WIND RIVER

Wind River® Workbench

USER'S GUIDE

2.6

Linux Version

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

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

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

http://www.windriver.com

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

http://www.windriver.com/support

29 Sep 06
Part #: DOC-15942-ND-00

Contents

PART III: PROJECTS

PART VII: UPDATING

PART I

Introduction

1 Overview

1.1 Introduction

Welcome to the *Wind River* Workbench *User's Guide (Linux Version)*. Wind River Workbench is a development suite that provides an efficient way to develop Linux target kernels and embedded applications.

Workbench runs on various host operating systems. The screenshots in this document were taken on a host running Red Hat Enterprise Linux so they may differ slightly from what you see.

1.2 Wind River Workbench for Linux Documentation

The following documentation is provided for the Workbench version that supports Linux target operating systems:

- **Wind River Workbench User's Guide (this document)**

  This guide describes how to configure your Workbench host and a Linux target to debug applications and kernel objects on the target. It describes how to use Workbench to develop projects, manage targets, and edit, compile, and debug code. This manual is available in print from the Wind River bookstore.

- **Wind River Workbench User Interface Reference**

  This provides specific reference material for the Workbench GUI. It provides detailed information on individual menu choices, buttons, and so on, that may not be covered in the User's Guide or may only be covered relative to the part of their functionality that is being discussed. In many cases, you can access relevant parts of this document by pressing the help key as described in *1.4 Document Conventions*, p.6.

- **Wind River Scope Tools documentation**

  This is a set of documents that describe how to use the Wind River Scope tools that are provided with Workbench. The tools include a memory use analyzer, an execution profiler, and a graphical application variable monitoring tool.

- **Wind River System Viewer documentation**

  This User's Guide describes how to use System Viewer which is included with Workbench. System Viewer is a logic analyzer for visualizing and troubleshooting complex embedded software. The *Wind River System Viewer API Reference* is also included.

- **Wind River WDB Agent Porting Guide**

  This manual describes how to port a standard version of the Wind River WDB agent patch to a custom Linux 2.4.*x* version kernel. The porting guide is available from the Wind River Online Support Web site:

  www.windriver.com/support

- **Wind River Workbench Tutorials for Linux**

  Several tutorials addressing specific Linux development topics are available from the Wind River Online Support Web site.

- **Wind River Workbench Host Shell User's Guide**

The host shell is a host-resident shell that provides a command line interface for debugging targets.

- **Wind River Workbench Online Help**

Wind River Workbench provides context-sensitive Help. To access the full Help set, select **Help > Help Contents in Wind River Workbench**. To see Help information for a particular view or dialog box, press the help key when in that view or dialog box. See *1.4 Document Conventions*, p.6 for details on the help key.

1.3 Roadmap to the Workbench User's Guide (Linux Version)

This document is divided into the following parts:

Part I. Introduction (this chapter) provides an overview of cross-development concepts, and outlines how Workbench helps cross-development

Part II. Getting Started provides an introductory tutorial that takes you from starting Workbench to using it to perform common debugging activities on sample applications in user mode. Additional chapters build on this knowledge to introduce kernel-mode and dual-mode methods of project development, and introduce working with Wind River Linux Platforms.

Part III. Projects provides detailed information on how to use projects in Workbench, including pre-defined projects, user-defined projects, using the Project Navigator, and a discussion of various advanced project scenarios.

Part IV. Development describes how to use Workbench to edit source code, build projects, and parse and analyze source-code symbol information.

Part V. Target Management discusses connecting to targets, and how to create new target connections.

Part VI. Debugging provides an in-depth look at debugging operations, including launching programs, managing breakpoints, and troubleshooting.

Part VII. Updating discusses integrating plug-ins to Workbench.

Part VIII. Reference describes how to use the Wind River host shell for command-line debugging, how to update workspaces on the command line for automated builds, how to use a remote host for debugging, and provides cross-compiler information for Wind River Linux targets. This section also contains a glossary and index.

1.4 Document Conventions

In this document, placeholders for which you must substitute a value are shown in italics. Literal values are shown in bold. For example, this document uses the placeholder *installDir* to refer to the location where you have installed Workbench. By default, this is **C:\WindRiver** on Windows hosts and **$HOME/WindRiver** on Linux and Solaris hosts.

Menu choices are shown in bold, for example **File > New > Project** means to select **File**, then **New**, then **Project**. Commands that you enter on a command line are shown in bold and system output is shown in typewriter text, for example:

```
$ pwd
/home/mary/WindRiver
$
```

You will be directed to online help by suggestions to press the help key for more information. The help key on Windows hosts is **F1**. On Linux hosts, it is the combination **CTRL+F1**. On Solaris hosts, it is the **Help** key.

1.5 Eclipse Concepts

Wind River Workbench is based on the latest release of the Eclipse platform. This section provides a brief introduction to the features of Workbench that are common to all Eclipse-based systems.

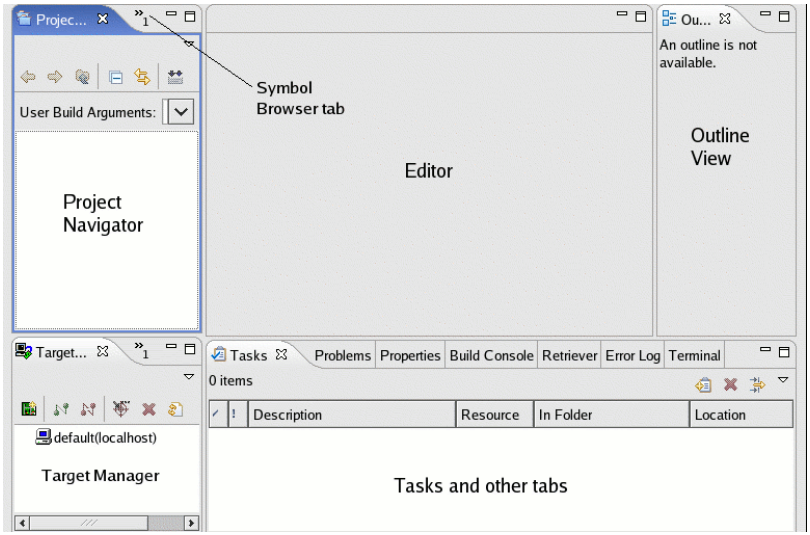
For a more detailed introduction, use the online Help in Workbench (select **Help > Help Contents**, select **Wind River Partners Documentation** and expand the **Eclipse Workbench User Guide** entry) or go to **www.eclipse.org/documentation/main.html** and navigate to the Eclipse *Workbench Users Guide*.

Eclipse-based systems are described with several terms: *window*, *view*, *tabbed notebook*, *editor*, and *perspective*. This section outlines these terms and explains how to use these features.

Windows

The term *window* is used only for the overall outer frame shown in Figure 1-1. Use **Window > New** to create a new outer frame as part of the same Workbench session; then use the new window to open additional views within the same perspective (see *Perspectives*, p.9).

Figure 1-1 **Workbench Window**



Views

The term *view* refers to the individual panes within a window; in Figure 1-1 these include the **Project Navigator** view on the top-left side of the screen, the **Outline** view on the top-right, the **Target Manager** view on the bottom-left, and the stacked view on the bottom-right with the title **Tasks**.

There are two rules to consider when using views:

1. Only one view (or Editor; see *Editors*, p.9) can be active at a time. The title bar of the active view is highlighted.
2. Only one instance of a type of view can be present in a perspective at a time.

NOTE: Multiple editors can be present to view multiple source files.

Many views include a menu that is accessed by clicking the down arrow to the right of the title bar. This menu typically contains items that apply to the entire contents of the view rather than a selected item within the view.

To open a view and add it to the existing perspective (see *Perspectives*, p.9), select **Window > Show View**. Select the desired view from the list, or select **Other** to display expandable lists with more choices. The view is added at its default location in the window, and you can move it if desired—see *Moving and Maximizing Views*, p.8.

Tabbed Notebooks

Several views can be stacked together in a *tabbed notebook* (often the result of opening additional views). For example, the **Tasks** view at the bottom-right of Figure 1-1 has six tabs along the top. The title of the selected tab shows in the title bar of the view; in this case, **Tasks (0 items)**. Click a tab to display that view.

Moving and Maximizing Views

Move a view by clicking either its title bar or its tab in a stacked notebook, and dragging it to a new location.

There are several ways to relocate a view:

- Drag the view to the edge of another view and drop it. The area is then split, and both views are tiled in the same area. The cursor changes to an appropriate directional arrow as you approach the edge of a view.

- Drag the view to the title bar of an existing view and drop it. The view will be added to a stacked notebook with tabs. When you drag the view to stack it, the cursor changes to an icon of a set of stacked folders.
- If you drag a view over a tab in an existing view, the view will be stacked in that notebook with its tab at the left of the existing view. You can also drag an existing tab to the right of another tab to arrange tabs to your liking.

To quickly maximize a view to fill the entire perspective area, double-click its title bar. Double-click the title bar again to restore it.

Editors

An *Editor* is a special type of view used to edit files. You can associate different Editors with different types of files such as C, C++, Ada, Assembler, and Make files. When you open a file, the associated Editor opens in the perspective's Editor area.

Any number of Editors can be open at once, but only one can be active at a time. By default, Editors are stacked in the Editor area, but you can tile them in order to view source files simultaneously.

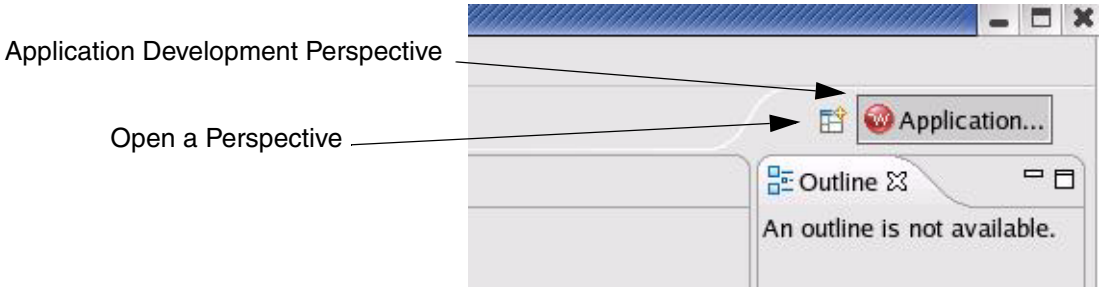
Tabs in the Editor area indicate the names of files that are currently open for editing. An asterisk (*) indicates that an Editor contains unsaved changes.

Perspectives

A *perspective* is the layout of the views and Editors in a window. A single window can maintain several perspectives, but shows only one perspective at a time.

To open new perspectives and switch between them, use the buttons in the shortcut bar along the top right edge of the Workbench window. When you start Workbench for the first time, two buttons display as shown in Figure 1-2.

Figure 1-2 **Perspectives Shortcut Bar**



Perspectives are a convenience for accomplishing specific tasks. For example, the Application Development perspective (Figure 1-1), is designed for creating projects, browsing files, and editing and building source code.

Click **Window > Open Perspective > Device Debug**. The **Device Debug** perspective appears, containing the **Debug** and **Breakpoints** views, and a tabbed notebook with the **Local Variables**, **Watch**, and **Register** views, which are useful when running and debugging a program. These views replace the **Outline** view of the **Application Development** perspective.

You can also create your own perspectives to suit your development needs. Follow these steps to create a customized perspective:

1. Arrange the views in the window as desired by opening any required views and moving them to an appropriate location.
2. Select **Window > Save Perspective As**, enter a name for your custom perspective.
3. Click **OK**.

Your customized perspective is saved in your workspace. The button in the shortcut bar displays the new name when you hover the cursor over it.

Switch back to the **Application Development** perspective by selecting it on the top right side and resetting it with **Window > Reset Perspective**. It should once again appear as shown in Figure 1-1.

Workspaces

Workbench uses a *workspace* to hold different types of information, including:

- Information about a set of projects, including project names, lists of files for each project, and build specifications.
- Information about the current session, including the types and positions of your windows when you last exited Workbench, current projects, and installed breakpoints.

When you start Workbench, you are asked to specify a workspace to store your project information. You must have write access to the workspace you are going to use.

The default location proposed by Wind River Workbench may be sufficient, but there are several situations in which specifying a different workspace is useful:

- To separate your workspace from the installation directory and from that of others (recommended).
- Your target and host may share a root file system. In those cases, it may be useful to place your workspace inside the root file system you export to the target. (See *3.8 Configuring a Target Connection*, p.35 for more information.)
- To keep different sets of projects separate: each workspace has its own set of projects.
- To run two or more instances of Workbench (each must have its own workspace and its own target root file system—see *3.8 Configuring a Target Connection*, p.35).

1.6 Understanding Cross-Development Concepts

Cross-development is the process of writing code on one system, known as a host, that will run on another system, known as a target.

Cross-development allows you to write code on a system that is readily available and familiar (such as a PC running Linux or Windows) and produce applications that run on hardware that you would have no other convenient way of programming, such as a chip destined for a mobile phone.

Hardware in a Cross-Development Environment

A typical host is equipped with large amounts of RAM and disk space, backup media, printers, and other peripherals. In contrast, a typical target has only the resources required by the real-time application and perhaps some small amount of additional resources for testing and debugging.

Working on the Host

You use the host just as you would if you were writing code to run on the host itself. You can manage project files to edit, compile, link, and store multiple version of your real-time application code. You can also configure the operating system that will run on the target.

Connecting the Target to the Host

A number of alternatives exist for connecting the target system to the host, but usually the connection consists of an Ethernet or serial link, or both.

Advantages of Using Wind River Workbench for Linux Development

Workbench ensures the smallest possible difference between the performance of the target you use during development and the performance of the target after deployment of the finished product by keeping development tools on the host.

With Workbench your application does not need to be fully linked. Partially completed modules can be downloaded for incremental testing and modules do not need to be linked with the run-time system, or even with each other. The host-resident shell and debugger can be used interactively to invoke and test either individual application routines or complete tasks.

Workbench loads the relocatable object modules directly and maintains a complete host-resident symbol table for the target. This symbol table is incremental: the target server incorporates symbols as it downloads each object module. You can examine variables, call routines, spawn tasks, disassemble code in memory, set breakpoints, trace subroutine calls, and so on, all using the original symbol names.

Workbench shortens the cycle between developing an idea and implementing it by allowing you to quickly download your incremental run-time code and dynamically link it with the operating system. Your application is available for symbolic interaction and testing with minimal delay.

The Workbench debugger allows you to view and debug applications in the original source code. Setting breakpoints, single-stepping, examining structures, and so on, are all done at the source level, using a convenient graphical interface.

1.7 Workbench for Linux Development

Workbench is an integrated development environment for creating device software to run on embedded Linux systems. Workbench is optimized for both small programs and very large ones with thousands of files and millions of lines of code. It includes a full project facility, advanced source-code analysis, simultaneous management of multiple targets, and a debugger with capabilities for managing multiple processes or threads on a single target or on multiple targets.

PART II

Getting Started

2 Introduction

2.1 Building Linux Projects

You can use Workbench to create projects for your generic Linux kernel and applications, and you can also use Workbench to create platform (kernel and file system) and application projects for Wind River Linux Platforms.

The choices you see in some menus may differ from those in the examples in this document depending on your Workbench licensing.

Types of Projects

Workbench has predefined project types for building embedded kernels and applications, and Wind River Linux platforms and applications. Additional project types are user-defined applications and native applications. Some of the major project types available to you are described in this section.

User Defined Projects

User defined projects are typically ones where the source code, including makefile, is provided by you or a third party. You import or reference the source and use the makefile provided to build it. Workbench provides the Project Navigator as a convenient way to access your build environment, and the Build Console will display build output.

Embedded Linux Kernel Projects

This is basically a user-defined build. Typically you would acquire your kernel source from someplace and it includes a makefile that the project uses. You then use the Workbench environment to facilitate your builds.

Embedded Linux Application Projects

The embedded application project is a managed build. Workbench provides default build settings that you can modify as you see fit. It also provides the makefiles and controls all phases of the build.

Wind River Linux Application Projects

These are similar to embedded application projects, except in this case the build specs provided are based on the tested and supported targets that are supported by Wind River Linux. Appropriate libraries, cross-development tools and so on are also provided, so much of the traditional gathering and debugging of a working environment is eliminated.

Wind River Linux Platform Projects

Wind River Linux Platform projects are designed for building the entire target platform, including the kernel and file system. They provide special tools to make platform configuration, modification, and maintenance easier. Root file systems are easily built using default settings or you can configure the file system through a convenient GUI configuration tool that the project provides. Default kernels are provided for supported targets, and kernel configuration is also easily performed with a provided GUI configuration tool.

Native Application Projects

Native application projects are managed projects that are developed and deployed on the host, so no cross-development is required. The build specs assume local development tools. Additional build specs support team development for when a team works with a combination of Windows, Linux, and Solaris host environments.

2.2 Linux Version Debugging Modes

The chapters in the Getting Started section describe how to use Workbench in different modes.

Depending on your needs and the version of the Linux kernel you are running on your target, you will use one or more of the following debug modes:

- User mode—This debug mode allows you to perform source code debugging of user mode applications, including multi-process and multi-thread debugging. In user mode, a usermode agent on the target communicates with Workbench on the host, where you can edit your source code, step through the application, set breakpoints, and so on. User mode debugging is available for targets with Linux version 2.6 kernels and also Linux versions 2.4.20 and later. Refer to *3. Developing Applications (User Mode)* for a tutorial on the use of Workbench in user mode.
- Kernel mode—This debug mode allows for source code debugging of Linux kernels version 2.6.10 and later. The kernel must be patched for the Kernel GNU Debugger (KGDB), which communicates with the standard GNU debugger (GDB) on the host. (KGDB is enabled by default for Wind River Linux kernels and only needs to be disabled when going to production.) Kernel mode source code debugging allows you to debug the kernel using suspends, breakpoints, and so on, as you would with user mode debugging. Kernel mode debugging is described in *5. Kernel Debugging (Kernel Mode)*.

NOTE: If user mode and KGDB debugging connections are used in parallel for the same target, the user mode connection stalls when a KGDB breakpoint is reached.

- Dual Mode—In dual mode, you can toggle between user mode and system mode. Dual mode is supported for Linux 2.4 version kernels. It is not supported on the newer 2.6 kernel versions. Dual mode requires that you build the appropriate Linux kernel with the supplied Wind River WDB agent. Dual mode operations, including building the kernel and downloading it to the target, are described in *F. Configuring Linux 2.4 Targets (Dual Mode)*.

3

Developing Applications (User Mode)

3.1 Introduction

This chapter is designed as a tutorial on application development with Workbench. It takes you from an introduction to using Wind River Workbench through running and debugging a cross-compiled application on a target system. It begins with the classic Hello World program to demonstrate default Workbench perspectives and views that you can use to compile, run, and debug a program. It also shows how to use a native-debug configuration in which your Workbench host serves as your target. The first example requires only that Workbench be installed on a Linux host computer.

Further examples expand on this functionality by building a project in cross-debug mode, in which the target has an architecture different from the host and you use the proper cross-compiler and related development tools for that architecture. As you continue with the tutorial, you will cross-compile an application for the target, connect to the target, and run and debug the application on the target. While reading the tutorial is useful, you will learn more quickly if you work through the tutorial on your computer.

3.2 Starting Workbench

The executable to start Workbench is located in *installDir*, the location where Workbench is installed.

NOTE: Before starting Workbench, make sure your path environment variable is set to include the path to your compiler.

The command to start Workbench from *installDir* is:

```
$ ./startWorkbench.sh
```

This is the basic Workbench startup command. There are also a number of arguments you can apply at startup as described in the section *Running Eclipse* in *Eclipse Workbench User Guide:Tasks* in the online Help.

NOTE: On a Windows host, from the **Start** menu select **Programs > Wind River > Workbench 2.6 > Wind River Workbench 2.6**. If you are using a Windows host, you can perform this part of the tutorial up to the point where you are asked to connect to the localhost target, at which point you should skip ahead to *3.4 Creating a Project*, p.26.

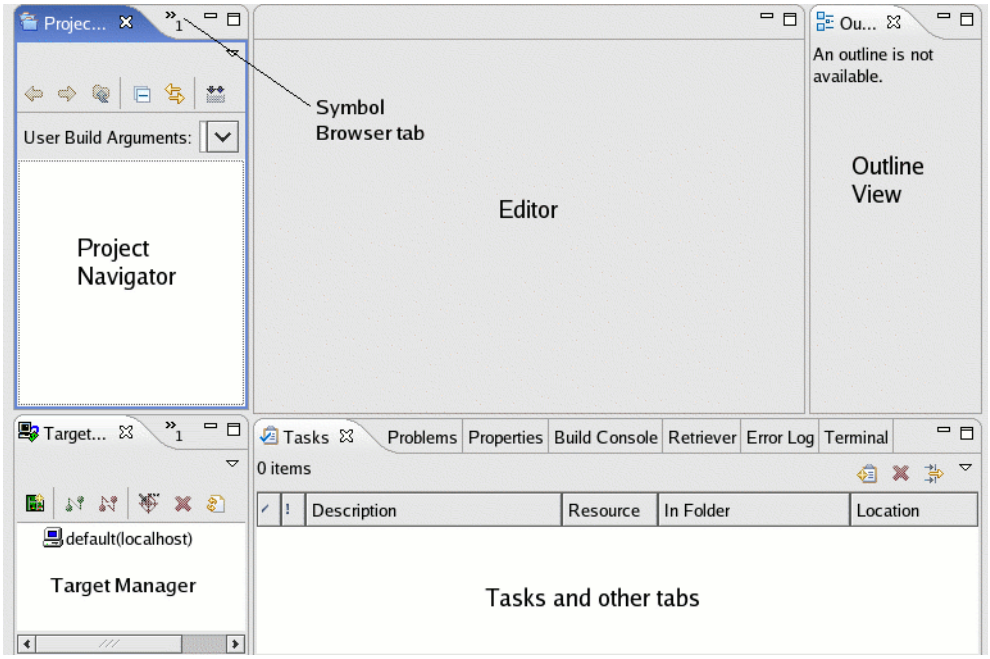
When Workbench starts, the initial **Welcome** screen appears with several options. For now, go ahead and begin using Workbench, and note that you can come back to the initial **Welcome** screen at anytime by selecting **Help > Welcome** from the Workbench window.

Click **Workbench**.

The **Application Development** perspective is displayed. XREF shows the default arrangement of the views in the Application Development perspective

NOTE: Workbench preserves its configuration when you close it, so that the next time you start Workbench, it resumes where you left off in your development.To restore its default settings, select **Window > Reset Perspective**.

Figure 3-1 **Default Application Development Perspective Views**



3.3 Using Workbench

The following example uses the native development environment in which development tools such as **gcc** are on your host computer and in your **$PATH** setting. You do not need cross-development tools for native development.

Workbench starts by displaying the Application Development perspective with several different views. (If you have not already started Workbench, follow the steps in *3.2 Starting Workbench*, p.22, and then continue with the procedure in this section.)

1. In the Project Navigator on the left, right-click and select **New > Example**.

NOTE: The particular project choices you see may differ depending on your installation.

2. In the **New Example** dialog box that opens, select **Native Sample Project** and click **Next**.
3. In the **New Project Sample** dialog box that opens, select **The Hello World Demonstration Program** and click **Finish**.
4. The **hello_world** sample project is created. Expand the *project folder* in the Project Navigator to see the sample contents.

   The sample project contains all the necessary files to build the program.
5. Right-click the **hello_world** project folder and select **Build Project**. The executable is built. The **Build Console** at the bottom of the Workbench window will display build output. If you expand the *build target* (blue container) in the Project Navigator you can see the resulting executable **hello_world**. This is the *build output* or *build result*.
6. Your compiled application is now ready to be run on your target, but you must first make a connection to the target. In this case, you will use your local Linux host as the target. (If you are not using a Linux host, skip ahead to *3.4 Creating a Project*, p.26.) In the Target Manager on the bottom left, select the **linux_native_localhost** entry and click the green connection button.
7. A default output window appears.
8. In the Target Manager, right-click **Processes** and select **Debug > Debug Process on Target**.
9. In the **Main** tab of the dialog, find **Exec Path on Target** and browse to the location of the Hello World program (**hello_world**) to select it. It will be under **workspace/hello_linux/Linux-gnu-native_DEBUG** or a similar location depending on your configuration. Select it and click **OK**.
10. Click **Debug**. The process executes until it comes to the **main()** routine in the program.

Workbench is now in the Device Debug perspective showing the Debug view in the upper right with execution stopped at the **main** routine of **hello.c**. Also, the source file is open in the Editor, and other views appropriate for typical debugging operations are open.

In the Debug view, note the process ID following the entry for **hello_world**.To display all the processes running on the target, click the arrow next to **Processes** in the Target Manager. Scroll down if necessary until you find the process ID of **hello_world** and expand it. It will show the process as stopped.

11. Move your mouse cursor over the buttons at the top of the Debug view to see the debug functions that you can perform.

NOTE: You can use function keys as shortcuts to perform many of the operations performed with the buttons. For example, **F6** performs the **Step Over** function and **F8** is **Resume**. Click the **Run** menu item to see function keys for other common activities.

12. Click the **Step Over** button (or press **F6**). The program advances one line in the **Editor**, and **Hello World!** displays in the Console view.
13. You can continue stepping through the program, or click the **Resume** button (**F8**) to complete it. When the program has completed, the Debug view provides its exit status.
14. To remove old information from the Debug view, click the **Remove All Terminated Launches** button.

3.4 Creating a Project

The Project Navigator lets you visually organize projects to reflect their inner dependencies, and therefore the order in which they are compiled and linked.

In this section, you will create a project folder and import existing source files into it.

The project you prepare in this section uses cross-debug mode, in which the output runs on an architecture different from your development architecture. The example supplied uses a PowerPC target and associated cross-development tools.

The rest of this chapter uses the **ball** sample program, written in C. This program implements a set of ASCII-character balls bouncing in a two-dimensional grid. As the balls bounce, they collide with each other and with the walls. You see the balls move by setting a breakpoint with the property **continue-on-break** at the outer move loop, and watch a global **grid** array variable in the **Memory** window.

Creating a Wind River Linux Application Project

Create a new application project in your workspace as follows:

1. Click the **Wind River Workbench** project icon in the upper-left corner of the Workbench window. You can find the names of the icons by letting your mouse cursor hover over them. (The icon looks like a Workbench window with a "W" on it.)
2. Select your target operating system (**Wind River Linux Platform**) in the new project dialog and click **Next**.
3. For **Build Type**, select **C Application** and click **Next**.
4. Enter a name for the project, such as **ball**, and click **Finish**.

The new ball project appears in the Project Navigator. Click the arrow to the left of the **ball** folder to expand it and see the files created for the project. Note that the default build target name (in parenthesis after **Build Targets)** indicates that **ball.out** will be created in a subdirectory of **ball** with that name. The build target name is based on the default build spec, which you will choose later in this tutorial—don't worry if it does not match your target architecture at this point.

Importing Existing Source Files

In the following procedure, you import the **ball** program example source files into your new project.

1. To import the existing source files of the ball program, right-click on the **ball** project you created in the previous procedure and select **Import**.
2. In the **Import** dialog box, select **File System** and click **Next**.
3. For the **From Directory** field, click **Browse**. Browse to *installDir***/workbench-2.6/samples/ball** and click **OK**.

4. In the **Import** dialog box, select the check box next to the **ball** folder.

   The four files in that folder are now pre-selected in the right pane, and the **Into folder** field includes **ball**.

5. Click **Finish**. The four new files are now included in the project in the Project Navigator.

3.5 Using the Editor

In this section you exercise various aspects of the built-in Editor before moving on to the next section where you will build the project you created in the previous section.

NOTE: If you prefer, you can specify that the Workbench editor emulate the **vi** or **emacs** editors by clicking the appropriate icon on the title bar. Refer to additional editor preference s in **Window > Preferences > General > Editor** and additional online information at **http://help.eclipse.org**.

Opening a Source File

The default **Application Development** perspective includes an area for the Editor in the center and an Outline view on the right.

Expand the **ball** folder and double-click the **main.c** file. The file displays in the Editor.

The Editor uses preference-based color syntax highlighting.

The code displays using a fixed font. Use the Workbench preferences settings to modify the font.

1. Select **Window > Preferences > General > Appearance > Colors and Fonts**.
2. Expand the **Basic** folder in the **Colors and Fonts** pane and select **Text Font**.

3. Click **Change**. Change the font to:

   - Family: Courier 10 Pitch
   - Style: Regular
   - Size: 10

   Click **OK**.

4. Move the **Preferences** dialog box out of the way to uncover the **main.c** Editor window if necessary, click **Apply** to see the changes without closing the dialog box.

5. Explore and experiment with any other preferences. Then click **Restore Defaults** if you prefer the default settings.

6. Click **OK**.

3.5.1 Navigating in Source

Several mechanisms make it easy to navigate in the source code.

Using the Outline View

In the Outline view, click any name in the list to immediately focus the Editor on the declaration of that name. Hover over the buttons on the Outline view toolbar to see how to filter the list of names in the view.

The Outline view is limited to the names declared in the file that is open in the Editor. To find the declaration of any symbol, right-click the symbol, then click **Navigate > Show Declaration**.

For example:

1. In the Outline view, click **main( )**.
2. In the Editor, click the call to **gridInit( )** to highlight that line.
3. Right-click **gridInit( )** and select **Show Declaration.**

   The **grid.c** file opens in the Editor, positioned at the declaration of **gridInit( )**.

Finding a Symbol

To open a more advanced symbol search dialog box:

1. Select **Navigate > Open Symbol**.

   The **Open Symbol** dialog box appears.

2. Enter **grid*Ball**.

   As you enter a **Pattern** for the symbol, including wild cards, Workbench lists all matching symbols. All symbols that match **grid*Ball** are displayed.

3. Click **Cancel**.

Finding a String

To find and optionally replace any string in the active Editor view, use **Edit > Find/Replace** (**CTRL+F**). Use **CTRL+K** to **Find Next** and **CTRL+SHIFT+K** to **Find Previous**. See the **Edit** menu for other choices.

The **Search** menu provides a **grep**-like search for strings in any file in the workspace or in any location.

3.5.2 Using Code Completion

Code completion suggests methods, properties, and events as you enter code.

To use code completion, you can right-click in the Editor and select **Source > Content Assist** or use **CTRL+SPACE** as a keyboard shortcut. A popup list displays valid choices based on the letters you have typed so far.

For example, with the **main.c** file in the Editor:

1. Position your cursor inside **main( )** to the right of the first **{** character and press **ENTER**.
2. Type the first two characters of **grid** and then invoke code completion by pressing **CTRL+SPACE**.

   A dialog box appears with suggestions.

   As you continue to type, your choices narrow.

3. Select **gridAddBall(BALL*, POINT): void** and press **ENTER** to add the routine to the Editor.

3.5.3 Getting Parameter Hints

Parameter hints describe what data types a routine accepts.

When you add a function using code completion, it appears with parameter hints.

You can request parameter hints as you enter code by right-clicking in the Editor and selecting **Source > Parameter Hints**. or use the keyboard shortcut **CTRL+SHIFT+SPACE**.

3.5.4 Using Bracket Matching

To use bracket matching, position the cursor before an opening bracket or after a closing bracket.

A rectangle encloses the corresponding bracket to make it easy to locate.

To jump between opening and closing brackets, press **CTRL+SHIFT+P**.

Bracket matching operates on the following characters:

- **( )**,
- **[ ]**,
- **{ }**,
- **" "**,
- **/* */**,
- **< >** *(C/C++ only)*

NOTE: Before you proceed with the tutorial, undo the changes you made to **main.c** by selecting **File > Revert**.

3.5.5 Using Bookmarks to Mark Errors

This procedure introduces an error in the source code and bookmarks it.

Introduce an Error

1. Click **main( )** in the Outline view to go to the **main** routine in the Editor.
2. Move down a few lines to find the call to **gridInit( )**.
3. Delete the semicolon after the call to **gridInit( )**.

Creating the Bookmark

1. Right-click the gutter of the Editor next to that line (the *gutter* is the shaded strip at the left edge of the Editor view) then select **Add Bookmark**.
2. In the **Add Bookmark** dialog box, enter **Correct Error** then click **OK**.

   A small bookmark icon appears in the gutter.
3. To save the file with the error, click the **Save** button on the main toolbar.
4. Close all open files by clicking the **X** on the tab of their Editor view.

Locating and Viewing the Bookmark

1. To open the **Bookmarks** tab, select **Window > Show View > Bookmarks**.

   The Bookmarks view shows all bookmarks in the project. Because there is only one bookmark in this project, only the **Correct Error** bookmark appears in the list.
2. Double-click the entry for **Correct Error**.

   The **main.c** file opens in the Editor with the bookmark location highlighted.
3. Close **main.c** without making any changes (leave the error).

3.6 Configuring Project Properties

Every project has a set of build properties associated with it. You can use these build properties to change target specifications, relocate build output, see and modify the actual build command line, and more.

1. In the Project Navigator select the ball project, right-click, and select **Properties** (at the bottom of the context menu).
2. In the **Properties** dialog, select **Build Properties** on the left.
3. Select the **Build Support and Specs** tab on the right. Note the list of available and enabled build specs, and the default and active build specs on the bottom.
4. Click **Disable All** and then locate the build spec for your target and select it. For example, select the check box for the **82xx** choice that matches your target.
5. Select the **Build Tools** tab and select **C-Linker** from the **Build tool** menu.

6. Click **OK** to save your modified build properties.

Note that the build target name after **Build Targets** in the Project Navigator is now set to the name of your build target.

3.7 Building the Project

If you completed *Creating a Project*, p.26, the ball project folder now includes all the files necessary to build and debug the program. In the example in *3.3 Using Workbench*, p.24, you built the Hello World program at this point by right-clicking on the project folder and selecting **Project Build**. That example built the program to run on the local host because the target connection was the localhost connection.

In this example, you will build the program with cross-development tools that generate a build target for your target architecture. The example uses a PPC target, but you can substitute your target architecture if it is different. Depending on your Wind River Linux installation, the *installDir*/**gnu** directory contains appropriate tools for your licensed architecture, including the C language cross-compiler used in this example.

3.7.1 Building ball With an Error

In *3.5.5 Using Bookmarks to Mark Errors*, p.31 you introduced an error in the ball program source code that you will fix in the following procedure that demonstrates how to find build errors using Workbench.

1. In the **Project Navigator**, right-click on the ball project and select **Build Project** from the context menu to build the project. Click **Continue** if a prompt appears concerning the include search path.

   Build output displays in the **Build Console** tab and entries also appear in the **Problems** tab. Click these tabs to move back and forth between their contents or rearrange your window to view them both simultaneously.

   Because there is an error n the **main.c** file, errors are encountered during the build process. Notice that Workbench enters a task in the **Problems** list showing the type of error, the file the error is in, the location of the file, and the location of the error. It also shows warnings that occur in other locations in the code because of the error.

2. Double-click the error in either the Build Console view or the Problems view.

   The Editor focuses on the erroneous line, with an identical **X** marking the position of the error in the file.

3. Replace the semicolon (**;**) character you deleted in earlier steps.
4. Right-click the bookmark icon in the gutter and select **Remove Bookmark**.
5. Save and close the file.

3.7.2 Displaying File History

At this point, several changes have been made to the **main.c** file. Workbench tracks all changes that are made to any files in the project.

To display the change history on the **main.c** file, right-click the file in the project tree and select **Compare With > Local History**.

The **Compare with Local History** dialog box appears. The upper area of the dialog box displays a list of the dates and times when the file was changed. When you select one of the list entries in the upper part, the lower part displays the file as of that time on the left, and the version before then on the right (that is, the changes associated with that save). Note the changes you have just made.

Click **OK** to close the dialog box.

3.7.3 Rebuilding the Project

Right-click the ball folder at the top of the project tree, select **Rebuild Project**.

Now the project builds without errors.

3.8 Configuring a Target Connection

When running applications on your target, the target must be configured with the appropriate Linux kernel and root file system.

Typically, a boot loader on your target will download a kernel from your host, and the target will also NFS-mount a root file system from the host. Refer to your target's boot documentation for details on how to supply boot parameters. (One example is given in *F. Configuring Linux 2.4 Targets (Dual Mode)*.) This chapter assumes the target board has already been configured, and shows you how to run and debug applications on the target using Workbench on your development host.

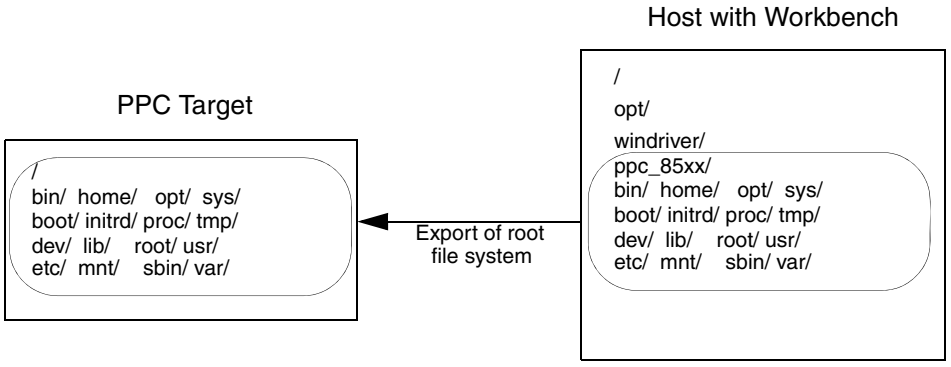
The output of your build must appear on the target so that it can be launched from there for debugging. One way to make your build output appear on the target is to have your Workbench workspace located in a directory that is shared between the target and host. This example assumes your workspace is not in a location shared with the target, so you will redirect the build output to a shared location—the root directory that the target mounts from the host. This allows you to have your workspace anywhere on the host, for example in your home directory.

NOTE: To redirect build products to the exported root file system, you must have write permission in that directory on the host.

3.8.1 Configure NFS

Configure NFS on your host to export a root file system to the target. In the example given, **/opt/windriver/ppc_85xx** is exported to the target named **ppc_target** where it is mounted as the root file system (**/**). Figure 3-2 illustrates the exported root file system used in this tutorial.

Figure 3-2 **Exported Root File System Example**



NOTE: Alternatively, if you are using a target that has its own file system, you can NFS-mount that file system on your host, for example:

```
# mount -t nfs targetbox:/ /opt/windriver/ppc_85xx
```

Redirect Build Output to the Target Root

You must configure your project build properties to export build results to the target filesystem as follows:

1. Right-click on the **ball** folder in the Project Navigator and select **Properties**.
2. Select **Build Properties** on the left and then select the **Build Paths** tab. In **Redirection root directory** browse to the directory you are exporting as the target's root or enter it manually. This is where your cross-compiled build output will go.
3. Click **OK**.

3.8.2 Run the Usermode Agent on the Target

When you created the localhost connection to run the Hello World program in native-debug mode, Workbench started a program called **usermode-agent** on your host. This program is used in Workbench communications between target and host. In cross-debug mode, you have to run the agent on your target before Workbench can connect to it.

For Wind River Linux targets, the **usermode-agent** program is located in **/usr/bin**.

On the target, change directory to the location of the program and execute it:

```
$ ./usermode-agent &
```

If your target is using the uClibc library, add the **-no-threads** argument:

```
$ ./usermode-agent -no-threads &
```

NOTE: You can run the agent as a user or as root, but note that the processes you launch from Workbench will run as the user that started the agent on the target.

Keep this window to your target open to observe possible errors when you run the program. You are now ready to connect Workbench on your host to the agent on the target.

NOTE: Wind River can only qualify **usermode-agent** binaries for Wind River Linux. If you are using another version of Linux you may find an appropriate **usermode-agent** binary in *installDir*/**linux-2.x/usermode-agent**. If you find that the supplied **usermode-agent** does not work properly, you can re-compile **usermode-agent** with your cross-development tools using the source provided. Refer to *Building the Usermode Agent*, p.45 for an example of building **usermode-agent** in Workbench.

3.9 Connecting to the Target

There are several ways to connect to your target and run the ball program. In the Hello World example, you right-clicked in the Target Manager and selected **Debug > Debug Process on Target**. You can also select **Run > Debug** from the main menu to get you to the same place. This is the *launch configuration* dialog for your project. Modify the default launch configuration to the way you want it and you can then use it at any time to connect to your target and launch the program in one step.

1. Select **Run > Debug** to begin to create a new debug launch configuration.
2. Under **Configurations**, select **Process on Target**.
3. Click the **New launch configuration** icon. An area for creating a launch configuration is displayed in the right part of the dialog box.

4. Beside **Name**, accept the supplied name or enter a new name for the launch configuration, such as **ball**.

1. Click **Add** beside the **Connection to use** entry.

   The **New Connection** dialog box appears.

2. Select **Wind River Linux User Mode Target Server Connection** and click **Next**.

3. Select your target operating system and click **Next**.

4. In the User Mode Agent Options dialog box browse to the location of usermode-agent in the target's root file system as seen from the host. This is typically **/usr/bin/usermode-agent** on the target so it will be something like **/opt/windriver/ppc-85xx/usr/bin/usermode-agent** here, depending on where you have located your target operating system.

   Click **Next**.

5. The **Target Server Connection for Linux User Mode** dialog box appears.

   In the **Target Name / IP Address** box, enter the hostname or IP address of your target. As you type, the information will be entered in the **Command Line** section at the bottom of the dialog box.

   In the **Target Filesystem** section, **Root Filesystem** box, enter the path to the directory that the target mounts. In this example, the target is mounting **/opt/windriver/ppc_85xx** as its root file system**.**

   Click **Next**.

6. The **Object Path Mappings** dialog box shows where Workbench looks on the host to find objects that are on the target. Note that the target's root (**/**) has been mapped to the exported file system (**/opt/windriver/ppc_85xx**). Click **Next**.

7. Click **Next** to accept the default auto-refresh settings, and click **Next** to accept the default breakpoint options.

8. Click **Finish** in the summary window.

9. The connection you created now appears in the **Connection to use** entry and the connection is attempted. If successful the **Target Manager** in the lower-left of your Workbench window shows the connection labeled as **[connected]**, and you can click the arrow next to it to expand it and see entries for the target architecture and processes.

10. Click **Browse** for the **Exec Path on Target** entry. Find the executable **ball.out** program in the root redirection directory (in this case, under **/ball**).

11. Click **Apply** and **Close**.

    Note that with Workbench connected to the agent on the target, and your build results redirected to the target, you are now ready to run and debug applications on the target from Workbench.

3.10 Running and Debugging on the Target

If your Debug view is not still open from the previous procedure, select **Run > Debug** to open it and select your launch on the left to display the launch configuration.

1. Click **Debug** to connect to your target and launch the program.

   Several events occur:

   - Workbench builds the program.
   - Workbench connects to the target if not already connected.
   - Workbench switches to the Device Debug perspective.
   - Workbench downloads the program and runs it on the target (in this case as an ordinary Linux application program).
   - The program executes up to **main( )** and breaks.
   - Workbench opens the source file, highlighting the location of the program counter where execution has stopped in the **main** routine.

NOTE: Instead of using the **Run > Debug > Debug** sequence of menu selections, you could use the shortcut function key **F11** or click the debug icon (looks like a bug) in the main menu bar.

2. You can see that the program has stopped at a breakpoint by looking at the Debug view in the upper-right corner of your window.
3. When you connect to your target and launch the program, the connection displays in the **Target Manager** with **[connected]** beside it. The type of target and version of Linux that is running is displayed under the connection definition. You can expand this node to display any tasks that are running.

Scroll down and find the stopped **ball.out** process. It will have the same process ID shown with the stopped process in the Debug view.

4. The Workbench window now displays the Device Debug perspective with the source file open and the line highlighted corresponding to the point at which the program stopped.

NOTE: You can connect to the target from the Target Manager or the Project Navigator instead of using a launch configuration. Connecting this way automatically creates a launch configuration for you, which can be accessed by right-clicking in the Debug view.

Now that you have created a launch configuration, you do not have to go through these steps each time you want to test your application—the connect, run, and attach debugger steps are all pre-configured. You can use a launch configuration in either run or debug mode. You can do any of the following:

- Click the **Run** button in the toolbar to run the most recently launched application, or press **CTRL F11**.
- Click the **Debug** button in the toolbar to debug the most recently launched application or press **F11**.
- Click the arrow next to the **Run** button in the toolbar to select an application to run from among your most recent launches.
- Click the arrow next to the **Debug** button in the toolbar to select an application to debug from among your most recent launches.

Refer to *16. Launching Programs* for more information on the run and debug modes of launch configuration.

The Device Debug perspective which opened when you launched the program is discussed next.

3.10.1 Using the Device Debug Perspective

The default Device Debug perspective includes views suited to executing and debugging a program, including the Project Navigator and Target Manager as before, but replacing the Outline view with the following elements:

- The Debug view
- The Breakpoints view
- A tabbed notebook that includes the Local Variables, Watch, Registers, and Console views

Notice that the new perspective is present in the shortcut bar at the top right edge of the Workbench window, where a button for **Device Debug perspective** has been added to the buttons for **Open a Perspective** and the **Application Development perspective**. Switch between the two perspectives by clicking their buttons.

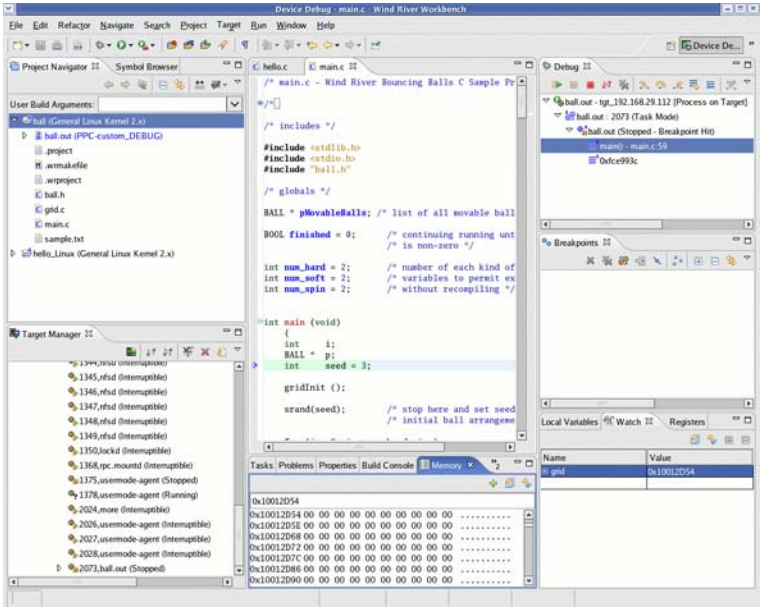
As with the Application Development perspective, the views in the Device Debug perspective can be repositioned to suit your needs.

The action of the ball sample program is viewed in the **Memory window.**

To set up the Device Debug perspective to match this tutorial, do the following:

1. Select **Window > Show View > Memory**.
2. Click the **Memory** tab at the bottom right, then click the title bar and drag it over the Build Console view (wait for an icon of a set of stacked folders to appear at the cursor as you move it), and drop the view.
3. Select **Window > Show View > Watch**. Get the address value of the grid global variable by entering **grid** under **Name** in the Watch view and pressing **ENTER**.
4. Enter the value displayed into the memory window address bar and press **ENTER**.

The Device Debug perspective now appears as shown.

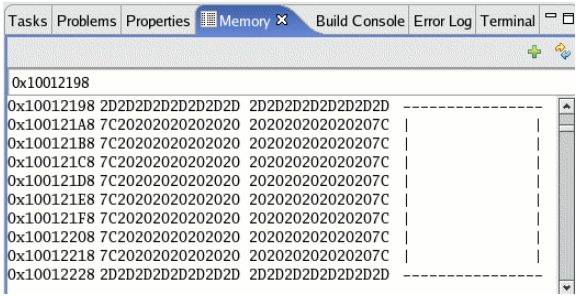


3.10.2 **Stepping to Initialize the Grid Array**

Press **F6** (or use the **Step Over** button in the Debug view) twice to step from the entry point of **main( )** to the call to **srand( )**. Using **F6** twice causes Workbench to step over and complete the execution of **gridInit( )**. (All the run controls are available on the **Run** menu, and also as arrow buttons in the **Debug** window.)

Adjust the **Memory** window to show the box outlining the grid on which the balls will bounce.

- Right-click in the **Memory** window and select **Display > Item size - 8 bytes**.
- Drag the right or left borders of the **Memory** window to make it exactly wide enough, and drag the top and bottom borders to make it high enough for the box in the text area of the window to appear correctly as shown below.



If the box does not appear, make sure the address you entered in the **Memory** window is that of the **grid** global variable.

3.10.3 Setting and Running to a Breakpoint

1. Now scroll down in **main.c** past the three initialization **for** loops and set a breakpoint at the **while** statement as described in the comment above it.

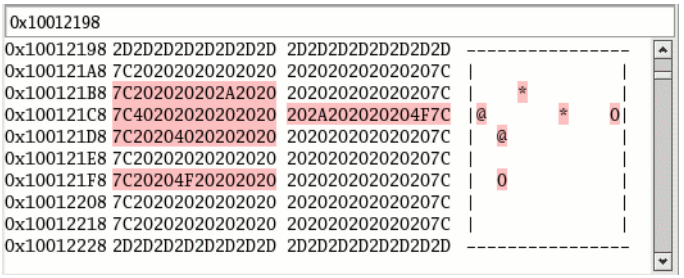
NOTE: If no comment appears above the **while** statement, click the **+** symbol in the gutter next to it to unfold the comment. You can click the **+** symbol again when you are done reading the comment to fold it away.

2. To set the breakpoint, double-click in the gutter next to the **while** statement.

   A blue dot appears in the gutter, and the Breakpoints view displays the module and line number of the breakpoint.

3. With the breakpoint set correctly, run to it by pressing **F8** or clicking **Resume** a few times. Execution stops each time it hits the breakpoint.

4. Examine the **Memory** window as you repeatedly resume execution. You should see the balls of the sample program move in the grid. The highlighting in the figure indicates changes since the last refresh.



3.10.4 **Modifying the Breakpoint**

Next, change the behavior of the breakpoint so that at each break, the display will refresh (showing the bouncing balls) without stopping execution.

1. Right-click the breakpoint in the gutter, or in the Breakpoints view, and select **Breakpoint Properties** from the context menu.

   The **Line Breakpoint Properties** dialog box appears.

2. Click the **Continue on Break** check box, click **OK**.

   Now press **F8** or click the **Resume** button to watch the balls bounce in the **Memory** window.

When you are done, you can click the red box button in the Debug view to terminate the process and click the grey Xs to remove the terminated launch.

3.11 **Creating Projects at External Locations**

In the example of the **ball** program, you imported existing files into a project in your workspace. In some cases, you may prefer to use the files in their existing location rather than, for example, maintaining copies of the files in two separate locations. You can do this by creating a user-defined project at the location of the source files. The project still appears in the Project Navigator like any project in your workspace, but the actual files, including Workbench metafiles (**.project** and **.wrproject**) are located in a location external to your workspace directory.

Building the Usermode Agent

The following example shows how to build the **usermode-agent** at an external location, using the source files that come with the Workbench installation.

1. Create a new user-defined project by right-clicking in the Project Navigator and selecting **New > User-Defined Project**, so that the makefile in the supplied source will be used.
2. If prompted, select your target operating system.
3. In the **New User-Defined Project** dialog box, assign a project name such as **usermode-agent**.
4. Select **Create project at external location**.
5. Click **Browse** and locate the **usermode-agent** source files located in *installDir*/**linux-2.x/usermode-agent/1.1/src**. Click **OK**, then **Next**.
6. In the **Project Structure** dialog box, click **Next**.
7. In the **Build Support** dialog box, select **User-defined build**. Leave the **Build command** text as it is for now. Click **Finish**.
8. Expand your new project in the **Project Navigator**. Notice that the project contains the **usermode-agent** source files as well as the project metafiles. Scroll to the **README** file and double-click it to open it in the Editor.
9. In the **README** file, find the appropriate **make** text for your target architecture and copy it. (If you are building **usermode-agent** for a Wind River Linux target, refer to *Building the Usermode Agent for Wind River Linux*, p.45.)
10. In the **Project Navigator**, right-click the project folder and select **Properties**. Select **Build Properties** and in the **Build Support** tab, replace the **Build command** text with the **make** text you copied from the **README** file, substituting any values as appropriate for your environment.
11. You can now build the project. The resulting binary will appear in the **bin/***arch* subdirectory of the **src** directory.

Building the Usermode Agent for Wind River Linux

If you are building the usermode agent for a Wind River Linux target, use the command:

$ **make -f Makefile ARCH=***ARCH* **CROSS_COMPILE=${WIND_GNU_PATH}/bin/***Cross_Compile* **LINUX_ROOT=/nonexistent TOPDIR=/***path_to_workspace/project_name*

See *E. Wind River Cross Compiler Prefixes* for correct values for *ARCH* and *Cross_Compile*.

4 Configuring Wind River Linux Platforms

4.1 Wind River Linux Platform Projects

This chapter describes how to create a new Wind River Linux Platform project, how to reconfigure the kernel and user space for that project, and how to easily deploy your new configuration on a target.

Creating a Wind River Linux Platform Project

When you use Workbench to create a new Wind River Linux Platform project, the choices you see in the project wizard are based on your installation. An initial **configure** command is then generated based on the dynamically-generated build spec.

Follow this procedure to create a new platform project:

1. Right-click in the Project Navigator and select **New > Wind River Linux Platform Project.**
2. Enter a name for the project and click **Next**.
3. Depending on your installation you will have different options for selecting a build spec, a root file system, and a kernel:
   - Select a build spec for your board from the **board** menu.
   - Select a file system size from the **RootFS** menu. Possible options are **glibc_cgl** for carrier grade Linux, **glibc_full** for standard Linux, **glibc_small** for a Busybox-based file system, and **uclibc_small** for one based on uClibc.
   - Select a kernel. Possible options are **cgl** and **small**.

   The **Command** window displays the **configure** command with arguments corresponding to your choices. Click **Next** unless you want to modify the **configure** command argument list as described in the next step.
4. If you want to add or modify arguments to the **configure** command, select **Options**. You can then enter options directly or select from the list provided, editing them as appropriate. Substitute actual values for **DIR**, **BUILD**, and so on, and choose between options specified in square brackets and separated by lines. For example, if you select the following option:

   ```
   --with-toolchain-version=[STRING|current]
   ```

   you would substitute a value for **STRING** or use "current" as in:

   ```
   --with-toolchain-version=current
   ```

   Note that if you select an argument from the list that contains a **DIR** value to be replaced, the **Browse** button becomes enabled and you can browse to choose the appropriate directory. Click **Add** to add each additional option.

   When you have finished adding options, click **OK** and then click **Next**.
5. Click **Next** to accept the default **Build Support** dialog box choices.
6. Click **Finish** in the **Static Analysis** dialog box to begin project configuration. If you enable **Static Analysis** you will incur a severe performance penalty as the tens of thousands of files involved in building a kernel are scanned.

The **configure** command is executed and output appears in the **Build Console**. The **configure** command output and additional information is also stored in the project's **creation.log** file.

At this point you can build the kernel, file system, and other targets, as described next. You can also reconfigure kernel parameters as described in *4.2 Configuring Wind River Linux Platform Kernels*, p.51 and add or remove RPMs from the file system as described in *4.4 Configuring User Space*, p.55 before building your target.

Contents of a Wind River Linux Platform Project

A new Wind River Linux platform project creates two subdirectories or folders in your workspace—one with the name of your project (*project*) and another with the name of your project with a **_prj** extension (*project*_**prj**).

NOTE: If you are importing an existing platform project into Workbench, the existing project location will be used and a *project*_**prj** folder will not be created. Wind River Linux Platforms and Workbench are integrated to allow you to use any combination of Linux command-line and Workbench GUI actions as desired.

The *project* folder contents appear in the Workbench Project Navigator and include the following:

- **Kernel Configuration** node—right-click and select **Build Target** to view and modify the kernel's **.config** file (see *4.2 Configuring Wind River Linux Platform Kernels*, p.51).
- **User Space Configuration** node—right-click and select **Build Target** to add or remove file system RPMs (see *4.4 Configuring User Space*, p.55).
- build nodes:

NOTE: The Build Console displays the full **make** command and output when building the following targets.

  - **all**—right-click and select **Build Target** to build all targets.
  - **cramfs**—right-click and select **Build Target** to build a CRAMFS file system.
  - **delete**—right-click and select **Build Target** to clean the project file. This should be used before removing a project by right-clicking on the project folder and selecting **Delete**.
  - **deploy**—deploy the project as described in *4.6 Automating Target Deployment*, p.66
  - **fs**—right-click and select **Build Target** to build the root file system.

  - **jffs2**—right-click and select **Build Target** to build a JFFS2 file system.
  - **kernel-build**—right-click and select **Build Target** to build the kernel.
  - **kernel-config**—right-click and select **Build Target** to configure the kernel using **config**.
  - **kenel-menuconfig**—right-click and select **Build Target** to configure the kernel using **make menuconfig**.
  - **kernel-rebuild**—right-click and select **Build Target** to rebuild the kernel.
  - **kernel-xconfig**—right-click and select **Build Target** to configure the kernel using **make xconfig**.
  - **ramfs**—right-click and select **Build Target** to build a RAMFS file system.
- project files:
  - **.project**—holds the binding of this project to the Wind River Linux target operating system. It also defines the Workbench link named **dist** from the project to the actual **dist** directory containing the source.[1]
  - **.wrproject**—holds the build targets for *project*, the **Kernel Configuration > General** configuration, and the fundamental build rules (**build**, **rebuild** and **clean**).
  - **creation.log**—contains detailed **configure** command output.
  - **Makefile.wr**—defines the make rules used by the build targets. This is used for Wind River Linux platform projects in place of **.wrmakefile**.

The *project*_**prj** folder holds the actual project contents such as the **dist** and **export** folders and is identical to a build directory created by the command line tools. This folder can exist outside of the workspace folder, for example when an existing project is imported into Workbench. In that case, the platform project has an Eclipse link to the actual location.

1. The **dist** link is an Eclipse link, not a Linux symbolic link, so you will not see a symlink named **dist** in the project folder.

4.2 Configuring Wind River Linux Platform Kernels

When you build a kernel in a Wind River Linux Platform project, you can easily view and modify the current kernel configuration with Workbench. This section summarizes the major uses of the kernel configuration tools. For detailed reference information on the **Kernel Configuration** node, refer to *Wind River Workbench User Interface Reference: Kernel Configuration Editor.*

Workbench provides a **Kernel Configuration** node for Wind River Linux Platform projects that presents the standard Linux configuration choices in a convenient GUI that quickly shows inter-dependencies and provides additional information on each configuration option. If you prefer, you can still use the **menuconfig** or **xconfig** interface, either from the command line or by clicking on the **menuconfig** or **xconfig** nodes in the platform project.

Kernel Configuration Node

When you double-click on **Kernel Configuration** in the project folder a tab with the name of your project opens in the central view, displaying a configuration window with two tabs: **Overview** and **Configuration**.

Overview Tab

The **Overview** tab displays the project configuration. This corresponds to the choices you made when creating a new Wind River Linux Platform project. The command-line arguments supplied to the **configure** command to create the initial configuration are shown at the bottom of the view.

Configuration Tab

The **Configuration** tab consists of two windows. The top window presents the items for configuration that may be familiar to you from using **menuconfig** or **xconfig** to configure kernels. The lower window consists of three tabs for **Help**, **Dependencies**, and **Kconfig** that provide additional information on the items you select in the top window.

NOTE: By moving your cursor around inside the Kernel Configuration view, you can find the drag points to enlarge view sections.

When you select an item in the top window, Help appears in the **Help** tab. Select the **Dependencies** tab to see any items your selected item depends on, or items that depend on it. Double-click on any item dependency to move to it in the top window. Select the **Kconfig** tab to see the definition of the item and the file system location where it is defined. If you click on the file path link shown in blue the file is added to the Edit view and you are placed at the location of the item definition.

Create a new kernel configuration by modifying the settings of menu items in the top window. The current configuration status of items is indicated by the icons associated with them. (Refer to *Wind River Workbench User Interface Reference: Kernel Configuration Editor* for a detailed description of the icons in the menu.)

The Configuration view does not show all items as some of them are disabled due to dependency conditions. To view all items, including disabled items, right-click in the Configuration view and select **Filters > Show All**. The disabled items are shown as greyed-out. You can still select them and view the **Dependencies** tab to understand the reasons they are disabled.

Modifying the Configuration

For example, if you want your kernel to include support for PCMCIA or CardBus cards, find **Bus options** in the scroll-down list, click on the arrow to expand it and then click on the arrow next to **PCCard**. This displays the entry for **PCCard support**. This is a tri-state configuration item which can be set to **n** for not included, **m** for included as module, or **y** for built-in. Double-click on the item to change the setting. See the **Help** tab for details on what the settings mean for the particular item.

To find options, for example those concerned with USB support, select **Edit > Find** and enter the string **USB**.

To generate the new kernel, right-click **reconfig** in the project folder and select **Build Target**.

To save the configuration file select **File > Save**.

4.3 Adding Kernel Modules to the Platform

You can create a kernel module as a subproject of a kernel project, and the kernel module subproject will inherit the kernel build properties. This makes creating a kernel module a simple task. Workbench comes with a sample kernel module project that demonstrates this and is shown in the following procedure.

NOTE: The following procedure assumes you have already built the kernel in a platform project. You will make the kernel module in this procedure as a subproject of that platform project.

1. From the main menu bar, select **File > New > Example** and select **Wind River Linux Sample Project**. Click **Next**.
2. Select **Wind River Linux Kernel Module Sample Project** and click **Next**.
3. Select **Kernel Module Debug 2.6 Demonstration** and double-click on it or click **Finish**. A new project called **moduledebug_2_6** appears in the Project Navigator.
4. Select the platform project that you have already built the kernel for and to which you are going to add this new kernel module project. Then right-click and select **Properties > Project References**.
5. Select the check box for **moduledebug_2_6** and click **OK**. The **moduledebug_2_6** project will disappear from the project level and appear as a subproject inside your selected platform project.
6. Expand the platform project and find **moduledebug_2_6**. Right-click on **moduledebug_2_6** and select **Build Project**.

NOTE: If you see **va_list** errors here it probably means that you have not yet built the kernel in the platform project. You should do that before building the kernel module subproject. You an build the kernel in a project by right-clicking on **kernel_build**, and selecting **Build Target**.

After the successful build you should see the **moduleDebugSample.ko** kernel module in the **moduledebug_2_6** subproject.

NOTE: If you select the subproject and right-click and select **Properties > Build Properties**, you can see how the kernel root directory, target architecture, and cross-compiler prefix have been set from the parent platform project.

Creating a Custom Kernel Module

It is a simple matter to configure the build system for a custom kernel module created from your source files once you have created a platform project. In this example you will directly attach the new module to an existing platform project. Alternatively, you You could skip that step and enter the **KERNEL**, **ARCH**, and **CROSS_COMPILE** values directly for an external kernel, or even leave those fields blank and attach or set them at a later date.

When you build this custom kernel module, note that the generated Makefile automatically includes the module's source files, and the module name has been set from the project name.

1. Right-click in the Project Navigator and select **New > Wind River Linux Kernel Module Project**.
2. Assign a name to your kernel module project and click **Next**.
3. Select a platform project as a superproject for this kernel module and click **Next**.
4. Note that the **Linux kernel information** at the bottom of the dialog is filled-in because the values have been imported from the platform superproject. Click **Finish**.
5. The kernel module project now appears as subproject of the platform superproject in the Project Navigator.
6. Right-click on the new kernel module project and select **Import > General > File System**. Browse to the directory *installDir***/wrlinux-1.4/samples/moduledebug_2_6** and click **OK**. Select the files **bpHere.c** and **moduledebug.c**.

NOTE: At this point, of course, you can import your own source files. The files you are importing in this example are provided as an illustration and can be used to demonstrate kernel module builds and testing. The **moduledebug.c** file provides the module init and exit points, and **bpHere.c** sets a breakpoint for testing purposes.

Click **Finish.**

7. You can now build the module, deploy it, and test it.

NOTE: If you have not built the platform kernel, you will not be able to build the kernel module until you do so.

Moving the Kernel Module Project

To move the kernel module project from one platform project to another, use the following procedure:

1. Right-click on the kernel module project and select **Build Clean**.
1. Right-click on the platform project that currently contains the kernel module subproject and select **Properties**.
2. Click on **Project References** and then deselect the kernel module project. Click **OK**. The kernel module project re-appears in the Project Navigator at the project level.
3. Right-click on the platform project that you want to contain the kernel module subproject and select **Properties**.
4. Click on **Project References** and then select the kernel module project. Click **OK**. The kernel module project now appears as a subproject of the new platform project.
5. Right-click on the kernel module project and select **Build Clean.**

4.4 Configuring User Space

Wind River Linux Platform projects have a **User Space Configuration** node that allows you to add and remove RPMs from the target file system.

Double-click **User Space Configuration** in the platform project to open the **Platform Configuration** dialog in the central view. A list of installed packages is displayed below **Installed Packages**. If you look in the **pkglist** file in the *project*_**prj** directory, you will see the same list of files. In Workbench, each package is listed with its version number and the amount of space it takes.

NOTE: By moving your cursor around inside the Package Configuration view, you can find the drag points to enlarge view sections.

To get more information on any package, select it and look at the tabs at the bottom of the view.

- The **General** tab gives a brief summary of what the package is.

- The **Contents** tab lists the various files and directories that the package consists of.
- The **Dependencies** tab displays package interdependencies in two windows: the **Requires** window lists the packages that are required by the package you have selected. The **Required By** windows lists the packages that require the package you have selected.
- The **Options** tab lets you turn debugging on or off for particular packages.
- The **Targets** tab
- The **Log** tab

Removing and Adding Packages

Use the **Remove** and **Add** buttons to remove and add packages from your configuration.

For example, you could do the following to remove packages:

1. Select the **Dependencies** tab and then select a package from the **Installed Packages** list that contains entries in both the **Required** and **Required By** windows such as **openssl**.
2. Click **Remove** to remove the installed package that you have selected. A dialog box appears that lists all the packages that will be removed (the selected package and those packages it is required by) and the total space taken by the packages.
3. Click **OK** to remove the packages.

The removed packages now appear in the **Installable Packages** list. Select F**ile > Save** and look in the **pkglist** file. The packages will have been removed from the list in that file.

To reinstall the same packages, do the following:

1. Select them from the **Installable Packages** list and click **Add**. A dialog box will prompt you with the package(s) to be added and the total space involved. If other packages are required for the package you have selected to be added, they will be shown in the dialog box as well.
2. Click **OK** and the package(s) appear in the **Installed Packages** list.

You can add back the other packages that you removed and then select **File > Save** to restore the package configuration to its original condition. If you look in the **pkglist** file, all packages are now listed there.

Note that you can select multiple packages to add or remove at one time.

Debugging Packages

To enable debugging for a package, do the following:

1. Select the package.
2. Select the **Options** tab.
3. In the **Options** tab, click the **Debug** check box.
4. Click **Apply**.

If you look in the **pkglist** file and find the package you just marked for debugging, you find the package name has **BUILD_TYPE=debug** next to it.

To turn off debugging, do the following:

1. Select the package
2. Select the **Options** tab
3. Unselect the **Debug** check box.
4. Click **Apply**.

Building Packages

Select the **Targets** tab for a set of buttons of build targets for the selected package. The build targets are the usual with the addition of **prepatch** which allows you to prepare the package for patching and is described in the next section.

4.5 Managing Patches

There are two patching models supported by the Wind River Linux build system.

The basic model, called the Classic patching model, carries forward from wrlinux-1.3 and is based on patch list files and the program **patch**. This is the default for command-line configured projects, since this is streamlined for the validated product.

The other model, called the Quilt patching model, is based on the industry standard patching tool called Quilt. While Quilt is also based on **patch**, it provides an additional rich set of patching features, such as pushing new patches to go forward, popping patches to go backward, capturing development changes into new patches, deriving patch annotation data, and more. The Quilt patching model is the default for Workbench-configured projects because it provides these advanced patching features.

Once a project is configured and built, you are free to continue with either model. If, however, you would like to ability to pop backwards down into the product's Wind River Linux provided patches, then it is better to use Quilt.

NOTE: There is a configure option **-enable-quilt=[no | yes]** that can override the default patching model for a particular project configuration.

This section demonstrates the Wind River patch manager and its many features and illustrates strategies for resolving potential rejects. Procedures show how to successfully apply patches, how errors are reported and can be resolved, and how to accept and generate rejects so that you can track them and resolve them at a later time.

Applying Patches

This example shows you how to apply patches by using a set of patches that are known to work, in this case the patches to the **apache-ssl** package.

Preparing to Patch

In this example, we need a configured platform project that includes Apache, for example one with the CGL or Standard file system.

1. Select **File > New > Wind River Linux Platform Project** and create a new platform project, for example one where **RootFS** is **glibc_cgl** and **kernel** is **cgl**. Click **Finish** to configure the project.

2. By default, the project uses the pre-built packages. Open up the project, and double-click on the **User Space Configuration** icon.

   In the Package Manager scroll down and select the **apache-ssl** package.

3. Click on the **Targets** tab below, and then click on the **prepatch** button. This unpacks the package and prepares it for patching. You will see the patch preparation progress in the Build Console.

4. When the prepatch step is complete, you must refresh the project file list. Right-click on the **build** directory of the project and select **Refresh**.

NOTE: Workbench does not automatically refresh the file list. This is because of the enormous number of files in a typical platform project (28,000 to 48,000), where every small change could otherwise force repeated and lengthy delays. Consequently, when changes are made (packages are expanded or compiled), you must refresh the project's file list manually.

Starting the Patch Dialog

1. Expand the **build** container within the project, right-click on the **apache_1.3.34** directory, and select **Team > Apply Patch**. This selects the patch default destination directory and opens the dialog.

2. Select the option **Patch list file**, and then browse to and select the file *workspace/project*_**prj/build/apache_1.3.34/WRLINUX_PATCHES** click **OK** and then click **Next**.

3. Select the first patch file in the list, **apache-native-cc**, and click **Next**. Do not click the **Finish** button at this point unless you are absolutely sure that the patch will apply without rejects.

NOTE: Where did this patch list file come from? When you built the **prepatch** target for this package, the Wind River Linux build system created two patch list files, one for the classic patch model (which is always placed in the file named **WRLINUX_PATCHES**), and one for the Quilt patch model (which is always placed in a directory called **WRLINUX_QUILT_PATCHES**). The first file lists the patches with their normal absolute path names, making it easy to select a patch file that we know will work.

4. Select the **apache_1.3.34** directory and click **Next**. Again, do not click **Finish** at this time.

Correcting a Patch Path Mismatch

1. You are now at the center of the patch application, the **Verify Patch** dialog box.

   As you can see, all of the patch hunks (a hunk is a group of contiguous lines) are marked with a red "x". This indicates that these hunks did not apply correctly to the source.

   Note that the patch hunk groups all have a leading path of **apache_1.3.33/**. The package you are patching has the version of **_1.3.34** in its directory name (ending in **4**, not **3**). The errors are caused by a mismatch of the path within the patch file, a not uncommon occurrence.

2. Click on the drop-down box for **Ignore leading path name segments**, and select **1**. Note that the hunks all lose the red "x" error indication. The path name issue has been resolved.

NOTE: If the hunks do not lose the error indication, or the lower source text box remains empty, then you may need to explicitly refresh the project's file list. Workbench caches the file list, and if you replace, update, clean, remove, or expand the file list without a refresh, Workbench may still have the obsolete file handle. If that is the case you must cancel this dialog, refresh the project's file list, and repeat the previous steps. If you see the **apache_1.3.34** directory within the build directory, you can limit your refresh to that directory, as opposed to refreshing the whole project.

Browsing Patch Hunks

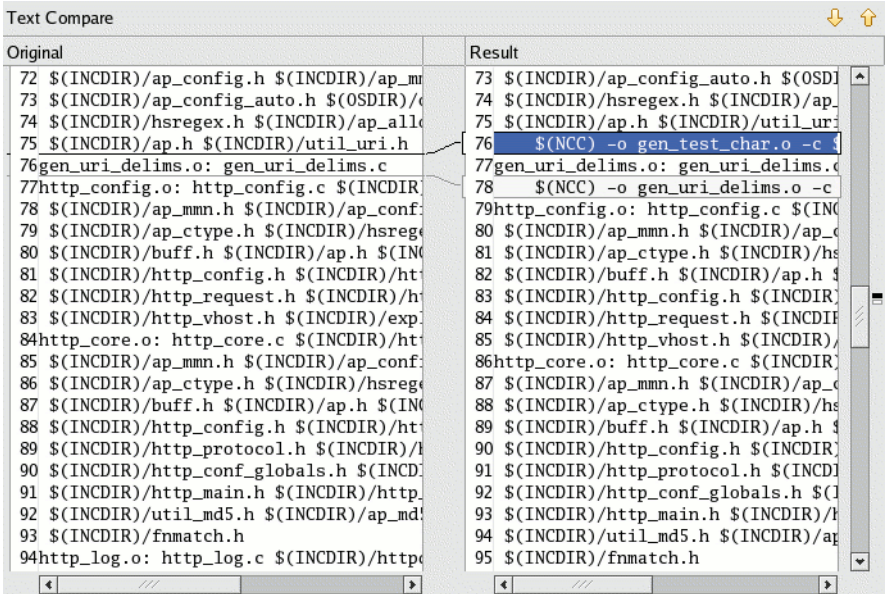
You can now browse the patch hunks, and see their effect on the source file.

1. There are three hunk groups, one for each target source file affected by this patch.
2. Click on one or more of the check-boxes for the file hunk group. When checked, the hunk is applied and the change becomes visible in the **Text Compare** area.
3. Double click on a hunk, and observe that **Text Compare** jumps to that location.
4. Click on the gold up and down arrows in the middle right of the dialog to go forward and back through the hunks.

   You can also click on the marks in the right-hand gutter of the **Text Compare** area to jump to hunk locations.

5. Observe how the insertion or deletion of patch text is represented in the **Text Compare** area. Note in Figure 4-1 how two inserted lines in one hunk go around an unchanged line and are indicated by two insert notations (highlighted).

Figure 4-1 **Patch Application Example**



Applying the Patch

Now that the patch is observed to apply cleanly (no red “x”s), you can finish applying it.

1. Click on all of the hunk groups so that all the patch hunks are selected to apply.
2. Click **Finish**. This applies the patch. If you were to look at the three target source files, you can see the changes.
3. Repeat this process with the second (and final) official patch for this package, the patch file **cross_compile.diff**.

Patch Reject Resolution

The following procedure applies a patch with known errors to demonstrate some patch resolution techniques. Copy the patch file **mypatch.patch** from the appendix *G. Broken Patch File Example* and place it in a browsable path on your filesystem.

1. Open the **Apply Patch** dialog (right-click on **apache_1.3.34** in the Project Navigator and select **Team > Apply Patch**), select the **File** option, and browse to the patch file **myApache.patch**. Click **Next**.
2. Select **apache_1.3.34** as the resource to patch. (Click **Next**.
3. When you get to the **Verify Patch** dialog, you will see that all of the patch hunks fail. Also, since the target file **configure** has no leading path, you know that this cannot be fixed by changing the leading path segments.
4. Set the **Maximum fuzz factor** to **0**. The patch manager looks for a place where the context lines of the patch match exactly with the file when this setting is zero. If set to **1**, it allows the first and last lines of the context match, and so on. A little "fuzz" is normal and we are at first being strict to demonstrate fuzz factor.
5. A typical check to make at this point is to select the **Reverse Patch** box. Note that immediately the first hunk now applies correctly, and that the source appears. This tells you that the patch file was created backwards, resulting in the hunk changes going in the reverse direction.
6. Select the first hunk by highlighting it —do not check its box yet. Selecting it in this way locates the **Original** and **Result** views at the location where the patch will be applied to **Result**. Now check the box for the first hunk and see how the patch is to be applied in **Result**.
7. Highlight the second hunk to browse to it, then check the box to see how the patch is to be applied in **Result**. Some fuzz factor will be required here—change the **Maximum fuzz factor** to **2** and note that the hunk can now be applied.
8. Highlight the third hunk and select its check box to see how the patch applies. The problem here is that the source and the patch do not match correctly with the leading and following context lines, due to the extra **<<< SOURCE FUDGE >>>** line expected. Two ways you can correct this are:
    a. Add the expected line to the source file. In Workbench, you can simply select the **<< SOURCE FUDGE >>** line in the **Result** view and paste it into the source file in the **Original** view in the correct location (as a new line after the blank line following **shadow="**). Then click **Update**.

b. Alternatively, you could edit the **myApatch.patch** file to incorporate the line in the patch. Edit **myApatch.patch** in a separate editor, make your changes, and save it. Then click **Back** twice and click **Forward** twice to get the new patch file applied.

9. You can experiment with the remainder of the hunks. Read the annotations on the patch file in *Annotated Patch File*, p.374 for additional information. Leave some hunks in the rejected state to use in the next procedures.

Accepting Rejects, Inline or into Reject Files

Rather than trying to fix the remaining rejected hunks, use this procedure to accept these rejects for resolution at a later time.

1. Click the check-boxes for the remaining hunks, so that they will apply even though there are errors. You can see that the rejected hunks appear within the **Result** text box, still in their **patch** syntax form.
2. Click the check box **Create Inline Rejection**. This will keep the inserted patch rejections within the target file.
3. Click **Finish**.

Alternatively, you could have kept the **Create Inline Rejection** option off. This would have resulted in the creation of a file called (in this case) **configure.rej** in the same directory as the respective target source file.

Review the Accepted Rejections in the Tasks List

Now that you have accepted some rejections, you can use Workbench to keep track of them.

1. In the Workbench main perspective, locate the **Tasks** tab in the center lower view. If there is no **Tasks** view, you may need to enable and place this particular view (**Window > Show View > Tasks**).
2. Observe that there is a new entry in this list—the note about the accepted rejection.
3. Double-click on this rejection task, and observe that the respective rejection location appears in the Editor.

Viewing Patching Annotation using Workbench

Workbench also supports a patch annotation feature, using a facility provided by Quilt.

1. In the project container, browse to the following file and select it:

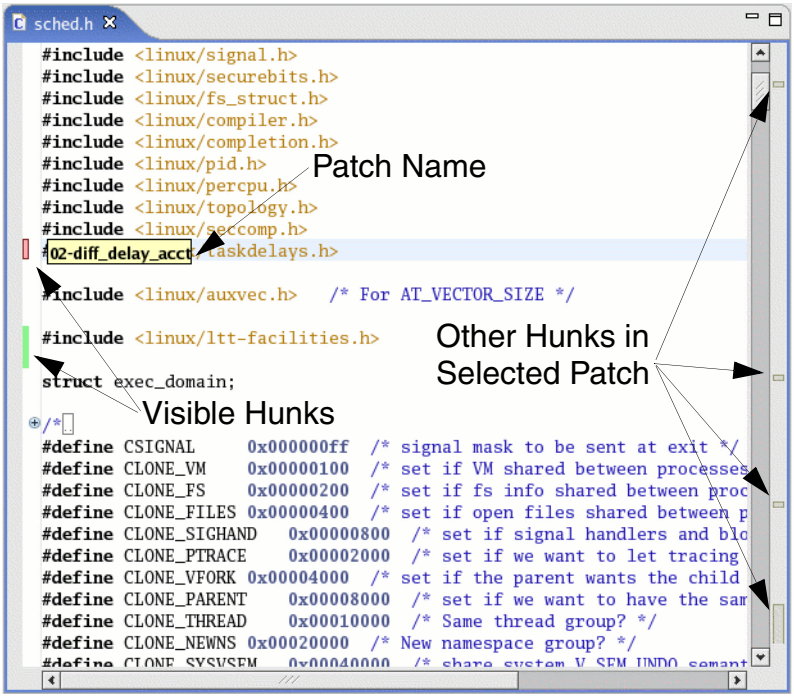
   `build/linux-2.6.14-cgl/include/linux/sched.h`

2. Right-click on this file and select **Show Patch Annotations**.

NOTE: If the menu item **Show Patch Annotations** does not appear, that means Workbench was not able to find the **WRLINUX_QUILT_PATCHES** directory generated and required by Quilt.

3. If prompted to enable **quick diff**, click Yes
4. After a moment, this file should appear. As you scroll up and down this file, observe the colored bars in the left hand gutter.
5. Hover the mouse over one of these colored bars. After a few seconds, a pop-up box appears, giving the name of the patch that contributed this line.
6. Observe that when one of these pop-up boxes appear, some number of boxes also appear in the gutter on the right. These boxes indicate the other locations in this file that were contributed by this patch (see Figure 4-2).

Figure 4-2 **Example of Patch Annotation**



4.6 Automating Target Deployment

You can cause your target to be rebooted with your latest kernel and file system builds using the **deploy** target in the Project Navigator.

Edit the *installDir*/**wrlinux-1.4/wrlinux/deploy.conf.in** file to include the values as shown in Table 4-1.

Table 4-1 **deploy.conf.in Values**

Field and Value	Description
USER = validUser	Substitute your user name for **validUser**. You will be prompted for the root password during deployment.
TARGET_FS_DIR = /nfsRoot/targets/safeToDelete/ testTarget/fs	Replace the provided path with the path on the host that the target mounts as its root file system. The path *must* contain the string **/safeToDelete**.
TARGET_SHUTDOWN_COMMAND = echo ssh targetServer /targetCmds/targetShutdown	Replace **targetServer** with the name or IP address of your target server, and replace **/targetCmds/targetShutdown** with the full path to the command you use to shut down the target.
TARGET_REBOOT_COMMAND = echo ssh targetServer /targetCmds/targetReboot	Replace **targetServer** with the name or IP address of your target server, and replace **/targetCmds/targetReboot** with the full path to the command you use to reboot your target.

The values you enter in the **deploy.conf.in** file are used by the **deploy.sh.in** script. For example in the following line in **deploy.sh.in**:

```
$TARGET_SHUTDOWN_COMMAND $TARGET_ID || exit 1
```

your shutdown command and target name will be substituted. Refer to the comments and commands in the **deploy.sh.in** script if you would like any further information on how these values are used.

Once you have configured the **deploy.conf.in** file you can easily deploy each new file system build to your target as follows:

1. Select **deploy** in the Project Navigator,
2. Right-click in the Project Navigator and select **Build Target**.

Your target will be deployed with the new file system.

5 Kernel Debugging (Kernel Mode)

5.1 Introduction

Wind River Workbench supports source-level debugging for Linux 2.6 kernels that support the Kernel GNU Debugger (KGDB). The KGDB functionality is provided with Wind River Linux platforms.

NOTE: If you are not using a Wind River Linux platform, contact Wind River support for information on configuring your kernel for symbolic debugging with Workbench.

With KGDB installed, you can debug the kernel much as you debug applications, including stepping through code, setting breakpoints, and examining variables. Kernel mode debugging between Workbench and the target takes place over a serial protocol connection which may be over a serial line or Ethernet.

NOTE: With Wind River Linux, kernel debugging is enabled by default and you only have to turn it off when going to production. With other versions of Linux be sure to enable symbolic debugging when building your kernel. For example, select **Kernel hacking > Compile the kernel with debug info** when using **make menuconfig**. You will use the generated **vmlinux** file for symbolic debugging.

5.2 Configuring the Target for Kernel Mode Debugging

When you have your Wind River Linux or other kernel configured for KGDB, you must configure your target with a kernel module as described in this section.

NOTE: When building your KGDB-enabled kernel, be sure to enable symbolic debugging. For example, select
Kernel hacking > Compile the kernel with debug info in the **make menuconfig** GUI.

5.2.1 Installing KGDB on the Target

Once you have built your kernel appropriately, you must install the KGDB kernel module on the target so that it can communicate with GDB on the host.

1. Locate the appropriate KGDB module in your kernel source tree. For an Ethernet connection it is **kgdboe.ko**, and for a serial connection it is **kgdb_8250.ko**.
2. If your KGDB kernel module is not in a file system exported to the target, copy it to the target.
3. If you are installing the Ethernet module **kgdboe.ko**, follow the instructions in this step. If you are installing the serial module **kgdb_8250.ko** for use with a direct or terminal server connection, skip to the next step.

   The **kgdboe** module usage syntax is:

   kgdboe=[*src-port*]**@**[*src-ip*]**/**[*dev*]**,**[*tgt-port*]**@***tgt-ip***/**[*tgt-macaddr*]

NOTE: In the usage statement, *tgt* is the host running Workbench. This command is entered on the target so the host is the target from the local point-of-view.

For example you might enter the following command on the target:

```
target_# modprobe kgdboe kgdboe=@/,@192.168.1.8/
```

where the host running Workbench has the IP address 192.168.1.8.

NOTE: The command above was entered on the target. Only the address of the host needs to be supplied as the target address default is the local host. The Ethernet device was not specified, because the target uses the default eth0.

To specify a different Ethernet device, for example **eth2**, enter:

```
target_# modprobe kgdboe kgdboe=@/eth2,@192.168.1.8/
```

4. If you are installing the serial module **kgdb_8250.ko** for a direct serial or terminal server connection, use the following syntax:

   modprobe kgdb_8250 kgdb8250=*port***,***speed*

   where *port* is the serial (comm) port on the target and *speed* is baud rate.

   For example:

   ```
   target_# modprobe kgdb_8250 kgdb8250=0,115200
   ```

   where the local (target) serial port to use is **/dev/ttyS0**.

NOTE: In the above command, **kgdb_8250** includes an underscore, and **kgbd8250=** does not.

5. You can verify your module has been installed with the **lsmod** command:

   ```
   target_# lsmod | grep kgdb
   kgdboe        6080   0
   ```

6. Remove the module only when you are not attached for debugging. To remove the module, use **rmmod**:

   ```
   target_# rmmod kgdboe
   ```

5.2.2 Booting the Target

You can configure your Workbench host to provide a kernel with TFTP and NFS-export a root file system to the target. An example of how to do this for Linux 2.4 kernels and the PPC32 architecture is described in *F. Configuring Linux 2.4 Targets (Dual Mode)*. For a target with the IA32 architecture, you could use the GNU cross-compiler tools and configure a PXE boot environment in which the target gets its IP address from a DHCP server and its kernel from your Workbench host acting as the PXE boot server.

For symbolic debugging, the **vmlinux** symbol file should be accessible to Workbench on the host.

5.3 Kernel Mode Debugging

This section discusses creating the connection between Workbench and the target and also how to attach to the kernel core on the target and perform debugging operations on it.

NOTE: This section assumes you have a target running a KGDB-enabled kernel.

5.3.1 Types of KGDB Connections

Your target kernel must be configured with KGDB serial or Ethernet capability. You can connect to the target in one of the following three ways:

- directly to the target with a standard 8250 serial connection using a null-modem cable.
- to a terminal server with TCP. You will need the IP address of the terminal server and a port number to connect to.
- by Ethernet with UDP. You will need the IP address of the target and the KGDB port number (default 6443). The target and host must be on the same subnet.

A Workbench wizard will take you through the process of configuring the connection, supplying defaults where applicable, as described in the following procedure.

5.3.2 Creating a KGDB Connection

1. To create a KGDB connection to the target kernel using Ethernet, click the **Create a New Connection** button in the **Target Manager** and select **Wind River Linux kgdb Connection** in the **Connection Type** dialog box. Click **Next**.
2. In this example, we installed the KGDB-over-Ethernet module **kgdboe.ko**, so select the **Linux KGDB via Ethernet** connection in the **GDB Serial Connection Templates** dialog box and click **Next**.

   If you installed the **kgdb_8250.ko** module for connection over an 8250 serial line you would select the **Linux KGDB via RS232** connection.

   NOTE: For a serial line connection, you must set the serial port permissions on the host to allow access, for example:

   ```
   # chmod 777 /dev/ttyS0
   ```

   or add your user name to the **uucp** group which has access to **/dev/ttyS0**.

   Click **Next**.
3. In the **GDB Remote Serial Protocol Connection Properties** page, if you are connecting over Ethernet, do the following:
   - For **Back End**, select **UDP** from the drop-down list (may already be selected).
   - For **CPU**, select the CPU type of your target from the drop-down list. If you are using Wind River Linux, leave this at the default setting **default from target** and the target will be automatically identified.
   - Check the box **Use character based break** (may already be checked).
   - In **Name/IP Address**, enter the hostname or IP address of the target.
   - For **Port**, enter 6443 (may already be entered).

   Click **Next**.

   If you installed the serial module to connect your target, do the following:
   - For **Back End**, select **RS232** for a direct connection, or **TCP** for a terminal server connection, from the drop-down list (may already be selected).
   - For **CPU**, select the CPU type of your target from the drop-down list. If you are using Wind River Linux, leave this at the default setting **default from target** and the target will be automatically identified.

- Uncheck the box **Use character based break** (may already be unchecked).
- Select the appropriate serial connection settings.

Regardless of connection type, the **Manual Options** box allows you to enter the location of a debug log, for example **debugcmds=/tmp/kgdb.log**.

NOTE: The **New Connection** dialog box is called **GDB Remote Serial Protocol Connection Properties** whether you are using a serial or Ethernet connection. The GDB-to-KGDB communication between host and target uses a serial communications protocol regardless of medium.

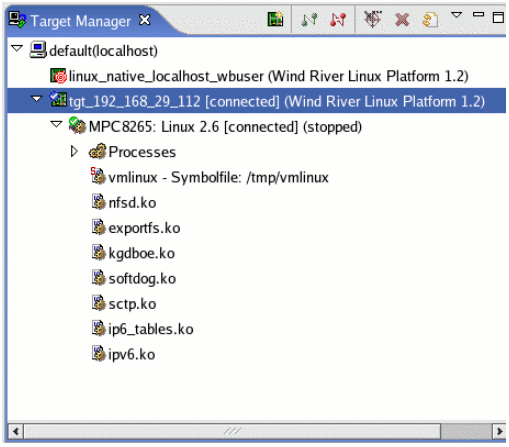
4. In the **Target Operating System Settings** dialog box, select your target kernel version from the **Booted Target OS on selected CPU** drop-down list, and enter the path to your **vmlinux** kernel symbol file in the **Kernel image** field.

   Click **Next**.

   In the **Object Path Mappings** dialog box, click **Add** to enter the path on your Workbench host that is the root file system on the target. For example, if you are exporting **/target** to the target, enter **/target/** (include the terminating slash) for the **Host Path**. Leave the **Target Path** field blank.

   Click **Next**.

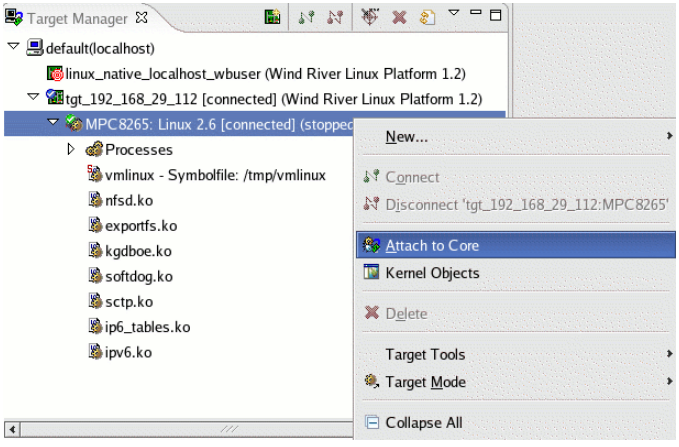
5. Click **Next** in the **Target State Refresh** dialog box.
6. Click **Finish** in the **Connection Summary** dialog box.
7. The connection will be attempted automatically. If successful, the **Target Manager** displays the **connected** message, and the kernel is shown as **Stopped**.



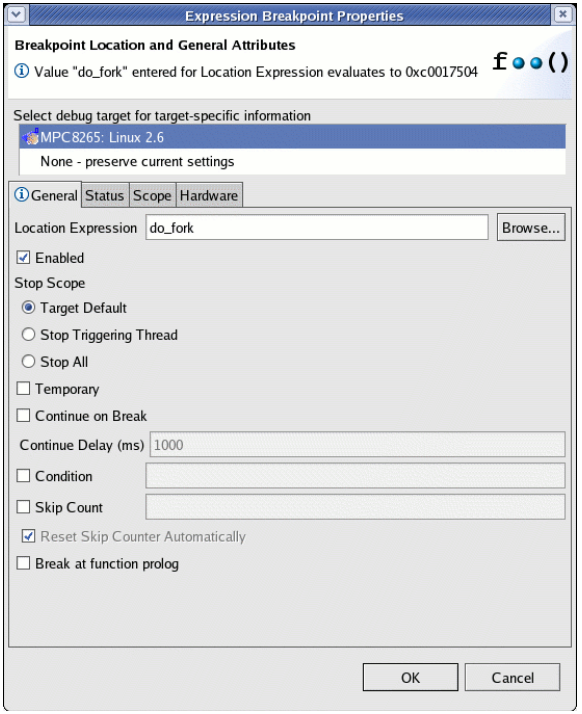
5.3.3 Attaching to Core and Debugging the Kernel

NOTE: You should have a terminal window open or console access on the target to perform all the steps in this procedure.

1. To begin kernel mode debugging, in the **Target Manager** right-click the *CPU*: *Kernel* entry and select **Attach to Core**.



2. Workbench shifts to the Device Debug perspective and the **Editor** opens to the stopped location in the **kgdb.c** source file.
3. If you have a terminal window open to the target, you will see that it is stopped. If you try to enter a command, for example **ls**, you will find that it does not return. To resume execution of the kernel, click the **Resume** button in the **Debug** view. Commands entered on the target now work.
4. With the kernel running again, set a breakpoint. Select **Run > Breakpoints > Add Expression Breakpoint** and in the **Breakpoint Location and General Attributes** dialog box, enter **do_fork** in the **Location Expression** box. Click **OK**.



5. Now enter a command on the target. It will not return because execution has stopped at the fork. The **Editor** will open the **fork.c** source file. Clicking the **Resume** button will allow the command to complete and the next fork, for example another command, will again cause a break.

CAUTION: It is not advisable to single-step in the kernel, especially over spin locks or Ethernet code when your KGDB connection is by Ethernet.

6. To complete this example, remove the breakpoint and click the **Resume** button to continue execution of the kernel. To end the session, right-click the target in the **Target Manager** and select **Disconnect**.

NOTE: If you recompile the kernel, you must disconnect before rebooting the target. This is not necessary with a Wind River Linux target as described next.

5.3.4 Rebooting the Wind River Linux Target

When you are connected to a Wind River Linux target with KGDB, you can reboot using the command line or the GUI.

When you reboot from the command line the debugger disconnects and waits until the target comes back up. When the target comes back up the debugger reloads symbols and re-plants breakpoints.

In Workbench, reboot the target by right-clicking on the connection in the Target Manager and selecting **Reset Connected Target**. Workbench reboots the target and beginning two seconds after reboot begins to try to contact the target. It will try to reconnect for approximately five minutes and then give up. These time periods are configurable if you are having trouble connecting to a target after reboots.

Configuring Target Reconnection Parameters

In the **Target Plugin Pass-through Options** field of the Target Manager's **Connection Properties** window, you can modify the amount of time Workbench waits before attempting to contact the target or how long it keeps trying to contact the target after a KGDB reboot.

To change the length of time that Workbench waits before giving up on the target, set the **resetwait** parameter. The default is **150** which is approximately five minutes.

To change the length of time Workbench waits before attempting to contact the target after rebooting, set the bootwait parameter. The default is **2** for two seconds.

Separate multiple parameters with semicolons. For example, to change both times you could enter:

```
bootwait=25;resetwait=50
```

Then close the dialog box and make the connection to the target.

5.4 Working with Kernel Modules

Workbench includes a sample project to demonstrate kernel module debugging as discussed in this section. This example requires that you have access to the kernel source you used to build the kernel and that you have terminal or console window access to the target.

5.4.1 Build the Sample Module

1. Start Workbench and select **File > New > Project**. In the **New Project** dialog box, expand the **Examples** node, select **Embedded** (or **Wind River) Linux Sample Project** and click **Next**.
2. In the **Sample Project Template** dialog box, select **Kernel Module Debug 2.6.10 Demonstration** and click **Finish**.
3. A new project called **moduledebug_2_6** appears in the **Project Navigator**. This is a user-defined project so it supplies its own **Makefile**. To modify the build properties for your environment, right-click the **moduledebug_2_6** project folder and select **Properties**.
4. Select **Build Properties** on the left of the **Properties** dialog box and select the **Build Support** tab. You must replace the values in brackets ({}) in the **Build command** box with appropriate values for your environment:
   - For **{linux_src_dir}**, substitute the appropriate path for your kernel source files.
   - For **{path_to_workspace}**, substitute the full path name of the workspace you are using.

- For **{ARCH}**, substitute the name of your target architecture, for example **ppc**.
- For **{cross_compile_prefix}**, substitute the prefix, including the terminating dash (**-**) of your cross-compiler commands. For example, if your cross-compiler tools have names such as **powerpc-gnu-gcc**, enter **powerpc-gnu-**.

NOTE: For Wind River Linux environments, see *E. Wind River Cross Compiler Prefixes* for architecture and cross-compiler prefix values.

Click **OK**.

Right-click the project folder and select **Build Project** to build the module. The result will be a **moduleDebugExample.ko** file which is the kernel module that you can now install on the target.

5.4.2 Install the Sample Module

Verify that the module is built correctly for your target by loading and removing it on the target.

1. Place the **moduleDebugExample.ko** file in the root file system that your host exports to the target, or copy it to the target with a network copy command.
2. To install the module, change directory to the module location or include the path to the module when you enter the **insmod** command:

```
kgdb_target_$ insmod moduleExampleDebug.ko
```

3. Verify that the kernel module is loaded:

```
kgdb_target_$ lsmod | grep moduleExampleDebug
moduleExampleDebug              5152   0
```

4. Remove the module and verify that it is removed:

```
kgdb_target_$ rmmod moduleExampleDebug
kgdb_target_$ lsmod | grep moduleExampleDebug
kgdb_target_$
```

5.4.3 Debugging Kernel Modules

This section provides a simple example of how to set a breakpoint in a kernel module and then step through it and resume processing to observe module operation.

Before you begin this procedure, you should cleanly disconnect from any running KGDB connection:

- If you currently have a KGDB connection to your target that is stopped, right-click the *CPU* **(***Kernel***)** entry and select **Attach to Core**. Click **Resume**. Then disconnect as described next.
- If you currently have a Workbench KGDB connection to your target, disconnect by selecting it in the **Target Manager** and clicking the **Disconnect** button.

1. On your target, install the **moduleDebugExample.ko** module with **insmod** as described in *5.4.2 Install the Sample Module*, p.79. Your module should be in the root (**/**) directory of the target file system.
2. To observe module output, enter the following command:

   ```
   # tail -f /var/log/kern.log
   ```

   You will see a sequence of **Global** and **local** messages with an incrementing count. This is output from **moduleDebugExample.ko**. If you do not see output and you have the kernel module loaded, use the following command to start the kernel logging daemon:

   ```
   # klogd
   ```
3. From Workbench, attach to your target with the KGDB connection you created in *5.3.2 Creating a KGDB Connection*, p.73.
4. In the **Target Manager**, expand **Processes** and find the entry for **moduleDebugExample.ko** below **vmlinux**. There should be a small red **S** in the icon for the kernel module indicating the symbols are loaded. If there is no **S**, right-click the entry and select **Load Symbols to Debug Server**.

NOTE: If you are unable to load symbols, be sure that you have your object path mappings set correctly as described in *5.3.2 Creating a KGDB Connection*, p.73.

5. Right click the *CPU* **(***Kernel***)** entry and select **Attach to Core**. Click **Resume** in the **Debug** view.
6. In the **Project Manager**, find the **bpHere.c** source file and double-click it to open it in the editor.
7. In the **bpHere.c** file, find the line with **putABreakPointHere** and put a breakpoint in the gutter on the left side of it by right-clicking and selecting **Add Breakpoint for System Context**.

8. Your kernel module output in the target window will stop when execution reaches the breakpoint. You can experiment with single-stepping or repeatedly resuming operation in the **Debug** view to control module output.
9. To complete this procedure, press **CTRL-C** in the target window to stop viewing output. Use **rmmod** to remove the module, and then disconnect from the target in the **Target Manager**.

5.4.4 Set a Hardware Breakpoint at Module Load

1. If you do not already have a KGDB connection to the target, make one now (see *5.3.2 Creating a KGDB Connection*, p.73). Then right-click the *CPU*: *Kernel* entry in the **Target Manager** and select **Attach to Core**. Click the **Resume** button in the **Debug** view so that the kernel is running. (See *5.3.3 Attaching to Core and Debugging the Kernel*, p.75.)
2. In Workbench, open the **module.c** file in the kernel source tree by selecting **File > Open File** and browsing to the **kernel** subdirectory. Double-click the **module.c** file to open it in the **Editor**.
3. Enter **CTRL-F** to open the **Find/Replace** dialog box and search for **sys_init_module;**—it will be at about line 1847 depending on your particular source file. Scroll down from there to find the following lines:

   ```
   /* Start the module */
   if (mod->init != NULL)
      ret = mod->init();
   ```

   Right-click in the gutter next to the second line shown (**if (mod->init != NULL)**), and select a System Context breakpoint.

   With a breakpoint set, close the **module.c** file.
4. In the terminal view or console on the target, install the module:

   ```
   kgdb_target_$ insmod moduleDebugExample.ko
   ```
5. In a moment, the **Debug** view in Workbench will show the **System Context** as stopped, **sys_init_module** highlighted, and the editor will open the **module.c** file to the location of the breakpoint. Note that the target window is stopped at the **insmod** command.
6. To resume processing, remove the breakpoint and click the **Resume** button in the **Debug** view. The **insmod** command completes. You can now remove the module:

   ```
   kgdb_target_$ rmmod moduleDebugExample.ko
   ```

5.4.5 Debug Kernel Module at Entry

The following procedure shows how to debug a kernel module at the module entry point.

1. In Workbench, open the **module.c** file in the kernel source tree. Select **File > Open** and browse to the kernel subdirectory of your build's Linux-2.6.14 source tree root. Double-click on the **module.c** file to open it in the Editor. For example, **module.c** might be found at:

   */home/user/workdir/myboard/***dist/linux-2.6.14/kernel/module.c**

2. Go to the routine **sys_init_module** and find the line that calls the module's **init** procedure (about line 1852 depending on your particular source file). Right-click in the gutter to insert a **System Context** breakpoint at the line as shown here:

   ```
   /* Start the Module */
   => if (mod->init != NULL)
        ret = mod->init();
   ```

   NOTE: If you find that the break point cannot be planted, make sure that you added the correct object mapping as described in *5.3.2 Creating a KGDB Connection*, p.73.

3. On the target, start the module.

   ```
   kgdb_target_$ insmod moduleDebugExample.ko
   ```

4. In a moment, the Debug view in Workbench will show the System Context as stopped, **load_module** highlighted, and the Editor will open the **module.c** file to the location of the breakpoint. Note that the target window is hung at the **insmod** command.

5. Ensure that **moduleDebugExample.ko** is registered by the Target Manager. Examine the Target Connection window for your target, and look for **moduleDebugExample.ko** in the list of installed kernel modules under **Processes**. If is not present, do the following to synchronize the target manager:

   a. Un-expand the icon for the target labeled **Processes**.

   b. Click **Refresh** in the Target Manager's tool bar.

   c. Re-expand the icon for the target labeled **Processes**.

   d. It the module does not yet appear, repeat steps a to d.

6. Ensure that the symbols for **moduleDebugExample.ko** are loaded. Examine the Target Connection window for your target, and look for **moduleDebugExample.ko** in the list of installed kernel modules under **Processes**. The icon for this module should be decorated with a red **S**. If it is not, right click on the module and select **Load Symbols**.
7. You can now single step into the initialization routine of the kernel module.
8. To resume processing, remove the breakpoint and click the Resume button in the Debug view. The **insmod** command completes. You can remove the module from the kernel with the **rmmod** command:

```
kgdb_target_$ rmmod moduleDebugExample.ko
```

PART III

Projects

6

Projects Overview

6.1 Introduction

Workbench uses *projects* as logical containers and as building blocks that can be linked together to create a software system. For example, the Project Navigator lets you visually organize projects into structures that reflect their inner dependencies, and therefore the order in which they are compiled and linked.

Pre-configured templates for various project types let you create or import projects using simple wizards that need only minimal input.

6.2 Workspace and Project Location

By default, your workspace directory is created within your Workbench installation directory. New projects are created in a subdirectory of the workspace directory, named with the project name. For Wind River Platform projects, an additional subdirectory named with the project name and a **_prj** extension is created (see *Contents of a Wind River Linux Platform Project*, p.49).

Wind River Workbench cannot know where your source files are located, so it initially suggests a default workspace directory within the installation directory. However, this is not a requirement, or even necessarily desirable. If you use a workspace directory outside of the Workbench installation tree this ensures that the integrity of your projects is preserved after product upgrades or installation modifications.

Normally, you would set your workspace directory at the root of your existing source code tree and create your Workbench projects there. For multiple, unrelated source code trees, you can use multiple workspaces.

Some considerations when deciding where to create your project:

Create project in workspace
Leave this selected if you want to create the project under the current workspace directory. This is typical for:

- Projects created from scratch with no existing sources.
- Projects where existing sources will be imported into them later on (for details, see *9.3 Adding Application Code to Projects*, p.108).
- Projects where you do not have write permission to the location of your source files.

Create project at external location
Select this option, click **Browse**, then navigate to a different location if you want to create the project outside the workspace. This is typical for:

- Projects being set up for already existing sources, removing the need to import or link to them later on.
- Projects being version-controlled, where sources are located outside the workspace.

Create project in workspace with content at external location
Select this option, click **Browse**, then navigate to your source location if you do not want to mix project files with your sources, or copy sources into your workspace. This is useful for:

- Projects where you do not have write permission to the location of your source files.
- Projects where team members have their own projects, but share common (sometimes read-only) source files. This option eliminates the need to create symbolic links to your external files before you can work with them in Workbench.

6.3 Creating New Projects

Although you can create projects anywhere, you would generally create them in your workspace directory (see *6.2 Workspace and Project Location*, p.88). If you follow this recommendation, there will generally be no need to navigate out of the workspace when you create projects. Note that if you do create projects outside the workspace, you must have write permission at the external location because Workbench project administration files are written to this location.

To create a new project, click the toolbar icon or select **File > New > Wind River Workbench Project** to open the New Wind River Workbench Project wizard. It will help you create one of the pre-configured project types. You can also select the specific type of project you want to create by clicking the toolbar icon or by selecting **File > New >** *ProjectType*. For more information about these projects, see *Overview of Preconfigured Project Types*, p.90.

To create one of the demonstration sample projects, select **File > New > Example** to open the New Example wizard. Each comes with instructions explaining the behavior of the program.

Whichever menu command you choose, a wizard will guide you through the process of creating the specific type of project you select.

6.3.1 Subsequent Modification of Project Creation Wizard Settings

All project creation wizard settings can be modified in the Project Properties once the project exists. To access the Project Properties from the Project Navigator, right-click the icon of the project you want to modify and select **Properties**. For more information about project properties, see *11.4 Accessing Build Properties*, p.136.

Project structural settings (sub- and superproject context of the project you are creating) can be most easily modified in the Project Navigator by dragging-and-dropping project folders into or outside other folders.

6.3.2 Projects and Application Code

All application code is managed by projects of one type or another. You can import an existing Workbench-compatible project as a whole, or you can add new or existing source code files to your projects. For more information, select **File > Import** to open the Import File dialog and press the help key for your host.

6.4 Overview of Preconfigured Project Types

Workbench offers the following preconfigured project types:

- Embedded Linux Kernel Project
- Embedded Linux Application Project
- Native Application Project
- User-defined Project

Depending on your installation and licensing, you may also have one or both of the following:

- Wind River Linux Application Project
- Wind River Linux Platform Project

The project types are described briefly below.

NOTE: You may see more project types depending on your installed software. Refer to the documentation on the particular software for details on those project types.

Embedded Linux Kernel Project

Use an Embedded *Linux kernel* project to configure (customize/scale) and build a Linux kernel to run on the target. (For Wind River Linux kernels, see *6. Projects Overview*.) Because your subsequent Linux application projects will run on the Linux kernel on the target, this is often a necessary first project to build, unless you already have a target kernel. See *F.6.1 Building the Kernel in Workbench as a Linux Kernel Project*, p.352 for a tutorial on building this type of project.

Embedded Linux Application Project

Embedded Linux application projects are built and reside on the host computer, but run on the target kernel. Workbench provides pre-defined build specs that you can use or modify for creating your embedded application projects. See *3.4 Creating a Project*, p.26, for a tutorial on building this type of project.

Native Application Project

A native application project is built and run on the host computer. In effect, your Workbench host serves as the target. See *3.3 Using Workbench*, p.24 for an example of building a native application project.

User-Defined Projects

User-defined projects do not use Workbench build support or pre-configured features. These projects can be anything and it is up to the user to organize and maintain the build system, target file system population, and so forth.

Wind River Linux Application Project

Wind River Linux application projects are developed on the host computer and deployed on the target. The Wind River Linux application projects dynamically generate build specs for Wind River Linux-supported board architectures and kernel and file system combinations. See *3.4 Creating a Project*, p.26 for more information on Wind River Linux application projects.

Wind River Linux Platform Project

Wind River Linux platform projects are developed on the Linux host computer and deployed on pre-defined targets. Platform projects can include prebuilt or customized kernels, and pre-defined or customized file systems.Workbench provides special support for Wind River platform projects including kernel and user space configuration tools, and multiple build targets. See *4.1 Wind River Linux Platform Projects*, p.47 for more details.

6.5 Projects and Project Structures

All individual projects of whatever type are self-contained units that have no inherent relationship with any other projects. The system is initially flat and unstructured. You can, however, construct hierarchies of project references within Workbench. These hierarchies will reflect inter-project dependencies and therefore also the build order.

When you attempt to create such hierarchies of references, this is validated by Workbench; that is, if a certain project type does not make sense as a subproject of some other project type, or even the same project type, such a reference will not be permitted.

6.5.1 Adding Subprojects to a Project

Workbench provides several ways to create a subproject/superproject structure:

- You can drag-and-drop project nodes in the Project Navigator. This is the easiest way to set up a structure among existing projects. Select the project that you want to make into a subproject and drag it onto its new parent (superproject).
- You can use the **Add as Project Reference** dialog. In the Project Navigator, right-click the project that you want to make into a subproject and choose **References > Add as Project Reference**, or select the project and choose **Project > Add as Project Reference**. In the dialog, you will see a list of valid superprojects; you can select more than one.
- You can use the **Project References** page in the **Properties** dialog. In the Project Navigator, right-click the project that you want to make into a *super*project and choose **Properties**, or select the project and choose **Project > Properties**. Then select **Project References**. In the dialog, you will see a list of projects; select the ones that you want to make into subprojects.

Subprojects appear as subnodes of their parents (superprojects).

Workbench validates subproject/superproject relationships based on project type and target operating system. It does not allow you to create certain combinations. In general, a user-defined project can be a subproject or superproject of any other project with a compatible target operating system.

Removing Subprojects

To undo a subproject/superproject relationship, use one of these methods:

- In the Project Navigator, right-click the subproject and choose **References > Remove Project Reference**, or select the subproject and choose **Project > Remove Project Reference**.
- In the Project Navigator, right-click the superproject and choose **Properties**, or select the superproject and choose **Project > Properties**. Then select **Project References** and unselect the subprojects that you want to disassociate from their current parent.

6.6 Project-Specific Execution Environments

If your development process requires you to maintain different build and external tool execution environments for each of your projects, Workbench allows you to create a **project.properties** file within each project that define which tools, tool versions, and environment variable settings should be used for each one.

You can share the **project.properties** file with your team to maintain consistency, and you should add it to source control along with your other project files.

1. In the Project Navigator, right-click your project, then select **New > File**.
2. In the New File dialog, create or link to a **project.properties** file:
   - To create a new file, type **project.properties** in the **File name** field, then click **Finish**.
   - To link to an existing **project.properties** file, click **Advanced**, then select **Link to file in the file system**. Type in the path or navigate to the file, then click **Finish**.

NOTE: When sharing files with a team, or accessing them from a common location, it is advisable to use a path variable instead of an absolute path since each team member's path to the location may be different.

To define a path variable, click **Variables**, then click **New**, then type a **Name** for the path variable and the location it represents (or click **File** or **Folder** to navigate to it). Click **OK** twice to return to the New File dialog; your path variable and its resolved location appear at the bottom of the dialog. Click **Finish**.

3. The new **project.properties** file appears under your project in the Project Navigator, and opens in the Editor so you can add or edit its content.
4. The **project.properties** file uses the same syntax as other properties files used by **wrenv** (such as **install.properties** and **package.properties**).

   As an example of what you can specify, the following lines define an extension to the **workbench** package, adding the variable **PROJECT_CONTEXT** to the environment with the value of **set**:

```
projectprops.name=projectprops
        projectprops.type=extension
        projectprops.subtype=projectprops
        projectprops.version=0.0.1
        projectprops.compatible=[workbench,,2.6]
        projectprops.eval.01=export PROJECT_CONTEXT=set
```

5. To find the information you will need to create your own extension, find the project's platform by looking to the right of your project's name in the Project Navigator (for example, it might say Wind River Linux *version*).
6. Open your *installDir*/**install.properties** file and look for the section listing the platform information. This is the type, subtype, and other information you must include to identify the package you want to extend.
7. Workbench uses the project properties specified in this file whenever you build a target in the project. To apply the project properties from the command line, include the **-i** option for both the **project.properties** and **install.properties** files when invoking **wrenv**.

```
-i installDir/install.properties -i installDir/workspace/myproject/project.properties
```

In both cases, the environment for **make** is altered to include the environment and properties specified in the file.

6.6.1 Using a project.properties file with a Shell

The **Project > Open Shell** menu item also takes advantage of the settings you specified in the **project.properties** file. This action is context sensitive, so the opened shell sets the environment of the selected project's platform, plus the extension from the properties file if one exists. If you did not have a project selected before opening the shell, a dialog appears with the environments you can choose.

6.6.2 Limitations When Using project.properties Files

A **project.properties** file creates an *extension* to a project, meaning it can include new tools, define variables, and specify versions. But it cannot exclude things that are already included, or overwrite existing variables, or undo PATH settings that are set within the properties you are trying to extend.

You cannot use a **project.properties** file with Native Application projects because they do not have a package associated with them and so cannot be extended.

7 Creating User-Defined Projects

7.1 Introduction

User-Defined Projects assume that you are responsible for setting up and maintaining your own build system, file system population, and so on. The user interface provides support for the following:

- You can configure the build command used to launch your build utility; this allows you to start builds from the Workbench GUI. You can also configure different rules for building, rebuilding and cleaning the project.
- You can create build targets in the Project Navigator that reflect rules in your makefiles; this allows you to select and build any of your make rules directly from the Project Navigator.
- Build output is captured to the Build Console.

7.2 Creating and Maintaining Makefiles

When you create a User-Defined project, Workbench checks the root location of the project's resources for the existence of a file named **Makefile**[2]. If it does not exist, Workbench creates a skeleton makefile with a default **all** rule and a **clean**. This allows you to use the **Build Project**, **Rebuild Project**, and **Clean Project** menu commands, as well as preventing the generation of build errors. You are responsible for maintaining this Makefile, and you can write any other rules into this file at any time.

If you base your User-Defined project on an existing project, the makefile of that project will be copied to the new project and will overwrite a makefile in the new project's location. If necessary, you can change the name of the new project's makefile using the **-f** make option to avoid overwriting an existing makefile.

7.3 Creating a User-Defined Project

Before creating the project, see the general comments on projects and project creation in *6. Projects Overview*.

1. Create a User-Defined project by selecting **File > New > Wind River Workbench Project**. The New Wind River Workbench Project wizard appears.
2. Select a target operating system, then click **Next**.
3. From the **Build type** drop-down list, select **User-Defined**. Click **Next**.
4. Type a name for your project.
5. Decide where to create your project:

   Create project in workspace
   Leave this selected if you want to create the project under the current workspace directory.

2. If you specified a different filename in the New Project wizard's **Build Command** field using the **-f** make option, which can include a relative or absolute path to a subdirectory, Workbench checks for the file you specified.

Create project at external location
Select this option, click **Browse**, then navigate to a different location if you want to create the project outside the workspace.

Create project in workspace with content at external location
Select this option, click **Browse**, then navigate to your source location if you do not want to mix project files with your sources, or copy sources into your workspace.

6. When you are ready, click **Finish**. Your project appears in the Project Navigator.

7.4 Configuring a User-Defined Project

Once you have created your project, you can configure its build targets, build specs, and build macros.

For general details about build properties, see *11.4 Accessing Build Properties*, p.136 or press the help key for your host.

1. To access build properties for your project, right-click it in the Project Navigator and select **Properties**.
2. From the **Properties** dialog, click **Build Properties**.

NOTE: Build tools and build paths cannot be configured for User-defined projects.

7.4.1 Configuring Build Support

Use this tab to configure build support for your project.

1. Build support is enabled by default. Click **Disabled** to disable it, and click **User-defined build** to re-enable it.
2. If necessary, edit the default build command.
3. Specify whether received build targets should be passed to the next level.
4. Specify when Workbench should refresh the project after a build.

Because a refresh of the entire project can take some time (depending on its size) Workbench does not do it by default. You may choose to refresh the current project, the current folder, the current folder and its subfolders, or nothing at all. This option applies to all build runs of the project.

7.4.2 Configuring Build Targets

Use this tab to configure make rules and define custom build targets for your project.

1. Type the desired make rules into the fields in the **Make rules** section. These rules are run when you select the corresponding options from the **Project** menu or when you right-click your project in the Project Navigator and select them from the context menu.

   The **Build Folder** and **Clean Folder** options are available when you select a folder in the Project Navigator.

2. To define a custom build target, click **New**. The **New Custom Build Target** dialog opens.

3. Type in a name for your build target, then type in the make rule or external command that Workbench should execute. You can also click **Variables** and add a context-sensitive variable to the make rule or command.

   The variables represented in the **Select Variable** dialog are context-sensitive, and depend on the current selection in the Project Navigator. For variables that contain a file-specific component, the corresponding target is only enabled when a file is selected and the variable can be evaluated. Build targets without file-specific components are always enabled.

4. Choose the type, whether it is a **Rule** or a **Command**.

5. Choose a refresh option for the build target, specifying whether Workbench should use the project setting, refresh the current folder or project, or do nothing. Click **OK** to close the dialog.

6. Edit a build target's options by clicking **Edit** or **Rename**. You can also edit the options (except name) by clicking in the column itself.

7. Continue configuring your project or click **OK** to close the Build Properties.

Once you have defined a build target, it is available when you right-click a project and select **Build Options**. The build targets are inherited by each folder within the project, eliminating the need to define the same build targets in each individual folder.

7.4.3 Configuring Build Specs

Use this tab to define and import build specs.

1. To define a new build spec for your project, click **New** and enter a build spec name. Click **OK**. If this is the first build spec for this project, it automatically appears in the **Default build spec** and **Active build spec** fields. Once you have defined more than one, you can choose a different default and active spec from the drop-down list.
2. To reset build properties to their default settings or import build settings from another project, click **Import** and select the source of the build settings.
3. Decide whether to clear build setting overrides, then click **Finish**.

NOTE: The Debug mode option is not available for User-defined builds, as this has an effect only on build tool-specific fields, which are not available for User-defined projects.

7.4.4 Configuring Build Macros

Use this tab to define global and build spec-specific macros that are added to the build command when executing builds.

Defining Global Macros

To define a global build macro for your project, click **New** next to **Build macro definitions**, then enter a **Name** and **Value** for the macro. Click **OK**.

You can define and use global build macros even if you don't define any build specs.

Defining Build Spec-Specific Macros

To define a build spec-specific macro, click **New** next to **Build spec-specific settings**, enter a **Name** for the macro, leave the **Value** blank, then click **OK**.

To define the value, select the macro, select the build spec for which the value should be applied, then click **Edit** and enter the **New value** and click **OK**.

The macro will always be appended to the build command when a build is launched, and the value will be set according to the active build spec, including empty values.

For example, if the build command is **make --no-print-directory** and the macro is **TEST_SPEC**, you can define values to be used with different build specs:

spec 1: Value = **spec1Val**
spec 2: Value = **spec2Val**
spec 3: Value =

The resulting build commands are as follows:

build command for spec 1: **make --no-print-directory TEST_SPEC=spec1Val**
build command for spec 2: **make --no-print-directory TEST_SPEC=spec2Val**
build command for spec 3: **make --no-print-directory TEST_SPEC=**

8 Native Application Projects

8.1 Introduction

Use a *Native Application project* for C/C++ applications developed for your host environment.

Workbench provides build and static analysis support for native GNU 2.9*x*, GNU 3.*x*, and Microsoft development utilities (assembler, compiler[3], linker, archiver) though you must acquire and install these utilities as they are not distributed with Workbench.

There is no debugger integration for native application projects in Workbench, so you must acquire and use the appropriate native tools for debugging as well.

3. Workbench supports the MinGW, Cygnus, and MS DevStudio compilers. Compilers for native development are distributed with Wind River Platforms, but not with Workbench.

8.2 Creating a Native Application Project

Before creating the project, please take a look at the general comments on projects and project creation in *6. Projects Overview*.

To create a Native Application project, proceed as follows.

1. Choose **File > New > Native Application Project**.

   The **New Native Application Project** wizard appears. If you have multiple operating systems installed, you are asked to select a target operating system. If you see this field, select a version from the drop-down list and click **Next**.

2. Enter a **Project name** and **Location**.

   If you choose **Create project in workspace** (default) the project will be created under the current workspace directory. If you choose to **Create project at external location**, you can navigate to a location outside the workspace (see also *6.2 Workspace and Project Location*, p.88 and *6.3 Creating New Projects*, p.89).

   The project appears in the Project Navigator. To see the project location, right-click on the project and select **Properties**, then select the **Info** node of the Properties dialog.

   When you are ready, click **Next**.

3. If you have created other projects, you are asked to define the project structure (the super- and subproject context) for the project you are creating.

   The text beside the **Link to superproject** check box refers to whatever project is currently highlighted in the Project Navigator (if you do not see this check box, no valid project is highlighted). If you select the check box, this will be the superproject of the project you are currently creating.

   The check boxes in the **Referenced subprojects** list represent the remaining projects in the workspace that can be validly referenced as subprojects by the project you are currently creating.

   After project creation, you can change the project structure in the Project Navigator using drag-and-drop.

   When you are ready, click **Next**.

NOTE: All settings in the following wizard pages are build related. You can therefore verify/modify them after project creation in the Build Properties node of the project's Properties, see *11. Building Projects*.

4. A Native Application project is a predefined project type that uses Workbench **Build support**, so you can only select either this, or no build support at all. If you are creating a project because you want to browse symbol information only and you are not interested in building it, you could also disable build support.

   The **Build command** specifies the make tool command line.

   Build output passing: If the project is a subproject in a tree, its own objects (implicit targets) as well as the explicit targets of its subprojects, can be passed on to be linked into the build targets of projects that are further up in the hierarchy.

   When you are ready, click **Next**.

5. **Build Specs**: The list of available build specs will always be available. By checkmarking individual specs, you enable them for the current project, which means that you will, in normal day to day work, only see relevant (enabled) specs in the user interface, rather than the whole list.

   If you are working on a Windows application, you would normally enable the **msvc_native** build spec, and disable the **gnu-native** build specs. If you are working on a Linux or Solaris native application, you would normally enable the GNU tool version you are using, and disable all others.

   The **Debug Mode** checkbox specifies wether or not the build output includes debug information.

   When you are ready, click **Next**.

6. **Build Target**: The **Build target name** is the same as the project name by default. You can change the name if necessary, but if you delete the contents of the field, no target will be created.

   Build tool: For a Native Application project you can select:

   - **Linker**: This is the default selection. The linker produces a a *BuildTargetName* (**.exe** for Windows native projects) executable file.

     The **Linker** output product cannot be passed up to superprojects, although the current project's own, unlinked object files can, as can any output products received from projects further down in the hierarchy (see step 4. above).

   - **Librarian**: This is the default selection if you specified that the project is to be linked into a project structure as a subproject. The Librarian produces a *TargetName***.a** (or **.lib** for Windows native projects) archive file**.**

The **Librarian** output product can be passed up to superprojects, as can the current project's own, unlinked object files, as well as any output products received from projects further down in the hierarchy (see step 4. above).

7. When you are ready, you can review your settings using the **Back** button or click **Finish**.

   The Native Application project is created and appears in the Project Navigator, either at the root level, or linked into a project tree, depending on your selection in step 3. above.

8.3 Application Code for a Native Application Project

After project creation you have the infrastructure for a Native Application project, but often no actual application code. If you are writing code from the beginning, you can add new files to a project. If you already have source code files, you will want to import these to the project. For more information please refer to *Importing Resources*, p.108, and *Adding New Files to Projects*, p.109.

9 Working in the Project Navigator

9.1 Introduction

The Project Navigator is your main graphical interface for working with projects. You use the Project Navigator to create, open, close, modify, and build projects. You also use it to add or import application code, to import, or customize build specifications, and to access your version control system.

Various filters, sorting mechanisms, and viewing options help to make project management and navigation more efficient. Use the arrow at the top-right of the Project Navigator to open a drop-down menu of these options.

9.2 Creating Projects

Creating projects is discussed in general under *6.3 Creating New Projects*, p.89. Specific descriptions for creating individual project types are provided in the other chapters in *Part III. Projects*.

9.3 Adding Application Code to Projects

After creating a project, you have the infrastructure for a given project type, but no actual application code. If you already have source code files, you will want to import these to the project.

Importing Resources

You can import various types of existing resources to (newly created) projects by choosing **File > Import**.

For details about the entries in the Import File dialog, see *Wind River Workbench User Interface Reference: Import File Dialog*.

NOTE: Importing resources creates a link to the location of those resources; it does not copy them into your workspace.

Later, if you want to delete a project, check the path in the **Confirm Project Delete** dialog very carefully when deciding whether to choose **Also delete contents under** '*path*' or **Do not delete contents**—choosing to delete the project contents may delete your original sources or the contents of a project in a different workspace, rather than the project in your current workspace.

Adding New Files to Projects

To add a new file to a project, choose **File > New > File**.

You are asked to **Enter or select the parent folder**, and to supply **Filename**.

For a description of the **Advanced** button, and what it reveals, press **F1** and select **New file wizard**.

9.4 Opening and Closing Projects

You can open or close a project by selecting it in the tree and choosing **Project > Open** (if it is currently closed), or **Project > Close** (if it is currently open). You can also use the corresponding commands on the Project Navigator's right-click context menu.

Closing a Project

- The icon changes to its closed state (by default grayed) and the tree collapses.
- All project member files that are open in the editor are closed.
- All subprojects that are linked exclusively to the closed project are closed as well. However, subprojects that are shared among multiple projects remain open as long as a parent project is still open, but can be closed explicitly at any time.
- In general, closed projects are excluded from all actions such as symbol information queries, and from workspace or project structure builds (that is, if a parent project of a closed subproject gets built).
- It is not possible to manipulate closed projects. You cannot add, delete, move, or rename resources, nor can you modify properties. The only possible modification is to delete the project itself.
- Closed projects require less memory.

9.5 Scoping and Navigation

There are a number of strategies and Workbench features that can help you manage the projects in your workspace, whether you are working with multiple projects related to a single software system, or multiple unrelated software systems.

- **Close projects**

  If you expect to be working in a different context (under a different root project) for a while, you can select the root project you are leaving, and right-click **Close Project**.

  If you close your root projects when you stop working on them, you will see just the symbols and resources for the project on which you are currently working (see also *Closing a Project*, p.109).

- **Go into a project**

  If you want to see, for example, the contents of only one software system in the Project Navigator, select its root project node and right-click **Go Into**. You can then use the navigation arrows at the top of the Project Navigator to go back out of the project you are in, or to navigate history views.

- **Open a project in a new window**

  If you expect to be switching back and forth between software systems (or other contexts) at short intervals, and you do not want to change your current configuration of open editors and layout of other views, you can open the other software system's root project in a new window (right-click **Open in New Window**). This essentially does the same as **Go Into** (see **Go Into a Project**), except that a new window is opened, thereby leaving your current Workbench layout intact.

- **Open a new window**

  You can open a new window by choosing **Window > New Window**. This opens a new window to the same workspace, leaving your current Workbench window layout intact while you work on some other context in the new window.

- **Use Working Sets**

  Using working sets lets you set the scope for all sorts of queries. You can, for example, create working sets for each of your different software systems, or any constellation of projects, and then scope the displayed Project Navigator content (and other query requests) using the pull-down at the top-right of the Project Navigator.

  To create a Working Set, from the drop-down menu, choose **Select Working Set**. In the dialog that appears, click **New**, then, in the next dialog, specify the **Resource** type.

  In the next dialog select, for example, a software-system root project and give the working set a name. When you click **Finish**, your new working set will appear in the **Select Working Set** dialog's list of available working sets.

  After the first time you select a working set in the **Select Working Set** dialog, the working set is inserted into the Project Navigator's drop-down menu, so that you can directly access it from there.

- **Use the Navigate Menu**

  For day-to-day work, there is generally no absolute need to see the contents of your software systems as presented in the Project Navigator.

  Using the **Navigate > Open Resource** (to navigate files) and **Navigate > Open Symbol** (to jump straight to a symbol definition) may often prove to be the most convenient and efficient way to navigate within, or among, systems.

9.6 Moving, Copying, and Deleting Resources and Nodes

The resources you see in the Project Navigator are normally displayed in their logical, as opposed to physical, configuration (see *6.5 Projects and Project Structures*, p.92). Depending on the type of resource (file, project folder) or purely logical element (target node) you are manipulating, different things will happen. The following section briefly summarizes what is meant by resource types and logical nodes.

9.6.1 Resources and Logical Nodes

Resources is a collective term for the *projects*, *folders*, and *files* that exist in Workbench.

There are three basic types of resources:

- *Files*

  Equivalent to files as you see them in the file system.

- *Folders*

  Equivalent to directories on a file system. In Workbench, folders are contained in projects or other folders. Folders can contain files and other folders.

- *Projects*

  Contain folders and files. Projects are used for builds, version management, sharing, and resource organization. Like folders, projects map to directories in the file system. When you create a project, you specify a location for it in the file system.

  When a project is open, the structure of the project can be changed and you will see the contents. A discussion of closed projects is provided under *Closing a Project*, p.109.

Logical nodes is a collective term for nodes in the Project Navigator that provide structural information or access points for project-specific tools.

- *Subprojects*

  A project is a resource in the root position. A project that references a superproject is, however, a logical entity; it is a reference only, not necessarily (or even normally) a physical subdirectory of the superproject's directory in the file system.

- *Build Target Nodes*

  These are purely logical nodes to associate the project's build output with the project.

- *Tool Access Nodes*

  These allow access to project-specific configuration tools.

9.6.2 Manipulating Files

Individual files, for example source code files, can be copied, moved, or deleted. These are physical manipulations. For example, if you hold down **CTRL** while you drag-and-drop a source file from one project to another, you will create a physical copy, and editing one copy will have no effect on the other.

9.6.3 Manipulating Project Nodes

Although copying, moving, or deleting project nodes are undertaken with the same commands you would use for normal files, the results are somewhat different because a project folder is a semi-logical entity. That is, a project is a normal resource in the root position. A project that is referenced as a subnode is, however, a logical entity; it is a reference only, not a physical instance.

If you copy/paste (or hold down **CTRL** while you drag-and-drop) a project folder node to a new location in the project editor (for example, under some other project node to be used as a subproject there) all that happens is that a reference to one and the same project is inserted. This means that if you modify the properties of one instance of the subproject node, all other instances (which are really only references) are also modified. One such property would be, for example, the project name. If you rename the project node in one context, it will also be renamed in all other contexts.

Moving and (Un-)Referencing Project Nodes

If you drag-and-drop a project folder, you are making a logical, structural change. However, if you select a project folder node and right-click **Move**, you will be asked to enter (browse for) a new file system location. All the files associated with the current project will then be physically moved to the location you select, without any visible change in the Project Navigator (you can verify the new location in the **Project Properties**).

When you drag-and-drop a project node, you are actually performing the equivalent of right-click **Add as Reference** or, if you have selected a subproject, also right-click **Remove Reference**. These commands open a dialog allowing you to either have the currently selected project reference other projects as a subproject, or, in the **Remove Reference** dialog, to remove the currently selected project from its structural (logical) context as a subproject, in which case it will be moved to the root level as a standalone project in the Project Navigator.

Deleting Project Nodes

To delete a subproject, which might potentially be linked into any number of other project structures, you first have to either unlink (right-click it and press **Delete**) all instances of the subproject, or get a *flat* view of your workspace. To do this, open the drop-down list at the top-right of the Project Navigator's toolbar and choose **Hide > Project Structure**. This hides the logical project organization and provides a flat view with a single instance of the (sub)project that you can then delete by pressing **Delete** again.

When you delete a project you are asked whether or not you want to delete the contents. If you choose not to delete the contents, the only thing that happens is that the project (and all its files) are no longer visible in the workspace; there are no file system changes.

9.6.4 Manipulating Target Nodes

Target nodes cannot be copied or moved. These are purely logical nodes that make no sense anywhere except in the projects for which they were created. If you copy or move entire projects, however, the target nodes and generated build-targets beneath them are also copied.

Deleting Target Nodes

Deleting a target node also removes the convenience node that represents the generated, physically existing build-target. However, the physically existing build-target (if built) is not deleted from the disk.

The convenience node referred to above, lets you see at a glance whether the target has been built or not, even if you have uncluttered your view in the Project Navigator by hiding build resources (in the drop-down menu at the top-right choose **Hide > Build Resources**) and/or collapsing the actual target node. If you have collapsed the node, the **+** sign will indicate that the build-target exists).

PART IV

Development

10 Navigating and Editing

10.1 Introduction

Workbench navigation views allow seamless cross-file navigation based on symbol information. For example, if you know the name of a function, you can navigate to that function without worrying about which file it is in. You can do this either from an editing context, or starting from the *The Symbol Browser*, p.119. On the other hand, if you prefer navigating within and between files, you can use the *The File Navigator*, p.120.

Static analysis is the parsing and analysis of source code symbol information. This information is used to provide code editing assistance features such as multi-language syntax highlighting, code completion, parameter hints, definition/declaration navigation for files within your projects.

Apart from the things you see directly in the Editor, static analysis also provides the data for code comprehension and navigation features such as include browsing, call trees, as well as resolving includes to provide the compiler with include search paths.

NOTE: Syntax highlighting is provided for file system files that you open in the Editor, but no other static analysis features are available for files that are outside your projects.

10.2 Wind River Workbench Context Navigation

Various filters are available on each tool's local toolbar. Hover the mouse over the buttons to see a tooltip describing what these buttons do. At the top-right, a pull-down menu provides additional filters, including *working sets* (if you have defined any). An active working set is marked by a bullet next to its name in the pull-down menu.

Generally, you will want to navigate to symbols, or analyze symbol-related information, from an Editor context. The entry points are:

- The right-click context menu of a symbol
- Keyboard shortcuts that act on the current selection in the Editor:

  F3 — Jump between associated code, for example, between definition/declaration or function definition/call. There is no navigation from workspace files to external files, i.e. files outside your projects.

  F4 — Open the type hierarchy of the current selection (see *Type Hierarchy View*, p.120).

  CTRL+ALT+H — Open the call tree of the current selection (see *Wind River Workbench User Interface Reference: Call Tree View*).

  CTRL+I — Open the include browser to view the includes of the current selection (see *Include Browser*, p.121).

- Keyboard shortcuts that open dialogs from which you can access symbols in any of your projects:

  SHIFT+F3 — Display the **Open Symbol** dialog.

SHIFT+F4 — Display the **Open Type Hierarchy** dialog.

ALT+SHIFT+H — Display the **Open Call Tree** dialog.

CTRL+SHIFT+R — Displays the **Open Resource** dialog.

These options are also available from the **Navigate** toolbar menu.

The Symbol Browser

By default, the Symbol Browser is a tab in the left pane of the main window, together with the Project Navigator.

Use the Symbol Browser for global navigation. Because the Symbol Browser presents a flat list of all the symbols in all the open projects in your workspace, you might want to constrain the list by using *Working Sets*. You can configure and select working sets using the Project Navigator's local pull-down menu.

In addition, very large symbol loads can cause delays of up to several minutes while Workbench loads the symbols. Loading smaller batches of symbols can decrease this delay. Specify the size of the symbol batch using the Preferences dialog. For more information, see *Wind River Workbench User Interface Reference: Debug View*.

Text Filtering

The **Name Filter** field at the top of the view provides match-as-you-type filtering. The field also supports wild cards: type a question mark (**?**) to match any single letter; type an asterisk (*****) to match any number of arbitrary letters. Selecting **Hide Matching** next to the **Name Filter** field inverts the filter you entered in the field, so you see only those entries that do not match your search criteria.

For a guide to the icons in the Symbol Browser, see *Wind River Workbench User Interface Reference: Symbol Browser View*.

The Outline View

The Outline view is to the right of the currently active Editor, and shows symbols in the currently active file.

Use the Outline view to sort, filter, and navigate the symbols in the context of the file in the currently active Editor, as well as to navigate out of the current file context by following call and reference relationships.

For a guide to the icons in the Outline view, see *Wind River Workbench User Interface Reference: Outline View*.

The File Navigator

If you have never used the File Navigator, you can open it by choosing **Window > Show View > Other**. In the dialog that opens, select **Wind River Workbench > File Navigator** and click **OK**. After the first time you open the File Navigator, a shortcut appears directly under the **Window > Show View** menu. By default, the File Navigator appears as a tab at the left of the Wind River Workbench window, along with the Project Navigator and the Symbol Browser.

The File Navigator presents a flat list of all the files in the open projects in your workspace, so you can constrain the list by using *Working Sets*. You can configure and select working sets using the File Navigator's local pull-down menu.

The left column of the File Navigator shows the file name, and is active; double-clicking on a file name opens the file in the Editor, and right-clicking on a file allows you to compile the file and build the project, among other tasks. The right column displays the project path location of the file.

The **File Filter** field at the top of the view works in the same way as the **Name Filter** field in the Symbol Browser, see *The Symbol Browser*, p.119.

Type Hierarchy View

Use the Type Hierarchy view to see hierarchical **typedef** and type-member information.

To open the Type Hierarchy view:

- Right-click a symbol in the Editor, Outline, or Symbol Browser view and select **Type Hierarchy view**.
- Click the toolbar button on the main toolbar.
- Select **Navigate > Open Type Hierarchy**.

For more information, see the *Wind River Workbench User Interface Reference: Type Hierarchy View*.

Include Browser

By default, the Include Browser appears as a tab at the bottom-right.

To open the Include Browser:

- Right-click a symbol in the Editor, Outline, or Symbol Browser view and select **Open Include Browser**.
- Right-click a file in the File Navigator or the Project Navigator and select **Include Browser**.
- Select **Navigate > Open Include Browser**.

Use the Include Browser to see which file includes, or is included by, the file you are examining. Use the buttons on the Include Browser's local toolbar to toggle between showing include and included-by relationships. Double-click on an included file in the Include Browser to open the file in the Editor at the include statement.

10.3 The Editor

The Editor is your primary view for editing and debugging source code. The Editor is language-aware, and can parse C, C++, Ada, and Assembler files. Many Editor features are configurable in the Preferences (see *Wind River Workbench User Interface Reference: Editor*).

Code Templates

The Editor uses templates to extend code assist (shortcut **CTRL+SPACE**) by inserting recurring patterns of text.

In the case of source code, common patterns are **for** loops, **if** statements and comment blocks. Those patterns can be parameterized with variable placeholders that are resolved and substituted when the template is inserted into the text. Unresolved variables can be link-edited after inserting the template, which means that the first unresolved variable is selected, and all occurrences of this variable are edited simultaneously when you enter the correct text.

An example template might look like the following:

```
for (int ${var} = 0; ${var} < ${max}; ++${var}) {
    ${cursor}
}
```

Provided Templates

Workbench provides the following templates. Auto-insert is turned on by default.

Name	Description
author	author name
catch	catch block
class	class declaration
comment	default multiline comment
do	do while statement
else	else block
elseif	else if block
for	for loop
for	for loop with temporary variable
if	if statement
ifelse	if else statement
main	main method
namespace	namespace declaration
new	create new object
stderr	print to standard error
stdout	print to standard output
switch	switch case statement
try	try catch block
using	using a namespace

Many template options are configurable in the Preferences (see *Wind River Workbench User Interface Reference: Editor*).

10.3.1 Configuring a Custom Editor

Workbench has a single global mapping between file types and associated editors. This mapping dictates which editor will be opened when you double-click a file in the Project Navigator, or when the debugger stops in a given file.

Configuring the custom editor through file associations will cause the correct editor to be opened, and the instruction pointer to be painted in the editor gutter. To view and modify the mappings, go to **Window > Preferences > General > Editors > File Associations**.

NOTE: Some debugger features require additional configuration; for details, see *18.2.1 Configuring Debug Settings for a Custom Editor*, p.229.

10.4 Search and Replace: The Retriever

The Retriever is a fast, index-based global text search/replace tool. The scope of a search can be anything from a single file to all open projects in the workspace. You can query for normal text strings, or regular expressions. Matches can be filtered according to location context (for example, show only matches occurring in comments). Text can be globally or individually replaced, and restored if necessary. You can create working sets from matched files, and you can save and reload existing queries.

Initiating Text Retrieval

Text retrieval is context sensitive to text selected in the Editor. If no text is selected in the Editor, an empty instance of the Retriever opens. If text is selected in the Editor, the retrieval is immediately initiated according to the criteria currently defined in the Retriever's **Find** tab.

To open the Retriever, or to initiate a context sensitive search, use:

- the keyboard shortcut **CTRL+2**.
- right-click in the Editor and choose **Retrieve in Files**.
- from the global menu, choose **Search > Retrieve in Files**.

- Click the **Retriever** tab in the lower panel of the Workbench window, where the Retriever appears by default.

For more information, see the *Wind River Workbench User Interface Reference: Retriever*.

10.5 Static Analysis

Editing, navigating, and code comprehension rely on static analysis parsing of source code.

You can enable static analysis in two ways: automatically, by leaving **Enable Static Analysis** selected when you create your project, or manually, by right-clicking your project in the Project Navigator and selecting **Static Analysis > Enable**.

NOTE: If this is your first use of static analysis, you may need to select **Static Analysis > Activate Plug-in** before you can access other static analysis features.

For information about global and project-specific preferences, see the *Wind River Workbench User Interface Reference: Static Analysis Preferences*.

Sharing Static Analysis Data with a Team

Static analysis of a large project can take quite a bit of time, so once you have parsed the source code of your project, you can share the generated data with your team members using your group's source control tool.

To share generated data with your team:

1. In the Project Navigator, right-click a project then select **Static Analysis > Export Shared Team Data**[1]. The **Team Data Export Options** dialog appears, where you can set export options and specify how to resolve file version differences in other workspaces (for details about this dialog, see *Wind River User Interface Reference: Static Analysis Preferences*). Click **Finish**. Workbench exports the data to the file system location you specified.

1. If this is your first use of Static Analysis, select **Static Analysis > Activate Plug-in**, then select **Static Analysis > Export Shared Team Data**.

2. Using your team's source control tool, make the generated data available to other team members (for example, by checking it into ClearCase). After that, when the project is imported into another workspace, Workbench will use the shared data instead of parsing the project.
3. Changes to the source-code are not propagated to the shared data automatically, they are stored local to the workspace. You must export the data again to make these changes available to team-members.
4. Once you have made local changes to a project in your workspace, Workbench uses that local data in preference to the shared data. To abandon your local changes and go back to using shared data, right-click a project and select **Import Shared Team Data**. Workbench launches a wizard that removes your local data in favor of the shared data.

Comparing Local Data with Shared Team Data

The tricky part about working with shared team data is to figure out which of the resources in your workspace have been changed relative to the shared data. In other words, you need a technique to compare the version of the file in your workspace with the one used to generate the shared data. Workbench comes with several mechanisms to do that:

- **Compare Timestamps**: In some setups (e.g. ClearCase dynamic views) a file of a certain version will have the same timestamp no matter in which workspace it appears. This makes it easy to check the version of a file against the shared data. This technique is preferable because it is fast, but many setups do not allow it (such as CVS and ClearCase snapshot views).
- **Use team data for all read-only files**: In other setups, all files in a workspace that are in sync with the repository of the source control tool are read-only. As a heuristic, we can use the shared data for all files that are read-only. The fact that a file of the workspace is in sync with the repository does not actually guarantee that the file version is the same as the one used to generate the shared data. If you update the shared data regularly though, the heuristic will be good.
- **Use checksums as a fallback**: If accessing the source code is reasonably fast and doesn't cause too much drag on the system (unlike in a ClearCase dynamic view) computing and comparing checksums for each file is a reasonable approach.

When you export shared data, you can specify in the dialog which of the techniques described above can be used by team members using the shared data. **Compare time stamps** and **Use team data for all files** will always be available to them in the import wizard, but you can disable **Use team data for all files** and **Compare checksums** by unselecting the corresponding check boxes in the export wizard.

When you import a project with shared data into your workspace, Workbench will choose the best available comparison method.

Team-Shared Exclusion Filter

The project property page, accessible by right-clicking a project and selecting **Static Analysis > Edit Exclusion Filter** now allows you to share filters with your team.

The filters are organized into a tree with two root nodes: one for shared filters and one for workspace private ones. You can convert shared filters into workspace-private ones if you would like to edit them, or you can share a private filter with your team if you find it particularly useful. For details about exclusion filters, see *Wind River User Interface Reference: Static Analysis Preferences*.

11 Building Projects

11.1 Introduction

The process of building in Workbench starts during project creation, when you select a build type for your projects, folders, and build targets. Individual build settings can be changed later, and in some cases you can switch from a managed build to a user-defined or disabled build, but if you want Workbench to manage your builds, you must select **Managed Build** in the New Project wizard.

Workbench offers several levels of build support:

Managed Build
: Workbench provides two types of managed build support—Standard and Flexible—for all project types except User-Defined projects.

Workbench provides default build settings (that you can change as necessary), creates makefiles, and controls all phases of the build.

There are advantages to each type of managed build, depending on many things including how much control you need over your build output and what your source tree looks like.

Table 11-1 shows a comparison of standard and flexible managed build features.

User-Defined build

With **User-Defined** builds, you are responsible for setting up and maintaining your own build system and Makefiles, but Workbench does provide minimal build support.

- It allows you to configure the build command used to launch your build utility, so you can start builds from the Workbench GUI.
- You can create build targets in the Project Navigator that reflect rules in your makefiles, so you can select and build any of your make rules directly from the Project Navigator.
- Workbench displays build output in the Build Console.

Disabled build

If you select Disabled build for a project or folder, Workbench provides no build support at all. This is useful for projects or folders that contain, for example, only header or documentation files that do not need to be built.

Disabling the build for such folders or projects improves performance both during makefile generation as well as during the build run itself.

NOTE: You cannot change from a lower level of build support to a managed build once the project is created. If you later want Workbench to manage your build, create a new project with the desired type of managed build support, either on top of the existing sources, or import your sources into it.

Table 11-1 **Comparison of Standard and Flexible Managed Build Features**

Standard Managed Build	Flexible Managed Build
Build structure parallels the file system structure.	Build structure can be defined independently from the file system structure.
Build order is determined by the project and folder hierarchies as displayed in the Project Navigator.	Build order is flexible, and you can customize settings per build tool and build target.
Project contains folders and files. Project metadata and build information is stored in the source code location.	Project contains folders, files, and information about how the build targets of the project are built (stored in the **.wrproject** file). No information is stored in **.wrfolder** files in the source code location.
Project can contain multiple build targets, but the build options are the same regardless of the build target the file is built into.	Project can contain multiple build targets. You can add the same file to multiple build targets, and set specific options depending on which build target the file is built into.
Build target can contain only files within the project.	Build targets can contain any file, folder, project, or other build target in the workspace, including linked resources. Virtual folders allow you to group objects from different sources and apply build settings to them.
Build target contains all contents of included folders.	Folders and files can be excluded from the build target using regular expressions.
Workbench creates one Makefile per folder with all build specs. Makefiles are based on data you enter at project creation time, or later in the Build Properties dialog.	Workbench creates one Makefile per build spec for the whole project.
Leveling chain is **project > folder > file**.	Leveling chain is **project > build target > folder > file**.

Table 11-1 **Comparison of Standard and Flexible Managed Build Features** (cont'd)

Standard Managed Build	Flexible Managed Build
Workbench generates include search paths for header files that are visible in the workspace.	Same
Build output is displayed in the Build Console.	Same

11.2 Configuring Workbench Managed Builds

The process of configuring Workbench managed builds differs significantly depending on whether you selected a standard or a flexible managed build.

11.2.1 Configuring Standard Managed Builds

Standard managed builds have not changed from previous versions of Workbench. When you select **Standard** in the New Project wizard, your project is created and contains a preliminary build target in addition to the usual project files.

To create the build target, right-click your project in the Project Navigator and select **Build Project**.

11.2.2 Configuring Flexible Managed Builds

When you select **Flexible**, your projects are created in the same way and also contain the usual project files, but you must create your build targets manually.

Adding Build Targets to Flexible Managed Builds

Once your project is created, you will see a **Build Targets** node inside it.

1. To add a build target to your project, right-click the **Build Targets** node and select **New Build Target**. The **New Build Target** dialog appears.
2. By default the **Build target name** and **Binary output name**[1] are the same as the project name, but if you are going to create multiple build targets you will want to type in more descriptive names. Choose the appropriate **Build tool** for your project, then click **Next**. The **Edit Content** dialog appears.
3. To display files, folders, and other build targets from outside your current project, select **Show all projects**. If you have created a **Working Set**, you can restrict the display by selecting it from the pull-down list.
4. You can add contents to your build target in several ways:
   a. You can select specific files, folders, projects, or other build targets in the left column and click **Add**. What you can add depends on the build tool you use; for example, you cannot add an executable build target to another build target.

      When choosing folders or projects, they can be added "flat" or with recursive content.

      – Clicking **Add** creates a "flat" structure, meaning that Workbench adds the exact items you choose and skips any subfolders and files.

      – Clicking **Add Recursive** creates a structure that includes subfolders and files.

NOTE: Adding linked resources to a build target may cause problems within a team if the linked resources are added using an absolute path instead of a variable.

To define a path variable, select **Window > Preferences > General > Workspace > Linked Resources**, click **New**, then enter a variable name and location.

1. Your build targets must have unique names, but you can use the same **Binary output name** for each one. This allows you to deliver an output file with the same name in multiple configurations. Workbench adds a build tool-appropriate file extension to the name you type, so do not include the file extension in this field.

b. You can create a virtual folder within your build target by clicking **Add Virtual Folder**, typing a descriptive name in the dialog, and clicking **OK**. Virtual folders allow you to group objects within the build target so you can apply the same build settings to them; they also provide a way to add files with the same name from different locations.

   i. To add contents to your virtual folder, right-click it in the Project Navigator and select **Edit Content**.

   ii. Select content as described in step a above, and click **Finish**.

5. To adjust the order of the build target contents, select items in the right column and click **Up**, **Down**, or **Remove**.

NOTE: Folders appear in the specified place in the list, but the files within them are added alphabetically.

6. When you have configured your build target, click **Finish**. It appears in the Project Navigator under the **Build Targets** node of your project.

Modifying Build Targets

There are several ways to modify your build target once it has been created.

Editing Content

To add additional items, adjust the order, or make any other changes to your build target, right-click it in the Project Navigator and select **Edit Content**. The **Edit Content** dialog appears, with the build target content displayed in the right column. Adjust the contents as necessary, then click **Finish**.

Renaming Build Targets and Virtual Folders

To rename your build target or virtual folder, select it in the Project Navigator, press **F2**, and type a new name.

Copying Build Targets

To copy a build target, right-click the build target and select **Copy**, then right-click the destination project's **Build Targets** node and select **Paste** (if you are pasting back into the original project, type a unique name for the new build target).

This is useful for setting up the same build targets in multiple projects with different project types (for example, a library for a native application and a downloadable kernel module will have the same contents but different flags).

NOTE: The build target and its contents are copied, but any overridden attributes are not.

Removing Content

To remove an item from the build target, right-click it in the Project Navigator and select **Remove from Build Target**, or just select it and press **Delete**.

Depending on the item you selected, the menu item may change to **Exclude from Build Target** if the item cannot be deleted (for example, recursive content cannot be deleted). Pressing **Delete** also reinstates an item by removing the exclusion.

Excluding Content

To exclude a specific item from the build target that was included recursively, right-click it in the Project Navigator and select **Exclude from Build Target**.

You can also use regular expressions to exclude groups of items.

1. To add a pattern to the excludes list, right-click a folder in the build target, then select **Properties**, then select the **Excludes** tab.
2. Click **Add File** to define a pattern to exclude specific files or file types. For example, type ***_test.c** to exclude any file named *filename*_**test.c**.

   You can include additional parts of the path to better define the file you want to exclude; for example, type **lib/standard_test.c** to exclude that specific file.
3. Click **Add Folder** to define a pattern to exclude folders within specific folders. For example, type ***/lib/*_test.c** to exclude any file located in a folder named **lib** and named *filename*_**test.c**.

Dragging and Dropping Content

To modify build target contents without opening the **Specify Content** dialog, you can drag and drop items in the Project Navigator.

- You can drop resources onto build target nodes or virtual folders to add them to a build. Workbench checks the validity of the action and reports errors if the move is not allowed. Workbench also asks you if the resource should be added “flat” or recursively.

- You can reorder build target contents by dragging and dropping an item on the same level.
- You can drop a build target node onto other build targets to add the first build target as a reference (for example, dropping a library onto an executable, or dropping an executable onto an executable). Workbench checks to make sure it is a valid operation before allowing you to complete the action.

NOTE: If your build target contains projects or folders, any files you add to them later will be automatically added to the build target as well. So you do not need to manually update your build target in this case.

Leveling Attributes

The leveling of build-specific settings in flexible managed builds is significantly different from the leveling of standard managed build projects. The leveling chain for flexible managed build projects is shown below.

Project > Target > Folder > File
Project > Target > Folder > Subfolder > File
Project > Target > Virtual folder > File
Project > Target > Virtual folder > Folder >
Project > Target > File

The folder level here is related to folders underneath a build target, as described in *Adding Build Targets to Flexible Managed Builds*, p.131.

The information that can be leveled is equivalent to the current implementation of standard managed build projects, plus additional information so that you can enable files to be built on a per build-spec basis (standard managed build allows this only on folder level).

You can now configure the build target with specific settings for all build tools on a build target level (for example, you can set compiler options for the source files related to that build target).

Understanding Flexible Managed Build Output

The output of a flexible managed build is significantly different from the output of a standard managed build.

Workbench does not create build redirection directories for each folder, as the objects might be built differently when building them for specific targets. Instead, Workbench creates a build-specific redirection directory, which you can configure on the **Build Properties** > **Build Paths** tab, underneath the project root directory.

In this redirection directory there is a directory for each build-target, and inside those are directories named **Debug** or **NonDebug** depending on the debug mode you chose for the build. Workbench generates the output files according to the structure you defined in the build target, and deposits them in the debug-mode directory.

In general, the build output is structured like this:

Project directory
Project dir/build specific redirection dir
Project dir/build specific redirection dir/target dir
Project dir/build specific redirection dir/target dir/debug mode dir
Project dir/build specific redirection dir/target dir/debug mode dir/binary output file of the build target

All objects belonging to the build target are stored within an additional **Objects** subfolder:

Project dir/build specific redirection dir/target dir/debug mode dir/Objects/structure of object files

Example Build Target and Build Output Structure

To understand how the build target structure influences the build output, below is an example of a project source tree.

```
proj1/
proj1/a.c
proj1/b.c
proj1/folder1/c.c
proj1/folder1/d.c
```

Target1 contains these two items:

```
a.c
folder1/*.c
```

Target2 contains these two items:

```
b.c
d.c
```

Configuring the project to use **spec1** as the active build spec, naming the redirection directory **spec1**, and turning debug-mode **on** produces the output structure seen below.

```
proj1/spec1/Target1/Debug/Target1.out
proj1/spec1/Target1/Debug/Objects/a.o
proj1/spec1/Target1/Debug/Objects/folder1/c.o
proj1/spec1/Target1/Debug/Objects/folder1/d.o

proj1/spec1/Target2/Debug/Target2.out
proj1/spec1/Target2/Debug/Objects/b.o
proj1/spec1/Target2/Debug/Objects/d.o
```

11.3 Configuring User-Defined Builds

When you create a User-Defined project, you can configure the build command, make rules, build target name, and build tool (for more information, see *7. Creating User-Defined Projects*). To create the build target, right-click your project in the Project Navigator and select **Build Project**.

To update the build settings, right-click your project in the Project Navigator and select **Properties**, then select **Build Properties**.

See *Wind River Workbench User Interface Reference: Build Properties* for more information about the settings described on the build properties tabs.

11.4 Accessing Build Properties

There are two ways to set build properties: in the Workbench preferences, to be automatically applied to all new projects of a specific type, and manually, on an individual project, folder, or file basis. The properties displayed will differ depending on the type of node and the type of project you selected, as well as the type of build associated with the project.

For details, see *Wind River Workbench User Interface Reference: Build Properties*.

11.4.1 Workbench Global Build Properties

To access global build properties, select **Window > Preferences** and choose the **Build Properties** node.

This node allows you to select a project type, then set default build properties to be applied to all new projects of that type.

11.4.2 Project-specific Build Properties

To access build properties from the Project Navigator, right-click a project and select **Properties**. In the Properties dialog, select the **Build Properties** node.

The project-specific Build Properties node has tabs that are practically identical to the ones in the Workbench preferences, but these settings apply to an existing project that is selected in the Project Navigator.

11.4.3 Folder, File, and Build Target Properties

Folders, files, and build-targets inherit (reference) project build properties where these are appropriate and applicable. However, these properties can be overridden at the folder/file level. Inherited properties are displayed in blue typeface, overridden properties are displayed in black typeface.

Overridden settings are maintained in the **.wrproject** file (and also in **.wrfolder** files in standard managed builds). These files should therefore also be version controlled. Note that you can revert to the inherited settings by clicking the eraser button next to a field.

11.4.4 Multiple Target Operating Systems and Versions

If you installed Workbench for multiple target operating systems and/or versions, you can set a default target operating system/version for new projects in the Workbench Preferences, under **General > Target Operating Systems**.

For existing projects, you can verify the target operating system (version) by right-clicking the project in the Project Navigator, then selecting **Properties**, then **Project Info**.

NOTE: In most cases, it will *not* be possible to successfully migrate a project from one target operating system or version to another simply by switching the selected **Target Operating System and Version**.

In the Project Navigator (and elsewhere), the target operating system and version are displayed next to the project name by default. You can toggle the display of this information in the Preferences, **General > Appearance > Label Decorations**, using the **Project Target Operating Systems** check box.

If you have multiple versions of the same operating system installed, the New Project wizard allows you to select which version to use when creating a new project.

11.5 Build Specs

A build spec is a group of build-related settings that lets you build the same project for different target architectures and/or different tool chains by simply switching from one build spec to another. Note that the architecture/tool chain associations are preconfigured examples; you can also create your own build specs (usually from copies of existing ones, using the **Copy** button.

It is important to remember that the build spec used when you build must match the target board.

11.6 Makefiles

The build system uses the build property settings to generate a self-contained makefile named **Makefile**.

- For standard managed builds, a Makefile is generated in each project and folder at each build run. This allows you to build individual folders, projects, and subtrees in a project structure.
- For flexible managed builds, only one Makefile is created per build spec.

By default makefiles are stored in project directories; if you specified an absolute **Redirection Root Directory** (see *Wind River Workbench User Interface Reference: Build Paths*), they are stored there, in subdirectories that match the project directory names.

The generated makefile is based on a template makefile named **.wrmakefile** that is copied over at project creation time. If you want to use custom make rules, enter these in **.wrmakefile**, *not* in **Makefile**, because this is regenerated for each build. The template makefile, **.wrmakefile**, references the generated macros in the placeholder **%IDE_GENERATED%**, so you can add custom rules either before or after this placeholder. You can also add ***.makefile** files to the project directory.

For other ways of setting custom rules, see *12.6 User-Defined Build-Targets in the Project Navigator*, p.151.

NOTE: If you configure your project for a remote build, the generated Makefile contains paths for remote locations rather than local ones. For more information about remote builds, see *12.9 Developing on Remote Hosts*, p.157.

11.6.1 Derived File Build Support

The Yacc Example

Workbench provides a sample project, **yacc_example**, that includes a makefile extension showing how you can implement derived file build support. It is based on **yacc** (Yet Another Compiler Compiler) which is not contained in the Workbench installation. To actually do a build of the example you need to have **yacc** or a compatible tool (like GNU's **bison**) installed on your system, and you should have extensive knowledge about make.

The makefile, **yacc.makefile**, demonstrates how a yacc compiler can be integrated with the managed build and contains information on how this works.

1. Create the example project by selecting **New > Project > Example > Native Sample Project > Yacc Demonstration Program**.
2. Right-click the **yacc_example** project folder, then select **New > Build Target**. The New Build Target dialog appears.
3. In the **Build target name** field, type **pre_build**.
4. From the **Build tool** drop-down list, select **(User-defined)**, then click **Finish** to create the build target.

5. In the Project Navigator, right-click **pre_build** and select **Build Target**. This will use the makefile extension **yacc.makefile** to compile the yacc source file to the corresponding C and header files. The build output appears in the Build Console.

NOTE: It is necessary to execute this build step prior to the project build, because the files generated by **yacc** will not be used by the managed build otherwise. This is due to the fact that the managed build generates the corresponding makefile before the build is started and all files that are part of the project at this time are taken into account.

6. When the build is finished, right-click the **yacc_example** folder and select **Build Project**.

Additional information on how you can extend the managed build is located in **yacc.makefile**. It makes use of the extensions provided in the makefile template .wrmakefile, which can also be adapted to specific needs.

General Approach

To implement derived file support for your own project, create a project-specific makefile called *name_of_your_choice***.makefile**. This file will automatically be used by the managed build and its make-rules will be executed on builds.

It is possible to include multiple ***.makefile** files in the project, but they are included in alphabetical order. So if multiple build steps must be done in a specific order, it is suggested that you use one ***.makefile** and specify the order of the tools to be called using appropriate make rules.

For example:

1. Execute a lex compiler.
2. Execute a yacc compiler (depending on lex output).
3. Execute a SQL C tool (depending on the yacc output).

Solution: (using the **generate_sources** make rule)

```
generate_sources :: do_lex do_yacc do_sql
do_lex:
    @...

do_yacc:
    @...

do_sql:
    @...
```

or

```
generate_sources :: $(LEX_GENERATED_SOURCES) $(YACC_GENERATED_SOURCES)
$(SQL_GENERATED_SOURCES)
```

Add appropriate rules like those shown in the file **yacc.makefile**.

12 Building: Use Cases

12.1 Introduction

This chapter suggests some of the ways you can go about completing various build-specific tasks in Wind River Workbench.

NOTE: This chapter discusses standard managed builds only. For details on flexible managed builds, refer to *11. Building Projects*.

12.2 Adding Compiler Flags

You may know the exact compiler flag you want to add, for example **-w**, and use the GUI to help you put it in the right place as shown in *Add a Compiler Flag by Hand*, p.144. You can also use the GUI to help you specify the correct compiler flag for what you want to do, as shown in *Add a Compiler Flag with GUI Assistance*, p.145.

NOTE: This section describes how to add and edit the compiler flags on specific projects. To make global build spec changes, see . However, for Wind River Linux projects, global changes are made in board templates so that the settings can be used by both the command line build system and Workbench, as described in *Wind River Linux Platforms User's Guide* (see also *12.7 Custom Build Specs for Wind River Linux Platform Projects*, p.153).

Add a Compiler Flag by Hand

If, for example, you are familiar with the GNU compiler command line and you just want to know *where* to enter the **-w** option.

1. In the Project Navigator, right-click an application project and select **Properties**.
2. In the Properties dialog box, select the **Build Properties** node.
3. In the **Build Properties** node, select the **Build Tools** tab.
4. In the **Build Tools** tab:
   - Set the **Build tool** to **C-compiler**
   - Set the **Active build spec** to, for example, **PENTIUM-gnu-native**.
   - In the field next to the **Tool Flags** button, append a space and **-w**.

     The contents of this, the **Tool Flags** field you have just modified, is expanded to the **%ToolFlags%** placeholder you see in the **Command** field above it. Because you entered the **-w** in the **Tool Flags** field, rather than the **Debug** or **Non Debug** mode fields, warnings will always be suppressed, rather than only in either **Debug** or **Non Debug** mode.

Add a Compiler Flag with GUI Assistance

If you are not familiar with the specific command line tool options that you want to use, the GUI may be able to help. For example:

1. In the Project Navigator, right-click an application project, and select **Properties**.
2. Click **Build Properties** and select the **Build Tools** tab.
3. In the **Build Tools** tab:
   - Set the **Build tool** to **C-compiler**
   - Set the **Active build spec** to, for example, **PENTIUM-gnu-native**.
   - We assumed you are unfamiliar with the GNU compiler options so, to open the GNU Compiler Options dialog box, click the **Tool Flags** button.
   - In the GNU Compiler Options dialog box, click your way down the navigation tree at the left of the dialog box and take a look at the available options.

     When you get to the **Compilation > Diagnostics** node, select the check box labelled **Suppress all compiler warnings**.

     Notice that **-w** now appears in the list of command line options at the right of the dialog box.

     Click **OK**.
4. Back in the **Build Tools** node of the Properties dialog box, you will see that the **-w** option you selected now appears in the field next to the **Tool Flags** button.

   The contents of this, the **Tool Flags** field, is expanded to the **%ToolFlags%** placeholder you see in the **Command** field above it.

12.3 Building Applications for Different Target Architectures

The target nodes under projects in the Project Navigator display, in blue, the name of the currently active build spec. You may want to switch build specs to build projects for different architectures.

If, for example, you want to build an application for testing on the localhost, and then build the same project to run on a real board, you would simply switch build specs as follows:

1. Right-click the project node and select **Build Options > Set Active Build Spec**.
2. In the dialog box that appears, select the build spec you want to change to and specify whether or not you want debug information.

   When you close the dialog box, you will notice that the label of the target node has changed. If you selected debug mode in the dialog box, the build spec name is suffixed with **_DEBUG**.
3. Build the project for the new architecture.

NOTE: To select the Active Build Spec directly from the Project Navigator, click the green checkmark in the Project Navigator toolbar.

12.4 Creating Library Build-Targets for Testing and Release

Assume you have a library that consists of the files **source1.c**, **source2.c**, and **test.c**. The file **test.c** implements a **main( )** function and is required exclusively for testing, and is not to be included in the release version of the library.

One way to handle this is to use different targets that are built with different tools as described below.

1. Create an application project to hold all the files mentioned above. Use this project type, because you will need to use both the **Linker** and the **Librarian** build tools later.

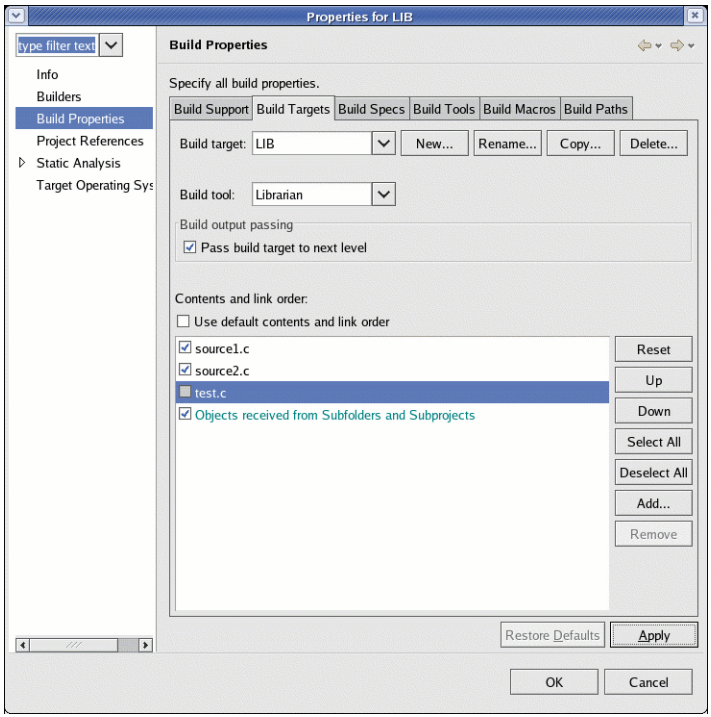
   In the first page of the project creation wizard, name the project, for example, **LIB** and click **Finish**. You will need to do some tweaking in the **Project's Properties** dialog box anyway, so you might as well do everything there.
2. Right-click the newly created **LIB** project, and select **Properties**. In the **Properties** dialog box, select the **Build Properties** node, then the **Build Targets** tab.

First create a build-target for the release version of your library.

- Change the **Build tool** to **Librarian**.
- Select **Pass build target to next level**.
- Clear the **Use default contents and link order** check box.
- Clear the check box next to **test.c**.
- Click **Apply**.

Figure 12-1 shows the results.

Figure 12-1 **Release Version of the Library**



3. Next, create a target for the test version of the library.
   - Click **New** then enter, for example, **LIB_test** in the dialog box that appears.

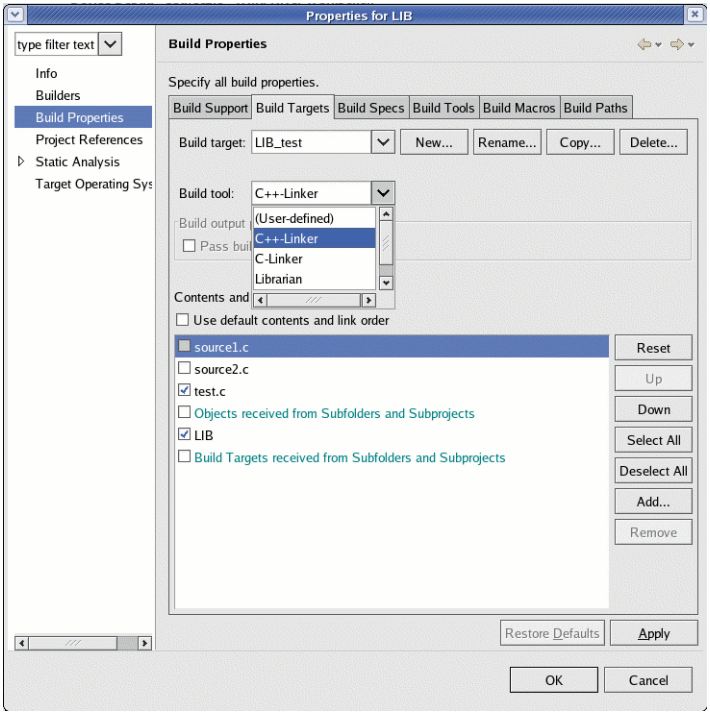
     Notice that the **Build Tool** is set to **C++-Linker**, this is because the C++ Linker is the default tool for Embedded Linux Application projects, and that the **LIB** (your previous build-target) has been added to the **Contents and link order** list.

– Clear the **Use default contents and link order** check box.

– In the **Contents and link order** list, select only the check boxes next to **LIB** and **test.c**; clear all other check boxes.

Figure 12-2 shows the results.

Figure 12-2 **Test Version of the Library**



After you close the **Properties** dialog box, there will be two new build-target nodes in the **LIB** project. If you build **LIB_test**, then **LIB** will automatically also be built if it is out of date.

12.5 Architecture-Specific Implementation of Functions

You can enable/disable build specs at the project as well as at the folder level. This allows architecture-specific implementation of functions within same project.

Figure 12-3 shows a simplified project tree with two subprojects, **arch 1** and **arch2**, that each use code that is specific to different target architectures. This is how projects could be set up to build a software target that requires the implementation of a function that is specific to different target boards, where only the active build spec in the topmost project has to be changed. The inner build spec relationships are outlined in Table 12-1.

Figure 12-3 **Simple Project Structure for Architecture-Specific Functions**

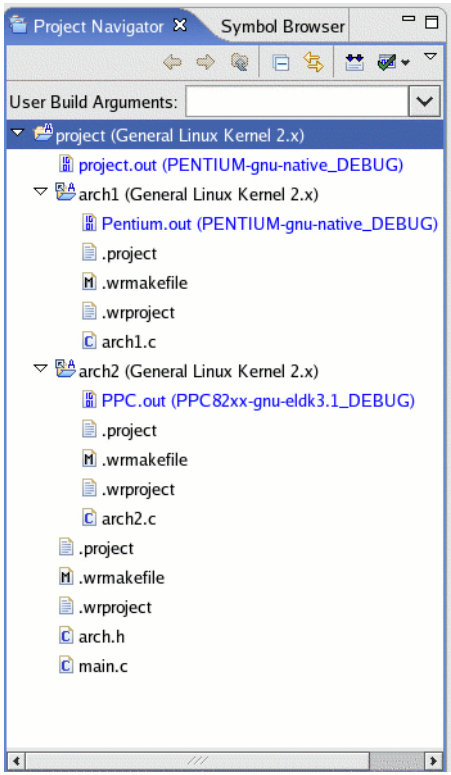


Table 12-1 **Project Content and Build Spec Configuration of the Structure in Figure 12-3**

Directories/Folders	Files	Enabled Build Specs
/project	**main.c, arch.h**	**PENTIUM-gnu-native** and **PPC82xx-gnu-eldk3.1**
/project/arch1	**arch1.c**	**PENTIUM-gnu-native** only
/project/arch2	**arch2.c**	**PPC82xx-gnu-eldk3.1** only

The function **int arch_specific (void)** is declared in **arch.h** and the file **arch1.c** implements **int arch_specific (void)** for **PENTIUM** (the only build spec enabled for the **arch1** project)**,** while the file **arch2.c** implements **int arch_specific (void)** for **PPC32** (the only build spec enabled for the **arch2** project).

If the active build spec for **project** is set to **PENTIUM-gnu-native**, the subproject **arch1** will be built, and its objects will be passed up to be linked into the **project** build-target. The **arch2** subproject will not be built, and its objects will not be passed up to be linked into the **project** build target because the **PENTIUM-gnu-native** build spec is not enabled for **arch2**.

The same applies if the **PPC82xx-gnu-eldk3.1** is the active build spec for **project**: the **arch2** subproject will be built, but the **arch1** subproject will not.

12.6 User-Defined Build-Targets in the Project Navigator

In the Project Navigator you can create custom build-targets that reflect rules in makefiles. This is especially useful if you have User-Defined projects, which are projects where the build is not managed by Workbench. However, you might also find this feature useful in other projects as well.

Custom Build-Targets in User-Defined Projects

You can define a custom build-target for a rule or rules in your Makefile. To do so:

1. Right-click a project or folder and select **New > Build Target**.
2. In the dialog box that appears, enter the rule(s) you want to create a target for. If you want to execute multiple rules, separate each one with a space.

   The names of the rule(s) you enter must exist in your makefile(s) to be executed when you build your new user-defined target.
3. Set the **Build tool** to **User-defined**.
4. Click **Finish**. The new build-target node appears under the project or folder you selected. The node icon has a superimposed **M** to identify it as a user-defined make rule.

   To execute the rule(s), right-click the new target node and select **Build Target**.

Custom Build-Targets in Workbench Managed Projects

First write the make rules you need into the **.wrmakefile** file in the project directory.

NOTE: For Wind River Platform projects, see *Custom Build Targets in Wind River Linux Platform Projects*, p.152.

1. Right-click a project or folder and select **New > Build Target**.
2. In the dialog box that appears, enter the rule name(s) you created in **.wrmakefile**. If you want to execute multiple rules, separate each one with a space.
3. Set the **Build tool** to **User-defined**.

4. Click **Finish**. The new build target node appears under the project or folder you selected. The node icon has a superimposed **M** to identify it as a user-defined rule.

   To execute the rule(s), right-click the new target node and select **Build Target**.

Custom Build Targets in Wind River Linux Platform Projects

To create a custom build target for a Wind River Linux Platform project, you edit **Makefile.wr**, not **.wrmakefile**. The contents of **.wrmakefile** (build properties and build targets) should be edited only through the project **Properties > Build Properties** dialogs.

NOTE: This example shows how to add a build target to invoke the uClibc configuration for PCD-based projects. This technique applies to bringing any other Wind River Linux or custom command line feature to Workbench Platform Projects.

First, write the rules you want add to **Makefile.wr**. For example, double-click **Makefile.wr** and add the following at the bottom of the file in the Editor view:

```
uclibc-config :
     xterm - e make -C $(DIST_DIR) uclibc.menuconfig
```

An **xterm** is used because this build rule, **menuconfig**, requires a shell that can support **xterm** commends, which is beyond the capabilities of the **Build Log** view within Workbench.

Note that at the top of **Makefile.wr** are the definitions **DIST_DIR** and **TARGET_DIR**. These provide the redirection to the Wind River Linux Platform project's content directory.

1. Right-click a project or folder and select **New > Build Target**.
2. In the dialog box that appears, enter the rule(s) you want to create a target for. If you want to execute multiple rules, separate each one with a space. For this example, add the build target **uclibc-config**.
3. Set the **Build tool** to **User-defined**.
4. Click **Finish**. The new build target node appears under the project or folder you selected. The node icon has a superimposed **M** to identify it as a user-defined rule.

   To execute the rule(s), right-click the new target node and select **Build Target**.

User Build Arguments

The User Build Argument view appears near the top of the Project Navigator. You can use the User Build Arguments view to execute any existing make rule, or override any macro, or anything else that is understood by make, at every build, regardless of what is being built. The view is toggled by choosing **User Build Arguments View** from the drop-down menu at the top right of the Project Navigator, or by clicking the button in the Project Navigator's toolbar.

If you make entries in the **User Build Arguments** view, the rule or rules, macro re-definitions, and so on, separated by a space, are appended to (and thus override) the existing makefile entries. This occurs on the fly at every build while the entries exist in the view.

12.7 Custom Build Specs for Wind River Linux Platform Projects

For Wind River Linux projects, the script file **wbblddefgen.tcl** translates the Wind River Linux templates to the Workbench build spec. This script includes a feature so that you can add custom build specs and have them automatically included in projects.

There are, however, cases where you may want to have build specs appear that are in addition to the ones derived from the templates. For example, Workbench and the Wind River Linux build system supports Thumb-based application builds, but this is not instantiated as a template because a kernel cannot be built with this configuration. This section describes how to add custom build specs, in this example Thumb-based, by adding build spec fragments to a special file.

NOTE: Wind River Linux does include pre-made build spec fragments. These can be copied directly from files that are named in the form **wbbldsample_***rootfs***-arm-thumb.txt.**

One use of this feature is to add build specs that explicitly add ARM Thumb support, as demonstrated here.

1. Go to the **wrlinux-1.3/scripts** directory:

   ```
   $ cd installDir/wrlinux-1.3/script
   ```

2. Create a file called **wbblddef_custom.txt**, and add the following contents:

```
BOARD=ARM-gnu-wrs-arm_versatile_926ejs-thumb-uclibc_small
TARGET_ARCH=arm
TARGET_LINUX_ARCH=arm
TARGET_TOOLCHAIN_ARCH=arm
TARGET_CPU_VARIANT=arm
TARGET_OS=linux-gnueabi
TARGET_COPT=
TARGET_CDEBUG=
TARGET_COMMON_CFLAGS=-mcpu=arm926ej-s -mthumb
LINUX_KERNEL_VERSION=2.6.14
TARGET_THE_PLATFORM=uclibc_small
TARGET_SUPPORTED_KERNEL=cgl small
TARGET_SUPPORTED_ROOTFS=glibc_full glibc_small uclibc_small
BOARD=END
```

NOTE: The line ending with **-mthumb** adds the Thumb support. Additional Thumb support examples are provided in *installDir***/wrlinux-1.3/scripts/wbbld-sample-***rootfs***-thumb.txt**.

3. Create a Wind River Linux project, and notice that a new build spec with the name of the **BOARD** field from the new file appears. (You must have installed ARM support to be able to build with this build spec.)

NOTE: The build specs for Wind River Linux application projects are of the form *CPU-toolchain-board-rootfs*.

The values used in the custom build spec describe the required values distilled from the templates (see *installDir***/wrlinux-1.3/wrlinux/templates/**). Sample board definitions can be found in the other **wbblddef_*.txt** files provided as part of the Application Developer support (see *installDir***/wrlinux-1.3/scripts**).

12.8 Stepping Through Assembly Code

You can view assembly code interleaved with the corresponding source code by clicking the **Toggle Assembly Stepping Mode** button as shown in Figure 12-4.

Figure 12-4 **Toggle Assembly and Source Code Viewing**

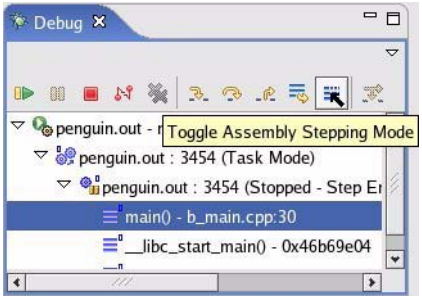
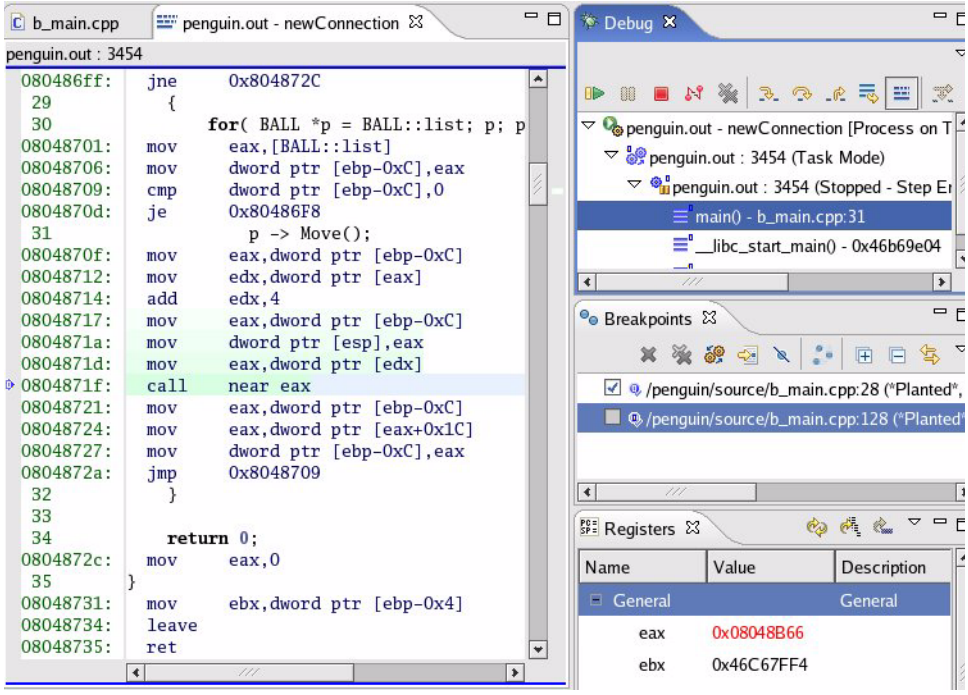


Figure 12-5 shows an example of mixed assembly and C++ source code (this is from the **Penguin** example program included with Workbench). The code has been single-stepped to the **call** instruction. Note that the previous steps are shown in lessening degrees of shading—this can help you see where you've been when stepping into and out of routines.

Figure 12-5 **Single Stepping Though Assembly Code**

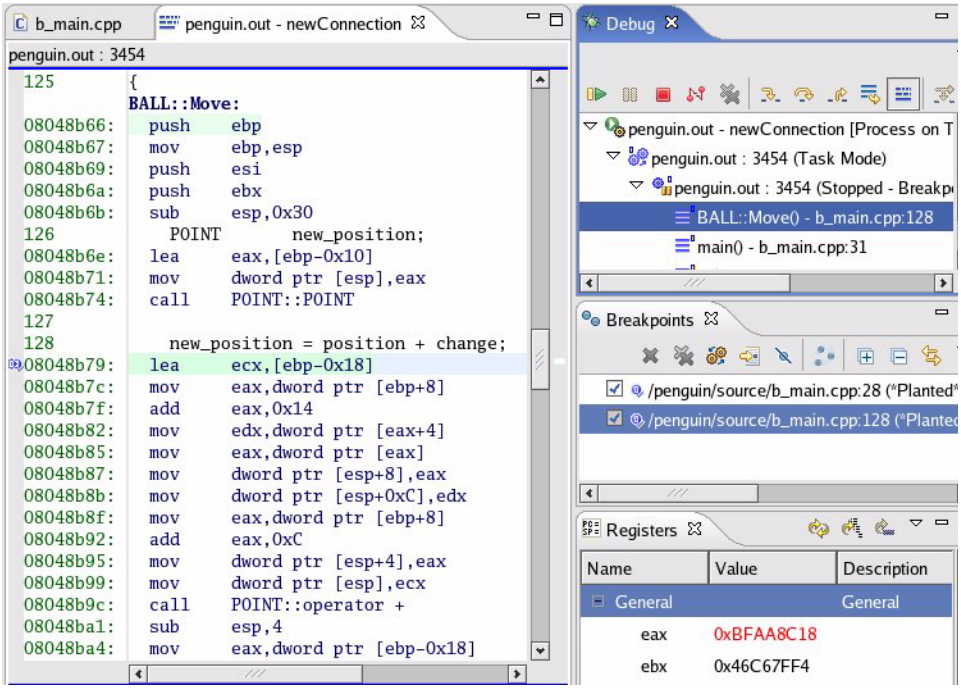


NOTE: The assembly code shown will differ depending on your target. Code produced for an Intel MPCBL0001 target is shown. For the ARM Versatile board, for example, the instruction corresponding to the **call** instruction is the **blx** instruction.

With the assembly language code visible you can step in and out of the inherited parent class methods for C++ classes as your C++ code is executing.

For example, in Figure 12-5, stepping into the call for the **p** object's **Move** method takes you into the **Move** method code for the parent class **BALL**, as shown in Figure 12-6.

Figure 12-6 **Example of C++ Assembly Code**

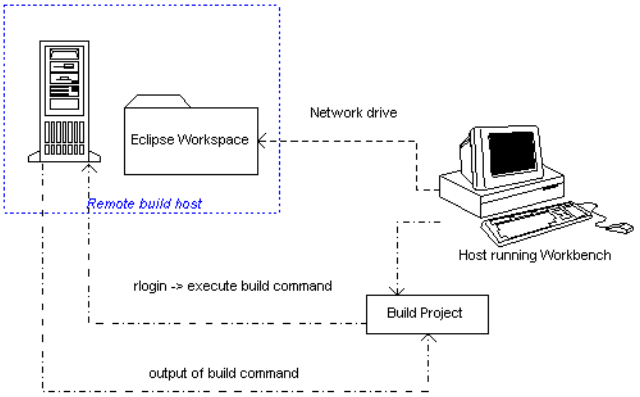


Also in Figure 12-6, you can see at the top (for line 126) the **call** to the parent class constructor **POINT::POINT**, which creates a new **POINT** object for the local variable **new_position**. Additionally, you can see the call (for line 128) to the parent class operator method **POINT::operator +**.

This is all hidden when debugging at the source code level.

12.9 Developing on Remote Hosts

The Workbench *remote build* feature allows you to develop, build and run your applications on a local host running Workbench, using a workspace that is located on a remote host as if it were on a local disk.



In the case of a managed build, Workbench generates the makefiles on the local machine running Workbench using a path mapping of the workspace root location, so that the generated makefiles will be correctly dumped for a build that is executed on the remote machine. When launching the build, a network connection (**rlogin** or **ssh**) is established to the build host, and the actual build command is executed there by using an intermediate script to allow you to set up the needed environment for the build process.

12.9.1 General Requirements

- The Workbench host tools and the tool chain must be installed on the remote machine.
- The workspace root directory has to be accessible from both machines.
- Only Eclipse projects located underneath the workspace root can be remotely built.
- An **rlogin** or **ssh** remote connection to the build machine must be possible.

12.9.2 Remote Build Scenarios

Local Windows, Remote UNIX:

The workspace root directory should be located on the remote UNIX host and mapped to a specific network drive on Windows. It may also be possible to locate the root directory on the Windows machine, but then there is the need to mount the Windows disk on the build host. This may lead to problems regarding permissions and performance, so a mapping of the workspace root-directory is definitely needed.

Local UNIX, Remote UNIX:

As it is possible to access the workspace root directory on both machines with the equivalent path (automount) it may be possible to skip the path mapping.

Local UNIX, Remote Windows:

This scenario is not supported, as you would need to execute the build command on Windows from a UNIX host.

12.9.3 Setting Up a Remote Environment

To set up your environment on the remote machine prior to a build or run, use the **Edit remote command script** button to include additional commands. It will open the file *workspaceDir*/**.metadata/.plugins/com.windriver.ide.core/remote_cmd.sh**.

For example, to extend the path before a build, add the highlighted lines to the default file:

```
#!/bin/sh

WORKSPACE_ROOT="%WorkspaceRoot%"
export WORKSPACE_ROOT
DISPLAY=%Display%
export DISPLAY

PATH=/MyTools/gmake_special/bin:$PATH
export PATH

cd $WORKSPACE_ROOT
```

```
cd "$1"
shift 1

exec "$@"
```

You can add any commands you need, but all commands must be in **sh** shell style.

12.9.4 Building Projects Remotely

1. Switch to a workspace that contains existing projects by selecting **File > Switch Workspace**. Type the path to the appropriate workspace, or click **Browse** and navigate to it.
2. In the Project Navigator, right-click a project and select **Build > Remote Connection**. The **Remote Connections** dialog box appears.
3. Click **Add** and type a descriptive name for this remote connection. Click **OK**.
4. In the **Connection Settings** fields, add the following information to create a remote connection:

   Connection Type
   Select **Rlogin** or **SSH**.

   Hostname
   The name of the build host (can also be an IP address).

   Username
   The username used to establish the connection (the remote user may differ from the local user).

   Remote Workspace Location
   The root directory of the workspace as seen on the remote host.

   Display (XServer)
   IP address of the machine where the output should be displayed.

   By clicking the **Advanced** button you can also access these fields:

   Password request string
   A string that will be recognized as a password request to prompt you for the password.

   Remember Password during Workbench sessions
   A switch to specify whether the password entered should be remembered during the current session. This is useful during a lengthy build/run session.

5. Click **Connect** to connect immediately. Remote connection settings are stored, and are specific to this workspace. They are not accessible from any other workspace.
6. The build is executed on the remote host, with the build output listed in the standard Workbench Build Console. The XServer (IP address listed in the Display field) is used whenever any type of X application is started, either during builds or runs.
7. To return to local development, select **Local Host** from the list of connections, then click **Connect**.

12.9.5 Running Applications Remotely

This section provides information about running native applications only, as running VxWorks projects remotely is handled differently.

Running native applications remotely is quite similar to running applications locally: a **Native Application** launch configuration must be created that defines the executable to be run, as well as remote execution settings for the launch. On the **Remote settings** tab are:

Remote Program
Enter the command that is used to launch the application. This may be useful for command-line applications that could then be launched within an xterm, for instance.

Remote Working Directory
This setting is optional, but if a remote working directory is given, it overrides the entry in the **Working Directory** field of the **Arguments** tab.

For remote runs, a new connection similar to the active connection will be established to allow control of Eclipse process handling, as the new remote process will be shown in the Debug view. The Remember password during Workbench sessions feature is very useful here.

Command-line application's output and input is redirected to the standard Eclipse console unless the application is started within an external process that creates a new window (such as **xterm**). The default for remote execution is a remote command like **xterm -e %Application%**, therefore a local XServer (like Exceed or Cygwin X) must be set up and running.

For more information about creating launch configurations, see *16. Launching Programs*.

12.9.6 RIogin Connection Description

The **rlogin** connection used in the Workbench remote build makes use of the standard **rlogin** protocol and ports. It establishes a connection on port 513 on the remote host, and the local port used must be in the range of 512 to 1023 per **rlogin** protocol convention.

On Windows the **rlogin** connection is implemented directly from within Workbench, so you do not need an existing **rlogin** client. The UNIX implementation is different, because for security reasons the local port (range: 512 to 1023) is restricted to root access, which cannot be granted from within Workbench. Therefore an external **rlogin** process is spawned using the command-line:

```
rlogin -l username hostname
```

rlogin on UNIX platforms makes use of **setUID root** to ensure that the needed root privileges are available.

The standard **rlogin** protocol doesn't support access to **stderr** of the remote connection, to all output is treated as **stdout**. Coloring in the Build Console of Workbench for **stderr** is therefore not available.

NOTE: On Linux the **rlogin** client and server daemon can be switched off per default. So if the machine is used as a Workbench (remote build client) host, the **rlogin** executable must be enabled (or built) and if the machine is acting as build server (remote build host) the **rlogin** daemon must be enabled. Details may be found in the system documentation of the host.

12.9.7 SSH Connection Description

The supported protocol is **SSH2**, and it establishes a connection on port 22 (the default **SSH** port).

Strict host key checking is disabled. Workbench does not use a known hosts file, so host key information is stored in memory, and you are not prompted if the host key changes.

Only password authentication is supported.

PART V

Target Management

13 Connecting to Targets

13.1 Introduction

A Workbench *target connection* runs on a host and manages communications between host tools and the target system itself. A connection must be configured and established before the host tools can interact with the target.

All host-side connection configuration work and connection-related activity is done in the Target Manager view. Connections are registered and made accessible to users by the Wind River Registry as described in *13.6 The Registry*, p.175. Connection data may be maintained in a central location and be shared between hosts with the use of remote registries as described in*13.6.2 Remote Registries*, p.176.

13.2 The Target Manager View

A connection to a Target Server must be defined and established before tools can communicate with a target system.

All host-side connection configuration work and connection-related activity is done in the Target Manager view. By default, the Target Manager view is on a tab at the bottom-left of Workbench. It is available in the Application Development perspective and in the Device Debug perspective. If the view is not visible, choose **Window > Show View > Target Manager** or, if you do not see it listed here, you will find it under **Window > Show View > Other**.

The most import tasks in the Target Manager view are:

- Defining new connections
- Connecting to targets
- Disconnecting from targets

Once you have connected to a target, more commands are enabled on the right-click context menu (see also *16. Launching Programs*).

13.3 Defining a New Connection

All connections types are defined from the Target Manager view (see *13.2 The Target Manager View*, p.166).

To open the **New Connection** wizard, either use the appropriate toolbar button, or right-click in the Target Manager and select **New > Connection**.

The first thing the **New Connection** wizard asks you to do is to select one of the following connection types:

- **Wind River Linux Application Core Dump Target Server Connection**—Create a new connection to a Linux 2.4 or 2.6 application core dump (see *19. Analyzing Core Files*).
- **Wind River Linux KGDB Connection**—Create a new kernel mode connection to a Linux 2.6.10 version kernel (see *5. Kernel Debugging (Kernel Mode)*).

- **Wind River Linux Target Server Connection for Linux User Mode**—Create a user mode connection for Linux 2.6 version kernels (see *3. Developing Applications (User Mode)*).
- **Wind River Linux Target Server Connection for Linux**—Create a dual mode connection for Linux 2.4 version kernels (see *F. Configuring Linux 2.4 Targets (Dual Mode)*).

Properties you set during the creation of a new connection using the **New Connection** wizard can be modified later by right-clicking the connection in the Target Manager and then selecting **Properties** from the context menu.

Note that if you change properties later, you will generally have to disconnect and reconnect in order for changes to take effect. To disconnect, right-click the connection and select **Disconnect**. To re-connect, right-click the connection and select **Connect**.

13.3.1 Target Server Connection Page

Depending on the type of connection you are making, the options you are presented and values you enter will differ. The following describes the values you will typically supply depending on the type of connection you are making. For details on all the options presented in the New Connection dialog, refer to *13.5 Connection Settings*, p.168.

Connecting to Core Dumps

Specify the name of the core dump file in **Core dump file**, and specify the path to the executable that caused the core dump in **Application image**.

For more information, refer to *19Analyzing Core Files*, p.247.

Connecting in Kernel Mode (KGDB Connection)

When configuring a kernel mode connection, you must first select the type of connection which will be either by a serial line (RS232) or over Ethernet. An additional option is provided in case you are using a terminal server. A final option is for custom situations for which you should refer to Wind River Support.

Specify an IP address and the path to your kernel symbol file (**Kernel image**).

For more information on connecting in Kernel mode and how to configure the target to support KGDB, refer to *5. Kernel Debugging (Kernel Mode)*.

Connecting in User Mode

Specify the target's IP address and root file system when making a user mode connection. The target must be running the usermode agent.

Refer to *3. Developing Applications (User Mode)* for details on configuring the usermode agent and on making a user mode connection.

Connecting in Dual Mode

Specify the target's IP address and root file system (for user mode) or kernel (for system mode) when making a dual mode connection to a Linux 2.4 kernel-based target. The kernel must be patched with the Wind River WDB agent.

Refer to *F. Configuring Linux 2.4 Targets (Dual Mode)* for information on configuring your target and host for dual mode connections.

13.4 Establishing a Connection

Once you have created your application projects and defined connections, you will want to run, test, and debug the projects on your target. To do this, you first need to connect to the target.

Typically, after configuring a new connection in the New Connection dialog, the connection to the target is attempted automatically. If you do not want the connection to be attempted at that time, unselect **Immediately connect to target if possible** on the final dialog screen.

Connect to and disconnect from targets in the Target Manager by selecting a connection node and then using either the appropriate toolbar button, or by right-clicking and selecting **Connect**. See *13.2 The Target Manager View*, p.166.

13.5 Connection Settings

When you have specified the type of connection to make, a new connection dialog box presents various parameters that you must configure.

Connection Template

For KGDB connections, select the type of connection that best describes how you are connecting to your target. Choose from the following connection types:

Linux KGDB via RS232
A serial cable connection.

Linux KGDB via Ethernet
An Ethernet connection.

Linux KGDB via Terminal Server
A connection in which you go through a terminal server to reach the target.

Custom Linux KGDB Connection
This template assumes no defaults.

Back End Settings

Back end
The **Back end** settings specify how a target server will communicate with a target.

Back end for a KGDB connection requires that you select from the following:

- **RS232** for a direct serial port connection (connection with a null-modem cable). You will then specify your serial line settings in the following dialog.
- **TCP** for connection to a terminal server with a direct serial connection.
- **UDP** for an Ethernet connection on the same subnet.
- **PIPE** is currently unsupported.

Back end for user mode and dual mode connections may be set to **wdbrpc** or **wdbproxy**. **wdbrpc** supports any kind of IP connection (for example, Ethernet). For dual mode connections, polled-mode Ethernet drivers are available in most cases to support system-mode debugging for the **wdbrpc** connection.

CAUTION: The target server *must* be configured with the same communication back end as the one built for the kernel image and used by the target agent.

CPU
Select **default from target** to have Workbench identify the target CPU for you, or select the target CPU from the drop-down menu.

Target Name/IP address
The **Name/IP Address** field specifies the network name or the IP address of the target hardware for networked targets. If you are using a serial port in dual mode, use this field to enter either **COM1** or **COM2**, or **/dev/ttyS0** or other port as appropriate.

Target File System and Kernel

Target File System and **Kernel** properties relate to the location of the target's root file system and the name and location of the target kernel. You specify the path to the kernel for KGDB and dual mode connections.

Root File System
Enter the full path of the target's root file system as it is on the host. This file system is typically NFS-exported from the host.

Kernel Image
This is the full path name including the kernel symbol file, for example, **/home/wbuser/WindRiver/workspace/new_prj/export/vmlinux-symbols**.

Advanced Options (KGDB Only)

Backend Communication Log File

Enter or browse to the location of a file to store log information.

Target Plugin Pass-through Options

(See *Configuring Target Reconnection Parameters*, p.77.)

Advanced Target Server Options

These options are passed to the **tgtsvr** program on the command line (dual mode and user mode only). Enter these options manually, or use the **Edit** button for GUI assisted editing.

Edit Target Server Options

Open the **Edit Target Server Options** window with the **Edit** button on the main wizard page. The options are subdivided into three tabbed groups: **Common**, **Memory**, and **Logging**.

The Common Tab

Timeout Options

Set how many seconds the target server will wait for an answer from the target before sending a timeout, how often to retry, and at what intervals it should ping the target, in seconds.

The Memory Tab

Memory Cache Size

To avoid excessive data-transfer transactions with the target, the target server maintains a cache on the host system. By default, this cache can grow up to a size of 1 MB.

A larger maximum cache size may be desirable if the memory pool used by host tools on the target is large, because transactions on memory outside the cache are far slower.

The Logging Tab

Options on the logging tab are used mainly for troubleshooting by Wind River support.

A maximum size can be specified for each of these files. If so, files are rewritten from the beginning when the maximum size is reached. If the file initially exists, it is deleted. This means that if the target server restarts (for example, due to a reboot), the log file will be reset.

For the WTX (Wind River Tool Exchange) log file, you can specify a *filter*, a regular expression that limits the type of information logged. In the absence of a filter, the log captures all WTX communication between host and target. Use this option in consultation with Customer Support.

Command Line

This will be filled-in based on the values you enter in this dialog box.

13.5.1 Target Operating System Settings

Clicking **Next** opens the Target Operating System Settings dialog box (Kernel mode only). Select the correct version of the Linux operating system, and specify the path to the kernel symbol file (for example, **/home/wbuser/WindRiver/workspace/new_prj/export/vmlinux-symbols**).

13.5.2 Object Path Mappings

Clicking **Next** will open the Object Path Mappings dialog box (Kernel mode and Dual mode).

Workbench uses *Object Path Mappings* in two ways:

- They allow the debugger to find symbol files for processes created on the target by creating a correspondence between a path on the target and the appropriate path on the host.
- Workbench also uses object path mappings to calculate target paths for processes that you want to launch by browsing to them with a host file system browser.

By default, the debug server attempts to load all of a module's symbols each time a module is loaded. In the rare cases where you want to download a module or start a process without loading the symbol file, unselect **Load module symbols to debug server automatically if possible**.

13.5.3 Specifying an Object File

If you are loading object code on the target using a custom loader, or associating symbols with already loaded modules, you can specify the object file that you want the debugger to use.

1. Right-click a container in the Target Manager, then select **Load/Add Symbols to Debug Server**. A dialog appears with your connection and core already filled in.
2. To add a new object file to the **Symbol Files and Order** list, click **Add**. Navigate to the file, then click **Open**.
3. In the **Symbol Load Options** section, select **Specify base start address** or **Specify start address for each section**.
4. When you are finished, click **OK**.

For more information about the fields in this dialog, click in the Target Manager, then press the help key for your host.

Pathname Prefix Mappings

This maps target path prefixes to host paths. Always use full paths, not relative paths.

For example, mapping **/** to **/opt/eldk/ppc_82xx/** tells the debugger that files accessible under **/** on the target can be found under **/opt/eldk/ppc_82xx/** on the host.

In most cases Workbench provides correct defaults. If necessary, click **Add** to add new mappings, or select existing mappings and click **Edit** to modify existing mappings.

NOTE: You cannot edit the supplied default mappings.

Basename Mappings

Use square brackets to enclose each mapping of target file basenames (left element) to host file basenames (right element), separated by a semi-colon (;). Mapping pairs (in square brackets) are separated by commas. You can use an asterisk (*) as a wildcard.

For example, if debug versions of files are identified by the extension ***.unstripped**, the mapping **[*;*.unstripped]** will ensure that the debugger loads *yourApp***.vxe.unstripped** when *yourApp***.vxe** is launched on the target.

13.5.4 Target State Refresh Page

Since retrieving status information from the target leads to considerable target traffic, this page allows you to configure how often and under what conditions the information displayed in the Target Manager is refreshed.

These settings can be changed later by right-clicking the target connection and selecting **Refresh Properties**.

Available CPU(s) on Target Board

Workbench can correctly identify the target CPU. In rare cases, a close variant might be misidentified, so you can manually set the CPU here.

Initial Target State Query and Settings

Specify whether Workbench should query the target on connect, on stopped events, and/or on running events. You can select all options if you like.

Target State Refresh Settings

Specify whether Workbench should refresh the target state only when you manually choose to do so, or if (and how often) the display should be refreshed automatically.

Listen to execution context life-cycle events

Specify whether Workbench should listen for life-cycle events or not.

13.5.5 Connection Summary Page (Target Server Connection)

This page proposes a unique **Connection name**, which you can modify if you wish, and displays a **Summary** of name and path mappings for review; to modify these mappings, use the **Back** button.

Shared

This option serves a dual purpose:

- When you define a target connection configuration, this connection is normally only visible for your user-id. If you define it as **Shared**, other users can also see the configuration in your registry, provided that they connect to your registry (by adding it as a remote registry on their computer, see *13.6.2 Remote Registries*, p.176).

- Normally, when you disconnect a target connection, the target server (and simulator) are killed because they are no longer needed. In a connection that is flagged as **Shared**, however, they are left running so that other users can connect to them. In other words, you can flag a connection as shared if you want to keep the target server (and simulator) running after you disconnect or exit Workbench.

Immediately connect to target if possible
If you do not want to connect to the target immediately, you can connect to the target later using one of the ways described in *18. Debugging Projects*.

If you have applications ready to run using the connection(s) you have just created, please see *16. Launching Programs*.

13.6 The Registry

The Wind River Registry is a database of target servers, boards, ports, and other items used by Workbench to communicate with targets. For details about the registry, see the **wtxregd** and **wtxreg** entries in *Wind River Host Tools API Reference* in the online Help.

Before any target connections have been defined, the default registry—which runs on the local host—appears as a single node in the Target Manager. (Under Linux, the default registry is a target-server connection for Linux user mode.) Additional registries can be established on remote hosts.

Registries serve a number of purposes:

- The registry stores target connection configuration data. Once you have defined a connection, this information is persistently stored across sessions and is accessible from other computers.

  You can also share connection configuration data that is stored in the registry. This allows easy access to targets that have already been defined by other team members.

NOTE: Having connection configuration data does not yet mean that the target is actually connected.

- The registry keeps track of the currently running target servers and manages access to them.
- Workbench needs the registry to detect and launch target servers.

  If Workbench does not detect a running default registry at start-up, it launches one. After quitting, the registry is kept running in case it is needed by other tools.

13.6.1 Launching the Registry

To launch the default registry, open the **Target** menu or right-click in the Target Manager and select **Launch Default Registry**.

NOTE: These menu items are only available if the registry is not running, and the default registry host is identical to the local host.

The registry stores its internal data in the file *installDir***/.wind/wtxregd.***hostname*. If this file is not writable on launch, the registry attempts to write to **/var/tmp/wtxregd.***hostname* instead. If this file is also not writable, the registry cannot start and an error message appears.

13.6.2 Remote Registries

If you have multiple target boards being used by multiple users, it makes sense to maintain connection data in a central place (the remote registry) that is accessible to everybody on the team. This saves everyone from having to remember communications parameters such as IP addresses and other settings for every board that they might need to use.

Creating a Remote Registry

You might want to create a new *master registry* on a networked remote host that is accessible to everybody. To do so:

1. Workbench needs to be installed and the registry needs to be running on the remote host. The easiest way to launch the registry is to start and quit Workbench. However, you can also launch the **wtxregd** program from the command line.

(For more information about **wtxregd**, see
Help > Help Contents > Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference.)

2. To use the remote registry on another host, right-click in the Target Manager, (see *13.2 The Target Manager View*, p.166), then select **New > Registry** from the context menu.

3. In the dialog that appears, enter either the host name or the IP address of the remote host.

   Workbench immediately attempts to connect to the remote registry. If the host is invalid, or if no registry is identified on the remote host, this information is displayed in the Target Manager. A valid connection will display the registry in the Target Manager and any active connections will be shown. Connect to the target just as you would to a target in your local registry.

13.6.3 Shutting Down the Registry

Because other tools use the registry, it is not automatically shut down when you quit Workbench. Before updating or uninstalling Workbench (or other products that use the registry), it is advisable to shut down the registry so that the new one starts with a fresh database. To shut down the registry:

- On Windows, right-click the registry button in the system tray, and choose **Shutdown**.
- On Linux and UNIX, execute **wtxregd stop**, or manually kill the **wtxregd** process.

If you want to migrate your existing registry database and all of your existing connection configurations to the new version, make a backup of the registry data file *installDir*/**.wind/wtxregd.***hostname* and copy it to the corresponding new product installation location.

13.6.4 Changing the Default Registry

Normally, the default registry runs on the local computer. You can change this if you want to force a default remote registry (see *13.6.2 Remote Registries*, p.176). To do this on Linux and UNIX, modify the **WIND_REGISTRY** environment variable. To do this on Windows, under the Windows Registry **HKEY_LOCAL_MACHINE** node, modify the field **Software\Wind River Systems\Wtx***N.N***\WIND_REGISTRY**.

14 Connecting with USB

14.1 Introduction

You can make a USB connection between a Workbench host and a target that supports the USB driver. Once you have established the connection, you can use it to debug applications just as you would with an Ethernet connection.

14.2 Configuring a Target for USB Connection

The way you configure your target depends on the host platform.

Target Configuration for a Linux Kernel 2.6 Host

The target kernel must support the USB device driver (USB Ethernet gadget). It may be installed as a module or it may be built into the kernel (this is the default for Wind River Linux kernels).

Determining Driver Status

To see if the USB Ethernet gadget is installed as a kernel module, examine the **lsmod** command output. To see if the gadget is built into your kernel, search the **dmesg** output for something similar to the following:

```
usb0: Ethernet Gadget, version: Equinox 2004
usb0: using pxa27x_udc, OUT ep3out-bulk IN ep2in-bulk STATUS ep1in-int
usb0: MAC 9a:8c:21:b0:9f:42
usb0: HOST MAC 26:06:24:43:81:a0
```

Your MAC and HOST MAC address will be different.

If **lsmod** shows the module installed or **dmesg** shows the gadget driver built in, you can proceed to *Configuring the USB Interface*, p.180.

Adding Driver Support

To add USB gadget driver support, build it into the kernel or add it as a module with **Device Drivers > USB support > USB Gadget Support > Support for USB Gadgets** using **make linux.xconfig** or the Wind River platform's **Kernel Configuration** node.

To install the kernel module, enter:

```
# modprobe g_ether
```

Configuring the USB Interface

With the kernel correctly configured and running, configure the **usb0** interface, for example:

```
# ifconfig usb0 10.0.0.150
```

where **10.0.0.150** is the IP address assigned to the target.

Target Configuration for a Linux Kernel 2.4 Host

The target kernel must support the **CDCEther** driver. This may be built into the kernel or installed as a module.

Determining Driver Status

To determine if your kernel already supports the **CDCEther** driver, examine the **lsmod** and **dmesg** output for **CDCEther**. If it appears in either output, you can proceed to configuring the USB interface.

Adding Driver Support

Configure **CDCEther** support into the kernel or create the **CDCEther.o** module with **USB support >USB Network Adaptors** using **make linux.xconfig**.

To install the kernel module:

```
# modprobe CDCEther
```

Configuring the USB Interface

With the kernel correctly configured and running, configure the **usb0** interface, for example:

```
# ifconfig usb0 10.0.0.150
```

where **10.0.0.150** is the IP address assigned to the target.

Target Configuration for a Windows Host

Reconfigure your kernel to enable RNDIS support as follows.

1. $ **cd dist**
2. $ **make linux.xconfig**
3. Go to **Device Drivers>USB Support>USB Gadget Support** and select **RNDIS Support**. Save and exit.
4. $ **make linux.build**

With the kernel correctly configured and running, configure the **usb0** interface, for example:

```
# ifconfig usb0 10.0.0.150
```

where **10.0.0.150** is the IP address assigned to the target.

14.3 Configuring a Host for USB Connection

The configuration procedure depends on the host OS.

Linux 2.6 Host Configuration

Driver Support

The host should have the **g_ether** module installed. Refer to *Target Configuration for a Linux Kernel 2.6 Host*, p.180 for details on installing the **g_ether** module.

Interface Configuration

For a Linux 2.6 host, configure and test the USB interface as follows.

1. Connect a USB cable with the host connector attached to the Workbench host and the peripheral connector attached to the target.
2. Configure the USB interface on the host as follows.

   ```
   # ifconfig usb0 10.0.0.151
   ```

   In this example, **10.0.0.151** is the address assigned to the host on the USB connection.
3. Verify that the connection is working with a **ping** command to the target:

   ```
   $ ping 10.0.0.150
   ```

   A successful ping of the target over the USB connection shows that your host and target are properly connected.

Linux 2.4 Host Configuration

Driver Support

The host should have the **CDCEther** module installed. Refer to *Target Configuration for a Linux Kernel 2.4 Host*, p.180 for details on installing the **CDCEther** module.

Interface Configuration

For a Linux 2.4 host to communicate over the USB connection you must configure the **eth***X* interface, where *X* is the next available Ethernet interface number, typically **eth1**.

1. Connect a USB cable with the host connector attached to the Workbench host and the peripheral connector attached to the target.
2. Configure the **eth***X* interface, for example:

   ```
   # ifconfig eth1 10.0.0.151
   ```

   In this example, **10.0.0.151** is the address assigned to the host on the USB connection.
3. Verify that the connection is working with a **ping** command to the target:

   ```
   $ ping 10.0.0.150
   ```

   A successful ping of the target over the USB connection shows that your host and target are properly connected.

Windows Host Configuration

You must install the **linux.inf** file on the windows host. The file is available in the **Documentation** directory of most kernels including all Wind River Linux kernels.

To install the file, first convert it to the DOS text file format with the **unix2dos** command:

```
$ unix2dos linux.inf
```

Connect a USB cable with the host connector attached to the Workbench host and the peripheral connector attached to the target.

When you attach the USB cable to the Windows host, Windows will recognize new hardware and start the new hardware wizard. Follow the prompts and install the **linux.inf** file.

15 Connecting with TIPC

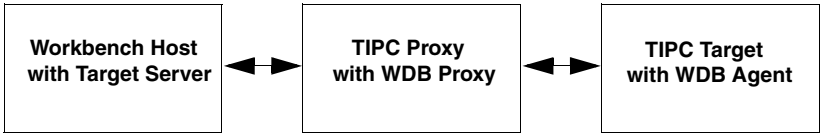
15.1 Overview

This appendix describes how to configure Linux TIPC targets and your Workbench host to support debugging. For detailed information about TIPC, see the official TIPC project Web site at **http://tipc.sourceforge.net/**.

The transparent inter-process communication (TIPC) infrastructure is designed for inter-node (cluster) communication. Targets located in a TIPC cluster may not have access to standard communication links or may not be able to communicate with hosts not located on the TIPC network. Because of this, host tools used for development may not be able to access those targets and debug them without special tools. To solve this communication problem between the TIPC target and TCP/IP hosts, Wind River provides the **wrproxy** process, which acts as a gateway between the host and the target. (For a more generalized use of **wrproxy**, see *B. Configuring a Wind River Proxy Host*.)

A basic diagram of a Workbench host configured to debug a TIPC target is shown in Figure 15-1.

Figure 15-1 **Workbench Host, Proxy, and TIPC Target**



The Workbench host communicates using UDP, the TIPC target communicates using TIPC, and the proxy translates between them.

Note that the functions of the three network hosts shown in Figure 15-1 may be combined in different ways, for example, the WDB proxy and WDB agent may both reside on a single target. You may even configure your Workbench host to support all functions if you want to test your debug capabilities in native mode before configuring external TIPC targets.

The following sections describe how to configure TIPC targets, configure a proxy, and configure your Workbench host to support debugging over TIPC.

15.2 Configuring TIPC Targets

To configure TIPC targets, you must install the TIPC kernel module on them. To configure them to communicate with Workbench, you must also run the WDB agent on them.

Note that TIPC communication between nodes in a cluster does not require UDP or TCP/IP networking services so those functions do not need to be included with the kernel, enabling a smaller kernel with fast, intra-node (TIPC) communication. For the TIPC-configured node to communicate with the Workbench host, however, a proxy must be provided that is capable of both TIPC and UDP communication capabilities. The proxy may be provided by one of the cluster nodes, a separate host, or the Workbench host itself as described in *15.3 Configuring a TIPC Proxy*, p.188.

15.2.1 Installing the TIPC Kernel Module

If you are using a Wind River Linux platform, the **tipc.ko** kernel module is supplied. If you do not have the kernel module, you can download the source from **http://sourceforge.net/projects/tipc** and then build it based on the instructions in the downloaded **README** file.

Once you have the kernel module, install it and configure it on the target as follows.

1. Load the TIPC kernel module:

   ```
   # insmod /lib/modules/2.6.10-gpp/net/tipc.ko
   ```

   (The example shown is for the module supplied with one version of the Wind River Linux platform. The location of your **tipc.ko** kernel module may differ.)

2. Set the local TIPC address:

   ```
   # tipc-config -a=1.1.1 -be=eth:eth0
   ```

   (Your actual command will differ if your network device is not **eth0** or if you chose an address different from **1.1.1**.)

3. Check that everything is configured properly:

   ```
   # tipc-config -a
   ```

   The output should display your current TIPC address, for example, **1.1.1**.

15.2.2 Running the usermode-agent

You must run **usermode-agent** on each target you want to reach. Find the correct agent for your target architecture in *installDir*/**linux-2.x/usermode-agent/1.1/bin/***arch*. For example, the correct agent for the PPC architecture, when your *installDir* is **WindRiver**, is **WindRiver/linux-2.x/usermode-agent/1.1/bin/ppc/usermode-agent**.

To launch **usermode-agent** on the target, copy it to the target, **cd** to the directory where it is located, and enter the following command:

```
$ ./usermode-agent -comm tipc &
```

15.3 Configuring a TIPC Proxy

The WDB proxy enables communication between the Workbench host and the TIPC target. The target server on the Workbench host (see *15.4 Configuring Your Workbench Host*, p.190) instructs the proxy agent to communicate using TIPC with a specified TIPC target address.

The WDB proxy agent is the **wrproxy** command (or **wrproxy.exe** with Windows). The host that runs **wrproxy** must have TIPC capability. To configure TIPC capability, install the TIPC kernel module (see *15.2.1 Installing the TIPC Kernel Module*, p.187), or build TIPC into the kernel and reboot it. When you have TIPC support in the kernel, configure the host with a TIPC address that is different from target TIPC addresses using **tipc-config** (see *15.2.1 Installing the TIPC Kernel Module*, p.187).

The **wrproxy** command is located in *installDir***/workbench-***version***/foundation/***version***/x86-linux2/bin/**. Enter the following command on the TIPC-capable network host that is to serve as the proxy between Workbench and the TIPC target:

```
$ wrproxy &
```

You can also use the **-p** *port* option to specify a different TCP port number for **wrproxy** to listen to (default **0x4444**), the -V option for verbose mode, or the **-h** option to get command help.

NOTE: If you specify a port other than the default port for the proxy, then you must specify the same port when configuring the target server as described in *15.4 Configuring Your Workbench Host*, p.190.

Figure 15-2 illustrates a configuration in which the proxy agent runs on the same host as Workbench. Figure 15-3 illustrates a configuration in which the proxy agent runs on one of the nodes in a cluster. Another example might be a separate host that runs **wrproxy**, between the targets in the cluster and the Workbench host.

Figure 15-2 **TIPC Configuration with WDB Proxy Agent on Workbench Host**

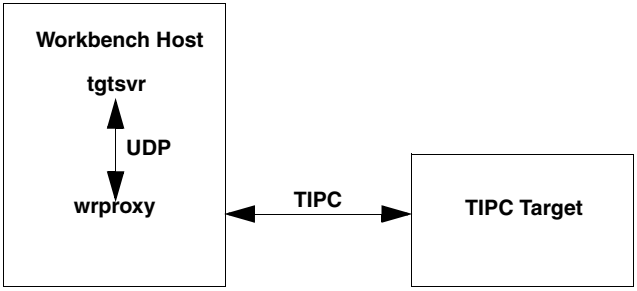
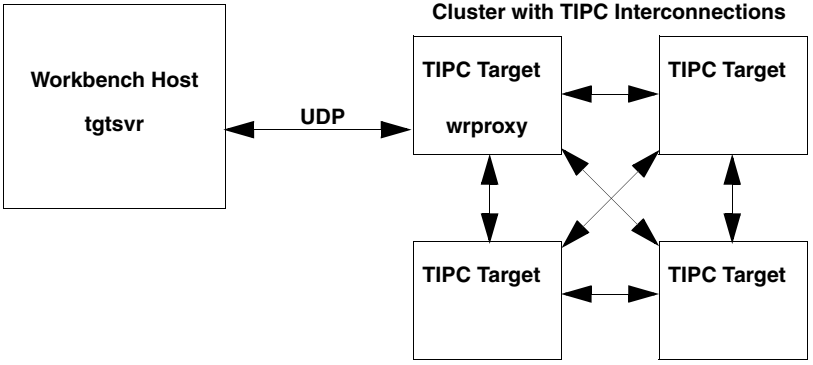


Figure 15-3 **TIPC Configuration with WDB Proxy Agent on Cluster Target**



15.4 Configuring Your Workbench Host, p.190 describes how to configure the target server on the Workbench host to connect to the proxy agent and reach the TIPC target that you want to connect to.

15.4 Configuring Your Workbench Host

Use the **tgtsvr** command to connect to the proxy for communication with a TIPC target. The following command shows the TIPC options to use:

tgtsvr [-V] -B wdbproxy -tipc -tgt *targetTipcAddress ProxyIpAddres*

For example, to connect to a target with a TIPC address of 1.1.8 using a proxy with the IP address 192.168.1.5, use the following command:

```
$ tgtsvr -B wdbproxy -tipc -tgt 1.1.8 192.168.1.5
```

Additional Information

A fuller syntax for the **tgtsvr** command is:

tgtsvr [-V] -B wdbproxy -tipc -tgt *targetTipcAddress* [**-tipcpt** *tipcPortType* **-tipcpi** *tipcPortInstance*] *wdbProxyIpAddress*|*name*

Table 15-1 explains the italicized parameter values in the command.

Table 15-1 **TIPC-Specific Parameter Values for Starting a Target Server**

Parameter	Value
targetTipcAddress	The TIPC address of the target with the **TIPC network stack**. For example: **1.1.8**.
tipcPortType	The TIPC port type to use in connecting to the WDB target agent. The default port type for the connection is **70**. You should accept the default port unless it is already in use.
tipcPortInstance	The TIPC port instance to use in connecting to the WDB target agent. The default port instance for the connection is **71**. You should accept the default port instance unless it is already in use.
wdbProxyIpAddress|*name*	The IP address or DNS name of the target with WDB Agent Proxy.

Note that if you change the default TIPC port configuration, you must also change the default TIPC port for the **usermode-agent** as described in *15.5 usermode-agent Reference*, p.191.

Alternatively, you can use the Workbench GUI to configure the host. Select **wdbproxy** as the backend when you create a new connection in the Target Manager and then fill in the fields with the values you would supply as command line arguments. The command line that is created at the bottom of the GUI should be similar to the example shown in this section.

15.5 usermode-agent Reference

This section explains the several possible options available when launching the usermode agent.

The listening port is the port used by the usermode agent to communicate with the target server on the host machine. If you change the listening port on the usermode agent side, then you have to specify the same port number to the target server.

Port option

The port option is:

-p `or` **-port 0x***pppp* `(UDP) | ` *xxxx*:**yyyy** `(TIPC)`

This option allows you to select an alternate listening port for the usermode agent.

Two network connection types are supported:

- UDP—this is the default connection type. If you don't specify a particular type of network connection, then this will be the used one.

  If you don't want to use the default UDP port (0x4321) then you can choose and set the one you want using this option. The port number can be entered in either decimal or hexadecimal format. To set the port number using the hexadecimal format, you need to use the **0x**%*x* format where %*x* represents the port number in hexadecimal base.

Example

To launch the usermode agent using UDP and port 6677:

```
$ usermode-agent -p 6677
```

or

```
$ usermode-agent -p 0x1A15
```

- TIPC—this is the TIPC network connection. If you don't want to use the default TIPC port type (70) and TIPC port instance (71) then you can choose and set the ones you want using this option. The port numbers can be entered in either decimal or hexadecimal format. To set the port numbers using the hexadecimal format, you need to use the **0x**%*x* format where %*x* represents the port number in hexadecimal base.

  To launch the usermode agent using TIPC and port type 1234, port instance 55:

  ```
  $ usermode-agent -p 1234:55
  ```

  or

  ```
  $ usermode-agent -p 0x4D2:37
  ```

Communication Option

The communication option allows to specify which kind of connection will be used for connection between target server and usermode agent.

Comm option is:

-comm serial | tipc

If the serial option is set then you can also specify the serial link device to use rather than the default one (/**dev/ttyAMA1**) and the baud speed for the serial link (115200 is the default baud speed).

To set a different device for the serial link connection, the flag **-dev** has to be used with the **-comm** serial option. For the baud speed, you need to set the **-baud** option combined with the **-comm** serial option.

Example

To launch the usermode agent using serial link connection and serial device **/dev/ttyS0**:

```
$ usermode-agent -comm serial -dev /dev/ttyS0
```

Example

To launch the usermode agent using serial link connection with default serial device and baud speed of 19200:

```
$ usermode-agent -comm serial -baud 19200
```

Example

To launch the usermode agent using serial link connection with serial device **/dev/ttyS0** and baud speed of 19200:

```
$ usermode-agent -comm serial -dev /dev/ttyS0 -baud 19200
```

If the **tipc** option is set then you can also specify the port type (default is 70) and port instance (default is 71) of the TIPC connection.

To set a different port type for the TIPC network connection, the flag **-tipcpt** has to be used, in either decimal or hexadecimal format.

To set a different port instance for the TIPC network connection, the flag **-tipcpi** has to be used, in either decimal or hexadecimal format.

Examples

To launch the usermode agent using TIPC network connection with default port type and default port instance:

```
$ usermode-agent -comm tipc
```

To launch the usermode agent using TIPC network connection with specific port type 123 and specific port instance 456:

```
$ usermode-agent -comm tipc -tipcpt 123 -tipcpi 456
```


Daemon mode

The **-daemon** option lets the usermode agent become a daemon after all initialization functions are completed. The output message, if any, are still reported on the device where the process has been started.

Environment Inheritance

The **-inherit-env** option makes all the child processes inheriting the environment from the parent environment. Since the usermode agent is the father of all the processes, then the processes will inherit the shell environment from which the usermode agent has been launched.

No Thread Support (Linux Thread Model Only)

The **-no-threads** option allows you to use the usermode agent on a kernel using Linux threading model even if the **libpthread** library is stripped. Basically, the **libpthread** library is used by the usermode agent to detect thread creation, destruction and so on. On a kernel using Linux threading model, if the **libpthread** library is stripped then the multithread debug would not be reliable so, by default, the usermode agent exit if this option is not set, to ensure a reliable debug scenario.

This option is useless if your kernel is running using NPTL threading model.

Other Options

The **-v** option displays version information about the usermode agent, that is, build and release information.

The **-V** option set the usermode agent to run in verbose mode. This is useful to have the listening information: port number, listening connection type and the target server connection to this usermode agent.

The **-help** or **-h** option displays all the possible startup options for the usermode agent.

PART VI

Debugging

16 Launching Programs

16.1 Introduction

Whenever you run a process or task from Workbench, a *launch configuration* is automatically created for you. A launch configuration is like a named script that captures the whole process of building, connecting a target, running, and possibly attaching a debugger. You can rerun your previous launches at any time by clicking a single button.

This chapter explains how to edit and fine-tune your launch configurations to provide a tight edit-compile-debug cycle, as well as how to manually attach the debugger to tasks and processes.

16.2 Creating a Launch Configuration

A launch configuration is similar to a macro because it allows you to group together the actions required to start a program. Then, with a single click, you can connect to your target, start your process, and if you wish, attach the debugger. Your configurations are stored persistently, and can be shared with your team.

Launch configurations can be run in one of two modes:

- *Run-mode* connects to your target, then launches a process.
- *Debug-mode* is like run-mode, but it automatically attaches the debugger after completing all other actions.

The same launch configuration can be executed in either mode.

To create a launch configuration:

1. Select **Run > Run** or **Run > Debug**. The **Create, manage, and run configurations** dialog box appears.
1. From the **Configurations** list, select the type of configuration you want to create, **Attach to Target** or **Process on Target** (explained below). Click **New**.
2. Once you click **New**, tabs appear and display the appropriate fields and options for your configuration type.

16.2.1 Editing an Attach to Target Launch Configuration

You do not create **Attach to Target** launch configurations manually. Instead, these configurations are created automatically when you attach to a process or kernel task from the Target Manager.

Attach to Target launch configurations are special in some ways:

- They do not actually run something but just connect a target and attach the debugger to some context that must already exist.

- They are visible only in Debug mode.
- **Attach to Target** launches can not be created manually. They are created automatically when you attach the debugger to a context using the Target Manager.

Once an **Attach to Target** launch is created, you can review and edit it in the Launch Configurations dialog box.

The Main Tab

The properties in the Main tab are for review only and cannot be changed.

What you can do is:

- Review your existing attaches and delete those that you no longer need.
- Rename your attaches and, if you think they are valuable, put them into your **Favorites** menu using the **Common** tab.
- Change the mapping between source paths compiled into your objects and source paths in your workspace by editing the Source Locator information in the **Sources** tab.
- Change the **Projects to Build** settings for the launch.

The Projects to Build Tab

The **Projects to Build** tab displays the projects that Workbench will build before launching the process in this configuration. To disable this, unselect **Window > Preferences > Run/Debug > Launching > Build (if required) before launching**.

To add to the list, click **Add Project**, select one or more projects from the dialog box, then click **OK**.

To rearrange the build order in the list, select a project then click **Up**, **Down**, or **Remove**.

Note that the Projects to Build list takes project-subproject relationships from the Project Navigator into account. Thus, when **myLib** is a subproject of **myProj** and you choose to add **myProj** to the list, you cannot add **myLib** to the list as well because it will be built automatically when you build **myProj**. Adding **myLib** as well would be redundant and is thus disabled.

The Source Tab

The **Sources** tab displays the order in which locations will be searched for source files during debugging. The search order is determined by a location's position in the list.

Configuring the Source Lookup Path is optional, and is only necessary if the build-target was compiled on a different host. See *18.2.8 Changing Source Lookup Settings*, p.237 for more information about the source locator.

1. On the **Sources** tab, click **Add** to configure the source lookup path.
2. Select the type of source to add to the lookup path.
3. Once you add a new source to the lookup path, you can adjust its position in the search order by clicking **Up** or **Down** to change its position in the list.

The Common Tab

The **Common** tab allows you to specify who can access this configuration, and whether it appears in the Workbench toolbar menu.

1. If this launch configuration is shared with others on your team, click **Shared**, then type or browse to the directory where the shared configuration is located.
2. If you want to be able to access this launch configuration from the **Debug** favorites menu (the drop-down menu next to the bug button on the Workbench toolbar), select **Debug** in the **Display in favorites menu** box.
3. If you want the process to launch in the background, ensure that box is selected.
4. Click **Apply** to save your settings but leave the dialog box open, click **Close** to save your launch configuration for later use, or click **Debug** to launch it now.

16.2.2 Creating a Process Launch Configuration

Once you click **New**, the **Main**, **Projects to Build**, **Debug Options** (in debug-mode), **Source**, and **Common** tabs appear.

The **Name** of the build target you selected in the **Project Navigator** appears at the top of the dialog box in the form *name - connection_name*. If you did not select a build target, or want to modify the name that appears, type a descriptive **Name** for your new launch configuration.

16.2.3 The Main Tab

The **Main** tab displays information about the output file that you want to download and run during the launch.

1. On the **Main** tab of the dialog box, keep the default **Connection registry** and **Connection to use** settings, or if you have more than one registry or connection defined in the **Target Manager**, you may select a different one from the pull-down list.

   To create a new registry or connection type, click **Add**.

2. The **Entry Point**, **Arguments**, **Priority**, and **Stack size** fields are only active when you are connected to a target. To retrieve the connection-specific properties from the target, and adjust them if necessary, click **Connect**. Once your target is connected, you can also click **Edit** to open the **Advanced Options** dialog box.

3. To adjust the options field, click **Select**. The **Options** dialog box appears.

16.2.4 The Projects to Build Tab

You can specify that other projects should be built (if necessary) before launching the current project with the **Projects to Build** tab. This only applies if you have checked **Build (if required) before launching** in the **Window > Preferences > Run/Debug > Launching** dialog box.

Click **Add project** to select one or more existing projects. You can change the order in which they are built with the **Up** and **Down** buttons, or click **Remove** to remove them from the list. Then click **Apply** and **Close**. When this configuration is launched, those projects will first be built first (if required), in the specified order, prior to launch of this project.

16.2.5 The Debug Options Tab

The **Debug Options** tab only appears for launch configurations in debug-mode.

With **Break on Entry** checked and **main** entered in the box, the process will break at launch on the entry to the **main()** routine for debugging operations.

16.2.6 The Source Tab

The **Source** tab displays the order in which locations will be searched for source files during debugging. The search order is determined by a location's position in the list.

Configuring the **Source Lookup Path** is optional, and is only necessary if the build-target was compiled on a different host. See *18.2.8 Changing Source Lookup Settings*, p.237 for more information about the source locator.

1. On the **Source** tab, click **Add** to configure the source lookup path.
2. Select the type of source to add to the lookup path; see *18.2.8 Changing Source Lookup Settings*, p.237 for a description of each type.
3. Once you add a new source to the lookup path, you can adjust its position in the search order by clicking **Up** or **Down** to change its position in the list.

16.2.7 The Common Tab

The **Common** tab allows you to specify whether this launch configuration is **Local** or **Shared** (local is the default), whether you want to access it from the Workbench toolbar buttons, and if the program should be launched in the background.

1. If this launch configuration is shared with others on your team, click **Shared**, then type in or browse to the directory where the shared configuration is located.
2. If you want to be able to access this launch configuration from the **Debug** favorites menu (the drop-down list next to the bug button on the Workbench toolbar), select **Debug** in the **Display in favorites menu** box.
3. If you want the process to launch in the background, ensure that box is checked.

4. Click **Apply** to save your settings, but leave the dialog box open, click **Close** to save your launch configuration for later use, or click **Debug** to launch it now.

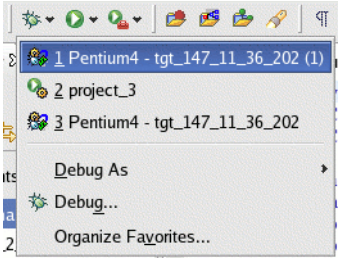
16.2.8 Using Launch Configurations to Run Programs

In a typical development scenario, you will run the same application many times in a single debugging session. After creating a launch configuration, you can click the **Debug** button to run a process and attach the debugger in a few seconds.

To launch a recently run process:

1. Click the pull-down arrow next to the **Debug** button and select the process from the configuration list. Figure 16-1.

Figure 16-1 **Debug Launch List**



If you ran the configuration recently, it will appear on the menu. If you selected **Debug** from the **Display in favorites menu** list (see *The Common Tab*, p.202) for a configuration, it will always appear on the list, whether you have run it recently or not.

2. To run a configuration not listed on the menu, click **Debug**, then choose the configuration from the **Configurations** list, click **Debug**.

Increasing the Launch History

Workbench stores a history of previously launched configurations. The default length of the launch history is 10, but you can increase the history length by selecting **Window > Preferences > Run/Debug > Launching** and increasing the number in the **Size of recently launched applications list** field.

Troubleshooting Launch Configurations

If you press the **Debug** button (or click the **Debug** button from the **Launch Configuration** dialog box) and get a "Cannot create context" error, check the **Exec Path** on the **Main** tab of the **Debug** dialog box (see page 204) to be sure it is correct. Also check your Object Path Mappings.

16.3 Remote Java Launches

There are four Java launch configuration types in Workbench:

- **Java Applet**
- **Java Application**
- **Remote Java Application**
- **Remote Java Launch and Connect**

The first three are standard Eclipse-with-JDT launch configuration types. **Java Applet** and **Java Application** are for native mode debugging of Java applets and Java applications. The **Remote Java Application** launch configuration type is for connecting to an already running Java application on a remote target. It does not launch the application on the target. You must start the application through whatever means are available and apply the necessary options for remote debugging. Also, the I/O capabilities of the Console view are not available. For details on these launch configuration types, refer to **http://help.eclipse.org**.

Wind River has added the **Remote Java Launch and Connect** launch configuration type which is documented here. This type of launch uses the usermode agent running on the remote target to start the application and then connects the debugger to the application. **Remote Java Launch and Connect** makes it easy to start debugging remote Java applications with a single click, and it also makes application I/O available in the Console view. This type of launch is required to debug combined Java and JNI code as described in *18.4 Java-JNI Cooperative Debugging*, p.239.

Create a Java Project

Create new Java projects in Workbench by selecting **New > Project > Java Project**, and import existing **.java** files or use the Editor to create new ones. Build the project by right-clicking on the project and selecting **Build Project**. A successful build generates class files and reports no errors. The following discussion assumes you are able to successfully build a Java application in Workbench and now wish to debug it on a target.

Prepare the Target

The target must have a supported Java runtime environment (see your *Release Notes* for details) and be running the Wind River usermode agent.

1. If your target does not have a **usermode-agent** executable on it, copy one to it.

   Source and binary files for **usermode-agent** may be found in your Workbench installation directory under the **linux-2.x/usermode-agent/2.0/** subdirectory.

NOTE: If you have both a **1.1** and **2.0** subdirectory, use the contents of the **2.0** directory for remote Java launches.

   If there is no binary available for your target, refer to *3.11 Creating Projects at External Locations*, p.44 for an example of a way to build **usermode-agent** from the source files with Workbench.

2. Run the usermode agent on the target, for example:

   ```
   target_$ ./usermode-agent -p 4321 &
   ```

   In this case, a port is specified so that additional instances of **usermode-agent** can be run on the target for other purposes.

NOTE: If you specify a port when invoking **usermode-agent**, you must also specify this port number when you create the target connection from the host. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.

3. Make the Java application class files available for execution on the target. For example, you could build the application in a Workbench project that is on a filesystem shared between the host and target, or simply copy the class files from the host to the target.

4. Make a target connection or create a new target connection to your target in the Target Manager. If you specified a specific port when starting **usermode-agent**

on the target, you must specify that port in the target connection dialog as well. (For details on how to create a target connection, refer to *13. Connecting to Targets*.)

Launch the Remote Debugging Session

The following procedure provides an example of how to initiate a remote Java launch and debugging session. Only some of the possible launch configuration settings are discussed. For details on all launch configuration settings, refer to *Wind River Workbench User Interface Reference: Launch Configuration Dialog*.

1. Select **Run > Debug** and then select the **Remote Java Launch and Connect** configuration type. Click **New** to open a set of tabs. Tabs marked with a red X require input from you.
2. In the **Main** tab:
   a. Specify a name for your launch in the **Name** field or leave the default name.
   b. Click **Browse** to find the correct project and click **Search** to identify the correct main class.
   c. Check **Stop in main** to cause the debugger to suspend execution on entry into the method **main** in the main class.
3. In the **Connection** tab:
   a. Select the correct target connection from the **Connection to use** pull-down menu.
   b. If you are using an NFS mount and have not specified object path mappings for your target connection, you can do that now by clicking **Properties**, select the **Object Path mappings** tab, and click **Add**. Enter the host paths and target paths that map. For example, if you have mounted your target's root filesystem on the host's **/target** directory, enter **/** for the target path and **/target** for the host path. Click **OK** and click **OK** again to return to the launch configuration tabs.
   c. Enter or browse to the path of the **java** binary on the target in the **Exec Path on Target** field. For example, if the **java** binary on the target resides in the **/usr/bin** directory, enter **/usr/bin/java**.
4. In the **Arguments** tab, enter the directory path to the class files in the **Classpath** field. For example, if the target path **/java/classfiles/** holds the class files for your Java application, enter **/java/classfiles/** in this field.

   Note that the complete command line is displayed at the bottom of this tab.

5. In the **Debug Options** tab, enter a port number. This is the debugging port number for JDT—not the usermode agent port number. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.
6. Click **Debug**.

   Workbench changes to the Debug perspective, and the Debug view shows the status of the launch. If you checked **Stop in main** in the **Main** tab of the launch configuration, you should see the application suspended at main. The source file should also be open in an Editor window, with the location where execution has been suspended highlighted.

   At this point you can examine variables, set breakpoints, step through your application, and so on. Output and standard error output will appear in the Console view, which is discussed in *16.6 Launches and the Console View*, p.212.

   Click the **Resume** icon in the Debug view toolbar to continue program execution.

16.4 Launching Programs Manually

Once a launch configuration has been established, you can run programs using the **Run** or **Target** selections on the menu bar, or with the **Run** and **Debug** buttons on the toolbar. Workbench will automatically try to connect to the target, if a connection is not already running.

16.5 Controlling Multiple Launches

You can create a Launch Control launch, consisting of a sequence of your launch configurations, each one of which is then considered a sub-launch. You can even add other Launch Control launches to a Launch Control configuration, the only restriction being that a Launch Control configuration cannot contain itself.

For detailed information on launch control settings, see the *Wind River Workbench User Interface Reference: Launch Configuration Dialog*.

Terminology

A *launch* is a specific instance of a *launch configuration*, and a launch configuration is a specific instance of a *launch type*. The launch is what occurs when you initiate a run or debug session.

A launch configuration is your definition of how the launch will occur, for example, what program will be run, what target it will run on, and what the arguments are.

A launch type defines the kind of launches that are supported by Workbench. There are several different kinds of launch types, for example, Java Application, Process on Target, and Launch Control. The launch type includes GUI elements that specify attributes specific to it.

You create a launch configuration based on a launch type, specifying the appropriate attribute values. You then initiate a launch based on a launch configuration. Launches also have a *mode*, the two standard modes being Run and Debug. A launch may be initiated by the **Run** or **Debug** buttons in Workbench (launches may be initiated other ways too). Note that some launch types are only available in one mode. For example, the **Remote Java Application** launch type can only be used in Debug mode.

Configuring a Launch Sequence

The following procedure assumes you have two or more launch configurations already defined (see *16.2 Creating a Launch Configuration*, p.198).

1. Select **Run > Debug** and the Debug dialog opens.
2. Select **Launch Control** from the **Configurations** list on the left, and then click **New** at the bottom. A new launch control configuration with the default name **New Configuration** appears. Change the name as desired.
3. Select the **Launch Control** tab. Note that your current launch configurations are listed under **Available Configurations** on the left, and a space on the right is labeled **Configurations to Launch**.

4. Select each launch that you want to add to your new launch configuration and click **Add** to add it to the list of configurations to launch. When you have a list of configurations to launch, you can organize them in the order you want them to launch by selecting a configuration and clicking **Move Up** or **Move Down**. The sub-launch at the top of the list will come first and the one at the bottom last. You can remove any sub-launch from the Launch Control configuration by selecting it and clicking **Remove**.

You now have a Launch Control configuration that will launch a sequence of sub-launches in the order specified in the **Configurations to Launch** list. You can also specify commands to perform before launches, after launches, and in response to a launch failure or an application error report as discussed in the next section.

Each launch in a Launch Control will open a Console view for I/O and error messages as described in *16.6 Launches and the Console View*, p.212.

Pre-Launch, Post-Launch, and Error Condition Commands

To access the launch configuration commands, select a sub-launch in your **Configurations to Launch** list and click **Properties** (or double-click the sub-launch). A properties page containing command information is displayed. Here you can specify pre-launch, post-launch, and error condition commands, which will inherit the environment variables shown below them unless you change them in the command. Your changes affect the launch you are working with only—other launches using the same configuration get the default values for the environment variables. Also, the set of environment variables differs for each launch configuration (see *Understanding the Command Environment*, p.211 for more on environment variables).

Preparing a Launch with a Pre-Launch Command

An example of the use of a pre-launch command is to prepare a target for use. For example, in a development environment you might have to reserve a target, and you would not want to attempt a launch without being sure you had a target to launch on. So a pre-launch command might be a script that reserves the board, puts **usermode-agent** in the root file system, reboots the board, and starts **usermode-agent**.

If the pre-launch command returns a non-zero return code then the launch is aborted and the error condition command is executed for each sub-launch previous to the failed sub-launch.

Using a Post-Launch Command

If your application requires additional set up after it has been launched, or if you would like to verify that it has launched correctly before proceeding to the next launch, use a post-launch command.

If the post-launch command returns a non-zero return code then the launch is aborted and the error condition command is executed for each sub-launch previous to the failed sub-launch as well as for the failed sub-launch.

Using the Error Condition Command

The error condition command of a launch is run when a launch fails, or a pre-launch or post-launch command returns a non-zero error code. This causes the error command of the current launch to run, and then each error command of any preceding launches to run. The error condition commands are executed in reverse order of the sequence in which the launches occurred. For example, if the fourth launch fails, the error condition command of the fourth launch is performed, then the error condition of the third launch, and so on. This is to deal with situations in which previous commands may have acquired locked resources—unlocking them in reverse order is important to prevent potential deadlock.

NOTE: To be precise, error commands are called in the reverse order that the pre-launch commands were called. An error command will never be called for a sub-launch that did not pass the pre-launch command step.

Inserting Commands using an Empty Sub-Launch

You can place a command into your Launch Control that is not associated with any particular sub-launch by adding an empty Launch Control to hold the command. Select Launch Control and click **New** and then specify a name for the dummy launch, for example, **Empty Launch**. Add the empty launch to the Launch Control and use the properties page to insert commands into the launch which aren't associated with any particular sub-launch.

Running All Pre-Launch Commands First

If you want to run each of the pre-launch commands for each launch first, check **Run Pre-Launch command for all launches first** on the main launch control page. The pre-launch commands will be executed in order, and only after they are all successfully completed will the first launch take place, followed by the second launch and so on. This provides for situations in which you do not want to continue with a complete launch Control sequence if any of the sub-launches cannot take place because, for example, a target is not available.

Launch Controls as Sub-Launches

You can use an existing Launch Control as a sub-launch, but do not attempt to create recursive launch controls in this way, as they will not run.

If the parent Launch Control's pre-initialize check box (**Run Pre-Launch command for all launches first**) is selected and the pre-initialize check box is set for the child Launch Control, the child will pre-initialize all of its sub-launches before operation continues on to the next sub-launch of the parent Launch Control. Otherwise, the child Launch Control will have its sub-launches initialize at the time that it is launched.

Understanding the Command Environment

The environment variables are collected from multiple locations and then provided on the Properties page as a convenience. Typically you will only read variable values, but you may want to change them in your pre-launch command. Your changes affect the launch you are working with only—other launches using the same configuration get the default values for the environment variables.

Environment variables are gathered from four different sources. First, variables may be defined on the Launch Control's **Environment** tab. These variables are not displayed on a sub-launch's Properties page because the information is readily available on the **Environment** tab. The next source for environment variables is from the sub-launch's **Environment** tab (if it has one). The third source for the list of environment variables is defined by the sub-launch's configuration type attributes. Each sub-launch configuration type defines its own set of attributes (further documentation on sub-launch attributes can be found in the Eclipse documentation for Launch Configuration). The final source of environment variables are defined by Launch Control and provide general support for the launch. The variables defined by Launch Control for each sub-launch are:

- **com_windriver_ide_launchcontrol_launch_mode**
- **com_windriver_ide_launchcontrol_env_file**
- **com_windriver_ide_launchcontrol_skip_next**

The environment variable **com_windriver_ide_launchcontrol_launch_mode** identifies the mode of a launch. The mode may be either **debug** or **run**, depending on how a launch is initiated (for example selecting the **Run > Debug** dialog to initiate a debug mode launch and **Run->Run** to initiate a run mode launch). Changing **com_windriver_ide_launchcontrol_launch_mode** has no effect—it is only provided for information about a current launch.

Since the command's environment terminates after the command completes any variables which need to be changed for a launch must be written to a file. The name of this file is provided in the environment variable **com_windriver_ide_launchcontrol_env_file**. The format of this file is a list of key value pairs on separate lines. Each key and value is separated by an = and the key identifies the variable name (this is a standard Java properties file). After a command is completed Launch Control will read this file and update any variables as specified in the file.

Launch control also defines the **com_windriver_ide_launchcontrol_skip_next** variable. Setting this variable to **true** in the Pre-Launch command causes the remainder of the sub-launch to be skipped. Setting this variable in post-launch or error commands has no effect.

An example of how this could be used is to check for the existence of a server application in a pre-launch command. If the application is already running then specifying **com_windriver_ide_launchcontrol_skip_next=true** in the **com_windriver_ide_launchcontrol_env_file** will cause the launch of the application to be skipped without invoking an error.

NOTE: Note that the Wind River environment variables for individual launches are subject to change and you should not count on them being maintained across releases. For details on variables beginning with the string **org_eclipse** refer to the documentation available at **http://help.eclipse.org**.

16.6 Launches and the Console View

Workbench supports the Eclipse **Console** view with Virtual IO (VIO) features that allow you to monitor the standard output and error output of your applications and to enter standard input. VIO connects the **Console** view to a particular context (process or task). You can also have multiple **Console** views and "pin" them to a particular context. Most **Console** view settings are available in the **Common** tab of your launch configuration, and you can specify **Console** view preferences in your Workbench preferences.

Note that **Console** view VIO is tied to the debugger and cannot always serve the same purposes as a terminal connection to the target. You cannot use it, for example, to monitor the boot loader or set boot parameters. The **Console** view is associated with a particular debugger context and is not a general purpose terminal connection.

Launches and the Console View

Each launch opens a Console view for I/O and error messages, provided the **Allocate Console** check box is selected in the Common tab of the launch (the default setting).

NOTE: This refers to the Common tab of each individual launch configuration, not the Common tab of the Launch Control configuration.

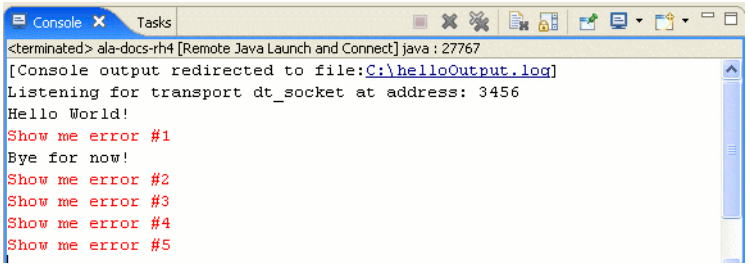
In the Common tab you can also specify a file where console output is appended or overwritten with each launch. The Console view itself offers several controls as described in the next section.

Note that you can also modify Console view settings such as buffer size and text colors by selecting your preferences at
Window > Preferences > Run/Debug > Console.

Console View Output

To open a **Console** view select **Window > Show View > Other > Basic > Console**. An example view is shown below.

Figure 16-2 **Example Console View**



The highlights of the view shown include the following:

- A title indicates which context (process or task) this view applies to.
- A comment indicates that in this case console file logging is occurring and identifies the log file location. Click on the filename to display it in the Editor.
- The standard output shown in the example is **Hello World!** and **Bye for now!** and is in black, the default color for standard output.
- The standard error outputs shown in the example are the **Show me error** messages which are in red, the default color for standard error output.

NOTE: The output appearing in the Console View can appear in a different order than the order the output was produced if both output and error output are present. The data from these two output types go through different channels and their transit times can be different.

Along with other standard functions, icons in the **Console** view toolbar allow you to pin the context to a Console view, select among different Console views, and create new Console views.

Select a specific process or task for a Console view by clicking the down arrow next to the **Display Selected Console** icon and making your selection. Click **Pin Console** to keep the Console view associated with that context. Select **Open Console > New Console View** to create additional Console views.

Refer to **http://help.eclipse.org** for further details on the Console view, or press **CTRL-F1** in the Console view for online help.

16.7 Attaching the Debugger to a Running Process

You can attach to a running process to debug it as follows:

1. In the Target Manager, expand **Processes** for the target connection and locate the process you want to attach to.
2. Right-click the process and select **Attach to Process**.

If you compiled the process with debug symbols, the symbols should load to allow source-level debugging.

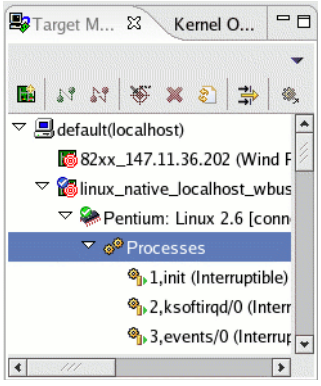
16.7.1 Running Processes

Right-clicking a process executable from the Project Navigator and selecting **Run Process on Target** or **Debug Process on Target** opens the appropriate launch configuration dialog box. See *16.2 Creating a Launch Configuration*, p.198 for more information about working with these dialog boxes.

This section explains how to launch processes from the Target Manager.

1. Right-click **Processes**, then select **Run/Debug Process**. The **Run Process** dialog box appears.
2. Type the path and filename (as seen by the target) into the **Exec Path on Target** field, or click the **Browse** button and navigate to the executable file.
3. To immediately put the program under debugger control at launch, select **Attach Debugger** and **Break on entry**; to let it run, clear the **Break on entry** check box. Click **OK**.

   Workbench runs the process on the target, and the executable and its host location, along with the individual tasks, appear below **Processes** in the Target Manager. If a red **S** appears, then symbol information has been loaded into the debugger.



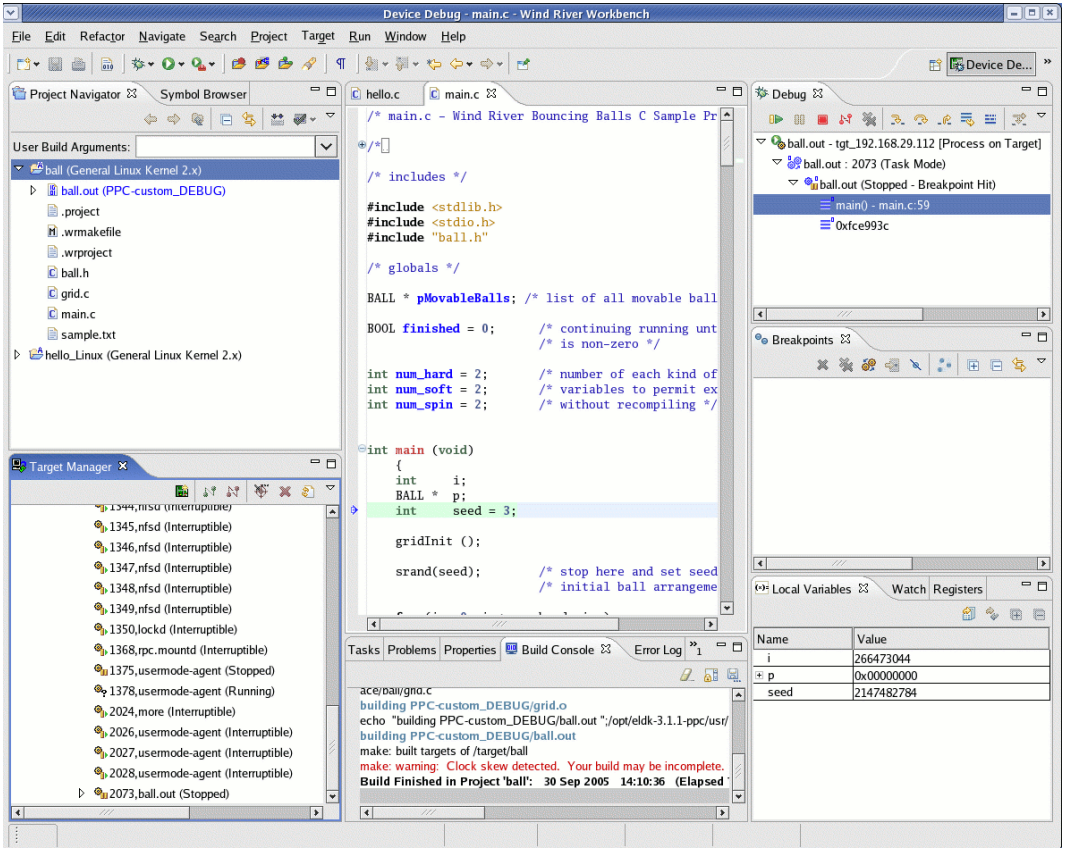
If you selected **Break on entry at main**, four other things happen as well:

- Workbench automatically switches to the Device Debug perspective (if it is not already open).
- The process is displayed in the Debug view.

- A temporary breakpoint is planted and appears in the Breakpoints view.
- The program executes up to the entry point at **main** and breaks.

The result is as shown in Figure 16-3.

Figure 16-3 **Process Running in Device Debug Perspective**



Whenever you manually run a process, a corresponding **Attach to Target** launch configuration with those properties is automatically created. For more information about how to use these configurations, see *Editing an Attach to Target Launch Configuration*, p.198.

16.8 Attaching to the Kernel

You can attach to the kernel in either KGDB or System (dual mode) debugging modes.

16.8.1 Attaching to Kernel Core (KGDB)

For an example of attaching to core with a KGDB connection, refer to *5.3.3 Attaching to Core and Debugging the Kernel*, p.75.

16.8.2 Attaching the Kernel in System Mode (Dual-Mode Agent)

To attach the kernel in System Mode, right-click the CPU button just below the Connection button and select **Attach to Kernel**.

This will create an **Attach to Target** launch configuration that automatically switches your target into System Mode before attaching the debugger. The Debugger will show a single node labelled **System Context** that represents the code that the CPU is currently executing. When you stop (suspend) the System Context, your entire System is stopped, including all the tasks, processes, and interrupt service routines. You can now also set breakpoints that will suspend the entire System when they are hit.

Note that System Mode breakpoints (breakpoints that are planted while a System Mode attach is active) will only be active when your target is in System Mode. You can switch your target between System Mode and User Mode by choosing the gear-wheel icon in the Target Manager, or by ticking the **Debug Mode** menu items in the Debugger. For more information about Debug Mode functionality, see *18.2.4 Using Debug Modes*, p.235.

16.9 Suggested Workflow

Launch Configurations allow for a very tight Edit-Compile-Debug cycle when you need to repeatedly change your code, build and run it. You can use the **F11** (Debug Last Launched) key to build the projects you have specified, connect your target (unless it is already connected), download, and run your most important program over and over again.

The only thing to watch is that you cannot rebuild your program or kernel while it is still being debugged (or its debug info is still loaded into the debugger). Depending on the size of the modules you run and debug, it can be the case that the debug server cannot load all the symbolic information for your modules into memory. By default, the size limit is set to 60MB (this can be changed by selecting **Window > Preferences > Target Manager > Debug Server Settings > Symbol File Handling Settings**.)

If a module is bigger than this limit, it will be locked against overwriting as long as the debugger has symbols loaded. This means that when you try to rebuild this module, you will see a dialog box asking you to unload the module's symbol information from the debugger before you continue building. You can usually unload symbolic information without problems, provided that you do not have a debug session open in the affected module. If you have a module open, you should terminate your debug session before continuing the new build and launch process.

17 Managing Breakpoints

17.1 Introduction

Breakpoints allow you to stop a running program at particular places in the code or when specific conditions exist.

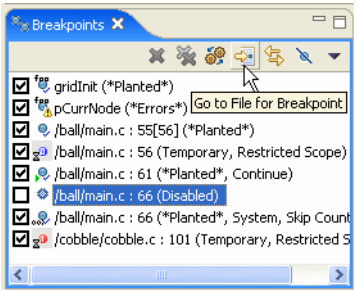
This chapter shows how you can use the Breakpoints view to keep track of all breakpoints, along with their conditions (if any).

You can create breakpoints in different ways: by double-clicking or right-clicking in the Editor's left overview ruler (also known as the **gutter**), or by opening the various breakpoint dialog boxes from the pull-down menu in the Breakpoints view itself.

17.2 Types of Breakpoints

Figure 17-1 shows the Breakpoints view with various types of breakpoints set.

Figure 17-1 **Breakpoints View**



See the sections below for when and how to use each type of breakpoint.

17.2.1 Line Breakpoints

Set a line breakpoint to stop your program at a particular line of source code.

Creating Line Breakpoints

To set a line breakpoint with an unrestricted scope (that will be hit by any process or task running on your target), double-click in the left gutter next to the line on which you want to set the breakpoint. A solid dot appears in the gutter, and the Breakpoints view displays the file and the line number of the breakpoint. You can also right-click in the gutter and select **Add Global Line Breakpoint**.

To set a line breakpoint that is restricted to just one task or process, right-click in the Editor gutter and select **Add Breakpoint for** *"selected thread"*. If the selected thread has a color in the Debug view, a dot with the same color will appear in the Editor gutter with the number of the thread inscribed inside it.

17.2.2 Expression Breakpoints

Set an expression breakpoint using any C expression that will evaluate to a memory address. This could be a function name, a function name plus a constant, a global variable, a line of assembly code, or just a memory address. Expression breakpoints appear in the Editor's gutter only when you are connected to a task.

Breakpoint conditions are evaluated after a breakpoint is triggered, in the context of the stopped process. Functions in the condition string are evaluated as addresses and are not executed. Other restrictions are similar to the C/C++ restrictions for calculating the address of a breakpoint using the Expression Breakpoint dialog box.

17.2.3 Hardware Breakpoints

Some processors provide specialized registers, called debug registers, which can be used to specify an area of memory to be monitored. For instance, IA-32 processors have four debug address registers, which can be used to set data breakpoints or control breakpoints.

Hardware breakpoints are particularly useful if you want to stop a process when a specific variable is written or read. For example, with hardware data breakpoints, a hardware trap is generated when a write or read occurs in a monitored area of memory. Hardware breakpoints are fast, but their availability is machine-dependent. On most CPUs that do support them, only four debug registers are provided, so only a maximum of four memory locations can be watched in this way.

There are two types of hardware breakpoints:

- A hardware *data breakpoint* occurs when a specific variable is read or written.
- A hardware *instruction breakpoint* or *code breakpoint* occurs when a specific instruction is read for execution.

Once a hardware breakpoint is trapped—either an instruction breakpoint or a data breakpoint—the debugger will behave in the same way as for a standard breakpoint and stop for user interaction.

Adding Hardware Instruction Breakpoints

There two ways to add a new hardware instruction breakpoint:

In the gutter (grey column) on the left of the source file, right-click and select **Add Hardware Code Breakpoint**. Or, double-click in the gutter to add a standard breakpoint and then, in the breakpoint view, right-click the breakpoint you've just added and select **Properties**. In the last pane (**Hardware**) of the **Properties** dialog box select **Enable Hardware Breakpoint**.

Adding Hardware Data Breakpoints

Set a hardware data breakpoint when:

- The debugger should break when an event (such as a read or write of a specific memory address) or a situation (such as data at one address matching data at another address) occurs.
- Threads are interfering with each other, or memory is being accessed improperly, or whenever the sequence or timing of runtime events is critical (hardware breakpoints are faster than software breakpoints).

To add a hardware data breakpoint, go to the breakpoint view, then click the down arrow in the top right of this view, and select **Add Data Breakpoint** to display the hardware data breakpoint dialog box. You are presented with four tabs. In the **General** tab, enter the variable you want to monitor in the **Address Expression** box. The **Status** and **Scope** tabs work the same way as they do for hardware code breakpoints. The **Hardware** tab will have fields bolded that you can check to make a selection from the drop-down list, for example to choose the access size (**Byte**, **Half-Word**, or **Word**) and the access type you want to monitor for this variable.

Disabling and Removing Hardware Breakpoints

You can disable and remove hardware breakpoints in the same ways that you disable and remove standard breakpoints.

Converting Breakpoints to Hardware Breakpoints

To cause the debugger to request that a line or expression breakpoint be a hardware code breakpoint, select the **Hardware** check box on the **General** tab of the **Line Breakpoint** or **Expression Breakpoint** dialog boxes.

This request does not guarantee that the hardware code breakpoint will be planted; that depends on whether the target supports hardware breakpoints, and if so, whether or not the total number supported by the target have already been planted. If the target does not support hardware code breakpoints, an error message will appear when the debugger tries to plant the breakpoint.

NOTE: Workbench will set only the number of code breakpoints, with the specific capabilities, supported by your hardware.

NOTE: If you create a breakpoint on a line that does not have any corresponding code, the debugger will plant the breakpoint on the next line that does have code. The breakpoint will appear on the new line in the Editor gutter.

In the Breakpoints view, the original line number will appear, with the new line number in square brackets **[ ]** after it. See the third breakpoint in Figure 17-1.

Comparing Software and Hardware Breakpoints

Software breakpoints work by replacing the destination instruction with a software interrupt. Therefore it is impossible to debug code in ROM using software breakpoints.

Hardware breakpoints work by comparing the break condition against the execution stream. Therefore they work in RAM, ROM or flash.

Complex breakpoints involve conditions. An example might be, “Break if the program writes *value* to *variable* if and only if **function_name** was called first.”

17.3 Manipulating Breakpoints

Now that you have an understanding of the different types of breakpoints, this section will show you how to work with them.

17.3.1 Exporting Breakpoints

To export breakpoint properties to a file:

1. Select **File > Export > Export Breakpoints**, then click **Next**. The Export Breakpoints dialog box appears.
1. Select the breakpoint whose properties you want to export, and type in a file name for the exported file. Click **Finish**.

17.3.2 Importing Breakpoints

To import breakpoint properties from a file:

1. Select **File > Import > Import Breakpoints**, then click **Next**. The Import Breakpoints dialog box appears.
2. Select the breakpoint file you want to import, then click **Next**. The Select Breakpoints dialog box appears.
3. Select one or more breakpoints to import, then click **Finish**. The breakpoint information will appear in the Breakpoints view.

17.3.3 Refreshing Breakpoints

Right-clicking a breakpoint in the Breakpoints view and selecting **Refresh Breakpoint** causes the breakpoint to be removed and reinserted on the target. This is useful if something has changed on the target (for example, a new module was downloaded) and the breakpoint is not automatically updated.

To refresh all breakpoints in this way, select **Refresh All Breakpoints** from the Breakpoints view toolbar drop-down menu.

17.3.4 Disabling Breakpoints

To disable a breakpoint, clear its check box in the Breakpoints view. This retains all breakpoint properties, but ensures that it will not stop the running process. To re-enable the breakpoint, select the box again.

17.3.5 Removing Breakpoints

There are several ways to remove a breakpoint:

- Right-click it in the Editor gutter, select **Remove Breakpoint**.
- Select it in the Breakpoints view, click the **Remove** button.
- Right-click it in the Breakpoints view, select **Remove**.

18 Debugging Projects

18.1 Introduction

Like other debuggers you may have used, the Wind River Workbench debugger allows you to download object modules, launch new processes, and take control of processes already running on the target.

However, unlike other debuggers, Workbench allows you to attach to multiple processes simultaneously, without affecting the state of the items you are attaching to or requiring you to disconnect from one process in order to attach to another.

This chapter introduces the Debug and Disassembly views, and shows you how to use them to debug your programs.

NOTE: You must compile your programs using debugging symbols (the **-g** compiler option) to use many debugger features. The compiler settings used by the Workbench project facility's Managed Builds include debugging symbols.

18.2 Using the Debug View

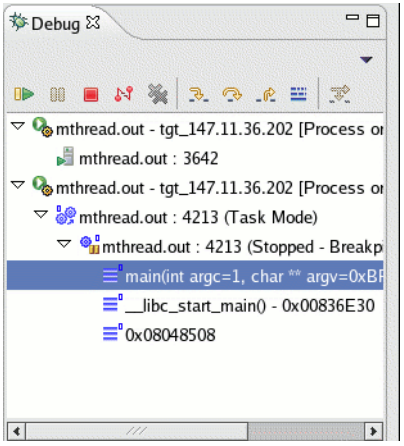
Use the Debug view to monitor and manipulate the processes running on your target.

To examine a process in the Debug view:

1. Connect to your target in the Target Manager view (see *13.4 Establishing a Connection*, p.168).
2. Launch one or more processes:
   - Using a launch configuration as described in *16.2 Creating a Launch Configuration*, p.198.
   - Manually, as described in *16.4 Launching Programs Manually*, p.207.
   - By attaching to an already running process, as described in *16.7 Attaching the Debugger to a Running Process*, p.214.
3. Once the debugger has attached to your process, it will appear in the Debug view.

The Debug view displays processes differently depending on whether the debugger is attached or not. Figure 18-1 shows two instances of the **mthread** process: the first instance does not have the debugger attached and the second does.

Figure 18-1 **Debug View with Unattached and Attached Processes**



18.2.1 Configuring Debug Settings for a Custom Editor

By default, the Workbench Editor opens when the debugger stops in a given file. To cause a different editor to open for particular file types, modify the mappings in **Window > Preferences > General > Editor > File Associations**.

Modifying these mappings takes care of editor selection and painting of the instruction pointer in the editor gutter. However, to associate other debugging actions with the new editor, you must modify the Eclipse extension point **org.eclipse.ui.editorActions**.

For example, the breakpoint double-click action associated with the Workbench Editor looks like this:

```
<extension point="org.eclipse.ui.editorActions">
   <editorContribution
       targetID="com.windriver.ide.editor.c"
       id="com.windriver.ide.debug.CSourceFileEditor.BreakpointRulerActions">
     <action
       label="Dummy.label"
       class="com.windriver.ide.debug.internal.ui.breakpoints.actions.ToggleB
reakpointRulerAction"
       actionID="RulerDoubleClick"
       id="com.windriver.ide.debug.ui.actions.toggleBreakpointRulerAction.c">
     </action>
</editorContribution>
```

Other features that are by default configured to work only with the Workbench Editor are **Run to line**, **Set PC to here**, and **Watch**. These features are configured through following extensions:

```
<viewerContribution
      targetID="#WREditorContext"
      id="com.windriver.ide.debug.ui.editprPopup.actions">
   <visibility>
      <and>
         <systemProperty
            name="com.windriver.ide.debug.ui.debuggerActive"
            value="true"/>
         <pluginState value="activated" id="com.windriver.ide.debug.ui"/>
      </and>
   </visibility>
   <action
        label="%WatchAction.label"
        icon="icons/actions/hover/watch_exp.gif"
        menubarPath="group.debug"
        helpContextId="com.windriver.ide.debug.ui.watchAction_context"
        class="com.windriver.ide.debug.internal.ui.actions.WatchAction"
        id="com.windriver.ide.debug.ui.editor.watchAction">
      <enablement>
        <systemProperty
           name="com.windriver.ide.debug.ui.debuggerActive"
           value="true">
        </systemProperty>
      </enablement>
   </action>
   <action
        label="%SetPcToHereAction.label"
        menubarPath="group.debug"
        helpContextId="com.windriver.ide.debug.ui.setPcToHereAction_context"
        class="com.windriver.ide.debug.internal.ui.actions.SetPcToHereAction"
        id="com.windriver.ide.debug.ui.editor.setPcToHereAction">
   </action>
   <action
        label="%RunToLineAction.label"
        icon="icons/actions/hover/run_to_line.gif"
        menubarPath="group.debug"
        helpContextId="com.windriver.ide.debug.ui.runToLineAction_context"
        definitionId="org.eclipse.debug.ui.commands.RunToLine"
        class="org.eclipse.debug.ui.actions.RunToLineActionDelegate"
        id="com.windriver.ide.debug.ui.editor.runToLineAction">
   </action>
</viewerContribution>
```

Please refer to Eclipse SDK documentation for more information on these extension points.

18.2.2 Understanding the Debug View Display

The Debug view displays a hierarchical tree for each process being debugged.

Below are examples of what might appear at each level of the tree, with a general description of each level.

ball (2) [Process on Target] = launch level
launch name [launch type]

> **16,ball.out (MPC8260: Linux 2.4)** = debug target level
> *process name (core name:OS name OS version)*
>
> > **16,ball.out (Stopped - Breakpoint)** = thread level
> > *thread name (state - reason for state change)*
> >
> > > **main( ) - main.c:59** = stack frame level
> > > *function(args) - file : line #*, can also be *address*

In Workbench 2.6, stack arguments and argument values are not displayed in the Debug view by default. This default setting was implemented to improve debugging performance.

To activate stack-level arguments in the Debug view, select **Window > Preferences > Run/Debug > Performance**, then select the **Retrieve stack arguments for stack frames in Debug View** and **Retrieve stack argument values for stack frames in Debug View** checkboxes. Click **OK**.

NOTE: The stack arguments reflect the current value of the stack argument variables, not the initial value of the stack arguments immediately after entering the function call.

How the Selection in the Debug View Affects Activities

Choosing a specific level of your debug target controls what you can do with it.

Selected Level	Action Allowed
launch	Terminate or disconnect from all processes/cores for the launch debug target.
debug target	Terminate or disconnect from the debug target. Perform run control that applies to the whole process: suspend/resume all threads.

	Assign color to the debug target and all its threads/tasks.
thread	Terminate or disconnect; terminates individual tasks/threads, if supported by process/core.
	Run control for thread: resume/suspend/step.
	Assign color to thread.
stack frame	Select of the stack frame causes the editor to display instruction pointer and source for stack frame.
	Perform same run control as on the thread.
	Assign color to thread.
	Assign corresponding color for parent thread.

Monitoring Multiple Processes

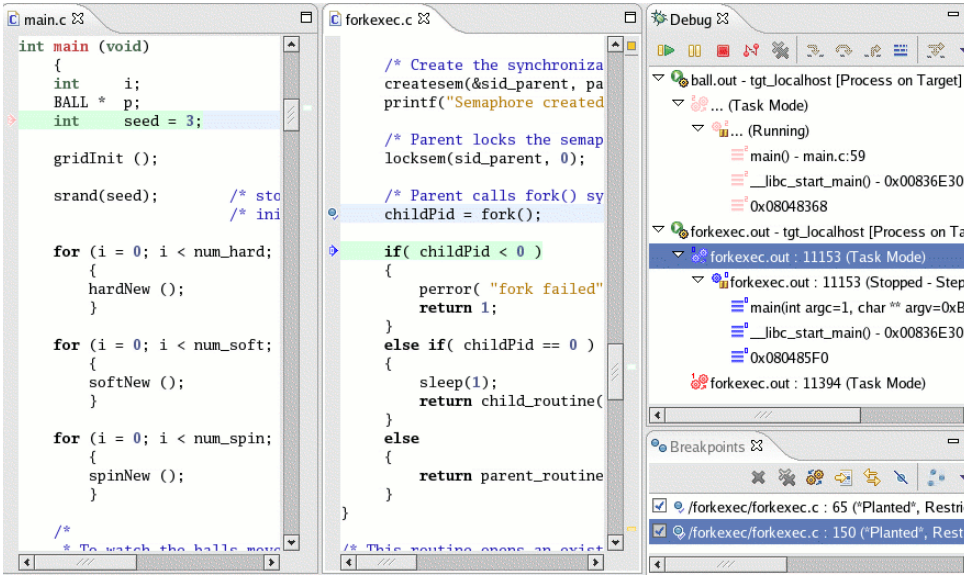
When you start processes under debugger control, or attach the debugger to running processes, they appear in the Debug view labeled with unique colors and numbers. Likewise, breakpoints that are restricted to a particular process display that process's color/number context in the Breakpoints and Editor views.

For example, in Figure 18-2, three processes are shown in the Debug view:

- The **ball** process, in pink in the Debug view, has been launched in debug mode and the program counter is shown, in pink, in the **main()** routine.
- The **forkexec** process is shown in blue. It has stopped at a breakpoint set at the **fork** system call. The breakpoint is shown as a solid circle and the program pointer is shown in blue with the number 0 in it. Note that the number 0 is also shown with the parent process in the Debug view.
- The third process, the forked child process, is shown in red in the Debug view.

The color assigned to a process or thread can be changed by right-clicking on the process or thread and selecting **Color >** *specific color*.

Figure 18-2 **Debug View with Editor and Breakpoint View**



The context pointer (the arrow in the left gutter in **main.c**) indicates the statement that will execute when the process resumes.

Colored Views

The color context of a process also carries through to other views whose scope is determined by the contents of the Debug view.

The data views that appear in the Device Debug perspective usually update to reflect whatever is currently selected in the Debug view. If you prefer, you can start colored views that are “pinned” to a process of a particular color and update only when that process changes.

To open a view of a particular color, select **Window > Show View > Other > Device Debug -** *color* **>** *view.*

For more information about how to set up Debug view settings, see the *Wind River Workbench User Interface Reference: Debug View.*

18.2.3 Stepping Through a Program

Once a process has stopped under debugger control (most often, at a breakpoint), you can single-step through the code, jump over subroutine calls, or resume execution; what you can do depends on what you selected in the Debug view.

When the program is stopped, you can resume operation by clicking **Resume** on the toolbar of the Debug view. If there are no more remaining breakpoints, interrupts, or signals, the program will run to completion (unless you click the **Suspend** button).

To step through the code one line at a time, click the Debug view's **Step Into** button. If you have other data views open, such as the Registers, Local Variables, or Global Variables views, they will update with current values as you step through the code.

The effect of **Step Into** is somewhat different if you click **Toggle Disassembly/ Instruction Step Mode** on the Debug view toolbar, or when the current routine has no debugging information. When this mode is set, the step buttons cause instruction-level steps to be executed instead of source-level steps. Also, the Disassembly view will be shown instead of the Editor.

To single-step without going into other subroutines, click **Step Over** instead of **Step Into**.

While stepping through a program, you may conclude that the problem you are interested in lies in the current subroutine's caller, rather than at the stack level where your process is suspended. Click the Debug view's **Step Return** button in that situation: execution continues until the current subroutine completes, then the debugger regains control in the calling statement.

Additional Run Control Options

The **Run > Stack Frame** menu provides other options for manipulating files.

Drop To Frame

The debugger resumes until the execution returns to the selected stack frame.

Run To Frame Address

The debugger resumes execution until the address of the selected stack frame is reached.

Set Breakpoint at *Frame Function*

Select this to create an expression breakpoint at *Frame Function* of the selected stack frame, with *Frame Function* replaced by the actual function name. This action is only available if the symbol data for the selected frame is present (and the function name is known).

Set PC to *Frame Function*

Select this to set the Program Counter register to the beginning of the *Frame Function* of the selected stack frame, with *Frame Function* replaced by an actual function name. This action also is only available if the symbol data for the selected frame is present (and the function name is known).

18.2.4 Using Debug Modes

Depending on the type of connection the debugger has to the target, the debugger may be capable of operating in different modes. Different debug modes have different capabilities and limitations, which are mostly related to how the debugger interacts with the target and the processes that are being debugged.

Your target connection type determines possible modes as follows:

- KGDB connection type—Only supports debugging the kernel using a single execution context. When the system context is suspended, the kernel, kernel threads, and user processes are suspended also.
- Usermode agent connection type—Supports debugging user processes. Processes and threads within processes are suspended and resumed independently of each other.
- Dual mode connection type—In dual mode, you must toggle between user and kernel connection type depending on your debugging needs:
  - Kernel mode (also called System mode)—Only supports debugging the kernel using a single execution context. When the system context is suspended, the kernel, kernel threads, and user processes are suspended also.
  - User mode—Supports debugging user processes. Processes and threads within processes are suspended and resumed independently of each other.

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

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

For example, if the system maintains network connections with other systems, suspending it will cause the others to lose their network connections with the debugged system.

18.2.5 Setting and Recognizing the Debug Mode of a Connection

Right-clicking on a connection in the Target Manager or the Debug view and selecting **Target Mode** allows you to specify a debug mode for the connection. The currently active mode is indicated by a checkmark.

When you create a new debug connection through a launch, the connection debug mode (either system mode or task mode) is saved as a property of the launch. This mode is listed in parentheses at the end of the label of the target node in the Debug view.

Switching Debug Modes

For target connections that support switching between modes, if you switch the debug mode while a debug connection is active, this debug connection will become unavailable in the Debug view. When a debug connection is unavailable, no operations can be performed on it, except for disconnecting the debug connection.

In the Target Manager, if you switch the target to system mode, every node in the tree will have a system mode icon painted on top. If the system mode icon does not appear, then the node and processes are in task or user mode.

18.2.6 Debugging Multiple Target Connections

You can debug processes on the same target using multiple target connections simultaneously. An example of this setup is a Linux target that has a user mode **ptrace** agent installed for debugging processes, and an OCD connection for halting the system and debugging the kernel.

In this situation, if the system is halted using the OCD (system mode) target connection, the user mode **ptrace** agent will also be halted, and the user mode target connection will be lost. When the system is resumed, the user mode target connection will be re-established.

The Target Manager and the Debug view (if a debug session is active) both provide feedback in this scenario. The Target Manager hides all the process information that was visible for the target, and displays a label **back-end connection lost** next to the target node. The Debug view does not end the active debug session, but it shows it as being **unavailable**, in the same manner as if the debug mode was switched.

18.2.7 Disconnecting and Terminating Processes

Disconnecting from a process or core detaches the debugger, but leaves the process or core in its current state.

Terminating a process actually kills the process on the target.

NOTE: If the selected target supports terminating individual threads, you can select a thread and terminate only that thread.

18.2.8 Changing Source Lookup Settings

The purpose of Source Lookup is to map the debugger source file paths to the actual source file locations in the workspace and the host file system. The debugger paths for source files are the paths that are read by the debugger from the symbol data of the executable that is being debugged. These paths were generated by the compiler when the executable was built, and the are often different from the paths to those files at time of debugging.

For information about how to set up Source Lookup Path settings, see the *Wind River Workbench User Interface Reference: Source Lookup Path Dialog*.

18.3 Using the Disassembly View

Use the Disassembly view:

- To examine a program when you do not have full source code for it (such as when your code calls external libraries).
- To examine a program that was compiled without debug information.
- When you suspect that your compiler is generating bad code (the view displays exactly what the compiler generated for each block of code).

18.3.1 Opening the Disassembly View

Unlike other Workbench views, the Disassembly view is not accessible from the **Window > Show View** menu—it appears automatically if the Debug view cannot display the appropriate source code file in the Editor (it appears as a tab in the Editor, labelled with the target connection being debugged).

The Disassembly view can be opened manually by clicking the Debug View's **Toggle Disassembly/Instruction Step Mode** toolbar button, and by right-clicking in the **Stack Trace View** and selecting **Go To Code**.

18.3.2 Understanding the Disassembly View Display

The Disassembly view shows source code from your file (when available), interspersed with instructions generated by the compiler. As you step through your code, the Disassembly view keeps track of the last four instructions where the process was suspended. The current instruction is highlighted in the strongest color, with each previous step fading in color intensity.

If the Disassembly view displays a color band at the top and bottom (here, the band is blue), then it is pinned to the process with that color context in the Debug view; if no color band is displayed, then the view will update as you select different processes in the Debug view.

For more information, see *Wind River Workbench User Reference: Disassembly View.*

18.4 Java-JNI Cooperative Debugging

Java-JNI (Java native interface) cooperative debugging allows you to debug the Java side of your Java application with the JDT debugger, and the JNI side of your application with a C/C++ native debugger, simultaneously. You can use Java-JNI cooperative debugging when you have a Java project that has Java classes with native methods, and have built the associated native libraries with debug information.

In addition to your Java classes, the JRE, and the native libraries required to run your application on the target, you must have the special helper class and its associated library file on the target as well. These are provided by Wind River and are required for Workbench to perform Java-JNI cooperative debugging.

The helper class file and its associated native library are available in the directory *installDir***/linux2-x/usermode-agent/2.0/bin/wrjnidebug/**. Copy the contents of this directory (the **wrjnidebughelper.ja**r file and the **libwrjnidebughelper.so** file) to a directory on the target. You will need to refer to the location of these files when creating your launch configuration as described in *Creating a Launch Configuration for Cooperative Debugging*, p.240.

Configuring a User Mode Connection for Cooperative Debugging

If you do not have a user mode connection to your target, create a new connection (see *3.8 Configuring a Target Connection*, p.35 for details). Be sure to specify the port number if you started **usermode-agent** with a port number.

If your target's root file system is accessible to the host computer on which you are running Workbench, specify the root file system in the **Target Server Options** page under the Target Connection properties for your connection. Right-click on your target connection in the Target Manager and select **Properties** to access this tab.

If, however, your target's root file system is not fully accessible to the host computer, then you need to manually add the mappings under **Object Path Mappings** in the Target Connection properties for your connection. Right-click on your target connection and select **Properties** to access this tab. Be sure to configure

your object path mapping so that the native debugger can locate the object files for the following:

- The **java** executable (for example, **/usr/bin/java**)

  Without access to symbols from the java executable, the native debugger will not be able to detect any shared libraries that the Java application loads, and hence you will not be able to debug your JNI code.

- The Linux run time loader (**ld.so** or **ld-linux.so**, usually located in **/lib**)

  Without access to symbols of the runtime loader, the native debugger will not be able to automatically detect shared libraries, such as your JNI libraries, as the Java VM loads them. This will again prevent you from debugging your JNI code.

- The JNI libraries that are part of your application

  Without access to symbols from your JNI libraries, you will not be able to debug your JNI library even if the native debugger can detect that the library has been loaded.

Note that if none of the target file system is accessible, you may need to copy the necessary object files from the target to a location that the host has access to and then specify that location in your object path mappings.

Creating a Launch Configuration for Cooperative Debugging

Use the following procedure to create a launch configuration for cooperative debugging. For details on all options available when creating a launch configuration, refer to *Wind River Workbench User Interface Reference Manual: Launch Configuration Dialog* online.

1. Select **Run > Debug**, select **Remote Java Launch and Connect**, and click **New** to create a new launch configuration.
2. In the **Main** tab:
   - Enter the name of your Java project for **Project** or select it by browsing to it.
   - Enter your main class for **Main class** or select **Search** to select it from a list.
   - Check **Stop in main** (optional, but will be used in this example).
3. In the **Connection** tab:
   - Select your target connection in the **Connection to Use** drop-down list.

- Specify the target path to the Java executable in **Exec Path on Target**, for example, **/usr/bin/java**.
- Click **Edit** in **Environment** and remove the **LD_BIND_NOW** entry. Click **Add** to add an entry with a **Name** of **LD_LIBRARY_PATH.** For **Value,** specify the target path to the directory or directories on the target where the JNI library (or libraries) that your application will call reside.

4. In the **Arguments** tab:
   - Under **Classpath**, enter the full target paths to both the directory where your Java classes for your application reside, and the full path of the **wrjnidebughelper.jar** file, located where you copied the contents of the **wrjnidebug** directory. Separate multiple paths by colons. For example:

     ```
     /usr/myapp:/usr/myapp/jni:/usr/agent/wrjnidebug/wrjnidebughelper.jar
     ```

     Note that these are the paths as visible to programs running on the target, not host paths.
5. In the **Debug Options** tab:
   - Specify a debug port for **Port Number**. This should be a unique number not used as a port by any other application including **usermode-agent**, and it should be greater than 1024 which is the last of the reserved port numbers, and less than 65536.
   - Check **Enable Debugger Cooperative Mode for JNI**.

Debugging In Java and Native Modes

Once you have created your launch configuration for cooperative debugging you can begin debugging and switch between Java and native modes as described in the following procedure.

1. Click **Debug** to start your application.

   Your application will come up stopped in its main class (if you selected **Stop in main** in the Main tab). In the Debug view you should see two hierarchical trees—one for the Java debugger (**Java HotSpot™ Client VM**) and one for the native debugger (**java**). Execute your Java application until it reaches a statement where there is a call to a native method and then click **Step Into**. It should step into the native method.

   If you did not configure the helper files location correctly, you will get an error message indicating that JNI transitions are disabled.

2. To return from JNI debugging to Java debugging, click **Resume** (not **Step Return**) to complete the step. Workbench will stop the Java thread and you can continue with debugging the Java side.
3. You can now move between the native and Java debuggers at any time and set breakpoints in either—as long as the Java debugger is not disabled (discussed next). For example, open any Java or native source file that is part of your application and double-click in the gutter to set a breakpoint.

Conditions that Disable the JDT Debugger

With Java-JNI cooperative debugging, the native debugger has control over the entire process including the JVM. Because of this, it is possible for the entire Java application including the JVM to be stopped by the native debugger. This will happen, for example, if you choose to manually suspend the entire process from the native debugger.

You can also set a native breakpoint in your JNI library that will stop the entire process. This can result in all the threads of the application, including the ones communicating with the JDT Java debugger, to be suspended. When this happens, the JDT debugger is not able to debug the Java side as it is unable to communicate with the JVM.

In Workbench, the Java debugger entries in the Debug View will display the message "**(debugger is disabled)(may be out of synch)**", indicating that the Java debugger is disabled and the Debug view display of Java information may no longer be accurate. In this condition, you cannot perform any Java debugging operations such as planting Java breakpoints, inspecting Java variables, and so on.

Re-Enabling the JDT Debugger

Click **Resume** to resume the entire process from the native debugger (not just one thread). The Java debugger will re-establish communication with the JVM and become enabled. You can now continue with Java-JNI cooperative debugging.

18.5 Remote Kernel Metrics

Remote kernel metrics (RKMs) are operating system signals (metrics) that are dynamically collected by the **rkm_monitor_linux** target agent. The metrics can be displayed in real-time utilizing the full-color features of the StethoScope GUI included with Workbench.

The RKM monitor acquires its data from the **/proc** filesystem on the target. If you do not have a **/proc** filesystem on your target, you may simply need to mount it, or you may need to rebuild your kernel to include it.

To mount the **/proc** filesystem, use the **mount** command as follows:

```
# mount -t proc proc /proc
```

If your kernel does not have **/proc** support built-in, you must re-build the kernel and enable it in the **File system** configuration section of your kernel configuration tool.

Building and Running the RKM Monitor

The RKM monitor agent is supplied in a Workbench sample project. Use the following procedure to build the RKM monitor and then run it on your Linux target.

NOTE: The following procedure assumes the results of your project build are available on the target, for example by locating your workspace on a shared NFS mount. It also assumes you have created a connection to the target in the Target Manager and have specified the target-host root filesystem mapping.

1. Start Workbench.
2. Select **File >New >Example >Native Sample Project**. Click **Next.**
3. Select **The RKM (Remote Kernel Metrics) Monitor Program** and click **Finish**.
4. In the Project Manager, right-click on **rkm_monitor_linux** and select **Build Project**.
5. Select **Run > Run** and select **Process on Target**.
6. In the Main tab, name it something like **rkm_monitor_linux** and choose your target connection from the **Connection to use** pull-down menu.
7. Click **Run** and **rkm_monitor** will start on the target as shown in your **Debug** view.

Running the RKM Monitor From the Command Line

You can also start the RKM monitor by specifying it on the command line. To see the various options available, enter the following from the directory containing the **rkm_monitor_linux** executable that you built:

```
$ ./rkm_monitor_linux -help
```

For example, to start the RKM monitor with a different index value, say 125 instead of the default 127, enter:

```
$ ./rkm_monitor_linux -index=125 &
```

In this way you can run multiple monitors which has several advantages. You might want to configure one monitor, for example, to collect a few signals for all processes at a low sampling frequency, and configure another to sample a complete set of metrics for a few processes at a high frequency.

By specifically selecting the signals you want to monitor, you can reduce the memory, CPU, and network resources required to monitor the large set of signals selected by default. In addition, the source for **rkm_monitor_linux** is included so you can create versions that monitor specific signals that are not made available by the default configuration, or even monitor specific portions of an application.

As another example, you might just want to monitor memory usage for the root user, taking 10 samples every second:

```
$ ./rkm_monitor_linux -samples=10 -processes user=root -sysmetrics memory &
```

When you attach StethoScope to the RKM monitor invoked as shown, you will only be able to view root memory usage.

Attach StethoScope to the RKM Monitor

Use the following procedure to view the remote kernel metrics in the StethoScope GUI that comes with Workbench.

1. With **rkm_monitor** running on the target, open StethoScope in Workbench by selecting **ScopeTools > StethoScope**.
2. Select Linux as your target type and click **OK**.
3. Select the correct target connection from the pull-down menu.

4. Enter 127 as the index value. This is the default to use for StethoScope with the RKM monitor. Use a different index number if you want to run another monitor on the target; this allows you to run multiple monitors on the target, selecting them by index number.
5. Click **OK**.

   A StethoScope windows opens.

Using StethoScope to View Remote Kernel Metrics

Your main navigation tool for StethoScope is the Signal Tree in the upper-left corner. As an example of how to use StethoScope with remote kernel metrics, use the following procedure.

1. Expand **System** and then **time**.
2. Check the **time** group to automatically check all of the metrics under **time** (**user**, **lowuser**, **system**, **idle**) which then display in units of jiffies (1/100th of a second CPU time) spent executing user, low-priority user, and system tasks, as well as the number of jiffies spent idle.
3. Hover your mouse over the StethoScope toolbar to find the **Zoom to Fit** icon. Click on it to contain all the signals in the graph window. Note that each signal has a unique color associated with it and this color is used for the lines in the graph. The MiniMonitor view lists the current values of monitored signals and the MiniDump view lists the value of each signal at each sampling.

By default, when any new processes are started they are added to the monitor. Note, however, that the buffer is reset when new signals are added, so you lose the history of what you had been monitoring. You can avoid this by, for example, monitoring only your own processes with appropriate command line options as described in *Running the RKM Monitor From the Command Line*, p.244.

For more information on the use of StethoScope, see your StethoScope documentation.

18.6 Run/Debug Preferences

For more information, see *Wind River Workbench User Reference: Debug View.*

19 Analyzing Core Files

19.1 Introduction

You can configure your target system to save status and memory information on programs that crash for specific reasons. For example, you can specify that the information should be saved if a process exceeds some size, or tries to access memory outside of its allowed memory space. This information is then saved on the target in a file called the core file, or core dump, typically named **core.***pid* where *pid* is the process ID of the program that crashed. The core dump is an ELF file that contains an image of the process memory space, details of the process state, and additional information at the time of the crash. You can then transfer the core file to your host and analyze the cause of the crash using the Workbench debugger at any time.

19.2 Acquiring Core Dump Files

With the bash shell, you control the creation of core dumps with the **ulimit** command. For example, to determine your current settings enter:

```
target_$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) 20
pending signals                 (-i) 1024
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 11768
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
target_$
```

Most Linux systems come with the default setting to not allow core file generation. In the example this is shown as a core file size setting of **0**. You can turn on core file generation by specifying a maximum size or **unlimited** as the core file size, for example:

```
target_$ ulimit -c unlimited
```

Certain conditions, for example when a program tries to access restricted memory, will then generate core dumps, as in:

```
target_$ ./segtest.out
Segmentation fault (core dumped)
```

You can also turn on core file generation by setting a core file size (**unlimited** or as specified), and then cause a core file to be generated by setting a limit on some condition. For example:

```
target_$ ulimit -c
0
target_$ ulimit -c 12000
target_$ ulimit -c
12000
target_$ ulimit -m
unlimited
target_$ ulimit -m 2000
target_$ ulimit -m
2000
```

The shell is now set to generate a core file of a maximum size of 12000 KB if the memory size for a process exceeds 2000 KB.

NOTE: Your **ulimit** settings entered at the command line apply to the current shell only. If you want to continue to generate core dumps across logins, add the **ulimit** commands to a shell startup file, for example to **.bash_login**.

Transfer the core dump to your host if it is not already on a file system that your target mounts from the host. You can now analyze the core file using the Workbench debugging tools.

19.3 Attaching Workbench to a Core File

Use the Target Manager to create a connection to the core file. You do not need to be connected to the target if you have access to the target file system because core file analysis takes place on the host. Note that when you are using Workbench to analyze the core file, you are not debugging an active process, you are only examining the state and conditions of a process at the moment it failed to determine the cause of the failure.

1. Click the **Create a New Target Connection** button in the Target Manager, select **Wind River Linux Application Core Dump Target Server Connection**, and click **Next**.
2. Enter or browse to the path for the core dump file you wish to analyze in the **Core dump file** field.
3. You can enter the CPU number for **Force CPU number** and the version of Linux for **Force OS version**. The CPU number for your CPU can be found in the text file *$WIND_FOUNDATION_PATH*/**resource/target/architecturedb**. For example, the **architecturedb** file shows that the CPU number to enter for an XScale CPU is **1120**:

```
...
[CPU_1120]
cpuname = xscale
cpuFamilyName = XSCALE
...
```

The OS version value to enter is either **2.4** or **2.6**.

NOTE: The core file does not contain information on the type of CPU or the version of the operating system that it was created with. If you get a **Failed to connect target** error message indicating a target-CPU mismatch, it is likely that specifying the CPU number and OS version will resolve it.

4. Enter or browse to the path of the application image that created the core dump file in the **File** field.
5. The command line your selections have created is displayed at the bottom of the dialog. To add additional options for memory cache size, logging, and symbol loading, click the **Edit** button next to the **Options** field and make your selections. Click **Next**.
6. Specify the location of the target root by clicking on **Add** and entering the **Target path** (for example **/**) and the **Host path** (for example **/target/rootfs**), click **OK,** and then click **Next**.
7. Click **Next** in the **Target Refresh Dialog** box and then click **Finish** in the **Connection Summary** box.

   Your core file connection appears in the Target Manager.

You can now connect to the core dump by right-clicking on the "stopped" process in the Target manager and selecting **Attach to Process**. The Debug view will show the debugger attached to the process at the point of the failure and an editor window will open at the error location in the source file.

Core File Analysis

You can now perform various activities on the core file, for example view a stack trace, a memory dump, a thread list, local and global variables, and register values. But remember this is only a read-only view of the process at the time of the core dump.

Ending the Session

To end the core file debugging session, disconnect in the Debug view and disconnect in the Target Manager.

20 Troubleshooting

20.1 Introduction

This chapter displays some of the errors or problems that may occur at different points in the development process, and what steps you can take to correct them. It also provides information about the log files that Workbench can collect, and how you can create a ZIP file of those logs to send to Wind River support.

If you are experiencing a problem with Workbench that is not covered in this chapter, please see the *Wind River Workbench Release Notes* for your platform.

20.2 Startup Problems

This section discusses some of the problems that might cause Workbench to have trouble starting.

Workspace Metadata is Corrupted

If Workbench crashes, some of your settings could get corrupted, preventing Workbench from restarting properly.

1. To test if your workspace is the source of the problem, start Workbench, specifying a different workspace name.

   On Windows

   Select **Start > Programs > Wind River > Workbench 2.6 > Wind River** Workbench **2.6**, then when Workbench asks you to choose a workspace, enter a new name (**workspace2** or whatever you prefer).

   Or, if the Workbench startup process does not get all the way to the Workspace Launcher dialog box, or does not start at all, start it from a terminal window:

   > *installDir*`\workbench-2.6\wrwb\platform\eclipse\x86-win32\bin\wrwb.exe -data` *newWorkspace*

   On Linux or Solaris

   Start Workbench from a terminal window, specifying a new workspace name:

   > `./startWorkbench.sh -data` *newWorkspace*

   NOTE: For more information on Workbench startup options, see **Help > Help Contents > Wind River Partners Documentation > Eclipse Workbench User Guide > Tasks > Running Eclipse**.

2. If Workbench starts successfully with a new workspace, exit Workbench, then delete the **.metadata** directory in your original Workbench installation (*installDir***/workspace/.metadata**).

3. Restart Workbench using your original workspace. The **.metadata** directory will be recreated and should work correctly.

4. Because the **.metadata** directory contains project information, that information will be lost when you delete the directory.

   To recreate your project settings, reimport your projects into Workbench (**File > Import > Existing Project into Workspace**).

.workbench-2.6 Directory is Corrupted

1. To test if your *homeDir*/**.workbench-2.6** directory is the source of the problem, rename it to a different name, then restart Workbench.

NOTE: Make sure you rename the *homeDir*/**.workbench-2.6** directory, not the *installDir*/**workbench-2.6** directory.

2. If Workbench starts successfully, exit Workbench, then delete the old version of your *homeDir*/**.workbench-2.6** directory (the one you renamed).
3. Restart Workbench. The *homeDir*/**.workbench-2.6** will be recreated and should work correctly.
4. Because the **.workbench-2.6** directory contains Eclipse configuration information, any information about manually configured Eclipse extensions or plug-ins will be lost when you delete the directory.

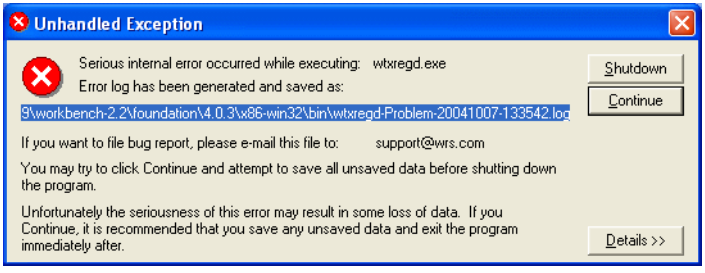
   To make them available again within Workbench, re-register them (**Help > Software Updates > Manage Configuration**).

Registry Unreachable (Windows)

When Workbench starts and it does not detect a default Wind River registry, it launches one. After you quit Workbench, the registry is kept running since it is needed by all Wind River tools. You do not need to ever kill the registry.

If you do stop it, however, it stores its internal database in the file *installDir*/**workbench-2.6/foundation/.wind/wtxregd.***hostname*.

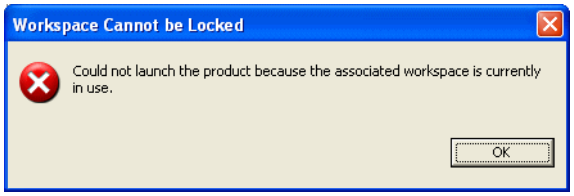
If this file later becomes unwritable, the registry cannot start, and Workbench will display an error.



This error may also occur if you install Workbench to a directory to which you do not have write access, such as installing Workbench as an administrator and then trying to run it as yourself.

Workspace Cannot be Locked (Linux and Solaris)

If you start Workbench and select a workspace, you may see a **Workspace Cannot be Locked** error.



There are three possible causes for this error:

1. Another user has opened the same workspace. A workspace can only be used by one user at a time.
2. You installed Workbench on a file system that does not support locking.

   Use the following command at a terminal prompt to start Workbench so that it creates your workspace on a file system which does allow locking, such as a directory on a local disk:

   `./startWorkbench.sh -configuration` *directory that allows locking*

   For example:

   `./startWorkbench.sh -configuration /usr/local/`*yourName*

   NOTE: For more information on Workbench startup options, see **Help > Help Contents > Wind River Partners Documentation > Eclipse Workbench User Guide > Tasks > Running Eclipse**.
3. On some window managers (e.g. gnome) you can close the window without closing the program itself and deleting all running processes. This results in running processes maintaining a lock on special files in the workspace that mark a workspace as open.

   To solve the problem, kill all Workbench and Java processes that have open file handles in your workspace directory.

20.2.1 Pango Error on Linux

If the file **pango.modules** is not world readable for some reason, Workbench will not start and you may see an error in a terminal window similar to

```
** (<unknown>:21465): WARNING **: No builtin or dynamically loaded modules
were found. Pango will not work correctly. This probably means there was an
error in the creation of:
   '/etc/pango/pango.modules'
You may be able to recreate this file by running pango-querymodules.
```

Changing the file's permissions to **644** will cause Workbench to launch properly.

20.3 General Problems

If you are experiencing a problem with Workbench that is not covered in this chapter, please see the *Wind River Workbench Release Notes* for your platform.

20.3.1 JDT Dependency

Some third party plug-ins are dependent on JDT. If a plug-in you are interested in requires JDT, you should download it from the official Eclipse Web site:

http://download.eclipse.org/downloads/drops/R-3.0-200406251208/download.php?dropFile=eclipse-JDT-3.0.zip

A list of official mirror sites is here:

http://www.eclipse.org/downloads

20.3.2 Help System Does Not Display on Linux

Workbench comes preconfigured to use Mozilla on Linux, and it expects it to be in your path. If Mozilla is not installed or is not in your path, you must install and set the correct path to the browser or Workbench will not display help or other documentation.

To manually set the browser path in Workbench:

1. Select **Window > Preferences > Help**.

2. Click **Custom Browser (user defined program)**, then in the **Custom Browser command field** type or browse to your browser launch program, click **OK**.

   Sample browser launch commands are **"/usr/bin/firefox" %1**, **"kfmclient openURL %1"**, and **"/opt/mozilla/mozilla %1"**. Enter the command line as appropriate for your browser.

20.3.3 Help System Does Not Display on Windows

The help system can sometimes fail to display help or other documentation due to a problem in McAfee VirusScan 8.0.0i (and possibly other virus scanners as well).

For McAfee VirusScan 8.0.0i, the problem is known to be resolved with patch10 which can be obtained from Network Associates. As a workaround, the problem can be avoided by making sure that McAfee on-access-scan is turned **on** and allowed to scan the **TEMP** directory as well as ***.jar** files.

More details regarding this issue have been collected by Eclipse Bugzilla #87371 at **https://bugs.eclipse.org/bugs/show_bug.cgi?id=87371**.

20.3.4 Resetting Workbench to its Default Settings

If Workbench crashes, some of your settings could get corrupted, preventing Workbench from restarting properly. To reset all your settings to their defaults, delete your *$HOME*/**.workbench-2.6** directory which will be recreated when Workbench restarts.

⚠ **CAUTION:** Remove the directory **.workbench-2.6** (begins with a "dot") in your home directory, not the directory **workbench-2.6** in the Workbench installation directory.

20.4 Error Messages

Some errors display an error dialog box directly on the screen, while others that occurred during background processing only display this icon in the lower right corner of Workbench window.

Hovering your mouse over the icon displays a pop-up with a synopsis of the error. Later, if you closed the error dialog box but want to see the entire error message again, double-click the icon to display the error dialog box or look in the *Eclipse Log*, p.267.

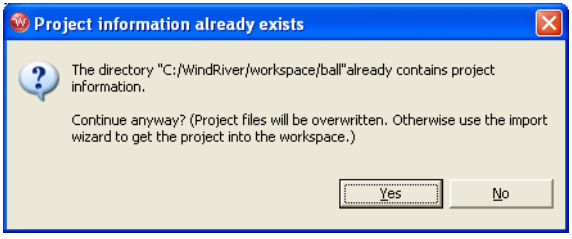
This section explains error messages that appear in each Workbench component.

20.4.1 Project System Errors

For general information about the Project System, see *6. Projects Overview*.

Project Already Exists

If you deleted a project from the Project Navigator but chose not to delete the project contents from your workspace, then you try to create a new project with the same name as the old project, you will see the following error:



If you click **Yes**, your old project contents *will be overwritten* with the new project. If you want to recreate the old project in Workbench, click **No**, then right-click in the Project Navigator, select **Import**, then select **Existing Project into Workspace**.

Type the name of your old project, or browse to the old project directory in your workspace, click **OK**, then click **Finish**. Your old project will appear in the Project Navigator.

Cannot Create Project Files in Read-only Location

When Workbench creates a project, it creates a **.wrproject** file and other metadata files it needs to track settings, preferences, and other project-specific information. So if your source files are in a read-only location, Workbench cannot create your project there.

To work around this problem, you must create a new project in your workspace, then create a folder that links to the location of your source files.

1. Create a User-defined Project in your workspace by selecting **File > New > User-Defined Project**. The Target Operating System dialog box appears.
2. Select a target operating system from the drop-down list, then click **Next**. The Project dialog box appears.
3. Type in a name for your project, select **Create project in workspace**, then click **Next**.
4. Click **Next** to accept the default settings in the next dialog boxes, then click **Finish** to create your project.
5. In the Project Navigator, right-click your new project and select **New > Folder**. The **Folder** dialog box appears.
6. Type in a name for your folder, then click **Advanced** and select the **Link to folder in the file system** checkbox.
7. Type the path or click **Browse** and navigate to your source root directory, then click **OK** to create the new folder.
8. Click the plus next to the folder to open it, and you will see the source files from your read-only source directory. Eclipse calls items incorporated into projects in this way **linked resources**.

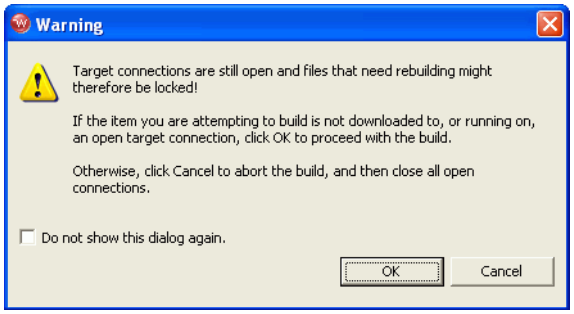
NOTE: This mechanism cannot be used for managed-build projects, only for user-defined projects.

20.4.2 Build System Errors

For general information about the Build System, see *11. Building Projects*.

Building Projects While Connected to a Target

If you right-click a project in the Project Navigator and select **Build Project** while you have a target connection active in the Target Manager, you will see this error:



This dialog box warns you that a build may fail because the debugger may still have a lock on your files. You can continue your build by clicking **OK**, but be advised that when you see an error message in the Build Console similar to **dld: Can't create file XXX: Permission denied** you will need to disconnect your target and restart the build.

The best workflow for cases where you continually need to rebuild objects that are in use by your target is as follows:

- Create a launch configuration for your debugging task. When you need to disconnect your target in order to free your images for the build process, the launch configuration allows you to automatically connect, download, and run your process with a single click once the build is finished.

  You can even specify that your project should be rebuilt before it is launched by selecting **Window > Preferences > Run/Debug > Launching**, and then selecting **Build (if necessary) before launching**. For more information about launch configurations, see *16. Launching Programs*.

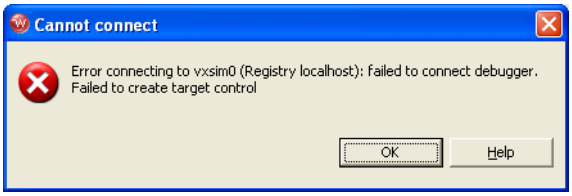
- When you work with processes, make sure that your process is terminated before you rebuild or relaunch. You can then safely ignore the warning (and check the **Do not show this dialog again** box).
- When you work with Downloadable Kernel Modules or user-built kernel images, just let the build proceed. If the **Link error** message appears, either disconnect your target or unload all modules, then rebuild or relaunch.

20.4.3 Target Manager Errors

For general information about the Target Manager, see *13. Connecting to Targets*.

Troubleshooting Connecting to a Target

If you see the following error:



Or if you have other trouble connecting to your target, try these steps:

1. Check that the target is switched on and the network connection is active. In a terminal window on the host, type:

   ping *n.n.n.n*

   where **n.n.n.n** is the IP address of your target.

2. Verify the target **Name/IP address** in the **Edit the Target Connection** dialog box (right-click the target connection in the Target Manager then select **Properties**.)

3. Choose the actual target CPU type from the drop-down list if the **CPU type** in the **Edit the Target Connection** dialog box is set to **default from target**.

4. Verify that a target server is running. If it is not:

   a. Open the Error Log view, then find and copy the message containing the command line used to launch the target server.

   b. Paste the target server command line into a terminal window, then hit **ENTER**.

   c. Check to see if the target server is now running. If not, check the Error Log view for any error messages.

5. Check if the **dfwserver** is running (on Linux and Solaris, use the **ps** command from a terminal window; on Windows, check the Windows Task Manager). If multiple **dfwservers** are running, kill them all, then try to reconnect.

6. Check that the WDB connection to the target is fully operational by right-clicking a target in the Target Manager and selecting **Target Tools > Run WTX Connection Test**. This tool will verify that the communication link is correct. If there are errors, you can use the WTX and WDB logs to better track down what is wrong with the target.

Exception on Attach Errors

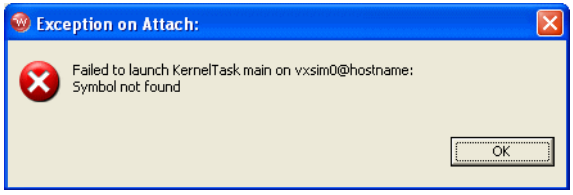
If you try to run a task and the Target Manager is unable to comply, it will display an **Exception on Attach** error containing useful information.

Build errors can lead to a problem launching your task or process; if one of the following suggestions does not solve the problem, try launching one of the pre-built example projects delivered with Workbench.

If the host shell was running when you tried to launch your task or process, try closing the host shell and launching again.

Error When Running a Task Without Downloading First

You will see the following error if you try to run a kernel task without first downloading it to your target:



Processes can be run directly from the Project Navigator, but kernel tasks must be downloaded before running. Right-click the output file, select **Download**, fill in the Download dialog box, then click **OK**.

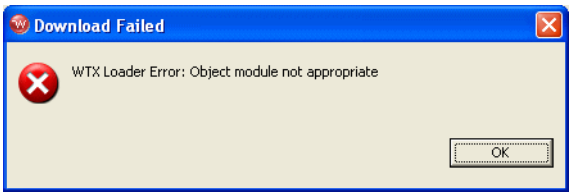
If you see this error and you did download the file, open a host shell for your connection, and try to run the task from the host shell. Type:

```
lkup entrypoint
```

to see if your entry point is there.

Downloading an Output File Built with the Wrong Build Spec

If you built a project with a build spec for one target, then try to download the output file to a different target (for example, you build the project for the simulator, but now you want to run it on a hardware target), you will see this error:

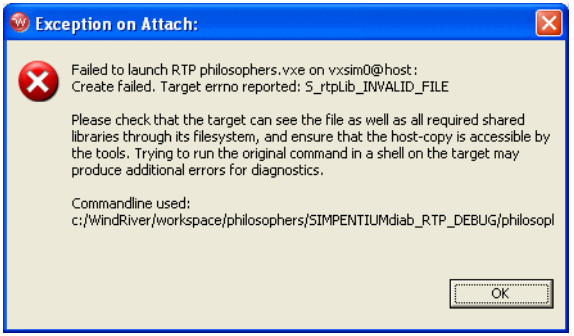


To select the correct build spec, right-click the output file in the Project Navigator, select **Set Active Build Spec**, select the appropriate build spec from the dialog box, then rebuild your project.

Your project should now download properly.

Error if Exec Path on Target is Incorrect

If the **Exec Path on Target** field of the **Run Real-time Processes** dialog box does not contain the correct target-side path to the executable file (if, for example, it contains the equivalent host-side path instead) you will see this error:



If the target-side path looks correct but you still get this error, recheck the path you gave.

Even if you used the **Browse** button to locate the file, it will be located in the host file system. The Object Path Mapping that is defined for your target connection will translate it to a path in the target file system, which is then visible in the Exec Path edit field. If your Object Path Mapping is wrong, the Exec Path will be wrong, so it is important to check.

Troubleshooting Running a Process

If you have trouble running your process from the **Run Process** or **Run Real-time Process** dialog boxes, try these steps:

1. If the error **Cannot create context** appears, verify that the **Exec Path on Target** is a path that is actually visible on the target (and doesn't contain the equivalent host-side path instead).
   a. Right-click the process executable in the Project Navigator or right-click **Processes** or **Real-time Processes** in the Target Manager and select **Run Real-time Process.**
   b. Copy the exec path and paste it into the **Output View > Target Console Tab** (at the bottom of the view). Verify that the program runs directly on the target.
2. If the program runs but symbols are not found, manually load the symbols by right-clicking the process and selecting **Load Symbols**.
3. Check your **Object Path Mappings** to be sure that target paths are mapped to the correct host paths. See *13.5.2 Object Path Mappings*, p.172 for details on setting up your Object Path Mappings.
   a. Open a host shell and type:
   ```
   ls execpath
   ```
   If you have a target shell, type the same command.
   b. In the host shell, type:
   ```
   devs
   ```
   to see if the prefix of the Exec Path (for example, **host:**) is correct.
4. If the Exec Path is correct, try increasing the back-end timeout value of your target server connection (see *Advanced Target Server Options*, p.170 for details).
5. From a target shell or Linux console, try to launch the process.
6. Verify that the kernel node in the Target Manager view has a small **S** added to the icon, indicating that symbols have been loaded for the Kernel.

a. If not, verify that the last line of your **Object Path Mappings** table displays a target path of **<any>** corresponding to a host path of **<leave path unchanged>**.

20.4.4 Launch Configuration Errors

If a launch configuration is having problems, delete it by clicking **Delete** below the **Debug** dialog box **Configurations** list.

If you cannot delete the launch configuration using the **Delete** button, navigate to *installDir*/**workspace/.metadata/.plugins/org.eclipse.debug.core/.launches** and delete the **.launch** file with the exact name of the problematic launch configuration.

NOTE: Do *not* delete any of the **com.windriver.ide.*.launch** files.

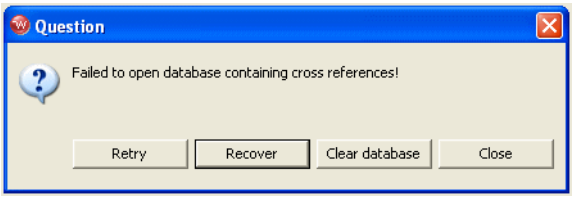
Troubleshooting Launch Configurations

If you click the **Debug** button (or click the **Debug** button from the **Launch Configuration** dialog box) and get a "Cannot create context" error, check the **Exec Path** on the **Main** tab of the **Debug** dialog box to be sure it is correct. Also check your **Object Path Mappings**. See *13.5.2 Object Path Mappings*, p.172 for information about Object Path Mappings.

For general information about launch configurations, see *16. Launching Programs*.

20.4.5 Static Analysis Errors

If at any point Workbench is unable to open the cross reference database, you will see this error:



There are many reasons the cross reference database may be inaccessible, including:

- The database was not closed properly at the end of the last Workbench session running within the same workspace. This happens if the process running Workbench crashed or was killed.
- Various problems with the file system, including wrong permissions, a network drive that is unavailable, or a disk that is full.

You have several choices for how to respond to this error dialog box:

- **Retry**—the same operation is performed again, possibly with the same failure again.
- **Recover**—the database is opened and a repair operation is attempted. This may take some time but you may recover your cross reference data.
- **Clear Database**—the database is deleted and a new one is created. All your cross reference data is lost and your workspace will be reparsed the next time you open the call tree.
- **Close**—the database is closed. No cross reference data is available, nor will it be generated. At the beginning of the next Workbench session, an attempt to open the database will be made again.

20.5 Error Log View

See the *Wind River Workbench User Interface Reference: Error Log View.*

20.6 Error Logs Generated by Workbench

Workbench has the ability to generate a variety of useful log files. Some Workbench logs are always enabled, some can be enabled using options within Workbench, and some must be enabled by adding options to the executable command when you start Workbench.

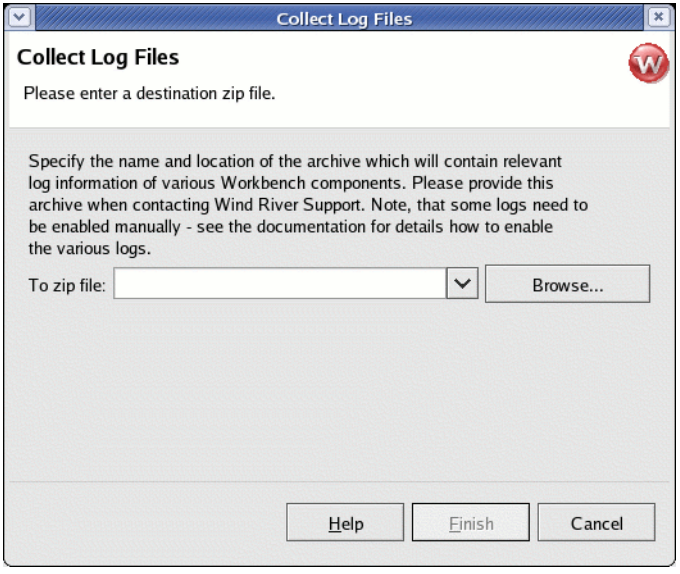
This section describes the logs, tells you how to enable them (if necessary), and how to collect them into a ZIP file you can send to Wind River support representatives.

20.6.1 Creating a ZIP file of Logs

Once all the logs you are interested in have been enabled, Workbench automatically collects the information as you work.

To create a ZIP file to send to a Wind River support representative:

1. Select **Help > Collect Log Files**. The dialog box opens.



2. Type the full path and filename of the ZIP file you want to create (or browse to a location and enter a filename) then click **Finish**. The ZIP file is created in the specified location, and contains all information collected to that point.
3. To discontinue logging (for those logs that are not always enabled) uncheck the boxes on the Target Server Options tab, or restart Workbench without the additional options.

20.6.2 Eclipse Log

The information displayed in the *20.5 Error Log View*, p.265 is a subset of this log's contents.

How to Enable Log

This log is always enabled.

What is Logged

- All uncaught exceptions thrown by Eclipse java code.
- Most errors and warnings that display an error dialog box in Workbench.
- Additional warnings and informational messages.

What it Can Help Troubleshoot

- Unexpected error alerts.
- Bugs in Workbench java code.
- Bugs involving intercomponent communication.

Supported?

Yes.

20.6.3 DFW GDB/MI Log

The GDB/MI log is a record of all communication between the debugger back end (the "debugger framework", or DFW) and other views within Workbench, including the Target Manager, debugger views, and OCD views.

How to Enable Log

This log is always enabled.

What is Logged

All commands sent between Workbench and the debugger back end.

What it Can Help Troubleshoot

Debugger and Target Manager-related bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.4 DFW Debug Tracing Log

How to Enable Log

This log is always enabled.

What is Logged

Internal exceptions in the debugger back end.

What it Can Help Troubleshoot

The debugger back end.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.5 Debugger Views GDB/MI Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-vmargs -DDFE.Debug=true
```

What is Logged

Same as *DFW GDB/MI Log*, p.267, except with Workbench time-stamps.

What it Can Help Troubleshoot

Debugger and Target Manager-related bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.6 Debugger Views Internal Errors Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-vmargs -DDFE.Debug=true
```

What is Logged

Exceptions caught by the Debugger views messaging framework.

What it Can Help Troubleshoot

Debugger views bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.7 Debugger Views Broadcast Message Debug Tracing Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-vmargs -DDFE.Debug=true
```

What is Logged

Debugger views internal broadcast messages.

What it Can Help Troubleshoot

Debugger views bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.8 Target Server Output Log

This log contains the messages printed by the target server while running. These messages typically indicate errors during various requests sent to it, such as load operations. Upon startup, if a fatal error occurs (such as a corefile checksum mismatch) then this error will be printed before the target server exits.

How to Enable Log

- Enable this log from the Target Manager by right-clicking the target connection, selecting **Properties**, selecting the **Target Server Options** tab, then clicking **Edit**.

  Select the **Logging** tab, then check the box next to **Enable output logging** and provide a filename and maximum file size for the log. Click **OK**.

- Enable this log from the command line using the **-l** *path/filename* and **-lm** *maximumFileSize* options to the target server executable. For more information about target server commands, see **Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference > tgtsvr**.

What is Logged

- Fatal errors on startup, such as library mismatches and errors during exchange with the registry.
- Standard errors, such as load failure and RPC timeout.

What it Can Help Troubleshoot

- Debugger back end
- Target Server
- Target Agent

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.9 Target Server Back End Log

This log records all requests sent to the WDB agent.

How to Enable Log

- Enable this log from the Target Manager by right-clicking the target connection, selecting **Properties**, selecting the **Target Server Options** tab, then clicking **Edit**.

  Select the **Logging** tab, then check the box next to **Enable backend logging** and provide a filename and maximum file size for the log. Click **OK**.

- Enable this log from the command line using the **-Bd** *path/filename* and **-Bm** *maximumFileSize* options to the target server executable. For more information about target server commands, see **Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference > tgtsvr**.

What is Logged

Each WDB request sent to the agent. For more information about WDB services, see **Wind River Documentation > References > Host API and Command References > Wind River WDB Protocol API Reference**.

What it Can Help Troubleshoot

- Debugger back end
- Target Server
- Target Agent

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.10 Target Server WTX Log

This log records all requests sent to the target server.

How to Enable Log

- Enable this log from the Target Manager by right-clicking the target connection, selecting **Properties**, selecting the **Target Server Options** tab, then clicking **Edit**.

  Select the **Logging** tab, then check the box next to **Enable WTX logging** and provide a filename and maximum file size for the log. Click **OK**.

- Enable this log from the command line using the **-Wd** *path/filename* and **-Wm** *maximumFileSize* options to the target server executable. For more information about target server commands, see **Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference > tgtsvr**.

What is Logged

Each WTX request sent to the target server. For more information about WTX services, see **Wind River Documentation > References > Host API and Command References > WTX Reference > wtxMsg.**

What it Can Help Troubleshoot

- Debugger back end
- Target Server
- Target Agent

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.6.11 Target Manager Debug Tracing Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-debug -vmargs -Dcom.windriver.ide.target.DEBUG=1.
```

What is Logged

Target Manager internal debug errors.

What it Can Help Troubleshoot

Inconsistencies in the debugger back end.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

20.7 Technical Support

If you have questions or problems with Workbench or with Wind River Linux after completing the above troubleshooting section, or if you think you have found an error in the software, please contact the Wind River Customer Support organization (contact information is listed in the release notes for your platform). Your comments and suggestions are welcome.

PART VII

Updating

20 Integrating Plug-ins

20.1 Introduction

Because Wind River Workbench is based on Eclipse, you can incorporate new modules into Workbench without having to recompile or reinstall it. These new modules are called *plug-ins*, and they can deliver new functionality and tools to your copy of Wind River Workbench.

Many developers enjoy creating new plug-ins and sharing their creations with other Eclipse users, so you will find many Web sites with interesting tools and programs available for you to download and incorporate into your Workbench installation.

Some plug-ins are dependent on Java Development Tools (JDT), which is automatically installed when you install Workbench (the install option is called **Wind River Java Development Tools**).

20.2 Finding New Plug-ins

In addition to the Eclipse Web site, **http://www.eclipse.org**, many other Web sites offer a wide variety of Eclipse plug-ins. Here are a few:

http://www.eclipse-plugins.info/eclipse/plugins.jsp

http://www.eclipseplugincentral.com/

http://eclipse-plugins.2y.net/eclipse/

http://www.sourceforge.net/

20.3 Incorporating New Plug-ins into Workbench

Many developers who download plug-ins prefer to create a new directory for each one, rather than unzipping the files directly into their Workbench installation directory. There are many advantages to this approach:

- The default Workbench installation does not change.
- You do not lose any of your plug-ins if you update or reinstall Workbench.
- Plug-ins do not overwrite each other's files.
- You know which files to replace when an update to the plug-in is available.

20.3.1 Creating a Plug-in Directory Structure

To make your plug-ins easier to manage, create a directory structure for them outside your Workbench installation directory.

1. Create a directory to hold your plug-ins. It can have any descriptive name you want, for example, **eclipseplugins**.
2. Inside this directory, create a directory for each plug-in you want to install. These directories can also have any descriptive name you want, for example, **clearcase**.

NOTE: Before continuing, download the plug-in's **.zip** or other archive file and look at its contents. Some plug-ins provide the **eclipse** directory structure and the **.eclipseextension** file for you, others do not.

- If the destination path for the files begins with **eclipse**, and you see an **.eclipseextension** file in the list, you may skip the rest of this section and extract the plug-in's files into the directory you created in step 2.
- If the destination path begins with **plugins** and **features**, then you must complete the rest of the steps in this section.

3. Inside each plug-in directory, create a directory named **eclipse**. This directory *must* be named eclipse, and a separate **eclipse** directory is required inside each plug-in directory.
4. Inside each eclipse directory, create an empty file named **.eclipseextension**. This file *must* be named **.eclipseextension** (with no **.txt** or any other file extension), and a separate **.eclipseextension** file is required inside each **eclipse** directory.
5. Extract your plug-in into the **eclipse** directory. Two directories, called **features** and **plugins**, appear in the directory alongside the **.eclipseextension** file.

NOTE: For any plug-in to work properly, its **features** and **plugins** directories as well as an empty file called **.eclipseextension** *must* be located inside a directory called **eclipse**.

20.3.2 Installing a ClearCase Plug-in

Once you have created a plug-in directory structure and have found a plug-in you want to use with Workbench, download and install it according to the instructions provided by the plug-in's developer (almost every plug-in comes with release notes containing installation instructions).

This section will show you how to download and install a plug-in on Windows.

Downloading the IBM Rational ClearCase Plug-in

Wind River recommends the IBM Rational ClearCase plug-in.

1. Follow steps 1 and 2 in *20.3.1 Creating a Plug-in Directory Structure*, p.278 (the IBM ClearCase plug-in creates the **eclipse** directory and the **.eclipseextension** file for you.)

   For the purposes of this example, name the top-level directory **eclipseplugins**, and name the plug-in directory **clearcaseIBM**.

2. Navigate to **http://www-128.ibm.com/developerworks/rational/library/1376.html** and click the **Plug-ins** link under **ClearCase**. The Rational ClearCase Plug-ins page opens.

3. Click the **Download** link to the right of the appropriate version of the package file. For this example, select **IBM Rational ClearCase SCM adapter for Eclipse 3.1: Windows** (this file works for Eclipse 3.2 as well).

4. Extract the **.zip** file to your **/eclipseplugins/clearcaseIBM** directory. The **eclipse** directory is created for you, and inside are two directories, called **features** and **plugins**, alongside the **.eclipseextension** file.

Adding Plug-in Functionality to Workbench

1. Before starting Workbench, make sure that **/usr/atria/bin** (where the ClearCase tools are installed) is in your path.

2. Start Workbench, then select **Help > Software Updates > Manage Configuration**. The **Product Configuration** dialog appears.

3. Select **Add an Extension Location** in the Wind River Workbench pane.

4. Navigate to your **eclipseplugins/***plug-in***/eclipse** directory. Click **OK**.

5. Workbench will ask if you want to restart. To properly incorporate ClearCase functionality, click **Yes**.

Incorporating the IBM Rational Plug-in

1. When Workbench restarts, activate the plug-in by selecting **Window > Customize Perspective**.

2. In the **Customize Perspective** dialog, switch to the **Commands** tab.

3. Select the **ClearCase** option in the **Available command groups** column, then click **OK**. A new **ClearCase** menu and icons appear on the main Workbench toolbar.

4. From the **ClearCase** menu, select **Connect to Rational ClearCase** to activate ClearCase functionality.

To configure the ClearCase plug-in, select **Window > Preferences > Team > ClearCase SCM Adapter**.

For more information about using the ClearCase plug-in, see **Help > Help Contents > Rational ClearCase SCM Adapter**.

For more information about ClearCase functionality, refer to your ClearCase product documentation.

20.4 Disabling Plug-in Functionality

You can disable plug-in functionality without uninstalling the downloaded files. This gives you the opportunity to re-enable them at a later time if you want.

1. To disable a plug-in, select **Help > Software Updates > Manage Configuration**. The **Product Configuration** dialog appears.
2. In the left column, open the folder of the plug-in you want to uninstall, select the plug-in itself, then click **Disable**.
3. Workbench will ask if you want to restart. To properly disable the plug-in's functionality, click **Yes**.

20.5 Managing Multiple Plug-in Configurations

If you have many plug-ins installed, you may find it useful to create different configurations that include or exclude specific plug-ins.

When you make a plug-in available to Workbench, its extension location is stored in the Eclipse configuration area.

When starting Workbench, you can specify which configuration you want to start by using the **-configuration** *path* option, where *path* represents your Eclipse configuration directory.

On Windows:

From a shell, type:

```
% cd installdir\workbench-2.x\wrwb\platform\eclipse\x86-win32\bin
% .\wrwb.exe -configuration path
```

On Linux and Solaris:

Use the option as a parameter to the **startWorkbench.sh** script:

```
% ./startWorkbench.sh -configuration path &
```

For more information about using **-configuration** and other Eclipse startup parameters, see **Help > Help Contents > Wind River Partners Documentation > Eclipse Workbench User Guide > Tasks > Running Eclipse**.

21 Using Workbench in an Eclipse Environment

21.1 Introduction

It is possible to install Workbench in a standard Eclipse environment, though some fixes and improvements that Wind River has made to Workbench will not be available.

21.2 Recommended Software Versions and Limitations

Java Runtime Version

Wind River tests, supports, and recommends using the JRE 1.5.0_08 for Workbench plug-ins.

Wind River adds a package to that JRE version, and not having that package will make the Terminal view inoperable.

Eclipse Version

Workbench 2.6 is based on Eclipse 3.2. Wind River patches Eclipse to fix some Eclipse debugger bugs. These fixes will be lost when using a standard Eclipse environment.

See the getting started for your platform for supported and recommended host requirements for Workbench 2.6.

Defaults and Branding

Eclipse uses different default preferences from those set by Workbench. The dialog described in *21.3 Setting Up Workbench*, p.284 allows you to select whether to use Workbench preferences or existing Eclipse preferences.

In a standard Eclipse environment, the Eclipse branding (splash screen, welcome screen, etc.) is used instead of the Wind River branding.

21.3 Setting Up Workbench

This setup requires a complete Eclipse and Workbench installation. Follow the respective installation instructions for each product.

1. From within Workbench, select **Help > Register into Eclipse**. The Register into Eclipse dialog appears.
2. In the **Directory** field, type in or **Browse** to your Eclipse 3.2 directory.
3. In the **Registration Options** section, select **Use Wind River default preferences**, or leave it unselected to maintain existing Eclipse preferences.

   If you decide to use Wind River default preferences, some changes you will notice are that autobuild is disabled, and the Workbench Application Development perspective and help home become the defaults.

4. If you decided to maintain existing Eclipse preferences you can still use the much faster Wind River (index based) search engine by leaving **Use Wind River search engine** selected. To use the Eclipse default search engine, unselect it.
5. If you want to track the installation process, leave **Log installation process** selected (click **Browse** to change the path where the file should be created). Uncheck it if you do not want Workbench to create a log file.
6. When you are done, click **Finish**. Workbench will be available the next time you launch Eclipse. No special steps are necessary to launch Eclipse.

NOTE: Any errors discovered during installation appear in the Error Log view.

21.4 Using CDT and Workbench in an Eclipse Environment

The following tips will help you understand how to use Eclipse C/C++ Development Tooling (CDT) and Workbench together in the same Eclipse environment.

NOTE: When starting Eclipse after registering Workbench, you will see three errors in the Error Log.

These errors are not a problem. They appear because Workbench ships some CDT plug-ins that are already on your system, and Eclipse is reporting that the new ones will not be installed over the existing ones.

21.4.1 Workflow in the Project Navigator

Some menus and actions are slightly different when using CDT and Workbench together.

Application Development Perspective (Workbench)

CDT projects appear in this perspective along with Workbench projects.

Building CDT Projects

The context menu of the Project Navigator contains entries for **Build Project** and **Rebuild Project**, but the **Rebuild Project** entry executes a normal build for CDT projects. The **Clean Project** entry is missing for CDT projects.

Running Native Applications

The **Run Native Application** menu is enabled for CDT projects. When executed, it creates a Workbench Native Application launch with correct parameters. Because Workbench Native Application launches do not support debugging, to debug your application you must create a CDT **Local C/C++ Application** launch from the **Run > Run As** menu.

Selecting Projects to Build

When selecting multiple projects (including Workbench and CDT projects) and executing any build action, the build action is only executed on Workbench projects.

Displaying File and Editor Associations

The Workbench Project Navigator displays icons for the default editor of a file, if file associations have been defined. If CDT is the default editor, the corresponding icons will also show up in the Application Development perspective.

C/C++ Perspective (CDT)

Static Analysis

Static analysis is available from the context menu of the Project Navigator.

Building Workbench Projects

CDT **Build Project** and **Clean Project** actions are enabled for Workbench projects, and they execute the appropriate build commands correctly.

Working with Workbench Binary Targets

There are no actions to directly run, debug or download a Workbench project's binary target in this perspective.

21.4.2 Workflow in the Build Console

Application Development Perspective (Workbench)

When adding a CDT project as a sub-project (project reference) to a Workbench project, the **Clear Build Console** flag is ignored when executing a build on this project.

C/C++ Perspective (CDT)

Executing a build on a Workbench project from this perspective correctly opens the Workbench Build Console.

General

When navigating to errors from the Workbench Build Console or the Problems view, the file containing the error opens in the assigned editor.

21.4.3 Workflow in the Editor

Opening Files in an Editor

The editor that should be used for files cannot be determined. It depends on the settings defined in the appropriate **plugin.xml** files, and on the order in which the Workbench and CDT plug-ins are loaded.

Only one default editor can be associated with each file type, and it is the same for both perspectives. Files can be opened with the **Open With** menu, allowing you to select the editor. When executed, that editor is associated with, and becomes the default for, this specific file.

NOTE: To assign a default editor for all files with a given signature, you must define a file association in the preferences by selecting **Window > Preferences**, then choosing **General > Editors > File Associations**.

For example, to add a default editor for all ***.c** files, click **Add** and enter ***.c**. The list of available editors appears. Select one, then click **Default**.

21.4.4 Workflow for Debugging

Workbench and CDT Perspectives

Regardless of any direct file association created using the **Open With** command, the default editor opens when debugging a file.

For example, associating ***.c** files with the default Workbench editor opens the Workbench editor in the CDT Debug and the Workbench Device Debug perspectives.

The reverse is also true: if you associate a file type with the CDT editor, it will open when those files are debugged even if you have made an association with a different editor using **Open With**.

22

Using Workbench with Version Control

22.1 Introduction

This chapter provides tips on using Workbench with version-controlled files, which Workbench project files you should add to version control when archiving your projects, and how to manage build output when your sources are version controlled.

22.2 Using Workbench with ClearCase Views

When using Workbench with ClearCase dynamic views, create your workspace on your local file system for best performance. For recommendations about setting up your workspaces and views, see **Help > Help Contents > Rational ClearCase SCM Adapter > Concepts > Managing workspaces**.

Wind River does not recommend that you place the Eclipse workspace directory in a view-private directory. If you create projects in the default location under the

workspace directory, ClearCase prompts you to add the project to source control. This process requires all parent directories to be under source control, including the workspace directory.

Instead, create workspace directories outside of a ClearCase view. If you want to create projects under source control, you should unselect the **Create project in workspace** check box in the project creation dialog and then navigate to a path in a VOB.

In addition, you should also redirect all build output files to the local file system by changing the **Redirection root directory** in the **Build Properties > Build Paths** tab of your product. All build output files such as object files and generated Makefiles will be redirected.

For more information about the redirecting build output and the redirection root directory, open the build properties dialog, press the help key for your host, and see the *Build Paths* section.

22.2.1 Adding Workbench Project Files to Version Control

To add Workbench project files to version control without putting your workspace into a ClearCase view, check-in the following automatically generated files along with your source files:

Project File	Description
.project	Eclipse platform project file containing general information about the project.
.wrproject	Workbench project file containing mostly general build properties.
.wrfolder	Workbench project file containing folder-level build properties (located in subfolders of your projects).
.wrmakefile	Workbench managed build makefile template used to generate Makefiles.
***.makefile**	Workbench managed build extension makefile fragments (e.g some Platform projects)

For user-defined projects, all Makefile files need to be version controlled, too.

You should avoid manually adding source files to any build macro in any project type containing absolute paths—they should be substituted by environment variables (provided by **wrenv** for example) wherever possible.

NOTE: The **.metadata** directory should not be version controlled, as it contains mostly user- and workspace-specific information with absolute paths in it.

For more information about IBM Rational ClearCase, see **http://www-130.ibm.com/developerworks/rational/products/clearcase**.

Choosing Not to Add Build Output Files to ClearCase

After installing the ClearCase plugin, you may be prompted to add any build output files to ClearCase.

There are two ways to avoid this if you wish:

1. Using Workbench Preferences.
   a. Open the **Window > Preferences > Team > ClearCase SCM Adapter** preferences page.
   b. From the **When new resources are added** pull-down list, select **Do nothing**.
2. Using **Derived Resource** option.
   a. Configure your build so the build output goes into one (or a few) well-known directories such as **bin** or **output**.
   b. Check in the empty **bin** or **output** directories into ClearCase.
   c. In the Project Navigator, right-click the directory you checked in, select **Properties**, and on the **Info** page, select **Derived**.
   d. From now on, the Clearcase plug-in will not prompt you about **Derived** resources.

NOTE: If you use Workbench managed builds, they will automatically mark the build output directories as derived so ClearCase will not try to add the build output files to source control. If you use a different builder, you may have to configure it to mark resources as derived.

PART VIII

Reference

A Host Shell

A.1 Overview

The host shell is a host-resident command shell that provides a GDB command line interface. It allows you to download, monitor and debug applications using a subset of the standard GDB commands. The host shell also provides a Tcl interpreter allowing you to write simple Tcl scripts to interface with the GDB interpreter's synchronous commands. This section describes how to begin using the host shell from a Linux or Solaris host. For more detailed reference information on the host shell, see *A.2 Host Shell Commands and Options*, p.301.

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

managed by the debugger framework, resides on the host, although the information it contains refers to the target system.

To start the host shell, perform the following steps:

1. Source the Wind River environment script.
2. Run the **usermode-agent** command.
3. Run the **hostShell** command.

Each of these steps is described in the following sections.

Running the Host Shell

NOTE: Before you can run the host shell, you must have a DFW session name and a target definition name in the registry. Once you have used the Workbench GUI (which registers the DFW session name) and created a target connection with the Target Manager (to your localhost or any other host), you will then have the necessary registry configuration to proceed with using the host shell as described in this section.

To start running the host shell, use the following procedure. (The old procedure of using **windsh** *tgtsvrname* to start the host shell as described in *A.2.9 Deprecated Commands*, p.322 is deprecated and may not be supported in future versions.)

1. Source the environment script.

   Change directory to the Workbench installation directory and use the **eval** command to source the Wind River environment script as shown in the following example:

   ```
   $ cd WindRiver/
   $ eval `./wrenv.sh -p linux-2.x -o print_env -f sh`
   $
   ```

   NOTE: Users of the **csh** should substitute **-f csh** for the **-f sh** shown in the example.

   For more information on **wrenv.sh**, see *Initializing Your Environment*, p.301.

2. Execute the **usermode-agent** binary on your host. For example, from the Workbench installation directory, enter:

   ```
   $ ./linux-2.x/usermode-agent/1.1/bin/ia/i386/usermode-agent &
   [1] 3127

       WDB AGENT 1.1 READY
   (c) Copyright Wind River Systems Inc. 2005
   ```

```
All rights reserved.

$
```

3. Start the host shell. You can start the host shell by providing a session name and target definition at the command line, or you can be prompted to enter these values interactively.

   a. To start the host shell with a session name and target definition, enter:
   `$ hostShell -ds dfw-session-name -dt target-definition-name`

   The required *dfw-session-name* and *target-definition-name* are stored in the Workbench registry.

   Every running debugger server, automatically started by the GUI or manually started by the user, registers itself automatically in the Workbench registry. To locate the *dfw-session-name* use the following commands:

   ```
   $ wtxtcl
   wtxtcl> wtxInfoQ .* dfwserver
   {dfw-wb24-philb dfwserver {host;VAN-KENEACH;hostip;147.11.80.58;…}}
   ```

   The first tag (in the example, **dfw-wb24-philb**) is the one needed by the **-ds** option.

   The target definition entries are present in the Workbench registry and are traditionally set by the Target Manager wizard, or may be set by the user. To locate the *target-definition-name* use the following command:

   ```
   wtxtcl> wtxInfoQ .* tgtconncfg
   {philb_1126537537278 tgtconncfg {{_map_;generic;… }}}
   {philb_1126598342655 tgtconncfg {{_map_;generic,… }}}
   wtxtcl>
   ```

   The first tag (in the example, **philb_1126537537278**) is the one needed by the **-dt** option.

   In this example, your full command line to start the host shell would be:

   ```
   $ hostShell -ds dfw-wb24-philb -dt philb_112653753727
   ```

   b. Alternatively, you can just specify **hostShell** on the command line without any arguments to be prompted interactively. You will be shown a list from which to select the debugger server and target definition. For example:
   `$ hostShell`

   ```
   List of DFW servers available:
       dfw-wb24-wbuser

   Please select a DFW server amongst the list above (enter '.' to
   skip):
   dfw-wb24-wbuser
   ```

```
List of target connection definitions available:
    wbuser_1127844872638 (tgt_192.168.29.121)

Please select a target connection definition amongst the list above
(enter '.' to skip): wbuser_1127844872638
```

With either method of host shell invocation, once you have supplied a session name and target server, the host shell will start up:

```
Checking License... OK

Creating new target connection 'linux_native_localhost_wbuser' -OK.
Detected target plugin 'unifiedtargetplugin'.
Establishing direct WTX connection with target server
'linux_native_localhost_wbuser'.
Loading the 'gdb' shell interpreter... OK.
```


```
  Copyright 1995-2005 Wind River Systems, Inc.

(gdb)
```

The default prompt for the host shell is the GDB **(gdb)** prompt.

Using GDB

You can now use **GDB**. For example, to get help on **GDB** commands:

```
(gdb) help
List of classes of commands:

 aliases -- Aliases of other commands
 breakpoints -- Making program stop at certain points
 data -- Examining data
```

```
 files -- Specifying and examining files
 internals -- Maintenance commands
 obscure -- Obscure features
 running -- Running the program
 stack -- Examining the stack
 status -- Status inquiries
 support -- Support facilities
 tracepoints -- Tracing of program execution without stopping the program
 user-defined -- User-defined commands

 Type "help" followed by a class name for a list of commands in that
class.
 Type "help" followed by command name for full documentation.
 Command name abbreviations are allowed if unambiguous.
(gdb)
```

Some typical functions and their commands follow.

Insert breakpoints:

```
(gdb) break functionname
(gdb) break filename:line number
```

Continue:

```
(gdb) c
```

Display stack:

```
(gdb) bt
```

Display variables:

```
(gdb) p var
```

Disconnect from the target:

```
(gdb) disconnect
```

Quit GDB

```
(gdb) quit
```

Using KGDB

This section describes how to configure a host shell for Linux kernel debugging with KGDB. It assumes that a DFW server is already running and that a target connection has been configured.

Start the host shell as described in *Starting the Host Shell*, p.302.

You can begin kernel debugging at the GDB prompt. Use the command **attach kernel** to select the Linux kernel as working context. This stops the kernel. You are then able to debug the kernel with the supported GDB APIs. A sample session is shown below.

```
(gdb) attach kernel
(gdb)
(gdb) bt
#0  breakpoint () at kgdb.c:1689
#1  tasklet_action () at softirq.c:254
#2  __do_softirq () at softirq.c:97
#3  do_softirq () at irq.c:176
#4  irq_exit () at handle.c:83
#5  do_IRQ () at irq.c:104
(gdb) break schedule
Breakpoint 3 at 0xC03EE658: file sched.c, line 2541.
(gdb) continue
Continuing.

Breakpoint 3, schedule () at thread_info.h:91
91                      __asm__("andl %%esp,%0; ":"=r" (ti) : "0"
(~(THREAD_SIZE - 1)));
(gdb) list
85
86
87              /* how to get the thread information struct from C */
88              static inline struct thread_info *current_thread_info(void)
89              {
90                      struct thread_info *ti;
91                      __asm__("andl %%esp,%0; ":"=r" (ti) : "0"
(~(THREAD_SIZE - 1)));
92                      return ti;
93              }
94
(gdb)
```

A.2 Host Shell Commands and Options

This section provides details about operating the host shell.

A.2.1 Host Shell Basics

This section describes how to initialize your environment and start the host shell.

Initializing Your Environment

To use the tools efficiently from the command line, you need to configure some environment variables and other settings. The best way to do this is with the **wrenv** environment utility, which sets up a development shell based on information in the **install.properties** file.

When using the Workbench tools from the command line, always begin by invoking the environment utility as shown in *Invoking wrenv*, p.301. The **wrenv** utility, which is also run by the IDE on startup, guarantees a consistent, portable execution environment that works from the IDE, from the command line, and in automated build systems. Throughout this guide, whenever host operating-system commands are shown or described, it is assumed that you are working from a properly configured development shell created by **wrenv**.

Invoking wrenv

Assuming a standard installation of Workbench, you can invoke **wrenv** as follows.

UNIX

From your operating-system shell prompt, enter:

```
% eval `installDir/wrenv.sh -p linux-2.x -o print_env -f shell`
```

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

```
% eval `./wrenv.sh -p linux-2.x -o print_env -f sh`
```

to invoke **wrenv** from within your installation directory for the **sh** shell.

Windows

You can invoke **wrenv** from the command prompt by entering:

```
C:\> installDir\wrenv.exe -p linux-2.x
```

Workbench also supplies a fully configured Windows version of the Z shell (**sh.exe**). The Z shell, sometimes called **zsh**, gives Windows users a UNIX-like command-line interface.

Starting the Host Shell

To start the host shell, enter:

```
$ hostShell -ds dfw-session-name -dt target-definition-name
```

For more information, see *Starting the Host Shell*, p.302.

Host Shell Initialization Script

If you create the file *installDir*/**.wind/wb/windsh.tcl**, and Tcl commands it contains are executed at host shell startup.

Stopping the Host Shell

Regardless of how you start it, you can terminate a host shell session by typing **quit** at the prompt or pressing **CTRL+D**.

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

Option	Description
-c, **-command** *expr*	Execute expression & exit shell (batch mode).
-dt *target*	DFW target definition name.
-ds *session*	DFW server session to use.
-dp *port*	DFW server port to use.
-e, **-execute** *expr*	Execute Tcl expression after initialization.
-h, **-help**	Print help.
-host *host*	Retrieve target server data from host's registry.
-m, **-mode** *mode*	Indicate mode to start either Tcl or Gdb
-n, **-noinit**	Do not read home Tcl initialization file.

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

Option	Description
-p, **-poll** *ms*	Set event poll interval in milliseconds [default=200].
-q, **-quiet**	Do not echo script commands as they are executed.
-r [*pathname*]	Root pathname mappings.
-s, **-startup** *file*	Startup file of shell commands to execute.
-T, **-Tclmode**	Start in Tcl mode.
-v, **-version**	Print Wind River host-based shell version.

Switching Interpreters

At times you may want to switch between the Tcl interpreter and the GDB interpreter. From a prompt, type these special commands followed by **ENTER**:

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

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

For example, to evaluate a Tcl interpreter command from within the GDB interpreter, type:

```
(gdb) tcl puts -nonewline "hello tcl" ; tcl expr 4 + 2
hello tcl
6
```

Setting Shell Environment Variables

The host shell has a set of environment variables that configure different aspects of the shell's interaction with the target and with the user. These environment variables can be displayed and modified using the Tcl routine **shConfig**. Table A-2 provides a list of the host shell's environment variables and their significance.

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

Table A-2 **Shell Environment Variables**

Variable	Meaning
ROOT_PATH_MAPPING	This variable indicates how host and target paths should be mapped to the host file system on which the DFW server used by the host shell is running. If this value is not set, a direct path mapping will be assumed (for example, a pathname given by **/folk/user** will be searched, no translation to another path will be performed). See *A.2.2 Root Path Mapping*, p.305 for further details on this variable.
PAGING_NUMBER	Indicates the paging number. For example the **list** call in the GDB interpreter may return a long list of source code that will need multiple pages to be displayed entirely. Changing the paging number allows you to display **PAGING_NUMBER** lines of text for each page.
LINE_LENGTH	Indicates the maximum number of characters permitted in one line of the host shell's window.
INTERPRETER Tcl | Gdb	Indicates the host shell's current interpreter mode and permits the user to switch from one mode to another.
LINE_EDIT_MODE	Sets the line edit mode to use. Set to **emacs** or **vi**. Default is **vi**. For example, to switch from vi mode to emacs mode, from the GDB interpreter, enter: `(gdb) tcl shConfig LINE_EDIT_MODE emacs`

A.2.2 Root Path Mapping

The root path mapping environment variable is used by both the host shell and the debugger framework as a matching mechanism between the host and target paths.

Positioning this variable is often mandatory because host and target paths can be located on different machines, different file systems, or visible through different NFS mounting points.

This environment variable is made of a list of couples of paths, with the syntax:

[*targetpath1*,*hostpath1*][*targetpath2*,*hostpath2*] . . . **[,]**

The client tools apply a substitution on the target paths, replacing the matching pattern *targetpath* by the corresponding *hostpath*.

For example, if the processes to debug on your linux target, **labou**, are located under **/etc/usr/procs/** and if the debugger server is running on another UNIX machine, accessing the **procs** directory through NFS **/net/labou/etc/usr/procs**, then you will need to use **[,/net/labou]** as root path mapping.

This way the debugger server will be able to locate the processes to debug from the target path.

For example, you can launch a host shell with this root path mapping by running the following:

$ **hostShell -r "[,/net/labou][,]" -ds** *dfw-session-name* **-dt** *target-definition-name*

The **[,]** sentinel is mandatory.

A.2.3 Using the Tcl Interpreter

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

Running the Tcl Interpreter

To start the host shell in the Tcl interpreter, use the **-m** option:

$ **hostShell -ds** *dfw-session-name* **-dt** *target-definition-name* **-m Tcl**

To switch to the Tcl interpreter from GDB mode, type a question mark (**?**) at the **(gdb)** prompt; the prompt will change to **tcl>** to remind you of the shell's new mode. If you want to use a Tcl command without changing to Tcl mode, type a **?** followed by a space character before your line of Tcl code.

If you want to switch to the Tcl interpreter, enter **tcl** or **?** the host shell's prompt will change to **tcl>**. To switch back to the GDB interpreter from the Tcl interpreter, enter **gdb**, and the prompt will change to **(gdb)**.

Scripting the GDB Interpreter with Tcl

It is possible to write Tcl scripts to interface with the host shell's GDB interpreter. When calling the GDB interpreter from within the Tcl interpreter the output of the call is displayed on the host shell's UI, and that output may also be directed to Tcl variables. A typical use case would be a Tcl script that loops waiting for a variable to change to a particular value, stepping the source code until the value has changed. The following code illustrates such a script written using the host shell's Tcl interpreter interfacing with the GDB interpreter:

```
# load the application "nextTest"
tcl> gdb file /folk/laurac/bin/nextTest
Reading symbols from /folk/laurac/bin/nextTest...done.
# set a breakpoint on the function "main"
tcl> gdb break main
Breakpoint 1 at 0x63000326: file nextTest.c, line 22.
# run the application - the breakpoint will be hit
tcl> gdb run
Starting program: /folk/laurac/bin/nextTest
Breakpoint 1, main () at nextTest.c:22
22                  i = dummy ();
# list the application's source code
tcl> gdb list
16                   }
17
18              int main ()
19                 {
20                 int i, j;
21
22                 i = dummy ();
23                 i += (5 * (i * 200) - 33);
24                 j = 200;
25                 return i;

# loop reading the value of the variable j, when it reaches 200 exit the loop
tcl> while {1} {
=> set var [gdb print j]
=> if {[regexp 200 $var]} {
=> break
=> }
=> gdb step
=> }

tcl> gdb print j
%6 = 200
# the loop has exited, read the registers at this point
```

```
tcl> set registers [gdb info registers]
eax     =       0x000000C8
ecx     =       0x630263F8
edx     =       0x630260E8
ebx     =       0xFFFFFFFD
esp     =       0x63023EF0
ebp     =       0x63023EF8
esi     =       0x00000000
edi     =       0x00000000
eflags  =       0x00000202
pc      =       0x63000343
st0     =       0x00000000000000000000
st1     =       0x00000000000000000000
st2     =       0x00000000000000000000
st3     =       0x00000000000000000000
st4     =       0x00000000000000000000
st5     =       0x400AB3F0000000000000
st6     =       0x4010A8BFC00000000000
st7     =       0x401DAAAAAAA800000000
fpcr    =       0xFFFF027F
fpsr    =       0xFFFF0000
fptag   =       0xFFFFFFFF
# continue the application
tcl> gdb c
Continuing.
```

Accessing Low Level GDB/MI APIs

The Tcl interpreter provides access to the low level GDB/MI APIs., which allows you to interact with the debugger framework.

To send a GDB/MI request to the debugger framework, you can use the gdb **mi** command followed by the GDBMI command, for example:

```
tcl> gdb mi "-wrs-log"
^done,ls="/wind/river/DFW121/host/x86-linux2/bin/dfwserver.log",lt="/view/phi
lb.62/wind/river/DFW121/host/x86-linux2/bin/dfwstatus.log"
tcl>
```

In this example, the GDBMI command **-wrs-log** is executed.

NOTE: It is only possible to execute synchronous GDB commands with the Tcl interface.

A.2.4 Using the GDB Interpreter

To see a list of the command classes available in GDB mode, enter help at a **(gdb)** prompt. Table A-3 lists the command classes available with GDB. For more information on a class, enter **help** *class* to see a list of available commands in that class. To get help on a specific command, enter **help** *command*.

Table A-4 lists general commands available within the GDB interpreter.

Table A-3 **GDB Command Classes**

Command Class	Command Class Description
aliases	Aliases of other commands.
breakpoints	Making program stop at certain points.
data	Examining data.
files	Specifying and examining files.
internals	Maintenance commands.
obscure	Obscure features.
running	Running the program.
stack	Examining the stack.
status	Status inquiries.
support	Support facilities.
tracepoints	Tracing of program execution without stopping the program.
user-defined	User-defined commands.

General GDB Commands

Table A-4 lists general commands available within the GDB interpreter.

Table A-4 **General GDB Interpreter Commands**

Command	Description
help *command*	Print a description of the command.
cd *directory*	Change the current directory.
pwd	Show the current directory.
path *path*	Append *path* to the **path** variable.
show path	Show the **path** variable.
echo *string*	Echo the string.
list *line* | *symbol* | *file:line*	Display 10 lines of a source file, centered around a line number or symbol.
shell *command*	Run a shell command (such as **ls** or **dir**).
source *scriptfile*	Run a script of GDB commands.
directory *dir*	Append *dir* to the **directory** variable (for source file searches.)
q[uit]	Quit the GDB interpreter.

Working with Breakpoints

Table A-5 shows commands available for setting and manipulating breakpoints.

Table A-5 **GDB Interpreter Breakpoint Commands**

Command	Description
break *line* | *function* | **address*	Set a breakpoint at the specified line number, function name, or address.
condition	Specify breakpoint number *N* to break only if *COND* is true. Usage is **condition** *N COND*, where *N* is an integer and *COND* is an expression to be evaluated whenever breakpoint *N* is reached.

Table A-5 **GDB Interpreter Breakpoint Commands**

Command	Description
delete *[breakpointid [breakpointid ...]]*	Delete breakpoints specified by breakpoint number. Separate multiple breakpoint numbers with spaces. Use no arguments to delete all breakpoints.
disable *[breakpointid [breakpointid ...]]*	Disable breakpoints specified by breakpoint number. Separate multiple breakpoint numbers with spaces. Use no arguments to delete all breakpoints.
enable *[breakpointid [breakpointid ...]]*	Enable breakpoints specified by breakpoint number. Separate multiple breakpoint numbers with spaces. Use no arguments to enable breakpoints until commanded otherwise.
hbreak	Set a hardware assisted breakpoint.
ignore *number count*	Set the ignore-count of breakpoint *number* to *count*.
tbreak *line* | *function* | **address*	Set a temporary breakpoint at the specified *line* number, *function* name, or *address*. The breakpoint is deleted when hit.
thbreak	Set a temporary hardware assisted breakpoint.

Specifying Files to Debug

Table A-6 lists commands that specify the file(s) to be debugged.

Table A-6 **GDB Interpreter File Context Commands**

Command	Description
file *filename*	Defines *filename* as the program to be debugged.
exec-file *filename*	Specifies that the program to be run is found in *filename*.

Table A-6 **GDB Interpreter File Context Commands**

Command	Description
load *filename*	Loads a module.
unload *filename*	Unloads a module.
attach *processid*	Attaches to a process.
detach	Detaches from the debugged process.
thread *threadid*	Selects a thread as the current task to debug.
add-symbol-file *file addr*	Reads additional symbol table information from the *file* located at memory address *addr*.

Running and Stepping Through a File

Table A-7 contains commands to run and step through programs.

Table A-7 **GDB Interpreter Running and Stepping Commands**

Command	Description
run	Runs a process for debugging (use **set arguments** and **set environment** if program needs them).
kill *processid*	Kills the process with *processid*.
interrupt	Interrupts a running task or process.
continue	Continues an interrupted task or process.
step [*n*]	Steps through a program (if *n* is used, step *n* times).
stepi [*n*]	Steps through one machine instruction (if *n* is used, step through *n* instructions).
next [*n*]	Continues to the next source line in the current stack frame (if *n* is used, continue through *n* lines).
nexti [*n*]	Execute one machine instruction, but if it is a function call, proceed until the function returns (if *n* is used, execute *n* instructions).

Table A-7 **GDB Interpreter Running and Stepping Commands** (cont'd)

Command	Description
until	Continue running until a source line past the current line, in the current stack frame, is reached.
jump *address*	Moves the instruction pointer to *address*.
finish	Finishes execution of current block.

Displaying Disassembler and Memory Information

Table A-8 lists commands for disassembling code and displaying contents of memory.

Table A-8 **GDB Interpreter Disassembly and Memory Commands**

Command	Description
disassemble *address*	Disassembles code at a specified *address*.
x [*/format*] *address*	Displays memory starting at *address*. *format* is one of the formats used by **print**: **s** for null-terminated string, or **i** for machine instruction. Default is **x** for hexadecimal initially, but the default changes each time you use either **x** or **print**.

Examining Stack Traces and Frames

Table A-9 shows commands for selecting and displaying stack frames.

Table A-9 **GDB Interpreter Stack Trace and Frame Commands**

Command	Description
bt [*n*]	Displays back trace of *n* frames.
frame [*n*]	Selects frame number *n*.
up [*n*]	Move *n* frames up the stack.
down [*n*]	Moves *n* frames down the stack.

Displaying Information and Expressions

Table A-10 lists commands that display functions, registers, expressions, and other debugging information.

Table A-10 **GDB Interpreter Information and Expression Commands**

Command	Description
info args	Shows function arguments.
info breakpoints	Shows breakpoints.
info extensions	Shows file extensions (c, c++, ...)
info functions	Shows all functions.
info locals	Shows local variables.
info registers	Shows contents of registers.
info source	Shows current source file.
info sources	Shows all source files of current process.
info target	Displays information about the target.
info threads	Shows all threads.
info warranty	Shows disclaimer information.
print /x *expression*	Evaluates and prints an *expression* in hexadecimal format.

Displaying and Setting Variables

Table A-11 lists commands for displaying and setting variables.

Table A-11 **GDB Interpreter Variable Display and Set Commands**

Command	Description
set args *arguments*	Specifies the *arguments* to be used the next time a debugged program is run.
set emacs	Sets display into **emacs** mode.

Table A-11 **GDB Interpreter Variable Display and Set Commands**

Command	Description
set environment *varname* **=** *value*	Sets environment variable *varname* to *value*. *value* may be any string interpreted by the program.
set tgt-path-mapping	Sets target to host pathname mappings.
set variable *expression*	Sets variable value to *expression*.
show args	Shows arguments of debugged program.
show environment	Shows environment of debugged program.

A.2.5 Using the Built-in Line Editor

The host shell provides various line editing facilities available from the library ledLib (Line Editing Library). LedLib serves as an interface between the user input and the underlying command line interpreters, and facilitates the user's interactive shell session by providing the ability to scroll, search, and edit previously typed commands. Any input will be treated by ledLib until the user hits the **ENTER** key, at which point the command typed will be sent on to the appropriate interpreter.

The line editing library also provides path completion.

vi-Style Editing

The **ESC** key switches the shell from normal input mode to edit mode. The history navigation, completion, and editing commands in Table A-12 and Table A-14 are available in edit mode.

Some line editing commands switch the line editor to insert mode until an **ESC** is typed (as in vi) or until an **ENTER** gives the line to one of the shell interpreters. **ENTER** always gives the line as input to the current shell interpreter, from either input or edit mode.

To locate a line entered previously, press **ESC** followed by one of the search commands listed in Table A-13; you can then edit and execute the line with one of the commands from the table.

Switching Modes and Controlling the Editor

Table A-12 lists commands that give you basic control over the editor.

Table A-12 **vi-Style Basic Control Commands**

Command	Description
ESC	Switch to line editing mode from regular input mode.
ENTER	Give line to current interpreter and leave edit mode.
CTRL+D	Complete pathname (edit mode).
[tab]	Complete pathname (edit mode).
CTRL+H	Delete a character (backspace).
CTRL+U	Delete entire line (edit mode).
CTRL+L	Redraw line (edit mode).
CTRL+S	Suspend output.
CTRL+Q	Resume output.
CTRL+W	Display HTML reference entry for a routine.

Moving and Searching in the Editor

Table A-13 lists commands for moving and searching in input mode.

Table A-13 **vi-Style Movement and Search Commands**

Command	Description
*n***G**	Go to command number *n*. The default value for *n* is 1.
/*s* or **?***s*	Search for string *s* backward or forward in history.
n	Repeat last search.
*n***k** or *n***-**	Get *n*th previous shell command.
*n***j** or *n***+**	Get *n*th next shell command.
*n***h**	Go left *n* characters (also **CTRL+H**).

Table A-13 **vi-Style Movement and Search Commands** (cont'd)

Command	Description
*n***l** or **SPACE**	Go right *n* characters.
*n***w** or *n***W**	Go *n* words forward, or *n* large words. *Words* are separated by spaces or punctuation; *large words* are separated by spaces only.
*n***e** or *n***E**	Go to end of the *n*th next word, or *n*th next large word.
*n***b** or *n***B**	Go back *n* words, or *n* large words.
$	Go to end of line.
0 or **^**	Go to beginning of line, or to first nonblank character.
f*c* or **F***c*	Find character *c*, searching forward or backward.

Inserting and Changing Text

Table A-14 lists commands to insert and change text in the editor.

Table A-14 **vi-Style Insertion and Change Commands**

Command	Description
a or **A** ...**ESC**	Append, or append at end of line (**ESC** ends input).
i or **I** ...**ESC**	Insert, or insert at beginning of line (**ESC** ends input).
*n***s** ...**ESC**	Change *n* characters (**ESC** ends input).
cw ...**ESC**	Change word (**ESC** ends input).
cc or **S** ...**ESC**	Change entire line (**ESC** ends input).
c$ or **C** ...**ESC**	Change from cursor to end of line (**ESC** ends input).
c0 ...**ESC**	Change from cursor to beginning of line (**ESC** ends input).
R ...**ESC**	Type over characters (**ESC** ends input).
*n***r***c*	Replace the following *n* characters with *c*.
~	Toggle between lower and upper case.

Deleting Text

Table A-15 shows commands for deleting text.

Table A-15 **vi-Style Commands for Deleting Text**

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

Put and Undo Commands

Table A-16 shows put and undo commands.

Table A-16 **vi-Style Put and Undo Commands**

Command	Description
p or **P**	Put last deletion after cursor, or in front of cursor.
u	Undo last command.

emacs-Style Editing

The shell history mechanism is similar to the UNIX tcsh shell history facility, with a built-in line editor similar to emacs that allows previously typed commands to be edited.

To edit a command, the arrow keys can be used on most of the terminals. Up arrow and down arrow move up and down through the history list, like **CTRL+P** and **CTRL+N**. Left arrow and right arrow move the cursor left and right one character, like **CTRL+B** and **CTRL+F**.

Moving the Cursor

Table A-17 lists commands for moving the cursor in emacs mode.

Table A-17 **emacs-Style Cursor Motion Commands**

Command	Description
CTRL+B	Move cursor back (left) one character.
CTRL+F	Move cursor forward (right) one character.
ESC+b	Move cursor back one word.
ESC+f	Move cursor forward one word.
CTRL+A	Move cursor to beginning of line.
CTRL+E	Move cursor to end of line.

Deleting and Recalling Text

Table A-18 shows commands for deleting and recalling text.

Table A-18 **emacs-Style Deletion and Recall Commands**

Command	Description
DEL or **CTRL+H**	Delete character to left of cursor.
CTRL+D	Delete character under cursor.
ESC+d	Delete word.
ESC+DEL	Delete previous word.
CTRL+K	Delete from cursor to end of line.
CTRL+U	Delete entire line.
CTRL+P	Get previous command in the history.
CTRL+N	Get next command in the history.
!*n*	Recall command *n* from the history.
!*substr*	Recall first command from the history matching *substr*.

Special Commands

Table A-19 shows some special emacs-mode commands.

Table A-19 **Special emacs-Style Commands**

Command	Description
CTRL+U	Delete line and leave edit mode.
CTRL+L	Redraw line.
CTRL+D	Complete symbol name.
ENTER	Give line to interpreter and leave edit mode.

Command and Path Completion

Path completion will attempt to complete a directory name when the **TAB** key is pressed. This functionality is available from all interpreter modes.

A.2.6 Running the Host Shell in Batch Mode

The host shell can also be run in batch mode, with commands passed to the host shell using the **-c** option followed by the command(s) to execute.

The commands must be delimited with double quote characters. For example, to launch the host shell in batch mode, executing the GDB commands to load, list the source code, and run an application:

```
$ hostShell -dev dfw-session-name -m gdb -c "file helloworld; list; run" tgtsvr@host
```

A.2.7 Recording and Replaying Host Shell Commands

The **RECORD** and **RECORD_FILE** variables allow you to record a sequence of host shell commands to a file. You can see the current status of these variables with the **shConfig** command as follows:

```
-> shConfig
...
RECORD = off
RECORD_FILE = .
...
->
```

Set **RECORD_FILE** to the file name where you want to record the commands. Set **RECORD** to **on** to begin recording commands:

```
-> shConfig "RECORD=on"
```

If **RECORD_FILE** is not set, it will be set to **shellRecordFile***pid*.**cmds** in the directory **/tmp** (or **%TEMP%** for Windows).

To turn off recording, set **RECORD** to **off**. If you turn recording back on, the previous file will be used and overwritten unless you have already specified a new record file.

To replay the script, use the **-s** *record_file* command line option to **hostShell**.

You can also source it from the GDB interpreter with the **source** command.

A.2.8 Extending the GDB interpreter

Some parts of the GDB interpreter are written in TCL. This allows the user to easily extend the interpreter by adding commands that can communicate with the debugger server. The extension is limited here to synchronous commands, that is, commands that always return immediately with a possible result immediately available in the command result.

This section explains how to add such user commands by modifying the TCL shell code.

The TCL resources of the GDB host shell interpreter are located under **$(WIND_FOUNDATION_PATH)/resource/windsh/***OS***/tcl/gdbInterp/**.

The file **cmds.tcl** contains the definitions and the bodies of most of the interpreter commands.

The following example explains how to extend the GDB interpreter by adding two new commands.

Code to add at the beginning of the file (declare the commands)

```
# array of user related commands

variable userGdbArr

#
# setupUserCmds - set up the object commands array
#

proc setupUserCmds {} {
    variable userGdbArr
```

```
    set userGdbArr(0) {"a1" "gdb::usera1Cmd" "" "Help of the a1 command"
"Usage of the a1 command"}
    set userGdbArr(1) {"a2" "gdb::usera2Cmd" "xy:z:" "Help of the a2 command"
"Usage of the a2 command\noptions:\n\t-x : x option, no parameter\n\t-y <y> :
y option, 1 parameter\n\t-z <z> : z option , 1 parameter"}
}
```

Code to add in the middle of the file (body of the commands)

```
proc usera1Cmd {args} {
    puts stdout "arguments: $args"

    # get location of the debugger server log
    puts stdout [miGdb "-wrs-log"]
}

proc usera2Cmd {args} {
    puts stdout "arguments: $args"
    puts stdout "first argument: [lindex $args 0]"
    puts stdout "second argument: [lindex $args 1]"
    puts stdout "third argument: [lindex $args 2]"

    # send a string to the debugger server and get its echo
    puts stdout [miGdb "-wrs-echo $args"]
}
```

Code to add at the end of the file (register the commands)

```
gdb::setupUserCmds
shellGdbTopicAdd "user" "List of the shell commands user defined."
shellGdbArrayAdd "user" gdb::userGdbArr
```

Output, calling a1 and a2 commands

```
(gdb) a1
arguments:
^done,ls="/wind/river/DFW121/host/x86-linux2/bin/dfwserver.log",
lt="/view/philb.62/wind/river/DFW121/host/x86-linux2/bin/dfwstatus.log"

(gdb) a2 /x /y 1 /z 2
arguments: TRUE {TRUE "1"} {TRUE "2"}
first argument: TRUE
second argument: TRUE "1"
third argument: TRUE "2"
^done,echo="TRUE {TRUE 1} {TRUE 2}"
```

A.2.9 Deprecated Commands

After initializing the environment with the **eval** command and starting the usermode agent as it is now done, the sequence of commands starting at Step 3 below was formerly used to start the host shell. These commands are deprecated and may not be supported in future releases. Wind River recommends that you start the host shell as described in *Starting the Host Shell*, p.302.

1. Source the environment script.

```
$ eval `./wrenv.sh -p linux-2.x -o print_env -f sh`
```

2. Execute the **usermode-agent** binary on your host.

```
$ ./linux-2.x/usermode-agent/1.1/bin/ia/i386/usermode-agent &
[1] 3127
```

3. Run the registry (if not already running):

```
$ wtxregd.ex &
[2] 3128
$
```

4. Run a target server on the local host:

```
$ tgtsvr -n LINUX -V localhost &
[3] 3129
Thu May 12 12:26:36 2005
    Target name is localhost
tgtsvr.ex (LINUX@hamlet): Thu May 12 12:26:36 2005
    Checking License ...OK
    WTX Library version: 4.0.6.10
    Tgtsvr core version: 4.0.6.12
    Connecting to target agent... succeeded.
    Loading plug-in for UserMode-Linux... succeeded.
    Linux plugin version: 4.0.6.3
    Linux kernel signature is Linux version 2.6.9-5.EL ( #1 Wed Jan 5
19:22:18 EST 2005 ) WDB 1.0.2
    Target agent is 'ptrace' user mode agent

$
```

5. Launch the host shell:

```
$ windsh LINUX
Checking License... OK

Getting DFW plugins from directories
/home/wbuser/WindRiver/workbench-2.4/dfw/0109b/host/x86-linux2/dfwplugins
(gdb)
^done
(gdb)
^done,lt="/home/wbuser/WindRiver/workbench-2.4/dfw/0109b/host/x86-linux2/
bin/dfwstatus.log",lt="/home/wbuser/WindRiver/workbench-2.4/dfw/0109b/hos
t/x86-linux2/bin/dfwstatus.log"
```

```
(gdb)
^done,ls="/home/wbuser/WindRiver/workbench-2.4/dfw/0109b/host/x86-linux2/
bin/dfwserver.log",ls="/home/wbuser/WindRiver/workbench-2.4/dfw/0109b/hos
t/x86-linux2/bin/dfwserver.log"
(gdb)
(listening - gdbmi on port 32806)
initialization finished, pid=3142
got accept
got accept
Creating new target connection 'LINUX' /OK.
Detected target plugin 'unifiedtargetplugin'.
Establishing direct WTX connection with target server 'LINUX'.
Loading the 'gdb' shell interpreter... OK.

      /////   /////   /////   /////   /////       |
     /////   /////   /////   /////   /////        |
    /////   /////   /////   /////   /////         |
    /////   /////   /////   /////   /////         |
   //////  //////  //////  //////  //////         |
   //////  //////  //////  //////  //////         |   W I N D   R I V E R
    /////   /////   /////   /////   /////         |
    /////   /////   /////   /////   /////         |
     /////   /////   /////   /////   /////        |   Development  System
      ////    ////    ////    ////    ////        |
       ////    ////    ////    ////    ////       |
        ////    ////    ////    ////    ////      |   Host  Based   Shell
         ////    ////    ////    ////    ////     |
          ////    ////    ////    ////    ////    |
           ///     ///     ///     ///     ///    |   Version  2.0
           ///     ///     ///     ///     ///    |
            //      //      //      //      //    |
            //      //      //      //      //    |
            //      //      //      //      //    |
           //      //      //      //      //     |

      Copyright 1995-2005 Wind River Systems, Inc.

(gdb)
```

You are automatically placed at the debugger prompt, **(gdb)**.

For more information on host shell startup, see *Starting the Host Shell*, p.302.

B

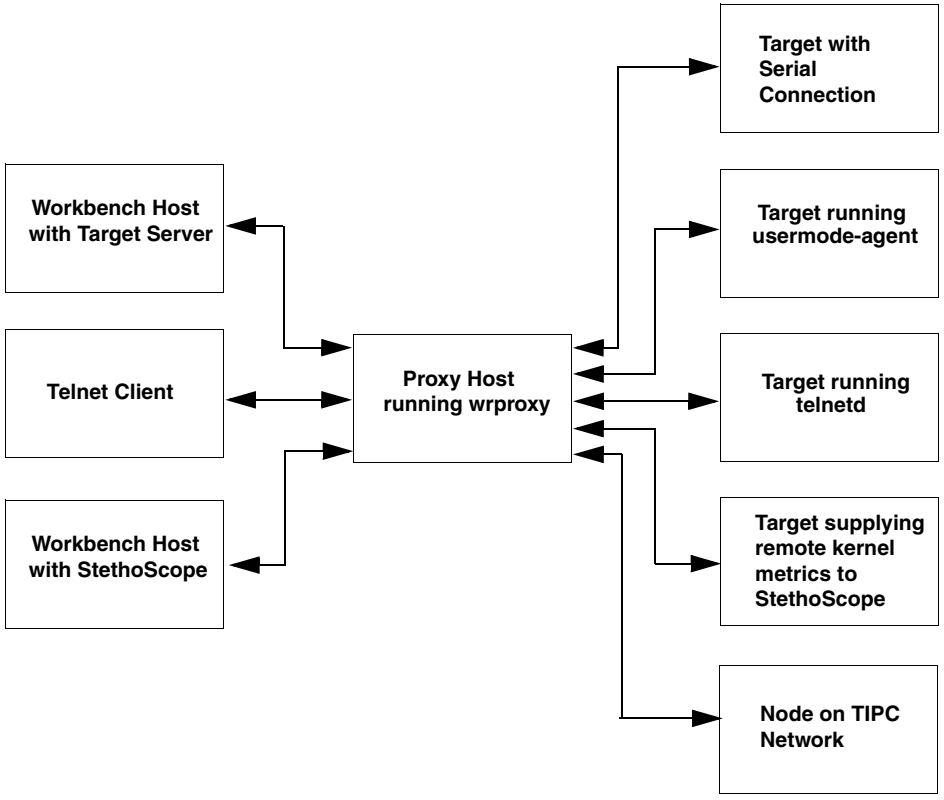
Configuring a Wind River Proxy Host

B.1 Overview

The Wind River proxy allows you to access targets not directly accessible to your Workbench host. For example, you might run the proxy server on a firewall and use it to access multiple targets behind the firewall.

The proxy supports TCP, UDP, and TIPC (Linux only) connections with targets. Many different host tools and target agents can be connected. A simple illustration of this is shown in Figure B-1.

Figure B-1 **Wind River Proxy Example**



The proxy host itself can be one that runs any operating system supported for Workbench hosts or any host running Wind River Linux. You run the **wrproxy** command supplied with Workbench on the proxy host and configure it to route access from various tools to specific targets. The mapping is done by TCP/IP port number, so that access to a particular port on the proxy host is directed to a pre-defined target. You can start **wrproxy** and then manually configure it, or you can create a configuration script that **wrproxy** reads at startup.

B.2 Configuring wrproxy

The **wrproxy** command (or **wrproxy.exe** on Windows) is located in *installDir*/**workbench-***version*/**foundation/***version*/**x86-***version*/**bin/**. Copy it to the host that will serve as your proxy host. The following discussion assumes you have copied **wrproxy** to your proxy host and are configuring it from the proxy host.

Configuring wrproxy Manually

To configure **wrproxy** manually, start it with a TCP/IP port number that you will use as the proxy control port, for example:

```
$ ./wrproxy -p 1234 &
```

You can now configure **wrproxy** by connecting to it at the specified port.

Use the **create** command to configure **wrproxy** to map client (host tool) accesses on a proxy port to a particular target. The following example configures accesses to the proxy port 1235 to connect to the Telnet port of the host **my_target**:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
create type=tcpsock;port=23;tgt=my_target;pport=1235
ok pport=1235
```

(Refer to *create*, p.331 for details on **create** command arguments.)

If you now connect to the proxy host at port 1235, you are connected to the Telnet port of **my_target**:

```
$ telnet localhost 1235
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

my_target login:
```

Creating a wrproxy Configuration Script

If you are typically using the same Wind River proxy configurations over time, it can be useful to use a startup script to configure it rather than doing it manually each time. You can cause **wrproxy** to read a startup script by invoking it as **wrproxy -s** *startupscript*. The script contains the commands that configure **wrproxy** as well as comments that begin with the **#** character. A simple startup script that configures the same port setup performed manually in the previous example might look like this:

```
# This is an example of a wrproxy startup script

# Configure the proxy host port 1235 to connect to my_target Telnet

create type=tcpsock;port=23;tgt=my_target;pport=1235

# list the port configuration

list

# end of script
```

When you start **wrproxy** with this script, it gets configured as in the previous example and sends input and output to standard output:

```
$ ./wrproxy -s wrproxy_startup &
[2] 6660
Executing startup script...

create type=tcpsock;port=23;tgt=my_target;pport=1235
ok pport=1235
list
ok pport=1235;type=tcpsock;port=23;tgt=my_target
$
```

Since no control port was specified with the **-p** option at startup, the default port 17476 is used.

NOTE: There is no password management in **wrproxy**. If you want to be sure that no new connections (tunnels) are made remotely using the control port, use the **-nocontrol** option with the **-s** *startupscript* option which will disable the proxy control port.

The startup script accepts the **create**, **list**, and **delete** commands as described in *Configuration Commands*, p.329.

B.3 wrproxy Command Summary

The following section summarizes all of the Wind River proxy commands.

NOTE: For all commands, unknown parameters are ignored; they are not considered errors. In addition, the client should not make any assumption on the number of values returned by the command as this could be changed in the future. For example, the **create** command will always return the value for **pport** but additional information may be returned in a future version of the Wind River proxy.

Invocation Commands

The **wrproxy** command accepts the following startup options:

- **-p**[**ort**]—specify TCP control port. If not specified, the default of **0x4444** (17476) is used. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.
- -**V**—enable verbose mode.
- **-v**[**ersion**]—print **wrproxy** command version number.
- **-s** *startupscript*—specify a startup script that contains **wrproxy** configuration commands.
- **-h**[*elp*]—print **wrproxy** command help.
- **-nocontrol**—disable control port.

Configuration Commands

You can use the following commands interactively, and all except the **connect** command in a Wind River proxy startup script.

connect

Create a new Wind River proxy connection and automatically connect to it. Unlike the create command (see *create*, p.331) the connection is established immediately and all packets sent to the connection are immediately routed between the target and host.

Usage

connect type=*type***;mode=***mode***;proto=***proto* ; *connection_specific_parameters*

Where the arguments to the connect command are as follows:

type is:

- **udpsock**—UDP socket connection.
- **tcpsock**—TCP socket connection.
- **tipcsock**—TIPC socket connection (Linux only).

mode describes how the connection is handled between the proxy and the client (for example the Workbench host) and is:

- **raw**—raw mode (default).
- **packet**—packet size is sent first followed by packet content; the packet is handled only when fully received.

proto describes how the connection is handled between the proxy and the target and is:

- **raw**—proxy does not handle any protocol (default).
- **wdbserial**—(VxWorks targets only) proxy converts packet to wdbserial. When **proto** is **wdbserial**, some control characters are inserted by the proxy in the packet sent to the target so that the generated packet will be understood correctly by the target using a WDB serial backend. This is typically used to connect to a WDB agent running on a target through a serial line that is connected to the serial port of a port server (this serial line is then accessible by the proxy using a well-known TCP port of the port server).

Connection-specific Parameters

- **udpsock** and **tcpsock** connection:

  port=*port***;tgt=***tgtAddr*

  Where *port* is the TCP/UDP port number and *tgtAddr* is the target IP address.

- **tipcsock** connection (Linux only):

  tipcpt=*tipcPortType***;tipcpi=***tipcPortInstance***;tgt=***tgtAddr*

  Where *tipcPortType* is the TIPC port type, *tipcPortInstance* is the TIPC port instance and *tgtAddr* is the TIPC target address.

The response of the Wind River proxy to the **connect** command is a string as follows:

```
ok
```

or

```
error errorString
```

where *errorString* describes the cause of the error.

create

Create a new proxy port mapping to a target. The connection is not established immediately as with the connect command (see *connect*, p.329) but only when a client connects to the specified port number.

Usage

create type=*type***;port=***port***;tgt=***target***;pport=***pport*

where the arguments to the **create** command are as follows:

type=*type* is:

- **udpsock**—UDP socket connection.
- **tcpsock**—TCP socket connection. (Only **tcpsock** is allowed for a VxWorks proxy host.)
- **tipcsock**—TIPC socket connection.

port—this is the port to connect to on the target.

NOTE: If you do not assign a **port** number, the default value of **0x4444** is used.

tgt=*target*—is the host name or IP address of the target when **type** is **tcpsock** or **udpsock**, and **port** provides the UDP or TCP port number. When **type** is **tipcsock** this is the target TIPC address, and **tipcpi** provides the TIPC port instance and **tipcpt** provides the TIPC port type.

pport=*proxy_TCP_port_number*—specify the TCP port number that clients (host tools) should connect to for connection to *target_host*. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.

NOTE: If you do not specify a **pport** value, one will be assigned automatically and returned in the command output.

port=*target_TCP_port_number*—specify the TCP port to connect to on the target. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.

A simple example of using the **create** command to configure a Telnet server port connection is given in *B.2 Configuring wrproxy*, p.327.

delete

Delete the proxy configuration for a specific port.

Usage

delete pport=*port_number*

To delete the proxy configuration of a specific port, use the **`delete`** command with the port number, for example:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
delete pport=1235
ok^]
telnet> q
Connection closed.
```

list

List your current configuration with the **`list`** command.

Usage

list

For example, to list your current configuration, connect to the proxy control port and enter the **list** command:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
list
ok pport=1235;type=tcpsock;port=23;tgt=my_target
```

C

Command-line Updating of Workspaces

C.1 Overview

The Workbench installation includes a **wrws_update** script that allows you to update workspaces from the command-line. This can be used, for example, to update workspaces in a nightly build script. The following section provides a reference page for the command.

C.2 wrws_update Reference

A script for updating an existing workspace is available in the Workbench installation and is named:

wrws_update.bat (Windows only)

wrws_update.sh (Windows, Linux, and Solaris)

This script launches a GUI-less Eclipse application that can be used to update makefiles, symbols (static analysis), and the retriever index.

Execution

Specify the location of the **wrws_update** script or add it to your path and execute it with optional parameters, for example:

```
$ wrws_update.sh -data workspace_dir
```

The workspace must be closed for the command to execute. If you do not specify any options to the command, all update operations are performed (**-all projects**, **-generate makefiles**, **--update symbols**, **-update index**).

Options

General Options

-h, --help
Print command help.

-q, **--quiet**
Do not produce standard output.

Eclipse Options

-data *workspace_dir*
The script uses the default workspace (if known), but it can also update other workspaces by specifying the **-data** *workspace_dir* option, just as Workbench does. (The script accepts the same command-line options as Workbench. For example, to increase virtual memory specify **-vmargs -Xmx***mem_size*.)

Global Options

-a, **--all-projects**

Update all projects, this option will force all closed projects to be opened. Opened projects will be closed after finishing the update.

-l, **--specify-list-of-projects** *argument*
Specify a list of projects to be updated. This option reduces the scope of the nightly update to the specified list of projects. Needed closed projects will be opened and unneeded opened ones closed. After finishing the update the previous state is restored. Separate the list with "," for example: **cobble,helloWorld**.

Build Options

-b, **--build-projects** *argument*
Launch build for projects. Several strings are valid as arguments, including: **build** (default), **clean**, and **rebuild**.

-e, **--enableTraceBuild**
Enable trace build output.

-f, **--debugMode** *argument*
Build using specific debug or non-debug mode where applicable. The *argument*, if specified, can be **0** or **1**, otherwise the current mode is used per project.

-u, **--buildArgs** *argument*
Specify a list of additional build options. Separate the list with "," for example: **-i,MY_VAR=***value*.

Nightly Update Options

-i, **--update-index**
Update search-database index.

-m, **--generate-makefiles**
Regenerate Makefiles where necessary.

-s, **--update-symbols** *argument*
Update symbol database (static analysis). To create the data from scratch, you can supply 'rebuild' as argument.

-t, **--create-team-symbols** *argument*
Export symbol databases for shared use in a team. The argument is a quoted comma-separated list of options. Valid options are **timestamp**, **readonly**, and **checksum**. The default is **timestamp,readonly,checksum**. See the online documentation for details on these options.

-x, **--update-xref** *argument*
Update cross references (static analysis). To create the data from scratch, you can supply 'rebuild' as argument.

Output

Any errors that might occur during the updates are printed out to standard error output. Other information (for example, status, what has been done, and so on) are printed out to standard output.

NOTE: No configuration management-specific actions or commands are executed within this script and the launched application. Configuration management specific synchronizations or updates relevant to the workspace (for example, **cvs-update**, ClearCase view synchronization, and so on) have to be done before this script is started.

D

Command-line Importing of Projects

D.1 Overview

The Workbench installation includes a **wrws_import** script that allows you to import existing projects into workspaces from the command-line. The following section provides a reference page for the command.

D.2 wrws_import Reference

A script for launching a GUI-less Eclipse application that can be used to import existing projects into the workspace is available in the Workbench installation and is named:

wrws_import.bat (Windows only)

wrws_import.sh (Windows, Linux, and Solaris)

Execution

Specify the location of the **wrws_import** script or add it to your path and execute it with optional parameters, for example:

```
$ wrws_import.sh -data workspace_dir
```

Options

General Options

-d, --debug *argument*
Provide more information. The argument, if given, specifies the level of verbosity. Default is **2**, the possible options are: [**2, 3, 4**].

-h, --help
Print command help.

-q, --quiet
Do not produce standard output.

Eclipse Options

-data *workspace_dir*
Specify the Eclipse workspace with this option.

Import Project Options

-f, --files *argument*

Specify a list of project files to be imported. Separate the items in the list with commas (**,**). For example: *dir1***/.project**,*dir2***/.project**.

-r, --recurse-directory *argument*
Specify a directory to recursively search for projects to be imported.

NOTE: This script will not stop or fail if some projects already exist in the Workspace, the way the **Import existing projects into workspace** wizard does. It will just print out the information and continue.

E

Wind River Cross Compiler Prefixes

Cross Compiler Prefixes for Supported Architectures

Table E-1 provides the correct cross compiler prefixes to use when building for the different Wind River Linux-supported architectures.

Table E-1 **Wind River Linux Cross-Compiler and Architecture Mappings**

ARCH	*Cross_Compile*
arm	**arm-wrs-linux-gnueabi-arm-glibc_full-** **arm-wrs-linux-gnueabi-arm-glibc_small-** **arm-wrs-linux-gnueabi-arm-uclibc_small-** **arm-wrs-linux-gnueabi-armv5teb-glibc_cgl-** **arm-wrs-linux-gnueabi-armv5teb-glibc_small-** **arm-wrs-linux-gnueabi-armv5tel-glibc_full-** **arm-wrs-linux-gnueabi-armv5tel-glibc_small-** **arm-wrs-linux-gnueabi-armv5tel-uclibc_small-**
i386	**i586-wrs-linux-gnu-i686p4-glibc_cgl-** **i586-wrs-linux-gnu-i686p4-glibc_full-** **i586-wrs-linux-gnu-i686p4-glibc_small-**

Table E-1 **Wind River Linux Cross-Compiler and Architecture Mappings**

ARCH	*Cross_Compile*
mips	**mips-wrs-linux-gnu-mips32_eb-glibc_cgl-** **mips-wrs-linux-gnu-mips32_eb-glibc_full-** **mips-wrs-linux-gnu-mips32_eb-glibc_small-** **mips-wrs-linux-gnu-mips32_eb-uclibc_small-** **mips-wrs-linux-gnu-mips32_el-glibc_full-** **mips-wrs-linux-gnu-mips32_el-glibc_small-** **mips-wrs-linux-gnu-mips32_el-uclibc_small-** **mips-wrs-linux-gnu-mips32f_eb-glibc_cgl-** **mips-wrs-linux-gnu-mips32f_eb-glibc_full-** **mips-wrs-linux-gnu-mips32f_el-glibc_cgl-** **mips-wrs-linux-gnu-mips32f_el-glibc_full-**
ppc	**powerpc-wrs-linux-gnu-603e-glibc_full-** **powerpc-wrs-linux-gnu-603e-glibc_small-** **powerpc-wrs-linux-gnu-e500-glibc_cgl-** **powerpc-wrs-linux-gnu-e500-glibc_full-** **powerpc-wrs-linux-gnu-e500-glibc_small-** **powerpc-wrs-linux-gnu-ppc440-glibc_full-** **powerpc-wrs-linux-gnu-ppc440-glibc_small-** **powerpc-wrs-linux-gnu-ppc7400-glibc_cgl-** **powerpc-wrs-linux-gnu-ppc7400-glibc_small-** **powerpc-wrs-linux-gnu-ppc750-glibc_cgl-** **powerpc-wrs-linux-gnu-ppc750-glibc_small-** **powerpc-wrs-linux-gnu-ppc970-glibc_cgl-** **powerpc-wrs-linux-gnu-ppc970-glibc_full-** **powerpc-wrs-linux-gnu-ppc970-glibc_small-** **powerpc-wrs-linux-gnu-ppc970_64-glibc_cgl-** **powerpc-wrs-linux-gnu-ppc970_64-glibc_small-**

F

Configuring Linux 2.4 Targets (Dual Mode)

F.1 Introduction

Dual mode is for use in kernel and application debugging on Linux 2.4 kernels. It does not work on Linux 2.6 kernels, which includes Wind River Linux kernels. For details on debugging generic Linux 2.6 kernels, refer to *5. Kernel Debugging (Kernel Mode)*. For details on debugging generic Linux 2.6 applications, refer to *3. Developing Applications (User Mode)*. Refer to *4. Configuring Wind River Linux Platforms* for specifics on developing Wind River Linux platforms.

NOTE: For support with using Workbench to develop embedded projects that are not based on Wind River Linux, contact Wind River Support Services.

Wind River Workbench can be used with Linux 2.4.x kernels with a dual mode of user and system connection that requires the kernel to be patched with the Wind River WDB agent. This chapter describes how to acquire the necessary tools and then patch, configure, and build your Linux 2.4.*x* target kernel so that it can communicate with Workbench. It also describes how to download to a target, and provides a tutorial on the use of dual mode with a PPC target and associated tools. Refer to the Wind River online support site for additional examples of how to use the user and system modes available in dual mode.

To communicate with Workbench, your Linux 2.4.*x* target kernel must be patched with the Wind River WDB agent. The WDB agent includes a small implementation of UDP/IP, which Workbench uses to communicate with your target.

Information about downloading a kernel is provided in *F.4 Obtaining a Kernel*, p.348. See the **target.ref.linux** files on the Wind River online support site for the specific kernels supported by your board.

Although you may have a custom version of a kernel that you wish to use with Workbench, Wind River highly recommends that the first time you try this procedure, you use one of the standard versions of the kernel. Doing so will familiarize you with the process and expose any issues that may arise when you try to work through these steps with a custom kernel.

Before Workbench can be used with a Linux target, the host must be correctly configured, a bootloader must be configured and built, the target itself must have its jumpers and other hardware set correctly, and finally the bootloader must be loaded into the target's flash memory.

The next several sections point to online documentation which covers these steps—including a variety of target hardware—in detail.

F.2 Setting Up the Linux Host

To build your embedded Linux system, verify that your host setup includes the following:

- A running TFTP server with a TFTP directory that has full read and write access.
- Any firewall software disabled.
- A running NFS server.
- Workbench installed.

You may also require root access to your system, so make sure you know the root password before continuing.

F.3 Tools

This section provides a summary of tools that are used to enable a dual mode environment, and then lists the specific tools that are used in the examples in this chapter.

Summary

You must choose the correct kernel, cross-compiler and bootloader for the target you are using. The following list outlines in general terms the tools you should have available to get your embedded Linux system working correctly, and provides some pointers to further information. You will need the following:

- **A target reference board, with necessary cables and power supply**

  Make sure that your target is based on a processor architecture supported by Workbench. For complete information on supported targets, please see the **target.ref.linux** files on the Wind River Online Support site.

- **Wind River Workbench**

  In addition to the Workbench functionality, the Workbench installation includes patches that are required for building your embedded system.

- **Cross-compilers**

  A cross-compiler and related tools are required if your target is of an architecture different from your host. Cross-compilers, like targets, are a matter of personal preference. There are many available. You will need this tool to compile your Linux kernel. You may also need to use it to compile a bootloader into an image to download to your target.

- **Bootloaders**

  The bootloader is used to configure your target to run the Linux kernel and to pass parameters to the kernel. There are several bootloaders and methods of booting a target from which to choose. For information on appropriate bootloaders for particular boards, please see the **target.ref.linux** files on the Wind River online support site.

- **Linux kernel**

  Workbench dual mode supports the 2.4.*x* versions of the Linux kernel. The supported architectures have been tested with various versions of the 2.4.*x* kernel. Make sure that the target kernel you choose is supported by your target board. Information on obtaining a kernel is provided in *F.4 Obtaining a Kernel*, p.348.

- **A debugger and emulator, or flash programmer**

  These are the tools you need to program the bootloader into the flash memory on your target.

The following sections describe the specific tools used in the examples.

Target

The examples in this chapter use the Wind River SBC 8260 target.

Cross-compiler

The examples use version 3.0 of the ELDK cross-compiler. This cross-compiler is used to build the bootloader, Linux kernel, and applications. In addition, the ELDK installation provides a root file system that you can use to develop your application.

NOTE: Although ELDK 3.0 is described in this manual, ELDK 2.1 is also supported for the:

- Wind River SBC 8260
- Wind River Power QUICC II
- IBM Walnut 405GP

Instructions for ELDK 3.0 also apply to ELDK 2.1.

ELDK is available at:

http://www.denx.de/e/index1.php

Make sure you install the correct version of ELDK.

For information on installing the ELDK and using it correctly on your Linux host, see:

http://www.denx.de/wiki/DULG/ELDK

After you install ELDK, Wind River recommends including the following in your path:

/ELDK_installDir`/usr/bin`

To verify your path, enter the following:

```
$ echo $PATH
```

For example, if ELDK is installed in the **/opt/eldk** directory, **/opt/eldk/usr/bin** should be displayed.

NOTE: Including ELDK in your path is required in order to build a Linux kernel with your tools. Compiling application projects with Workbench does not require ELDK in your path, but including it makes compilation easier.

Bootloaders

The examples in this chapter are for the U-Boot bootloader; however, you may find other bootloaders more suitable.

U-Boot is commonly used with PowerPC architectures, and is available for download at:

http://sourceforge.net/projects/u-boot/

This chapter describes U-Boot versions 1.1.0 and 1.1.1. Make sure you download the appropriate version for your target.

Kernels

The primary source for Linux kernels is:

http://www.kernel.org/pub/linux/kernel

For more information on obtaining your kernel source, see *F.4 Obtaining a Kernel*, p.348.

Debugger and Emulator or Flash Programmer

Some bootloaders, like U-Boot, need to be programmed into flash. If you are using Wind River ICE or Wind River Probe, see the *Wind River Workbench On-Chip Debugging Guide*.

F.4 Obtaining a Kernel

The central repository for Linux kernels is located at the following location:

http://www.kernel.org/pub/linux/kernel.

Go to that Web site and download the kernel that you want to use to build your embedded system. Be careful to download a kernel version that is supported for your target as described in the **target.ref.linux** file.

NOTE: If you have a custom version of a kernel that you want to use to build your system, you do not need to download a kernel from **http://www.kernel.org**.

The kernel source code you download is supplied in a single compressed file. For example, if you download the 2.4.20 version of the kernel, the file is called **linux-2.4.20.tar.gz** or **linux-2.4.20.tar.bz2**.

Extract the kernel source code by entering the following:

```
$ tar xvzf linux-2.4.20.tar.gz
```

or

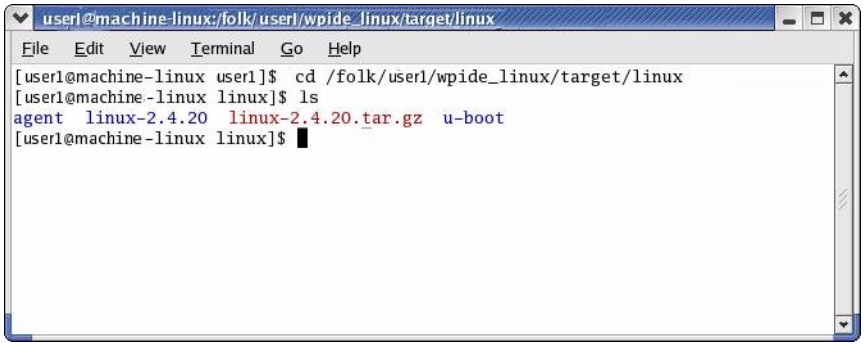
```
$ tar xvjf linux-2.4.20.tar.bz2
```

depending on what type of file you downloaded.

NOTE: If you download a different kernel, modify the command to reflect the name of that kernel.

The Linux kernel source code extracts. When it is finished, the root kernel source directory resides in the same location as the original compressed file, as shown in Figure F-1.

Figure F-1 **New Linux Kernel Directory**



F.5 Applying the WDB Patch

For Workbench to communicate properly with your kernel, the WDB agent must be running on the target. The WDB agent is supplied as two kernel patches in the *installDir***/linux-2.x/linux-2.4/agent/2.4** directory of the Workbench installation.

To install the WDB agent, apply two patch files to your kernel.

- The first is the kernel hooks patch, and it is specific to the kernel you are using. Kernel hooks patches are named **linux-2.4**.*X*-**wdb2.4.patch**, where *X* is the version of the Linux kernel you are using.

  For example, if you are using a Linux 2.4.26 kernel, the kernel hooks patch you need to use is called **linux-2.4.26-wdb2.4.patch.**

- The second patch file is not dependent on kernel architecture, and it supplies the WDB Agent functionality. That file is called **wdbagent-2.4.tar**.

The WDB agent patches are designed to integrate smoothly with the default versions of the kernel. If you have a custom Linux kernel that you are using to build your embedded system, you may need to port the WDB agent to fit your specific version. See the *WDB Agent Porting Guide* for information about how to port the WDB agent to your kernel.

To apply the patches to a standard version of the kernel, do the following:

1. Change directories until you are in the top level of your kernel source.

   ```
   $ cd /pathToKernelInstallDir/
   ```

2. Patch the kernel with the kernel hooks patch.

   ```
   $ patch -p1 <
   installDir/linux-2.x/linux-2.4/agent/2.4/linux-2.4.X-wdb2.4.patch
   ```

   For example, to patch a Linux 2.4.26 kernel, type:

   ```
   $ patch -p1 <
   installDir/linux-2.x/linux-2.4/agent/2.4/linux-2.4.26-wdb2.4.patch
   ```

   Text will be output to the screen as the kernel hooks patch is applied.

   If you are using a standard version of the kernel, no errors occur. If you are not using a standard kernel, see the *WDB Agent Porting Guide* for information about how to port the patch to your kernel.

NOTE: If you are using the standard version of the kernel and you encounter errors when you apply the patch, be sure that you are using the correct patch for your kernel version.

3. Verify that you are still at the root of your kernel source.
4. Patch the kernel with the WDB agent.

   ```
   $ tar xvf installDir/linux-2.x/linux-2.4/agent/2.4/wdbagent-2.4.tar
   ```

The WDB agent code extracts into a directory called **wdbagent** in your kernel tree.

5. If you are using a version of the Linux kernel other than 2.4.26, change the file permissions of the **mkimage.wrapper** file to executable.

NOTE: If you are using the Linux 2.4.26 kernel, skip this step.

```
$ chmod +x arch/ppc/boot/utils/mkimage.wrapper
```

NOTE: Changing the permissions on this file is required because the source code from **http://www.kernel.org** does not include execute permissions on the **mkimage.wrapper** script.

F.6 Configuring the Kernel

Prior to working through this section, do the following:

- Make sure that the cross-compiler, such as ELDK, is installed and working correctly as described in *Cross-compiler*, p.346.
- Make sure that Workbench is installed on your host.

You can use Workbench to build your Linux kernel, or you can build it from the command line. This section includes instructions for building it both ways.

To configure your kernel for use with a root file system you will load separately, no additional files are required. This section assumes that you are using ELDK and a standard version of the Linux kernel.

NOTE: The cross-compile options included in this section are specific to ELDK. You may need to modify the commands if you are using a different cross-compiler.

This section also assumes you are using U-Boot to load the Linux kernel onto the target, and describes the steps necessary to make the bootable image required by U-Boot. For more information about U-Boot, see *Bootloaders*, p.347.

Prior to beginning, make sure that your kernel source code is available on a local or networked drive that you can access through Workbench. Also make sure that you correctly applied the patch described in *F.5 Applying the WDB Patch*, p.349.

You can choose to build your kernel using Workbench or from the command line. Workbench instructions are provided in *F.6.1 Building the Kernel in Workbench as a Linux Kernel Project*, p.352, and command-line instructions are provided in *F.6.2 Building the Kernel from the Command Line*, p.358. See the section that best suits your development needs.

F.6.1 Building the Kernel in Workbench as a Linux Kernel Project

The following steps describe how to use Workbench to build your Linux kernel.

NOTE: Before starting Workbench, make sure the path environment variable is set to include the path to your compiler.

1. Start Workbench running on your host.

   To start Workbench, change directories to *installDir* and enter the following:

   ```
   $ ./startWorkbench.sh
   ```

   When you start Workbench, you are asked to specify a workspace for storing your project information. You must have write access to the workspace you use.

   After you specify a workspace and click **OK**, and Workbench displays the Welcome screen.

NOTE: If you have opened Workbench previously, and are using the same workspace, the **Welcome** screen does not display. Instead, Workbench opens directly to the Application Development perspective.

2. Click **Start**, then **Workbench** to open the Application Development perspective.
3. Right-click in the **Project Navigator**, select **New > Project**.

   The **New Project** dialog box appears.
4. Expand the **Project** node, click **Embedded Linux Kernel Project** and click **Next**.
5. Enter a name for your kernel project in the **Project Name** field.

   The **Location** area of the dialog box lets you specify where you want to store your project files. To use the location where you placed the kernel files, select Create project at external location, and browse to the top level directory of your kernel source. Click **OK** and then click **Next**.

6. The **Build Support** dialog box appears. This is where you specify how Workbench should build your kernel.

   Leave the **Build support** area of the dialog box set to **User-defined build**.

   Specify the path to your kernel source tree and the required cross-compiler information in the **Build command** field.

   To add the path to your kernel, append the line **-C** */pathToKernelInstallDir* to the existing line in the dialog box.

   Add the following cross-compile and PowerPC architecture commands to the same line:

   `ARCH=ppc CROSS_COMPILE=`*CrossCompilePrefix*

NOTE: Recall that your *CrossCompilePrefix* is unique to your architecture. Refer to the bootloader page of the online support site for information about the *CrossCompilePrefix* for your architecture.

NOTE: If you want to debug the kernel, include the line **CFLAGS_KERNEL+=-gdwarf-2** in the dialog box. This will build your kernel with symbol data.

NOTE: The cross-compile commands used in this section are specific to the ELDK tools. You may need to modify this line if you are using a different cross-compiler.

   Click **Next**.

7. The **Static Analysis** dialog box appears. By default, Workbench builds your kernel with static analysis data. For very large projects, such as a kernel project, it can be faster to turn static analysis off. If you want to turn it off, uncheck the box **Enable Static Analysis**. Click **Finish**.

   The project is created, and when finished, the name of the project appears in the **Project Navigator** view of Workbench.

NOTE: The path to your ELDK installation must be included in your default Linux path to build a kernel, as described in *F.3 Tools*, p.345.

8. Click the arrow beside the project name to expand the list of files in your project.

Adding a Build Target to the Project

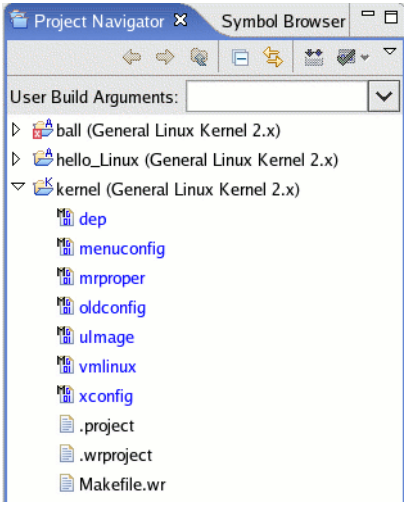
To build an image of your kernel that the bootloader can download to your target, add a build target to the project.

1. Right-click your kernel project name, select **New > Build Target**.
2. If you are using a Linux 2.4.26 kernel, enter **uImage** in the **Build target name** field of the dialog box.

   If you are using any other version of the Linux kernel, enter **pImage** in the **Build target name** field of the dialog box.
3. Click **Finish**.

   A new build target displays in the **Project Navigator**, as shown in Figure F-2.

Figure F-2 **New Build Target**



NOTE: The **uImage** and **pImage** build targets rely on the build tools that you are using to create a PowerPC bootable image being available. For PowerPC, the utility is called **mkimage**, and it is included with the ELDK tools.

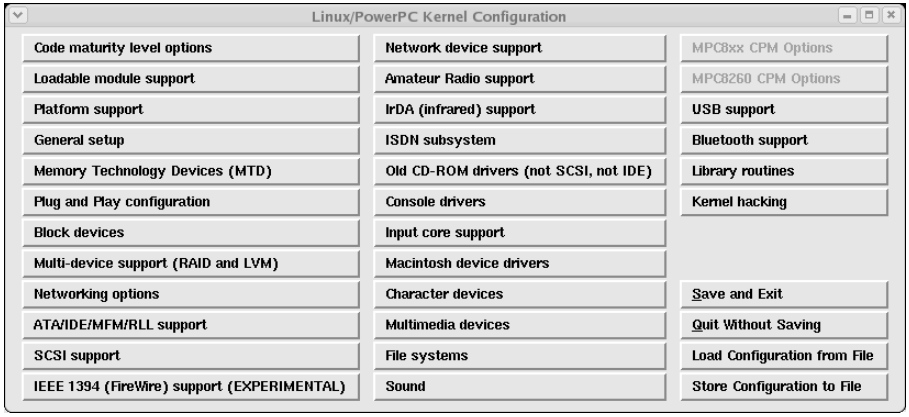
Building a Bootable Kernel Image

Now everything you need to build the kernel is ready. The following steps describe how to proceed.

1. Right-click the **xconfig** build target in the **Project Navigator**, select **Build Target**.

   Build output displays in the **Build Console**, and the **Linux Kernel Configuration** menu displays as shown in Figure F-3.

Figure F-3 **Linux Kernel Configuration Menu**

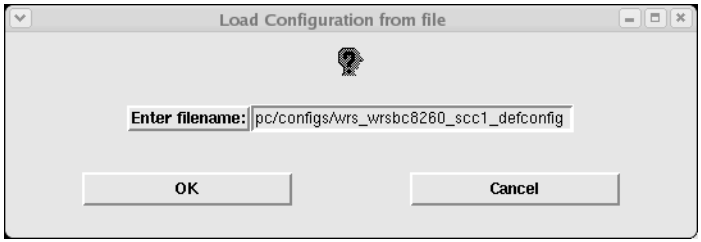


NOTE: The **Linux Kernel Configuration** menu is not part of Workbench. It is a graphical utility provided with your Linux kernel that you can use for configuration purposes.

2. Click **Load Configuration from File**.

   The **Load Configuration from file** dialog box appears as shown in Figure F-4.

Figure F-4 **Load Configuration**



The configuration file names are unique to your target and the Ethernet device you are using. The configuration file names for your architecture are in the appropriate **target.ref.linux** file.

The configuration files are located in **arch/ppc/configs**. Enter **arch/ppc/configs/***Configuration_File_Name* for your target into the **Load Configuration from file** dialog box.

Table F-1 describes configuration file names for the Linux 2.4.26 kernel. These configuration files include the WDB agent as a buildable option; select the specified Ethernet port.

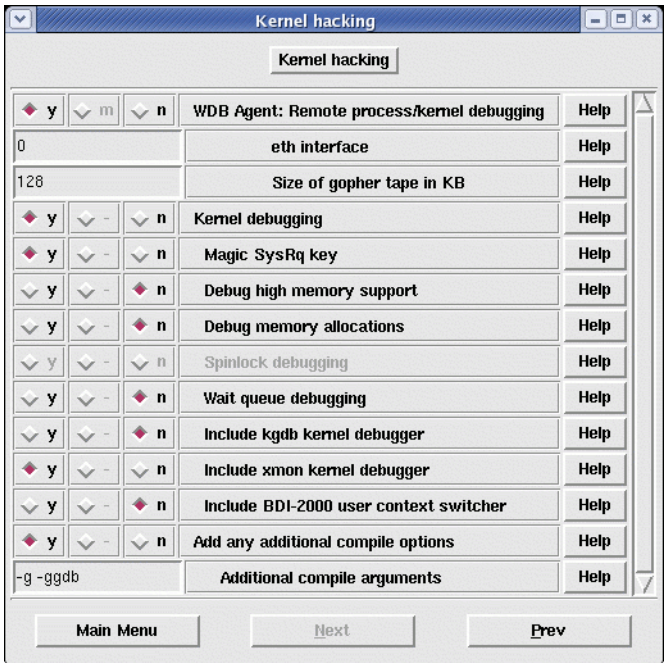
Table F-1 **Kernel Configuration File Names**

Target	Ethernet Device	Configuration File Name
Wind River SBC 8260	SCC1	wrs_wrsbc8260_scc1_defconfig
Wind River SBC 8260	FCC2	wrs_wrsbc8260_fcc2_defconfig
Wind River PowerQUICC II	SCC1	wrs_wrsbc8260_scc1_defconfig
Wind River PowerQUICC II	FCC2	wrs_wrsbc8260_fcc2_defconfig
Sandpoint 8245	EE Pro 100 PCI Card or Realtek 8139 Card	wrs_SandpointX3-8245_defconfig
IBM Walnut 405GP	CH1	wrs_Walnut405GP_defconfig
IBM Ebony 440GP	CH1	wrs_Ebony440GP_defconfig
Wind River SBC 8560	gianfar or FCC2	wrs_wrsbc8560_gianfar_defconfig wrs_wrsbc8560_fcc2_defconfig

Click **OK** to close the dialog box.

3. From the **Kernel Configuration** menu, click **Kernel hacking**. Verify that the **WDB Agent: Remote process/kernel debugging** option is set to **Yes**, as shown in Figure F-5.

Figure F-5 **Kernel Hacking**



4. Click the **Main Menu** to close the dialog box.

NOTE: If you are using a target that is not supported, you may need to modify the kernel parameters in this menu manually. Make sure that in the **Kernel Hacking** dialog box, the **WDB Agent: Remote process/kernel debugging** option is set to **Y**. Pay attention to network settings, boot arguments, and board-specific settings when you configure your kernel.

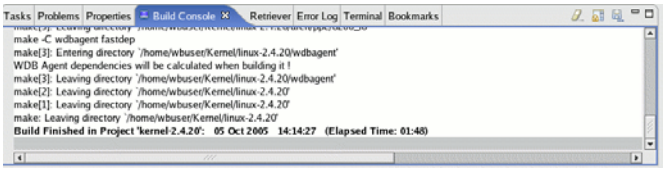
5. Click **Save and Exit** to exit the **Linux Kernel Configuration** menu.

NOTE: Make sure to click **Save and Exit** instead of just closing the dialog box. Otherwise, the changes you made are not saved.

6. Click **OK** to close the confirmation dialog box that displays and return to Workbench.
7. Right-click the **dep** build target in the **Project Navigator**, select **Build Target**.

   This generates all the kernel dependencies. Build output displays, as shown in Figure F-6.

Figure F-6 **Kernel Dependencies Build Output**



8. Right-click either the **pImage** or the **uImage** build target in the **Project Navigator**, select **Build Target**.

   This builds the final bootable image and symbol file.

   When it completes, a **.elf** file called **vmlinux** displays at the root of your kernel tree, and a bootable image file is included in the *kernelInstallDir*/**arch/ppc/boot/images** directory. If you built a **pImage** file, the image file is called **vmlinux.PPCBoot**. If you built a **uImage** file, the image file is called **vmlinux.UBoot**.

Now you have a bootable image of your kernel. See *F.11 Downloading the Kernel to the Target*, p.369 for information about how to use U-Boot to download the kernel to your target.

F.6.2 Building the Kernel from the Command Line

Prior to building your kernel, make sure that you have correctly applied the WDB Agent patch as described in *F.5 Applying the WDB Patch*, p.349.

The following steps describe how to build your kernel. They use the **xconfig** kernel configuration menu as described in *Building a Bootable Kernel Image*, p.355, so refer to that section for further details.

1. **cd** into the top level of your kernel directory.
2. Enter the following command:

   ```
   $ make ARCH=ppc CROSS_COMPILE=CrossCompilePrefix xconfig
   ```

NOTE: Recall that your *CrossCompilePrefix* is unique to your architecture. Refer to the bootloader page of the online support site for information about the *CrossCompilePrefix* for your architecture.

The **Linux Kernel Configuration** menu displays.

3. Click **Load Configuration from File**.

   Enter **arch/ppc/configs/***Configuration_File_Name* for your target into the **Load Configuration from file** dialog box.

4. Click **OK** to close the dialog box.

5. Click **Kernel hacking**, verify that the **WDB Agent: Remote process /kernel debugging** option is set to **Yes.**

6. Click **Main Menu** to close the dialog box.

NOTE: If you are not using one of the supported targets, you may need to modify the kernel parameters in this menu manually. Make sure that in the **Kernel Hacking** dialog box, the **WDB Agent: Remote process/kernel debugging** option is set to **Y**. Pay attention to network settings, boot arguments, and board-specific settings when you configure your kernel.

7. Click **Save and Exit** to exit the **Linux Kernel Configuration** menu.

NOTE: Be sure to click **Save and Exit** instead of just closing the dialog box. Otherwise, the changes you made are not saved.

8. Click **OK** to close the confirmation dialog box that displays and return to the command line.

9. Build the kernel dependency information by typing:

   ```
   $ make ARCH=ppc CROSS_COMPILE=CrossCompilePrefix dep
   ```

   The kernel dependencies build.

10. Build the compressed kernel by typing:

    ```
    $ make ARCH=ppc CROSS_COMPILE=CrossCompilePrefix ImageType
    ```

    If your are using a Linux 2.4.26 kernel, *ImageType* is **uImage**. If you are using any of the other supported kernels, *ImageType* is **pImage**.

    For example, to build the compressed kernel for Sandpoint 8245 target with a Linux 2.4.26 kernel, type:

    ```
    $ make ARCH=ppc CROSS_COMPILE=ppc_82xx- uImage
    ```

This builds the final bootable image and symbol file.

When it completes, a .**elf** file called **vmlinux** displays at the root of your kernel tree, and a bootable image file is included in the *kernelInstallDir*/**arch/ppc/boot/images** directory. If you built a **pImage** file, the image file is called **vmlinux.PPCBoot**. If you built a **uImage** file, the image file is called **vmlinux.UBoot**.

You now have a bootable image of your kernel.

F.7 Preparing to Load the Linux Kernel

Loading the Linux Kernel onto the target is a three-step procedure:

1. Exporting the target file system.
2. Booting the target.
3. Downloading and launching the kernel.

The second step, booting the target, may be performed using a variety of methods and bootloaders, depending on the type of target and on the method and bootloader preferred.

Examples in this chapter describe booting a Power PC target board, using the U-Boot bootloader. They also describes how to launch U-Boot on your target, configure it correctly for your system and kernel, and use it to load your Linux kernel.

Before You Begin

Prior to working through this section, do the following:

- Turn the power off on your target.
- Remove the emulator or flash programmer from your system.
- Connect the serial cable on your target to your Linux host.
- Make sure that the power and the Ethernet connections are secure on your target.

NOTE: Leave the power to your board turned off at this time.

- Work through *F.6 Configuring the Kernel*, p.351 to make sure you have a kernel file to load to your target.
- Decide how you want to mount your kernel and root file system.

During the development phase of your project, it is easiest to mount your root file system on an NFS server. To do that, make sure you built the build target using **pImage** or **uImage**, and that you have a separate root file system available to mount. If you are using the ELDK tools, you can export the standard root file system provided in the installation. If you did not export the root file system when you installed ELDK, see *F.8 Exporting the ELDK Root File System*, p.361.

F.8 Exporting the ELDK Root File System

The ELDK installation includes a root file system. If you already have a root file system you want to use with your kernel, skip this step and go to *F.9 Launching U-Boot*, p.362.

To export the ELDK root file system:

1. Log in to your Linux host as root.
2. Change directories to the **/etc** directory.

   ```
   root$ cd /etc
   ```

3. Edit the **Exports** file to add the following line:

   /ELDK_installDir/ppc_*target_arch* *(rw,sync,no_root_squash)

   For example, if you are using a PowerPC 82xx target, type:

   */ELDK_installDir***/ppc_82xx *(rw,sync,no_root_squash)**

NOTE: This file is blank until you add the line specified in Step 3.

4. Save the file and close it.
5. Export the file system.

   ```
   root$ /usr/sbin/exportfs -a
   ```

6. Restart your portmap and NFS services.

```
root$ /sbin/service portmap restart
root$ /sbin/service nfs restart
```

7. Verify that your file system is exported correctly. Type:

```
$ /usr/sbin/exportfs
/opt/eldk3.0/ppc_82xx
              <World>
$
```

The root file system within the */ELDK_installDir*/**ppc**_*target_arch* directory is now exported and ready for mounting.

F.9 Launching U-Boot

You must configure a serial connection to the target, connect, and launch U-boot as described in this section.

Configuring a Serial Terminal

Configure a serial terminal that you can use to communicate with your target. Workbench includes a **Terminal** view you can use to open a serial channel. When you start Workbench, the **Terminal** view is included in the group of tabbed views at the bottom of In the **Application Development Perspective**.

The following steps describe how to set up the **Terminal** view to communicate with your target.

1. Bring the **Terminal** view to the front of the group by clicking on its tab.
2. Click the **Settings** button to open the **Terminal Settings** dialog box.
3. Configure the **Terminal Settings** dialog box as follows:
   a. Set the **Connection Type** to **Serial**
   b. Set the port to the **tty** port your are using (either **ttyS0** or **ttyS1**)
   c. Configure the **Baud Rate** to match the speed that you specified in the U-Boot configuration—for example, if you set the port speed in the U-Boot file to use 115200, make sure that the serial connection is also set to 115200.

d. Set the **Data Bits** to **8**, **Stop Bits** to **1**, **Parity** to **None**, and both **Flow In** and **Flow Out** to **None.**

4. Click **OK** to close the **Terminal Settings** dialog box.
5. Click the **Connect** button to open the serial channel.

NOTE: If you are not running your Linux host as a root user, make sure that the permissions are set correctly for you to access the serial port. To set the serial port permissions correctly, issue one of the following commands (depending on which port you plan to use:

```
$ chmod 777 /dev/ttyS0
$ chmod 777 /dev/ttyS1
```

If you do not have the permissions set correctly, only the **NET** option is available in the **Connection Type** area of the **Terminal Settings** dialog box.

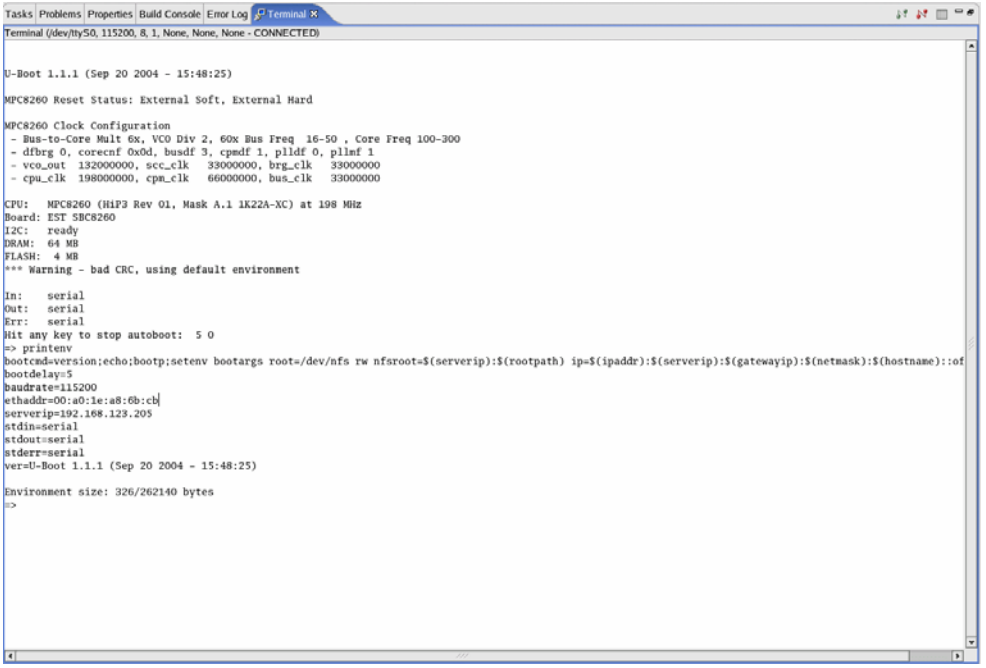
Launching U-Boot

Apply power to your target. If you have defined your serial connection correctly, the U-Boot initialization sequence begins. After a few seconds, the following message displays:

```
Hit any key to stop autoboot: 5
```

Press a key before the autoboot sequence reaches 0. A prompt appears, as shown in Figure F-7.

Figure F-7 **U-Boot Boot Sequence**



NOTE: Figure F-7 shows the startup sequence for U-Boot 1.1.1 on a Wind River SBC 8260 target. Values may be different with different targets and versions of U-Boot.

F.10 Configuring U-Boot

Configure U-Boot on your target to download the Linux kernel you built in *F.6.1 Building the Kernel in Workbench as a Linux Kernel Project*, p.352. Before you begin, make sure that you have the following:

- Your target connected with U-Boot in the flash.
- Your Linux image on your host with a mountable root file system.

- A TFTP server running on your Linux host, which is used to load the kernel from your host to the target.
- An IP address on your network that you can assign to your target.
- An Ethernet cable that connects your network to the correct Ethernet port on your target (either the FCC port or the SCC port, depending on your board).
- A serial cable connecting the serial port on your target to the your Linux host, as described in *F.9 Launching U-Boot*, p.362.

F.10.1 Setting up the Kernel Files

In *F.6.1 Building the Kernel in Workbench as a Linux Kernel Project*, p.352, you created two files. A symbol file called **vmlinux**, which is located at the root of the kernel tree, and a bootable image file called either **vmlinux.PPCBoot** or **vmlinux.UBoot**, located in the *kernelInstallDir*/**arch/ppc/boot**
/images directory.

Move these files so that U-Boot can find them correctly. Move the **vmlinux.PPCBoot** or **vmlinux.UBoot** file to your TFTP directory. Move the **vmlinux** file into the top level of the root file system you intend to export via NFS.

NOTE: To load your kernel, you must have a TFTP server actively running on your Linux host. Make sure that it is running prior to continuing.

For example, if you exported the standard ELDK root file system as described in *F.8 Exporting the ELDK Root File System*, p.361, copy the **vmlinux** file to */ELDK_installDir/target_architecture*.

As a specific example, if you installed ELDK in the **/opt/eldk** directory for an IBM Walnut 405GP target, copy the **vmlinux** file to **/opt/eldk/ppc_4xx**.

F.10.2 Configuring U-Boot

The following commands are used to make changes to U-Boot in the flash memory on your target:

printenv—displays the current U-Boot settings

setenv—modifies U-Boot settings

saveenv—saves any changes you have made to U-Boot settings

NOTE: Make sure that you have an IP address that you can assign to your target. In addition, make sure you have worked through *F.9 Launching U-Boot*, p.362, and have a prompt visible in your serial terminal.

Enter **printenv** at the prompt in your serial terminal, press **ENTER**. Output displays in the terminal as shown.

```
=> printenv
bootcmd=version;echo;bootp;setenv bootargs root=/dev/nfs rw
nfsroot=$(serverip):
$(rootpath)ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname):eth0:
off;bootm
bootdelay=5
baudrate=115200
ethaddr=00:a0:1e:a8:7b:cb
serverip=192.168.123.205
stdin=serial
stdout=serial
stderr=serial
ver=U-Boot 1.1.1 (Dec  8 2003 - 18:18:49)

Environment size: 326/262140 bytes
=>
```

These are the current U-Boot settings that are stored in the flash memory on your target.

NOTE: The settings displayed above may not match the settings on your target exactly.

F.10.3 Setting the Host Parameters

Configure the host parameters as follows:

1. Enter **setenv serverip** *ipAddressOfHost*.

   This sets the **serverip** parameter to the IP address of your host computer that is running the TFTP server.

2. Enter **setenv netmask** *netmask*.

   This sets the **netmask** parameter to the netmask address of your local network.

3. Enter **setenv gatewayip** *ipAddressOfGateway*.

   If your network requires a gateway IP address, this command sets that variable.

4. Enter **setenv rootpath** *path*.

   Use this command to set the **rootpath** parameter to the directory on your host computer where your root file system is located. If you used the ELDK file system, described in *F.8 Exporting the ELDK Root File System*, p.361, set *path* to */ELDK_installDir/target_architecture*.

5. Enter **setenv baudrate** *baudrate*.

   This command sets the **baudrate** parameter to the baud rate of the U-Boot console connection. If the baud rate you configured U-Boot to use does not match the value shown in **printenv baudrate**, use this command to set *baudrate* to the rate specified in the U-Boot configuration.

   After you change this, the following message displays:

   ```
   ## Switch baudrate to baudrate bps and press ENTER ...
   ```

   This is asking you to change the baud rate on your target to match the baud rate you specify with the **setenv baudrate** command. If you set the baud rate to match the one specified in your U-Boot configuration, press **ENTER** to bypass this step.

F.10.4 Setting the Target Parameters

Next, configure the target parameters.

1. Enter **setenv ipaddr** *ipAddress*.

   This command specifies the IP address that you want to assign to your target. The IP address you assign must not be used anywhere else on your network. Contact your System Administrator if you are not sure which IP address to use.

2. Enter **setenv hostname** *nameOfTarget*

   Use this command to assign a name to your target.

F.10.5 Setting Root File System Parameters

This section describes how to set up the root file system parameters for an NFS mounted file system. The following steps describe how to configure the parameters.

1. Enter **setenv bootfile** *imageName*.

   Use this command to specify the name of your Linux kernel to download.

2. Enter the following command to set the boot arguments:

```
setenv bootargs root=/dev/nfs rw nfsroot=$(serverip):$(rootpath)
ip=$(ipaddr):$(serverip):$(gatewayip):$(netmask):$(hostname):eth0:off
```

⚠ **CAUTION:** Setting the boot arguments on your target is critical. Any error in these parameters may prevent the kernel from booting, or may cause a system failure at a later time.

⚠ **CAUTION:** Enter this command on a single line, and make sure that you include a space between **$(rootpath)** and **ip**.

⚠ **CAUTION:** You must retype the boot arguments every time you change any of the referenced variables. Otherwise you will be unable to load your Linux kernel.

Enter the command on one line, exactly as it is shown. This will reference all the settings you configured in previous steps.

F.10.6 Verifying and Saving the Parameters

Enter **printenv** to display the parameters that you configured. Output displays similar to that shown below.

```
=> printenv
bootdelay=5
ethaddr=56:9a:38:2c:60:04
serverip=172.16.17.17
netmask=255.255.0.0
gatewayip=172.16.1.1
rootpath=/opt/eldk/ppc_82xx
baudrate=115200
ipaddr=172.16.8.88
hostname=sbc8260
bootfile=vmlinux.PPCBoot
bootargs=root=/dev/nfs rw nfsroot=172.16.17.17:/opt/eldk/ppc_82xx
ip=172.16.8.88:172.16.17.17:172.16.1.1:255.255.0.0:sbc8260:eth0:off
stdin=serial
stdout=serial
```

```
stderr=serial
ver=U-Boot 1.1.1 (Dec  8 2003 - 18:18:49)

Environment size: 443/262140 bytes
=>
```

Verify that all the settings are correct for your system. When you are satisfied that they are, enter **saveenv** to save the changes. The following output displays:

```
=> saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... done
Protected 1 sectors
=>
```

When this is complete, your changes are saved in the flash memory on your target.

CAUTION: If you do not save the changes on your target using the **saveenv** command, the settings will be lost when you reboot your target. Make sure you issue the **saveenv** command to preserve your settings.

NOTE: The following sections describe how to download and launch the Linux kernel in two steps. To automate the sequence, see *Automating the Boot Sequence*, p.372.

F.11 Downloading the Kernel to the Target

When you have finished modifying all the parameters on the target, you are ready to download the Linux kernel to your target.

Before you begin, make sure that you:

- Work through all of *F.10 Configuring U-Boot*, p.364, and have a prompt visible in the serial terminal
- Connect an Ethernet cable from your network to the Ethernet port on your target.

To load the kernel, enter **tftpboot** at the prompt and press **ENTER**.

```
=> tftpboot
TFTP from server 172.16.17.17; our IP address is 172.16.8.88
Filename 'vmlinux.PPCBoot'.
Load address: 0x400000
Loading: err: c01
#################################################################
         ########################################
done
Bytes transferred = 545719 (853b7 hex)
=>
```

If the image loads correctly, output similar to the above displays.

NOTE: If **T**s display instead of **#**s during the boot process, there is a problem with your network connection. Check that all cable connections are tight, and confirm that the network addresses you configured are correct.

F.12 Launching the Linux Kernel

When you have finished downloading the kernel, enter **bootm** and press **ENTER** to launch it.

The Linux initialization sequence begins, and output displays as in the example shown below.

```
=> bootm
## Booting image at 00400000 ...
   Image Name:   Linux-2.4.20wdb
   Created:      2004-04-08  14:12:08 UTC
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    545655 Bytes = 532.9 kB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Uncompressing Kernel Image ... OK
Memory BAT mapping: BAT2=64Mb, BAT3=0Mb, residual: 0Mb
Linux version 2.4.20wdb  (user1@machine-linux) (gcc version 2.95.4 20010319
 (prerelease/franzo/20011204)) #1 Thu Apr 8 10:09:58 EDT 2004
On node 0 totalpages: 16384
zone(0): 16384 pages.
zone(1): 0 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/nfs rw nfsroot=172.16.17.17:/opt/eldk/ppc_82xx
ip
=172.16.8.88:172.16.17.17:172.16.1.1:255.255.0.0:sbc8260:eth0:off
Calibrating delay loop... 131.89 BogoMIPS
```

```
Memory: 63316k available (1012k kernel code, 376k data, 52k init, 0k highmem)
Dentry cache hash table entries: 8192 (order: 4, 65536 bytes)
Inode cache hash table entries: 4096 (order: 3, 32768 bytes)
Mount-cache hash table entries: 1024 (order: 1, 8192 bytes)
Buffer-cache hash table entries: 4096 (order: 2, 16384 bytes)
Page-cache hash table entries: 16384 (order: 4, 65536 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
CPM UART driver version 0.01
ttyS00 at 0x0000 is a SMC
ttyS01 at 0x0040 is a SMC
ttyS02 at 0x8100 is a SCC
ttyS03 at 0x8200 is a SCC
pty: 256 Unix98 ptys configured
eth0: SCC ENET Version 0.1, 56:9a:38:2c:60:04
RAMDISK driver initialized: 16 RAM disks of 4096K size 1024 blocksize
loop: loaded (max 8 devices)
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP, IGMP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 4096 bind 4096)
IP-Config: Complete:
      device=eth0, addr=172.16.8.88, mask=255.255.0.0, gw=172.16.1.1,
     host=sbc8260, domain=, nis-domain=(none),
     bootserver=172.16.17.17, rootserver=172.16.17.17, rootpath=
NET4: Unix domain sockets 1.0/SMP for Linux NET4.0.
Looking up port of RPC 100003/2 on 172.16.17.17
Looking up port of RPC 100005/1 on 172.16.17.17
VFS: Mounted root (nfs filesystem).
Freeing unused kernel memory: 52k init
tty_io.c: process 1 (swapper) used obsolete /dev/cua - update software to use
/d
ev/ttyS0
WDB Agent Ready (network:eth0)
INIT: version 2.78 booting
                Welcome to DENX Embedded Linux Environment
                Press 'I' to enter interactive startup.
Mounting proc filesystem:  [  OK  ]
Configuring kernel parameters:  [  OK  ]
Timed out waiting for time change.
Setting clock : Wed Dec 31 19:00:11 EST 2004 [  OK  ]
Activating swap partitions:  [  OK  ]
Setting hostname sbc8260:  [  OK  ]
modprobe: Can't open dependencies file /lib/modules/2.4.20wdb /modules.dep
(No such file or directory)
Finding module dependencies:  depmod: Can't open /lib/modules/2.4.20wdb
/modules
.dep for writing
[FAILED]
Checking filesystems
[  OK  ]
Mounting local filesystems:  [  OK  ]
Enabling swap space:  [  OK  ]
```

```
INIT: Entering runlevel: 3
Entering non-interactive startup
Setting network parameters:  [  OK  ]
Bringing up interface lo:  [  OK  ]
Starting system logger: [  OK  ]
Starting kernel logger: [  OK  ]
Starting ntpd: [  OK  ]
Starting xinetd: [  OK  ]

sbc8260 login:
```

At the **login** prompt, type **root** and press **ENTER**. A prompt displays. Once the prompt displays, your Linux kernel is running properly on your target. To test the kernel, **ping** your Linux host from this shell.

```
bash-2.05# ping 172.16.17.17
PING 172.16.17.17 (172.16.17.17) from 172.16.8.88 : 56(84) bytes of data.
64 bytes from 172.16.17.17: icmp_seq=0 ttl=64 time=483 usec
64 bytes from 172.16.17.17: icmp_seq=1 ttl=64 time=471 usec
64 bytes from 172.16.17.17: icmp_seq=2 ttl=64 time=487 usec

--- 172.16.17.17 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/mdev = 0.471/0.480/0.487/0.019 ms
bash-2.05#
```

Automating the Boot Sequence

To automate the Linux boot process so that it happens automatically when you reset the target, follow the steps below.

1. In the serial terminal, enter **setenv bootdelay** *time*.

   Use this command to specify the number of seconds that you would like the target to wait before booting.

2. Enter **setenv bootcmd tftpboot\; bootm**.

   This creates the script that will run on the target when booting.

3. Enter **saveenv** to save the settings.

   Now you can reset your target, and in *time* seconds, your target will automatically boot.

Once Linux is running on your target, you can connect to it using Workbench.

G

Broken Patch File Example

G.1 The myApache.patch Sample File

This appendix contains the example patch file for Apache, with intentional reject issues meant to be used with the example procedure in *Patch Reject Resolution*, p.62.

This patch file has also been produced as a reverse patch (from new to old, instead of from old to new) for demonstration purposes, which explains why the new text has the "-" markers instead of the expected "+" markers.

Text File myApache.patch

```
--- configure.new2006-09-21 23:38:37.000000000 -0700
+++ configure2006-09-21 23:31:30.000000000 -0700
@@ -14,7 +14,6 @@
 # See the License for the specific language governing permissions and
 # limitations under the License.

-## Here is a Simple Patch

 ##
 ##  configure -- Apache Autoconf-style Interface (APACI)
@@ -26,9 +25,6 @@
 DIFS='
 ` ;# <<< FUZZ FACTOR >>>

-## Here is some patch lines to
-## demonstrate the fuzz factor
-
 ##
 ##  avoid brain dead shells on Ultrix and friends
 ##
@@ -55,9 +51,6 @@
```

```
 configstatus=config.status
 shadow=''

 ## <<< SOURCE FUDGE >>>
-## Here are some patch lines to
-## demonstrate source line editing
-
 ##
 ##  pre-determine runtime modes
 ##
@@ -80,9 +73,6 @@
         ;;
 esac

 ## <<< REJECT ME >>>
-## Here are some patch lines to
-## demonstrate in-line rejection
-
 ##
 ##  determine platform id
 ##
@@ -98,9 +88,6 @@
     echo "Configuring for Apache, Version $APV"
 fi

 ## <<< REJECT ME >>>
-## Here are some more patch lines to
-## also demonstrate in-line rejection
-
 ##
 ##  important hint for the first-time users
 ##
```

Annotated Patch File

```
01  --- configure.new2006-09-21 23:38:37.000000000 -0700
02  +++ configure2006-09-21 23:31:30.000000000 -0700
03  @@ -14,7 +14,6 @@
04   # See the License for the specific language governing permissions and
05   # limitations under the License.
06
07  -## Here is a Simple Patch
08
09   ##
10   ##  configure - Apache Autoconf-style Interface (APACI)
11  @@ -26,9 +25,6 @@
12   DIFS='
13   ' ;# <<< FUZZ FACTOR >>>
14
15      -## Here is some patch lines to
16  -## demonstrate the fuzz factor
17  -
```

```
18   ##
19   ##  avoid brain dead shells on Ultrix and friends
20   ##
21  @@ -55,9 +51,6 @@
22   configstatus=config.status
23   shadow=''
24
25   ## <<< SOURCE FUDGE >>>
26      -## Here are some patch lines to
27  -## demonstrate source line editing
28  -
29   ##
30   ##  pre-determine runtime modes
31   ##
32  @@ -80,9 +73,6 @@
33           ;;
34   esac
35
36   ## <<< REJECT ME >>>
37  -## Here are some patch lines to
38  -## demonstrate in-line rejection
39  -
40   ##
41   ##  determine platform id
42   ##
43
44   ##
45  @@ -98,9 +88,6 @@
46       echo "Configuring for Apache, Version $APV"
47   fi
48
49   ## <<< REJECT ME >>>
50  -## Here are some more patch lines to
51  -## also demonstrate in-line rejection
52  -
53   ##
54   ##  important hint for the first-time users
55   ##
```

Annotation

Line 7: This is a normal patch hunk, that will apply successfully, once the patch is applied with the **Reverse patch** option.

Line 13: This hunk has the extra content **;# <<< FUZZ FACTOR >>>** two lines above the inserted text. This hunk will apply once the **Maximum fuzz factor** is set to **2**.

Line 25: This hunk has the extra content **## <<< SOURCE FUDGE >>>** one line above the inserted text. This cannot be fixed by adjusting the **Maximum fuzz factor**. This hunk can be fixed by either adjusting the source file to include the extra text, or by adjusting the patch file to remove or add the extra text.

Line 36: The last two hunks also have extra text that does not allow the hunks to apply. In the example case, these hunks are included as unresolved rejects, and are automatically placed in a reject file, or included as inline reject hunks within the target source file (if that option is selected).

H

Glossary

This glossary contains terms used in Wind River Workbench. For basic Eclipse terms, see the Eclipse glossary[1].

If the term you want is not listed in one of these glossaries, you can search for it throughout all online documentation.

1. At the top of the **Help > Help Contents** window, type your term into the **Search** field.
2. Click **Go**. Topics containing the term will appear in the **Search Results** list.
3. To open a topic in the list, click it.

For more information about online help, see **Help > Help Contents > Wind River Documentation > Eclipse Platform Documentation > Eclipse Workbench User Guide > Tasks > Using the help system**.

active view

The view that is currently selected, as shown by its highlighted title bar. Many menus change based on which is the active view, and the active view is the focus of keyboard and mouse input.

1. To access the Eclipse glossary, see **Help > Help Contents > Wind River Documentation > Eclipse Platform Documentation > Eclipse Platform Plug-inDeveloper Guide > Reference > Other reference information > Glossary of Terms**.

back end

Functionality configured into a target server which allows it to communicate with various target agents, based on the mode of communication that you establish between the host and the target (network, serial, and so on).

color context

The color assigned to a particular process in the Debug view; this color carries over to breakpoints in the Editor and to other views that derive their context from the Debug view.

cross-development

The process of writing code on one system, known as the host, that will run on another system, known as the target.

debuggable objects

Debuggable objects are the executable application files, kernels, kernel modules, and libraries that can be accessed by both the host and Linux target. These objects are ideally compiled without optimization, compiled with the appropriate debug flags (for example with **-g**, or **-g-dwarf-2**), and are not stripped of symbols.

editor

An Editor is a visual component within Wind River Workbench. It is typically used to edit or browse a file or other resource.

Modifications made in an Editor follow an open-save-close life cycle model. Multiple instances of an Editor type may exist within a Workbench window.

help key

Press the help key in Workbench to get context-sensitive help. The help key is host-dependent. On a Windows host, press **F1**. On a Linux host, press **CTRL+F1**. On a Solaris host, press the **HELP** key.

host shell

A Wind River command shell that provides a command-line environment for GDB and KGDB debugging. The host shell also provides Tcl scripting support.

hunk

A hunk is a contiguous group of source lines generated when the **diff** program is applied to compare files. The **patch** program and the Quilt patch program based upon it use **diff** to create patches, which are then internally represented as one or more hunks to apply to a file to patch it.

JDT

Java Development Toolkit provided by the Eclipse organization (**http://www.eclipse.org**) and included with Workbench.

JNI

Java Native Interface is a means of calling non-Java code (native code) from within a Java application.

kernel mode

For Linux 2.6.10 and higher kernels, two connection modes are supported: kernel and user mode connections. Kernel mode connections allow **kgdb** debugging of the kernel in a manner analogous to debugging applications in user mode.

kernel module

A piece of code, such as a device driver, that can be loaded and unloaded without the need to rebuild and reboot the kernel.

launch configuration

A run-mode launch configuration is a set of instructions that instructs the IDE to connect to your target and launch a process or application. A debug-mode launch configuration completes these actions and then attaches the debugger.

object path mappings

The object path mappings specify where the debuggable objects are to be found for both the debugger running on the host and the Linux target. In Workbench, this is set within the Target Manager view's **Target Connection Properties**.

overview ruler

The vertical borders on each side of the Editor view. Breakpoints, bookmarks, and other indicators appear in the overview ruler.

perspective

A perspective is a group of views and Editors in the Workbench window. One or more perspectives can exist in a single Workbench window. Each perspective contains one or more views and Editors. Within a window, each perspective may have a different set of views but all perspectives share the same set of Editors.

plug-in

An independent module, available from Wind River, the Eclipse Foundation, or from many Internet Web sites, that delivers new functionality to Wind River Workbench without the need to recompile or reinstall it.

program counter

The address of the current instruction when a process is suspended.

project

A collection of source code files, build settings, and binaries that are used to create a downloadable application or bootable system image.

registry

The registry associates a target server's name with the network address needed to connect to that target server, thereby allowing you to select a target server by a convenient name.

source lookup path

The source lookup path specifies the location that the Workbench debugger uses to identify and open each source file as it is being debugged. This is set in the Debug view in Workbench.

system mode

When in system mode, the debugger is focussed on kernel processes and threads. When a process is suspended, all processes stop. Compare with user mode. System and user are the two modes of a dual mode connection, and are supported for Linux 2.4.x kernels only. For 2.6 kernels, separate kernel and user mode connections are supported.

target agent

The target agent runs on the target, and is the interface between Wind River Linux and all other Wind River Workbench tools running on the host or target.

target server

The target server runs on the host, and connects the Wind River Workbench tools to the target agent. There is one server for each target; all host tools access the target through this server.

TIPC

Transparent inter-process communication protocol typically used by nodes within a cluster. Wind River provides a proxy and usermode agent program that allow Workbench to access targets within the TIPC cluster.

user mode

When in user mode, the debugger is focused on user applications and processes. When a process is suspended, other processes continue to run. For Linux 2.4.x kernels, user mode is a part of a dual mode connection. Compare with system mode. For Linux 2.6.10 an higher kernels, user mode is a separate connection type. Compare to kernel mode.

view

A view is a visual component within Workbench. It is typically used to navigate a hierarchy of information (like the resources in your Workbench), open an Editor, or display properties for the active Editor.

Modifications made in a view are saved immediately. Only one instance of a particular view type may exist within a Workbench window.

workspace

The directory where your projects are created. To share the build objects of your projects with a target, the workspace (directory) may be in a file system that is exported to the target, or you may redirect build objects from your workspace to a location exported to the target.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z