // Log diagnostic message
    WPWR_LOG_MSG ("Ace_T::~Ace_T() : succeeded!\n");
}
```

Finally, you need to provide stubs for the mandatory back-end methods. To validate the back-end framework, it is helpful if these methods log a message and return the appropriate value to indicate success. Back-end methods have three possible return values. You may find it helpful to implement the three functions shown below so that you can see the return value and which method was invoked.

```c
LOCAL UINT32 stubUINT32 (const char * pMethod)
{
    WPWR_LOG_MSG ("Ace_T: - method %s.\n", pMethod);
    return (WDB_OK);
}
```

```c
LOCAL BOOL   stubBOOL (const char * pMethod)
{
    WPWR_LOG_MSG ("Ace_T: - method %s.\n", pMethod);
    return (TRUE);
}
```

```c
LOCAL void   stubVOID (const char * pMethod)
{
    WPWR_LOG_MSG ("Ace_T: - method %s.\n", pMethod);
}
```

NOTE: UINT32 is a Wind River convention for a 32-bit, unsigned integer. It is declared in `installDir/host/include/host.h`.

Use the appropriate stub function for each method’s return type to implement the remaining mandatory methods. For example, `Ace_T:tgtPing()` would be implemented as follows:
UINT32 Ace_T::tgtPing_m (void)  
  
  return (stubUINT32 ("tgtPing_m ( )")); 
  

The only function besides Ace_T’s constructor you should implement without the stub functions is the Ace_T::tgtConnect_m() method. This method initializes WDB_TGT_INFO, which describes the target’s configuration.

UINT32 Ace_T::tgtConnect_m (WDB_TGT_INFO * pWdbTgtInfo)  
  
  WPWR_LOG_MSG ("Establishing ACE SuperBDM connection... ");  
  // XXX - this information should be read from the $WIND_TGT_INFO  

First, record information about the debug agent, in this case the new emulator back end. agentInfo.mtu specifies the maximum amount of data that can be sent or received by the agent. agentInfo.mode is always set to WDB_MODE_EXTERN for a system-level debug agent like our emulator; this value is defined in wdb.h.

  pWdbTgtInfo->agentInfo.agentVersion = "ACE BDM 1.0.1";  
  pWdbTgtInfo->agentInfo.mtu = Ace_T::MaxMtu;  
  pWdbTgtInfo->agentInfo.mode = WDB_MODE_EXTERN;  

⚠️ CAUTION: pWdbTgtInfo->agentInfo.agentVersion specifies which version of Tornado the back end supports. This string must include the appropriate version number for initialization to proceed.

Next, describe the real-time operating system supported. At present only VxWorks is supported. Possible values for rtInfo.cpuType are defined in installDir/host/include/cputypes.h. Set the value of rtInfo.endian to 1234 or 4321 for big-Endian or little-Endian targets, respectively.

  pWdbTgtInfo->rtInfo.rtType = WDB_RT_VXWORKS;  
  pWdbTgtInfo->rtInfo.rtVersion = "5.3";  
  pWdbTgtInfo->rtInfo.cpuType = CPU32;  
  pWdbTgtInfo->rtInfo.hasFpp = FALSE;  
  pWdbTgtInfo->rtInfo.hasWriteProtect = 0;  
  pWdbTgtInfo->rtInfo.pageSize = 0xffffffff;  
  /* The CPU32 target is always Big Endian */  
  pWdbTgtInfo->rtInfo.endian = 1234;  

⚠️ NOTE: pWdbTgtInfo->rtInfo.cpuType is architecture specific.
The target server invokes `Backend_T::tgtConnect_m()` to perform any generic target connection work which may be needed. This function parses the target information configuration file by invoking `Backend_T::tgtInfoGet_m()`. See the source code in `backend.cpp` for more information on `Backend_T::tgtInfoGet_m()`.

```c
if (Backend_T::tgtConnect_m (pWdbTgtInfo) != WDB_OK) {
    WPWR_LOG_ERR ("Backend_T::tgtConnect_m () failed.\n");
    return (WDB_ERR_PROC_FAILED);
}

WPWR_LOG_MSG ("succeeded.\n");
return (WDB_OK);
```

Now the skeleton is complete and ready for building and testing.

**Building the Back End on UNIX**

Tornado provides makefile stubs, which implement the generic part of a makefile. This facilitates the development and porting of makefiles. Including these stubs in a makefile assures that the generic part of the makefile is correctly implemented in a host-independent fashion.

**Building the Back-End Developer’s Kit (BEDK) Library**

In order to use the BEDK library you must build this library before building the back end. A makefile and source code for the BEDK are provided in the BEDK directory, `installDir/host/src/tgtsvr/backend/bedk`. The makefile is designed to be host-independent to facilitate porting the BEDK to hosts that are not SunOS 4.1.3.

**Building a New Back End**

Once the BEDK library is built, create a makefile to build the new back end. Use the makefile provided for `acecpu32` as an example. After the usual comments, define the necessary suffixes to build a back-end DLL on your host platform from C++. Inference rules are already defined in the makefile fragments provided by Tornado, but you should define any extra macros which are needed for the back end. The `ACE_XXX` macros specify the location of the ACE C API library and header files.

```
# Makefile - for acecpu32 backend
...
.SUFFIXES: .cpp .so .sl
```
Target Server Back End

\begin{verbatim}
ACE_BASE          = /folk/bss/sig/bdm/ace/api/aceapi_1.1
ACE_LD_FLAGS      = -L$(ACE_BASE)/SunOS/lib -lACE68K
ACE_INC           = -I$(ACE_BASE)/include

EXTRA_CFLAGS     = -DACETARGET_68K
include $(WIND_BASE)/host/include/make/generic.mh
include $(WIND_BASE)/host/include/make/$(HOST).mh

Because the BEDK library uses Rogue Wave's Tools.h++ library, you must
define the following macros which specify where Tools.h++ is installed:

RW_ROOT           = /vobs/devtools/rogue/$(HOST)
RW_INC            = -I$(RW_ROOT)
RW_LIB            = $(RW_ROOT)/lib/librwtool.a
RW_LD_FLAGS       = -L/vobs/devtools/rogue/$(HOST)/lib -lrwtool

Next, define INCLUDES to specify the include files location. Define BKEND_OBJS

to be the list of objects that should be built and linked into the new back end:

INCLUDES     = $(WIND_INC) $(WIND_SHARE_INC) $(ACE_INC) $(RW_INC) \
               -I$(WIND_BASE)/share/src/agents/bedk \
               -I$(GNU_ROOT)/lib/g++include

BKEND_OBJS = acecpu32.o acecpu32Backend.o event.o
FULL_BKD_OBJS = $(patsubst %.o, $(HOST)/%.o, $(BKEND_OBJS))

FULL_BKD_OBJS causes the object files to be placed in a subdirectory named after
your \texttt{hostType} (on UNIX, the \texttt{WIND_HOST\_TYPE} environment variable, for
example, \texttt{sun4-sunos4}).

Specify the libraries to be linked with the back end, including their path names.
The Tornado makefile stub defines a macro \texttt{gnu\_ROOT}, which specifies the root of
the tree containing the GNU ToolKit used to build the back end. You may need to
redefine it for your configuration. If you forget to link a required library with the
back end, the target server will crash when it loads the back end. An error message
will state which symbol was unresolved. Run \texttt{c++filt} or an equivalent tool to
demangle the symbol; you can then use \texttt{nm} to determine which library contains
that symbol and link the library with the back end.
\end{verbatim}
BKEND_LIB_EXTRA = \\
-L./bedk \\
-lbedk \\
$(ACE_LD_FLAGS) \\
$(RW_LD_FLAGS) \\
-L$(gnu_ROOT)/lib \\
-lg++ \\
-liostream \\
-L$(gnu_ROOT)/lib/gcc-lib/sparc-sun-sunos4.1.3_U1/2.6.3/ \\
-ldl -lc -lgcc

Specify the name of the back end using SH_BKEND_OBJ. Then use LOCAL_CFLAGS to define the local compiler options: -DPORTABLE specifies portable versions of host libraries, -DHOST specifies host versions of header files, and $(DYN_LK_FLAGS) sets the options for compiling a DLL.

SH_BKEND_OBJS = acecpu32.$(SHLIB_EXT)

... LOCAL_CFLAGS = -DPORTABLE -DHOST $(DYN_LK_FLAGS) -fno-builtin \\ 
                 -g -fno-inline

Finally, there is a target rule and an inference rule to build the back-end DLL:

default:lib objdircre $(SH_BKEND_OBJS)

%.$(SHLIB_EXT):$(FULL_BKD_OBJS)

  $(SHARED_LD) $(SHARED_LDFLAGS) -o$(SH_BKEND_LIB)/$*.$(SHLIB_EXT) \\
  $(BKEND_OBJS) $(BKEND_LIB_EXTRA) $(BKEND_XDR_OBJS)

... include $(WIND_BASE)/host/include/make/generic2.mh ... 

Once your makefile is complete, build the skeleton back end with make.

Building on Windows NT and Windows 95

1. Use Microsoft Visual C++ 5.0.

2. Create a project for a multi-threaded DLL which includes BEDK source, your back end source, and any third party sources.

3. In the project settings:

   - Link with Rogue Wave and other necessary libraries. Link only with multi-threaded DLLs. Do not link with libc or libcmt.

   - Define the macros RW_NO_STL and _RWTOOLSDLL (required by Roque Wave). Also define WIN32 and HOST (required by Wind River).
4. Build Rogue Wave’s multi-threaded DLL:

\nmake -f makefile.msc ENVIRON=WIN32 THREAD=MULTI BINDING=DLL

Define RW_NO_WSTR in rw/compiler.h. Define RW_NO_STL and _RWTOOLSDLL.

5. Do not mix debug and release objects and libraries.

**Testing**

There are two milestones that should be achieved while testing the skeleton back end: successfully loading the back end you build, and successfully invoking back end methods with wtxtcl.

**Loading the Back End**

After you have built the back end, the next step is to invoke the target server. Invoke the target server using Tornado’s **Launch** tool, or enter:

```
% tgsrvr -V -B acecpu32 -c $WIND_BASE/target/config/ace360/vxWorks -A hostNameOfEmulator
```

where -V enables verbose diagnostic output, -B specifies which back end to use, -core specifies the path to the VxWorks image loaded on the target, -A causes all of VxWorks’ symbols to be loaded into the target server symbol table (this flag is optional), and hostNameOfEmulator is the emulator’s host name.

If you do not have a VxWorks image for the target board, you can use the following instead:

```
% tgsrvr -V -B acecpu32 -f a.out hostNameOfEmulator
```

where -f specifies the object module format (OMF) to use. Use the same OMF which Wind River uses for your target architecture. For more information on the target server and its options see the reference pages and the Tornado User’s Guide: Getting Started.

If you are using UNIX and fail to link all the necessary libraries into the back end, the target server crashes. (This problem will not occur under Windows because Microsoft Visual C++ will generate error messages for unresolved externals during the link phase.) The UNIX error message is similar to the following:
To solve this problem, you need to determine which library implements the undefined symbol, \texttt{\_\_builtin\_new}. You will find \texttt{nm} invaluable for solving this problem, because it displays the symbols in a module. If the undefined symbol is a mangled name, use \texttt{c++filt} to convert the name to an unmangled name to make your search easier.

A side effect of the target server’s crash is that the target server does not unregister with the registry daemon, \texttt{wtxregd}. You can confirm this using the registry status command, \texttt{wtxreg}:

\begin{verbatim}
% wtxreg
Registry for acheron:
 Name    Arch     Mb   #tools User      Idle
------------------------      ------------- ------ ------ --------- -----
myTarget@acheron  DOWN
\end{verbatim}

You cannot restart a target server with the same name until you clean up the registry. To do so, use the \texttt{wtxcl} script \texttt{unreg}, shown in \textit{Shutting Down the Target Server} on p. 17, or use \texttt{Launch}.

When the back end is correctly linked, invoking the target server should produce the following messages:

\begin{verbatim}
tgtsvr (myTarget@acheron): Tue May  7 12:37:27 1996
 Attaching backend... succeeded.
 Establishing ACE SuperBDM connection... succeeded.
 Attaching C++ interface... succeeded.
 Attached a.out OMF reader.
 Warning: Core file checksums do not match.
\end{verbatim}

Note that the \texttt{acecpu32} back end does not perform the checksum because it would take too long, given this emulator’s low bandwidth memory-read capability. Consequently, the target server logs a warning. This warning will not appear if your back end can perform a checksum, and if it matches the target server checksum of VxWorks’ text segment.

\textbf{Invoking Back-End Methods}

Once you have built a back end that the target server can load, use \texttt{wtxcl} to invoke back-end methods. This validates that tools can use the Tornado framework to invoke back-end methods. A typical test session involves starting the \texttt{wtxcl} shell, attaching the target server and testing various back-end methods by issuing \texttt{wtxcl} commands:
If your implementation is successful, the back end logs appropriate messages when you invoke a back-end method. Most WTX Tcl procedures, like `wtxMemRead`, directly map onto a corresponding back-end method. The first WTX Tcl command which you enter is always `wtxToolAttach` to connect `wtxcl` to a specific target server. Then invoke appropriate WTX Tcl procedures to test the back-end framework.

If a `wtxcl` method fails, use `installDir/host/include/wtxerr.h` to convert the error code into a meaningful message. For example:

```
% wtxcl
wtxcl> wtxToolAttach myTarget
myTarget@acheron
wtxcl> wtxMemRead 0x0 16
mblk0
...
wtxcl> exit
```

To convert 0x1000d4 into a meaningful value, convert 0xd4 to a decimal value. You should find a corresponding entry in `wtxerr.h`. You may find it helpful to create a script to do this using `grep` and `bc`. Once you have converted the value, you find:

```
wx_ERR_AGENT_NO_AGENT_PROC = (WXERR_BASE_NUM | 212)
```

### 2.4.4 Implementing Mandatory Member Functions

Once the back-end framework is validated, you can implement the mandatory member functions. First implement memory and register read and write methods. Once this milestone is achieved, implement the remaining mandatory back-end functions including state management and event handling. Studying the `acecpu32` back-end example in parallel with this section will be helpful, especially in understanding the use of the WDB data structures. The required back-end methods are shown in Table 2-1.

**Implement Read and Write**

The next step is to implement the methods to read and write memory and registers. This validates that you can operate the emulator from Tornado and that the WDB data structures are correctly passing information between the target server and the back end.
Implementing Ace_T::memRead_m()

The first back-end methods to implement are the methods for reading and writing memory. This section presents a detailed example for the memory-read method. The memory-write method is similar.

---

Table 2-1  Mandatory Member Functions

<table>
<thead>
<tr>
<th>Member Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>tgtConnect_m()</td>
<td>Connect back end to emulator.</td>
</tr>
<tr>
<td>tgtDisconnect_m()</td>
<td>Disconnect back end from emulator.</td>
</tr>
<tr>
<td>memRead_m()</td>
<td>Read data from target memory.</td>
</tr>
<tr>
<td>memWrite_m()</td>
<td>Write data to target memory.</td>
</tr>
<tr>
<td>regsGet_m()</td>
<td>Read target registers.</td>
</tr>
<tr>
<td>regsSet_m()</td>
<td>Write data to target register(s).</td>
</tr>
<tr>
<td>eventpointAdd_m()</td>
<td>Set a breakpoint.</td>
</tr>
<tr>
<td>eventpointDelete_m()</td>
<td>Remove a breakpoint.</td>
</tr>
<tr>
<td>contextSuspend_m()</td>
<td>Suspend a context on the target.</td>
</tr>
<tr>
<td>contextResume_m()</td>
<td>Resume a context on the target.</td>
</tr>
<tr>
<td>contextCont_m()</td>
<td>Continue a target context.</td>
</tr>
<tr>
<td>contextStep_m()</td>
<td>Step a target context.</td>
</tr>
<tr>
<td>eventGet_m()</td>
<td>Get WDB event data.</td>
</tr>
<tr>
<td>evtPending_m()</td>
<td>Check if an event is pending.</td>
</tr>
<tr>
<td>evtPendingClear_m()</td>
<td>Clear an event after processing.</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>Functions Specific to the Back End Developer's Kit</td>
<td></td>
</tr>
<tr>
<td>fdGet_m()</td>
<td>Get an event file descriptor.</td>
</tr>
<tr>
<td>halt_m()</td>
<td>Halt the system.</td>
</tr>
<tr>
<td>unhalt_m()</td>
<td>Return the system to its state prior to halting.</td>
</tr>
</tbody>
</table>

---
When the target server calls `Ace_T::memRead_m()`, it passes a pointer to a `WDB_MEM_REGION` structure describing the region of target memory to be read, and a pointer to a `WDB_MEM_XFER` structure describing where the data read from the target should be returned.

```c
UINT32 Ace_T::memRead_m
{
    WDB_MEM_REGION * pMemRegion,
    WDB_MEM_XFER   * pMemXfer
}

int                status;
ACE_OperatingMode  oldState;
```

To check that an access request is for a valid address, `Ace_T::memRead_m()` compares the request to the bounds of target memory, which are stored in the `WDB_TGT_INFO` structure parsed by `Backend_T::tgtInfoGet_m()`.

```c
// Check that memory access is valid
UINT32 aceMemBase = wdbTgtInfo_.rtInfo.memBase;
UINT32 aceMemSize = wdbTgtInfo_.rtInfo.memSize;

if  ((UINT32) pMemRegion->baseAddr < aceMemBase) ||
    ((UINT32) pMemRegion->baseAddr + pMemRegion->numBytes) >
     (aceMemBase + aceMemSize))
{
    ::wpwrLogWarn("Invalid memory access of %#x bytes at %#x.\n", pMemRegion->numBytes, pMemRegion->baseAddr);
}
```

Next, the `WDB_MEM_XFER` structure, which will hold the result of the memory read, is initialized. The `param` field is used in a request-dependent manner. In this example, `param` contains the address of the buffer to store the data in.

```c
// Book keeping for WTX protocol
pMemXfer->source           = (WDB_OPQ_DATA_T) pMemRegion->baseAddr;
pMemXfer->numBytes         = pMemRegion->numBytes;

The target server will not break a memory read request up into several small requests if the initial request is larger than the `agentInfo.mtu` value specified in the `WDB_TGT_INFO` data returned by the `Ace_T::tgtConnect_m()` request. Consequently, the back end needs to break up the request if necessary.

```c
// The target server does not break up requests which
// exceed the back end’s MTU, so we need to do that here.
TGT_INT_T numLeft;          // num bytes left to read.
```
TGT_INT_T numToRead; // num bytes to read in this iteration.
UINT8 * pTgtAddr; // address to start next read.
UINT8 * pDestAddr; // where to put result of read.

pTgtAddr = (UINT8 *) pMemRegion->baseAddr;
pDestAddr = (UINT8 *) pMemRegion->param;

for {
    numLeft = pMemRegion->numBytes;
    numLeft > 0;
    numLeft -= Ace_T::MaxMtu
}

numToRead = ((numLeft > Ace_T::MaxMtu) ?
    (TGT_INT_T) Ace_T::MaxMtu : numLeft);

Next, the back end uses a function in the ACE C API to read memory from the
target via the emulator:

    status = ::ACE_ReadTargetMemory
    {
        emulator_,
        (ACE_TargetAddress) pTgtAddr,
        (void *) pDestAddr,
        (uint32) numToRead
    };

If the ACE C API request fails, the back end calls Ace_T::wdbErrFigure_m() to
convert the ACE error code into a WDB error code and log an appropriate error
message. If the method's return type is UINT32, the back-end methods must return
WDB_OK or a suitable WDB error code, defined in wdb.h.

    if (status != ACE_SUCCESS)
    {
        (void) stateRestore_m (oldState);
        return (wdbErrFigure_m("ACE_ReadTargetMemory ()",
            status, WDB_ERR_MEM_ACCES));
    }

    pTgtAddr += numToRead;
pDestAddr += numToRead;
}

return (WDB_OK);

The next step is to rebuild the back end, start the target server, and test the
memory-read method. In this example, wtxcli is used to attach to the target server,
read target memory, and display the result. wtxMemRead returns a memory block
handle (mblk0 in the example) to provide more efficient memory manipulation. For more information, see the 4. The WTX Protocol.

```% wtxtcl
wtxtcl> wtxToolAttach myTarget
myTarget@acheron-1
wtxtcl> wtxMemRead targetAddress 16
mblk0
wtxtcl> memBlockGet -l mblk0
0xdddeadbeef 0xdddeadbeef 0xdddeadbeef 0xdddeadbeef
wtxtcl> exit```

You may find it useful to create a library of scripts to exercise simple back-end requests. See `installDir/host/src/tgtsvr/backend/bek` for examples of such scripts. You can invoke them from `wtxcl` with the `source` command.

**Implementing **\texttt{Ace\_T::regsSet\_m()}\**

The implementation of the methods to read and write registers is analogous to the memory read method. There are three possible sources of confusion about register operations. First, the WDB protocol allows just part of a register set to be accessed. Second, you should use the Wind River-defined register-set structure. Third, register information must be handled in opaque format; in other words, it must have the same byte ordering and alignment as the target. The definition of the register set is in `installDir/target/h/archType/regsArchType.h`, where `archType` is the target architecture type.

```c
UINT32 Ace_T::regsSet_m (WDB_REG_WRITE_DESC * pRegWrite)
{
    ACE_AllRegisters aceRegSet;
    ACE_OperatingMode oldState;
    int status; // ACE API return value
```

Register access is one of the least portable parts of the back end. To handle the register set for your target architecture, you need to define a register-set structure which mimics the definition in `regsArchType.h`. The ACE emulator supports CPU32 targets, so a 68k register-set type, `REG_SET_68K`, is declared in `acecpu32Backend.h` and shown here:

```c
REG_SET_68K    wrsRegSet;
char *        pWrsRegBuf; // pointer to treat reg set as opaque
```

Having established the correct register-set structure, perform some sanity checking on the arguments. Note the use of WDB error codes.

```c
    if (pRegWrite->context.contextType != WDB_CTX_SYSTEM)
    {
        return (WDB_ERR_INVALID_CONTEXT);
    }
```
if (pRegWrite->regSetType != WTX_REG_SET_IU) {
    return (WDB_ERR_INVALID_PARAMS);
}

CrossWind, the GDB-based source-level debugger, accesses only one register at a time. Consequently, it is important to optimize for the case of reading and writing only one register. The back end API also requires support for reading and writing the complete register set, so acecpu32 includes both cases.

if (pRegWrite->memXfer.numBytes <= 4) {
    uint32            aceRegSize;
    ACE_Register      whichAceReg;     // which register to read.
    }

Next, the register-set method converts the Wind River register identifier, which is a byte offset into the register set, into an ACE register identifier, which is an enumerated type. The Wind River byte offset is stored in memXfer.destination.

status = whichAceRegGet_m (pRegWrite->memXfer.destination,
                           &whichAceReg, &aceRegSize);
if (status != WDB_OK) {
    return (status);
}

The 68k does not support direct modification of the A7 register, so you must modify the SSP.

if (whichAceReg == ACE_REG_A7) {
    whichAceReg = ACE_REG_SSP;
}

CrossWind treats the status register as 32 bits: 16 bits of padding plus 16 bits of data. The ACE API treats the status register as 16 bits of data. The appropriate conversion is performed here:

if ((whichAceReg == ACE_REG_SR) &&
    (pRegWrite->memXfer.numBytes == 4)) {
    UINT8 *    pRegWriteVal = (UINT8 *) pRegWrite->memXfer.source;
    pRegWriteVal += 2;    // Skip padding in WRS's REG_SET.

    // memXfer.source contains register information
    // in the same representation it would appear in
    // target memory (i.e., it is opaque). The
    // ACE API expects this information in host byte
    // ordering, so we must convert the register value.
*(UINT16 *) pRegWriteVal = FIX_16 (*(UINT16 *) pRegWriteVal);

status = regSetOne_m (whichAceReg, pRegWriteVal);
}
else
{
  // memXfer.source contains register information
  // in the same representation it would appear in
  // target memory (i.e., it is opaque). The
  // ACE API expects this information in host byte
  // ordering, so we must convert the register value.

  UINT32 aceRegBuf = FIX_32 (*(UINT32 *)
pRegWrite->memXfer.source);

  A special vendor-specific helper function, **Ace_T::regSetOne_m()**, efficiently
  writes just one register, returning a WDB error code:

  status = regSetOne_m (whichAceReg, (UINT8 *) &aceRegBuf);
}

if (status != WDB_OK)
{
  return (status);
}

return (stateRestore_m (oldState));

// End of writing only one register

Having dealt with the special case of accessing only one register, the
**Ace_T::regsSet_m()** now performs the more general case by reading the target’s
entire register set, modifying the desired registers, and re-writing the register set.

// Read all target registers

status = ::ACE_ReadAllTargetRegisters (emulator_, &aceRegSet);

if (status != ACE_SUCCESS)
{
  ...
  return (wdbErrFigure_m ("ACE_ReadAllTargetRegisters ()",
                          status, WDB_ERR_PROC_FAILED));
}

// Force ACE register info into WRS data structures (defined in
// $WIND_BASE/target/h/arch/mc68k/regsMc68k.h)
::memset (&wrsRegSet, 0x00, sizeof (wrsRegSet));

// memXfer.source contains register information
// in the same representation it would appear in
NOTE: The ACE API returns data in host byte order, which must be converted to opaque format.

In the example below, note how `memXfer` specifies which part of the register set to modify: `memXfer.destination` specifies the byte offset into the target register set where the new data should be stored; and `memXfer.source` contains a pointer to an opaque chunk of register data, formatted like the corresponding part of a Wind River register-set structure. For example, if the byte offset is 64 bytes, `memXfer.source` points to two bytes of padding, followed by two bytes of status-register information, and so on for a CPU32 register set.

```c
// Modify desired registers
pWrsRegBuf = (char *) &wrsRegSet;
::memcpy ((void *) (pWrsRegBuf + pRegWrite->memXfer.destination),
(void *) pRegWrite->memXfer.source,
pRegWrite->memXfer.numBytes);

// Store in ACE's reg set.
aceRegSet.Dx[0] = FIX_32 (wrsRegSet.dataReg[0]); /* data registers */
aceRegSet.Dx[1] = FIX_32 (wrsRegSet.dataReg[1]);
aceRegSet.Dx[2] = FIX_32 (wrsRegSet.dataReg[2]);
aceRegSet.Dx[3] = FIX_32 (wrsRegSet.dataReg[3]);
aceRegSet.Dx[5] = FIX_32 (wrsRegSet.dataReg[5]);
aceRegSet.Dx[6] = FIX_32 (wrsRegSet.dataReg[6]);
aceRegSet.Dx[7] = FIX_32 (wrsRegSet.dataReg[7]);
```
aceRegSet.Ax[0] = FIX_32 (wrsRegSet.addrReg[0]); /* address regs */
aceRegSet.Ax[1] = FIX_32 (wrsRegSet.addrReg[1]);
aceRegSet.Ax[3] = FIX_32 (wrsRegSet.addrReg[3]);
aceRegSet.SSP   = FIX_32 (wrsRegSet.addrReg[7]); // This assumes use of vxWorks!
aceRegSet.SR    = FIX_16 (wrsRegSet.sr);
aceRegSet.DC    = FIX_32 ((uint32) wrsRegSet.pc);
status = ::ACE_WriteAllTargetRegisters (emulator_, &aceRegSet);

if (status != ACE_SUCCESS)
{
    ...
    return (wdbErrFigure_m ("ACE_WriteAllTargetRegisters ()",
        status, WDB_ERR_PROC_FAILED));
}
else
{
    return (WDB_OK);
}

Tornado expects to receive event information (including exception information) and register information in target byte order. Register contents should be transferred in opaque fashion (in other words, bit for bit as they appear in the registers).

Finish implementing and testing the methods for reading and writing memory and registers. Use your emulator’s tools to verify that wtxtcl commands correctly operate the emulator.

Implement Context Management

By implementing the read and write memory and register methods, you demonstrated that the new back end can actually operate the emulator. The next step is to add support for basic context management: setting breakpoints, suspending and resuming the system, and so on. Follow the same process of adding functionality and testing it with wtxtcl and your emulator’s tools.

The key difference in this step is that you need to manage state: is the target running? do you need to stop it to perform the operation? We first examine the
methods for acecpu32 back-end state management in detail. Then we discuss caveats for the other context management methods.

State Management

The ACE back end uses `Ace_T::stateBDM_m()` and `Ace_T::stateRestore_m()` to manage the target state. The first method is used to halt the system by putting the target in background debug mode or BDM. The second method is used to return the system to the state it was in before it was halted. Once these methods are implemented, the functionality must be made available to the `Backend_T` class by implementing the mandatory member functions `halt_m()` and `unhalt_m()`; this allows the methods implemented in `Backend_T` to manage target state.

The examples step through `Ace_T::stateBDM_m()` in detail. When this method is called, it is passed a variable where it can record the old state. This variable is passed to `Ace_T::stateRestore_m()`, which restores the system to that state. This implementation supports nested calls of these state management methods. (Note: the argument to `Ace_T::stateBDM_m()` is a reference!)

```c
UINT32 Ace_T::stateBDM_m (ACE_OperatingMode & oldState)
{
    int status;

    An ACE C API method is used to determine the target state. Based on this state, `Ace_T::stateBDM_m()` performs the appropriate operation.

    oldState = (ACE_OperatingMode) ::ACE_GetOperatingMode (emulator_);

    switch (oldState)
    {
    
    If the state is `ACE_MODE_BDM`, the target is already halted.

    case ACE_MODE_BDM:
        status = ACE_SUCCESS;
        break;

    If the system is running, use the ACE C API to halt the target:

    case ACE_MODE_RUN:    /* Running */
    case ACE_MODE_TRC:    /* Non-realtime trace */
        status = ::ACE_StopTarget (emulator_);
        if (status != ACE_SUCCESS)
        {

6. Many microprocessors provide a special debug state which allows a debugging tool to take advantage of on-chip debug functionality. For the CPU32 this mode is called BDM. The microprocessor is halted and put into a special state so that you can perform debug operations.

76
If the system is in the error state, attempt to recover by reinitializing the system:

```c
    case ACE_MODE_ERR: /* Error */
        status = ::ACE_Initialize (emulator_, ACE_INN_COMMAND);
        if (status != ACE_SUCCESS)
            
```

If the system is in any other state, an error message is logged and 

Ace_T::stateBDM_m() fails.

```c
    case ACE_MODE_NOCONNECT: /* Not connected... */
    case ACE_MODE_SIM: /* Simulation mode */
    case ACE_MODE_PFA: /* Performance */
        status = ACE_FAILURE;
        WPWR_LOG_ERR ("Ace_T::stateBDM_m () : invalid state %s.\n",
                      ::ACE_OperatingModeImage (oldState));
        break;
    default:
        status = ACE_FAILURE;
        WPWR_LOG_ERR ("Ace_T::stateBDM_m () : invalid mode %d.\n",
                       oldState);
        break;
    }
```

The function Ace_T::state Restore_m() returns the system to its previous state by following a similar pattern.

You must also implement halt_m() and unhalt_m(), which wrap stateBDM_m() and stateRestore_m(). Note that oldTgtState_ is a data member of Ace_T:

```c
UINT32 Ace_T::halt_m ()
{
    UINT32          status;
    status = stateBDM_m (oldTgtState_);
    return (status);
}
Implementation Hints for Other Context-Management Methods

Now that you have implemented state management, you can implement the other context-management methods. Remember to add state management to the memory and register methods you have already written! Here are the key considerations to be aware of when implementing these methods:

- `eventpointAdd_m()` should support software breakpoints. If possible, support hardware breakpoints as well.

- `contextStep_m()` should support stepping out of a range of addresses, and stepping just a single instruction. Remember to lock interrupts when stepping if the emulator does not automatically do so. If interrupts are not locked, stepping will always result in stopping in an interrupt service routine, because external interrupts typically continue to occur when a system is halted and will cause the program counter to acquire a value outside of the targeted address range.

- When CrossWind requests a step-range operation, it expects execution to stop on the ending address specified. The ACE C API, however, stops when the program counter has stepped past the ending address. Consequently, `Ace_T::contextStep_m()` compensates for this by calling the ACE C API function with an ending address one less than the address specified by CrossWind.

- Finally, the Tornado tools expect a breakpoint event to be generated when the step-range request completes. The ACE C API does not automatically generate such an event. Consequently, `Ace_T::contextStep_m()` must synthesize a breakpoint event.

See `acecpu32Backend.cpp` for a sample implementation.

Again, use `wtxtcl` to test the context-management methods. Download some code to the target; if possible, download VxWorks using existing emulator tools. Otherwise, download code which performs a process such as spinning in a loop incrementing a counter. Try setting breakpoints, continuing, stepping, etc. Make sure that the program counter changes appropriately.

Implement Event Handling

The only core feature which has not yet been implemented is event handling. Event handling allows the tools to become aware of target events such as hitting a breakpoint or generating an exception. There are several components to back-end event management:
- asynchronous event notification
- synchronous event notification
- capturing event information
- responding to exception events

**Asynchronous Event Notification**

The target server is notified of events asynchronously by activity on the event file descriptor specified in the TGT_OPS structure’s tgtEventFd field. The target server monitors this file descriptor via `select()` . When the file descriptor becomes active, the target server invokes `eventGet_m()` to get the WDB_EVT_DATA structure describing the event. It then clears the event by calling `eventPendingClear_m()` . Typically, the event file descriptor is the socket file descriptor associated with a network connection to the emulator. In any case, the back end must implement the `fdGet_m()` method which returns an event file descriptor for the target server to monitor.

When an event occurs, the back end needs to capture the WDB_EVT_DATA information associated with the event and store it on a queue, because more than one event could occur before the target server handles them. The target server retrieves this data by calling `eventGet_m()` .

**Synchronous Event Notification**

When necessary, the target server will explicitly query the back end to determine if an event has occurred by calling `evtPending_m()` . This method should return TRUE if an event has occurred and has not yet been processed by the target server.

**Capturing Event Information**

Whenever an event occurs, the back end should capture the event’s information in a WDB_EVT_DATA structure and store this information for the target server. The acecpu32 back end manages target information by using a call-back provided by the ACE C API to capture event information and put it on the event queue, `Ace_T::eventQueue_`. When a target event occurs, the ACE API invokes the call-back. The call-back constructs an Event_T object, which maintains the WDB_EVT_DATA information for the event, and puts it on the event queue. Note that `Ace_T::eventCallBack()` is a static member function so that the ACE C API can call it.

```c
void Ace_T::eventCallBack
    (ACE_ConnectionHandle emulator,
```
Ace_T::pTheAceBkEnd_s() provides a safe way to get a pointer to the Ace_T back end, without needing to pass the this pointer around.

```cpp
ACE_Event * pEvent,
void * pData
)
{

Ace_T::pTheAceBkEnd_s()
provides a safe way to get a pointer to the Ace_T back end, without needing to pass the this pointer around.

// Get a back end pointer
Ace_T * pBackend = pTheAceBkEnd_s();

The Event_T constructor, shown below, captures the event information and creates the right kind of event.

// Create an appropriate Event_T object
Event_T * pWdbEvent = new Event_T (pEvent);

// Queue the event
pBackend->eventQueuePut_m (pWdbEvent);
}

The Rogue Wave Tools.h++ library makes the implementation of the queue mechanism easy. The event queue is a RWSlistCollectablesQueue and Event_T is derived from RWCollectable, so it is not necessary to write any code to implement and debug the event queue. (See Figure 2-8 and Figure 2-9.)

The Event_T constructor captures the event information by determining what kind of event occurred, gathering the necessary event data, and storing it:

```cpp
Event_T::Event_T (ACE_Event * pEvent)
:
RWCollectable ()
{

// Transfer pEvent into WDB_EVT_DESC
switch (pEvent->any.type)
{
    case ACE_BREAKPOINT:
        bpEventMake_m (pEvent);
        break;
    case ACE_SIGNAL:
        signalEventMake_m (pEvent);
        break;
    case ACE_MODECHANGE:
        modeChangeEventMake_m (pEvent);
        break;
    default:
        wdbEvent_.evtType = WDB_EVT_NONE;
        WPWR_LOG_ERR("Invalid ACE event type %#x.\n",
            pEvent->any.type);
It is easy to implement `eventGet_m()`, `evtPending_m()`, and `evtPendingClear_m()` using the `Event_T` class and event queuing mechanism, because they only need to manage the event queue using the Rogue Wave member functions.

### Exception Events

Exception events are more complicated to handle than other types, because it is necessary to provide information about the exception stack frame in the event data. The easiest way to gather exception information is to install an exception hook on the target to capture the data, notify the host, and suspend the system. For CPU32 microprocessors, this can be implemented using the special `bgnd` instruction which puts the target in background mode. If a similar notification scheme is not available on your host, you can set a breakpoint on your exception hook and, on every breakpoint event, check to see if it was triggered by the breakpoint in the exception hook.

The `acecpu32` back end uses an approach based on the `bgnd` instruction. First, a special library, `bdmExcLib`, is linked with VxWorks; in the `usrInit()` startup code, `bdmExcLibInit()` is called to initialize the library and install the exception hook. When the `bgnd` instruction is executed, the ACE API generates an `ACE_SIGNAL` event, and the back end builds the appropriate exception event.

A global variable, `bdmExcInfo`, is used to store information about the exception so that the back end can easily locate this information and upload it to the host for processing.

```
BDM_EXC_INFO bdmExcInfo = {0, 0, 0, FALSE};
...
STATUS bdmExcLibInit (void)
{
    if (bdmIsInitialized)
        return (ERROR);
    if (_func_excBaseHook != NULL)
        return (ERROR);

    _func_excBaseHook is a hook to allow a developer to run code whenever an exception occurs. Whatever function’s address is stored in it will be invoked by VxWorks’ exception handler whenever the exception occurs. With the `acecpu32`
back end, the VxWorks exception-handling code invokes `bdmExcHook()` whenever an exception occurs, which gathers the necessary information about the exception for the back end.

```c
/* Install exception hook */
__func_excBaseHook = bdmExcHook;
bdmIsInitialized = TRUE;

return (OK);
}

LOCAL int bdmExcHook
{
    int       vec,
    char *    pESF,
    WDB_IU_REGS * pRegs
}

bdmExcInfo.bdmIsException is equal to TRUE only if the `bgnd` instruction is executed by the hook for gathering exception information. We gather the other necessary information about the exception, suspend the system, and notify the back end of the exception by executing the `bgnd` instruction. When we resume the system, we clear the `bdmExcInfo.bdmIsException` flag so that an exception can be differentiated from the execution of a user’s `bgnd` instruction:

```c
    { 
        bdmExcInfo.bdmIsException = TRUE;
        
        bdmExcInfo.excVector       = vec;
        bdmExcInfo.pESF            = pESF;
        bdmExcInfo.pIuRegs         = pRegs;

        __asm__ (".word 0x4afa"); /* 'BGND' */
        
        bdmExcInfo.bdmIsException = FALSE;
        
        /* Return FALSE so that VxWorks' exception handler will continue */
        
        return (FALSE);
    }
```

Once an exception has occurred, until execution is resumed on the target, requests for register information should return the register information pointed to by `pRegs` rather than the current CPU register values. For an example, see `Ace_T::regsGet_m()` in `acecpu32.cpp`.
Testing

Once event handling has been implemented, CrossWind, Tornado’s source-level debugger, should operate with the new back end. First test the back end’s new functionality using \texttt{wtxcl}. (Note: to register \texttt{wtxcl} for an event use the \texttt{wtxRegisterForEvent} command. For more information, see \textit{4. The WTX Protocol}.) Then bring up CrossWind.

A typical CrossWind test session will proceed as follows:

1. Start the target server.
2. Invoke CrossWind:
   \begin{verbatim}
   % crosswind -t targetArch &
   \end{verbatim}
   Use \texttt{targetArch} to specify your target’s architecture, \texttt{68k}, \texttt{sparc}, \texttt{i960}, etc.
3. Attach CrossWind to the target server.
   \begin{verbatim}
   (gdb) target wtx targetServerName
   \end{verbatim}
4. If the back end is slow, or if you are using a debugger to test the back end, increase the WTX timeout (specified in seconds) so that CrossWind will not timeout when issuing commands to the target server:
   \begin{verbatim}
   (gdb) tcl wtxTimeout 6000
   \end{verbatim}
   The \texttt{tcl} command causes CrossWind to pass the remainder of the line to the \texttt{wtxcl} interpreter. 60 is the default timeout.
5. Emulators typically provide only system-level debugging, so put CrossWind in external debug mode:
   \begin{verbatim}
   (gdb) attach system
   \end{verbatim}
6. Set the target’s program counter to the address where you want to begin execution. You can use the symbol \texttt{sysInit} for the entry-point of bootable VxWorks:
   \begin{verbatim}
   (gdb) set $pc=sysInit
   \end{verbatim}
   Or you can point directly to the address:
   \begin{verbatim}
   (gdb) set $pc=0x404000
   \end{verbatim}

\textbf{NOTE:} The following steps show how to enter the commands in CrossWind’s command panel. Some of these commands could also be performed using the GUI.
7. Set breakpoints at key milestones in VxWorks startup: \texttt{usrInit()}, \texttt{sysHwInit()}, \texttt{usrRoot()}, and so on:

\begin{verbatim}
(gdb) b usrInit
(gdb) b sysHwInit
(gdb) ...
\end{verbatim}

8. It is also helpful to have \texttt{usrRoot()} spawn a demo application by editing the configuration file \texttt{installDir/target/h/usrConfig.h} (where \texttt{usrRoot()} is configured) as shown below:

Example 2-2  Spawning a Demo Task

\begin{verbatim}
/* usrConfig.c - user-defined system configuration library */
...
extern void usrLoop (int arg1, ...);

usrRoot
{
    char * pMemPoolStart, /* start of system memory partition */
    unsigned memPoolSize /* initial size of mem pool */
}
...
#ifdef INCLUDE_SHELL
...
#endif /* INCLUDE_SHELL */

/* XXX - for BDM testing */
taskSpawn ("uLoop", 200, 0, 10000, (FUNCPTR) usrLoop,
            0,0,0,0,0,0,0,0,0);
}
/* XXX - for BDM testing */
SEM_ID    loopSemId;
int       loopCntGlobal;
char *    loopString = "XXXThis is a string!\n";
int       loopDelay = 1;
BOOL      causeExc = FALSE;

void usrCauseExc (void)
{
    volatile int top     = 1;
    volatile int bottom  = 0;

    top = top / bottom; /* generate exception */
}
\end{verbatim}
void usrLoop (int argl, ...)  
{
    static   int loopCnt = 0;
    volatile int ix      = 0;

    loopString++;  
    loopString++;  

    if (loopSemId == NULL)  
        loopSemId = semCCreate (SEM_Q_PRIORITY, 1);  

FOREVER  
{
    taskDelay (loopDelay);  

    printf ("
...I am alive...Count = %d\n", ix);  

    ix++;  
    loopCnt         = ix;  
    loopCntGlobal    = ix;  

    semGive (loopSemId);  

    if (causeExc)  
    {
        usrCauseExc ();  
    }  
}

When you are ready to test your application’s exception handling, change  
causeExec to TRUE to generate an exception.  

9. Start execution on the target:  

   (gdb) cont

NOTE: The WindSh and Browser should come up also, but they may not work  
correctly until the system is multitasking, in other words, until usrRoot() has been  
entered. Don’t use them until the system is multitasking. Once this stage is  
reached, another key milestone is to get the i() command to work from the  
WindSh.  

Compiling and Loading VxWorks for Debugging  

Do not forget to compile VxWorks with the -g flag to generate debugging symbols  
for the system startup code.  

Since VxWorks is a fully linked module, the default load flags for the CrossWind  
load command are not appropriate for loading it. The correct command from the  
CrossWind command line is:
(gdb) wtx-obj-module-load LOAD_GLOBAL_SYMBOLS|LOAD_FULLY_LINKED vxWorks

To simplify the process, you can create a new gdb command by placing the following file in $HOME/.wind/gdb.tcl:

```tcl
# gdb.tcl - User customizations to CrossWind’s GDB engine
#
# modification history
# ----------------------
#
# 01a,07nov96, bss written.
#
#
# loadf - load a fully linked object module
#
proc loadf (file) {
    wtxObjModuleLoad LOAD_GLOBAL_SYMBOLS|LOAD_FULLY_LINKED $file
}
#
# register the new command with GDB

gdb tclproc loadf loadf
```

### 2.4.5 Implementing Optional Member Functions

The basic back-end functionality is now operational. If you have the resources, you may decide to support the optional methods `serviceCall_m()`, `directCall_m()`, `contextCreate_m()`, `contextKill_m()`, and `funcCall()`. You may also decide to implement some of the memory operations provided by `Backend_T`, such as `memScan_m()` or `memChecksum_m()`, particularly if your emulator provides a faster way of performing these operations than `Backend_T`'s generic implementation.

Dynamic loading of object modules onto the target requires, in addition to a high bandwidth memory access capability, support for `cacheTxtUpdate_m()` and `directCall_m()`. The required emulator support for `cacheTxtUpdate_m()` may not be available. Implementing `directCall_m()` is complex: it involves saving the processor’s state, setting up a stack for the call, performing the call, notifying the host that the call has completed, and then returning the system to its previous state. Consequently, most emulator back ends do not support dynamic loading.
2.4.6 Performance

Memory access is usually the biggest performance bottleneck. Everything possible should be done to optimize the speed of memory reads and writes. Also, optimize the register read and write operations for the case of a single register. Finally, if your emulator supports primitives for performing common memory operations such as checksum, scanning for a pattern, and filling a block of memory with a pattern, over-ride `Backend_T`'s implementation and use methods optimized for your emulator.
Object-Module Loader

3.1 Introduction

The object-module loader is the component of the target server that inserts application modules into a remote target system while the target image is running. Loading and unloading object modules dynamically is a key service provided by the target server. The target server provides this and other services commonly required by the Tornado tools in addition to acting as a broker for the communication path to the target.

You can support additional object-module formats with Tornado by writing a new loader. The following sections provide an overview of loader usage and loader architecture, followed by a discussion (in general terms) of how to develop a reader for a new object-module format.

For an overview of the target server, the back end, and their components, see 1.3 The Target Server and the WTX Protocol, p.3 and 2. Target Server Back End. This chapter provides an overall orientation to object-module loaders; throughout this chapter, refer to the reference entries in Part 2 for details.
3.2 Terminology

Although the following terms are in common usage, we review the distinctions among them:

**object file**
The unrelocated output of a compiler or assembler (conventionally produced with a compiler’s `-c` option, and named with a `.o` suffix).

**object module**
The contents of an object file.

**object module format (OMF)**
The convention in which the data in an object module is organized. Several different OMF standards exist, such as `a.out`, COFF, and ELF. On the whole, OMFs are independent from the underlying architecture. However, the relocation information contained in object modules is architecture dependent.

**relocatable file**
An object file for which text and data sections are in a transitory form, where some addresses are not yet known. These sections must be modified (relocated or linked) when they are loaded in order to be executable. In a cross-development environment, information not known about an object module at compilation time includes the program execution address (also called the entry point) and the addresses of externally defined symbols (such as library routines). By default, the loader assumes that some module addresses are unknown and that relocation is required.

**executable file**
A file that is fully linked and ready to run at a specified address. For such files, no relocation stage is required. The loader is able to load files that do not require a relocation stage, but it must be told that the file type is fully linked.

**object module header**
The portion of an object module that holds various information, such as the type of object module (often called the magic number) and the sizes of the different sections.

**raw data**
The main part of an object module, corresponding to the program code and data. The loader can handle four kinds of sections:

- **Text sections** hold the program code.
– **Data sections** hold the initialized program variables.
– **Bss sections** hold the uninitialized program variables.
– **Literal sections** hold the program constants and strings.

Depending on the OMF, an object module may contain several sections of each type. (This is true of both COFF and ELF.) It is also possible that the text and data sections may be the only two sections in the module. (The *a.out* format follows this model; however, *a.out* sections are typically referred to as segments.)

**symbol entries**
Information about every symbol defined (*local symbols*) or referred to (*external symbols*) in the object module. The term *symbol* refers to every variable, routine, or constant that is accessible from other parts of the program (*public*).

**relocation entries**
Information required to adapt the program to its execution environment. The relocation entries point to each place where a relocation is required, and indicate the type of operation required to perform it.

**string table**
A table containing all symbol strings. For *a.out* and ELF it holds all strings; for COFF it holds only strings larger than a certain size.

### 3.3 Loader Overview

**Linking and Loading**

The following steps are required to link and load a module onto the target:

- Read and analyze the object file.
- Process the object-module symbols.
- Relocate the symbols.
- Transfer the module code and data to the target environment.

These steps are apportioned between two internal parts of the loader: symbols are relocated by the *relocation unit* (RU - the portion of the loader that performs relocations) and all other steps are performed by the *object-module-format manager* (OMF manager).
Bootstrapping the Target Server Symbol Table

In addition loading modules to the target, the loader plays an additional role when the target server establishes a connection with a target agent. The loader attempts to obtain the pathname to the core file from the agent. The core file is the executable file on the host that corresponds to the runtime system initially executing on the target. This file is not downloaded to the target system because the core image is already present there (typically in PROM). If the core file is found, the loader reads this file in order to bootstrap the target server symbol table. If the checksums of the text regions do not agree, the loader prints an error message, because a symbol table built from this core file might not contain the correct addresses on the target.

If the core file is not known, the target server exits with an error message. Try connecting the target server again using the -c option to specify the core file path and name.

3.4 Using the Target Server Loader

Tornado provides access to the loader by several routes:

- the Tornado shell
- the WTX Tcl API
- the WTX C API
- the WTX Java API

3.4.1 Loader Usage from the Tornado Shell

The Tornado shell provides an interpretive environment that offers easy access to the loader. From the shell, access the loader through the ld() command:

```
-> ld symbolsVisibility, commonSymbolsPolicy, "objectFile"
```

The symbolsVisibility argument may be:

-1 Load no symbols to the target server symbol table (mutually exclusive with other options). There is no way to know the contents of the module.

0 Load only global symbols to the target server symbol table. This is the default.

1 Load both local and global symbols to the target server symbol table.
The `commonSymbolsPolicy` argument may be:

0  Keep common symbols isolated, visible from the object module only. This option prevents any matching with already-existing symbols (in other words, the relocations that refer to these symbols are kept local). Common symbols are added to the symbol table unless `LOAD_NO_SYMBOLS` is set. This is the default option with behavior equivalent to `LOAD_COMMON_MATCH_NONE`. (For definitions of these and other options, see 3.5.4 Loader Options, p.101.)

1  Seek a matching symbol in the target server symbol table, considering only symbols in user modules. If no matching symbol exists, the behavior is equivalent to `LOAD_COMMON_MATCH_NONE`.

2  Seek a matching symbol in the target server symbol table, considering all symbols. If no matching symbol exists, the behavior is equivalent to `LOAD_COMMON_MATCH_NONE`.

For a complete discussion of loader behavior, see 3.5 Loader Architecture, p.95.

To call the loader from the shell, enter:

```
-> ld 0, 0, "/home/users/me/myFile.o"
```

You can also use input redirection:

```
-> ld </home/users/me/myFile.o
```

### 3.4.2 Loader Usage From wtxtcl

Another way to access the loader facilities of the target server is through the WTX Tcl binding to the WTX protocol. This environment offers the scripting advantages of Tcl and the ability to manipulate the WTX protocol directly. From the `wtxtcl` tool (see 4.4.2 Starting a wtxtcl Session, p.168), access the loader through the `wtxObjModuleLoad` routine:

```
wtxObjModuleLoad [option] object-file
```

The `option` argument is an optional parameter from the following list:

- `LOAD_NO_SYMBOLS`
- `LOAD_LOCAL_SYMBOLS`
- `LOAD_GLOBAL_SYMBOLS`
- `LOAD_ALL_SYMBOLS`
- `LOAD_FULLY_LINKED`
- `LOAD_NO_DOWNLOAD`
- `LOAD_CORE_FILE`
- `LOAD_COMMON_MATCH_NONE`
- `LOAD_COMMON_MATCH_USER`
- `LOAD_COMMON_MATCH_ALL`
- `LOAD_HIDDEN_MODULE`
- `LOAD_BAL_OPTIM`
- `LOAD_FILE_OUTPUT`
You can combine several options by linking them with the “|” character. For a detailed discussion of these parameters, see 3.5.4 Loader Options, p.101.

Examples:

```
wtxtcl> wtxObjModuleLoad /home/users/me/myFile.o
wtxtcl> wtxObjModuleLoad LOAD_ALL_SYMBOLS /home/users/me/myFile.o
wtxtcl> wtxObjModuleLoad LOAD_GLOBAL_SYMBOLS|LOAD_COMMON_MATCH_USER \ /home/users/me/myFile.o
```

### 3.4.3 Loader Usage From the WTX C API

You can also call the loader from C programs on the host, using the WTX C language binding. For details about using `wtxObjModuleLoad()`, see the online reference material under Tornado API Reference>WTX C Library. For general information on using C, see 4.5 WTX C API, p.185.

### 3.4.4 Asynchronous Load Operation

The target-server loader implementation allows you to submit load operations asynchronously. Three APIs provide routines that support the loader: the Tcl, C and Java APIs. Each API provides three routines to handle this feature. The routines are listed below according to the Tcl convention:

- `wtxObjModuleLoadStart`
  - Submit a load operation. This routine returns when the file has been written to the target server memory.

- `wtxObjModuleLoadCancel`
  - Cancel a submitted load.

- `wtxObjModuleLoadProgressReport`
  - Get information about the status of a submitted load. The returned information can be one of the following:
    - `LOAD_PENDING`
      - The load is in queue.
    - `LOAD_INITIALIZED`
      - The being processed.
    - `LOAD_RELOCATING`
      - The loader is relocating symbols.
LOAD_DOWNLOADING  
The loader is downloading the segments on the target.

LOAD_COMPLETE  
The load has been done.

LOAD_ABORTED  
The load has been cancelled.

3.5 Loader Architecture

The loader has a three-layered structure. At the generic level is a library of routines that are common to all configurations. At the object-module format level are several OMF managers to handle different object-module formats. At the architecture level are several RUs (relocation units), each supporting a specific architecture. Figure 3-1 illustrates the relationship between the three levels.

Figure 3-1 Target Server Loader Architecture
3.5.1 Object-Module Configuration in Target Memory

Since the object-file sections hold a program (a set of routines), and since this set of routines must be installed in the real target memory, there is a strong relationship between section configuration in the file and in the target memory. How the loader locates the sections in memory depends on whether the file type is relocatable or fully linked.

Relocatable Files

The loader uses a three-segment model for loading relocatable modules. It coalesces the module sections into three segments in target memory: a text segment, a data segment, and a bss segment. It combines sections of the same or compatible type into one segment (see Figure 3-2). Note that the order of coalescence is strictly the order in which the text and literal sections exist in the file.

Figure 3-2 Section Combination in Relocatable Files
Because the loader creates only three segments, it handles literal sections in the same way as text sections. If literal sections are present in a relocatable object module, the loader coalesces them with the text sections. The sections are similar in that contents of both types of sections must not be modified while the program is executing.

**NOTE:** The RU also treats other constant data sections, such as ELF `.rodata` sections, as literal sections and coalesces them into the data segment.

**Fully Linked Files**

The loader retains the file configuration for fully linked files. Each section is represented separately in target memory. This allows an application to take advantage of a target system with non-contiguous RAM or with high-speed static RAM and dynamic RAM.

### 3.5.2 Module Management

The generic module management library keeps track of what modules have been loaded in memory, where their segments are, how much memory they use, and which symbols belong to which modules.

During loading, a temporary data structure, SEG_INFO, describes the contents of an object module. The SEG_INFO structure contains the following elements:

- the text segment address
- the data segment address
- the bss segment address
- the text segment size
- the size of the text segment that is write-protected (if any)
- the data segment size
- the bss segment size
- the text segment option for memory management
- the data segment option for memory management
- the bss segment option for memory management

When the loader finishes loading the object module, it transfers the SEG_INFO contents to a more permanent representation. Each module representation is a member of a linked list and is located by means of a MODULE_ID handle. The module representation contains the following information:
- the name of the object file
- the symbol group number
- the format of the object module (a.out, COFF, ELF, and so on)
- the target CPU type
- the loader options (also known as load flags)
- a list of the object-module undefined symbols (if any)
- a linked list of the object-module segments

Each segment in the linked list has a representation that contains:

- the segment address
- the segment size
- the segment type (text, data, or bss)
- the memory management information
- the segment checksum

Type definitions for both MODULE_ID and SEG_INFO are in module.h.

The ability to unload a module is an important outgrowth of module management. Unloading a module implies freeing the memory associated with the module segments and removing the module symbols from the target server symbol table.

When creating a new OMF manager, you can use the code from existing loaders as a model for providing module management.

### 3.5.3 Symbol Management

The loader processes the object-module symbols by reading the file symbol table. Part of this processing is required for relocation. Additional processing is necessary to add the module symbols to the target server symbol table.

The target server symbol table holds all the symbols related to the remote system, including symbols from the target agent and symbols from dynamically loaded user modules. Maintaining this table within the target server on the host saves memory on the target and gives host tools access to all public symbols (and local symbols, if desired) that are present on the target system. The target server symbol table holds one entry per symbol, with the following information:

- the name of the symbol
- the address of the symbol
- the segment type of the symbol (text, data, or bss)
- the object-file name from which the symbol comes
The target server symbol table is filled at target server start-up time with the core-file symbol table. (For more information on this process, see *Bootstrapping the Target Server Symbol Table*, p.92.)

**Symbol Processing**

When a relocatable file is loaded, its symbol table is analyzed and the symbols are processed depending on the categories they belong to:

- **Defined Symbols**
  
  These symbols refer to objects (routines, variables, or constants) that are present in the module text, data, or bss segments. They may be static (their scope limited to the module itself) or global (accessible to objects outside the module). Defined symbols need no relocation and may be added directly to the target server symbol table if desired.

- **Undefined Symbols**
  
  These symbols are referred to in the module text or data but are not present in the module. They must be global. For such symbols, the loader must locate the symbol in the target server symbol table in order to relocate the module. If the symbol is not found, it is considered **unknown** and its name is added to a list that is returned to the tool requesting the load.

**WARNING:** While a module containing undefined symbols is downloaded anyway, the consequence is that the module may be partly or totally unusable since no relocation can be done on the unknown symbols. This partial linkage permits testing pieces of an application during development as long as the tested pieces do not hold references to undefined symbols.

- **Common Symbols**
  
  These symbols are handled as a special case. Consider the example below:

  ```c
  #include <stdio.h>
  int willBeCommon;
  void main (void) {}
  {
  ...
  }
  ```

  The symbol `willBeCommon` is uninitialized, so it is technically an undefined symbol. Such a definition becomes **common**. ANSI C allows multiple files to use
uninitialized symbols to refer to the same variable and expects the loader to resolve these references consistently.

It is often helpful in an incremental environment to be able to share a symbol among several files and to load the files in any order; however, it is also extremely risky to treat common variables in this way. When linked, a common definition could resolve to any defined symbol in the symbol table depending on what symbols are defined in already-loaded modules. The default for Tornado treats common symbols as undefined. However, you can set the loader options to permit common symbols to be loaded; then only the first instance of the symbol will be added to the symbol table and other instances will use the same address. For more information, see 3.5.4 Loader Options, p.101.

NOTE: External symbols are public (visible to everyone both within and outside the file that declares them). The term external is not used in this document to mean “declared outside the file” (as opposed to “declared inside the file”).

Symbol Type Definitions

The loader defines a set of symbol types (SYM_TYPE) in symbol.h in order to handle the symbols more easily. The loader uses the types to classify the module symbols. The symbol types defined by the loader are:

SYM_UNDEF
The symbol is not defined in the module.

SYM_LOCAL
The scope of the symbol is limited to the module.

SYM_GLOBAL
The symbol is accessible to anyone.

SYM_ABS
The symbol has an absolute value that must be kept as it is.

SYM_TEXT
The symbol belongs to the text segment.

SYM_DATA
The symbol belongs to the data segment.

SYM_BSS
The symbol belongs to the bss segment.

SYM_COMM
The symbol is a common symbol.
3.5.4 Loader Options

The loader behavior may be tuned by using options. Many of these options may be combined, although some are mutually exclusive. The option names given below are the names used internally by the loader. They are defined in loadlib.h.

Symbol Scope

As explained in 3.5.3 Symbol Management, any defined symbol may be added to the target server symbol table. Several options allow the user to specify exactly which symbols should be added to the table:

LOAD_LOCAL_SYMBOLS
Add only local symbols to the target server symbol table. Note that this may include some debugging symbols, depending on the OMF.

LOAD_GLOBAL_SYMBOLS
Add only global symbols to the target server symbol table (the default).

LOAD_ALL_SYMBOLS
Add both local and global symbols to the target server symbol table.

LOAD_NO_SYMBOLS
Add no symbols to the target server symbol table. (This option is mutually exclusive with others.) With this option, there is no way to know the contents of the module.

WARNING: These options affect not only the visibility of the symbols but also the load operations that occur after the module is loaded. For instance, if you load a module with the LOAD_NO_SYMBOLS flag set, none of its global symbols is added to the target server symbol table. Not only are they hidden from the other modules, but they are also unreachable; no reference is possible to these symbols. If you load another object module that refers to a symbol (for example, a routine) in the previous module, the loader cannot find the reference and the symbol is considered unknown. It is also impossible to call a routine within such a module symbolically from the shell. The only way to call the routine is by using its address.
Module Visibility

When a module is loaded, the target server normally records it. Any tool can request and receive information related to the module. This default behavior can be changed by using the following option:

LOAD_HIDDEN_MODULE
Do not keep the module information. Note that this option does not hide the module symbols. To do so, it must be combined with LOAD_NO_SYMBOLS.

Module Type

The loader default considers all files relocatable object-modules requiring a relocation stage (see 3.5.1 Object-Module Configuration in Target Memory, p.96). To load files that do not require a relocation stage, the loader must be told that the file type is fully linked. Two options are available for fully linked files:

LOAD_FULLY_LINKED
Required for any fully linked file. When it is used, no relocation is done.

LOAD_NO_DOWNLOAD
This option suppresses downloading the file to target memory. The relocation is done in the target server cache only.

LOAD_CORE_FILE
It is used exclusively at connection time to load the core file. This is a combination of LOAD_NO_DOWNLOAD and LOAD_FULLY_LINKED.

⚠️ CAUTION: The loader does not allocate memory on the target system when a fully-linked file is loaded as it does for relocatable files. The file segments are located at the addresses described in the file header; this depends on the OMF.

Common Symbols

Three mutually exclusive options provide the ability to specify how the loader handles common symbols. They are listed in order of decreasing strictness:

LOAD_COMMON_MATCH_NONE
Keep common symbols isolated, visible from the object module only. This option prevents any matching with already-existing symbols (in other words, the relocations that refer to these symbols are kept local). Common symbols
are added to the symbol table unless LOAD_NO_SYMBOLS is set. This is the default option.

LOAD_COMMON_MATCH_USER
Seek a matching symbol in the target server symbol table, but consider only symbols in user modules. If no matching symbol exists, act like LOAD_COMMON_MATCH_NONE.

LOAD_COMMON_MATCH_ALL
Seek a matching symbol in the target server symbol table, but consider all symbols. If no matching symbol exists, act like LOAD_COMMON_MATCH_NONE.

If several matching symbols exist for options LOAD_COMMON_MATCH_USER and LOAD_COMMON_MATCH_ALL, the order of precedence is: symbols in the bss segment, then symbols in the data segment, then symbols in the text segment. If several matching symbols exist within a single segment type, the symbol most recently added to the target server symbol table is used.

Special Options

The loader accepts special options, with scope limited to specific cases:

LOAD_BAL_OPTIM (i960 targets only)
Replace the i960 CALL opcode with the BAL opcode. This makes the loader slower, and is not a default option.

LOAD_FILE_OUTPUT
Dump the module text and data segments to a file instead of to target memory. This option is used only for testing. LOAD_FILE_OUTPUT is mutually exclusive with LOAD_FULLY_LINKED.

3.6 Memory Management

The loader manages memory allocated from two entirely different memory pools. The loader is a component of a host process, the target server, and is able to allocate and free memory from the host heap using standard routines such as malloc() and free(). However, the loader also manipulates data that it then transfers to the target system. The transfer requires the loader to allocate and free memory on the remote
The loader does this through an internal library that manages a region of target system memory (the target server memory pool). The memory manager on the host also implements a host cache for portions of the target memory; this minimizes the number of data transfers to and from the target.

To understand how the target server manages the target server memory pool, refer to the online reference material under Tornado API Reference>Target Server Internal Routines. The routines listed are ANSI-compliant implementations; they have unique names to avoid conflicting with the host-system C library.

The interface to target server cache management is \texttt{tgtMemCacheSet()}. This routine specifies the attributes of a block of host-mapped target memory. Blocks defined through this routine must not overlap. You must not change block boundaries once a block is created; however, you may call this routine again on a previously defined block to change its attributes.

The memory attributes of the target server memory pool are the following:

- \texttt{MEM\_NONE}
  
  Return the region to the default attribute.

- \texttt{MEM\_NO\_WRITE}
  
  Flag as errors any writes to these memory regions by the host.

- \texttt{MEM\_HOST}
  
  Allocate a block of memory to store all reads and writes locally. Do not modify target memory when writing to this region. This allows a module to be fully linked on the host before actually downloading it to the target.

- \texttt{MEM\_TEXT}
  
  Allocate and store a block of memory on the host. Perform reads locally, but update both the local block and the corresponding address on the target when writing (like a write-through cache). This attribute is well suited for text segments once a module is fully linked.

To access target memory, all memory transfer requests must go through the routines \texttt{tgtMemRead()} and \texttt{tgtMemWrite()}. For more information, see the synopsis of \texttt{tgtlib} in the online reference material under Tornado API Reference>Target Server Internal Routines.

### 3.6.1 Memory Alignment

All memory obtained from the target server’s target-memory manager is free from memory alignment problems because the target server memory manager always
returns addresses that conform to the target alignment requirements. All
\texttt{tgtMemXxx()} routines gracefully handle memory alignment.

However, alignment problems may occur when the object-module headers are
processed by the loader. In general, the headers contain data that is \textit{packed} (without
padding) whereas the host architecture may require padding to represent the
header information. For example, 16-bit quantities are often enlarged to 32-bit
quantities by padding.

This can cause problems if the object module is not directly processed as a file but
rather read entirely into memory first and subsequently processed. This is done to
improve performance, but it can also lead to read-word operations at unaligned
addresses. Some host architectures accept this, but others do not. Use the
\texttt{UNPACK\_16} and \texttt{UNPACK\_32} macros (defined in \texttt{host.h}) to prevent portability
problems. These macros offer the ability to read 16-bit (or 32-bit) data regardless of
the alignment.

Another kind of alignment problem is related to the three-segment model imposed
by the target server loader. In this model, sections of the same type are loaded
together within a global area allocated for the coalesced segment. Each section is
aligned according to the requirement specified in the object file, or, if none is
specified, each is loaded at an address that fits within the target architectures
alignment requirement. The requirement specified in the object file cannot be
larger than the target alignment requirement. Because of alignment requirements,
two coalesced sections may require a “hole” between them (depending on their
sizes) to assure proper alignment. Be sure to account for this detail when
computing the size of the segment sufficient to hold the coalesced sections.

\subsection*{3.6.2 Target Server Memory Cache}

The goal of the target server memory cache is to speed up object-module download
and minimize the impact of module loading on target performance. It achieves this
goal by performing all object-module relocations in host memory and then
copying the final relocated module to the target.

In an ideal world, the target server memory cache would be the same size as the
target server memory pool on the target. Then the cache could shadow the entire
memory pool. In the real world, the size of the cache is limited by the size of host
memory. The default for the target server cache is 1 MB. Use the \texttt{-m cacheSize}
option to specify a larger size.

If the target server memory cache is smaller than the target server memory pool,
the first object modules load quickly. However, once the cache is full, each
relocation generates a memory-write transaction with the target. This slows the load process, especially if you are using a serial line.

When the target server memory pool is full and an additional module is loaded, the target server uses `malloc()` to request more memory. The target server always requests 1 KB more than it needs in order to limit the number of requests to the target for small chunks of memory.

Rules for sizing the cache and the target server memory pool:

- Always try to make the target server memory cache large enough to hold all the object modules to be loaded:

  ```
  target -m bigEnoughSize
  ```

- If you cannot allocate that much memory, make the cache size as large as your host can accommodate without swapping.

- If possible, install additional memory on the development host.

- Obtain optimum efficiency by making the `WDB_POOL_SIZE` in `config.h` at least as large as required to load all the object modules. However, this is not essential since it is extended as necessary on the fly.

### 3.6.3 Target Virtual Memory and Cache Management

Although the loader has memory attributes that can prevent corruption of target memory by the host (`MEM_NO_WRITE`), it cannot prevent the target from corrupting its own memory. Memory on the target side can be protected by the optional product VxVMI, which utilizes the CPU memory management unit (MMU) to designate regions of memory as read-only. The loader has access to this product (if it is installed) through the routine `tgtMemProtect()`. Such protection can be useful when text segments are loaded; they represent program code that must not be corrupted by the execution of another program.

The target instruction cache, if present, creates problems for the loader if it is not updated when modules are downloaded. If incoherence exists between the instruction cache and the target memory, incorrect instructions can be executed from the cache. To prevent this, the loader updates the target instruction cache by calling `tgtCacheTextUpdate()` each time it loads a module.
3.6.4 Type Abstractions and Address Manipulation

Type abstraction is a way to avoid dependencies on the host definition of the size of various types. Tornado uses types with explicit sizes whenever information from the object modules must be manipulated. This information includes, for instance, the header-field contents or any other structural information such as relocation entries.

Type abstractions can be found in `host.h` and `loadlib.h`.

The type `void *` is used whenever an address is manipulated (through the more convenient type `SYM_ADRS`).

3.7 Generic Loader-Library Interface

The target server includes a generic loader-support library that acts as a common part for all loaders. It includes the loader interfaces with the target server and provides utility routines that facilitate creation of new OMF managers and RUs.

For information about the generic loader-library interfaces, see the online reference material under Tornado API Reference>Target Server Internal Routines. To utilize this interface in writing a new OMF reader, see 3.9 Writing an OMF Manager, p.108.

3.8 Target Server Loader Thread

The target server has a specific thread which handles all the load operations. This design has the following advantages:

- The load process can be done asynchronously, so the client which submitted the load can return without waiting the load to complete.
- The OMF reader need not be reentrant because only a single thread performs the load operation.
- A load operation can be cancelled at user request.

Figure 3-3 illustrates how a load is performed by the target server.
The WTX threads (servicing a client) submit a load operation to the FIFO list. They return to the client when the memory module is loaded in the target server memory.

The target server load thread takes the first request and processes it. Once the module loaded on the target, the loader thread puts the module information into the “complete load” list for the WTX client.

If a WTX client cancels its submitted load, the loader thread stops the current load operation, and processes the next one.

### 3.9 Writing an OMF Manager

The remainder of this chapter describes how to write a new OMF manager. Code examples are based on Wind River Systems OMF managers. They do not cover every possible situation, but they do illustrate common OMF issues.

In the remaining sections, comments enclosed between square braces (\([comment]\)) in the code fragments indicate lines of code not detailed for the sake of clarity.
Function calls with Omf as part of the name, with or without parameters indicated by (...), are examples of how to use function calls.

### 3.9.1 Dynamically Linked Library Implementation

An OMF manager and its RUs form a single component, the OMF reader. The OMF reader is implemented as two dynamically linked libraries (DLL): one for the OMF manager and a second for the relocation unit. The DLLs offer the ability to load and unload portions of code at run time. They can be shared by programs without being linked into the programs. DLLs have several advantages:

- **Program Size**: Since only the required pieces of code are loaded, the program (in this case the target server) uses less memory.
- **Scalability**: It is possible to build an application dynamically from a reduced set of routines that exactly fits the user’s needs.
- **Flexibility**: It is easy to add function without recompiling the main program.

**NOTE**: DLLs should export only necessary data to maintain good performance.

The OMF-reader code is common to all supported platforms.

### 3.9.2 Installing a Shared-Library Manager

The target server reads installDir/host/resource/target/architecturedb when it starts. This resource file establishes the relationship between the target CPU family, the object-module format, the relocation unit, and the corresponding shared libraries. OMF and RU names are established according to the conventions outlined in 3.9.3 Naming Conventions, p. 110. Shared libraries are listed without their extensions because the extension is architecture dependent (.so for Solaris, .sl for HP-UX, and .dll for Windows NT).

At initialization, the target server attaches in turn each OMF manager listed in the resource file. It tests whether or not the OMF manager can handle the core file and if not, detaches it. When the target server finds an OMF manager that can read the core file format, it keeps that OMF manager attached and continues with initialization. If it finds no appropriate OMF manager, it exits with an error message.
### 3.9.3 Naming Conventions

In addition to the Wind River Systems coding conventions, specific naming conventions are adopted by the loader for the OMF-manager and RU interface routines, and for the related shared libraries file names.

#### Interface Routines

The `Omf` part of both OMF and RU routine names depends on the OMF strings supported for a given CPU, which are defined by **Object File Type** in the `installDir/host/resource/target/architecturedb` file. These strings are read by the target server and used to build the routine names. The following rules apply:

- Only the alphanumeric portion of the string is kept.
- No matter what letter case is used in the string, for OMF routines the first letter is converted to uppercase and all others are converted to lowercase. For example, “a.out” becomes “Aout” and “COFF” becomes “Coff”. For RU routines, all characters are lowercase.

The `Cpu` part of the RU routine name is defined in the **ExtensionName** field in the `installDir/host/resource/target/architecturedb` file.

- **OMF Manager:**

  The OMF manager has three interface routines: `loadOmfFmtManage()`, `loadOmfFmtCheck()`, and `loadOmfFmtInit()`. `Omf` is the only variant portion of the name and reflects the OMF as follows:

<table>
<thead>
<tr>
<th>OMF</th>
<th>Variant part</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.out</td>
<td>Aout</td>
<td>loadAoutFmtManage()</td>
</tr>
<tr>
<td>COFF</td>
<td>Coff</td>
<td>loadCoffFmtManage()</td>
</tr>
<tr>
<td>ELF</td>
<td>Elf</td>
<td>loadElfFmtManage()</td>
</tr>
</tbody>
</table>

- **RU:**

  The interface routines are similar to `omfCpuSegReloc()`. Again, `omf` and `Cpu` are the only variant parts; They reflect the OMF and the CPU as shown:
File Names

The variant portions of the file names are derived by rules similar to those for the RU routine names.

- Only the alphanumeric portion of the string is kept.
- No matter what letter case is used in the string, all letters are converted to lowercase. For example, “a.out” becomes “aout” and “COFF” becomes “coff”.

**OMF Manager:**

The file is called `loadomf` where `omf` is the only variant portion and reflects the OMF name:

<table>
<thead>
<tr>
<th>Table 3-2 RU Function Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMF</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td><code>a.out</code></td>
</tr>
<tr>
<td><code>a.out</code></td>
</tr>
<tr>
<td><code>a.out</code></td>
</tr>
<tr>
<td>COFF</td>
</tr>
<tr>
<td>COFF</td>
</tr>
<tr>
<td>ELF</td>
</tr>
<tr>
<td>ELF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3-3 OMF Manager File Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMF</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td><code>a.out</code></td>
</tr>
<tr>
<td>COFF</td>
</tr>
<tr>
<td>ELF</td>
</tr>
</tbody>
</table>
The file is called `omfcpu` where `omf` and `cpu` are the only variant portions and reflect the OMF and the CPU as follows:

<table>
<thead>
<tr>
<th>OMF</th>
<th>Cpu name</th>
<th>Variant Parts</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.out</td>
<td>MC68040</td>
<td>aout - 68k</td>
<td>aout68k</td>
</tr>
<tr>
<td>a.out</td>
<td>SPARC</td>
<td>aout - Sparc</td>
<td>aoutSparc</td>
</tr>
<tr>
<td>a.out</td>
<td>X86</td>
<td>Aout - X86</td>
<td>aoutX86</td>
</tr>
<tr>
<td>COFF</td>
<td>1960JX</td>
<td>coff - 1960</td>
<td>coff1960</td>
</tr>
<tr>
<td>COFF</td>
<td>ARM7TDI</td>
<td>coff - Arm</td>
<td>coffArm</td>
</tr>
<tr>
<td>ELF</td>
<td>PowerPC</td>
<td>Elf - Ppc</td>
<td>elfPpc</td>
</tr>
<tr>
<td>ELF</td>
<td>MIPS</td>
<td>Elf - Mips</td>
<td>elfMips</td>
</tr>
</tbody>
</table>

### 3.9.4 The RU Interface

Relocation is highly dependent on both the OMF and the target architecture. This is why the relocation unit is a separate shared library. Only one relocator can be used at a time with a given target, so only one relocation unit is linked into the target server at run-time. The relocation unit is loaded during the OMF initialization phase (see `loadOmfFmtInit()` on p. 114).

Each relocation unit must contain the following two interface routines or entry points:

- `omfcpuSegReloc()`
- `omfcpuModuleVerify()`

It may also contain an optional initialization routine:

- `omfcpuRelocInit()`

`omfcpuRelocInit()`

This optional routine, if it exists in the relocation unit, is the first entry point. Its purpose is to enable the relocation unit to initialize itself.
omfcpuModuleVerify() 

This routine is the first entry point if there is no optional routine; it checks whether or not the given module is for the current target CPU.

```c
BOOL omfcpuModuleVerify (uint32 machtype, BOOL * pSwapIsRequired);
```

The input parameter is:

**uint32 machtype**

The Module’s magic number.

The output parameter is:

**BOOL * pSwapIsRequired**

A pointer to a boolean saying if object module is in the opposite byte order and requires that all information be swapped.

The return value is:

**STATUS**

TRUE or FALSE.

This routine is the underlying routine called by loadOmfModuleIsOk() (see Example 3-4).

Example 3-1 omfcpuModuleVerify()

```c
/************************************************************************
* oMFcpuModuleVerify - check the object module format for cpu target *
* This routine contains the heuristic required to determine if the object *
* file belongs to the OMF handled by this OMF reader, with care for the *
* target architecture. *
* It is the underlying routine for loadOmfModuleIsOk(). *
* RETURNS: TRUE or FALSE if the object module can't be handled. *
*/

BOOL omfcpuModuleVerify
{
    UINT16 magicNumber, /* Module’s magic number */
    BOOL * SwapIsRequired /* TRUE if header fields must be swapped */

    { 
        BOOL moduleIsForTarget = FALSE; /* TRUE if intended for target*/

        *SwapIsRequired = FALSE;

        switch(magicNumber)
```c
{
    case (SWAB_16 (MYCPUMAGIC)):
        *SwapIsRequired = TRUE;
        break;
    case (MYCPUMAGIC):
        moduleIsForTarget = TRUE;
        break;
    default : 
        errno = WTX_ERR_LOADER_UNKNOWN_OBJ_MODULE_FORMAT;
    }
    return moduleIsForTarget;
}

omfcpuSegReloc()

This routine performs the relocation of a given segment. The relocation process is explained in 3.9.14 Relocating the Object Modules, p.142, and an example of a segment relocator is also explained in Example 3-14.

### 3.9.5 The OMF Interface

Each OMF manager must contain the following three interface routines or entry points:
- `loadOmfFmtInit()`
- `loadOmfFmtCheck()`
- `loadOmfFmtManage()`

*loadOmfFmtInit()*

This routine is the first entry point; it loads the correct relocation unit for the current target CPU.

```c
STATUS loadOmfFmtInit (void)
```

The return value is:

**STATUS**

OK or ERROR. (See `installDir/host/include/host.h` for a description of error types and macros.)
Example 3-2  loadOmFmtInit()

/************************************************************************
* loadOmFmtInit - initialize the OMF loader
* This routine initializes the correct relocator unit for the current target
* CPU.
* RETURNS: OK or ERROR.
*/

STATUS loadOmFmtInit (void)
{
    /* OMF relocs routines (+ 1 for the optionnal SegInit routine) */
    DYNLK_FUNC relocDllFv[NB_MANDATORY_RELOC_RTNS + 1];
    relocDllFv[0].name = (char *)tgtSegRelocatorRtnNameGet();
    relocDllFv[1].name = (char *)tgtModuleVerifyRtnNameGet();
    relocDllFv[2].name = (char *)tgtRelocInitRtnNameGet();

    /* Link the relocator */
    if (loadRelocLink( relocDllFv, NB_MANDATORY_RELOC_RTNS+1) != OK)
    {
        return ERROR;
    }

    /* relocation routine pointer */
    omfRelSegRtn = (FUNCPTR) relocDllFv[0].func;

    /* module verification routine */
    omfModuleVerify = (FUNCPTR) relocDllFv[1].func;

    /* if an Init routine exists for this relocator, run it */
    if (relocDllFv[2].func != NULL)
        (*relocDllFv[2].func)();

    return (OK);
}

loadOmFmtCheck()

This routine is the second entry point; it checks the file format.

STATUS loadOmFmtCheck (int moduleFd, BOOL * pFormatIsKnown);

The input parameter is:

int moduleFd
    The file descriptor of the module object.
The output parameter is:

```c
BOOL * pFormatIsKnown
```

A pointer to a boolean that holds the answer.

The return value is:

```c
STATUS
OK or ERROR.
```

The target server calls `loadOmfFmtCheck()` during initialization to determine if a file is in the appropriate format for the loader. Files submitted to this routine should be core files only. (See 3.9.2 Installing a Shared-Library Manager, p.109.) Note that in this example, the first four bytes in the object module are arbitrarily considered a magic number.

Example 3-3  `loadOmfFmtCheck()`

```c
/************************************************************************
* loadOmfFmtCheck - see if the object file is in a known format
* This routine contains the heuristic required to determine if the object
* file belongs to the OMF handled by this OMF reader, and is intended for
* the appropriate target architecture.
* RETURNS: OK or ERROR if file cannot be read.
*/

STATUS loadOmfFmtCheck
{
    int         moduleFd,       /* module object descriptor */
    BOOL *      pFormatIsKnown  /* hold the answer */

    UINT32      magicNumber;    /* OMF magic number */
    BOOL        swap;           /* not used here */

    /* Get the magic number */
    if (read (moduleFd, (char *)&magicNumber, 4) != 4)
        return (ERROR);

    /* Check the module format */
    if (loadOmfModuleIsOk (magicNumber, &swap)*pFormatIsKnown = TRUE;
else *pFormatIsKnown = FALSE;

    return (OK);
}
```
loadOmfFmtManage()

This routine is the third and main entry point; it carries out the load process.

```c
STATUS loadOmfFmtManage (char * pObjMod, int loadFlag, void ** ppText,
                        void ** ppData, void ** ppBss, SYMTAB_ID symTbl,
                        MODULE_ID moduleId, SEG_INFO *pSeg);
```

The input parameters are:

- **char * pObjMod**
  A pointer to the beginning of the object-module copy in memory.

- **int loadFlag**
  The loader option(s). See `loadlib.h` for the definition of all macros associated with the options.

- **SYMTAB_ID symTbl**
  The target symbol table. This is actually a pointer to a SYMTAB structure. See `symlib.h` for a description of this structure.

- **MODULE_ID moduleId**
  A pointer to the object-module descriptor (MODULE structure). See `module.h` for a description of this structure.

The input-output parameters are:

- **void ** ppText**
  Holds the required address where the text segment must be stored, or NULL if no special address is required. When the routine returns, it is set with the final address of the text segment (which may or may not have changed).

- **void ** ppData**
  Holds the required address for the data segment, or NULL. It returns the final address of the data segment.

- **void ** ppBss**
  Holds the required address for the bss segment, or NULL. It returns the final address of the bss segment.

- **SEG_INFO * pSeg**
  A pointer to a structure that holds all information about segments required to process the object file. See `loadLib.h` for a description of the SEG_INFO structure.
The return value is:

**STATUS**

- **OK** or **ERROR**.

**NOTE:** When no address is required (NULL pointers are given), the value **LOAD_NO_ADDRESS** is assigned to the fields **pAddrText**, **pAddrData**, and **pAddrBss** of the **SEG_INFO** structure.

The steps taken by **loadOmfFmtManage()** are summarized below and in Example 3-4. The steps flagged with (*) are described in more detail in later code examples.

1. Check the loader options for consistency.
2. Check whether the host and target byte orders are identical. (*)
3. Read in the module headers. (*)
4. Determine the final size of the three segments: text, data, **bss**. (*)
5. Read in the module symbol table and string table. (*)
6. Determine the compiler signature and publish the builder. (*)
7. Allocate memory for the segments, if required.
8. Read in the various module sections. (*)
9. Check that the core-file checksum and the target-text-section checksum match.
10. Process the module symbol table. (*)
11. Relocate the text and data segments, if required. (*)
12. Download text and data segments, if required. (*)
13. Apply virtual memory protection and cache update, if required.

**NOTE:** If the loader calls **loadOutputToFile()**, it does not download the module to the target memory. It returns with a **loadOutputToFile()** status code instead. This facility allows automated testing without live hardware.

**NOTE:** Even for a fully linked file, the **SEG_INFO** structure must be filled in because it is used for module management, core file verification, and cache and virtual memory management. If a fully linked file holds several text, data, and **bss** sections, for instance, one of each type should be chosen as representative and the information loaded into the **SEG_INFO** structure.
Example 3-4  \textit{loadOmfFmtManage()}

```c
 STATUS loadOmfFmtManage(
    char * pObjMod, /* pointer to the beginning of the module */
    int loadFlag, /* control of loader behavior */
    void ** ppText, /* load text segment at addr pointed to by */
    void ** ppData, /* load data segment at addr pointed to by */
    void ** ppBss, /* load bss segment at addr pointed to by */
    SYMTAB_ID symTbl, /* symbol table to use */
    MODULE_ID moduleId, /* module id */
    SEG_INFO * pSeg /* info about loaded segments */
)
{
    [local variable declarations]

    /* Check the load flag combination */
    if (LoadFlagCheck (loadFlag, ppText, ppData, ppBss) != OK)
    {
        wpwrLogErr ("Illegal combination of flags\n");
        errno = WTX_ERR_LOADER_ILLEGAL_FLAGS_COMBINATION;
        return (ERROR);
    }

    /* First, we check if the module is really Omf, and if it is */
    * intended for the appropriate target architecture.
```
* We also find out if the host and target byte orders are identical. If not, we have to swap the header contents, the symbol info, and the relocation commands. */

```c
if (! loadOmfModuleIsOk (...) )
{
    wprwLogErr ("Not an omf module for the %s architecture.\n", tgtCpuFamilyNameGet ());
    return (ERROR);
}
/* Read object module header */
if (loadOmfMdlHdrRd (...) != OK)
    return (ERROR);
/* Allocate memory for the various tables */
[in general array of symbols and array of symbol address]
/* Read in section headers */
if (loadOmfScnHdrRd (...) != OK)
    goto error;
/* Replace a null address by a dedicated flag (LOAD_NO_ADDRESS) */
   pSeg->pAddrText = (ppText == NULL) ? LOAD_NO_ADDRESS : *ppText;
   pSeg->pAddrData = (pData == NULL) ? LOAD_NO_ADDRESS : *pData;
   pSeg->pAddrBss  = (pBss  == NULL) ? LOAD_NO_ADDRESS : *pBss;
/* Determine segment sizes */
loadOmfSegSizeGet (...);
/* Read the module symbol table */
if (loadOmfSymTabRd (...) != OK)
    goto error;
/* Search information about compiler */
if (loadFlag & LOAD_CORE_FILE)
    loadOmfToolSignatureSearch (...);
/*
   * Allocate the segments according to user requirements.
   * This allocates memory for the image on the host.
   */
if (! (loadFlag & LOAD_FULLY_LINKED) &&
   (LoadSegmentAllocate (pSeg) != OK))
    goto error;
/* We are now about to store the segment contents in the cache */
if (loadOmf/SegStore (...) != OK)
goto error;

/*
 * Build or update the target server symbol table with symbols found
 * in the module symbol tables.
 */
if (loadOmf/SymTabProcess (...) != OK)
goto error;

/* Relocate text and data segments (if not already linked) */
if (((! (loadFlag & LOAD_FULLY_LINKED)) &&
( loadOmf/SegReloc (...) != OK))
goto error;

/* clean up dynamically allocated temporary buffers */

/* Return load addresses, where called for */
if (ppText != NULL)
  *ppText = pSeg->pAddrText;
if (pData != NULL)
  *ppData = pSeg->pAddrData;
if (pBss != NULL)
  *ppBss = pSeg->pAddrBss;

/* Write segments in a file if required (testing session) */
if (loadFlag & LOAD_FILE_OUTPUT)
  return (LoadOutPutToFile (moduleId->name, pSeg));

/*
 * If file is relocatable, everything (text and data) is now flushed
 * to the target.
 * Note that in the case of a core file, we don't want to transfer
 * anything to the target memory (since the core file text and data
 * are already in the target).
 */
if (((! (loadFlag & LOAD_FULLY_LINKED)) &&
( loadOmf/CacheFlush (pSeg, textIsCached, dataIsCached) != OK))
goto error;

/* If virtual memory management is on, apply write protection */
if (((! (loadFlag & LOAD_FULLY_LINKED)) &&
( pSeg->flagsText & SEG_WRITE_PROTECTION)) &&
( TgtMemProtect (pSeg->pAddrText, pSeg->sizeProtectedText,
TRUE) != OK))
goto error;

return (OK);

/*
 * error:
 * free target memory cache nodes, clean up dynamically allocated
 * temporary buffers and return ERROR
 */
error:

wpwrLogErr ("Unrecoverable trouble while loading module.\n");

if (textIsCached)
    TgtMemCacheSet (pSeg->pAddrText, pSeg->sizeText, MEM_NONE, FALSE);

if (dataIsCached)
    TgtMemCacheSet (pSeg->pAddrData, pSeg->sizeData, MEM_NONE, FALSE);

[free memory previously allocated]

return (ERROR);
}

3.9.6 Byte Order

The OMF manager manages the byte ordering in the object module. In a cross-development environment, the host byte order, which is the byte order of the target server, may be different than the target byte order. The cross-compiler usually produces object modules in target byte order; thus the headers, symbol table, and relocation entries may be the wrong byte order for the loader. Even the pieces that require relocation within the text and data segments may be affected since computation is done on the host for the target.

The OMF manager determines the byte ordering in the object module by reading a well known datum, also known as the magic number. If the OMF manager does not recognize this datum, it swaps the datum and tries again. If the OMF manager recognizes it after swapping, it knows that the object module is in the opposite byte order and requires that all information be swapped. However, if the datum is still not recognized after a swap, then the object module is probably not in the expected format.

Some cross-compilers produce structural information (headers, symbol table, relocation entries) in host byte order, but sections in target byte order. In this case,
the OMF manager cannot find a well known datum within the sections. The programmer must address this case in the relocation routines.

Several macros in **host.h** deal with byte ordering:

**SWAB_16**
Swap 2 bytes: AB becomes BA.

**SWAB_32**
Swap 4 bytes: ABCD becomes DCBA.

**SWAB_16_IF**
Apply **SWAB_16** operation if a boolean is true.

**SWAB_32_IF**
Apply **SWAB_32** operation if a boolean is true.

Note that the length of the string table, when written down in the module, may need to be swapped as well.

Example 3-5  
**loadOmfModuleIsOk()**

```c
LOCAL BOOL loadOmfModuleIsOk(  
    UINT16 magicNumber,    /* OMF magic number */  
    BOOL * pSwapIsRequired /* TRUE if header fields have to be swapped */  
)  
{
    BOOL moduleIsForTarget = FALSE;  /* TRUE if intended for target*/
    BOOL byteOrderDiffer = FALSE;    /* TRUE if byte same order */

    /* check if the module is in the correct OMF */
    /* Your code here */

    /* check if the module if for the current CPU */
    return (omfModuleVerify ((machType, pSwapIsRequired)));
}
```
3.9.7 OMF Header Processing

Object-module headers hold vital information about the whole module or pieces of the module. The numbers and characteristics of the headers depend on the OMF. For instance, a.out has one header for the whole module while COFF has a general header, an optional header, plus one header per section. The header information, which includes the section sizes, the number of symbols, and the number of relocation entries, is used for all module processing.

It is good programming practice to store the OMF description (the C objects that represent the OMF entities such as headers, symbol entries, relocation entries, and specific values) in a separate header file (filename.h). Headers then fit naturally into C structures. See a_out.h for example.

Reading in the object-module headers involves reading values from the object-module image in memory into structure fields. Complicating issues include byte ordering, type abstraction, and data element size in both the header and the structure. For all these reasons, the object-module headers have to be read field by field, as shown in Example 3-6.

In the example, a pointer of type void * is used to access all fields of the header in memory. Because no size is associated with the type void, the pointer must be advanced with a cast operation. The type size used for this cast is related to the size of the header field that must be read.

Example 3-6 loadOmfMdlHdrRd()

```c
LOCAL STATUS loadOmfMdlHdrRd
{
    char * pObjMod,        /* pointer to beginning of object file */
    OMF_HDR * pHdr,           /* pointer to header structure to fill */
    BOOL        swapIsRequired  /* byte order must be swapped */
}
{
    void * pHeaderField = (void *) pObjMod; /* ptr to each field */

    /*
     * Fields are read one by one since we must avoid compiler padding.
     * Type abstractions such INT32, UINT32 are used since we don't want
     * to be dependent on the host sizes for short, long, and so on.
     */
```
3.9.8 Determining the Size of the Segments

The OMF reader must determine the size of the three segments, which may consist of several sections of the same or equivalent type. In the simplest situation, when there is only one section per segment, the segment sizes can be determined immediately. Otherwise, the sizes of the gathered sections are added to get the segment sizes. Example 3-7 calls `loadOmfSegSizeGet()` to determine segment size.

Example 3-7  * loadOmfSegSizeGet()

    /************************************************************************
    * loadOmfSegSizeGet - determine segment sizes
    *
    * This function fills in the size fields in the SEG_INFO structure.
    *
    * RETURNS: nothing
    */

    LOCAL void loadOmfSegSizeGet
    {
    int         sectionNumber, /* number of sections */
    SCNHDR *    pScnHdrArray, /* pointer to array of section headers */
    SEG_INFO *  pSeg           /* section addresses and sizes */
    }
    {
    int         sectionIndex; /* loop counter */
    int         nbytes;       /* additional bytes required for alignment */
    int         dataAlign = -1; /* Alignment for the first data section */
    }
int bssAlign = -1; /* Alignment for the first bss section */
[other local variables]
pSeg->sizeText = 0;
pSeg->sizeData = 0;
pSeg->sizeBss = 0;

/* loop thru all sections */
for (sectionIndex = 0; sectionIndex < sectionNumber; sectionIndex++){
    [Get section's type]
    /* Following the three-segment model, all sections of the same type
     * are loaded following each other within the area allocated for the
     * segment. But sections must be loaded at addresses that fit with
     * the alignment requested or, by default, with the target
     * architecture's alignment requirement. So the segment size is the
     * total size of the sections integrated in this segment plus the
     * number of bytes that create the required offset to align the next
     * sections:
     *           +-------------+ <- segment base address (aligned thanks to the
     *           |:::::::::::::|    memory manager). This is also the load
     *           |::::Text:::::|    address of the first section.
     *           |:Section:1:::|
     *           |:::::::::::::|
     *           |:::::::::::::|
     *           +-------------+ <- end of first section.
     *           | ///////////// | <- offset needed to reach next aligned addr.
     *           +--------------+ <- aligned load address of next section within
     *           |:::::::::::::|    the segment.
     *           |:::::::::::::|
     *           |:::::::::::::|
     *           |::::Text:::::|
     *           |:Section:2:::|
     *           |:::::::::::::|
     *           |:::::::::::::|
     *           |:::::::::::::|
     *           |:::::::::::::|
     *           +-------------+
     */
    if ([section is of type text] || [section is of type literal])
}
{ /*
* The contents of literal sections are considered to be
* "text" by the loader (see loadOmfScnRd()). So, the size of
* the literal sections is added to the size of the text
* sections.
*/
    nbytes = LoadAlignGet (alignment, (void *)pSeg->sizeText);
    pSeg->sizeText += (section's size) + nbytes;
}

else if ((section is of type data))
{
    /* Record alignment for data sections */
    if (dataAlign < 0)
        dataAlign = [alignment];
    nbytes = LoadAlignGet (alignment, (void *)pSeg->sizeData);
    pSeg->sizeData += (section's size) + nbytes;
}
else if ((section is of type bss))
{
    /* Record alignment for bss sections */
    if (bssAlign < 0)
        bssAlign = [alignment];
    nbytes = LoadAlignGet (alignment, (void *) pSeg->sizeBss);
    pSeg->sizeBss += (section's size) + nbytes;
}
else
    wpwLogWarn ("Ignored section %d\n", sectionIndex);

/*
* If only one memory area is to be allocated for the three segments,
* take care of the alignment between the three segments. The text
* segment is always aligned thanks to tgtMemMalloc().
*/

if (pSeg->pAddrText == LOAD_NO_ADDRESS &&
    pSeg->pAddrData == LOAD_NO_ADDRESS &&
    pSeg->pAddrBss  == LOAD_NO_ADDRESS)
{
    if (pSeg->sizeData > 0)
        pSeg->sizeText +=
            LoadAlignGet (dataAlign, (void *) pSeg->sizeText);
    if (pSeg->sizeBss > 0)
        pSeg->sizeData +=
            LoadAlignGet (bssAlign,
                          (void *)(pSeg->sizeText + pSeg->sizeData));
}
3.9.9 Reading the Object-Module Symbol Table

The object-module symbol table generally consists of one entry per symbol, with each entry holding several fields. Typically a symbol entry contains the symbol name (or a reference to the symbol name when this is stored in a string table), the symbol type, its value, and so on. This information is used when adding the symbol to the target server symbol table and when relocating the segments.

Reading in the file symbol table involves looping through the list and storing the fields for all entries in an array of structures. All the pitfalls discussed under reading headers apply here as well. Moreover, since we repeatedly read the same pattern of data, it is possible that fields are read across byte boundaries; this could lead to an unaligned access error. The macros UNPACK_16 and UNPACK_32 can be used to prevent this problem.

When the number of symbols is known or can be computed, it is easy to loop through the module symbol table. In Example 3-8, the position of the string table in the object module is deduced from the position of the symbol table. This is OMF dependent, not a general case.

Example 3-8  
\texttt{loadOmfSymTabRd()}  

```
LOCAL void * loadOmfSymTabRd(  
    void *      pSymEntry,      /* ptr to symbol info in object module */  
    SYMENT *    pSymsArray,     /* array of symbol entries */  
    UINT        nbSymbols,      /* number of symbols in module */  
    BOOL        swapIsRequired  /* if TRUE, byte order must be swapped */  
)
{
    int         symIndex;                       /* loop counter */  
    SYMENT *    pSymbol;                        /* ptr to symbol entry */  
    void *      pSymEntryField = pSymEntry;     /* ptr on each field */

    /* loop thru all symbols */

    for (symIndex = 0; symIndex < nbSymbols; symIndex++)
    {
        pSymbol = pSymsArray + symIndex;

        /* read in next entry */
    }
```
/*
 * Fields are read one by one since we are concerned about compiler
 * padding. Type abstractions such INT32, UINT32 _must_ be used
 * since we do not want to be dependent on the host sizes for
 * short, long, and so on. All 32-bit fields are accessed
 * thru the UNPACK_32 macro since they may be unaligned.
 */

pSymbol->firstField = UNPACK_32 (pSymEntryField);
pSymEntryField = (INT32 *)pSymEntryField + 1;
pSymbol->secondField = UNPACK_32 (pSymEntryField);
pSymEntryField = (INT32 *)pSymEntryField + 1;
pSymbol->thirdField = UNPACK_32 (pSymEntryField);
pSymEntryField = (INT32 *)pSymEntryField + 1;
pSymbol->fourthField = *((INT16 *)pSymEntryField);
pSymEntryField = (INT16 *)pSymEntryField + 1;

/* Take care of byte order between target and host */
SWAB_32_IF (swapIsRequired, pSymbol->firstField);
SWAB_32_IF (swapIsRequired, pSymbol->secondField);
SWAB_32_IF (swapIsRequired, pSymbol->thirdField);
SWAB_16_IF (swapIsRequired, pSymbol->fourthField);
}

/*
 * Return the address of the first byte immediately following the
 * symbol table. This address is the address of the string table.
 */
return (pSymEntryField);
}

3.9.10 Determining the Compiler Signature

Example 3-9 shows a routine that looks for a compiler signature in the string table. The signatures are held in an array of strings that must be null-terminated.

Example 3-9 loadOmfToolSignatureSearch()

/*************************************************************************
* loadOmfToolSignatureSearch - get information about the compiler
* 
* This routine gets a compiler signature from the module string table.
* If a known signature is found, it sets the appropriate builder.
* A builder is a string used in the making of the target server.
* Two builders are defined for now: "gnu" and "diab". They apply
* to targets built with the GNU tool chain and the DIAB DATA tool chain.
* (The DIAB DATA tool chain is supported for compatibility only.)
*************************************************************************/
* Note that if no signature can be found, the builder is set to
  * "unknown".
  *
  * Signatures are stored in the signaturesTable table. A NULL pointer
  * must be the last element of this table. Each signature is compared
  * with the string evaluated in the string table. The only signatures
  * recognized are "gcc2_compiled." (GNU tool chain) and "vxWorks" (DIAB
  * tool chain).

  * RETURNS: N/A
  */
LOCAL void loadOmfToolSignatureSearch
{
  void *      pStart,                 /* address to start from */
  UINT32      length                  /* max length to check */
}
void *      pEnd;                   /* last address to check */
char *      signaturesTable[] =      /* compiler signatures */
{ "gcc2_compiled.",                       /* GNU gcc */
  NULL                            /* End mark */
};
char **     ppSignature;            /* ptr to current signature */
char *      pString;                /* string being evaluated */
pEnd = (char *)pStart + length;
pString = (char *)pStart;

  /* Loop until the end of the string table if necessary */
while (pString <= (char *)pEnd)
{ 
  ppSignature = signaturesTable;
  /* Check string against compiler signatures */
  while (*ppSignature != NULL)
  { 
    if (strcmp (pString, *ppSignature) == 0)
      { 
        /* If found a signature record the "builder" */
        LoadCoreBuilderSet ("gnu");
        return;
      }
    ppSignature++;
  }

  /* Walk to the next string, if any */
  for (; (pString <= (char *)pEnd), (*pString != 0); pString ++);
  pString ++;
}
3.9.11 Allocating Memory For the Segments if Required

When the loader is ready to allocate memory on the target for the three segments of a relocatable object module, it calls loadSegmentsAllocate(). (For details, see the online reference material under Tornado API Reference>Target Server Internal Routines.) Fully linked modules, on the other hand, need not have memory allocated; they are fully independent of the run-time system-managed heap.

When the loader allocates memory on the target, it attempts to allocate a single block of memory for all three segments. If there is not enough contiguous memory on the target to allocate a single block, it attempts to allocate separate blocks for each of the three segments. If this is also impossible, the load fails.

3.9.12 Reading in the Segments

The groundwork has been laid and the loader is ready to read in the segments. Example 3-10 shows how loadOmfSegStore() processes the various types of files:

- Relocatable file segments are written in the cache but not downloaded to the target memory until after the relocation process is complete.
- Fully linked file text segments are written in the cache and immediately downloaded to target memory. Data and bss segments are not written in the cache but are immediately downloaded to the target memory.
- Core file text segments are written in the cache but never downloaded to target memory. Other core file segments are ignored.

For a relocatable object module, the loader manipulates the segments on the host side in the target server memory cache before downloading the segments to the target. Performing all the memory manipulation on the host side minimizes the impact on the target system. Example 3-11 shows how loadOmfScnRd(), which is called by loadOmfSegStore(), writes the segment contents to the target server cache. This operation coalesces any sections of equivalent type. Once segments of a relocatable module are transferred into the target memory cache, the loader can perform the relocations.
Example 3-10  \textit{loadOmfSegStore()}

```c
/**
 * loadOmfSegStore - store the module segments in target memory
 *
 * This routine stores the module's segments in target memory. It takes care
 * of host cache management and of the module type (relocatable, fully
 * linked, or core file).
 *
 * RETURNS: OK or ERROR if the segment contents cannot be stored in target
 * memory.
 */
LOCAL STATUS loadOmfSegStore
{
    SEG_INFO * pSeg,    /* info about loaded segments */
    int      loadFlag,    /* control of loaders behavior */
    char *   pObjMod,    /* pointer to beginning of module */
    BOOL *   pTextIsCached, /* text segment in host cache */
    BOOL *   pDataIsCached, /* data segment in host cache */
    SCN_ADRS_TBL * pSectionAdrsTbl, /* tbl of section addr when loaded */
    [other parameters]
}
{
    [local variable declarations]

    /* We do not want to transfer the segment contents to the target
    * immediately. We want to keep them in the cache until all processing
    * is complete (for example, all relocations in relocatable modules).
    * We do this by setting the MEM_HOST attribute for text and data
    * segments. If the file is fully linked (as is a core file) with no
    * address specified, we do the same for each section.
    *
    * Read in text, data, and literal sections. In the case of a relocatable
    * module, section contents are coalesced so that we end up with a
    * three-segment model in memory: text, data, and bss. If the file is
    * fully linked, just read all the sections, if not empty, where they
    * should be located.
    */

    if (!(loadFlag & LOAD_FULLY_LINKED)) /* Relocatable file */
    {
        *pTextIsCached = ((pSeg->pAddrText != LOAD_NO_ADDRESS) &&
            (TgtMemCacheSet (pSeg->pAddrText,
                pSeg->sizeText, MEM_HOST, FALSE) == OK));

        *pDataIsCached = ((pSeg->pAddrData != LOAD_NO_ADDRESS) &&
            (TgtMemCacheSet (pSeg->pAddrData,
                pSeg->sizeData, MEM_HOST, FALSE) == OK));

        if (!loadOmfScnRd (...) != OK)
            return (ERROR);
    }
```
else if (loadFlag & LOAD_CORE_FILE) /* core file */
{
    if (LoadCoreFileCheck (pSeg->pAddrText,
                           pObjMod + [offset to text],
                           pSeg->sizeText) != OK)
        wpwrLogWarn ("Core file checksums do not match.\n");

    /* Core files may have several text sections. Loop thru these
    * sections. Note that we ignore data or bss sections (no need to
    * have them in cache).
    */

    for (scnNum = 0; scnNum < [number of sections]; scnNum++)
    {
        /* Only loadable sections are of interest to us, and we do not
        * want to consider sections that have a null size in the file
        * (bss sections), or sections that occupy no room in the target
        * memory.
        */

        if ([section is executable text] &&
            (TgtMemCacheSet ([section virtual address],
                             [section size],
                             MEM_HOST,
                             FALSE) == OK) &&
            (TgtMemWrite ([address of segment contents],
                           [REMPTR] [section virtual address],
                           [section size]) != OK) ||
            (TgtMemCacheSet ([section virtual address],
                             [section size],
                             MEM_TEXT,
                             FALSE) != OK))
            return (ERROR);
    }
}

else /*
   * A fully linked file other than a core file may have several
   * text, data, or bss sections.
   * Loop thru these sections. Note that only text sections are
   * cached; other sections are immediately downloaded.
   */

    for (scnNum = 0; scnNum < [number of sections]; scnNum++)
    {
        if ([section is to be loaded in target memory])
        {
            if ([section is executable text])
            {
                *pTextIsCached =
(TgtMemCacheSet ([section virtual addr],
[section size],
MEM_HOST,
FALSE) == OK);

if (TgtMemWrite ([address of segment contents],
(REMTR) [section virtual address],
[section size]) != OK)
    return (ERROR);

if (*pTextIsCached &&
    (TgtMemCacheSet ([section virtual address],
        [section size], MEM_TEXT,
        TRUE) != OK))
    return (ERROR);

else
    if (TgtMemWrite ([address of section contents],
            (REMPTR) [section virtual address],
            [section size]) != OK)
        return (ERROR);

    return (OK);

Example 3-11  loadOmfScnRd()

/************************************************************************
* loadOmfScnRd - read sections into the target memory
* This routine actually copies the sections contents into the image of
* the target memory on the host. All sections of the same type are
* coalesced, resulting in a three segment model.
* RETURNS: OK or ERROR if a section cannot be read in.
*/

LOCAL STATUS loadOmfScnRd
(
    char * pObjMod,       /* pointer to beginning of object file */
    int sectionNumber,   /* number of sections */
    SCNHDR * pScnHdrArray,/* pointer to array of section headers */
    SCN_ADRS_TBL sectionAddrTbl,/* table of section addresses */
    SEG_INFO * pSeg       /* segments information */
)
{
    int sectionIndex;   /* loop counter */
    SCNHDR * pScnHdr;   /* pointer to a section header */
    SCN_ADRS * pScnAddr;  /* pointer to address of section */
    SCN_ADRS pTgtLoadAddr; /* target address to load data at */
    INT32 scnSize;  /* section size */
    void * pTextCurAddr; /* current addr where text is loaded */
    void * pDataCurAddr; /* current addr where data are loaded */
void * pBssCurAddr; /* current addr where bss is "loaded" */
void * offset; /* offset of section raw contents */
int nbytes; /* addnl bytes required for alignment */

pTextCurAddr = pSeg->pAddrText;
pDataCurAddr = pSeg->pAddrData;
pBssCurAddr = pSeg->pAddrBss;

/* Loop thru all the sections */
for (sectionIndex = 0; sectionIndex < sectionNumber; sectionIndex++)
{
    pScnHdr = pScnHdrArray + sectionIndex;
pScnAddr = sectionAddrTbl + sectionIndex;
pTgtLoadAddr = NULL;
    scnSize = [size of section];

    /* About the section alignment, see explanations and diagram in loadOmf/SegSizeGet(). */
    if (scnSize != 0)
    {
        if ([section of type text] || [section of type literal])
        {
            /*
             * Text sections and literal sections are merged in one text
             * segment. Note that we could have Text-Literal-Text...
             */
            pTgtLoadAddr = pTextCurAddr;
            nbytes = LoadAlignGet ([alignment], pTgtLoadAddr);
            pTgtLoadAddr = (UINT8 *)pTgtLoadAddr + nbytes;
            pTextCurAddr = (UINT8 *)pTgtLoadAddr + scnSize;
        }
        else if ([section of type data])
        {
            /* Data sections */
            pTgtLoadAddr = pDataCurAddr;
            nbytes = LoadAlignGet ([alignment], pTgtLoadAddr);
            pTgtLoadAddr = (UINT8 *)pTgtLoadAddr + nbytes;
            pDataCurAddr = (UINT8 *)pTgtLoadAddr + scnSize;
        }
        else if ([section of type bss])
        {
            /*
             * Bss sections. Such sections must not be downloaded
             * since they do not actually exist in the object module.
             * However, for relocation purposes, we need to know
             * where they are located in target memory.
             */
            pTgtLoadAddr = pBssCurAddr;
        }
    }
}
nbytes = LoadAlignGet ([alignment], pTgtLoadAddr);
pTgtLoadAddr = (UINT8 *)pTgtLoadAddr + nbytes;
pBssCurAddr = (UINT8 *)pTgtLoadAddr + scnSize;
}
else
    /* ignore any other type of sections */
    continue;

    /*
    * Advance to position in file and copy the section into the
    * target memory image on the host (only if the section
    * exists in module).
    */
    if (module holds section contents)
    {
        offset = (void *) (pObjMod + pScnHdr->pScnAddr);
        if (TgtMemWrite (offset, (REMPTR) pTgtLoadAddr,
                        scnSize) != OK)
            return (ERROR);
    }

    /* record the load address of each section */
    *pScnAddr = pTgtLoadAddr;
}
return (OK);

3.9.13 Target Server Symbol-Table Processing

The target server maintains a symbol table based on the loader symbol management options (see 3.5.3 Symbol Management, p.98). Typically every defined symbol is added to the target server symbol table.

A symbol meets the criteria for adding to the symbol table if:

- The symbol will not be discarded (for example, is not a debug symbol).
- The symbol is neither undefined nor common.
- The options allow adding the symbol to the symbol table.

If the symbol satisfies these constraints, take the following steps to add it to the symbol table:

- Get the symbol name (from the symbol entry or from the string table).
- Set the symbol type.
- Compute the address to which the symbol refers.
- Add the symbol to the target server symbol table.

The address computation is OMF dependent and may be complex. It is based on the following equation:

\[
\text{reference address} = \text{section's base address} + \text{symbol value}
\]

The base address of each section can be computed as follows:

\[
\text{base address of segment} + (\text{size of previous section in segment} + \\
\text{alignment computed on size of previous section}) \times n
\]

where \(n\) is the number of previous sections before the section being computed.

The loader returns the base address of the segments. The size of the previous sections must be obtained from the object file section headers. The alignment (the number of bytes in the “hole” between sections) is computed as follows:

\[
\text{requested alignment} - (\text{size of section} \mod \text{requested alignment})
\]

NOTE: Some OMF readers require the reference addresses in order to perform the relocations. These addresses may be stored in the same way as undefined symbols.

Symbol-table management is performed with the symbol utility library. For more information, see the online reference material under Tornado API Reference>Target Server Internal Routines.

When handling an undefined symbol, the OMF manager follows these steps:

1. It tries to find a matching reference in the target symbol table.
2. If no reference is found, it records the symbol as unknown.
3. It records the address referred by the undefined symbol or NULL if there is no address.

It is critical to store the addresses of the undefined symbols because they are required in order to relocate the references to these symbols in the module segments. Usually an array of SYM_ADRS (see loadlib.h) is used for this purpose.

The matching reference is searched by symFindByNameAndType(), which is summarized in the online reference material under Tornado API Reference>Target Server Internal Routines. If the symbol is not found, loadUndefSymAdd() records the symbol in the unknown symbol list, which is returned to the caller when the load is complete. Note that an unknown symbol does not generate an error (see 3.5.3 Symbol Management, p.98).
When handling a common symbol, the OMF uses the symbol management policy specified in the loader options. This is done by calling `loadCommonManage()`. The final address of the common symbol is stored with the other addresses referred to by the undefined symbols so that relocation can be performed.

The string table needs no further processing because it is loaded into memory in the correct format (strings followed by a NULL character). Only a pointer to the beginning of this table is required. The string table stores the symbol names in the target server symbol table. A secondary role may be to hold a symbol name specific to the tool chain used to compile the file. This name is used as a signature to return the compiler type to the target server when processing the core file. Depending on the OMF, there may be no string table, or the signature may be found in the symbol table or in some other location. You must write a heuristic that suits the particular OMF.

Example 3-12 is a sample symbol-table processing routine. The parameters vary depending upon the OMF-manager implementation, but the following parameters are typical:

- the module identifier (`MODULE_ID`)
- the loader option (`loadFlag`)
- the target-symbol-table identifier (`SYMTAB_ID`)
- the symbol entries (usually an array of structures holding the entries)
- the number of symbol entries
- the section addresses (stored in `SEG_INFO` or in a specific array if required)
- the location where undefined symbol addresses are stored
- the module string table in case all symbol names are not in the symbol entries

Example 3-12  

```
void loadOmfSymTabProcess()
{
  /************************************************************************
  * loadOmfSymTabProcess - process an object module symbol table
  *
  * A pointer is passed to a coff symbol table and processes each of the
  * external symbols defined therein. This processing performs two functions:
  *
  * 1) Defined symbols are entered in the target system symbol table as
  *    specified by the "loadFlag" argument:
  *    LOAD_ALL_SYMBOLS = all defined symbols (LOCAL and GLOBAL) added,
  *    LOAD_GLOBAL_SYMBOLS = only external (GLOBAL) symbols added,
  *    LOAD_NO_SYMBOLS = no symbols added;
  *
  * 2) Any symbols of type "undefined external" are looked up in the target
```
server symbol table to determine their values. If found, they are entered in an array. This array is indexed by the symbol number (position in the symbol table). Note that all symbol values, not just undefined externals, are entered in this array. The values are used to perform relocations.

Note that common symbols have type undefined external; the value field of the symbol is non-zero for common symbols, indicating the size of the object.

If an undefined external cannot be found in the target server symbol table, a warning message is printed, the noUndefSym field of the module is set to FALSE, and the name of the symbol is added to the list in the module. Note that this is not considered to be an error since the loader attempts to resolve undefined externals as additional modules are loaded.

* RETURNS: OK or ERROR.
*/

LOCAL STATUS loadOmfSymTabProcess
{
  MODULE_ID moduleId,      /* module id */
  int loadFlag,      /* control of loader behavior */
  SYMENT * pSymsArray,    /* pointer to symbol array */
  SCN_ADRS_TBL sectionAddrTbl,/* array of section addresses */
  SYM_ADRS_TBL symAdrsTbl,    /* array of in symbol absolute values */
  char * pSymStrings,   /* symbol string table */
  SYMTAB_ID symTbl,        /* symbol table to use */
  int symNumber,        /* num of symbols in module symbol table */
  SCNHDR * pScnHdrArray   /* pointer to Omf section header array */
}

[local variables declarations]

/* Loop thru all symbol table entries in object file */

for (symIndex = 0; symIndex < symNumber; symIndex++)
{
  pSymbol = pSymsArray + symIndex;

  /* Get rid of debug stuff */
  if (symbol is a debug symbol)
    continue;

  /* Get symbol's name from string table */
  name = pSymStrings + index to reach the symbol's name;

  if (symbol is not undefined) && (symbol is not common)
  {
    /*
      * Symbol is neither an undefined external nor a common symbol.
      * Determine symbol section and address bias
      * If the object file is already absolutely located (by the
* linker on the host), then the symbol values are already
* correct. There is no need to bias them. Bias is also not
* needed when the symbol is absolute.
*/
if ((loadFlag & LOAD_FULLY_LINKED) || (symbol is absolute))
bias = 0;
else
    bias = (void *)((section’s base address + symbol’s value));
/* Determine the symbol type. */
/* For an absolute symbol, don’t consider the section type */
if (is symbol is absolute)
symType = SYM_ABS;
/* Is it a symbol from a text or literal section? */
else if (symbol is of type text or literal)
symType = SYM_TEXT;
/* Is it a symbol from a data section? */
else if (symbol is of type data)
symType = SYM_DATA;
/* Is it a symbol from a bss section? */
else if (symbol is of type bss)
symType = SYM_BSS;
/* If none of these, we don’t know how to handle this
* type of symbol */
else {
    wpwrLogWarn("Unknown sym type for symbol %s\n", name);
    continue;
}
/* Determine if symbol should be put into symbol table. */
if (((loadFlag & LOAD_LOCAL_SYMBOLS) && [symbol not global]) ||
    ((loadFlag & LOAD_GLOBAL_SYMBOLS) && [symbol is global]))
{
    if ([symbol is global])
        symType |= SYM_GLOBAL;
    else
        symType |= SYM_LOCAL;
    /* Add symbol to symbol table. */

    if (SymAdd (symTbl, name, (char *)([symbol value] +
        (INT32)bias), symType, moduleId->group) !=OK)
    {
    }
wpwrLogErr ("Can't add '%s' to sym table\n", name);
status = ERROR;
}
}

/*
 * Add the symbol address to the externals table.
 * For omf, we add all symbol addresses to the externals
 * table, not only those symbols added to the target server
 * symbol table. This is required by the relocation process.
 */
symAdrsTbl [symIndex] = (SYM_ADRS)(SYM_ADRS)({sym's value}+(INT32)bias);
}
else
{
/*
 * A "common" symbol type is denoted by "ndefined external"
 * with its value set to non-zero.
 */
if ({symbol is common})
{
    /* follow common symbol management policy */
    if (LoadCommonManage {symbol value}, name, symTbl,&adrs,
        loadFlag, moduleId->group) != OK)
        status = ERROR;
    }
else
    /* look up undefined external symbol in symbol table */
    if (SymFindByNameAndType (symTbl,name,(char **)&adrs,
        &symType, SYM_GLOBAL,
        SYM_GLOBAL) != OK)
    {
        /* symbol not found in symbol table */
        adrs = NULL;
        /* Record the symbol name for further request */
        LoadUndefSymAdd (moduleId, name);
    }
    /* add symbol address to externals table */
    symAdrsTbl [symIndex] = adrs;
}
return {status};
3.9.14 Relocating the Object Modules

Relocation is highly dependent on both the OMF and the target architecture. It is therefore difficult to give precise information. The relocation unit resides in a separate shared library which is linked in during the OMF reader initialization (see 3.9.4 The RU Interface, p. 112).

Relocations occur only for relocatable files, which have segments not immediately usable for execution. Relocatable files generally hold references to undefined symbols, whose addresses have not yet been determined. Even references to defined symbols may be relative to the beginning of the section, which is initially assumed to be address zero. The relocation is performed by `omfCpuSegReloc()` (see Relocation Process, p. 143), which in turn calls `omfCpuRelocEntryRd()` to fill in the relocation structure (see 3.9.5 The OMF Interface, p. 114).

Reading in Relocation Entries

Object modules hold information about how to relocate the sections. This information is generally presented as a table of relocation entries. A relocation entry (OMF dependent) is composed of:

- the offset of the symbol reference within the section
- a reference to the symbol entry in the symbol table (called an index)
- the type of relocation process to be done

Note that the offset within a section requires that each section address be recorded when the sections are read in. We cannot use the segment base address here, but need the real address where each section is stored within the segment. A routine such `loadOmfScnRd()`, shown in Example 3-11, should be modified to fill an array with these section addresses.

Reading the relocation entries requires the same care as reading the headers; a swap may be required if byte ordering differs, and unaligned accesses must be avoided while reading the entry fields. Example 3-13 is a routine for reading the relocation entries.

Example 3-13 `omfCpuRelocEntryRd()`

```c
/************************************************************************
* omfCpuRelocEntryRd - read in an OMF relocation entry
* This routine fills a relocation structure with information from the
* object module in memory. It swaps the bytes if this is required.
/************************************************************************
```
LOCAL void *
omfCpuRelocEntryRd
{
  void * pRelocEntry, /* ptr to relocation cmd in object file */
  RELOC * pReloc,   /* ptr to relocation structure to fill */
  BOOL swapIsRequired /* if TRUE, byte order must be swapped */

  void * pRelocField = pRelocEntry; /* points to each "field" */

  pReloc->offset = UNPACK_32 (pRelocField);
  pRelocField = (long *)&pRelocField + 1;

  pReloc->index = UNPACK_32 (pRelocField);
  pRelocField = (long *)&pRelocField + 1;

  pReloc->type = *(unsigned short *)pRelocField);
  pRelocField = (unsigned short *)&pRelocField + 1;

  /* Take care of the byte order between host and target */
  SWAB_32_IF (swapIsRequired, pReloc->offset);
  SWAB_32_IF (swapIsRequired, pReloc->index);
  SWAB_16_IF (swapIsRequired, pReloc->type);

  return (pRelocField);
}

Relocation Process

The relocation process itself has three parts:

1. Computing the address of the code that needs to be modified within the section. It is generally computed as follows:

   code address = section address + offset of reference within section

2. Computing the address of the symbol referred to. It comes from the symbol table processing; addresses for all undefined symbols are recorded, usually in an array.

3. Code replacement. This consists of modifying the code located in Step (1) by integrating the symbol address located in Step (2). This modification depends on the type of relocation process. For some OMF readers, the symbol address must be combined with a value found at the code address; for others, the relocation entry holds all such values.
After the relocation computation is complete, store the resulting value at the code address with `tgtMemWrite()`. If you must write 8-bit or 16-bit values, call `tgtMemWriteByte()` and `tgtMemWriteShort()`. Example 3-14 shows the relocations.

Example 3-14  `omfCpuSegReloc()`

```c
STATUS omfCpuSegReloc

/**************************************************************************
* omfCpuSegReloc - perform relocation for the CPU family
*
* This routine reads the specified relocation command entry and performs
* all the relocations specified therein.
* Absolute symbol addresses are looked up in the 'externals' table.
* RETURNS: OK or ERROR.
*/

void * pNextRelocCmd, /* ptr to current relocation command */
SCN_ADRS * pScnAddr,   /* section address once loaded */
SYM_ADRS_TBL symAdrsTbl, /* array of absolute symbol values */
BOOL swapIsRequired /* if TRUE, byte order must be swapped */

/* Relocation loop */
for (relocNum = 0; relocNum < [# of relocation entries]; relocNum++)
{
    /* read relocation command */
    pNextRelocCmd = omfCpuRelocEntryRd (pNextRelocCmd, &relocCmd, swapIsRequired);

    /* Calculate the actual remote address that needs relocation and
    * perform external or section-relative relocation.
    */
    pAdrs = (void *)((INT32)*pScnAddr + relocCmd.offset);
    switch (relocCmd.type)
    {
      case RELOC_TYPE_1:
        TgtMemRead (pAdrs, &value, 4);   
        SWAB_32_IF (swapIsRequired, value); /* host fmt */
        SWAB_32_IF (swapIsRequired, value); /* tgt fmt */
        TgtMemWriteManyInts (pAdrs, value);
        break;
```
case RELOC_TYPE_2:
    TgtMemRead (pAdrs, &value, 1);
    value = [relocation computation];
    tgtMemWriteByte (pAdrs, value);
    break;

case RELOC_TYPE_3:
    TgtMemRead (pAdrs, &value, 2);
    SWAB_16_IF (swapIsRequired, value);       /* host fmt */
    value = [relocation computation];
    SWAB_16_IF (swapIsRequired, value);       /* tgt fmt */
    tgtMemWriteShort (pAdrs, value);
    break;

default:
    wpwrLogErr("Unrecognized reloc type \%d\n", relocCmd.type);
    errno = UNRECOGNIZED_RELOC_ENTRY;
    status = ERROR;
    break;
}

return (status);

3.9.15 Downloading Relocated Modules to Target Memory

After all relocation manipulations are done, download the text and data segments to target memory using loadOmfCacheFlush() as shown in Example 3-15. This process includes a modification of the block attributes using tgtMemCacheSet() with the push flag set to TRUE. It also cleans up the area used by the bss segment. Remember that non-relocatable files were already downloaded to the target where necessary at the same time that relocatable files were loaded into cache for relocation. Remember also that the core file is not pushed to the target memory. (See 3.9.12 Reading in the Segments, p.131.)

Example 3-15  loadOmfCacheFlush()

/*****loadOmfCacheFlush - flush the target server cache to target's memory*****/
loadOmfCacheFlush()

* This routine synchronizes cache on host with target memory and sets the
  appropriate attributes, depending on the segment types.

* RETURNS: OK, or ERROR if host cache and target memory cannot be
  synchronized.
LOCAL STATUS loadOmjCacheFlush
{
    SEG_INFO * pSeg,  /* info about loaded segments */
    BOOL    textIsCached, /* text segment in host cache */
    BOOL    dataIsCached /* data segment in host cache */
}

/*
 * Everything (text and data) is flushed to the target when the
 * attribute is changed (to MEM_TEXT or MEM_NONE).
 */
if ((pSeg->sizeBss != 0) &&
    (TgtMemSet (pSeg->pAddrBss, pSeg->sizeBss, 0) != OK))
    return (ERROR);

/*
 * Update the load state with the beginning of the download.
 * We give the two segments byte count (Bss is not transferred, but
 * initialized.
 */
asyncLoadUpdate (LOAD_DOWNLOADING, pSeg->sizeText+pSeg->sizeData, 0);

/*
 * Synchronize cache on host and target memory. The text segment is set
 * to MEM_TEXT since only writes needs to go to the target. The data
 * segment is set to MEM_NONE so that reads and writes go to the
 * target (with automatic synchronization between cache and target).
 * Update the target instruction cache with the text segment in memory.
 */
if (textIsCached &&
    (TgtMemCacheSet (pSeg->pAddrText, pSeg->sizeText, MEM_TEXT,
                    TRUE) != OK))
    return (ERROR);

if (TgtMemCacheTextUpdate (pSeg->pAddrText, pSeg->sizeText) != OK)
    return (ERROR);

if (dataIsCached &&
    (TgtMemCacheSet (pSeg->pAddrData, pSeg->sizeData, MEM_NONE,
                    TRUE) != OK))
    return (ERROR);

return (OK);
3.9.16 Managing Memory

The final responsibility of the OMF manager is to apply write protection on the text segment (if the target has an MMU) and to update the target instruction and data caches.

Achieve synchronization between the target memory and the target caches with \texttt{tgtCacheTextUpdate()} . The OMF manager does not have to know if such caches actually exist on the target system. The update request is satisfied as well as possible by the target agent. Note that this routine is called for every text and literal section of a fully linked executable that does not correspond to the three-segment model. However, you need not call this routine for the core file because it is not downloaded to the target.

Virtual memory management is more complex. It is not possible to protect fully linked files since these files may not be page-aligned. For the text segment of relocatable object modules, the loader manages alignment automatically using \texttt{loadSegmentAllocate()}. The text segment is automatically allocated on a page boundary if text protection is implemented on the target and the flag \texttt{SEG_WRITE_PROTECTION} is set in the \texttt{flagsText} field of the \texttt{SEG_INFO} structure. The OMF manager checks that the module is relocatable and that the flag \texttt{SEG_WRITE_PROTECTION} is set before calling \texttt{tgtTextProtect()}.

3.9.17 Notes on Asynchronous Load Operation

The target server can report load progress information to the WTX client which submitted the load operation. In order to do this, the target server exports a routine called \texttt{asyncLoadUpdate()} which the loader should call to update the load progress state. For example, the loader should call \texttt{asyncLoadUpdate()} before downloading the segments to the target (See Example 3-15). For information on how the target server retrieves the status, see 3.4.4 Asynchronous Load Operation, p.94.

The target server regularly updates the downloaded bytes count while a load is in process. If a load is cancelled, the target server returns directly from the download operation, reporting an error. The OMF reader must free all the allocated resources.

3.9.18 Utility Functions and Error Logging

Log errors and warnings with \texttt{wpwrLogErr()} and \texttt{wpwrLogWarn()}. The target server only displays messages if it is running in verbose mode.
The routine `wpwrLogErr()` is used for critical failures; it prevents further module processing. The routine `wpwrLogWarn()` is used for non-critical problems. These routines accept parameters in a format similar to `printf()`. A message begins with a capital letter and does not have a period at the end (for example, “This is an error message”). Examples of these calls are found in the code examples already presented.

For a complete survey of utility routines available from the target server, see the online reference material under Tornado API Reference>Target Server Internal Routines.
4

The WTX Protocol

4.1 Introduction

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

The Tornado 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 Tornado tools running on the development host (for example, a shell or a debugger) to debug and monitor a real-time target system.

4.2 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 4-1). Target servers can also connect to VxSim, a target 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.
4.2.1 Language Support

The WTX protocol is usually accessed by Tcl commands or C functions rather than directly (see Figure 4-2). Introductory material for protocol and language bindings is presented in 4.4 WTX Tcl API, p.167 and in 4.5 WTX C API, p.185. Tornado provides a Tcl interpreter and 4.4 WTX Tcl API, p.167 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.

The WTX Java API provides a set of Java classes permitting access to the WTX protocol services. It is built on top of the WTX C API using the Java Native Interface. Use this interface to create tools that access the target server or the Tornado registry when Java is the implementation language. This API is unsupported. It can be downloaded from the Wind River System WEB server (www.wrs.com).

The online reference material under Tornado API Guide>WTX Protocol, WTX Tcl Library, and WTX C Library gives detailed descriptions of WTX services, error messages, and API parameters.
4.2.2 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 4-3 is a schematic representation of a series of interactions between WindSh (a tool) and the target server, showing how WTX commands fit into the process.

Figure 4-3 WTX Protocol Links Tool to Target

<table>
<thead>
<tr>
<th>HOST SHELL (WindSh)</th>
<th>TARGET SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach to target server</td>
<td>WTX_TOOL.Attach</td>
</tr>
<tr>
<td>get target and target server info</td>
<td>WTX_TS_INFO_GET</td>
</tr>
<tr>
<td>create a task</td>
<td>WTX_CONTEXT_CREATE</td>
</tr>
<tr>
<td>poll for target events</td>
<td>WTX_EVENT_GET</td>
</tr>
</tbody>
</table>
4.2.3 WTX Message Format

All WTX messages are defined in `installDir/host/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/host/include/wtxerr.h`.

```c
typedef struct wtx_core /* WTX message core */
{
    UINT32 objId; /* identifier */
    WTX_ERROR_T errCode; /* service error code */
    UINT32 protVersion; /* WTX protocol version */
} WTX_CORE;
```

4.2.4 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 Jan  8 17:59:37 1999
Target Server Name        : t80-202@couesnon
Target Name               : t80-202
Target Server Options     : tgtsvr.ex -V -Wd /tmp/WTXtrace t80-202
Host                      : SunOS couesnon 5.5.1 Generic_103640-12 sun4u

0       8    WTX_TOOL_ATTACH                        Fri Jan  8 17:59:48 1999
In
    WTX_MSG_TOOL_DESC
    WTX_CORE
        objId             0x0
        errCode           0
        protVersion       0
    WTX_TOOL_DESC
       id                0x0
       toolName          wtxtcl
       toolArgv          Unknown
       toolVersion       Unknown
       userName          chrisc@couesnon
       pReserved         Unknown
       next              Unknown
    Out status          Ok
    WTX_MSG_TOOL_DESC
    WTX_CORE
```
4.3 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.

4.3.1 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.
4.3.2 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 CrossWind, the Tornado debugger (see the Tornado User's Guide).

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

4.3.3 Attaching to a Target Server

Binding to a target server is mediated by the Tornado 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.

4.3.4 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 \texttt{WTX\_GOPHER\_EVAL}, which is discussed in 4.3.11 Gopher Support, p.160.

### 4.3.5 Target Memory Disassembly

WTX-based tools can request the disassembly of a target memory area with \texttt{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.

### 4.3.6 Object-Module Management

The object-module loader allows the target server to load and unload relocatable or fully linked object modules to the target. Modules may be loaded from any tool. The object-module formats (OMFs) supported depend on what OMF readers are available to the target server. See \texttt{installDir/host/resource/target/architecturedb} for a list of the OMF readers available on your host.

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. The loader also supports text section protection if the optional product VxVMI is included on the target.

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. \texttt{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.
4.3.7 Symbol Management

The target server provides symbol-table management for the target. 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.

4.3.8 Context Management

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

4.3.9 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 4-4, 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.

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 Programmer’s Guide: Local File Systems.
4.3.10 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:

```plaintext
eventName [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). Table 4-1 lists all event parameters.
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.

<table>
<thead>
<tr>
<th>Event</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTX_START</td>
<td>contextType(h) contextId(h)</td>
</tr>
<tr>
<td>CTX_EXIT</td>
<td>contextType(h) contextId(h) returnVal(h) errnoVal(h)</td>
</tr>
<tr>
<td>TEXT_ACCESS</td>
<td>contextId(h) contextType(h) pc(h) fp(h) sp(h)</td>
</tr>
<tr>
<td>DATA_ACCESS</td>
<td>contextId(h) contextType(h) pc(h) fp(h) sp(h) addr(h)</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>contextType(h) contextId(h) excVector(h) esfAddr(h)</td>
</tr>
<tr>
<td>VIO_WRITE</td>
<td>channelId(h) addlData addlDataLen</td>
</tr>
<tr>
<td>EVTPT_ADDED</td>
<td>evptNum(h)</td>
</tr>
<tr>
<td>EVTPT_DELETED</td>
<td>evptNum(h)</td>
</tr>
<tr>
<td>CALL_RETURN</td>
<td>callTaskId(h) returnVal(h) errnoVal(h)</td>
</tr>
<tr>
<td>TGT_RESET</td>
<td></td>
</tr>
<tr>
<td>TS_KILLED</td>
<td></td>
</tr>
<tr>
<td>OBJ_LOADED</td>
<td>objModId(h) objModName(s)</td>
</tr>
<tr>
<td>OBJ_UNLOADED</td>
<td>objModId(h) objModName(s) objModGroup(h)</td>
</tr>
<tr>
<td>SYM_ADDED</td>
<td>symbol(s) symVal(h)</td>
</tr>
<tr>
<td>SYM_REMOVED</td>
<td>symbol(s) symType(s)</td>
</tr>
<tr>
<td>TOOL_ATTACH</td>
<td>toolName(s)</td>
</tr>
<tr>
<td>TOOL_DETACH</td>
<td>toolName(s)</td>
</tr>
<tr>
<td>TOOL_MSG</td>
<td>destination(s) sender(s) message(s) anything(s)</td>
</tr>
<tr>
<td>USER</td>
<td>message(s)</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td></td>
</tr>
</tbody>
</table>

* The values can be either hexadecimal values (h) or strings (s).

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/host/include/wxtypes.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:

```c
#include "wtx.h"

/* handle events */
LOCAL void eventHandler
  (
    WTX_EVENT_DESC * event
  )
```
/* print the received event */

if (event->event != NULL)
    printf("Received the event : %s\n", event->event);
}

/* register to receive events */

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

4.3.11 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. 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 Tornado 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 4-2 shows the elements of the Gopher language and how they affect the pointer and the tape.

<table>
<thead>
<tr>
<th>Item</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>Sets the pointer to the value integer.</td>
</tr>
<tr>
<td>[+-] integer</td>
<td>Increments or decrements the pointer by integer.</td>
</tr>
</tbody>
</table>
This section describes a Gopher script that traverses and collects information from the following two sample data structures:

```c
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;
```

```c
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 (l), 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 subroutine {...} 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} > * }
```

NOTE: Wind River Systems has developed a “Gopher generator” tool which generates gopher scripts from a C structure. This tool is not supported, but is available free of charge. It can be downloaded from the Wind River System WEB server (www.wrs.com).
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/share/src/agents/wdb/wdb.h` as follows:

- GOPHER_UINT32 0
- GOPHER_STRING 1
- GOPHER_UINT16 2
- GOPHER_UINT8 3
- GOPHER_FLOAT32 4
- GOPHER_FLOAT64 5
- GOPHER_FLOAT80 6

The tape is formatted as shown in Figure 4-5.

![Gopher Result Tape Diagram](image)

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
the appropriate WTX command. Both Tcl and C language APIs provide commands for this purpose.

- **Tcl**

  Our 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`:

  ```tcl
  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 our example, the Tcl API returns the following list:

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

  For further information, see the online reference material under Tornado API Guide>WTX Tcl Library.

- **C**

  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 4-1 shows how to submit the Gopher script and format the result tape. For further information, see the online reference material under Tornado API Guide>WTX Tcl Library.

**Example 4-1 Evaluating a Gopher Script Using the C API**

```c
/* find the starting address */
if ((pSymbol = wtxSymFind (hWtx, "nodeList", NULL, 0, 0, 0)) == 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 * ( 0 < +8 *$ > < +4 * (@@ 0) > * )", startAddress);

if ((pResult = wtxGopherEval (hWtx, 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 (type == GOPHER_FLOAT80 && targetByteOrder == BIG_ENDIAN)
    bufIx += 2;
break;
}

4.4 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 4.5 WTX C API, p.185) may benefit from working through these Tcl examples.

4.4.1 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. Tornado 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 8.0, and on UNIX hosts includes the Extended Tcl 8.0 package.

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 A. Tcl Coding Conventions; in other words, underscores are removed and all words but the first are capitalized.

Each WTX Tcl command has an online reference entry in HTML format under Tornado API Guide>WTX Tcl Library. For information on the online reference material, see the Tornado User's Guide: Documentation Guide and Tornado Getting Started. Most WTX Tcl API commands also return a brief syntax message if they are invoked without arguments.

4.4.2 Starting a wtxtcl Session

One of the interactive tools in Tornado is called wtxtcl, a standalone Tcl interpreter for the WTX extensions. All Tornado tools also include their own Tcl interpreter, because Tcl is the language of much of the Tornado implementation. All the interpreters have access to the WTX extensions. However, wtxtcl is an ideal environment for experimenting with and testing the WTX interface because it contains only the WTX extensions.

When you call wtxtcl, Tornado 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.

4.4.3 Attaching to a Target Server

Before the majority of wtxtcl commands can be used, you must attach to an active target server. The Tornado 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 Tornado registry.

```
wtxtcl> wtxInfoQ
{vxsim7@aven tgtsvr rpc/aven/147.11.80.6/570425345/1/tcp/42781}
{mv147@aven tgtsvr rpc/aven/147.11.80.6/570425347/1/tcp/44830}
{jim@dnestr tgtsvr rpc/dnestr/147.11.80.4/570425345/1/tcp/52411}
{wtxregd@aven registry rpc/aven/147.11.80.6/570425344/1/tcp/52157}
```

The routine returns a series of three-element lists, each of the form {name type key}. The Tornado registry includes information about itself (note the fourth line of the output). To limit the returned information, you can use regular expressions on the wtxInfoQ command to qualify each of the three fields. For example, to see only target servers running on host aven, type:
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
```

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 Tornado launcher (UNIX) or the Manage Target Servers dialog box (Windows) if your target server is selected.

The value returned by this command, `t80-202@aven`, 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.

### 4.4.4 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 RPC) {255 0} {5 1 0 8192 1234} {1 5.4} {Motorola MVME162} /aven:/installDir/target/config/mv162/vxWorks {0 4194304} {0x96c60 223546}
chrisc [Fri Jan 8 17:59:38 1999] [Fri Jan 8 17:59:38 1999] unreserved 0
myTool 2.0
{0x1a5278 wtxtcl {} {} myself@aven}
```

For a detailed explanation of the fields in this structure, see the online reference material under `Tornado API Guide>WTX Protocol`. 
4.4.5 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 0x2000 256
mblk0
```

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 0x2000 256]
mblk1
```

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

```
wtxtcl> memBlockGet $mb
0x03 0x00 0x00 0x39 0x82 0x10 0x60...
```

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 online reference material under Tornado API Guide>WTX Protocol and WTX Tcl Library.
4.4.6 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 0x13e7bc 0x5 0 1 vxWorks}
wtxtcl> wtxMemDisassemble -opcodes 0x13e7bc
{printf} 4 {} {9421ff80} {stwu        r1, 0xff80(r1)}
{} 4 {} {7c0802a6} {mfspr       r0, LR}
{} 4 {} {90010084} {stw         r0, 0x84(r1)}
{} 4 {} {90610008} {stw         r3, 0x8(r1)}
{} 4 {} {9081000c} {stw         r4, 0xc(r1)}
{} 4 {} {90a1000e} {stw         r5, 0x10(r1)}
{} 4 {} {90c10010} {stw         r6, 0x14(r1)}
{} 4 {} {91010014} {stw         r7, 0x18(r1)}
{} 4 {} {91210018} {stw         r8, 0x1c(r1)}
{} 4 {} {9121001c} {stw         r9, 0x20(r1)}
```

4.4.7 Working with the Symbol Table

The target server symbol table can be queried and modified from wtxcl. 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:

```
wtxcl> wtxSymFind -name taskDelay
_taskDelay 0x22ed4 0x5 0 0 ""
wtxcl> wtxSymFind -value 0x226b4
_taskSuspend 0x226b4 0x5 0 0 ""
```

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} \n{return [lindex [wtxSymFind -name $name] 1]}
wtxtcl> symbol taskDelay
0x22ed4
```

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

```
wtxcl> wtxSymListGet -name "^taskD.*" [taskDeleteHookAdd 0x14b624 0x5 0 1 vxWorks]
[taskDeleteTable 0x1a2368 0x9 0 1 vxWorks]
[taskDeleteHookDelete 0x14b6b4 0x5 0 1 vxWorks]
[taskDeleteForce 0x15ff48 0x5 0 1 vxWorks]
```
You can also use `wtxSymListGet` to obtain a list of the symbols found near a given value:

```
wtxtcl> wtxSymListGet -value 0x68000
(_ip_output 0x67ae0 0x5 0 0 vxWorks)
(_ip_insertoptions 0x681ac 0x5 0 0 vxWorks)
(_ip_optcopy 0x68320 0x5 0 0 vxWorks)
...
```

### 4.4.8 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 Tornado 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 /work/dir/umod.o
0x624ef8 0x36c540 0x36d5a8 0x36d5d8
_missingVar
_missingFunc
```

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 [wtxObjModuleInfo $mod]
}
```

```
0x71db0 vxWorks a.out 0 0x64 {0 0x2000 0x957e8} {0 0x977e8 0x10f50} {0 0xa8738 0x87b0} 0x1324b0 mod.o a.out 0x1 0x4 {0 0x39e9e8 0x1048} {0 0x39fa30 0x30} {0 0x39fa60 0xa8} 0x132760 umod.o a.out 0x2 0x4 {0 0x39d530 0x1068} {0 0x39e598 0x30} {0 0x39e5c8 0xa8} 0x39cfa60 0xa8
```
4.4.9 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 \texttt{wtxcl} 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 \texttt{wtxcl}.

The first step is to spawn a task named \texttt{uMyTask} at priority 100 with VxWorks options \texttt{0x3 (VX\_UNBREAKABLE | VX\_SUPERVISOR\_MODE)}. The entry point is the \texttt{taskDelay()} routine, whose address comes from \texttt{symbol}, which you created in 4.4.7 Working with the Symbol Table, p.171. The argument \texttt{10000} causes the task to delay for 10,000 ticks before exiting. (For more information on command parameters, see the online reference material under Tornado API Guide>WTX Tcl Library.)

\begin{verbatim}
\texttt{wtxcl} > \texttt{wtxContextCreate 1 uMyTask 100 0x3 0 5000 \[symbol taskDelay\] 0 0 10000 0x3b6870}
\end{verbatim}

The output, \texttt{0x3b6780}, is the ID of the new task. If you are running a Tornado shell, you can use \texttt{if()} 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:

\begin{verbatim}
\texttt{wtxcl} > \texttt{wtxContextResume CONTEXT\_TASK 0x3b6870 0}
\end{verbatim}

Now the task runs, and vanishes after the task delay has expired. To find out what other commands operate on contexts, see the online reference material under Tornado API Guide>WTX Tcl Library. To see a list of these commands, type:

\begin{verbatim}
\texttt{wtxcl} > \texttt{info commands wtxContext*}
\texttt{wtxContextResume wtxContextKill wtxContextStep wtxContextSuspend}
\texttt{wtxContextCreate wtxContextCont}
\end{verbatim}

4.4.10 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 4.3.10 Event Management, p.157. 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. Table 4-1 contains a detailed list of the events and their parameters. 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:

```tcl
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 4.4.9 Working with Tasks, p.173, without setting the VX_UNBREAKABLE bit in the task options:

```tcl
wtxtcl> set ctxId [wtxContextCreate CONTEXT_TASK tick 100 0x1
  0 2000 [symbol tickGet] 0 0]
0x4aeec90
```

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 online reference material under Tornado API Guide->WTX Protocol). 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:

```tcl
wtxtcl> wtxEnumList
ACTION_TYPE CONTEXT_TYPE EVENT_TYPE REG_SET_TYPE LOAD_FLAG VIO_CTL_REQUEST
AGENT_MODE TASK_OPTION OBJ_KILL_REQUEST OPEN_FLAG CONTEXT_STATUS
wtxtcl> wtxEnumInfo EVENT_TYPE
WTX_EVENT_NONE 0x0
WTX_EVENT_CTX_START 0x1
WTX_EVENT_CTX_EXIT 0x2
WTX_EVENT_TEXT_ACCESS 0x3
WTX_EVENT_DATA_ACCESS 0x4
```
WTX_EVENT_EXCEPTION 0x5
WTX_EVENT_VIO_WRITE 0x6
WTX_EVENT_HW_BP 0x7
WTX_EVENT_CALL_RETURN 0x8

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:

```wxtcl
wtxEnumInfo -abbrev EVENT_TYPE
```

WTX_EVENT_NONE WTX_EVENT_NONE NONE none 0x0
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_DATA_ACCESS WTX_EVENT_DATA_ACCESS DATA_ACCESS dacc 0x4
WTX_EVENT_EXCEPTION WTX_EVENT_EXCEPTION EXCEPTION exc 0x5
WTX_EVENT_VIO_WRITE WTX_EVENT_VIO_WRITE VIO_WRITE viow 0x6
WTX_EVENT_HW_BP WTX_EVENT_HW_BP HW_BP hwbp 0x7
WTX_EVENT_CALL_RETURN WTX_EVENT_CALL_RETURN CALL_RETURN cret 0x8

⚠️ **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:

```wxtcl
wtxEventpointAdd tacc [symbol tickGet] task $ctxId \ notify|stop 0 0 0 0x1
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:

```wxtcl
wtxRegisterForEvent .*
proc listEvents {} {while {[set event [wtxEventGet]] != ""]{
{puts $event}}
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
CALL_RETURN 0x4ae2720 0x5 0xd0003
TOOL_ATTACH browser 0x450f90
... TEXT_ACCESS 0x4aeecc90 0x3 0x426d5c 0x4aeecc4c 0x4aeecc20
```

When you see the `TEXT_ACCESS` line of output, the breakpoint event has arrived. The following message appears in WindSh:

```
-> Break at 0x00426d5c: _tickGet                  Task: 0x4aeecc90 (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 0x4aeecc90 0x3ce474 0xd0003
```

### 4.4.11 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 Tornado 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 Tornado). For more information on configuring Tornado and the target, see the *Tornado User’s Guide*. 
Be sure to launch the target server without any target redirection option (for more details, see *Tornado User’s Guide: Target Server*), otherwise the example may not work properly.

First, open a virtual I/O channel and redirect the target’s standard input, output, and error streams to this channel. To start WindSh on the target server, click the WindSh button in the Tornado window or start the shell from the Windows command prompt or a UNIX shell prompt:

```
% windsh vxsim7@aven
...```

In WindSh, unset the `SH_GET_TASK_IO` configuration parameter to avoid any conflict with the WindSh local redirection (see *Tornado User’s Guide: 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:
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:

```
wmxcl> memBlockWriteFile mblk0 -
```

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

```
wmxcl> 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 WindSh:

```
-> ?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 `wmxcl` and issue this command:

```
wmxcl> wtxVioWrite 1 -string "99\n"
```

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

```wmxcl
value = 1 = 0x1
-> myVioData
myVioData = 0x39d188: value = 99 = 0x63 = 'c'
```
4.4.12 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]
0x358908
```

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.175), or `eventPoll` (see Checking the Event Queue, p.174), as shown in the following example:

```
wtxtcl> wtxEventGet
CALL_RETURN 0x358908 0x3c
```

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
wtxtcl> eventPoll 1000
CALL_RETURN 0x358908 0 0
```
4.4.13 Working With Multiple Connections

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

\begin{verbatim}
set h1 [wtxToolAttach vxsim7]
set h2 [wtxToolAttach mv147]

display handles:
set h1 vxsim7@aven
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 \texttt{mv147}. To examine and manipulate the handle stack, use the \texttt{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 \texttt{vxsim7} at the top of the stack; further WTX commands are directed there.

\begin{verbatim}
wtxHandle mv147@aven vxsim7@aven
wtxHandle $h1 vxsim7@aven
\end{verbatim}

You can also direct any WTX Tcl command to any connected handle without manipulating the stack. The generic \texttt{-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 \texttt{wtxHandle} without arguments provides the list of connected targets, and running \texttt{wtxTsInfoGet} with the \texttt{-hwtx} option queries the correct target. Output is directed to \texttt{stdout}.

\begin{verbatim}
foreach handle [wtxHandle] {
set info [wtxTsInfoGet -hwtx $handle]
puts stdout [lindex $info 4]
}
\end{verbatim}

4.4.14 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 ""
```

### 4.4.15 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 0xeeeeeee 0x100
  }
}
Error: WTX Error 0x100ca (AGENT_MEM_ACCES_ERROR)
wtxtcl> set errorInfo
WTX Error 0x100ca (AGENT_MEM_ACCES_ERROR)
while executing
  "wtxMemRead 0xeeeeeee 0x100"
("while" body line 2)
invoked from within
  "while 1 {
    wtxMemRead 0xeeeeeee 0x100
  }"
("foreach" body line 2)
invoked from within
  "foreach x {1 2 3} {
    while 1 {
      wtxMemRead 0xeeeeeee 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 online reference material under `Tornado API Guide>WTX Tcl Library`.) 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.

```tcl
proc myErrorHandler {handle cmd err tag} {
    puts stdout "handle $handle\n    cmd $cmd\n    err $err\ntag $tag"
    error $err
}
```

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

```tcl
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:

```tcl
wtxMemRead 0xeeeeeeee 0x1000
handle vxsim7@aven
cmd wtxMemRead 0xeeeeeeee 0x1000
err WTX Error 0x100ca (AGENT_MEM_ACCES_ERROR)
tag
Error: WTX Error 0x100ca (AGENT_MEM_ACCES_ERROR)
```

The first four lines of output were printed by the handler. The fifth was printed by the `wttcl` 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.
4.5 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 4.4 WTX Tcl API, p.167. 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.

4.5.1 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 VxWorks Programmer's Guide: Coding Conventions; in other words, underscores are removed and all words but the first are capitalized.

Each WTX C interface has an online reference entry under Tornado API Guide>WTX C Library. For information on the online reference material, see the Tornado User's Guide: Documentation Guide and Tornado 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.

4.5.2 WTX C API Archive

The WTX C API is archived in `installDir/host/hostType/lib/libwtxapi`. For a complete description of the use of this library to build a Tornado application see 6. Adding Tools to Tornado.

4.5.3 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);

4.5.4 Obtaining Target Server Information

The Tornado registry maintains information about executing target servers. To find
the list of registered target servers, you can call \texttt{wtxInfoQ}(). To find information
on a particular target server, you can call \texttt{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);
...

4.5.5 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 \texttt{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 \texttt{wtxTsInfoGet}().

#include "wtx.h"

/* attach to a target server */

if (wtxToolAttach (wtxh, "tJohn.*", "newapp") != OK)
    {
        wtxTerminate (wtxh); /* terminate session */
        return (ERROR);
    }
...
4.5.6 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 6. Adding Tools to Tornado.

To observe this application, start the target browser as well as the application:

```
% wtxapp targetServerName &
% browser targetServerName &
```

If you place the browser in auto-update mode, the memory usage charts demonstrate the functionality of this simple application. On UNIX, if the Tornado launcher is running and your target server is selected, you can observe `wtxapp` in the list of attached tools.

Example 4-2  WTX C API Application

/* 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
--------------
01o,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 deallocates 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)) == 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, 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, 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
*/

```c
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
*/

```c
LOCAL void wtxAppTerminate (void)
{ 
    int ix;
    printf ("
Freeing all the memory blocks
"); 
    for (ix = 0; ix < NUM_BLOCKS; ix++)
    { 
        if (blockTab [ix] != (TGT_ADDR_T) 0)
            wtxMemFree (wtxh, blockTab [ix]);
    }
    wtxToolDetach (wtxh); /* detach from target server */
    wtxTerminate (wtxh); /* terminate WTX session */
    exit (0); /* bye */
    
```

"The WTX Protocol"
/* 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
#endif
}

4.6 Integrating WTX with Applications

4.6.1 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 Tornado. 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()`:

```c
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()`:

```c
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()`:

```c
void wtxTclHandleRevoke (HWTX hWtx)
```

### 4.6.2 Integrating WTX Tcl with Other Tcl Applications

You can integrate WTX Tcl with other Tcl interpreters. The shared library `installDir/host/hostType/lib/libwtxtcl` can be linked with any Tcl 8.0 application. You must initialize the library with a call to `wtxTclInit()`, supplying the Tcl interpreter handle as an argument:

```c
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.
5

Extending Tornado Tools

5.1 Introduction

This chapter shows how to extend Tornado tools by adding new procedures and capabilities to the Tornado shell and how to run a new procedure from the browser. The primary tool for developing these extended capabilities is wtxtcl, the Tcl binding to the WTX API. Most Tornado tools are implemented using Tcl, and it is straightforward to add function using Tcl as well. This chapter assumes familiarity with the material in 4.4 WTX Tcl API, p.167.

5.2 Outlining the Example Problem

This chapter steps through the process of extending tool functionality. The example problem is in two parts: determine the status of all attached network interfaces, and display this information both with the Tornado shell (WindSh) and with the Tornado browser. Determining the network interface status requires that you create code to query the state of certain data structures on the target. The information needed is similar to that produced by the target shell routine ifShow(). Once you extract the information, you can create a new host shell procedure that displays the information. After completing the shell implementation, you can add a new display to the Tornado browser as well.

To see what information is needed, review the output of the target shell ifShow() routine.
-> *ifShow*

dc (unit number 0):

Flags: (0x8063) UP BROADCAST RUNNING ARP MULTICAST
Type: ETHERNET_CSMACD
Internet address: 147.11.36.111
Broadcast address: 147.11.36.255
Netmask 0xffff0000 Subnetmask 0xffffffff
Ethernet address is 08:00:3e:24:dd:70
Metric is 0
Maximum Transfer Unit size is 1500
30954 packets received; 6 packets sent
30954 multicast packets received
5 multicast packets sent
0 input errors; 0 output errors
0 collisions; 0 dropped

To simplify this example, the network mask information is omitted (it is maintained in a different linked list); instead, the example focuses on the information that is directly available in the *ifnet* linked list of structures.

The solution is implemented through three files of Tcl code. The Tornado shell and browser share the Tcl code which gathers information from the target. The shell calls these information-gathering routines and then formats the output for printing. The browser uses these routines to supply graphical objects with data, providing a more interactive presentation of the same data. Figure 5-1 shows the directory location and relationship of the three elements.

All Tcl-based Tornado tools have a directory in the resource hierarchy where you can place files of Tcl extension code: *installDir*/host/resource/tcl/app-config/toolName. When each tool starts, it reads all the extension files it finds in order. This means any extension to a tool can be packaged as a set of files which are inserted into the Tornado tree. The tools automatically read these files, allowing Tornado to be extended without modifying any of the existing files.

Since extension files are read in a specific order, a “patch” file can be inserted after an existing file to modify the behavior of an existing service (for example, by redefining a Tcl procedure found in the first file). To add a later modification that extends the *NetShow* extension you are creating, you might provide a file called *01NetShow2.tcl*, which would be read immediately after *01NetShow.tcl*.

The files located in *installDir*/host/resource/tcl are not automatically sourced. Include a source statement in any extension files that need to access this shared code.

1. Files are read in shell collating order (the same order that the files would be displayed by the *ls* routine). This is why these files typically start with two numbers: so that new additions to this directory can be interleaved with existing files easily.
Figure 5-1  **Organization of Tcl Extension Files**

![Diagram](image)

**net-core.tcl**
Tcl code to gather network information in Tcl list format

**01NetShow.tcl**
Tcl code to format information in `ifShow()` format

**01NetBrws.tcl**
Tcl code to prepare information for a browser panel

**CAUTION:** During development, you may wish to place your working files in your `homeDir/.wind` directory. UNIX users can define the `SHOME` constant to be `homeDir`. Files in the `SHOME/.wind` directory are automatically sourced when you restart the tool, and this practice protects other Tornado users from any problems with your development code. Windows users will need to source the files. (For more information, see the *Tornado User’s Guide: Directories and Files.*)

### 5.3 Extracting Information from a Structure

Extracting the information needed from the `ifnet` structure can be broken into several steps. First, determine how the data is stored. Then locate the first element, identify the remaining elements, and finally, query each element to obtain its contents. As you proceed through this chapter, you will build up expressions as you go. For example, you will write a Tcl expression and then use the value returned by that routine as an argument for the next Tcl expression.
5.3.1 Definitions and Structures

A network interface is an entity over which packets can be sent. It can represent a physical Ethernet hardware device or a logical device (such as the loopback interface) that simply relays packets. Generally, when a VxWorks target has Ethernet capability, there are at least two interfaces: one for the device and one loopback interface for local packet communication.

A network interface in VxWorks is represented as a linked list of structures. The structure that contains most of the information for a network interface, **struct ifnet**, and the subsidiary structure, **struct if_data**, are found in the include file `installDir/target/h/net/if.h`:

```c
struct ifnet {
    char *              if_name;        /* name, as "en" or "lo" */
    struct ifnet *      if_next;        /* all struct ifnets are chained */
    struct ifaddr *     if_addrlist;    /* linked list of addresses per if */
    int                 if_pcount;      /* number of promiscuous listeners */
    caddr_t             if_bpf;         /* packet filter structure */
    u_short             if_index;       /* numeric abbreviation for if */
    short               if_unit;        /* sub-unit for lower level driver */
    short               if_timer;       /* time until if_watchdog called */
    struct if_data      if_data;        /* stats and other data about if */
    struct mblk *       pInmMblk;       /* chain of multicast addresses */
    ...
    struct ifqueue      if_snd;         /* output queue */
};
```

```c
struct  if_data {
    /* generic interface information */
    u_char  ifi_type;               /* ethernet, tokenring, etc. */
    u_char  ifi_addrlen;            /* media address length */
    u_char  ifi_hdrlen;             /* media header length */
    u_long  ifi_mtu;                /* maximum transmission unit */
    u_long  ifi_metric;             /* routing metric (external only) */
    u_long  ifi_baudrate;           /* linespeed */
    /* volatile statistics */
    u_long  ifi_ipackets;           /* packets received on interface */
    u_long  ifi_ierrors;            /* input errors on interface */
    u_long  ifi_opackets;           /* packets sent on interface */
    u_long  ifi_oerrors;            /* output errors on interface */
    u_long  ifi_collisions;         /* collisions on csma interfaces */
    u_long  ifi_ibytes;             /* total number of octets received */
    u_long  ifi_obytes;             /* total number of octets sent */
    u_long  ifi_imcasts;            /* packets received via multicast */
    u_long  ifi_omcasts;            /* packets sent via multicast */
    u_long  ifi_iqdrops;            /* dropped on input, this interface */
    u_long  ifi_noproto;            /* destined for unsupported protocol */
    u_long  ifi_lastchange;         /* last updated */
};
```
In VxWorks, the global variable \texttt{ifnet} points to the structure that contains the first network interface. If there is an additional network interface, the last item in the structure points to the next structure. (See Figure 5-2.)

Figure 5-2  \textbf{A Linked List of Structures}

5.3.2 \textit{Locating the Start of the First Structure}

The first piece of information needed is the address of the first node of the list, which is stored in the global variable \texttt{ifnet}. The address of this symbol can be found in the target server symbol table. The \texttt{wtxtcl} procedure to query the symbol table is:

\begin{verbatim}
  wtxtcl> wtxSymFind -name ifnet
    _ifnet 0x95ae0 0x9 0 0 ""
\end{verbatim}
This procedure returns a list of information about the symbol, which includes a pointer to its location in memory. Use `wtxMemRead` to find the location of the first structure by extracting the symbol address from the list returned by `wtxSymFind` and using this as an argument to `wtxMemRead`:

```
mb1k0
wtxtcl> set ifnet [memBlockGet -l $block]
0x001a3098
wtxtcl> memBlockDelete $block
wtxtcl> set ifnet
0x001a3098
```

Read a memory block of 4 bytes and store the block handle in the variable `block`. Then fetch the 4 bytes in this block to the variable `ifnet` and free the block. You now know the location of the first element of the linked list.

### 5.3.3 Listing All Network Interfaces

One algorithm to move from one structure of the linked list to the next looks like the following:

```
p = ifnet;
while (p != NULL)
{
    print p;
    p = p->if_next;
}
```

You could implement this with WTX Tcl by repeatedly using `wtxMemRead`, but in Tornado there is a simpler way: you can write a Gopher script and send it to the target agent from the Tornado shell. The target agent executes it in one step and returns the results. Here is the Gopher implementation of the algorithm:

```
ifnet {! + 4 *}
```

For detailed information on writing a Gopher script, see 4.3.11 Gopher Support, p.160. A summary of the function of the script tokens follows.

- `ifnet` is an unsigned integer constant which sets the value of the Gopher pointer.
- `open brace` (`{`) introduces a loop; the subprogram inside the braces is executed while the pointer is not `NULL`.
- `exclamation` (`!`) copies the value of the pointer to the Gopher tape (result buffer).
Extending Tornado Tools

+4 increments the pointer by this value. (We computed the value 4 by inspecting the sizes of the constituent elements.)

asterisk (*)

dereferences the pointer.

close brace ()

ends the subroutine. (Note that for all Gopher constructions using matching delimiters (l), <>, and so on) the “outer” value of the pointer is saved during the execution of the inner material, and restored when the closing delimiter is executed.)

Now that you have a Gopher script to move the pointer from the first element of the first structure to the first element of each successive structure in the linked list, you need to write a Tcl routine that uses the Gopher script to traverse the linked list, printing out the address of each node. The wtxGopherEval procedure allows you to send a Gopher script to the target for interpretation and execution. The procedure netIfList returns a list of network interface descriptors (pointers to ifnet structures):

```tcl
proc netIfList {} {
    set ifnet [memBlockGet -l $block]
    return [wtxGopherEval "$ifnet {! +4 *}"]
}
```

Try this routine from the shell by using the ? character to toggle to the shell Tcl prompt:

```
-> ?
tcl> source homeDir/.wind/netIfList.tcl
0x94e8c 0x9531c
```

Restart the shell before switching to the Tcl prompt or source the file.

5.3.4 Listing the Contents of Each Structure

The next step is to write a routine that extracts the relevant fields from one network interface descriptor in the form of a list. The list serves as an intermediate representation which is the basis for the shell’s and browser’s presentations of the data.

Use the Gopher language interpreter to extract the data. (For a detailed discussion of the Gopher operators, see 4.3.11 Gopher Support, p. 160.) The first field of interest in the ifnet structure is the if_name field. You used netIfList to find the address of
the first byte of the structure, which is also the address of the string. Use the following Gopher string to dereference this pointer and write the string to the tape:

```plaintext
<*$>
```

Remember that the angle brackets delineate a subcontext in which the outer pointer value is saved. When the * is executed, the pointer is dereferenced, but outside the angle brackets, the old value of the pointer is preserved. The operator $ copies the string to the tape. After leaving the subcontext, the pointer still points to the top of the structure.

Now, fetch the next two interesting fields, if_unit and if_flags. The pointer is still pointing to if_name, therefore use a signed constant to advance it over the twenty-two bytes of the pointer (+22). The @ operator with the w suffix (16-bit word) writes the value at the pointer to the tape. A side effect of the @ operator is that the pointer value is advanced past the data written; in effect, @w copies the 16-bit value at the pointer to the tape and increases the value of the pointer by 2. Next, fetch if_mtu and if_metric, which are stored in the if_data structure. Use a signed constant (+2) to skip over the field if_timer, and (+4) to skip the first bytes (including one byte for padding) of the if_data structure to get the if_mtu and if_metric fields. To read the four fields starting with the pointer at the if_unit field, the fragment is:

```plaintext
+22 @w +2 @w +4 @@
```

For the five packet-statistics fields (ifi_ipackets through ifi_collisions), advance the pointer to the first field and use the @ operator five times.

```plaintext
+4 @@@@@
```

The routine netIfInfo combines these fragments to return the values of interest, given the address of a single network interface.

```plaintext
proc netIfInfo {netIf} {
    return [wtxGopherEval "$netIf <*$> +22 @w +2 @w +4 @@ +4 @@@@@"]
}
```

Test the two routines with the following expression:

```plaintext
tcl> foreach netIf [netIfList] {puts stdout [netIfInfo $netIf]}
eti 0 0x8063 0x5dc 0 0x19b 0 0x1 0 0
lo 0 0x8069 0x8000 0 0 0 0 0 0
```

202
You now have most of the information needed to duplicate the output produced by the target shell routine `ifShow()`. You still need the Internet addresses of the interfaces, which are stored in a linked list attached to each node of the `ifNet` list by way of the `if_addr` structure. The following example shows the definition of the `if_addr` structure:

```c
struct ifaddr {
    struct sockaddr * ifa_addr;       /* address of interface */
    struct sockaddr * ifa_dstaddr;    /* other end of p-to-p link */
    #define ifa_broadaddr ifa_dstaddr /* broadcast address interface */
    struct sockaddr * ifa_netmask;    /* used to determine subnet */
    struct ifnet * ifa_ifp;           /* back-pointer to interface */
    struct ifaddr * ifa_next;         /* next address for interface */
    void (*ifa_rtrequest)();         /* check or clean routes */
    u_short ifa_flags;               /* mostly rt_flags for cloning */
    short ifa_refcnt;                /* count of references */
    int ifa_metric;                  /* cost of going out this */
};
```

The second set of four bytes in the `sockaddr` structures contains the Internet addresses. Note that they are stored in Internet byte order (big-endian). If you were to use an `@l` operator to extract them, the result would come out swapped if the target architecture is little-endian. To avoid this, read the addresses as four one-byte values. Given a pointer to the first node in a list of these structures and the various offsets, the Gopher script to extract the two addresses in each node (`ifa_addr` and `ifa_dstaddr`) is:

```c
<+{+4 @b@b@b@b 0}> +4 <+{+4 @b@b@b@b 0}> +12 *0
```

The `netIfAddrList` routine takes a network interface descriptor, advances to the `if_addrlist` field and dereferences it, and then repeatedly executes this fragment to copy the Internet addresses (we skip the first address, which represents an internal linkage).

```c
proc netIfAddrList {netIf} {
    return [wtxGopherEval "$netIf +8 * {<+{+4 @b@b@b@b 0}> +4
      <+{+4 @b@b@b@b 0}>+12 *0}>"
```
Testing from the shell yields:

```tcl
foreach netIf [netIfList] {puts stdout [netIfAddrList $netIf]}
0x93 0xb 0x50 0xca 0x93 0xb 0x50 0xff
0x7f 0 0 0x1 0x7f 0 0 0x1
```

Each interface has two addresses so each result has eight octets. The program that calls this routine must partition the results into four-octet sets.

By designing these routines to return results in Tcl list form, you have created information gathering routines that can be used in multiple contexts. They can be used by the shell, the browser, or other any Tcl application. Storing them in the `resource/tcl` directory makes them available to all Tcl applications (see Figure 5-1).

NOTE: Wind River Systems has developed a “Gopher generator” tool which generates gopher scripts from a C structure. This tool is not supported, but is available free of charge. It can be downloaded from the Wind River System WEB server (www.wrs.com).

5.4 The Shell Presentation

At this point, you have extracted all the information needed to write your own WindSh version of `ifShow()`. The remaining task is to format the `ifShow()` output. The first two formatting routines are general and should be placed in `installDir/host/resource/tcl/net-core.tcl`. Place the final procedure in `installDir/host/resource/tcl/app-config/WindSh/01NetShow.tcl` because it is shell specific.

The first formatting routine, `ifFlagsFormat`, converts the `if_flags` field to a list of bit names by checking each flag bit (see the include file `installDir/target/h/net/if.h`):

```tcl
proc ifFlagsFormat {flags} {
    set result ""
    if {($flags & 0x1)} {append result "UP 
    if {($flags & 0x2)} {append result "BROADCAST 
    if {($flags & 0x4)} {append result "DEBUG 
    if {($flags & 0x8)} {append result "LOOPBACK 
    if {($flags & 0x10)} {append result "POINT-TO-POINT 
    if {!($flags & 0x20)) {append result "TRAILERS 
    if {($flags & 0x40)} {append result "RUNNING "
    ```
Extending Tornado Tools

if {!($flags & 0x80)}   {append result "ARP ")
if {$flags & 0x100}     {append result "PROMISC ")
if {$flags & 0x200}     {append result "ALLMULTI ")

return $result
}

The second formatting routine, ifAddrFormat, prints an Internet address in the standard dotted-decimal form.

proc ifAddrFormat {octetList} {
    return [format "%d.%d.%d.%d" 
            [lindex $octetList 0] 
            [lindex $octetList 1] 
            [lindex $octetList 2] 
            [lindex $octetList 3]]
}

All these routines are combined in a single file, which is stored in the installDir/host/resource/tcl directory.

Example 5-1 net-core.tcl

proc netIfList {} {
    set ifnet [memBlockGet -l $block]
    return [wtxGopherEval "$ifnet {! +4 *}" ]
}

proc netIfInfo {netIf} {
    return [wtxGopherEval "$netIf <*$> +22 @w +2 @w +4 @@ +4 @@@@"]
}

proc netIfAddrList {netIf} {
    return [wtxGopherEval "$netIf +8 * {+16 * {<*{+4 @b@b@b@b 0}> +4 <*+4 @b@b@b@b 0>>+12 *}0}"
          <+{(+4 @b@b@b@b 0)>+12 *0}>]
}

proc ifFlagsFormat {flags} {
    set result ""
    if {$flags & 0x1}       {append result "UP ")
    if {$flags & 0x2}       {append result "BROADCAST ")
    if {$flags & 0x4}       {append result "DEBUG ")
    if {$flags & 0x8}       {append result "LOOPBACK ")
    if {$flags & 0x10}      {append result "POINT-TO-POINT ")
    if !($flags & 0x20)     {append result "TRAILERS ")
    if {$flags & 0x40}      {append result "RUNNING ")
    if !($flags & 0x80))   {append result "ARP ")
    if {$flags & 0x100}     {append result "PROMISC ")
    if {$flags & 0x200}     {append result "ALLMULTI ")

    return $result
}
To complete the new version of the shell routine `ifShowSh`, call the procedures you created in the proper sequence and format the output. This final routine, `ifShowSh`, is declared as a `shellproc` rather than a `proc`. WindSh makes a Tcl procedure declared as a `shellproc` available to the C-expression-interpreter prompt of the shell. It does this by defining the procedure under another name and recording the original name of the procedure in a list of functions that may be called from the C shell prompt. Such functions must take a variable argument list, indicated in Tcl by the word `args`, because such functions are invoked by the C-expression interpreter with ten integer arguments, just as built-in shell routines are. Store this file in the `installDir/host/resource/tcl/app-config/WindSh` directory.

Example 5-2 01NetShow.tcl

```tcl
# source the data extraction and formatting routines
source [wtxPath host resource tcl]net-core.tcl
# extract the data and display it in the shell
shellproc ifShowSh {args} {
    foreach netIf [netIfList] {
        set ifInfo [netIfInfo $netIf]
        puts stdout [format "%s (unit number %d):" 
            [lindex $ifInfo 0] [lindex $ifInfo 1]]
        puts stdout [format "Flags: (%#x) %s" 
            [lindex $ifInfo 2] [ifFlagsFormat [lindex $ifInfo 2]]]
        set ifAddrList [netIfAddrList $netIf]
        while {([llength $ifAddrList] >= 8)} {
            set iAddr [lrange $ifAddrList 0 3]
            set dAddr [lrange $ifAddrList 4 7]
            set ifAddrList [lrange $ifAddrList 12 end]
            puts stdout "Internet address: [ifAddrFormat $iAddr] "
            puts stdout "Destination/Broadcast address: \"
                [ifAddrFormat $dAddr]"
        }
        puts stdout [format "Metric is %d" 
            [lindex $ifInfo 4]]
        puts stdout [format "Maximum Transfer Unit size is %d" 
            [lindex $ifInfo 3]]
        puts stdout [format "%d packets received; %d packets sent" 
            [lindex $ifInfo 5] [lindex $ifInfo 7]]
        puts stdout [format "%d input errors; %d output errors" 
            [lindex $ifInfo 6] [lindex $ifInfo 8]]
    }
}```
puts stdout [format " %d collisions" \
  [lindex $ifInfo 9]]
}
}

A side effect of installing a shell extension as a shellproc is that it becomes available at the CrossWind prompt. When CrossWind starts, it reads the WindSh Tcl files and makes new procedures that invoke the shell functions available. These procedures are automatically renamed to be consistent with GDB naming conventions; for example, you can invoke *ifShow()* by typing `wind-if-show` at the GDB prompt. This can be done any time after GDB is attached to a WTX target:

```gdb
(gdb) tar wtx mv147
Connecting to target server...
Connected to mv147@aven.
    Attached to target server mv147@aven, CPU is MC68020.
Looking for all loaded modules:
    vxWorks: (no debugging symbols found)...ok
Done.
(gdb) wind-if-show
ln (unit number 0):
    Flags: (0x63) UP BROADCAST RUNNING ARP
    Internet address: 147.11.80.39
    Destination/Broadcast address: 147.11.255.255...
```

The shell extension *ifShow()* does not take any parameters, but it could easily be modified to do so. Shellprocs (that is, Tcl routines designed to be invoked by the WindSh C interpreter) are always invoked with ten integer arguments. If a string is given as a parameter at the shell’s C prompt, that string is stored on the target and the address is substituted as the parameter. This allows shellprocs to simulate target routines; for example, if a shellproc were to spawn a task whose name was given as one of the parameters of the shellproc, it would be possible to supply that string address directly to the *taskSpawn()* call.

### 5.5 The Browser Presentation

Most information in the Tornado browser is presented in the form of windows. For an introduction to the browser, see the *Tornado User’s Guide: Browser*. The following sections step through the process of creating display windows in the browser and using them to display the data extracted in *5.3 Extracting Information from a Structure*, p.197.
5.5.1 UNIX Implementation

Creating the Hierarchy Window Display

In the UNIX browser, most information is presented in hierarchy windows. These are windows that display information in a hierarchical or list format. The windows are defined using UI Tcl; for more information, see the on-line help.

Use the core routines described in 5.3 Extracting Information from a Structure, p.197 to produce the browser panel shown in Figure 5-3. The steps in creating the new window display are:

- defining the window
- defining the data format
- displaying the data in the window

Defining a New Window

Hierarchy windows are created with the Tcl procedure hierarchyCreate. The -destroy option calls the routine ifDestroyed if the user dismisses the new window from the window manager (see Installing the New Browser Panel, p.211). The -change option causes any values that have been updated between one evaluation and the next to be highlighted. (In Figure 5-3 this occurred with packets received,
packets sent, and errors.) The new window is called Network Interfaces. The complete command to create the window is:

```
hierarchyCreate -destroy ifDestroyed -change "Network Interfaces"
```

**Defining the Data Layout**

The next step is to use `hierarchySetStructure` to describe the structure of the hierarchy that the window will display. For a detailed explanation of this and other procedures, see the on-line help.

The format of data for a hierarchy window mirrors that of the structure string. Arrays of objects are surrounded by braces. Simple objects follow one another separated by spaces; nested objects are placed in braces. The output of the procedure is a nested list in the standard Tcl format.

The first six elements of the display (name through mtu) are simple objects, represented in the structure description by their names:

```
name flags addr destaddr metric mtu
```

Each nested element consists of a bracketed list which itself has two elements, the first being the name of the header item (the folder) and the second the sublist of subordinate items:

```
{packets {received sent}}
{errors {input output collisions}}
```

To arrange for an array of structures (in this example, a listing of data for all network interfaces), create a list whose first element is the special name + and whose second element is the substructure, enclosed in braces. This bracketed list becomes the second argument for the `hierarchySetStructure` procedure; the first argument is the window name.

```
hierarchySetStructure "Network Interfaces" \
  {+ {name flags addr destaddr metric mtu \ 
     {packets {received sent}} \ 
     {errors {input output collisions}}}}
```

This code must be in the browser initialization file so that the browser creates this hierarchy when it starts (`host/resource/tcl/app-config/Browser/01NetBrws.tcl`).

**Generating the Data**

The next step is to create a procedure that arranges the data from the network information-gathering routines (described in 5.3 Extracting Information from a Structure, p. 197) into a form suitable for the `hierarchySetValue` procedure. The
format of data for a hierarchy mirrors that of the structure string. For the window shown in Figure 5-3, the value string used to fill the data slots is:

```plaintext
{{ul0 {(0xf1) UP POINT-TO-POINT RUNNING } 127.0.1.7 147.11.80.6 \ 0 1536 {459 922} {0 0 0}} 
{lo0 {(0x69) UP LOOPBACK RUNNING ARP } 127.0.0.1 0.0.0.0 \ 0 4096 {3 3} {0 0 0}}}
```

Create a procedure that generates such a list (including formatting braces) from the data returned by the gathering routines. The Tcl `lappend` procedure serves this purpose by automatically adding braces when necessary (as when the new list element contains embedded spaces). First consider the data for just one interface, whose address is presumed to be in the variable `netIf`. Initialize a variable, `thisIf`, to the empty string, and then append the items called for one by one.

```tcl
set thisIf ""
set ifInfo [netIfInfo $netIf]
lappend thisIf [format "%s%d" [lindex $ifInfo 0] \ [lindex $ifInfo 1]]
lappend thisIf [format "(%#x) %s" \ [lindex $ifInfo 2] [ifFlagsFormat [lindex $ifInfo 2]]]
set ifAddrList [netIfAddrList $netIf]
lappend thisIf [ifAddrFormat [lrange $ifAddrList 0 3]]
lappend thisIf [ifAddrFormat [lrange $ifAddrList 4 7]]
# metric
lappend thisIf [lindex $ifInfo 4]
# mtu
lappend thisIf [expr [lindex $ifInfo 3]]
# add packet arrays
lappend thisIf [list [expr [lindex $ifInfo 5]] \ [expr [lindex $ifInfo 7]]]
# add error/collision counts
lappend thisIf [list [expr [lindex $ifInfo 6]] \ [expr [lindex $ifInfo 8]] \ [expr [lindex $ifInfo 9]]]
```

To complete the data collection for the window, you need only wrap this code in a loop which queries all network interfaces and accumulates the list generated by the above code into a “list of lists.” Note how the nesting of the `lappend` procedure simplifies the creation of nested lists.

```tcl
proc getAllIf {} {
set allIf ""
foreach netIf [netIfList] {
# code fragment above
lappend allIf $thisIf
}
return $allIf
}
```
This makes it possible to generate the data by calling `hierarchySetValues`. The output will be in a form ready to be displayed as in Figure 5-3:

```
hierarchySetValues "Network Interfaces" [getAllIf]
```

### Installing the New Browser Panel

Two steps are required to add this panel to the browser. First, create a toolbar button to request the display of the panel. Second, insert the panel value-computation routine into the browser main loop so that the information is updated regularly.

Add a toolbar button containing the string “netif” to the browser toolbar, with the following command:

```
toolBarItemCreate netif button {ifPost}
```

The Tcl procedure `ifPost` uses a global variable to keep track of whether the network interface browser is posted or not.

```
set netIfPosted 0
proc ifPost {} {
    global netIfPosted
    hierarchyPost "Network Interfaces"
    set netIfPosted 1
    ifBrowse
}
```

When you created the hierarchy window (in Defining a New Window, p.208), you indicated you wanted to call the procedure `ifDestroyed` when the window manager dismisses the interface browser. The purpose of `ifDestroyed` is to update the state variable.

```
proc ifDestroyed {name} {
    global netIfPosted
    set netIfPosted 0
}
```

You want to update the network interface panel each time the browser updates other windows, either because the user presses the immediate update button or because the browser is in continuous update mode and the update interval elapses.

---

2. To have an icon appear in the toolbar button instead of the name, place an X bitmap file with the same name as the button in the directory `installDir/resource/bitmaps/Browser/controls`. You can do this with any Tornado tool that has a toolbar.
To do this, call `browserAuxProcAdd` with the name of a procedure to invoke each time the browser is in its update cycle:

```tcl
browserAuxProcAdd ifBrowse
```

This procedure, `ifBrowse`, checks the status of the global variable to see if the panel has been posted. If so, it collects the data and installs it in the hierarchy window. This completes the process of collecting data and displaying it in the browser.

```tcl
proc ifBrowse {} {
    global netIfPosted
    if {! $netIfPosted} return
    hierarchySetValues "Network Interfaces" $allIf
}
```

The browser procedures are combined in a file in `installDir/host/resource/tcl/app-config/Browser/01NetBrws.tcl`, which also sources the data extraction routines developed earlier.

```
Example 5-3  01NetBrws.tcl
```

This file should be stored in the `installDir/host/resource/tcl/app-config/Browser` directory.

```tcl
# source the data extraction and formatting routines
source [wtxPath host resource tcl]net-core.tcl

# create the hierarchy window to display the network interface info
hierarchyCreate -destroy ifDestroyed -change "Network Interfaces"

# describe the structure to be displayed in the window
hierarchySetStructure "Network Interfaces" \\
    {+ {name flags addr destaddr metric mtu \\
        {packets {received sent}}
        {errors {input output collision}}}}

# track whether the window is posted or not
set netIfPosted 0
proc ifPost {} {
    global netIfPosted
    hierarchyPost "Network Interfaces"
    set netIfPosted 1
    ifBrowse
}

# update the state variable if we close the window
proc ifDestroyed {name} {

global netIfPosted
set netIfPosted 0
}

# loop through and collect the data for all interfaces
proc getAllIf {} {
    set allIf ""
    foreach netIf [netIfList] {
        set thisIf ""
        set ifInfo [netIfInfo $netIf]
        lappend thisIf [format "%s%d" [lindex $ifInfo 0] [lindex $ifInfo 1]]
        lappend thisIf [format "(%#x) %s" [lindex $ifInfo 2] [ifFlagsFormat [lindex $ifInfo 2]]]
        set ifAddrList [netIfAddrList $netIf]
        lappend thisIf [ifAddrFormat [lrange $ifAddrList 0 3]]
        lappend thisIf [ifAddrFormat [lrange $ifAddrList 4 7]]
        # metric
        lappend thisIf [lindex $ifInfo 4]
        # mtu
        lappend thisIf [expr [lindex $ifInfo 3]]
        # add packet arrays
        lappend thisIf [list [expr [lindex $ifInfo 5]] [expr [lindex $ifInfo 7]]]
        # add error/collision counts
        lappend thisIf [list [expr [lindex $ifInfo 6]] [expr [lindex $ifInfo 8]] [expr [lindex $ifInfo 9]]]
        lappend allIf $thisIf
    }
    return $allIf
}

# see if the window has been posted; if so, collect and display all data
proc ifBrowse {} {
    global netIfPosted
    if {![! $netIfPosted]} return
    hierarchySetValues "Network Interfaces" [getAllIf]
}

# create the button on the toolbar
toolBarItemCreate netif button {ifPost}

# update the display when other browser displays are updated
browserAuxProcAdd ifBrowse
5.5.2 Windows Implementation

The following sample Tcl application extends the Tornado browser by adding a new page to display network interface information. Add extension files following these guidelines:

1. Place all files you want auto-sourced by the browser into the subdirectory `installDir\resource\tcl\app-config\browser` and name them `filename.win32.tcl`. You must source the file that contains the host-independent data extraction and formatting routines:

   ```tcl
   source [wtxPath host resource tcl]net-core.tcl
   ```

2. Append a page title string to the browser global `pageList` variable:

   ```tcl
   lappend pageList "Network Information"
   ```

   This makes the Windows browser aware of the new browser screen.

3. Create a hook routine to be called by the browser when your page description string is selected from the browser page list: `invokeNetworkInformation`.

   ```tcl
   proc invokeNetworkInformation {} {
     global currentBrowserProc
     global autoResize

     allControlsDestroy Browser

     setupNetworkInfoPage

     set currentBrowserProc refreshNetworkInfoPage
     if {$autoResize} {
       windowSizeCallbackSet Browser "adjustNetworkInfoPage"
     }
   }
   ```

   The name has the form:

   `invokeDescriptionStringW/oSpaces`

   where the description string is the page title string appended to `pageList`. This procedure destroys all the previous browser controls and calls procedures to initialize the page, refresh the data, and adjust the tree control when the user resizes the browser window.

4. Perform initial setup for the browser page. This step creates a tree control, initializes its structure, gets the required network information, and inserts it into the tree control.

   ```tcl
   proc setupNetworkInfoPage {} {
     global autoResize
   ```
beginWaitCursor

if {$autoResize} {
    set clientSz [windowClientSizeGet Browser]
    set maxXExtent [expr [lindex $clientSz 0] - 14]
    set maxYExtent [expr [lindex $clientSz 1] - 14]
} {
    set maxXExtent 300
    set maxYExtent 400
}

controlCreate Browser [list hierarchy -name networkInfo \
    -x 7 -y 7 -w $maxXExtent -h $maxYExtent]

controlStructureSet Browser.networkInfo \
    "+ {name flags addr destaddr metric mtu \n        {packets {received sent}}\n        {errors {input output collision}}}"

refreshNetworkInfoPage

endWaitNetworkInfoPage

5. The following callback adjusts the tree control so that it fits properly when the
user resizes the browser window.

proc adjustNetworkInfoPage {} {
    global autoResize

    if [controlExists Browser.networkInfo] {
        set clientSz [windowClientSizeGet Browser]

        if {$autoResize} {
            set maxXExtent [expr [lindex $clientSz 0] - 14]
            set maxYExtent [expr [lindex $clientSz 1] - 14]
        } {
            set maxXExtent 300
            set maxYExtent 160
        }

        controlPositionSet Browser.networkInfo 7 25
        controlSizeSet Browser.networkInfo $maxXExtent \ 
            [expr $maxYExtent - 18]
    }
}

6. The following callback refreshes the data displayed by the tree control.
getAllIf is the wrapper for the shared code in net-core.tcl. It loops through and
collects the data for all network interfaces. (For the complete procedure, see
Example 5-4.)

proc refreshNetworkInfoPage {} {
    controlValuesSet Browser.networkInfo [getAllIf]
}
Example 5-4 gives the complete code to install the sample Tcl application (developed in Generating the Data, p.209) in the browser on a Windows host.

```
Example 5-4  01NetBrws.win32.tcl

This file should be stored in the
installDir\host\resource\tcl\app-config\browser directory.

#########################################################################
# 01NetBrws.win32.tcl - sample UITcl code illustrating user extension of the
# Tornado Browser.
# # Copyright 1994–1997 Wind River Systems, Inc.
# # This sample Tcl application extends the Tornado browser by adding a new
# page to display network interface information.
# #######################################################################
# Do not source this file if the platform is not win32
if {![string match "*win32*" [wtxHostType]]} return

# source the host-independent data extraction and formatting routines
source [wtxPath host resource tcl]net-core.tcl

# hook our page into the browser
lappend pageList "Network Information"

# hook for Target Server Information panel called by Browser
proc invokeNetworkInformation {} {
    global currentBrowserProc
    global autoResize
    allControlsDestroy Browser

    setupNetworkInfoPage

    # hook-in refresh button
    set currentBrowserProc refreshNetworkInfoPage

    # ensure window size reflects Browser's
    if {$autoResize} {
        windowSizeCallbackSet Browser "adjustNetworkInfoPage"
    }
}

# initial setup of the browser page
proc setupNetworkInfoPage {} {
```
Extending Tornado Tools

```c
global autoResize

beginWaitCursor

if {$autoResize} {
    set clientSz [windowClientSizeGet Browser]
    set maxXExtent [expr [lindex $clientSz 0] - 14]
    set maxYExtent [expr [lindex $clientSz 1] - 14]
} {
    set maxXExtent 300
    set maxYExtent 400
}

controlCreate Browser [list hierarchy -name networkInfo \
    -x 7 -y 7 -w $maxXExtent -h $maxYExtent]

controlStructureSet Browser.networkInfo \ 
    "+ {name flags addr destaddr metric mtu \ 
    {packets {received sent}}\ 
    {errors {input output collision}}""

refreshNetworkInfoPage

endWaitCursor

# adjust page layout when user resizes

proc adjustNetworkInfoPage {} {
    global autoResize

    if [controlExists Browser.networkInfo] {
        set clientSz [windowClientSizeGet Browser]
        if {$autoResize} {
            set maxXExtent [expr [lindex $clientSz 0] - 14]
            set maxYExtent [expr [lindex $clientSz 1] - 14]
        } {
            set maxXExtent 300
            set maxYExtent 160
        }

        controlPositionSet Browser.networkInfo 7 25
        controlSizeSet Browser.networkInfo $maxXExtent \ 
            [expr $maxYExtent - 18]
    }
}

# refresh the data displayed by the page

proc refreshNetworkInfoPage {} {
    controlValuesSet Browser.networkInfo [getAllIf]
}
```

217
# loop through and collect the data for all interfaces

 proc getAllIf {} {
    set allIf ""
    foreach netIf [netIfList] {
        set thisIf ""
        set ifInfo [netIfInfo $netIf]
        lappend thisIf [format "%s%d" [lindex $ifInfo 0] \ 
            [lindex $ifInfo 1]]
        lappend thisIf [format "(#%x) %s" \ 
            [lindex $ifInfo 2] [ifFlagsFormat \ 
                [lindex $ifInfo 2]]]
        set ifAddrList [netIfAddrList $netIf]
        lappend thisIf [ifAddrFormat [lrange $ifAddrList 0 3]]
        lappend thisIf [ifAddrFormat [lrange $ifAddrList 4 7]]
        # metric
        lappend thisIf [lindex $ifInfo 4]
        # mtu
        lappend thisIf [expr [lindex $ifInfo 3]]
        # add packet arrays
        lappend thisIf [list [expr [lindex $ifInfo 5]] \ 
            [expr [lindex $ifInfo 7]]]
        # add error/collision counts
        lappend thisIf [list [expr [lindex $ifInfo 6]] \ 
            [expr [lindex $ifInfo 8]] \ 
            [expr [lindex $ifInfo 9]]]
        lappend allIf $thisIf
    }
    return $allIf
}
6

Adding Tools to Tornado

6.1 Introduction

Wind River Systems encourages users and third parties to customize the Tornado environment and to add features to it. This chapter describes how to create a new Tornado tool and add it to the Tornado launcher (in the UNIX environment) or to the Tornado launch toolbar (in the Windows environment). It also provides information about sharing Tornado development such as enhancements of existing tools or useful utilities with other users.

The Tornado environment is structured to make adding tools easy. For information on coding conventions and the Tornado tree, see A. Tcl Coding Conventions and VxWorks Programmer’s Guide: C Coding Conventions.

6.2 Creating a Tornado Application

This section describes how to build and install a simple Tornado application based on the WTX protocol. It also describes how to generate HTML documentation for the new tool.
6.2.1 Example Specifications

The Tornado tool we build and install in this example was presented in 4. *The WTX Protocol* as an example of using the WTX C API. It allocates memory from the target memory pool managed by the target server until no more memory is available. It then frees the allocated memory and repeats this sequence until the user issues a **CTRL+C**. The tool is called *wtxapp*. When the tool is used with the Tornado browser in auto-update mode, it displays a changing memory-in-use chart.

The tool main program is made up of two files, *wtxapp.h* and *wtxapp.c*. It makes use of the WTX C API requests *wtxMemAlloc()* and *wtxMemFree()* . The code for the application is located in 4.5.6 Application Example, p.187.

Note that the *wtxapp* tool installs a signal handler that not only stops the program on **CTRL+C** but also terminates the communication session with the target server properly. All Tornado tools should implement this type of signal handling.

6.2.2 Directories and File Naming

Wind River Systems recommends creating a dedicated directory for each Tornado tool in *installDir/host/src*. This directory contains all source and header files that constitute the tool plus a makefile. For *wtxapp* the files are:

- *installDir/host/src/wtxapp/Makefile*
- *installDir/host/src/wtxapp/wtxapp.c*
- *installDir/host/src/wtxapp/wtxapp.h*

**NOTE:** For UNIX hosts, *installDir* is $WIND_BASE$. For Windows users who use the command line, *installDir* is %WIND_BASE%.

6.2.3 Building the Application

Use a makefile that incorporates the Tornado generic makefile rules and macros found in the directory *installDir/host/include/make* to build Tornado applications. This type of makefile automatically generates a secondary makefile called *Makefile.gen* that contains the dependencies list. It installs the *wtxapp* executable in the *bin* directory for the host (for example, in *installDir/host/sun4-solaris2/bin*). The makefile for *wtxapp* appears in Example 6-1.
The Tornado include files are located in the `installDir/host/include` directory. This path is available in the makefile through the `INCLUDES` macro.

`wtxapp` uses routines from `libwtxapi` shared library. This library is included by the `$UTIL_LIB` macro in the makefile. `wtxapp` is also linked with extra libraries required by a particular tool chain on a given host. These extra libraries are defined by `EXTRA_LIBS` in the makefile.

For more information about the available makefile macros defined with Tornado, see `installDir/host/include/make/generic.mh`.

Example 6-1  

`wtxapp` Makefile

```makefile
# Makefile - for simple Tornado Application
#
# modification history
# --------------------
# 01c,28jan99,wcc  add $(CFLAGS) to $(WTXAPP) rule
# 01b,13jan97,bcc  port to Windows
# 01a,25jul95,p_m  written
#
# DESCRIPTION
# This file contains the makefile rules for building a simple
# Tornado application
#
# SUBSTITUTION
#
# INCLUDES
#       $(WIND_BASE)/host/include/make/generic.mh
#       $(WIND_BASE)/host/include/make/$(HOST).mh
#       $(WIND_BASE)/host/include/make/generic2.mh
#
include $(WIND_BASE)/host/include/make/generic.mh
include $(WIND_BASE)/host/include/make/$(HOST).mh

INCLUDES = $(WIND_INC)
WTXAPP = $(WIND_BIN)/wtxapp

# documentation related variables

DOC_FILES = wtxapp.c
DOC_DIR = $(DOCS_ROOT)/tornado-app/wtxapp
VERSION_DEFINE = $(CFLAGS)
LIBRARIES =
PROGRAMS = $(WTXAPP)
TESTS =
DOCS = doc

## Add special build targets and default here.
```
default: prog

$(WTXAPP): wtxapp.c $(SH_UTIL_LIB)
    $(CC) $(CFLAGS) -DHOST=$(HOST) $(INCLUDES) wtxapp.c -o $@
    $(UTIL_LIB) $(EXTRA_LIBS)

doc: $(DOC_FILES)
    $(WIND_BIN)/refgen $(REFGEN_OPT) -book Tornado_Applications -chapter WTX_Applications -out $(DOC_DIR) $(DOC_FILES)
    $(WIND_BIN)/htmlLink $(DOC_DIR)

include $(WIND_BASE)/host/include/make/generic2.mh

6.2.4 Compiling the Application

UNIX Hosts

The specific commands for compiling wtxapp depend on your host. For example, to compile on a Sun4 workstation running Solaris 2.5.1., issue the following commands from a UNIX shell (this example assume the c-shell; issue the appropriate commands for your shell):

```
setenv WIND_BASE installdir
setenv WIND_HOST_TYPE sun4-solaris2
set path=(installDir/host/$WIND_HOST_TYPE/bin path)
cd installDir/host/src/wtxapp
make HOST=$WIND_HOST_TYPE
```

Windows Hosts

Compiling on a Windows host requires creating a Visual C++ project.
1. Start Microsoft Visual C++ 5.0.
2. Select the Project tab under the New item on the File menu.
3. Select Win32 Console Application and fill in wtxapp in the Project name field. Be sure that the correct path is displayed in the Location field.
4. Select wtxapp.c in the Insert Files into Project dialog box under the Add to Project>Files item on the Project menu.
5. Select Settings on the Project menu. First select the Win32 Debug build target in the Settings for combo box. After debugging, you can change to the Win32 Release target.
Adding Tools to Tornado

6. For your debug build, select the following settings:

Click on the C/C++ tab and:
- Select Code Generation in the Category combo box. Then select Debug Multithreaded DLL under the Use Run-time library option.
- Select Preprocessor in the Category combo box. Add \HOST to Preprocessor definitions and ..\..\include to Additional Include Directories.

Click on the Link tab and leave General selected in the Category combo box:
- Enter ..\..\x86-win32\bin\wtxapp-d.exe under the Output file name option.
- Enter ..\..\x86-win32\lib\wtxapidll-d.lib at the end of the list of Object/library modules.

7. For your release build, select the following settings:

Click on the C/C++ tab and:
- Select Code Generation in the Category combo box. Then select Multithreaded DLL under the Use Run-time library option.
- Select Preprocessor in the Category combo box. Add \HOST to Preprocessor definitions and ..\..\include to Additional Include Directories.

Click on the Link tab and leave General selected in the Category combo box:
- Enter ..\..\x86-win32\bin\wtxapp.exe under the Output file name option.
- Add ..\..\x86-win32\lib\wtxapidll.lib at the end of the list of Object/library modules.

8. Click OK and go to the Build>Set Active Configuration. Select either the Debug or Release version.

9. You are now ready to start a target server and run your new application.

6.2.5 Testing the Application

UNIX Hosts

To test the application, call it from the Unix shell:

```
% wtxapp targetServer
```
Windows Hosts

To test the application, create a custom tool as follows:

1. Select the Customize option under the Tools menu.
2. Click Add and enter wtxApp under the Menu Text: option.
3. Enter one of the following lines under the Tool Command option, depending on whether you want to run the debug version or the release version:
   
   wtxapp-d.exe targetServer@host
   wtxapp.exe targetServer@host

4. Fill in your working directory under the Working Directory option and select Redirect to Child Window.
5. Now click OK.
6. Select wtxApp from the Tools menu to run your application.

6.2.6 Creating Documentation

The man makefile rule permits the HTML documentation generation for the specified files. The wtxapp application code located in 4.5.6 Application Example, p.187 can generate an example of documentation.

The DOC_FILES macro specifies which files are to be documented. The following excerpt from a makefile generates HTML documentation for the given files:

```makefile
DOC_FILES   = wtxapp.c
DOC_DIR     = $(DOCS_ROOT)/tornado-app/wtxapp
... 
DOCS        = doc
...

doc: $(DOC_FILES)
$(WIND_BIN)/refgen $(REFGEN_OPT) -book Tornado_Applications \  
-chapter WTX_Applications -out $(DOC_DIR) $(DOC_FILES) 
$(WIND_BIN)/htmlLink $(DOC_DIR)
```

For the make man command to generate the documentation, the DOCS macro of the makefile must be set to the target doc.

⚠️ CAUTION: The DOCS macro must appear in all the makefiles, or a make man (or rman) would build the application instead of building the doc.
In the makefile, the documentation is generated for the `wtxapp.c` file, and will belong to the book Tornado Applications, in the chapter WTX Applications. The directory where it is located is specified by the `-out` option of `refgen`.

### 6.3 Installing a New Application in Tornado

Tornado applications can be launched from the host command shell, as you did when testing this application. However, it is more convenient to launch them from the launcher (on UNIX hosts) or from the launch toolbar (on Windows hosts). The next sections explain how to install a new application.

#### 6.3.1 The Tornado Launcher (UNIX Hosts)

Once the new Tornado application is working, you can add it to the Tornado launcher. To do this, install a Tcl resource file containing the command that adds a button to the launcher in the `Launch` directory. The name of this file is related to the application you are starting and is prefixed with “01” so that it will be sourced with other applications early in the initialization process. The complete path to the new resource file is:

```
installDir/host/resource/tcl/app-config/Launch/01Wtxapp.tcl
```

The Tcl program shown in Example 6-2 uses the `objectCreate` command to create a new button in the Tornado launcher. This button is associated with the Tcl routine `launchWtxapp` that gets the name of the target server currently selected in the launcher target window and passes it as a parameter to `wtxapp`.

The second parameter of `objectCreate` is the name of a bitmap file to be displayed as an icon at the bottom of the launcher. Add the bitmap file to the resource bitmap directory for the launcher. In this example the complete path to the bitmap is `installDir/host/resource/bitmap/Launch/objects/wtxapp`. This file can be created with an icon editor like `bitmap` from the X11 distribution. For more information about `objectCreate`, see the online material under Tornado API Reference>GUI Tcl Library (UNIX).
Example 6-2  Install wtxapp in Launcher

```
# Wtxapp.tcl - simple wtx application support for launch
#
# modification history
# --------------------
# 01a,25jul95,p_m  written
#*/

# Add a wtxapp activation button to bottom of the launcher
# The associated icon is the default icon (WRS logo) if there is not a
# bitmap file called "wtxapp" in
# installDir/host/resource/bitmaps/Launch/objects.

objectCreate app wtxapp    wtxapp       {launchWtxapp}

########################################################################
#
# wtxapp launch on selected target server
#

proc launchWtxapp () {
    global env
    global tgtsvr_selected
    global tgtsvr_user
    global tgtsvr_lock

    # check for a selected target server
    if {$tgtsvr_selected == ""} {
        noticePost error "Select a target first."
        return
    }

    # check if target server is locked
    if {$tgtsvr_lock == "locked"} {
        noticePost error "Target server locked by $tgtsvr_user."
        return
    }

    # check if we have access authorization on the selected target server
    if {$tgtsvr_lock == "unauthorized"} {
        noticePost error "Access not authorized."
        return
    }

    exec wtxapp $tgtsvr_selected &
}
```
6.3.2 The Launch Toolbar (Windows Hosts)

Once the new Tornado application is working, you can add it to the Tornado launch toolbar. To do this, install a Tcl resource file containing a command to add a button to the toolbar in the Tornado directory. The name of this file is related to the application you are starting and is prefixed with “01” so that it will be sourced with other applications early in the initialization process. The complete path to the new resource file is:

```
installDir\host\resource\tcl\app-config\Tornado\01Wtxapp.win32.tcl
```

The Tcl program shown in Example 6-3 uses the `controlCreate` command to create a new button in the Tornado launcher. This button is associated with the `launchWtxapp` Tcl routine that gets the name of the target server currently selected in the launcher target window and passes it as a parameter to `wtxapp`.

The `-bitmap` parameter of `controlCreate` is the name of a bitmap file to be displayed as an icon on the button. Add the bitmap file to the resource bitmap directory for the toolbar. In this example the complete path to the bitmap is `installDir\host\resource\bitmaps\Launch\controls\wtxapp.bmp`. This file can be created with an icon editor such as Microsoft Paintbrush. For more information about `controlCreate` see the online material under Tornado API Reference>GUI Tcl Library (Windows).

Example 6-3 Install wtxapp on Launch Toolbar

```tcl
# Wtxapp.win32.tcl - simple wtx application support for launcher
#
# modification history
# --------------------
# 01a, 05may96, j_k written
#*/

# Add a wtxapp activation button to the launch toolbar
# The associated bitmap will be the default bitmap (WRS logo) if
# there is no bitmap file called “wtxapp” in
# installDir/host/resource/bitmaps/Tornado, no bitmap appears

set wtxappBitmap [wtxPath host resource bitmaps Tornado]Wtxapp.bmp
controlCreate launch [list toolbarbutton -name wtxapp \
       -callback launchWtxapp \
       -tooltip "Launch Wtxapp" \
       -bitmap $wtxappBitmap]
```
# wtxapp launch on selected target server
#
proc launchWtxapp {} {
    global env

    # first get the name of the selected target
    set tgtsrvr_selected [selectedTargetGet]
    if {$tgtsrvr_selected == ""} {
        messagebox -stopicon "Select a target first."
        return
    }

    # Then try to attach to the tool without any errors.
    # If any, return the error to be displayed for the user.
    if {[catch {wtxToolAttach $tgtsrvr_selected wtxapp} attachRetVal]} {
        messagebox -stopicon "Tool attach failed. $attachRetVal"
        return
    }

    # Next try to get the target server information.
    # If any errors, report to the user
    if [catch "wtxTsInfoGet" info] {
        if {[lindex $info 3] == "(SVR_TARGET_NOT_ATTACHED)"} {
            wtxToolDetach
            messagebox -stopicon "Target not attached"
            return
        }
        messagebox -stopicon "Couldn’t obtain target server information"
        return
    }

    # If we get here, looks pretty good
    set tgtsrvr_user [lindex $info 8]
    set tgtsrvr_lock [lindex $info 11]

    # Now check if target server is locked
    if {$tgtsrvr_lock == "locked"} {
        messagebox -stopicon "Target server locked by $tgtsrvr_user."
        return
    }

    # Now check if we have access authorization on the selected target server
    if {$tgtsrvr_lock == "locked"} {
        messagebox -stopicon "Access not authorized."
        return
    }
# Finally since everything looks good lets launch the target server

```bash
if {{catch {toolLaunch "Wtxapp"  "wtxapp $tgtsvr_selected" \\
\$env(WIND_BASE)/host/x86-win32/bin 0 0 0 0 0 0} result}} {
  puts "\nError: wtxapp launch failed: $result\n"
}
}
```

6.4 Distributing and Publishing New Applications

Users wishing to exchange applications and information with other users can find sources on the World Wide Web. The WRS Web page provides links to a range of subjects, including the Wind River Users Group. Browse the Web page for information on the public software archive, instructions for joining the mailing list “exploder,” and information about users’ group annual meetings. If you do not have access to the Internet, contact your sales representative.

Third parties developing Tornado-compatible products should contact their local Wind River Systems sales office directly or through e-mail (inquiries@wrs.com).
7.1 Introduction

This chapter explains how to integrate third-party toolchains with Tornado 2.0. Integrating a toolchain with Tornado generally involves the following steps:

- Creating a makefile fragment containing the toolchain information.
- Exposing the toolchain to the Tornado Project facility.
- Testing.

Throughout this discussion, Tool denotes the name of your toolchain (for example, gnu or diab).

7.2 Creating the Makefile fragment

For each Cpu your toolchain supports, you must create a file called target/h/make/make.CpuTool. The easiest starting point is by copying installDir/target/h/make/make.Cpugnu and overriding macros for your particular toolchain. At the end of this chapter is a version of make.PPC604diab that has been tested with Tornado 2.0. In it you will find comments about what kinds of changes were made.
7.3 Exposing the Toolchain to the Project Facility

Start a command shell and set up your environment as documented in the Tornado Users Guide. Type the following commands at the command prompt (substituting appropriate values for installDir and Tool):

\% cd installDir/host/resource/tcl/app-config/Project
\% wtxtcl gnuInfoGen.tcl Tool

This generates a file called tcInfo_Tool that the project facility uses when it starts.

7.4 Testing

Copy a Tornado BSP that uses the same CPU as the one you want to test. For example:

\% cd installDir/target/config
\% cp -r mv1604 mv1604_Tool

Go to your newly created BSP directory and edit the makefile. Change the line "TOOL=gnu" to "TOOL=Tool". Also invert the order of the following two lines:

\% include $(TGT_DIR)/h/make/make.$(CPU)$TOOL
\% include $(TGT_DIR)/h/make/defs.$WIND_HOST_TYPE

This allows the toolchain-specific build macros to override all other default definitions. Then go into the BSP directory and type make release. This builds several images, depending on the BSP. It usually builds:

- vxWorks standard vxWorks image
- vxWorks.st vxWorks with built-in symbol table
- bootrom.hex bootrom
- bsp2prj project facility image

Once everything is working, test it from the project facility. Try creating “downloadable application” projects for your toolchain and building them. Try creating a “system image” project from a BSP that uses GNU as the default toolchain. From the “build view”, create a “new build” and create the build by using your new toolchain. Verify that image builds.
7.5 Sample Makefile

# make.PPC604diab - diab compilation tools definitions for PowerPC 604
#
# modification history
# --------------------
# 01a,23Oct98,ms   written from make.PPC604gnu and Diab release notes.
#
# DESCRIPTION
# This file contains PowerPC 604 specific definitions and flags for the
# diab tools (compiler, assembler, linker etc.)
#
# The following macros are changed for T2
# CPP use cc<X> -E -P -xc. cpp<X> no longer exists.
#
# */

# Core definitions (all copied from make.PPC604gnu)

CPU           = PPC604
TOOL          = diab
VX_CPU_FAMILY = PPC
ARCH_DIR        = ppc
TOOLENV               = ppc

# Tools (all these definitions need to be overriden)
# new tools

AR            = dar
AS            = das
CC            = dcc
CXX           = $(CC)
LD            = dld

# override WRS's CC_COMPILER="-traditional"

COMPILE_TRADITIONAL = $(CC) -c $(CFLAGS)
COMPILE_SYMTBL  = $(CC) -c $(CFLAGS)
override CC_COMPILER = $(CC_DDI_FLAGS) -Xdollar-in-ident
override C++_COMPILER = $(CC_DDI_FLAGS) -Xdollar-in-ident

# use gnu tools for the following

CPP           = cpppc -E -P -xc
NM            = nmppc
RANLIB         = ranlib
EXTRACT_BIN_NAME= elfToBin
BINXSYM_NAME  = elfXsym
BINHEX_NAME   = elfHex
DEFEND_GEN_UTIL = $(CPP) -M

# Flags

CC_DDI_FLAGS  = -tPPC403ES:tornado -X4 -Xsmall-data=0 -Xsmall-const=0 -Xstrings-in-text=0
CC_OPTIM_DRIVER =
CC_OPTIM_NORMAL = -O
CC_OPTIM_TARGET = -O
CC_ARCH_SPEC =
CC_WARNINGS_ALL =
DEFINE_CC = -D__asm__=__asm
CFLAGS_AS = $(CC_INCLUDE) $(CC_DEFINE)
LDFLAGS = -X -N -Xsection-align=4
LD_PARTIAL_FLAGS = -X -r4 -Xsection-align=4

# new link flags for the config tool
LD_RAM_FLAGS = -Bt=$(RAM_LOW_ADRS)
LD_ROM_RES_FLAGS = $(ROM_LDFLAGS) -Bt=$(ROM_TEXT_ADRS) -Bd=$(RAM_LOW_ADRS)
LD_ROM_CPY_FLAGS = $(ROM_LDFLAGS) -Bt=$(ROM_TEXT_ADRS) -Bd=$(RAM_HIGH_ADRS)
LD_ROM_CMP_FLAGS = $(ROM_LDFLAGS) -Bt=$(ROM_TEXT_ADRS) -Bd=$(RAM_HIGH_ADRS)

# old link flags for BSP Makefiles
LD_LOW_FLAGS = -Bt=$(RAM_LOW_ADRS)
LD_HIGH_FLAGS = -Bt=$(RAM_HIGH_ADRS)
RES_LOW_FLAGS = -Bt=$(ROM_TEXT_ADRS) -Bd=$(RAM_LOW_ADRS)
RES_HIGH_FLAGS = -Bt=$(ROM_TEXT_ADRS) -Bd=$(RAM_HIGH_ADRS)

# Misc
# WRS default definition of LIBS is lib<CPU><TOOL>.
# This has to be overriden, since WRS only supplies lib<CPU>gnu.
LIBS = $(WIND_BASE)/target/lib/libPPC604gnuvx.a

# override implicit rule for the .s.o rule. Since this
# toolchain's assembler doesn't do C-preprocessing, we
# build assembly files in two steps.
IMPLICIT_RULE_S_O = override
.s.o :
  @ $(RM) @
  $(CC) -P $(CC_INCLUDE) $(CC_DEFINES) <$
  $(AS) $(CFLAGS_AS) *.
  $(RM) *.

# end of make.PPC604gnu
A

Tcl Coding Conventions

A.1 Introduction

This document defines the Wind River Systems standard for all Tcl code and for the accompanying documentation included in source code. The conventions are intended, in part, to encourage higher quality code; every source module is required to have certain essential documentation, and the code and documentation is required to be in a format that is readable and accessible.

The conventions are also intended to provide a level of uniformity in the code produced by different programmers. Uniformity allows programmers to work on code written by others with less overhead in adjusting to stylistic differences. Also it allows automated processing of the source; tools can be written to generate reference entries, module summaries, change reports, and so on.

The conventions described here are grouped as follows:

- **File Heading.** Regardless of the programming language, a single convention specifies a heading at the top of every source file.

- **Tcl Coding Conventions.** The language-specific conventions are roughly parallel to those for C and C++, but must differ to account for the particulars of each programming language. These conventions are divided into the following categories:
  - Module Layout
  - Procedure Layout
  - Code outside of procedure
  - Code Layout
  - Naming Conventions
  - Style
A.2 File Heading

Every file containing C or Tcl code—whether it is a header file, a resource file, or a file that implements a host tool, a library of routines, or an application—must contain a standard file heading. The conventions in this section define the standard for the heading that must come at the beginning of every source file.

The file heading consists of the blocks described below. The blocks are separated by one or more empty lines (in Tcl, empty comment lines) and contain no empty lines within the block. This facilitates automated processing of the heading.

- **Title:** The title consists of a one-line comment containing the tool, library, or applications name followed by a short description. The name must be the same as the file name. This line will become the title of automatically generated reference entries and indexes.

- **Copyright:** The copyright consists of a single-line comment containing the appropriate copyright information.

- **Modification History:** The modification history consists of a comment block: in C, a multi-line comment, and in Tcl a series of comment lines. Each entry in the modification history consists of the version number, date of modification, initials of the programmer who made the change, and a complete description of the change. If the modification fixes an SPR, then the modification history must include the SPR number.

  The version number is a two-digit number and a letter (for example, 03c). The letter is incremented for internal changes, and the number is incremented for large changes, especially those that materially affect the module’s external interface.

  The following example shows a standard file heading from a C source file:

  **Example A-1 Standard File Heading (C Version)**

  ```c
  /* fooLib.c - foo subroutine library */
  /* Copyright 1984-1995 Wind River Systems, Inc. */
  /*
   modification history
   02a,15sep92,nfs added defines MAX_FOOS and MIN_FATS.
   01b,15feb86,dnw added routines fooGet() and fooPut();
   added check for invalid index in fooFind().
   01a,10feb86,dnw written.
  */
  ```
A.3 Tcl Module Layout

A module is any unit of code that resides in a single Tcl file. The conventions in this section define the standard module heading that must come at the beginning of every Tcl module following the standard file heading. The module heading consists of the blocks described below; the blocks are separated by one or more blank lines.

After the modification history and before the first function or executable code of the module, the following sections are included in the following order, if appropriate:

- **General Module Documentation**: The module documentation is a block of single-line Tcl comments beginning by the keyword `DESCRIPTION` and consisting of a complete description of the overall module purpose and function, especially the external interface. The description includes the heading `RESOURCE FILES` followed by a list of relevant Tcl files sourced inside the file.

- **Globals**: The globals block consists of a one-line Tcl comment containing the word `globals` followed by one or more Tcl declarations, one per line. This block groups together all declarations in the module that are intended to be visible outside the module.

The format of these blocks is shown in the following example (which also includes the Tcl version of the file heading):

Example A-2  Tcl File and Module Headings

```tcl
# Browser.tcl - Browser Tcl implementation file
#
# Copyright 1994-1995 Wind River Systems, Inc.
#
# modification history
# ------------------
# 02b,30oct95,jco added About menu and source browser.tcl in .wind.
# 02a,02sep95,pad fixed communications loss with license daemon (SPR #1234).
# 01c,05mar95,jcf upgraded spy dialog
# 01b,08feb95,p_m take care of loadFlags in wtmObjModuleInfoGet.
# 01a,06dec94,c_s written.
#
# DESCRIPTION
# This module is the Tcl code for the browser. It creates the main window and
# initializes the objects in it, such as the task list and memory charts.
#
# RESOURCE FILES
# wpwr/host/resource/tcl/shelbrws.tcl
# wpwr/host/resource/tcl/app-config/Browser/*.tcl
# ...
```
A.4 Tcl Procedure Layout

The following conventions define the standard layout for every procedure in a module.

Each procedure is preceded by the procedure documentation, a series of Tcl comments that includes the following blocks. The documentation contains no blank lines, but each block is delimited with a line containing a single pound symbol (#) in the first column.

- **Banner**: A Tcl comment that consists of 78 pound symbols (#) across the page.
- **Title**: One line containing the routine name followed by a short, one-line description. The routine name in the title must match the declared routine name. This line becomes the title of automatically generated reference entries and indexes.
- **Description**: A full description of what the routine does and how to use it.
- **Synopsis**: The word SYNOPSIS: followed by a the synopsis of the procedure—its name and parameter list between .TS and .TE macros. Optional parameters are shown in square brackets. A variable list of arguments is represented by three dots (...).
- **Parameters**: For each parameter, the .IP macro followed by the parameter name on one line, followed by its complete description on the next line. Include the default value and domain of definition in each parameter description.
- **Returns**: The word RETURNS: followed by a description of the possible explicit result values of the subroutine (that is, values returned with the Tcl return command).

```
# set browserUpdate 0 ;# no auto update by default

RETURNS:
A list of 11 items: vxTicks taskId status priority pc sp errno
timeout entry priNormal name
```
If the return value is meaningless enter N/A:

RETURNS: N/A

- Errors: The word ERRORS: followed by all the error messages or error code (or both, if necessary) raised in the procedure by the Tcl error command.

ERRORS:
"Cannot find symbol in symbol table"

If no error statement is invoked in the procedure, enter N/A.

ERRORS: N/A

The procedure documentation ends with an empty Tcl comment starting in column one.

The procedure declaration follows the procedure heading and is separated from the documentation block by a single blank line. The format of the procedure and parameter declarations is shown in A.6 Declaration Formats, p.241.

The following is an example of a standard procedure layout.

Example A-3  Standard Tcl Procedure Layout

```
# browse - browse an object, given its ID
#
# This routine is bound to the "Show" button, and is invoked when
# that button is clicked. If the argument (the contents of...
#
# SYNOPSIS
# .tS
# browse [objAddr | symbol | &symbol]
# .tE
#
# PARAMETERS
# .IP <objAddr>
# the address of an object to browse
# .IP <symbol>
# a symbolic address whose contents is the address of
# an object to browse
# .IP <&symbol>
# a symbolic address that is the address of an object to browse
#
# RETURNS: N/A
#
# ERRORS: N/A
```

239
A.5 Tcl Code Outside Procedures

Tcl allows code that is not in a procedure. This code is interpreted immediately when the file is read by the Tcl interpreter. Aside from the global-variable initialization done in the globals block near the top of the file, collect all such material at the bottom of the file.

However, it improves clarity—when possible—to collect any initialization code in an initialization procedure, leaving only a single call to that procedure at the bottom of the file. This is especially true for dialog creation and initialization, and more generally for all commands related to graphic objects.

Tcl code outside procedures must also have a documentation heading, including the following blocks:

- **Banner**: A Tcl comment that consists of 78 pound symbols (#) across the page.
- **Title**: One line containing the file name followed by a short, one-line description. The file name in the title must match the file name in the file heading.
- **Description**: A description of the out-of-procedure code.

The following is a sample heading for Tcl code outside all procedures.

Example A-4 Heading for Out-of-Procedure Tcl Code

```
# The following is a sample heading for Tcl code outside all procedures.
```

```
# OISpy.tcl - Initialization code
#
# This code is executed when the file is sourced. It executes the module
# entry routine which does all the necessary initialization to get a
# runnable spy utility.
#
#
# Call the entry point for the module
spyInit
```
A.6 Declaration Formats

Include only one declaration per line. Declarations are indented in accordance with *Indentation*, p. 243, and begin at the current indentation level. The remainder of this section describes the declaration formats for variables and procedures.

**Variables**

For global variables, the Tcl `set` command appears first on the line, separated from the identifier by a tab character. Complete the declaration with a meaningful comment at the end of the same line. Variables, values, and comments should be aligned, as in the following example:

```tcl
set rootMemNBytes 0 ;# memory for TCB and root stack
set rootTaskId 0 ;# root task ID
set symSortByName 1 ;# boolean for alphabetical sort
```

**Procedures**

The procedure name and list of parameters appear on the first line, followed by the opening curly brace. The declarations of global variables used inside the procedure begin on the next line, one on each separate line. The rest of the procedure code begins after a blank line. For example:

```tcl
proc lstFind {list node} {
    global firstNode
    global lastNode
    ...
}
```
A.7 Code Layout

The maximum length for any line of code is 80 characters. If more than 80 characters are required, use the backslash character to continue on the next line.

The rest of this section describes conventions for the graphic layout of Tcl code, covering the following elements:

- vertical spacing
- horizontal spacing
- indentation
- comments

Vertical Spacing

- Use blank lines to make code more readable and to group logically related sections of code together. Put a blank line before and after comment lines.
- Do not put more than one declaration on a line. Each variable and function argument must be declared on a separate line.
- Do not put more than one statement on a line. The only exceptions are:
  - A for statement where the initial, conditional, and loop statements can be written on a single line:
    ```tcl
    for {set i 0} {$i < 10} {incr i 3} {
    ```
  - A switch statement whose actions are short and nearly identical (see the switch statement format in Indentation, p.243).
- The if statement is not an exception. The conditionally executed statement always goes on a separate line from the conditional expression:
  ```tcl
  if {$i > $count} {
    set i $count
  }
  ```
- Opening braces ({}), defining a command body, are always on the same line as the command itself.
- Closing braces (}) and switch patterns always have their own line.
**Horizontal Spacing**

- Put spaces around binary operators. Put spaces before an open parenthesis, open brace and open square bracket if it follows a command or assignment statement. For example:

  ```tcl
  set status [fooGet $foo [expr $i + 3] $value]
  if {&value & &mask} {
  ```

- Line up continuation lines with the part of the preceding line they continue:

  ```tcl
  set a [expr ($b + $c) * \ 
  ($d + $e)]
  set status [fooList $foo $a $b $c \ 
  $d $e]
  if {($a == $b) && \ 
  ($c == $d)} {
  ```

**Indentation**

- Indentation levels are every four characters (columns 1, 5, 9, 13, ...).

- The module and procedure headings and the procedure declarations start in column one.

- The closing brace of a command body is always aligned on the same column as the command it is related to:

  ```tcl
  while { condition }{
      statements
  }
  ```

  ```tcl
  foreach i $elem {
      statements
  }
  ```

- Add one more indentation level after any of the following:
  - procedure declarations
  - conditionals (see below)
  - looping constructs
  - switch statements
  - switch patterns
• The `else` of a conditional is on the same line as the closing brace of the first command body. It is followed by the opening brace of the second command body. Thus the form of the conditional is:

```plaintext
if { condition } {  
  statements
} else {   
  statements
}
```

The form of the conditional statement with an `elseif` is:

```plaintext
if { condition } {  
  statements
} elseif { condition } {  
  statements
} else {   
  statements
}
```

• The general form of the `switch` statement is:

```plaintext
switch [flags] value {  
  a {  
    statements
  }  
  b {   
    statements
  }  
  default {   
    statements
  }
}
```

If the actions are very short and nearly identical in all cases, an alternate form of the switch statement is acceptable:

```plaintext
switch [flags] value {  
  a {set x $aVar}  
  b {set x $bVar}  
  c {set x $cVar}
}
```

• Comments have the same indentation level as the section of code to which they refer (see Comments, p. 245).

• Opening body braces (`{`) have no specific indentation; they follow the command on the same line.
Comments

- Place comments within code so that they precede the section of code to which they refer and have the same level of indentation. Separate such comments from the code by a single blank line.
  - Begin single-line comments with the pound symbol as in the following:
    
    ```tcl
    # This is the correct format for a single-line comment
    set foo 0
    ```
  - Multi-line comments have each line beginning with the pound symbol as in the example below. Do not use a backslash to continue a comment across lines.
    
    ```tcl
    # This is the CORRECT format for a multiline comment
    # in a section of code.
    set foo 0
    # This is the INCORRECT format for a multiline comment \n    # in a section of code.
    set foo 0
    ```

- Comments on global variables appear on the same line as the variable declaration, using the semicolon (;) character:
  
  ```tcl
  set day night ;# This is a global variable
  ```

A.8 Naming Conventions

The following conventions define the standards for naming modules, routines and variables. The purpose of these conventions is uniformity and readability of code.

- When creating names, remember that code is written once but read many times. Make names meaningful and readable. Avoid obscure abbreviations.

- Names of routines and variables are composed of upper- and lowercase characters and no underbars. Capitalize each “word” except the first:
  
  ```tcl
  aVariableName
  ```
Every module has a short prefix (two to five characters). The prefix is attached to the module name and to all externally available procedures and variables. (Names that are not available externally need not follow this convention.)

- **fooLib.tcl** module name
- **fooObjFind** procedure name
- **fooCount** variable name

Names of procedures follow the *module-noun-verb* rule. Start the procedure name with the module prefix, followed by the noun or object that the procedure manipulates. Conclude the name with the verb or action that the procedure performs:

- **fooObjFind** foo - object - find
- **sysNvRamGet** system - non volatile RAM - get
- **taskInfoGet** task - info - get

### A.9 Tcl Style

The following conventions define additional standards of programming style:

- **Comments**: Insufficiently commented code is unacceptable.
- **Procedure Length**: Procedures should have a reasonably small number of lines, less than 50 if possible.
- **Case Statement**: Do not use the *case* keyword. Use *switch* instead.
- **expr and Control Flow Commands**: Do not use *expr* in commands such as *if*, *for* or *while* except to convert a variable from one format to another:
  - **CORRECT**: `if {$id != 0} {
  - **CORRECT**: `if {[expr $id] != 0} {
  - **INCORRECT**: `if {[expr $id != 0]} {

- **expr and incr**: Do not use *expr* to increment or decrement the value of a variable. Use *incr* instead.
  - **CORRECT**: `incr index`
  - **CORRECT**: `incr index -4`
INCORRECT:  

```
set index [expr $index + 1]
```

- **wtxPath and wtxHostType**: Use these routines when developing tools for Tornado. With no arguments, `wtxPath` returns the value of the environment variable `WIND_BASE` with a "/" appended. With an argument list, the result of `wtxPath` is an absolute path rooted in `WIND_BASE` with each argument as a directory segment. Use this command in Tornado tools to read resource files. The `wtxHostType` call returns the host-type string for the current process (the environment variable `WIND_HOST_TYPE` if properly set, has the same value). For example:

```
source [wtxPath host resource tcl]wtxcore.tcl
set backenddir [wtxPath host [wtxHostType] lib backend]
```

- **catch Command**: The `catch` command is very useful to intercept errors raised by underlying procedures so that a script does not abort prematurely. However, use the `catch` command with caution. It can obscure the real source of a problem, thus causing errors that are particularly hard to diagnose. In particular, do not use `catch` to capture the return value of a command without testing it. Note also that if the intercepted error cannot be handled, the error must be resubmitted exactly as it was received (or translated to one of the defined errors in the current procedure):

```
CORRECT:  

```

```if [catch "dataFetch $list" result] {
    if {$result == "known problem"} {
        specialCaseHandle
    } else {
        error $result
    }
```

INCORRECT:  

```
catch "dataFetch $list" result
```

- **if then else Statement**: In an `if` command, you may omit the keyword `then` before the first command body; but do not omit `else` if there is a second command body.

```
CORRECT:  

```

```if {$id != 0} {
    ...
} else {
    ...
```

INCORRECT:  

```
if {$id !=0} then {
    ...
} {
    ...
```
Return Values: Tcl procedures only return strings; whatever meaning the string has (a list for instance) is up to the application. Therefore each constant value that a procedure can return must be described in the procedure documentation, in the RETURNS: block. If a complex element is returned, provide a complete description of the element layout. Do not use the return statement to indicate that an abnormal situation has occurred; use the error statement in that situation.

The following illustrates a complex return value consisting of a description:

```tcl
# Return a list of 11 items: vxTicks taskId status priority pc
# sp errno timeout entry prINormal name
return [concat [lrange $tiList 0 1] [lrange $tiList 3 end]]
```

The following illustrates and simple return value:

```tcl
# This code checks whether the VxMP component is installed:
if [catch "wtxSymFind -name smObjPoolMinusOne" result] {
  if {[wtxErrorName $result] == "SYMTBL_SYMBOL_NOT_FOUND"} {
    return -1       # VxMP is not installed
  } else {
    error $result
  }
} else {
  return 0             # VxMP is installed
}
```

Error Conditions: The Tcl error command raises an error condition that can be trapped by the catch command. If not caught, an error condition terminates script execution. For example:

```tcl
if {$defaultTaskId == 0} {
  error "No default task has been established."
}
```

Because every error message and error code must be described in the procedure header in the ERRORS: block, it is sometimes useful to call error in order to replace an underlying error message with an error expressed in terms directly relevant to the current procedure. For example:

```tcl
if [catch "wtxObjModuleLoad $periodModule" status] {
  error "Cannot add period support module to Target ($status)"
}
```
application program interfaces, see APIs
architecture (Tornado)  1–14
directory tree  11–13
loader  95–103
target server  4–6
asynchronous events thread  8

functions  26–31
implementing  19–31
initializing  16, 20–26
manager  6
message logging  30–31
non-WDB  18–19, 47–87
and rebooting  28
shutting down  17
tool requests  17
WDB  17–18, 30, 32–47
Back-End Developer’s Kit  51–53
background debug mode (BDM)  76
-Bd option (target server)  30
BDM (background debug mode)  76
BKEND_INFO structure  23, 25
initializing  54, 59
BKEND_OBJ  63
bkendEventGet()  27, 28
bkendEvtPending()  27, 28
bkendInit()  39–40
bkendlib  10
bkendlog  10
bkendTgtConnect()  20
bootstraping  92
breakpoints  78
browser
display windows, creating  207–218
UNIX  208–213
Windows  214–218
network interface panels, updating  211–213

a.out  90
ACTION_TYPE  175
APIs (Tornado)  8–10
   see also individual interface entries
   graphical user interface  8
   OMF reader interface  10
   target-server back-end interface  10
   target-server core  10
   WDB agent interface  6, 10
   WTX C  8, 9, 185–193
   WTX protocol  9, 149–193
   WTX Tcl  9, 167–184
application program interfaces, see APIs
architecture (Tornado)  1–14
directory tree  11–13
loader  95–103
target server  4–6
asynchronous events thread  8

-B option (target server)  20, 23
back ends (target server)  6, 8, 15–87
   see also Back-End Developer’s Kit; non-WDB back ends; WDB back ends
   attaching  20

background debug mode (BDM)  76
-Bd option (target server)  30
BDM (background debug mode)  76
BKEND_INFO structure  23, 25
initializing  54, 59
BKEND_OBJ  63
bkendEventGet()  27, 28
bkendEvtPending()  27, 28
bkendInit()  39–40
bkendlib  10
bkendlog  10
bkendTgtConnect()  20
bootstraping  92
breakpoints  78
browser
display windows, creating  207–218
UNIX  208–213
Windows  214–218
network interface panels, updating  211–213

B
toolbar buttons, creating 211
byte order (OMF manager) 122–123

C

C language
see also WTX C API
Gopher scripts, sending 164
and WTX protocol 185

cache
managing 104, 106
sizing 106
target server memory 105–106

cacheTxtUpdate_m() 86

CALL_RETURN 158
cexit 176
checksum 34, 35, 48, 51
CLIENT data structure 39, 41, 43
clut_ops functions (RPC) 43
clntty_create() 41
code
layout
Tcl 242–245
coding conventions
see also VxWorks Programmer’s Guide
C Coding Conventions
Tcl 235–248

COFF 90
common symbols 102
context management 75–78
breakpoints 78
singlestepping 78
CONTEXT_TYPE 175
cntContextCreate_m() 86
cntContextKill_m() 86

core files 92, 102
CrossWind (debugger) 83–86
CTX_EXIT 158
CTX_START 158

D

DATA_ACCESS 158
debugging
non-WDB back ends 83–86
development tools (host), see tools, host
development
-DHOST compiler option 64
directCall_m() 86
directory tree (Tornado) 11–13
DLL, see dynamically linked libraries
documentation
coding conventions (Tcl) 235–248
new applications, creating for 229
-DPORTABLE compiler option 64
$(DYN_LK_FLAGS) compiler option 64
dynamically linked libraries (DLLs) 20
attaching back ends and target servers 20
compiling 64
entry point 39
implementing 109
writing 38–42

E

ELF 90
event handling 78–82, 157–159
asynchronous event notification 27, 38, 79
capturing information 79–81
event queue, checking the 174
event strings 157–158, 159
exceptions 81–82
generating 175–177
registering tools 174
retrieving 175–177
synchronous event notification 27, 79
and Tornado tools 173–177
event strings 157–158, 159, 174
parameters 158
EVENT_TYPE 175
eventpoints 175
eventPoll command (Tcl) 180
EXCEPTION 158
exception handling 182–184
executable files 90
F
FULL_BKD_OBJ 63
fully linked files 97, 102
funcCall() 86

G
-g compiler option 85
Gopher 2–3, 48, 51, 160–167
result tape 163
scripts
sending to the target agent 163–167
writing 200
graphical user interface (GUI) 8

H
hierarchy windows (UNIX browser) 208
HOST 45
host ID 34
-hwtx option (Tcl command) 181

I
INCLUDES 44, 63, 221
initialization
back ends 16, 20–26
non-WDB 53–55
WDB 39–42
L
launch toolbar (Tornado)
installing new applications (Windows) 227–229
launcher (Tornado)
installing new applications (UNIX) 225–226
ld () 92
lib directory 11
linkage 90
linking 91
see also loader
listEvents command (Tcl) 178, 180
LOAD_ALL_SYMBOLS 93, 101
LOAD_BAL_OPTIM 93
LOAD_BAL_OPTIM (1069) 103
LOAD_COMMON_MATCH_ALL 93, 103
LOAD_COMMON_MATCH_NONE 93, 102
LOAD_COMMON_MATCH_USER 93, 103
LOAD_CORE_FILE 93, 102
LOAD_FILE_OUTPUT 93, 103
LOAD_FULLY_LINKED 93, 102, 103
LOAD_GLOBAL_SYMBOLS 93, 101
LOAD_HIDDEN_MODULE 93, 102
LOAD_LOCAL_SYMBOLS 93, 94, 101
LOAD_NO_SYMBOLS 93, 101, 102
loadCommonManage() 138
loader thread 7
loader, object-module 5, 89–148
see also object modules
addresses, manipulating 107
architecture 95–103
C programs, calling from 94
core files 102
fully linked files 97, 102
linking 91
memory, managing 103
options 101–103
relocatable files 96, 102
shell, using from the 92
symbol types, defining 100
Tornado API 2.0
Programmer's Guide

Tcl, accessing with 93
type abstraction 107
unloading 98
and virtual memory 106
loadlib 10
loadOutputToFile() 118
loadSegmentsAllocate() 131
loadUndefSymAdd() 137
LOCAL_CFLAGS 64

testing 83–86
using 66–67
write functions
for memory 67–71
for registers 71–75

MODULE_ID 97–98

msleep command (Tcl) 174

N
naming conventions 110–112
network interfaces 198
drivers, writing 46
non-WDB back ends 18–19, 47–87
building 62–65
UNIX 62–64
Windows 64–65
checksum 51
class library 51–52
coding conventions 50
context management 75–78
designing 48–49
error logging 55
event handling 78–82
Gopher 51
implementing 47, 49–50, 52
initialization routines, writing 53–55
loading 65–66
mandatory functions, implementing 67–86
testing 83–86
message logging 55
methods
implementing 58–62
using 66–67
optional functions 86
performance 87
skeleton, creating 53–62
declaring 56–58
testing 49, 50, 53, 65–67, 83–86
writing 47–87

notify packets 38
Index

O

OBJ_LOADED 158
OBJ_UNLOADED 158
object file 90
object module format, see OMF
object module headers 90
alignment problems, resolving 105
processing 124
object module representation 97–98
object modules 90
common symbols 102
fully linked files 97, 102
hiding 102
managing 97–98
MODULE_ID 97–98
relocatable files 96, 102
SEG_INFO 97
target memory, configuring in 96–97
OMF (object module format) 5, 90
byte order 122–123
compiler signature, determining the 129
dynamically linked libraries (DLLs), using 109
events, logging 147
header processing 124
interface functions 110–122
manager 91
writing a 108–148
memory
allocating for required segments 131
managing 147
object modules, relocating 112–114, 142–145
reader interface 10
relocated modules to target memory,
downloading 145–146
segments
reading 131–136
sizing 125–127
shared library manager, installing a 109
symbol table
object-module, reading the 128
target server, processing the 136–141
warnings, logging 147

P

pbkendInfo routine 25
performance, back end 87
pTgtOps routine 24

R

raw data 90
read functions (back end)
for memory 67–71
for registers 71–75
rebooting (target) 28
registry (Tornado) 29, 154
see also wtxregd
relocatable files 90, 96, 102
symbol tables 99
relocation entries 91
relocation unit (RU) 91
interface 112–114
resource directory 12–13
RPC (Remote Procedure Calls) 18, 33
CLIENT data structure 39
core 38, 41
networks, communicating over
unsupported 43
reply errors 36–38
rpcCore library 38, 42
rpcCoreInit() 40, 41, 42

S

SEG_INFO 97, 118, 147
SEG_WRITE_PROTECTION 147
serviceCall_m() 86
SH_BKEND_OBJS 64
shared libraries, see dynamically linked libraries
(DLLs)
shell (Tornado)
extensions, installing 204–207
loader, using the 92, 94
show routines 160
standard file headings 236
state management 76–77
stepping 78
string table 91
SWAB_16 123
SWAB_16_IF 123
SWAB_32 123
SWAB_32_IF 123
SYM_ABS 94, 100
SYM_ADDED 158
SYM_ADRS 107, 137
SYM_BSS 95, 100
SYM_COMM 100
SYM_DATA 95, 100
SYM_GLOBAL 94, 100
SYM_LOCAL 94, 100
SYM_REMOVED 158
SYM_TEXT 95, 100
SYM_TYPE 100
SYM_UNDEF 100
symbol command (Tcl) 171
symbol entries 91
symbol tables
  common symbols 99, 102
defined symbols 99
external symbols 100
loading 92
managing 98–100, 101
target-server 5, 20, 21, 22, 92, 98–100, 101, 156
undefined symbols 99
symbols, see symbol tables
symFindByNameAndType() 137
symlib 10

tacc 176
target servers 3–6, 16–17
  see also back ends; symbol tables
  architecture 4–6
  -B option 20
  back-end manager 6
  -Bd option 30
cache, managing 104, 105–106
core 5, 17
event notification 27, 38
exiting 17
front ends 4
generic loader support library 107
rebooting targets 28
symbol table 5, 92, 98–100, 101, 156
target memory cache 6
target memory manager 5, 6
-V option 55
WTX protocol dispatch 4
target-server back-end interface API 10
target-server core API 10
target-server memory pool 104
  sizing 106
targetServerRestart() 28
Tcl (tool command language) 8
  see also wtxcl tool; WTX Tcl API
coding conventions 235–248
Gopher scripts, sending 164
interpreter, see wtxcl tool
loader, using the 93
unwinding 182
  and WTX protocol 150
tcl directory 13
text section protection 155
TEXT_ACCESS 158
TGT_OPS structure 23–25, 39, 42
initializing 54, 59
TGT_RESET 158
tgtCacheTextUpdate() 106, 147
tgtlib 10
tgtmem 10
tgtMemCacheSet() 104, 145
tgtMemProtect() 106
tgtMemRead() 104
tgtMemWrite() 104, 144
tgtMemWriteByte() 144
tgtMemWriteShort() 144
tgtTextProtect() 147
threads (target server) 7–8
timeout handling 181
TOOL_ATTACH 158
TOOL_DETACH 158
TOOL_MSG 158

T
tools, host development
  see also browser; CrossWind; launch toolbar;
  launcher; shell; wtxtcl tool
  adding 219–229
  building an application 220–222
  compiling an application
    UNIX 222
    Windows 222–223
  creating an application 219–224
  directories and files, naming 220
  documenting 229
  installing an application
    UNIX 225–226
    Windows 227–229
  man pages, creating (UNIX) 224
  testing an application
    UNIX 223
    Windows 224
  event strings 158
  and events 173–177
  extending 195–218
  registering for event notification 174
  requests, sending 17
Tools.h++ 63
  Tornado registry, see registry
type abstractions 107

U

UDP (User Datagram Protocol) 33
UDP/IP interpreter 45
UNKNOWN 158
unloader, object-module, see loader
UNPACK_16 128
UNPACK_32 128
unwinding 182
USER 158

V

-V option (target server) 55
VIO_WRITE 158

vioFd symbol 178
virtual I/O (VIO) 156, 177–180
virtual I/O thread 8
Visual C++ 44, 64
void * type 107
VxVMI (optional product) 106, 155

W

WDB
  see also WDB agent; WDB back ends; WDB protocol
  packet drivers 46
  reply packets 35–36
  request packets 33–35
WDB agent 2–3
  Gopher 2–3
  interface 6, 10
  reply errors 36–38
  reply packets 35–36
  sending and receiving datagrams 45–47
  target-side code 45–46
WDB back ends 17–18, 30, 32–47
  building
    UNIX 43–44
    Windows 44–45
  host-side support, creating 38–45
  initializing 39–42
  RPC client-side transport, initializing 41–42
  RPC communication, enabling 41, 43
  target link type, defining 42
  writing 32–47
WDB protocol 1, 2–3, 33–38
WDB_EVENT_GET 38
WDB_EVT_DATA 79
WDB_MEM_REGION 69
WDB_MEM_XFER 69
WDB_POOL_SIZE 106
WDB_TARGET_CONNECT 35
WDB_TARGET_PING 35
WDB_TGT_INFO 61
wdbserialInit() 40–42
WIN32 macro (Windows only) 45
Wind DeBug protocol, see WDB protocol
Wind River tool exchange protocol, see WTX protocol

wpwrLogErr()  147
wpwrLogWarn()  147
wpwrutil  10
wpwrutil library  55
write functions (back end)
  for memory  67–71
  for registers  71–75

WTX C API  8, 9, 185–193
  see also C language
  archive (libwpwr.a)  185
  initializing a session  185
  target server
    attaching to a  186
    information about, getting  186
    Tcl API, using with the  192
    using  187–192

WTX protocol  1, 2, 3–6, 9, 149–193
  creating a Tornado application  219–224
  dispatch  4
  errors, handling  182–184
  Gopher support  160–167
  language support  150
  message format  152
  requests
    events, managing  157–159
    object module, managing  155
    sessions, managing  153
    symbolic debugging facilities  154
    symbols, managing  156
    target memory access  154, 155
    target servers, attaching to  154
    tasks, managing  156
    virtual I/O  156
    timeouts, handling  181
    virtual I/O  177–180

WTX_CORE  152

WTX Tcl API  9, 167–184
  see also Tcl
  C API, using with the  192
  enumerated constant types  175
  errors, handling  182–184
  events, working with  173–177
  memory, handling  170–171
  object modules, working with  172
  target functions, calling  180
  target server
    attaching to a  168–169
    information about, getting  169
    multiple connections, working with  181
    symbol table, working with the  171–172
    tasks, working with  173
    Tcl interpreters, using with other  193
    timeouts, handling  181
    virtual I/O, working with  177–180

  wtxtcl session, starting a  168
  WTX Tcl handle  169
  WTX thread  7

WTX_CORE  152

WTX_EVENT_CTX_EXIT  176
WTX_EVENT_TEXT_ACCESS  176
WTX_GOPHER_EVAL  155
WTX_MEM_ALLOC  154, 155
WTX_MEM_FREE  154
WTX_MEM_READ  154
WTX_OBJ_MODULE_LOAD  155
wtxErrorGet()  185
wtxEventGet command (Tcl)  174, 180
wtxEventpointAdd command (Tcl)  175
wtxEventPoll command (Tcl)  174
wtxFuncCall command (Tcl)  180
wtxGopherEval()  164
wtxHandle command (Tcl)  181
wtxInfo()  186
wtxInfoQ command (Tcl)  168
wtxInfoQ()  186
wtxInitialize()  185
wtxMemAlloc command (Tcl)  170
wtxMemRead command (Tcl)  170, 171
wtxObjModuleInfo command (Tcl)  172
wtxObjModuleList command (Tcl)  172
wtxObjModuleLoad command (Tcl)  93, 172
wtxregd (WTX registry)  16, 154
  cleaning up  29
  shutting down target servers  17
wtxRegisterForEvent command (Tcl)  83, 174
wtxSymFind()  164
wtxSymListGet command (Tcl)  171, 172
wtxTcl tool 49, 50, 66–67, 83, 168
   loader, using the 93
wtxTclHandle() 193
wtxTclHandleGrant() 193
wtxTclHandleRevoke() 193
wtxTclInit() 193
wtxTest tool 47
wtxToolAttach command (Tcl) 181, 193
wtxToolAttach() 186, 193
wtxTsInfoGet command (Tcl) 169
wtxTsInfoGet() 186

X

XDR streams 34, 35
xdr_WDB_PARAM_WRAPPER 35
xdr_WDB_REPLY_WRAPPER 37