

WIND RIVER

Wind River Linux

USER'S GUIDE

5.0.1

Copyright Notice

Copyright © 2014 Wind River Systems, Inc.

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

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

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

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at one of the following locations:

installDir/product_name/3rd_party_licensor_notice.pdf
installDir/legal-notices/

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

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): 800-545-WIND
Telephone: 510-748-4100
Facsimile: 510-749-2010

For additional contact information, see the Wind River Web site:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

15 Jan 2014

Contents

PART I: INTRODUCTION

1 Overview	15
Wind River Linux Overview	15
Kernel and File System Components	16
Supported Run-time Boards	18
Optional Add-on Products	20
Product Updates	22
2 Run-time Software Configuration and Deployment Workflow	27
3 Development Environment	29
Directory Structure	29
Metadata	32
Configuration Files and Platform Projects	33
Assigning Empty Values in BitBake Configuration Files	38
README Files in the Development Environment	39
Viewing a Specific README File in the Installation	39
Cloning a Layer to View Installation README Files	40
Viewing All Installation README Files in a Web Browser	40
4 Build Environment	41
About the Project Directory	41
Creating a Project Directory	42
Directory Structure for Platform Projects	43
Feature Templates in the Project Directory	47
Kernel Configuration Fragments in the Project Directory	51
Viewing Template Descriptions	54
About the layers/local Directory	54
About README Files in the Build Environment	56
Adding a Layer to a Platform Project to View README Files	56
Adding All Layers to a Platform Project to View All README Files	57

PART II: PLATFORM PROJECT IMAGE DEVELOPMENT

5 Configuration and Build	61
Introduction	61
About Creating the Platform Project Build Directory	62
About Configuring a Platform Project Image	62
Initializing the Wind River Linux Environment	63
About the Configure Script	63
About Building Platform Project Images	72
About the make Command	72
Yocto Project Equivalent make Commands	73
About Build Logs	75
Build-Time Optimizations	76
Examples of Configuring and Building	77
Configuring and Building a Complete Run-time	77
Commands for Building a Kernel Only	78
Configuring and Building a Flash-capable Run-time	78
Configuring and Building a Debug-capable Run-time	79
Building a Target Package	80
About Creating Custom Configurations Using rootfs.cfg	80
About the rootfs.cfg File	82
About New Custom rootfs Configuration	84
EGLIBC File Systems	85
Creating and Customizing EGLIBC Platform Project Images	86
EGLIBC Option Mapping Reference	88
6 Localization	91
About Localization	91
Determining which Locales are Available	91
Setting Localization	93
7 Portability	95
About Platform Project Portability	95
Copying or Moving a Platform Project	96
Updating a Platform Project to a New Wind River Linux Installation Location	96
8 Layers	99
About Layers	99
Layers Included in a Standard Installation	100
Installed Layers vs. Custom Layers	102
Layer Structure by Layer Type	103
About Layer Processing and Configuration	105
About Processing a Project Configuration	105
Creating a New Layer	106

Enabling a Layer	107
Disabling a Layer	108
9 Recipes	109
About Recipes	109
A Sample Application Recipe File	110
About Recipe Files and Kernel Modules	111
Extending Recipes with .bbappend Files	111
Creating a Recipe File	112
Identifying the LIC_FILES_CHKSUM Value	113
10 Templates	115
About Templates	115
Adding Feature Templates	116
Adding Kernel Configuration Fragments	117
11 Finalizing the File System Layout with changelist.xml	119
About File System Layout XML Files	119
About File and Directory Management with XML	119
Device Options Reference	120
Directory Options Reference	121
File Options Reference	122
Pipe Options Reference	123
Symlink Options Reference	123
The Touched/Accessed touch.xml Database File	124

PART III: USERSPACE DEVELOPMENT

12 Developing Userspace Applications	127
Introduction	127
About Application Development	127
Cross Development Tools and Toolchain	128
About Sysroots and Multilibs	128
Creating a Sample Application	132
Exporting the SDK	136
Exporting the SDK	136
Exporting the SDK for Windows Application Development	137
Adding Applications to a Platform Project Image	138
Options for Adding an Application to a Platform Project Image	138
Adding New Application Packages to an Existing Project	139
Adding an Application to a Root File System Using changelist.xml	140

Adding an Application to a Root File System with fs_final*.sh Scripts	141
Configuring a New Project to Add Application Packages	142
Verifying the Project Includes the New Application Package	143
Importing Packages	144
About the Package Importer Tool (import-package)	144
Importing a Sample Application Project as a Package	144
Importing a Source Package from the Web (wget)	146
Importing a SRPM Package from the Web	148
Listing Package Interdependencies	151
13 Understanding the User Space and Kernel Patch Model	153
Patch Principles and Workflow	153
Patching Principles	154
Kernel Patching with scc	155
14 Patching Userspace Packages	159
Introduction to Patching Userspace Packages	159
Patching with Quilt	160
Create an Alias to exportPatches.tcl to save time	161
Preparing the Development Host for Patching	161
Patching and Exporting a Package to a Layer	162
Verifying an Exported Patch	164
Incorporating a Patch into a Platform Project Image	165
15 Modifying Package Lists	167
About the Package Manager	167
Launching the Package Manager	168
Removing Packages	169
About Modifying Package Lists	172
Adding a Package	173
About Adding Templates	173
Removing a Package	174

PART IV: KERNEL DEVELOPMENT

16 Patching and Configuring the Kernel	177
About Kernel Configuration and Patching	177
Configuration	178
Patching	189
17 Creating Alternate Kernels from kernel.org Source	199

18 Exporting Custom Kernel Headers	201
About Exporting Custom Kernel Headers for Cross-compile	201
Adding a File or Directory to be Exported when Rebuilding a Kernel	201
Exporting Custom Kernel Headers	202
19 Using the preempt-rt Kernel Type	205
Introduction to Using the preempt-rt Kernel Type	205
Enabling Real-time	207
Configuring preempt-rt Preemption Level	207

PART V: DEBUGGING AND ENABLING ANALYSIS TOOLS SUPPORT

20 Kernel Debugging	213
Kernel Debugging	213
Debugging with KGDB Using an Ethernet Port (KGDBOE)	214
Debugging with KGDB Using the Serial Console (KGDBOC)	216
Disabling KGDB in the Kernel	217
Kernel Debugging with QEMU	218
21 Userspace Debugging	219
Adding Debugging Symbols to a Platform Project	219
Adding Debugging Symbols for a Specific Package	220
Dynamic Instrumentation of User Applications with uprobes	221
Configuring uprobes with perf	222
Dynamically Obtain User Application Data with uprobes	223
Dynamically Obtain Object Data with uprobes	225
Debugging Individual Packages	228
Debugging Packages on the Target Using gdb	228
Debugging Packages on the Target Using gdbserver	229
22 Analysis Tools Support	231
About Analysis Tools Support	231
Using Dynamic Probes with ftrace	231
Preparing to use a kprobe	233
Setting up a kprobe	234
Enabling and Using a kprobe	235
Disabling a kprobe	236
Analysis Tools Support Examples	237
Adding Analysis Tools Support for MIPS Targets	237
Adding Analysis Tools Support for Non-MIPS Targets	237

PART VI: USING SIMULATED TARGET PLATFORMS FOR DEVELOPMENT

23 QEMU Targets	241
QEMU Targets	241
QEMU Target Deployment Options	241
QEMU Targets	244
TUN/TAP Networking with QEMU	246
24 Wind River Simics Targets	251
Wind River Simics Targets	251
Using Simics from the Command Line	251

PART VII: DEPLOYMENT

25 Managing Target Platforms	257
Customizing Password and Group Files	257
Using an fs_final.sh Script to Edit the Password and Group File	258
Using an fs_final_sh Script to Overwrite the Password and Group File	259
About Idconfig	259
Enabling Idconfig Support	259
Connecting to a LAN	260
Adding an RPM Package to a Running Target	261
Adding Reference Manual Page Support to a Target	262
Using Pseudo	263
About Using Pseudo (fakestart.sh)	263
Examining Files using Pseudo	263
Navigating the Target File System with Pseudo	264
26 Deploying Flash or Disk Target Platforms	265
About Configuring and Building Bootable Targets	265
About Configuring a Boot Disk with a USB/ISO Image (Two Build Directories)	266
Host-Based Installation of Wind River Linux Images	266
Booting and Installing from a USB or ISO Device	268
Booting and Installing with QEMU	269
Configuring and Building the Host Install (Self-Contained Option)	273
Configuring and Building the Host Install (Two Build Directories Option)	274
Creating Bootable USB Images	275
Creating ubifs Bootable Flash Images	278

Enforcing Read-only Root Target File Systems	279
Installing a Bootable Image to a Disk	279
Installing or Updating bzImage	280
27 Deploying initramfs System Images	283
About initramfs System Images	283
Creating initramfs Images	284
Adding Packages to initramfs Images	285
28 Deploying KVM System Images	287
About Creating and Deploying KVM Guest Images	287
Create the Host and Guest Systems	289
Deploying a KVM Host and Guest	290

PART VIII: TESTING

29 Running Linux Standard Base (LSB) Tests	295
About the LSB Tests	295
Testing LSB on Previously Configured and Built Target Platforms	296
Disabling Grsecurity Kernel Configurations on CGL Kernels	297
Running LSB Distribution Tests	297
Running LSB Application Tests	299

PART IX: OPTIMIZATION

30 About Optimization	305
31 Analyzing and Optimizing Runtime Footprint	307
Analyzing and Optimizing Runtime Footprint	307
Collecting Platform Project Footprint Data	308
Footprint (fetch-footprint.sh) Command Option Reference	311
32 Reducing the Footprint	313
About BusyBox	313
Configuring a Platform Project Image to Use BusyBox	313
About devshell	314
About Static Linking	314
About the Library Optimization Option	315
33 Analyzing and Optimizing Boot Time	317

Analyzing and Optimizing Boot Time	317
Creating a Project to Collect Boot Time Data	318
Analyzing Early Boot Time	319
About Reducing Early Boot Time	321
Reducing Network Initialization Time with Sleep Statements	321
Reducing Device Initialization Time	323
Removing Unnecessary Device Initialization Times	323

PART X: TARGET-BASED NETWORKING

34 About Target-based Networking	329
35 Setting Target and Server Host Names	331
36 Connecting a Board	333
Configuring a Serial Connection to a Board	333
Setting-up cu and UUCP	333
Setting up the Workbench Terminal	334
About Configuring PXE	334
Configuring PXE	336
Configuring DHCP	337
Configuring DHCP for PXE	338
Configuring NFS	340
Configuring TFTP	341

PART XI: REFERENCE

37 Additional Documentation and Resources	345
Document Conventions	345
Wind River Linux Documentation	346
Additional Resources	347
Open Source Documentation	348
External Documentation	350
38 Common make Command Target Reference	351
39 Build Variables	361
40 Package Variable Listing	367
41 Lua Scripting in Spec Files	369

42 Kernel Audit Directory Contents 371

PART I

Introduction

Overview.....	15
Run-time Software Configuration and Deployment Workflow.....	27
Development Environment.....	29
Build Environment.....	41

1

Overview

Wind River Linux Overview 15

Wind River Linux Overview

Wind River Linux is a software development environment that creates optimized Linux distributions for embedded devices.

Wind River Linux 5.0.1 is based on the Yocto Project implementation of the OpenEmbedded Core (OE-Core) metadata project. The Yocto Project uses build recipes and configuration files to define the core platform project image and the applications and functionality it provides.

Wind River Linux builds on this core functionality and adds Wind River-specific extensions, tools, and services to facilitate the rapid development of embedded Linux platforms. This support includes:

- Straightforward platform project configuration, build, and deployment that simplifies Yocto Project development
- A range of popular BSPs to support most embedded hardware platforms
- An enhanced command-line-interface (CLI) to the system
- Developer-specific layer for platform project development and management
- Platform project portability to copy or move platform projects, or create a stand-alone platform project
- A custom USB image tool for platform project images

If you have experience with using the Yocto Project, Wind River Linux supports all Yocto Project build commands, but also offers simplified configure and build commands based on the Wind River Linux 4 build system. This functionality greatly reduces your development time. Wind River Linux provides development environments for a number of host platforms and supports a large and ever-growing set of targets, or the platform hardware you are creating your embedded system for. For details on which development hosts are supported, refer to the Release Notes. For supported target boards, refer to Wind River Online Support.

The build system consists of a complete development environment that includes a comprehensive set of standard Linux run-time components, both as binary and source packages. It also includes cross-development tools that can be used to configure and build customized run-time systems and applications for a range of commercial-off-the-shelf (COTS) hardware.

Wind River supports boards according to customer demand. Please contact Wind River if yours is not yet officially supported.

Wind River Workbench is included as part of Wind River Linux to provide a robust development and debugging environment.

For more information about Wind River Linux, see <http://www.windriver.com/products>



NOTE: Wind River Linux 5 uses a different build system than previous product releases. As a result, the new Yocto Project-based build system may not directly support custom layers from a previous release. Custom layers of this type will need to be migrated to the new build system. For information on migrating these layers, see the *Wind River Linux Migration Guide: About Migrating Wind River Linux Projects*.

Kernel and File System Components

Wind River Linux supports a range of kernel type profiles and file systems.



NOTE: Not all kernel feature and file system combinations are supported on any particular board. For further information on validated combinations, contact your Wind River representative.

Kernel Type Profiles

A kernel type profile implements a supported set of kernel features. Each contains features that are compatible with each other and excludes features that are not compatible. Kernel type profiles use a combination of kernel configuration, kernel patches, and build system changes to support their features.

The kernel types are layered to build a set of increasingly specific or enhanced functionality. The set of features that is available and tested on all boards is called the standard kernel profile. This standard profile is included with Wind River Linux 5.0.1.

Kernel type profiles that add or modify the functionality of the standard profile are called *enhanced kernel profiles*. Enhanced profiles are available on a selected set of boards and are mutually exclusive with other enhanced profiles, such as those included with our add-on products. A single board may be supported by multiple mutually exclusive (runtime) enhanced profiles along with the standard profile.



NOTE: All features of the standard kernel profile work within any particular enhanced profile for future Wind River Linux product releases.

Wind River Linux provides the following kernel type profiles:

standard

All boards support the standard profile, fundamental kernel features are implemented in this profile to provide a common platform for all boards.

cgl

This is the Carrier Grade Linux profile, designed to support the CGL specification from the Linux Foundation. See <http://www.linuxfoundation.org/collaborate/workgroups/cgl> for a summary and details on the CGL specification.

This option is available with the Wind River Linux Carrier Grade Profile add-on product. For additional information, see [Optional Add-on Products](#) on page 20 .

preempt_rt

This kernel profile provides the **PREEMPT_RT** kernel patches to enable conditional hard real-time support. Note that this profile does not imply deterministic pre-emption. For additional information, see [Introduction to Using the preempt-rt Kernel Type](#) on page 205.



NOTE: A single board may be in one or more enhanced kernel profiles.

For detailed instructions on reconfiguring and customizing Wind River Linux kernels, see [About Kernel Configuration and Patching](#) on page 177

Supported File Systems

Root file systems are defined in the `projectDir/layers/wr-base/templates/rootfs.cfg` file. There are five basic file systems:

Glibc_Core (glibc_core)

A smaller footprint version of the Glibc Standard file system, including all packages necessary to boot a smaller file system that is not based on BusyBox. This includes the standard kernel, a minimal BusyBox, and sysvinit.

Glibc Standard (glibc_std)

A full file system, with Glibc but without CGL-relevant packages or extensions.

Glibc Standard Sato (glibc_std_sato)

A full file system with Glibc, optimized for the Sato graphical (sato-gui) interface. Sato is part of Poky, the Yocto Project platform builder. For additional information, see:

<http://www.yoctoproject.org/docs/1.0/poky-ref-manual/poky-ref-manual.html#what-is-poky>

Glibc Small (glibc_small)

A much smaller, BusyBox-based file system, with Glibc. This includes a standard kernel, a reduced size BusyBox, and sysvinit. Note that with this file system, device nodes are not created automatically. You must create them manually.

Glibc CGL (glibc_cgl)

A full file system, with CGL-relevant packages and CGL extensions. This option is available with the Wind River Linux Carrier Grade Profile add-on product. For additional information, see [Optional Add-on Products](#) on page 20.

Run-time components are available as source and as pre-built binaries.

Combinations of File System and Kernel Feature Profiles

The following table shows which file systems are available with each kernel profile.

File Systems	Standard	preempt_rt	cgl	Tiny
glibc_core	Yes	Yes	No	No

File Systems	Standard	preempt_rt	cgl	Tiny
glibc_std	Yes ¹	Yes	No	No
glibc_std_sato	Yes	Yes	No	No
glibc_small	Yes	Yes	No	No
glibc_cgl	No ²	No	Yes	No



NOTE: Contact your Wind River representative for details on which kernel features and file systems are supported for your board.

Kernel Features

Wind River Linux 5.0.1 supports the following, additional kernel features:

- Support for the Intel® Performance Counter Monitor (PCM) for Intel® Xeon® Core and Intel® Atom® BSPs. The Intel® PCM provides sample C++ routines and utilities to estimate the processor internal resource utilization to help developers gain significant performance boosts for their platforms.

For information on using the Intel® PCM, see <http://software.intel.com/en-us/articles/intel-performance-counter-monitor>.

- Support for **turbostat**, a Linux tool to observe the proper operation on systems that use Intel® Turbo Boost Technology (<http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html>). For additional information on **turbostat**, see <http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=103a8fea9b420d5faef43bb87332a28e2129816a>.

For a list of additional kernel features, listed as configuration fragments you can add to any platform project image, see [Kernel Configuration Fragments in the Project Directory](#) on page 51.

Supported Run-time Boards

Supported run-time boards define the target platforms that Wind River Linux supports.

Wind River Linux comes complete with pre-built Linux kernels and pre-built run-time file system packages (and will build identical and configurable kernels and file systems from source) for many boards from a variety of manufacturers. For the most recent list of supported boards, see Wind River Online Support.

Bootloaders and Board README Files

In most cases, you just use the boot loader that comes with the board to boot Wind River Linux. Wind River supplies a boot loader for boards if the boot loader that comes with the board

¹ In some cases this combination may not be supported as it is not needed on equipment used for networking only. Individual board **README** files contain details.

² In cases where a board cannot support the **cgl** kernel profile it instead supports the **standard** kernel profile and the **glibc_cgl** root file system with some features of the user space gracefully failing for lack of kernel support.

requires modification to work optimally with Wind River Linux. Any Wind River-provided bootloader will be contained in the BSP template along with the **README** file for the board.



NOTE: Wind River strongly recommends that you read the **README** file for your board, located within `installDir/wrlinux-5/layers/wr-bsps/boardname`. This file contains important information on board bring-up, boot loaders, board features, and board limitations. The board **README** and other **README** files can also be found in your `projectDir/READMEES` directory after you configure a project.

Information on setting up target servers and booting supported boards, as well as details on booting with ISO, hard disk, and flash RAM, can be found in [About Configuring and Building Bootable Targets](#) on page 265 and [About Target-based Networking](#) on page 329.

BitBake Name Limitations

Currently, the BitBake build system does not support custom BSP names with underscore (_) characters. For example, if you create a custom BSP and name it `intel_x86_64_custom_headers`, when you try to configure and build the project, the build system will replace each of the underscore characters with dashes (-), effectively changing your BSP name to `intel-x86-64-custom-headers`. If your BSP or other project configuration files have a dependency on the explicit name of the BSP, then this will cause the build to fail.

To prevent failures of this nature, do not use underscore characters in your BSP names.

BSP Name Cross-reference List

Previous releases of Wind River Linux provided BSPs that have been renamed, and improved to work with Wind River Linux 5.0.1. The following table provides correlation between Wind River Linux 4.x and 5.0.1.

Processor Family	4.x BSP Name	5.0.1 BSP Name
ARM	stm_spear13xx	stm_spear13xx
ARM	ti_omap3530evm	ti_omap3
MIPS	cav_octeon_cn63xx cav_octeon_cn68xx cav_octeon2	cav_octeon2
PPC	fsl_p2020	fsl_p2020
PPC	fsl_p204x fsl_p3041 fsl_p4080	fsl_e500mc
PPC	fsl_p50x0	fsl_p50xx
PPC	lsi_acp3400	lsi_acp34xx
x86	common_pc_64 intel_core_qm57_pch intel_core_qm67_pch intel_core_qm77_pch intel_xeon_3420_pch	intel_xeon_core

Processor Family	4.x BSP Name	5.0.1 BSP Name
x86	intel_xeon_5520_ioh	
	intel_xeon_c600_pch	
	intel_xeon_core_dh89xx_pch	
	westmere_ep	
	intel_atom_eg20t_pch	intel_x86
QEMU MIPS	intel_atom_z530	
	kbc_km2m806	
	intel_atom_n4xx_d5xx_82801hm	
	qemu_mips32	qemumips
QEMU PPC	qemu_ppc32	qemuppc
QEMU x86 (32-bit)	Part of common_pc	qemux86
QEMU x86 (64-bit)	Part of common_pc_64	qemu86_64
Kernel Virtual Machine (KVM) x86	common_pc_64_kvm_guest	x86-64-kvm-guest

Optional Add-on Products

Wind River Linux supports optional add-on products that extend the capability of the product to meet your current and future development needs.

This version of Wind River Linux supports the following add-on products:

Wind River Linux Carrier Grade Profile

The Wind River Linux Carrier Grade Profile add-on product extends your development capabilities to create platforms that meet Carrier Grade Linux (CGL) requirements. This adds the following features to your Wind River Linux 5.0.1 installation:

- Configuration profile to easily create CGL platforms for all supported run-time boards.
- Tests for validating your CGL platform
- Virtual routing and forwarding support

For additional information, or to purchase the Wind River Linux Carrier Grade Profile, contact your Wind River sales representative.

Wind River Linux Open Virtualization Profile: Virtual Node

The Open Virtualization Profile (OVP) is a Wind River Linux add-on that provides the resources and tools required to build, deploy, and manage a virtualization environment built around open components.

The OVP builds on the Wind River Linux kernel to provide performance enhancements, management extensions, and application services for virtualized systems.

OVP incorporates technologies that enable unique features and services throughout the overall solution, from low level kernel performance improvements for host and guest systems, to the integration of system tuning, profiling, and benchmarking tools. OVP also provides an extensive set of security and management interfaces for application development with full

support for remote management of the virtualized environment. Professional services, training, and a comprehensive set of documentation complement the OVP to provide an open and high performance virtualization solution.

For additional information, or to purchase Wind River Linux Open Virtualization Profile, contact your Wind River sales representative.

Wind River Linux Performance Studio for Intel® Architecture (IAPS)

Performance Studio integrates the latest generation of Intel development tools: with Wind River Linux 4.2 and higher to deliver dramatic performance and productivity enhancements for teams developing embedded applications on Intel® Architecture platforms. Optimizing your code means you can tap the full power of your hardware—using your Intel embedded platform of choice. This includes the Intel platforms running the latest embedded Core™, Xeon®, or Atom™ processors.

Performance Studio consists of the following tools:

- Intel® C/C++ Compiler, to boost performance on Intel architectures
- Intel® Integrated Performance Primitives, which provide platform-optimized algorithms, code samples and APIs for high-bandwidth applications
- Intel® VTune™ Amplifier XE, which delivers actionable analysis of code behavior and performance without having to instrument the source code

Wind River Test Management

Wind River Test Management is a test management solution that identifies high-risk segments in production code, enabling change-based, optimized testing, using real-time instrumentation of devices under test.

Wind River Test Management provides the following:

- Coverage and performance metrics on the same code you ship to customers
- Optimized test suite generator that runs only the test needed to validate changes
- Full-featured lab management system and a universal, open test execution engine to run any type of test on any device

For additional information, or to purchase Wind River Test Management, contact your Wind River sales representative.

Wind River Simics System Simulator

Wind River Simics is a fast, functionally-accurate, full system simulator. Simics creates a high-performance virtual environment in which any electronic system – from a single board to complex, heterogeneous, multi-board, multi-processor, multicore systems – can be defined, developed and deployed.

See [Wind River Simics Targets](#) on page 251 for basic information on using Simics with Wind River Linux.

Simics enables companies to adopt new approaches to the product development life cycle resulting in dramatic reduction in project risks, time to market, and development costs while also improving product quality and engineering efficiency. Simics allows engineering, integration and test teams to use approaches and techniques that are simply not possible on physical hardware.

To purchase Wind River Simics, contact your Wind River sales representative.

Product Updates

Wind River delivers new fixes, and occasionally new functionality, through the product installer.

Overview

When updates become available, they are posted to Wind River Customer Support. This includes new Rolling Cumulative Patch Layer(RCPL) releases which may include new functionality and documentation updates.

You can log into Wind River Customer Support at <http://www.windriver.com/support/> to view product information, or run the Wind River Maintenance Tool to obtain product updates specific to your Wind River Linux license.

About RCPL Releases

With RCPL patches, Wind River provides the flexibility of letting developers choose to incorporate the latest patch, or continue to use the version their project was initially developed with. Since updates are selected at project configuration, and not product installation, installing a new patch release does not force existing projects to use the new changes.

This allows one installation to support projects based off of older patch releases and also support new projects based off the new patch release. At anytime, you can use the **make upgrade** command to bring a platform project up to the latest RCPL release. For additional information, see [Updating Wind River Linux](#) on page 23.

Once you have updated the product, note that for new platform projects, the default for new **configure** script commands is to use the most recent RCPL installed on the machine. To specify an earlier patch release, you can use the **--with-rcpl-version=000x configure** script option, where *000x* is the RCPL version, for example, *0006*. This allows you to reproduce a previous environment, such as one used to release a product.

About the Experimental Feature Layer

The Experimental Feature Layer provides optional new features, new packages or package upgrades. If you want to use these new packages or features, the Experimental Feature Layer can be added to your installation. It not added by default. Once the Experimental Feature Layer has been installed these new features or packages can be added to the project using the appropriate options to the **configure** command. Without these **configure** options the Experimental Feature Layer will not be used by the project.



WARNING: The Experimental Feature Layer may contain up-revved versions of packages in the standard product and/or additional packages which have not been fully regression tested. While Wind River strives to maintain binary compatibility with the rest of the standard system, including this layer may cause issues in non-standard environments. All related bugs reported via Wind River Support channels will be accepted and investigated.

See [Updating Wind River Linux](#) on page 23 for details on installing the Experimental Feature layer.

The Experimental Feature layer currently provides these four packages:

libedit-20121213-3.0

A BSD replacement for libreadline.

fuse-2.9.3

FUSE (Filesystem in Userspace) is a simple interface for user-space programs to export a virtual file system to the Linux kernel. FUSE also aims to provide a secure method for non privileged users to create and mount their own file system implementations.

nmap-6.40

Nmap ("Network Mapper") is a free and open source utility for network discovery and security auditing.

syslog-ng-logrotate

This is a package that adds logrotate capability to the **syslog-ng** package.

Updating Wind River Linux

To obtain product and documentation updates, you will run the Wind River product maintenance tool.

The updated product and documentation will be automatically placed at the correct locations in your installation.



NOTE: The product maintenance tool can retrieve documentation updates and any additional functionality you are entitled to. During the update process, you can select the specific features you want to install.

To install an online update, follow the steps below.

Step 1 Close Wind River programs.

Before installing online updates with the maintenance tool, it is recommended that you exit any Wind River programs or tools that may be running, including the Wind River registry. If the maintenance tool is blocked by a process, it displays an error, showing the process ID.



NOTE: Before you exit Workbench, you can start the maintenance tool by selecting **Help > Update Wind River Products**.

Step 2 Launch the maintenance tool.

If you launched the maintenance tool from Workbench prior to exiting, proceed to Step 3. Otherwise, to start the maintenance tool, run the following commands from a command prompt:

```
$ cd installDir/maintenance/wrInstaller/hostType  
$ ./wrInstaller
```

Step 3 Proceed through the maintenance tool screens.

Choose the available product and documentation updates. For detailed instructions on configuring the maintenance tool, see the Help system within the installer program.

If you wish to use the optional and experimental [Experimental Feature Layer](#), see [Installing the Experimental Feature Layer](#) on page 24

Step 4 Update platform projects to the latest RCPL build.

Wind River provides a method to update platform projects to the latest installation build.

- a) Navigate to the platform project directory.
- b) Update the platform project.

```
$ make upgrade
```

Once the command completes, the platform project is configured to use the latest product update(s).

- c) Rebuild the platform project.

```
$ make
```

Installing the Experimental Feature Layer

Install the Experimental (Rolling) Feature Layer (RCFL) to use the additional features it provides in your platform project build.

To install the Experimental Feature Layer, you must first install the latest RCPL product update as described in [Updating Wind River Linux](#) on page 23. This update is required for the contents of the Experimental Feature Layer to work.



NOTE: This product add-on is optional and is not enabled by default.

Step 1 Start the Product Maintenance Tool.

```
$ installDir/maintenance/wrInstaller/x86-linux2/wrInstaller
```

Step 2 Select **Configure** to display the Configure online settings window.

Step 3 Add the Experimental Feature Layer location to the Product Maintenance Tool locations list.

- a) Click **Add** to display the Add Site window.
- b) Enter a name for the location.

For example, enter **Wind River Experimental Feature Layer**.

- c) Enter the following location.

<http://updates.windriver.com/repos/wrlinux/wrlinux-5.0/DVD-R180065.1-1-00.repos>

- d) Click **OK** to save the new location.
- e) Click **Apply** to return to the main Product Maintenance Tool window.
- f) Select **Online Content** and click **Next** to continue.

Step 4 Continue through the Product Maintenance Tool installation process.

The **Wind River Linux Experimental Feature Layer** will display in the list of available updates. Once installation completes successfully, it will be available for use.

Adding Packages from the Experimental Feature Layer

You can use the Experimental Feature layer to add extra features to a Wind River Linux project.

You can add packages from the Experimental Features Layer to a Wind River Linux project using the Wind River **configure** command.

Step 1 Run **configure** using the **--enable-addons=wr-rcfl** and **--with-layer=rcfl** options to add packages from the Experimental Feature Layer to your project.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--with-template=feature/debug,feature/analysis,features/syslog-ng-logrotate \  
--with-layer=wr-intel-support \  
--enable-addons=wr-rcfl \  
--with-layer=rcfl
```

Step 2 Include any of the packages provided by the Experimental Feature Layer.

To do this, you would use the **make -C build *packageName.addpkg*** command for each package that you want to add.

For example:

- To include **libedit**:

```
$ make -C build libedit.addpkg
```

- To include **FUSE**:

```
$ make -C build fuse.addpkg
```

- To include **nmap-6.40**:

```
$ make -C build nmap.addpkg
```

- If you used **--with-template=feature/debug,feature/analysis,features/syslog-ng-logrotate** when you ran **configure**, as shown in step 1 on page 25, it is included on your target. Otherwise, you can do so at any time using the following command prior to the build:

```
$ echo IMAGE_INSTALL += syslog-ng-logrotate >> project/default-image.bb
```

Step 3 Rebuild the platform project file system.

```
$ make fs
```

When these steps have completed successfully, the package(s) have been added to your project.

2

Run-time Software Configuration and Deployment Workflow

Use this information as a guideline for creating and deploying a complete run-time system.

Overview

In this context, run-time software refers to the platform project image and applications you create as part of developing your complete target operating system. There are different workflows for developing platform projects and/or application projects. For an overview with working examples of these workflows, see:

- *Wind River Linux Getting Started Guide: Platform Project Workflow*
- *Wind River Linux Getting Started Guide: Application Development Workflow*

In this section, we provide a detailed workflow that includes the entire run-time system, with links to relevant material to aid in your development. This includes planning the system, configuring and building the run-time system, customizing the system and adding applications, deploying to a target, and debugging with Workbench or some other tool.

Creating and deploying run-time software with Wind River Linux Platforms includes the following development sequence of events:



NOTE: If you already know what your project requires, and want to start working with Wind River Linux right away, see the *Wind River Linux Getting Started Guide* for information on creating, modifying, and debugging a run-time software platform.

Workflow

1. Plan your platform and/or application projects for your own use or your customers' needs.

This includes choosing a board, kernel, and root file system and determining any special hardware/software requirements your project may need.

2. Create the platform project directory.

The platform project directory is a directory that you create in the build environment, as opposed to the development environment. For additional information, see: [About Creating the Platform Project Build Directory](#) on page 62

For additional information on the build and/or development environments, see:

- [Directory Structure for Platform Projects](#) on page 43
 - [Directory Structure](#) on page 29
3. Configure and build the project.

Within your `projectDir`, issue a `configure` command with the necessary options to configure the appropriate build environment and makefiles. You then issue a `make` command to build a complete platform including the kernel and root file system.

For additional information, see:

- [Introduction](#) on page 61
 - [About Configuring a Platform Project Image](#) on page 62
 - [About Building Platform Project Images](#)
4. Launch the platform project image on the target.

You run the platform project image and any additional applications on a target so you can develop the system. A target may consist of a hardware board or it may be a virtual target for development purposes. If your target does not exactly match one of the supported boards, you can create a custom board support package, generally based on one of the provided definitions. Contact your Wind River representative information regarding creating custom BSPs.

For additional information, see:

- [QEMU Targets](#) on page 241
 - [Wind River Simics Targets](#) on page 251
 - [About Target-based Networking](#) on page 329
5. Update, develop, and debug the project.



NOTE: See the *Wind River Linux Getting Started Guide: About Developing Platform Projects* for hands-on examples on modifying and debugging your platform project.

Updating may require adding and/or patching packages, debugging an application on the target, and making necessary configurations for the host and target to communicate effectively.

Platform developers create a platform project and then produce a sysroot (with `make export-sdk`) for application developers. See [Exporting the SDK](#) on page 136. The sysroot provides the target runtime libraries and header files for use by the application developers on their development hosts. Because the sysroot duplicates application dependencies of the eventual runtime environment, applications are easily deployed after development.

Platform developers can also:

- Incorporate developed applications in a project. See [About Application Development](#) on page 127.
 - Debug applications and the kernel. See [Kernel Debugging](#) on page 213 and the section *Userspace Debugging*.
6. Optimize the platform project image.

In this context, optimization includes analyzing and optimizing the runtime footprint and/or boot time.

For additional information, see [About Optimization](#) on page 305.

3

Development Environment

[Directory Structure](#) 29

[Metadata](#) 32

[Configuration Files and Platform Projects](#) 33

[README Files in the Development Environment](#) 39

Directory Structure

Learn about the structure and content of the development environment including the function and contents of the prominent structural features of Wind River Linux and the BitBake build system—layers, recipes, and templates.

The build environment, including the use of the **configure** script and the **make** command to build runtime software, is described in *Directory Structure for Platform Projects* on page 43.

The Wind River Linux development environment can be installed anywhere on a supported host. This document uses the convention that it is installed in the `/opt/WindRiver/` directory. Throughout this document, the installed location is referred to as *installDir*.

The Yocto Project BitBake build system relies on a system *installDir* path that does not have any of the following characters:

Character	Description
+	Plus symbol
@	'At' symbol
~	Tilde symbol
^	Carat symbol
#	Pound symbol

Having any of these characters in the *installDir* path will cause project builds to fail.

Installed Development Environment Directory Structure

The following figure illustrates part of the development environment structure. Note that this structure includes a combination of Yocto Project and Wind River Linux components.

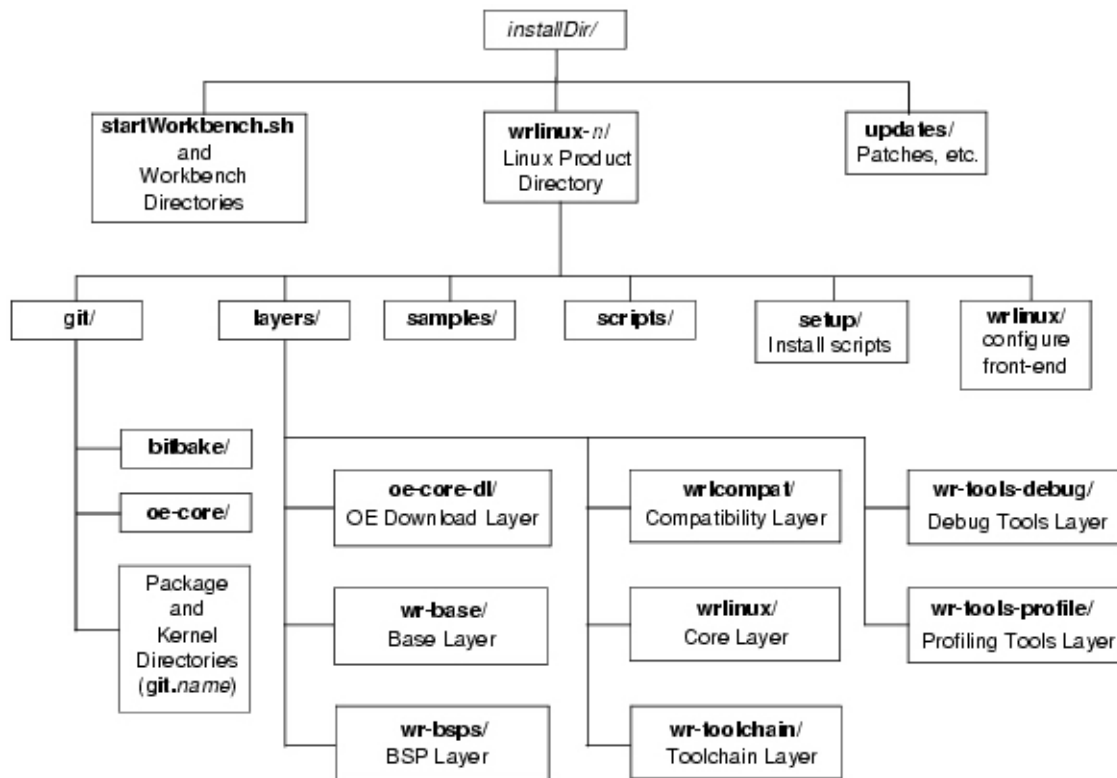


Figure 1: Development Environment Directories

The structure shown in [Development Environment Directories](#) depicts a simplified view of the development environment. When you compare this to the build environment in [Directory Structure for Platform Projects](#) on page 43, you can see that the development environment includes the layers (everything under **layers/**) necessary for configuring a platform project image.

➔ **NOTE:** Not all layers are shown in the preceding figure and additional layers may be added.

The directories and executables shown in the figure, above, are discussed in the following sections.

startWorkbench.sh and the Workbench Directories

The **startWorkbench.sh** shell script starts the Workbench application. You can start Workbench through clicking a desktop icon or, from the command line, entering the path and name of the script. Workbench is introduced in the *Wind River Workbench User's Guide*, and examples of its use are in *Wind River Workbench by Example, Linux Version*.

➔ **NOTE:** This script and the following directories are only available when you purchase and install Wind River Workbench.

Not specifically shown in *Development Environment Directories* are several directories of interest to Workbench users:

workbench-3.3/

Contains the Wind River Workbench installation.

docs/

Contains the documentation for the online help system. The **.html** and **.pdf** files may also be accessed directly by browsing. See *Wind River Linux Documentation* on page 346 for additional information.

The updates Directory

Use this location for patches and other updates from Wind River. In a new or default installation, this directory is typically empty.

The wrlinux-5 Directory

The **wrlinux-5/** directory contains the Wind River Linux development environment, with the contents as shown in and discussed in this section. Subdirectories include:

adt

Though not shown in *Development Environment Directories*, this directory contains the files required for using the Yocto Project Application Developer's Toolkit.

docs

Though not shown in *Development Environment Directories*, this directory contains intellectual property (ip) product disclosure documents.

git/

Contains the **oe-core** git repository with packages and configuration files used by BitBake to configure and build platform project images. This concept of using a live git repository in the build environment is new to Wind River Linux 5. After you configure a platform project, the contents of this repository at build time are copied to the **projectDir/layers/oe-core/meta** directory of the platform project. See *Directory Structure for Platform Projects* on page 43.

layers/

Contains layers required by the development environment. For additional information on layers in general, see *About Layers* on page 99.

Wind River Linux layers have the prefix **wr**, for example **wr-bsps**, which is the layer containing the kernel and file system sources as well as associated machine-related configuration files.

The **layers** directory may also contain other layers including optional products, such as the **wr-simics** layer directory.

See *Layers Included in a Standard Installation* on page 100 for a breakdown of the layers included with your installation.

scripts

Contains scripts useful for Workbench and otherwise, including a script to help in adding packages to a project (see *About the Package Importer Tool (import-package)* on page 144).

setup

Contains installation-related scripts that setup the development environment.

samples/

Sample projects that can be used in Workbench as well as from the command line. This includes the following sample projects:

clientserver

A client server application for testing communications.

hello_Linux

The classic “Hello World” application. See the *Wind River Linux Getting Started Guide: Creating and Deploying an Application* for instructions on adding this application to an existing platform project and launching it on a target for debugging.

moduledebug_userprj

A sample kernel module project that uses a user-defined makefile to pass macro values from the build specs to the existing project Makefile.

mthread

A multi-threaded application for demonstrating multi-threaded debugging and stepping into, over, and out of code threads.

penguin

An application similar to the ball program provided with Wind River Workbench, but uses a Linux penguin instead.

wrlinux/

Contains the **configure** script you use when configuring a project and a **config** directory that contains files setting default **configure** script behavior.

The **configure** script is a Wind River Linux-specific feature that greatly simplifies platform project development by creating the framework for a complete platform project image, using basic information about your target platform and file system. See [About the Configure Script](#) on page 63 and [About Configure Options](#) on page 65.

Metadata

The build system uses metadata, or “data about data”, to define all aspects of the platform project image and its applications, packages, and features.

Metadata resides in the development and build environments. From a build system perspective, metadata includes input from the following sources:

Configuration (**.conf**) files

These can include application, machine, and distribution policy-related configuration files. See [Configuration Files and Platform Projects](#) on page 33

Recipes (**.bb**) files

See [About Recipes](#) on page 109.

Classes (**.bbclass**) files.

Appends (**.bbappends**) files to existing layers and recipes. See [About Recipes](#) on page 109.

The build system uses this metadata as one source of input for platform project image creation. In the Wind River Linux development environment, other sources of input include:

- Project configuration information, such as BSP name, file system, and kernel type, entered using the `configure` command. See [Introduction](#) on page 61, and [About Building Platform Project Images](#).
- Custom layers and/or templates with their own configuration, recipes, classes, and append files. [About Layers](#) on page 99.
- Additional changes and additions that apply to the runtime file system only, and not the platform project image. See [Options for Adding an Application to a Platform Project Image](#) on page 138.

It is important to organize your metadata in a manner that lets you easily create, modify, and append to it. It is also important to understand what you are already working with so that you can leverage existing metadata and reuse it as necessary.

Since metadata is included in over 800 existing recipes, knowing how the existing data impacts your platform project build will help you understand what you already have. With this knowledge, you are better prepared to plan the use of append (`.bbappend`) files to extend or modify the capability, and only create new recipes when necessary.

The idea is to avoid overlaying entire recipes from other layers in your existing configuration, and not simply copy the entire recipe into your layer and modify it.

For additional information on using append files, see *The Yocto Project Development Manual: Using .bbappend Files*:

<http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html#using-bbappend-files>

See also:

- [Configuration Files and Platform Projects](#) on page 33
- [Creating a New Layer](#) on page 106
- [Creating a Recipe File](#) on page 112

Configuration Files and Platform Projects

Learn about the various `.conf` files that define specific aspects of your platform project build.

The `bblayers.conf` File

Each platform project has a `projectDir/bitbake_build/conf/bblayers.conf` file. This file provides a sequential list of what layers to include when building a platform project image. What makes this file unique, is how it simplifies the task of including or excluding layers. To include a layer, simply add the layer path to this file and save the file. To exclude a layer, remove the layer path and save the file.

See [Enabling a Layer](#) on page 107 and [Disabling a Layer](#) on page 108 for details on modifying your `bblayers.conf` file.

The following is an example of a `bblayers.conf` file used to create the platform project image from the *Wind River Linux Getting Started Guide*:

```
LCONF_VERSION = "6"

BBPATH = "${TOPDIR}"
BBFILES ?= ""
WRL_TOP_BUILD_DIR ?= "${TOPDIR}/.."
# resolve WRL_TOP_BUILD_DIR immediately with a canonical path
```

```
# to satisfy the bitbake logger
WRL_TOP_BUILD_DIR := "${@os.path.realpath(d.getVar('WRL_TOP_BUILD_DIR', True))}"

BBLAYERS = " \
  ${WRL_TOP_BUILD_DIR}/layers/wrlinux \
  ${WRL_TOP_BUILD_DIR}/layers/wrlcompat \
  ${WRL_TOP_BUILD_DIR}/layers/wr-toolchain \
  ${WRL_TOP_BUILD_DIR}/layers/oe-core/meta \
  ${WRL_TOP_BUILD_DIR}/layers/oe-core-dl \
  ${WRL_TOP_BUILD_DIR}/layers/meta-downloads \
  ${WRL_TOP_BUILD_DIR}/layers/wr-kernel \
  ${WRL_TOP_BUILD_DIR}/layers/wr-bsps/qemux86-64 \
  ${WRL_TOP_BUILD_DIR}/layers/wr-base \
  ${WRL_TOP_BUILD_DIR}/layers/wr-features \
  ${WRL_TOP_BUILD_DIR}/layers/wr-tools-profile \
  ${WRL_TOP_BUILD_DIR}/layers/wr-tools-debug \
  ${WRL_TOP_BUILD_DIR}/layers/meta-networking \
  ${WRL_TOP_BUILD_DIR}/layers/meta-webserver \
  ${WRL_TOP_BUILD_DIR}/layers/wr-prebuilts \
  ${WRL_TOP_BUILD_DIR}/layers/local \
"
```

Note that layers are processed from top to bottom in the **bblayers.conf** file.

The local.conf File

Each platform project directory has a *projectDir/local.conf* file. This file defines many aspects of the build environment, and the intended target architecture and file system.

The contents of this file include:

- A copy of the **configure** script command used to configure and build the platform project image.
- Machine, image, and kernel type definitions
- Distribution, networking and make variables

The following is an example of a **local.conf** file used to create the platform project image from the *Wind River Linux Getting Started Guide*:

```
# File originally generated by configure:
# /home/revo/WindRiver/wrlinux-5/wrlinux/configure --enable-board=qemux86-64 --enable-
rootfs=glibc_small --enable-rm-oldimags --with-template=feature/debug --enable-jobs=3 --
enable-parallel-pkgbuilds=3 --enable-reconfig --with-rcpl-version=0006

CONF_VERSION = "1"

#
# Distribution choice. Normally "wrlinux" but can be customized by
# the user with --with-custom-distro=<name>
#
DISTRO = "wrlinux"
WRLINUX_RCPLVERSION = ".6"

DL_DIR = "${WRL_TOP_BUILD_DIR}/bitbake_build/downloads"
#
# Parallelism Options
# These two options control how much parallelism BitBake should use. The first
# option determines how many tasks bitbake should run in parallel:
#
BB_NUMBER_THREADS ?= "3"
#
#
# The second option controls how many processes make should run in parallel when
# running compile tasks:
#
PARALLEL_MAKE ?= "-j 3"
#
# For a quad-core machine, BB_NUMBER_THREADS = 4, PARALLEL_MAKE = -j 4 would
# be appropriate for example.

MACHINE = "qemux86-64"
```

```
# Set default machine to select in the hob interface
HOB_MACHINE = "qemux86-64"
DEFAULT_IMAGE = "wrlinux-image-glibc-small"
LINUX_KERNEL_TYPE = "standard"

# Log file format configuration
BB_LOGFMT = "{task}/log.{task}.{pid}"
BB_RUNFMT = "{task}/run.{taskfunc}.{pid}"

# External cache directory for sstate
#SSTATE_DIR = "/home/revo/SSTATE_CACHE"

# Shared-state files from other locations
#SSTATE_MIRRORS ?= "\
#   file://.* http://someserver.tld/share/sstate/PATH \n \
#   file://.* file:///some/local/dir/sstate/PATH"

# Activate or de-activate CCACHE settings with CCACHE_DIR
#CCACHE_DIR = "/home/revo/Builds/test/ccache"
CCACHE_DISABLE = "1"
BB_HASHBASE_WHITELIST_append += "CCACHE_DISABLE"

PREFERRED_PROVIDER_virtual/kernel_qemux86-64 = "linux-windriver"
KTYPE_ENABLED = "standard"
# Uncomment the following line if you want to build an SDK
# that will on a 32 bit host when your host is 64 bit.
# against the native host compiler
#SDKMACHINE = "i686"
# Use the rpm package class by default, you can specify multiple
# package classes in the list
PACKAGE_CLASSES ?= "package_rpm"

# Enable empty root password
EXTRA_IMAGE_FEATURES += "debug-tweaks"

BBINCLUDELOGS = "yes"

# Enable wrlinux compatibility
# NOTE: WRL_TOP_BUILD_DIR is defined in bblayers.conf
INHERIT += "wrlcompat"
INHERIT += "wrlquiltprep"
INHERIT += "save_native_sstate"

# Additional image features
#
# The following is a list of additional classes to use when building
# images which enable extra features. Some available options which can
# be included in this variable are:
# - 'buildstats' collect build statistics
# - 'image-mklibs' to reduce shared library files size for an image
# - 'image-prelink' in order to prelink the filesystem image
# - 'image-swab' to perform host system intrusion detection
# NOTE: if listing mklibs & prelink both, then make sure mklibs is
#       before prelink
# NOTE: mklibs also needs to be explicitly enabled for a given image,
#       see local.conf.extended
#
# The mklibs library size optimization is more useful to smaller images,
# and less useful for bigger images. Also mklibs library optimization
# can break the ABI compatibility, so should not be applied to the
# images which are to be extended or upgraded later.
#
#This enabled mklibs library size optimization just for the specified image.
# example: MKLIBS_OPTIMIZED_IMAGES ?= "wrlinux-image-glibc-small"
#This enable mklibs library size optimization will be for all the images.
# example: MKLIBS_OPTIMIZED_IMAGES ?= "all"
#
##-- To turn on build stats uncomment the next line --##
#U_CLASSES += "buildstats"
##-- To turn on image-mklibs uncomment the following 2 lines --##
#MKLIBS_OPTIMIZED_IMAGES ?= "all"
#U_CLASSES += "image-mklibs"
##-- To turn on prelink uncomment the next next --##
U_CLASSES += "image-prelink"
##-- Simulator export variable class --##
```

```
U_CLASSES += "image-export-vars"
##-- Setup USER_CLASSES based on values above --##
USER_CLASSES ?= "${U_CLASSES}"

#
# SDK image filesystem normalization.
# If the SDK filesystem needs to be canonicalized (as in Win32/64 where case is
# insignificant and there are no symlinks) set NORMALIZE_SDK_FS to the type of
# normalization desired (or 'no' if none)
# Currently available normalizations:
# - winfs
NORMALIZE_SDK_FS ?= "no"

#
# Windows SDK ancilliary settings. If EXPORT_SYSROOT_HOSTS (named for backwards
# compatability with WB) contains a space-seperated token 'x86-win32', a second
# SDK archive will be constructed by normalizing the Linux SDK for Win32 as
# described above, and will be augmented with Win32 toolchain binaries.
# EXPORT_SYSROOT_HOSTS is expected to be set as an environment variable by WB.
# The variable TOOLCHAIN_WIN32_DIR can be set to point to the base directory
# of the Windows win32 toolchain directory hierarchy
# (e.g. ${WRL_TOP_BUILD_DIR}/layers/wr-toolchain/<vvvv-build>-other)
# if it is not installed as a peer to the normal toolchain layer
# TOOLCHAIN_WIN32_DIR is not currently imported from environment variables.
EXPORT_SYSROOT_HOSTS ?= "x86-linux2 "
#TOOLCHAIN_WIN32_DIR ?= ""

#Install the documentation pages on the target system
#EXTRA_IMAGE_FEATURES += "doc-pkgs"

#Install all staticdev-pkgs to SDK image
#SDKIMAGE_FEATURES += "staticdev-pkgs"

# incrementally erase temporary objects if built successfully
#INHERIT += "rm_work"

#Location of the bitbake_build/tmp directory
#TMPDIR ?= "${TOPDIR}/tmp"

# Control patch resolution process
PATCHRESOLVE = "user"

# Disable network access for the fetcher
BB_NO_NETWORK ?= "1"
#
ENABLE_BINARY_LOCALE_GENERATION = ""
GLIBC_INTERNAL_USE_BINARY_LOCALE = "precompiled"
# Enable debugging for all packages
#SELECTED_OPTIMIZATION = "${DEBUG_OPTIMIZATION}"
# Enable profiling for all packages
#SELECTED_OPTIMIZATION = "${PROFILING_OPTIMIZATION}"
# Use SELECTED_OPTIMIZATION <name> = "<flags>" for individual recipes
# Install the debug info packages on the target system
#EXTRA_IMAGE_FEATURES += "dbg-pkgs"
# Strip and split packages into separate debug info files
#INHIBIT_PACKAGE_DEBUG_SPLIT = "1"
#INHIBIT_PACKAGE_STRIP = "1"

## Syslinux options for boot menus when using isolinux or linux live
SYSLINUX_LABELS = "boot"
SYSLINUX_TIMEOUT = "0"
SYSLINUX_SPLASH = "${WRL_TOP_BUILD_DIR}/layers/wrlcompat/data/syslinux/splash.lss"
AUTO_SYSLINUXMENU = "1"

## File system boot image types
## iso=live hdd=ext3
IMAGE_FSTYPES += "tar.bz2"
#IMAGE_FSTYPES += "tar.gz"
#IMAGE_FSTYPES += "live"
NOISO = "1"
#IMAGE_FSTYPES += "ext3"
NOHDD = "1"
#IMAGE_FSTYPES += "jffs2"
#IMAGE_FSTYPES += "ubifs"
#MKUBIFS_ARGS ?= "-m 2048 -e 129024 -c 1996"
#IMAGE_FSTYPES += "cpio.gz"
```

```
# Specify the number of extra blocks of free space for an hdd image
# BOOTIMG_EXTRA_SPACE ?= "512"

# add/disable licenses
#
#LICENSE_FLAGS_WHITELIST += ""
#INCOMPATIBLE_LICENSE = ""

# The extra-addpkg.conf is used by make -C build PKG.addpkg
include extra-addpkg.conf

## Included feature templates
require ${WRL_TOP_BUILD_DIR}/layers/wr-kernel/templates/default/template.conf
require ${WRL_TOP_BUILD_DIR}/layers/wr-base/templates/default/template.conf
require ${WRL_TOP_BUILD_DIR}/layers/wr-tools-debug/templates/default/template.conf
```

The layer.conf Files

Each layer has a **layerDir/conf/layer.conf** file that the BitBake build system uses to process the layer on project configuration and build. This file is required to include the layer in the project build.

The following example is taken from the **layer.conf** file in the **projectDir/layers/local** layer directory ([About the layers/local Directory](#) on page 54) that is generated when you build a platform project.

```
# We have a conf and classes directory, add to BBPATH
BBPATH := "${LAYERDIR}:${BBPATH}"

# We have a packages directory, add to BBFILES
BBFILES := "${BBFILES} ${LAYERDIR}/recipes-*/*/*.bb \
           ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "local"
BBFILE_PATTERN_local := "^${LAYERDIR}/"
BBFILE_PRIORITY_local = "10"

# Add scripts to PATH
PATH := "${PATH}:${LAYERDIR}/scripts"

# Add a directory to allow local changelist.xml changes
WRL_CHANGE_LIST_PATH += "${LAYERDIR}/conf/image_final"

# Add a directory to allow local fs_final*.sh script changes
WRL_FS_FINAL_PATH += "${LAYERDIR}/conf/image_final"

# We have a pre-populated downloads directory, add to PREMIRRORS
PREMIRRORS_append := "\
git://.* file://${LAYERDIR}/downloads/ \n \
git://.* git://${LAYERDIR}/git/BASENAME;protocol=file \n \
svn://.* file://${LAYERDIR}/downloads/ \n \
ftp://.* file://${LAYERDIR}/downloads/ \n \
http://.* file://${LAYERDIR}/downloads/ \n \
https://.* file://${LAYERDIR}/downloads/ \n"
```

In general there is no need to modify this file since it incorporates everything that is needed to process your local application recipes. Of particular interest are the following variables:

BBFILES

This variable tells the build system to source recipe files (**.bb** and **.bbappend** files) from any directory named **recipes-*** inside the layer. In the default layer structure this would be the directory **recipes-sample**. As long as you place your application-specific recipes in a folder that begins with **recipes-**, they will be included in the build as defined by this variable.

BBFILE_PRIORITY

This variable sets the processing priority of the layer. If another layer depends on the configuration of this layer, or on recipes contained in this layer, then the priority should be higher than the dependant layer.

PREMIRRORS_append

This variable specifies places where the build process can source your upstream package sources from before the recipe specified URL is searched. Wind River Linux comes with all supported packages in a pre-mirror or local file path URL location so it is not necessary to access the internet in order to build a project. In particular, the source files can be retrieved automatically from a source code repository, either git or subversion as indicated.

The machine.conf Files

The **machine.conf** file, located in the layer at *layerName/conf/machine/machineName.conf* (see [Layer Structure by Layer Type](#) on page 103), specifies the BSP configuration for your platform project image.

Assigning Empty Values in BitBake Configuration Files

Learn how to assign an empty value to a variable in the Yocto Project BitBake build system.

Assigning an empty value in a platform project **.conf** file is different in BitBake than it is from conventional syntax. With BitBake, you must enter a space between the declaration's double quotes, while conventional syntax allows for double quotes with no spaces.

Assign Empty String

```
CONF_VALUE = " "
```



NOTE: Notice that the = " " declaration has a single space.

This assigns the empty string after BitBake strips the leading space. To retrieve this variable as part of your code and prevent **undefined** variable exceptions, use the following syntax:

```
pyvar = d.getVar("CONF_VALUE", True) or ""
```

Assign No Value

```
CONF_VALUE = ""
```

This assigns the empty string to the Python **None** value, and does not return an empty string when you try to retrieve the variable's value.

Perform the following steps to insert an empty string and test that it returns an empty value.

Step 1 Assign an empty string to the *projectDir/local.conf* file.

Options	Description
Command line	Run the following command from the <i>projectDir</i> :

```
$ echo VIRTUAL-RUNTIME_dev_manager = \" \" >> local.conf
```

Options	Description
Edit <code>local.conf</code> file	<ol style="list-style-type: none">1. Open the <code>projectDir/local.conf</code> file in an editor and add the following line to it: <pre>VIRTUAL-RUNTIME_dev_manager=" "</pre>2. Save the file.

In this example, we use the `VIRTUAL-RUNTIME_dev_manager` variable. You may substitute this for a variable that you wish to retrieve an empty string for.

Step 2 Rebuild the file system.

Run the following command from the `projectDir`.

```
$ make fs
```

Step 3 Retrieve the empty string.

Run the following command from the `projectDir/bitbake_build` directory.

```
$ bitbake -e | grep VIRTUAL-RUNTIME_dev_manager
```

If you used a different variable in the previous step, substitute `VIRTUAL-RUNTIME_dev_manager` for the name of that variable.

The system should return an output that displays the empty string, for example:

```
$ VIRTUAL-RUNTIME_dev_manager=""
```

README Files in the Development Environment

Wind River Linux provides multiple ways to view the **README** files that are part of your installation.

README files located in your installation, such as the one located in the `installDir/layers/example/lemon_layer` directory, actually reside in a git repository, and are not directly available to open to review. The following sections provide information on viewing **README** files.

Viewing a Specific README File in the Installation

Use `git show` to open the **README** file directly for viewing only.

The following procedure uses the **README** file located in the `installDir/wrlinux-5/layers/examples/lemon_layer` directory as an example. To view another **README** file, locate the path to it before you begin.

Step 1 Navigate to the location where the **README** file resides.

```
$ cd installDir/wrlinux-5/layers/examples/lemon_layer
```

Substitute the path as necessary to view other **README** files.

Step 2 View the **README** file.

```
$ git show HEAD:README
```

Cloning a Layer to View Installation README Files

Use **git clone** to clone a layer in the installation to a temporary directory to view or edit the **README** file.

The following procedure uses the **README** file located in the *installDir/wrlinux-5/layers/examples/lemon_layer* directory as an example. To clone another **README** file, locate the path to it before you begin.

Step 1 Navigate to a location on the host system to clone the layer to.

In this example, you will navigate to the **/tmp** directory.

```
$ cd /tmp
```

Substitute the path as necessary to place the layer in a different directory.

Step 2 Clone the layer.

```
$ git clone installDir/wrlinux-5/layers/examples/lemon_layer
```

Substitute the path as necessary to clone another layer to view a different **README** file.

Step 3 View the **README** file using the terminal, or open it in an editor.

Viewing All Installation README Files in a Web Browser

Use **git instaweb** to browse all layer installation files, including **README** files, in a web browser.

The following procedure uses a web browser to view the files located in git in the *installDir/layers* directory.



NOTE: To use this option, you must install and run **lighttpd** or **httpd** on your development host prior to performing the procedure.

Step 1 Navigate to the location of the layers.

```
$ cd installDir/wrlinux-5/layers
```

Step 2 Open the **git** web browser.

```
$ git instaweb
```

The web browser will open with the *installDir/layers* directory at the top-level.

Step 3 Navigate to the *examples/layers/lemon_layer* location and open the **README** file in the tree view.

4

Build Environment

[About the Project Directory](#) 41

[Creating a Project Directory](#) 42

[Directory Structure for Platform Projects](#) 43

[About README Files in the Build Environment](#) 56

About the Project Directory

The project directory, located in your build environment, maintains all files specific to your platform project development.

You should keep your build environment separate from the development environment. Wind River recommends you create a separate work directory with a project subdirectory holding the build environment. This concept is explained in the *Wind River Linux Getting Started Guide: About Developing Platform Projects*.

The main reason for this is that it is possible to corrupt your development environment by running the configure script from inside it (see [About the Configure Script](#) on page 63).



CAUTION: Running configure from the Wind River Linux install directory may corrupt your installation. Always run configure from the directory where your project resides.

The following figure shows one example of this directory structure.

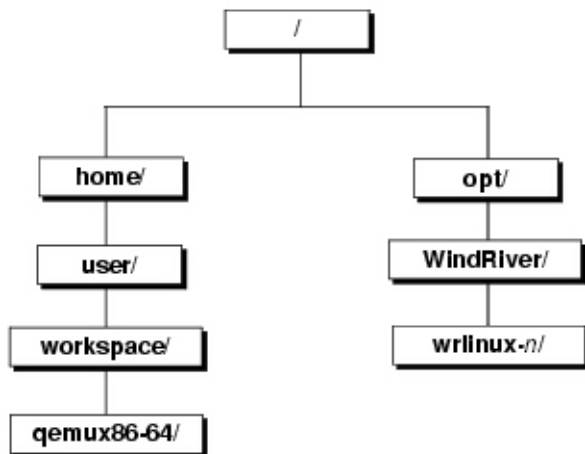


Figure 2: qemux86-64 as the Project Directory Containing the Build Environment

In this example, the general work directory is named **workspace**. Within **workspace** is the **qemux86-64** project directory which will hold the build environment, in this case for a common PC board. Directory names have been chosen for clarity; you can name them as you like. In this document, the variable **projectDir** refers to your project directory, which is **qemux86-64** in the example in the figure.



NOTE: When using Workbench to create a platform project, by default Workbench creates the **\$HOME/WindRiver/workspace/projectName** directory containing the Workbench specific project files, and an additional adjacent directory with a **_prj** suffix that contains a complete Wind River Linux platform project. The Workbench project directory contains select file system links to elements in the Linux platform project_prj project. For the example shown in the figure above, the Workbench project directory would be in

/home/user/WindRiver/workspace/qemux86-64_prj

Creating a Project Directory

A project directory or folder is the file system location where you configure and build your target software.

You can create a project directory in any location for which you have permission; for example, in a subdirectory of your home directory.

- Create a new directory for your project.

Options	Description
Command line	Create a new project directory with the mkdir command, for example:

```
$ mkdir -p $HOME/workspace/qemux86-64
```

Options	Description
Workbench	If you are using Workbench, the project directory is created for you when you click Finish in the configuration wizard. It is created as a folder in your workspace folder (by default, <i>\$HOME/WindRiver/workspace/</i>) with the name you assign and a _prj suffix, for example my_qemux86-64_prj

Directory Structure for Platform Projects

Wind River Linux automatically creates the build environment directory structure when you configure and build a platform project.

The following illustration depicts some of the subdirectories the **configure** script creates within a project directory.

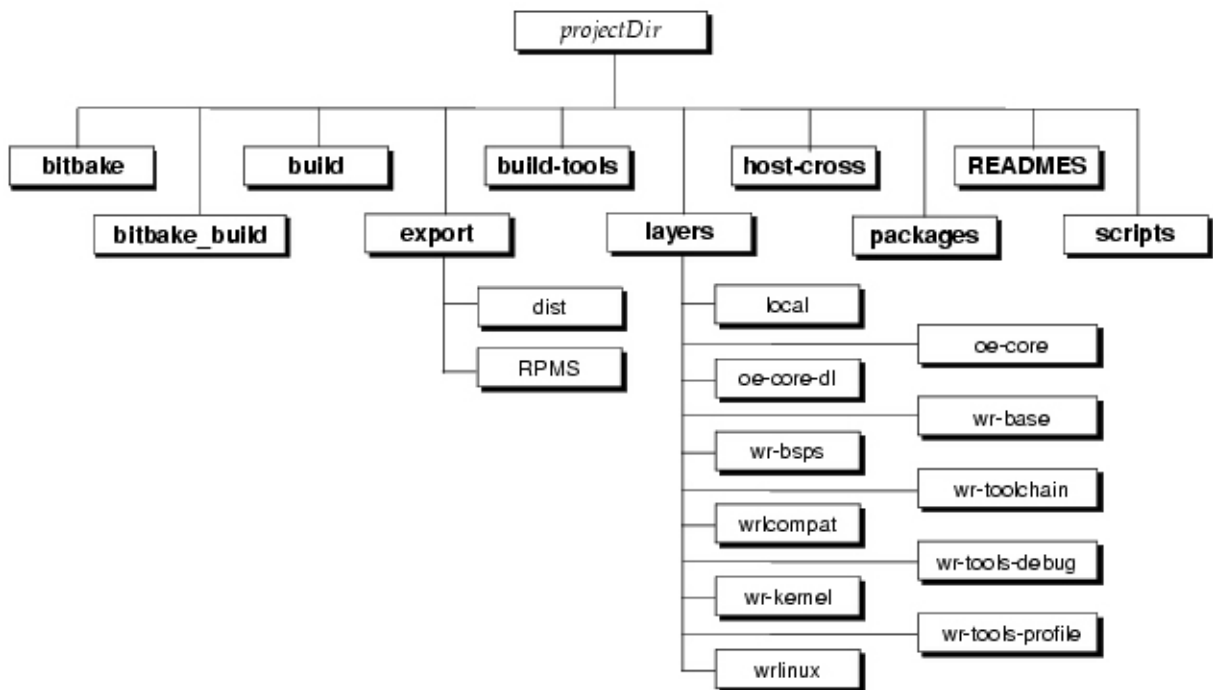


Figure 3: Partial Contents of the Project Directory

Selected directories and their contents are described in further detail below. Note that these directories are not present until you configure and build the platform project:

bitbake_build

The Yocto build systems working directory, many of the other directories contain file links to elements in this directory.

bitbake_build/conf

The directory where the **local.conf** build configuration file is located. A soft link to this file is created in the working directory for your convenience.

bitbake_build/conf/bblayers.conf

A file listing the layers used to build the target images.

build

A directory that hosts soft links to the build directories for each package. The directory also hosts a Makefile used to build special targets associated with the packages. Use the command **make -C build help** for more details.

export

A convenience directory that hosts soft links to important build files, and the target filesystem once it is built using the **make** command. Note that the name of the files are generated after the project's configuration options.

export/qemux86-64-glibc-std-standard-dist.tar.bz2

A compressed tar file of the root file system.

export/dist

A directory containing the target's root file system. It is used by default as an NFS mount when booting the Linux kernel image using QEMU.

export/images/bzImage-qemux86-64.bin

The Linux kernel image.

export/images/modules-version-WR5.0.1.0_standard-r0-qemux86-64.tgz

A compressed tar file containing the Linux kernel modules.

host-cross

The build tool chain that you can use to (cross-) compile programs for your target. Note that the directories inside host-cross are soft links to corresponding directories in

bitbake_build/tmp/sysroots

layers

A directory that contains the layers for the platform project, including:

layers/local

A folder created specifically for developers to hold their project-specific files. For example, if you add a sample project to your platform project, the build system adds a directory with the required files to the **layers/local/recipes-sample** directory. See [About the layers/local Directory](#) on page 54.

layers/oe-core

A folder that contains the core Open Embedded metadata in specific BitBake layers used to configure the project. The default layer directories include:

meta

Contains the git clone from the **installDir/wrlinux-5/git** directory in the installation development environment. See [Directory Structure](#) on page 29 .

meta-demoapps

Contains recipe files for demo applications provided with the Yocto Project.

meta-hob

Contains configuration and recipe files for the HOB, a GUI-based tool for creating custom BitBake images. See <https://wiki.yoctoproject.org/wiki/BitBake/GUI/Hob> for additional information.

meta-skeleton

Contains configuration and recipes for the platform projects base structure.

scripts

Contains macros and scripts for the build system.

layers/oe-core-dl

Contains downloaded packages and configuration files that comprise the package offerings from the Yocto Project. The **conf/layer.conf** file defines the mirror sites and order of locations that packages are retrieved from.

layers/wr-base

Contains the recipes and other configuration files that comprise the Wind River Linux base offering and make it possible to use the configure script to generate a platform project image. See *About the Configure Script* on page 63.

Additionally, this layer contains templates for adding additional features to your platform project image. See *Feature Templates in the Project Directory* on page 47 for additional information.

This layer does not include bug fixes.

layers/wr-bsps

Contains the BSP support files of the BSP that your platform project is configured for.

layers/wr-kernel

Contains recipes and machine configuration information for Wind River-supplied kernels and kernel features. Additionally, this layer contains templates for adding additional features to your platform project image. See *Kernel Configuration Fragments in the Project Directory* on page 51 for additional information.

layers/wrlcompat

In a typical Yocto Project build environment, the build output creates a specific directory structure. This structure is different than the Wind River Linux structure from previous releases. The **wrlcompat** layer ensures that build output is consistent with previous Wind River Linux (4.x) releases.

layers/wrlinux

Contains the recipes, configuration information, and files that support Wind River Linux tools and enhance development with the Yocto Project build system.

The **files** directory includes licensing information.

The **scripts** directory includes the following Wind River Linux scripts that simplify and enhance platform project image creation and development:

layers/oe-core-dl

Contains downloaded packages and configuration files that comprise the package offerings from the Yocto Project. The **conf/layer.conf** file defines the mirror sites and order of locations that packages are retrieved from.

layers/wr-base

Contains the recipes and other configuration files that comprise the Wind River Linux base offering and make it possible to use the configure script to generate a platform project image. See [About the Configure Script](#) on page 63. Additionally, this layer contains templates for adding additional features to your platform project image. See [Feature Templates in the Project Directory](#) on page 47.

layers/wr-bsps

Contains the BSP support files of the BSP that your platform project is configured for.

layers/wr-kernel

Contains recipes and machine configuration information for Wind River-supplied kernels and kernel features. Additionally, this layer contains templates for adding additional features to your platform project image. See [Kernel Configuration Fragments in the Project Directory](#) on page 51.

layers/wrlcompat

In a typical Yocto Project build environment, the build output creates a specific directory structure. This structure is different than the Wind River Linux structure from previous releases. The wrlcompat layer ensures that build output is consistent with previous Wind River Linux (4.x) releases.

layers/wrlinux

Contains the recipes, configuration information, and files that support Wind River Linux tools and enhance development with the Yocto Project build system.

The **files** directory includes licensing information.

The **scripts** directory includes the following Wind River Linux scripts that simplify and enhance platform project image creation and development:

config-target.pl

Configures the target platform project from configure scripts input.

create-usb.pl

Creates a platform project image file suitable for deployment on a USB drive.

fs_changelist.lua

Creates a file system changelist file. See [Lua Scripting in Spec Files](#) on page 369.

rsim

Launches a QEMU session from the command-line with a single make start-target command.

layers/wr-toolchain

Contains files, recipes, configuration information and documentation to support the GNU toolchain supplied by Wind River for development.

layers/wr-toolchain-shim

Provides configuration glue to allow selection of an automatically integrated toolchain layer, which in turn contains both rules for building the toolchain from source, and rules for using the pre-built binaries. This layer also provides tuning files and configuration overrides for those layers.

layers/wr-tools-debug

Contains configuration information and files to support debugging and ptrace with Workbench and Wind River tools. Additionally, this layer contains templates for adding additional features to your platform project image. See [Feature Templates in the Project Directory](#) on page 47.

layers/wr-tools-profile

Contains configuration information and files to support Wind River Linux development tools, including: analysis, boot time, code coverage, Valgrind, and lttng. You can add these features to your platform project using templates. See [Feature Templates in the Project Directory](#) on page 47.

packages

A soft link to **bitbake_build/downloads**, a directory where you can find tar files of all packages used in the build. Note that the tar files are soft links to the actual files in the Wind River Linux installation path that are bundled with the distribution.

READMEs

Contains **README** files for the Wind River Linux layers included in your platform project image configuration and build.

scripts

Contains macros and scripts for the build system.

Feature Templates in the Project Directory

Each feature template consists of configuration files that add the required system settings and packages necessary to add the feature to your platform project.

These templates can also include kernel or general file system changes that the respective feature may require.

In the development environment for the installation, templates are located in the *projectDir/layers* directory, in the following subdirectories:

Table 1 **Build Environment Template Sub-directories**

<i>projectDir</i> /layers sub-directory	Sub-directory contents
<i>wr-base/templates/feature/</i>	benchmark Adds benchmark packages.
	debug Adds debug-specific functionality, including the following tools: elfutils , ltrace , memstat , strace , and the Wind River LTTng trace daemon.
	demo Adds general purpose functionality for testing small file system (glibc_small) target platforms.
	gdb Adds the GNU debugger for command-line debugging.

<i>projectDir</i> /layers sub-directory	Sub-directory contents
	test Adds a collection of tests from the Linux Test Project (LTP) package.
wr-features/templates/feature/	ipv6 Configuration file for enabling ipv6. LAMP Appends the following images to the build: mysql5 , modphp , phpmyadmin , apache2 , and xdebug lsbtesting Adds recipes, tasks, images, and missing packages, as necessary, for running the Linux Standard Base (LSB) tests on your target platform. mysql Adds the mysql5 package. mysql-odbc Adds packages necessary to use MySQL over ODBC.
wr-kernel/templates/feature/	compat-wireless Provides a mechanism to build a compat-wireless out-of-tree kernel module then install it on the target. edac Configuration file for edac initramfs Configures the build to create an initramfs image. initramfs-integrated Configures the build to create an initramfs image that will be integrated with the kernel image. kdump Configuration file to enable kdump kernel-debug Configuration file to enable kernel debugging kernel-tune Configuration file to add necessary user land tools kexec Configuration file to enable kernel execution of a new kernel over the currently running kernel. kvm

<i>projectDir</i> /layers sub-directory	Sub-directory contents
	Configuration file to enable kernel-based virtual machine (KVM)
	kvm-kmod Provides a mechanism to build the out-of-tree KVM kernel modules.
	libhugetlbfs Enables Huge TLB support in the kernel and adds the libhugetlbfs package to the image.
	ltnng2 Configuration file to configure the kernel for ltnng2 and disable ltn .
	msa Enables the Microstate Accounting kernel feature.
	nfsd Provides a mechanism to build the NSFD kernel module for NFS server capability on the target.
	qemu-linaro Makes the qemu-linaro.bb file available. In particular, the qemu-linaro.bb file will be qemu-native once it is available.
	qorIQ-debug Enables the qorIQ-debug kernel in the root file system, which allows access to the hardware Debug IP of the QorIQ P4080, P3041, and P5020 devices from user-space applications.
wr-toolchain/templates/feature	target-toolchain Appends toolchain wrappers and symlinks to the Wind River Linux build
wr-tools-debug/templates/feature/	debug-wb Adds required target agents to support command line and Workbench debugging.
	libwrsqtehelper Adds a library of Qt object wrapper functions that can help display Qt objects that are otherwise opaque when used with Wind River Workbench.
wr-tools-profile/templates/feature/	analysis Adds support for Wind River Analysis tools, including interfaces to Wind River Workbench for performance profiling and memory analysis.
	boottime

<i>projectDir</i> /layers sub-directory	Sub-directory contents
	Adds boot time profiling tools.
	code_coverage Enables the Code Coverage agent for use with the Wind River Code Coverage Analyzer development tool.
	debug-python Adds Wind River's python development solution to allow for debugging of python scripts through the PyDev plugin.
	footprint Adds the fetch-footprint.sh footprint utility to the target file system to enable analysis of file system footprint..
	oprofile Add support for oprofile system-wide profiling.
	system-stats Adds the system and process monitoring utilities.
	valgrind Adds the Valgrind instrumentation framework.
	valgrind_small Adds the small footprint Valgrind instrumentation framework.
	wr-profiling-kernel-overrides Configuration files and .scc file to ensure profiling tools can access the stack on CGL systems.
	wrsv-ltt Adds a Wind River Linux implementation of the Linux Trace Toolkit (ltt).
<i>wrlcompat/templates/feature/</i>	export-tar Adds a step to recipe building that creates tarballs of SVN and git repositories, and copies over tarballs to a common directory (tmp/depoy/tar) to be used or copied over to other directories.
	ip-report Adds an IP report to the BitBake parser environment.
	package-report Adds a package report to the BitBake parser environment
	pseudo14 Sets the preferred pseudo version to 1.4.5
	pseudo15

<i>projectDir</i> /layers sub-directory	Sub-directory contents
	Sets the preferred pseudo version to 1.5
	pseudo151
	Sets the preferred pseudo version to 1.5.1

Kernel Configuration Fragments in the Project Directory

Each kernel configuration fragment includes a list of kernel feature descriptions that encapsulate a change to the kernel tree, depending on the fragment's application.

Kernel tree changes consist of patches, or configuration fragments, that are applied to the branch, or board/target-specific system kernel being built.

Currently, Wind River Linux 5.0.1 offers the following kernel configuration fragments:

Table 2 **Wind River Linux Kernel Configuration Fragments**

Configuration Fragment Name	Description
<code>cfg/l2tp.scc</code>	Layer 2 tunneling protocol support
<code>cfg/8250.scc</code>	Enable 8250 serial support
<code>cfg/boot-live.scc</code>	Live boot support
<code>cfg/cpu-hotplug.scc</code>	Enable CPU hotplug support
<code>cfg/dmaengine.scc</code>	Enable DMA engine core functionality
<code>cfg/dmm.scc</code>	Device mapper multipath support
<code>cfg/dpaa.scc</code>	Enable DPAA support
<code>cfg/drbd.scc</code>	DRDB block device support
<code>cfg/efi-ext.scc</code>	Enable extended EFI support
<code>cfg/efi.scc</code>	Core EFI support
<code>cfg/fs/btrfs.scc</code>	Enable btrfs filesystem support
<code>cfg/fs/debugfs.scc</code>	Enable debugfs support
<code>cfg/fs/devtmpfs.scc</code>	Enable devtmpfs for tmpfs/ramfs support at early bootup
<code>cfg/fs/ext2.scc</code>	Enable the Extended 2 (ext2) filesystem
<code>cfg/fs/ext3.scc</code>	Enable the Extended 3 (ext3) filesystem
<code>cfg/fs/ext4.scc</code>	Enable the Extended 4 (ext4) filesystem
<code>cfg/fs/flash_fs.scc</code>	Enable flash filesystem support (yaffs, jffs2, cramfs, mtd, etc.)
<code>cfg/fs/ocfs2.scc</code>	OCFS2 file system support

Configuration Fragment Name	Description
<code>cfg/fs/vfat.scc</code>	Enable VFAT support
<code>cfg/iscsi.scc</code>	iSCSI initiator over TCP/IP
<code>cfg/macvlan.scc</code>	MAC-VLAN support
<code>cfg/net/bridge.scc</code>	Enable Bridge Netfilter options
<code>cfg/net/ip6_nf.scc</code>	Enable Netfilter (IPv6) options
<code>cfg/net/ip_nf.scc</code>	Enable Netfilter (IPv4) options
<code>cfg/net/ipsec.scc</code>	Enable IPsec options
<code>cfg/net/ipsec6.scc</code>	Enable IPv6 IPsec options
<code>cfg/net/ipv6.scc</code>	Enable IPv6 options
<code>cfg/paravirt_kvm.scc</code>	Paravirtualized KVM guest support
<code>cfg/ptp-gianfar.scc</code>	Enable PTP 1588 using Gianfar support
<code>cfg/qemu-devices.scc</code>	Enable QEMU-supported devices
<code>cfg/rt-mutex-tester.scc</code>	Scriptable tester for rt mutexes
<code>cfg/smp.scc</code>	Enable SMP
<code>cfg/sound.scc</code>	OSS sound support
<code>cfg/timer/hpet.scc</code>	HPET timer support
<code>cfg/timer/hz_100.scc</code>	Enable 100Hz timer frequency
<code>cfg/timer/hz_250.scc</code>	Enable 250Hz timer frequency
<code>cfg/timer/hz_1000.scc</code>	Enable 1000Hz timer frequency
<code>cfg/timer/no_hz.scc</code>	Enable CONFIG_NO_HZ
<code>cfg/usb-mass-storage.scc</code>	Enable options required for USB mass storage devices
<code>cfg/vesafb.scc</code>	VESA framebuffer support
<code>cfg/virtio.scc</code>	virtio support (core, pci, balloon, ring, net, blk, mmio)
<code>cfg/x32.scc</code>	x86 x32 support
<code>features/aoe/aoe-enable.scc</code>	Enable ATA Over Ethernet (AOE)
<code>features/blktrace/blktrace.scc</code>	Enable blktrace
<code>features/cgroups/cgroups.scc</code>	Enable cgroups and selected controllers / namespaces and associated functionality
<code>features/dca/dca.scc</code>	Enable DCA for IOATDMA-capable devices
<code>features/edac/edac.scc</code>	Enable core EDAC functionality

Configuration Fragment Name	Description
<code>features/ftrace/ftrace.scc</code>	Enable Function Tracer
<code>features/fuse/fuse.scc</code>	Enable core FUSE functionality
<code>features/hrt/hrt.scc</code>	Enable high res timers and Generic Time
<code>features/hugetlb/hugetlb.scc</code>	Enable Huge TLB support
<code>features/i915/i915.scc</code>	Enable i915 driver
<code>features/igb/igb.scc</code>	Intel gigabit functionality
<code>features/intel-amt/mei/mei.scc</code>	Enable options for the Intel Management Engine interface
<code>features/intel-dpdk/intel-dpdk.scc</code>	Enable prerequisites for Intel DPDK
<code>features/intel-e1xxx/intel-e100.scc</code>	Enable Intel E100 and E1000 support
<code>features/ipmi/ipmi.scc</code>	Enable core ipmi support
<code>features/iwlgan/iwlgan.scc</code>	Enable iwlgan support
<code>features/iwlwifi/iwlwifi.scc</code>	Enable iwlwifi support
<code>features/kgdb/kgdb.scc</code>	Enable KGDB and KGDB access protocols
<code>features/kmemcheck/kmemcheck-enable.scc</code>	Enable kmemcheck
<code>features/kvm/qemu-kvm-enable.scc</code>	Enable KVM host support
<code>features/latencytop/latencytop.scc</code>	Enable latencytop
<code>features/lttng/lttng-enable.scc</code>	Enable Linux Trace Toolkit - next generation
<code>features/lttng2/lttng2-enable.scc</code>	Enable LTTNG 2
<code>features/mac80211/mac80211.scc</code>	Enable mac 80211 and WLAN support
<code>features/msa/msa-enable.scc</code>	Enable Microstate Accounting (MSA)
<code>features/namespaces/namespaces.scc</code>	Enable namespace support and experimental namespaces
<code>features/netfilter/netfilter.scc</code>	Enable netfilter and conn tracking
<code>features/nfsd/nfsd-enable.scc</code>	Enable NFS server support
<code>features/power/intel.scc</code>	Enable Intel Power Management options
<code>features/powertop/powertop.scc</code>	Enable powertop and profiling
<code>features/profiling/profiling.scc</code>	Enable profiling and timerstats
<code>features/ramconsole/ramconsole.scc</code>	Android RAM buffer console
<code>features/scsi/cdrom.scc</code>	Enable options for SCSI CD-ROM support
<code>features/scsi/disk.scc</code>	Enable options for SCSI disk support

Configuration Fragment Name	Description
<code>features/scsi/scsi.scc</code>	Enable options for SCSI support
<code>features/serial/8250.scc</code>	Enable 8250 serial support
<code>features/systemtap/systemtap.scc</code>	Enable options required for systemtap support
<code>features/taskstats/taskstats.scc</code>	Enable taskstats
<code>features/uio/uio.scc</code>	Enable UIO as a module
<code>features/uprobe/uprobe-enable.scc</code>	Enable options required for uprobes support
<code>features/usb-net/usb-net.scc</code>	Enable all options required for USB networking
<code>features/usb-base.scc</code>	Enable core options for USB support
<code>features/usb/ehci-hcd.scc</code>	Enable options for ehci (USB 2.0)
<code>features/usb/ohci-hcd.scc</code>	Enable options for ohci (USB 1.x)
<code>features/usb/uhci-hcd.scc</code>	Enable options for uhci (USB 1.x)
<code>features/usb/xhci-hcd.scc</code>	Enable options for xhci (USB 3.0)
<code>features/userstack/userstack.scc</code>	Enable User Space Stack Dump support
<code>features/wrnote/wrnote.scc</code>	wrnote for advanced debugging

Viewing Template Descriptions

To view a list of available template descriptions, use the Workbench Platform Project wizard.

See [About Templates](#) on page 115 for information on adding templates to your platform project image.

Step 1 Start Workbench.

Step 2 Create a new project.

In Project Explorer, right-click and select **New > Wind River Linux Platform Project**.

Step 3 Give the project a name.

Enter a project name, then click **Next**.

Step 4 Apply a template.

Click **Add** to the right of the **Templates** field. A list of available templates will display. Select a template in the list to view its description.

About the layers/local Directory

When you configure a platform project image, Wind River Linux automatically creates a sample local layer for use as a location to contain your application projects.

The local layer lives in the **layers/local** directory and is created with the following structure:

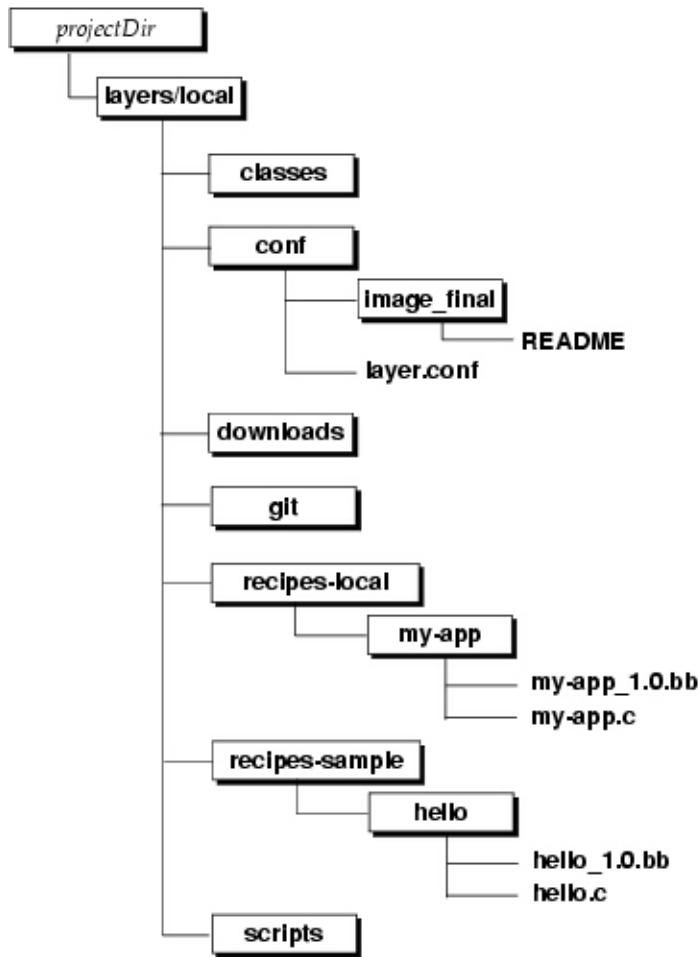


Figure 4: Local Layer Directory Structure with Sample Hello World Application

The `layers/local` also contains the final recipe processed by the build system `layers/local/recipes-img/images/wrlinux-image-filesystem.bb`. A link to this file called `default-image.bb` appears in the root directory of the project. This file and the your `local.conf` file form the basis of your project wide configuration.

In this example, the `projectDir/layers/local` directory also includes the `recipes-sample/hello` subdirectory, which is added to the local directory when you add the sample Hello World application project to your platform project image. See the *Wind River Linux Getting Started Guide: Developing an Application Project Using the Command Line*.

When you add a sample application, the contents are automatically added to the `layers/local/recipes-sample` directory.

When you import an application package using the `import-package` feature, the contents are added to the `layers/local/recipes-local` directory. See [About the Package Importer Tool \(`import-package`\)](#) on page 144.

The local layer is enabled by default. You can verify this by looking at the contents of the `projectDir/bitbake_build/conf/bblayers.conf` file. The main components of the layer are the layer configuration file (`projectDir/layers/local/conf/layer.conf`), and the application-specific source code and associated metadata located in the `recipes-sample/hello` folder. Layers and layer requirements are described in the following sections:

- [Directory Structure](#) on page 29
- [About Layers](#) on page 99
- [Creating a New Layer](#) on page 106

In this example, the sample **hello** application provides a guideline for adding your own applications to an existing platform project image. To be included in a platform project, each application requires:

- It be placed in a directory that includes a BitBake recipe file (**hello_1.0.bb**) and the application source (**hello.c**). [Creating a Recipe File](#) on page 112.
- The directory (above) be placed in the **projectDir/layers/local** directory. It is possible to use a directory that you create, but if you want to include the application in subsequent project builds, that directory must set up as a layer. [Creating a New Layer](#) on page 106.
- It be built and added to the platform project image. See the *Wind River Linux Getting Started Guide: Developing an Application Project Using the Command Line* and [About Application Development](#) on page 127

You can setup your local layer in any manner that best suits your development needs.

See also:

[Directory Structure for Platform Projects](#) on page 43

[Directory Structure](#) on page 29

About README Files in the Build Environment

Wind River Linux provides two options to view the **README** files that are part of your platform project build.

README files located in your installation, such as the one located in the **installDir/layers/example/lemon_layer** directory, actually reside in a git repository, and are not directly available to open to review, even when you configure and build a platform project.

Adding a Layer to a Platform Project to View README Files

Use the **--with-layer=** configure option to add a layer to a platform project to make a **README** file available for viewing or editing.

The following procedure uses the **README** file located in the **installDir/wrlinux-5/layers/examples/lemon_layer** directory as an example. To use another **README** file from a different layer, locate the layer path before you begin.

Step 1 Configure a platform project to include the layer with the **README** file.

In this example, you will use the **--with-layer=examples/lemon_layer** configure option to add the layer.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--with-layer=examples/lemon_layer
```

Substitute the **--with-layer=** path as necessary to add a different layer to view a different **README** file.

Step 2 View the README file.

Once the **configure** script completes, the README file is available in two locations for viewing or editing:

- The *projectDir*/READMEs directory
- The *projectDir*/layers/examples/lemon_layer directory

Adding All Layers to a Platform Project to View All README Files

Use the **--enable-checkout-all-layers=yes** configure option to add all layers to a platform project to make all README files available for viewing or editing.

The following procedure checks out all project layers from the git repository and add them to the *projectDir*/layers directory. Once you perform this procedure, the layer README files are accessible and do not require a git command to view or edit.

Step 1 Configure a platform project.

In this example, you will use the **--enable-checkout-all-layers=yes** configure option to add all layers.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-checkout-all-layers=yes
```

Step 2 View the README file.

Once the **configure** script completes, the README files are available in their respective layers, for example:

projectDir/layers/examples/lemon_layer

PART II

Platform Project Image Development

Configuration and Build.....	61
Localization.....	91
Portability.....	95
Layers.....	99
Recipes.....	109
Templates.....	115
Finalizing the File System Layout with changelist.xml.....	119

5

Configuration and Build

Introduction	61
About Building Platform Project Images	72
Build-Time Optimizations	76
Examples of Configuring and Building	77
About Creating Custom Configurations Using <code>rootfs.cfg</code>	80
EGLIBC File Systems	85

Introduction

With Wind River Linux, you must configure and build a platform project image to develop it to suit your needs.

For information on configuring and building platform projects in general, see the *Wind River Linux Getting Started Guide: Developing a Platform Project Image Using the Command-Line*.

Before you begin, see:

- [Directory Structure for Platform Projects](#) on page 43
- [Run-time Software Configuration and Deployment Workflow](#) on page 27

Design Benefits

The design of the Yocto Project BitBake build system with Wind River Linux offers several important benefits:

- If a pre-built kernel and file system are satisfactory for deployment, or current testing and development, you can build a complete run-time file system in minutes using prebuilt kernel and file system binaries.
- You can build specific parts from source files, saving time by building only the file system, or only the kernel, or a specific package, whichever element is of current interest.
- Your builds cannot contaminate the original packages, layers, recipes, templates, and configuration files, because the development environment is kept separate from the build

environment. For additional information, see [Directory Structure for Platform Projects](#) on page 43 and [Directory Structure for the Development Environment](#).

- You can include all of your project changes in the `projectDir/layers/local` directory to simplify development. See [About the layers/local Directory](#) on page 54
- By using custom layers and templates (see [About Layers](#) on page 99 and [About Templates](#) on page 115, you can add packages, modify file systems, and reconfigure kernels for repeatable, consistent builds, yet still keep your changes confined for easy removal, replacement, or duplication.

These last two features allow multiple builds, customized builds, and a strict version control system, while keeping the development environment pristine and intact.

You create the build environment as a regular user with the `configure` script. For additional information, see [About the Configure Script](#) on page 63. It is in this environment that you build (`make`) Wind River Linux run-time system, either default or customized, using software copied or linked from the development environment.

Although this information is oriented toward the command line, it will also give Workbench users a better understanding of the process of creating a Wind River Linux platform project.



NOTE: The `configure` script checks for required host updates, and notes them in `config.log`, a text file within your `projectDir` directory.

About Creating the Platform Project Build Directory

Before you configure and build a platform project image, you must first create the directory to manage your build.

The platform project directory is a directory that you create in the build environment, as opposed to the development environment. See [Directory Structure for the Development Environment](#), for an explanation of development environment in Wind River Linux.

You typically create a platform project within a work directory, called in this guide `workdir`. Within `workdir` you create a subdirectory for the particular project, which will be referred to in this guide as `projectDir`.

While you can name your `projectDir` anything you like, in this guide, we use examples that indicate the system configuration and contents of the project, for example `common_pc_small`, to indicate an x86 common pc target platform with a small file system. Another example is to use the BSP name and file system, such as `qemux86-64_small`. This example indicates the project is configured with the 64-bit QEMU x86 platform with a `glibc_small` file system.

After you create your project directory, you can define the shared state cache (`sscache`) and `ccache` environment variables to significantly help speed up your build process. See the *Wind River Linux Getting Started Guide: Creating a Platform Project from the Command-Line*, for information on setting these variables.

For information on the project directory, see

- [About the Project Directory](#) on page 41
- [Directory Structure for Platform Projects](#) on page 43

About Configuring a Platform Project Image

Use the `configure` script with supplied options to configure a project.

Project configuration is the actual creation of the project files based on the using the **configure** script with options to define the platform project.

Minimum **configure** options include the board support package (BSP), kernel, and rootfs (file system). Additionally, you can use layers, profiles, and templates to add additional features to your platform project, depending on your requirements.

The **configure** script accepts many optional arguments besides board, kernel, and root file system. For example you can specify additional features, build optimizations, restrict it to kernel builds, request different kinds of images from the build and more. See the **configure** command help (**-h** or **--help** option), [Configure Options Reference](#) on page 66, and additional examples in this guide for details.

When you have configured your project, you can build it, as described in *About Building Platform Project Images*.

Initializing the Wind River Linux Environment

Learn how to initialize the development environment and create an environment variable to simplify platform project configuration.

This procedure is optional, but can help save time when you configure platform projects.

Step 1 Navigate to the Wind River installation directory.

```
$ cd installDir
```

Step 2 Set the environment variable.

```
$ ./wrenv.sh -p wrlinux-5
```

You will then have an environment variable for the Wind River **configure** script, named `WIND_LINUX_CONFIGURE`.

Step 3 Use the environment variable to configure a platform project.

```
$ $WIND_LINUX_CONFIGURE options...
```

About the Configure Script

The configure script is the most important of several key configuration files—it initiates the entire configuration process.

It creates a subdirectory structure within the project directory and populates it with the script framework, configuration files and tools necessary to build the run-time system. It processes board templates and initial package files, and copies basic run-time file system configuration files (for the **etc** and **root** directories), from the development environment.

The script is always run with options. Which options you supply depend on which kernel and file system you wish to build for your board, which features you want to include, and whether you wish to build a complete run-time system, or only a kernel or only a file system.

The **configure** script produces a plain text log file, **config.log**, within the project directory, in this case, **workspace/qemux86-64**. This is a very useful file, recording **configure** options, automatic checking of host RPM updates, and so on. Workbench saves a similar log file, **creation.log** which contains the screen output of the configure command.

The configure script is located in `installDir/wrlinux-5/wrlinux/`, where `installDir` is the path to your Wind River software installation. Throughout this manual, this location is referred to as `configDir`.

→ **NOTE:** Do not run `configure`, builds (`make` target), or Workbench as root because this may interfere with the operations of the build system.

The examples in this documentation show running the `configure` script relative to an `installDir` of `/opt/WindRiver/wrlinux-5/wrlinux`, for example:

```
$ configDir/configure options...
```

If your installation is in a different location, replace `configDir` with the location of your installation. If you are using Workbench, you run the configure script by clicking **Finish** in the new project wizard.

To help simply using the configure script, you may wish to create an environment variable for it.

→ **NOTE:** You should not alias `configure` to be the full path to the `configure` script, or add the path to configure in your `PATH`, because this could cause problems. If, for example, you install or update a local host package that requires running the host operating system's `configure` command, the Wind River Linux `configure` script could be called instead. You could, however, alias a different name, for example `wrconfig`, to be the full path to the Wind River Linux `configure` script.

See [Examples of Configuring and Building](#) on page 77 for instructions on configuring and building various project types, kernels and packages.

A Common Configure Command Error

The `configure` script fails with an error if you have `."` in your `PATH` environment variable. In addition to being a security issue, having a `."` in your `PATH` can cause problems with the build. Remove `."` from your `PATH` (for example, by editing the `PATH` setting in your `.bashrc`, `.cshrc`, or other startup file and then reinitializing it) before running the `configure` script.

Specifying a Standard Configuration

As a minimum, you must specify at least a board, kernel, and root file system to the `configure` script.

To configure a platform project image, you must first create a project directory in the build environment. For additional information, see [About Creating the Platform Project Build Directory](#) on page 62.

Step 1 Navigate to the platform project directory (`projectDir`).

```
$ cd projectDir
```

Step 2 Run the `configure` script with options..

For example, to configure a project for a common PC platform with a standard Wind River Linux kernel and file system, use the following command:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=linux \  
--enable-rootfs=ext4 \  
--enable-target=pc \  
--enable-bsp=pc \  
--enable-board=pc \  
--enable-kernel=linux \  
--enable-rootfs=ext4 \  
--enable-target=pc \  
--enable-bsp=pc
```



```
--enable-kernel=standard \  
--enable-rootfs=glibc_std
```



NOTE: The board parameter, `qemux86-64` in this case, is also referred to as the board support package, or BSP.

While this configuration does not include any build time optimizations, Wind River recommends using them to speed up platform project builds. For additional information, see [Build-Time Optimizations](#) on page 76.

Step 3 Press **ENTER** to configure the project build directory.

Platform project configuration typically takes between one to two minutes.

Once a project is configured, you can build it. Refer to *About Building Platform Project Images* for examples of building projects with the **make** command.

About Configure Options

Use configure script options to tailor your platform project to your specific development needs.

The **configure** script requires that you specify a board (BSP), kernel type, and rootfs. These options provide the information necessary for the build system to create a complete runtime system.

To configure a build of a complete run-time system, the necessary options are:

--enable-board=*bspName*

--enable-kernel=*kernelType*

--enable-rootfs=*rootfsType*

After the project configures successfully, use the **make** command to build everything. See *About Building Platform Project Images*.



NOTE: With the exception of the `glibc_small` file system, the **configure** script creates file systems that by default contain debugging functionality. See [Examples of Configuring and Building](#) on page 77 for details on adding debugging capabilities to small file systems. The **glibc_small +debug** feature template adds additional debug support to small file systems.

Do not repeat arguments to the **configure** script, because only the last one will be used. For example, if you specify:

--enable-build=production --enable-build=profiling

The **configure** script sets the version to **profiling**. If you want to specify multiple non-exclusive features, use comma-separated lists, for example:

--with-template=*template1,template2*

or add features to the root file system with the “+” shorthand such as”

--enable-rootfs=glibc_small+feature1+feature2+feature3.

Configure Options Reference

The configure script can be run with a large number of options, as explained in this section. You can display a complete list with the following command:

```
$ configDir/configure --help
```

This section describes some of the more commonly used configure options.

Required Configure Options

The following table summarizes basic configuration options.

Table 3 Required Configure Options

Option	Description
<code>--enable-board=<i>boardname</i></code>	Specifies the target board. The list of board support packages that are currently installed is given in the <code>--help</code> output. A full list of supported boards can be found at Wind River Online Support. A board specification implicitly includes <code>cpu</code> and <code>arch</code> because the <code>board</code> template includes defaults through <code>include</code> files. This option is equivalent to specifying <code>--with-template=board/<i>boardname</i></code> .
<code>--enable-kernel=<i>kernel</i></code>	Specifies the kernel. This option is equivalent to specifying <code>--with-template=kernel/<i>kernel</i></code> option.
<code>--enable-rootfs=<i>rootfs</i></code>	Specifies the file system. This option is equivalent to specifying <code>--with-template=rootfs/<i>rootfs</i></code> option.

Additional Configure Options


With `configure` options that use `[yes|no]`, the default is `no`, which is the equivalent of not using the option at all.

Table 4 Additional Configure Options

Option	Description
<code>--enable-bootimage=<i>option</i></code>	Specifies the creation of a boot image using a comma-separated list containing one or more of the following image types: <code>ext3</code> , <code>ext4</code> , <code>hdd</code> , <code>iso</code> , <code>jffs2</code> , <code>tar.gz</code> , <code>tar.bz2</code> , <code>ubifs</code> , and <code>-tar.bz2</code> . <code>tar.bz2</code> is the default image type. To disable this, you must select <code>-tar.bz2</code> , or use <code>tar.gz</code> to set it as the default. Note that after the build completes, you must run the <code>make boot-image</code> command to actually build the

Option	Description
--enable-build=[debug production productiondebug profiling]	<p>image. Once complete, the image resides in the projectDir/export/ directory. See About Configuring and Building Bootable Targets on page 265.</p> <p>When building or rebuilding a platform project (make), use the following options to specify your build optimization:</p> <p>debug</p> <p>Use to compile and install binaries and libraries with debugging information (-g). Can be used for on-target debug or cross debug with Workbench. Performance optimizations that interfere with debug-ability are disabled with the following additional options:</p> <p>-O0 -fno-omit-frame-pointer</p> <p>The platform build produces a single file system image that includes feature/dbg template packages.</p> <p>The debug items are installed into a .debug subdirectory. For example, the /bin/bash debug file is located in /bin/.debug/bash, with corresponding sources in /usr/src/debug/bash</p> <p>production</p> <p>This is the default build optimization. Use to optimize and strip installed libraries and binaries. The default package compile options are used, which typically results in the best performance optimization. A size and performance optimized file system image is produced.</p> <p>Subsequently building the make fs-debug target will produce an additional file system image with debuginfo in the filename, containing only debug information and source, in the same .debug subdirectory format as the debug build. This debug information can be used as a reference file system for cross-debug with Workbench or gdbremote, or overlaid on the default file system image and deployed for on-target debug. But since the compile options are performance optimized, redundant code and some automatic variables will be optimized out, program</p>

Option	Description
	<p>flow may be reordered, and only limited stack tracing information is available.</p> <p>productiondebug</p> <p>Use to include production optimizations, but also install all symbols and debug info packages on the target system image. This option does enable on-target debug packages. This is the equivalent of building the production filesystem image and then the production fs-debug image and combining them.</p> <p>profiling</p> <p>Use profiling to compile programs and binaries with stack frames to enable use of profiling tools. The binaries are production optimized, but are not stripped (the debug information is in the binary, not a .debug directory). The on-target debug packages are also included. The following profiling optimizations are added to package compilation:</p> <ul style="list-style-type: none"> • -fno-omit-frame-pointer • -fvisibility=default
<p>--enable-build-tools=[auto yes]</p>	<p>auto or default</p> <p>By default, the build system performs a sanity check to determine whether standalone build tools will be installed for your platform project configuration. Omitting this configure option is the same as specifying auto.</p> <p>yes</p> <p>Use this option to skip the sanity check and force a build tools installation.</p>
<p>--enable-ccache=[yes no]</p>	<p>Specifies whether to use ccache to speedup project builds. While optional and not required to configure and build platform project images, ccache will help speed up the build process for repeated build and/or delete cycles in the project directory.</p> <p>To use ccache, Wind River recommends that the ccache and the platform project directories reside on the same physical volume and file system to limit potential negative effects on the cache speed. In particular, it is suggested not to install the ccache directory on a NFS-mounted system, since this is not a fully tested</p>

Option	Description
	scenario. Your development workstation must have ccache installed. For additional information on using ccache , see the <i>Wind River Linux Getting Started Guide: Configuring a Platform Project</i>
<code>--enable-checkout-all-layers=[yes no]<i>option</i></code>	Use to checkout all product layers in the installation and add them to the projectDir/layers directory. By default, the build system only adds layers that are part of your platform project configuration.
<code>--enable-doc-pages=target</code>	Use this option to include the documentation man/info pages in the file system of the target.
	 NOTE: This option only applies to the <code>glibc_std</code> file system.
<code>--enable-internet-download=[yes no]</code>	If used in your configure script, and a package is not found in the local installation, the build system will attempt to download it from the Internet.
<code>--enable-jobs=<i>number</i></code>	Specifies the maximum number of parallel jobs that make should perform. This should be set to the number of CPUs your system has available.
<code>--enable-ldconfig=[yes no]<i>number</i></code>	This option places a post-installation script in RPMs to update the <code>ldconfig</code> cache during installation, and also generates the etc/ld.so.conf and etc/ld.so.conf.d files on the image at the system level. For additional information, see About ldconfig on page 259.
<code>--enable-parallel-pkgbuilds=<i>number</i></code>	Sets the number of packages that can be built in parallel to speed up the build process. As a rule of thumb, set this number equal to the number of CPUs available in your workstation. Using the configure option above sets
<code>--enable-prelink=[yes no]</code>	Pre-links target binaries and libraries. If unused, defaults to yes.
<code>--enable-reconfig</code>	Enables the configure script to run with different or added options on a previously configured platform project.
<code>--enable-rm-oldimgs=[yes no]</code>	Removes old root file system files located in the projectDir/bitbake_build/tmp/deploy/images directory, and retains only the latest image.

Option	Description
--enable-rm-work=[yes no]	<p>Each time you run make fs to build a platform project file system, by default the build system retains a copy of the root file system created at the time the command is run. As a result, this can use up a lot of disk space for each successive build. Use this option to ensure only the latest *.bz2 system image is retained.</p> <p>Incrementally removes objects from your build area after the build successfully completes. This option adds the following line to the projectDir/local.conf file:</p> <pre data-bbox="865 638 1427 688">INHERIT += "rm_work"</pre> <p>This line erases the staging area used to compile a package when the package is successfully built. This can also save a significant amount of disk space when you consider a few packages in the glibc-std build take between 200MB to 300MB each to compile.</p>
--enable-scalable=mklb	<p>Specifies whether target binaries are to be optimized by removing some unused functions.</p>
--enable-sdkimage-staticlibs=[yes no]	<p>Use to install static libraries in your SDK images. By default, static libraries are not installed.</p>
--enable-stand-alone-project=[yes no]	<p>Creates a platform project that is completely stand-alone, and not dependant on the Wind River Linux installation (installDir) for project build and development.</p> <p>When you configure a platform project with this option, the project does not require the Wind River Linux installation to build and develop it, however, there are additional steps required to copy or move the project. For additional information, see About Platform Project Portability on page 95.</p>
--enable-target-installer=[yes no]	<p>Enables the target installer feature, which adds the necessary files to the platform project to create an installable Wind River Linux distribution. May be used with the --with-installer-target-build= option to specify another project's *.ext3 root file system..</p>



NOTE: Wind River Linux only supports *.ext3 root file system files for the target installer feature.

Option	Description
	If you do not specify a project directory with the --with-installer-target-build option, the project itself will be used to create the target file system.
--enable-test =[yes no]	Includes the standard suite of test packages for the file system and kernel.
--help	Prints an option summary similar to this table and exits without creating a project.
--with-installer-target-build = <i>full_path_to_target_system_*.ext3</i>	When used with the --enable-target-installer=yes option, this option specifies the location of a built platform project's projectDir/export/images/*.ext3 root file system image file that will be used to create an installable Wind River Linux distribution.
--with-license-flags-blacklist = <i>licenseType1, licenseType2, licenseType3...licenseTypeN</i>	Use this option to set a comma-separated list of license types that are excluded from the platform project image. If you specify a license type, for example, GPLv3 , to be blacklisted, any package specified to use that license type will not be included in the platform project image once built. If you include a configure option that adds packages that require a specific license to function, and that license type is blacklisted, the full contents specified by the option will not install. This may create an unsupported configuration.
--with-license-flags-whitelist = <i>licenseType1, licenseType2, licenseType3...licenseTypeN</i>	Use this option to set a comma-separated list of license types that are included automatically in the platform project image. Note that some software license types have legal requirements. As a result, you should consult your company's legal department's software policy regarding any license type you want to include.
--with-rcpl-version = <i>000x</i>	Once you have updated the product, new platform projects default to the latest installed RCPL release when you run configure script commands. Use this option to specify an earlier patch release, where <i>000x</i> is the RCPL version, for example, <i>0006</i> . This allows you to reproduce a previous environment, such as one used to release a product.
--with-template = <i>template1, template2, template3</i>	Appends the specified templates to the usual template list created by the configure options.

Option	Description
--without-template= <i>template1,template2,template3</i>	Specifies the default templates to exclude from the project and omit from the generated local.conf file.
--with-layer= <i>layer1,layer2,layer3...</i>	Specifies custom layers. The system will process any template of the same name found within a layer instead of the regular template within the development environment. (The regular template may, however, be included by the template in the custom layer.)
--without-layer= <i>layer1,layer2,layer3...</i>	Specifies the layers to exclude from the generated bblayer.conf file.

About Building Platform Project Images

About the make Command

The make command builds platform projects, application source, and packages.

Platform Projects

After you have configured a project as described in [About Configuring a Platform Project Image](#) on page 62, you can build it using the **make** command. The build produces the target software such as the kernel and file system for a particular board, depending on how you configured it.

When you run **make**, **make fs**, or **make all**, it builds (or rebuilds) the platform project using the specified options. If source changes are detected, the binary packages associated with those changes are automatically rebuilt. Like many Wind River Linux **make** targets, this is a wrapper to an equivalent of Yocto build command, in this case: **bitbake wrlinux-image-filesystem-type**; for a cross reference see [Yocto Project Equivalent make Commands](#) on page 73.



NOTE: Build times will differ depending on the particular configuration you are building, the amount of data that can be retrieved from sstate-cache, and on your development host resources.

When you build the platform project, this generates and extracts the root file system for the platform project initially. If you run one of the **make** commands again, it will only regenerate and extract the file system if something has changed.

To force a file system generation, simply touch the image ***.tar** file before running the **make** command. For example, from the **projectDir**:

```
$ touch export/*.tar*
```

In many cases you can reduce the amount of time required for project builds by specifying various caching and parallelizing options to **configure** or **make**. In addition, there are environment variables you can set if you want to always use these options, or only selectively not use them. Refer to [Build-Time Optimizations](#) on page 76 for more information on improving project build times.

In addition to the basic **make** commands that build the platform project and generate the root file system and kernel images, Wind River Linux provides commands to help simplify development tasks, such as generating the software development kit (SDK) for application development, and launching simulated QEMU or Simics target platforms. For a list of **make** commands, see [Common make Command Target Reference](#) on page 351.

Applications and Packages

Wind River Linux and the Yocto Project BitBake build system use the **make** command to perform various development actions on applications and packages. These actions include basic development tasks such as building, rebuilding, compiling, cleaning, installing and patching packages. For a list of **make** commands, see [Common make Command Target Reference](#) on page 351.

Packages and their dependencies are built by specifying the recipe associated with the package, for example:

```
make -C build recipeName
```

In this example, *recipeName* can refer to the package name without the ***.bb** suffix, or the git or version number associated with the recipe. For example, to build the **hello.bb**, **hello_git.bb**, or **hello_1.2.0.bb** package, you would use the following command:

```
make -C build hello
```

When the recipe builds, it will include any dependant recipes and their associated packages in the build process.

Yocto Project Equivalent make Commands

Wind River Linux is compatible with the Yocto project. Learn about the Yocto BitBake equivalents for common **make** commands.

Common make Command Equivalents

For a list of **make** commands, see [Common make Command Target Reference](#) on page 351.

Wind River Linux make command	Yocto Project BitBake equivalent
make bbs Sets up the BitBake environment, such as the variables required, before you can run BitBake commands. This command executes a new shell environment and configures the environment settings, including the working directory and <i>PATH</i> . To return to the previous environment, simply type exit to close the shell.	Source layers/oe-core/ <i>oe-init-buildenv</i> bitbake_build

Wind River Linux make command	Yocto Project BitBake equivalent
make fs	<p>bitbake <i>imageName</i></p> <p>For example, bitbake wrlinux-image-glibc-std. To determine the correct <i>imageName</i>, you can either:</p> <ul style="list-style-type: none"> Refer to your original configure line, where the option --enable-rootfs=glibc-std translates to wrlinux-image-glibc-std in the example above. Refer to bitbake_build/conf/local.conf and find the value assigned to DEFAULT_IMAGE. For the example above, this line will look like: <pre>DEFAULT_IMAGE = "wrlinux-image-glibc-std"</pre>
	<p>➔ NOTE: make fs also extracts the image and makes it ready for make start-target.</p> <p>It may be possible to build for other images, but only the configured image will have templates and other configurations applied. Other images may not work.</p>
make -C build <i>recipeName</i>	bitbake <i>recipeName</i>
Build the package's recipe <i>recipeName</i>	
make -C build <i>recipeName.rebuild</i>	bitbake -c rebuild <i>recipeName</i>
make -C build linux-windriver.build	bitbake linux-windriver
Build the Wind River Linux kernel's recipe.	
make -C build linux-windriver.rebuild	bitbake -c rebuild linux-windriver
Rebuild the Wind River Linux kernel's recipe.	

Wind River Linux 4.3 Compatibility

Some specific tasks are translated to better assist customers migrating from Wind River Linux 4.3. In the following table, you may substitute the *packageName* variable for *recipeName*, except where specified.

Wind River Linux make command	Yocto equivalent
make -C build <i>packageName.distclean</i>	bitbake -c cleansstate <i>recipeName</i>

Wind River Linux make command	Yocto equivalent
<code>make -C build <i>packageName.config</i></code>	<code>bitbake -c configure <i>recipeName</i></code>
<code>make -C build <i>packageName.download</i></code>	<code>bitbake -c fetch <i>recipeName</i></code>
<code>make -C build <i>packageName.rebuild</i></code> and <code>make -C build <i>packageName.rebuild_noddep</i></code> →	<code>bitbake -c compile <i>recipeName</i></code> NOTE: Notice that the bitbake flag in this case is <code>-C</code> rather than <code>-c</code> .
<code>make -C build <i>packageName.quilt</i></code> and <code>make -C build <i>packageName.quiltprep</i></code> →	<code>bitbake -c quiltprep <i>recipeName</i></code> NOTE: <code>quiltprep</code> is not a community step. It is a Wind River addition.
<code>make -C build <i>packageName.addpkg</i></code> → NOTE: You must specify the package name for this command option.	Edit <code>layers/local/<i>image.bb</i></code> and add the following line: <pre>IMAGE_INSTALL += "<i>packageName</i>"</pre>
<code>make -C build <i>packageName.rmpkg</i></code> → NOTE: You must specify the package name for this command option.	Edit <code>layers/local/<i>image.bb</i></code> and remove or comment out the following line: <pre>IMAGE_INSTALL += "<i>packageName</i>"</pre>
<code>make -C build <i>packageName.env</i></code>	<code>bitbake -e <i>recipeName</i></code>

About Build Logs

When you build package recipes, the build system creates symlinks to separate build output logs for each package in `projectDir/build/packageName/temp/`.

Generally speaking, the BitBake build system generates one log per task, and a typical package build runs four or more tasks. For example, the logs created for the hello package are located in `projectDir/build/hello-1.0-r1/temp` directory, and include:

- `log.do_compile`
- `log.do_package_write_rpm`
- `log.do_configure`
- `log.do_patch`
- `log.do_fetch`
- `log.do_populate_sysroot`
- `log.do_install`
- `log.task_order`
- `log.do_package`

See also the online output of the **make help** command (not **make -help**) in your project build directory after you have configured a project.

Build-Time Optimizations

There are several options you can use that can reduce your total build time and save build environment disk space.

You can do this with environment variables so that you can “set it and forget it”, or you can add command line options to your **configure** script or **make** command to perform the same optimizations. Using the command-line options will override your environment variable settings.

Use the examples in this section to implement the available configure and build optimization options. Note that you can control these settings through Workbench as well, as described in *Wind River Workbench by Example, Linux 5 Version*.

Configure and Build Optimization Option Combinations

You can combine configure and build optimization options in various ways. For example, you could have both of the following options on your configure line:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-jobs=5 \  
--enable-parallel-pkgbuilds=5
```

To allow multiple instances of **gcc** when a package is building, use the **--enable-jobs** option.

To allow multiple packages to build simultaneously, specify **--enable-parallel-pkgbuilds**.

In this example, the build system will build up to five packages at once with up to five instances of **gcc** per package, resulting in a maximum of up to 25 parallel operations occurring.



NOTE: In practice, that is very atypical since most packages cannot support nearly that level of parallelism and only the build stage can typically be parallelized at all. Because of that, the parallelism provided by building separate packages in parallel generally offers a much larger performance improvement than building individual packages in parallel.

In addition, the actual parallelism achieved depends on the dependency lists of the packages. For example, if you are building **glibc** and all of your other packages depend on **glibc**, those other packages must wait for **glibc** to be completed before they can be built.

The numbers shown in the example are not unreasonable for, for example, a dual-core workstation. For more powerful configurations, a rule of $n+1$ operations might be a reasonable configuration. For example, on an eight-core machine, you could set the two values to **9** for a maximum total of 81 concurrent build operations. Note that these are just suggestions, and you should find optimal settings for your specific environment.

Optimizing Toolchain and **glibc** Builds

To get the best performance on toolchain and **glibc** builds, use a smaller number of parallel packages (one is plenty), and a larger **--enable-jobs** value.

Minimizing Build Environment Disk Space

The following **configure** script options will help minimize your build environment disk space.

--enable-rm-oldimgs=yes

Each time you run the **make** or **make fs** command in the platform project directory, the build system creates a copy of the root file system at build time in the **projectDir/bitbake_build/tmp/ deploy/images/** directory. Each build creates files that can consume up to 10MB or more, depending on your platform project configuration.

After a few builds, this can consume a lot of disk space. You can ensure that only the latest version of the root file system file(s) is maintained using the **--enable-rm-oldimgs configure** script option. When you use this option as part of your platform project configuration, only the latest version of the root file system file(s) will reside in the directory.

--enable-rm-work=yes

Incrementally removes objects from your build area after the build successfully completes.

This option adds the following line to the **projectDir/local.conf** file:

```
INHERIT += "rm_work"
```

This line erases the staging area used to compile a package when the package is successfully built. This can also save a significant amount of disk space when you consider a few packages in the **glibc-std** build take between 200MB to 300MB each to compile.

Examples of Configuring and Building

Follow examples to learn various strategies for configuring and building platform projects and packages using Wind River Linux.

All **configure** script examples assume you are in a project directory that you have created as an ordinary user (not root). Running the following command in your build directory would create a project directory and navigate to it:

```
$ mkdir -p projectDir && cd projectDir
```

Where **projectDir** is the name you choose, for example, **qemux86-64-glibc-small**.

Configuring and Building a Complete Run-time

Use this **configure** script example as a basis to create your platform-project image.

The following example procedure assumes you are in a project directory that you have created as an ordinary user (not root).

A full project configuration requires you to specify a board (BSP), a kernel type, and a root file system type at a minimum. The options may be entered in any order, but the basic syntax is:

```
$ configDir/configure \  
--enable-board=BSP \  
--enable-kernel=kernel_type \  
--enable-rootfs=rootfs_type
```

Step 1 Specify the configuration for your run-time system with the **configure** command.

The following example configures the complete target software for a qemux86-64 BSP with a standard kernel and a small glibc-based file system:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small
```

The configure script usually take a minute or two to complete.

Step 2 Build the project.

```
$ make
```



NOTE: In this example, no **debug** or **demo** templates have been added to the small file system configuration, which makes for a smaller run time, but it is one that does not have debug tools, such as the **usermode-agent**, built in. See [Configuring and Building Complete Debug-Capable Run-time](#). for an example in which debug capabilities are added. By default, Workbench builds small file systems with debug and demo tools included.

Commands for Building a Kernel Only

Use these example **make** commands as a basis to configure and build a kernel.

Table 5 Common kernel build commands

To achieve this	Do this
Build the kernel	You can build the kernel by specifying the linux target: <pre>\$ make -C build linux-windriver</pre>
Configure the kernel	To configure Wind River Linux kernel options, you can use standard Linux tools such as xconfig or menuconfig , for example: <pre>\$ make -C build linux-windriver.menuconfig</pre> See About Kernel Configuration and Patching on page 177 for examples of kernel configuration.
Rebuild the kernel	If you have made changes to your project such as changing the kernel configuration, rebuild the kernel with this command: <pre>\$ make -C build linux-windriver.rebuild</pre>

Configuring and Building a Flash-capable Run-time

Use the following example **configure** script to create your flash-capable platform-project image.

You can configure a complete run-time system (kernel and file system) with subsequent creation of a flash file system enabled, using the **--enable-bootimage=** configure option.

Step 1 Configure a platform project to specify the flash boot image.

In this example, the `--enable-bootimage=` configure option defines a JFFS2 (journaling flash file system version 2) boot image.

```
$ configDir/configure \  
--enable-board=bsp \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-bootimage=jffs2
```

While this configuration does not include any build time optimizations, Wind River recommends using them to speed up platform project builds. For additional information, see [Build-Time Optimizations](#) on page 76.

Step 2 Build the project.

```
$ make
```

For additional supported images, see [About Configuring and Building Bootable Targets](#) on page 265.

Configuring and Building a Debug-capable Run-time

Use the following **configure** script examples to create a debug-capable platform-project image.

Use this example procedure to configure a complete run-time system (kernel and file system) with subsequent debugging enabled.

Step 1 Configure a platform project with debug features enabled.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small+debug \  
--enable-build=debug
```

This configure example uses two options to provide debug capability to the platform project:

--enable-build=debug

Adds application debugging features to the file system. See [Configure Options Reference](#) on page 66.

--enable-rootfs=glibc_small+debug

Adds the feature/debug template, which adds debug-specific functionality to the target file system, including the following tools: **elfutils**, **ltrace**, **memstat**, **strace**, and the Wind River LTTng trace daemon.



NOTE: This example provides a shorthand method for adding a template using the configure command. The standard method is to use the `--with-template=` configure option, which in this example would be `--with-template=feature/debug`.

Step 2 Optionally add basic graphics capability to your runtime.

Similarly, to add **demo** capability (basic graphics capabilities) to a **glibc_small** file system, you could either include the **--with-template=feature/demo** option to the configure command, or just specify the file system as **--enable-rootfs=glibc_small+demo** as follows:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small+demo
```

Step 3 Build the project.

```
$ make
```

For more information on the features provided by the **debug** and **demo** templates, see the **installDir/wrlinux-5/layers/wr-base/templates/feature/demo** and ***debug** directories.

Building a Target Package

After you have configured a platform project, you can build a particular target package, for example after making changes to its source code.

Use the following procedure to build a target package.

Step 1 Extract and patch the package source.

```
$ make -C build recipeName.patch
```

This extracts and patches the package source and places it under **build/package-version/** directory.

Step 2 Build the package source.

```
$ make -C build recipeName.rebuild
```

The package, and any packages that are dependant on it, will be rebuilt.

About Creating Custom Configurations Using **rootfs.cfg**

You can use the **projectDir/layers/wr-base/templates/rootfs.cfg** file as a reference to define your own **rootfs.cfg**, which creates one or more custom file system types to reduce the number of arguments that you pass to the configure script.

You can use a custom **rootfs.cfg** to automatically set the kernel type, and include specific layers and templates. This differs from the standard workflow where you use separate **configure** script options to define the root file system, kernel, layers, and templates. For example, a basic **configure** script command using the standard workflow may include the following options:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--with-template=feature/debug,feature/analysis \  
--with-layer=meta-selinux,wr-intel-support
```


In this example, even while using simplified `--with-layer` and `--with-template` options to specify additional layers and templates on the same line, the overall configuration requires five different command options.

By creating a new, custom `projectDir/myLayer/templates/rootfs.cfg` file based on `projectDir/layers/wr-base/templates/rootfs.cfg`, you can automatically include the options above. The result is a simplified `configure` script command that requires only three options, for example:

```
$ configDir/configure \  
--with-layer=path_to_myLayer \  
--enable-board=qemux86-64 \  
--enable-rootfs=glibc-custom
```

If you frequently use the same templates or layers as part of your platform development, or need to specify a different or custom kernel type, you can create a `rootfs.cfg` file, and define a new root file system, with other options, in the file, such as the `glibc-custom` rootfs option in the example above. Once created, the new file is available for use by the `configure` script as long as you specify the file's location using the `--with-layer=path_to_myLayer` in the `configure` script command.

New Custom rootfs Configuration Workflow

The workflow for creating a custom rootfs includes the following:

1. Configure and build, or have a previously configured platform project available.
2. Copy the `projectDir/layers/local` directory from the platform project above to a location on your development system, and rename the layer.

For example, you could copy and name it to

`/home/user/myLayer`

3. Create a new `/home/user/myLayer/templates/rootfs.cfg` file, and populate it with the features that you want to include. For example:

```
[rootfs]  
glibc-custom = image  
  
[glibc-custom "image"]  
default-ktype = standard  
compat = wrlinux-*  
default = wrlinux-image-glibc-core  
allow-bsp-pkgs = 0  
  
[glibc-custom "distro"]  
default = wrlinux  
compat = wrlinux  
  
[glibc-custom "vars"]  
layers = meta-selinux,wr-intel-support  
templates = feature/debug,feature/analysis
```

Depending on your project requirements, you can define a custom rootfs name, the kernel type, and add the layers and templates that you require.



NOTE: The kernel types, layers, and templates you include in this file must be part of your Wind River Linux installation. Missing layers and templates, or misspelled kernel type, layer, and template names, will return an error and halt the `configure` script.

4. Configure a new platform project, and:

- Use the **--with-layer=** configure option to point to the new layer.
- Refer to the new name you gave your rootfs in the **--enable=rootfs=** configure option.

For example:

```
$ configDir/configure \  
--with-layer=/home/user/myLayer \  
--enable-board=qemux86-64 \  
--enable-rootfs=glibc-custom
```

5. Build the platform project, and verify that the options, such as layers, templates, and kernel types, are included in the build.

As long as you specify the location of the layer that contains the **rootfs.cfg** file, you can reuse the new custom rootfs in any platform project configuration.

About the rootfs.cfg File

The **projectDir/layers/wr-base/templates/rootfs.cfg** file defines the available root file system options for configuring a platform project. The **--enable-rootfs=** option that you specify in the **configure** script command must have a valid entry in the **rootfs.cfg** file. The default file is as follows:

```
[rootfs]  
glibc-core = image  
glibc-small = image  
glibc-std = image  
glibc-std-sato = image  
[ktypes]  
standard = ktype  
preempt-rt = ktype  
[glibc-small "image"]  
default-ktype = standard  
compat = wrlinux-  
default = wrlinux-image-glibc-small  
allow-bsp-pkgs = 0  
[glibc-small "distro"]  
compat = wrlinux  
default = wrlinux  
[glibc-core "image"]  
default-ktype = standard  
compat = wrlinux-  
default = wrlinux-image-glibc-core  
allow-bsp-pkgs = 0  
[glibc-core "distro"]  
default = wrlinux  
compat = wrlinux  
[glibc-std-5.0 "image"]  
default-ktype = standard  
compat = wrlinux-  
default = wrlinux-image-glibc-std-5.0  
[glibc-std "image"]  
default-ktype = standard  
compat = wrlinux-  
default = wrlinux-image-glibc-std  
[glibc-std "distro"]  
default = wrlinux  
compat = wrlinux  
[glibc-std-sato "image"]  
default-ktype = standard  
compat = wrlinux-  
default = wrlinux-image-glibc-std-sato  
[glibc-std-sato "distro"]  
default = wrlinux  
compat = wrlinux
```

This example includes entries for the root file system and kernel options described in *Kernel and File System Components* on page 16, and includes the following:

[rootfs]

This section lists the names of the available root file systems. Each entry in this section requires its own separate **image** and **distro** entry in the file.

To create a new custom rootfs configuration, enter a name for it in this section on a separate line.



NOTE: Do not use underscores (`_`) in your **[rootfs]** entries. These are not recognized by the build system and can cause the build to fail.

[ktypes]

This section includes the available kernel types. Once defined, you can specify the kernel type in the **image** section if you want it to be used automatically when you specify the rootfs. If you have a custom kernel type, and want to make it available for platform project configuration, you would enter it here.

[rootfsName "image"]

The **image** entry lets you specify the following options:

default-ktype

This entry is optional, and is used to automatically include a specific kernel type when you use the rootfs name in your configure command. It must be a valid name defined in the **[ktypes]** section of the **rootfs.cfg** file.

compat

Specifies compatibility with Wind River Linux recipe file names that begin with **wrlinux**.

default

Specifies the recipe ***.bb** file name used to create the image. If you create a custom rootfs entry, you can use an existing recipe name, or create a new recipe. The name used must match an existing recipe file, located in a layer that's included in the **projectDir/bitbake_build/conf/bblayers.conf** file.

allow-bsp-pkgs=0

This optional entry prevents the addition of additional packages being added to the root file system that originate from the BSP. Exclude this entry to accept the default and allow BSP packages. In the example above, this entry is only added to the **[glibc-small "image"]** entry, to keep the footprint small.

[rootfsName "distro"]

The **distro** entry specifies the following options as **wrlinux** only. Currently, there is nothing you can change in these entries.

compat

The default entry is **wrlinux**.

default

The default entry is **wrlinux**.

[rootfsName "vars"]

The optional **vars** entry, not shown in the example above, lets you specify layers and templates to be automatically included when you specify the rootfs during project configuration. Valid entries include:

layers

Enter each layer name, separated by a comma. Only the layer name is required, you do not have to specify the path. For example:

```
[rootfsName "vars"]  
  layers = meta-selinux,wr-intel-support
```

templates

Enter each template name as you would in the configure script command, for example:

```
[rootfsName "vars"]  
  templates = feature/debug,feature/analysis
```

About New Custom rootfs Configuration

The workflow for creating a custom rootfs includes the following:

1. Configure and build, or have a previously configured platform project available.
2. Copy the **projectDir/layers/local directory** from the platform project above to a location on your development system, and rename the layer.

For example, you could copy and name it to

/home/user/myLayer

3. Create a new **/home/user/myLayer/templates/rootfs.cfg** file, and populate it with the features that you want to include. For example:

```
[rootfs]  
  glibc-custom = image  
  
[glibc-custom "image"]  
  default-ktype = standard  
  compat = wrlinux-*  
  default = wrlinux-image-glibc-core  
  allow-bsp-pkgs = 0  
  
[glibc-custom "distro"]  
  default = wrlinux  
  compat = wrlinux  
  
[glibc-custom "vars"]  
  layers = meta-selinux,wr-intel-support  
  templates = feature/debug,feature/analysis
```

Depending on your project requirements, you can define a custom rootfs name, the kernel type, and add the layers and templates that you require.



NOTE: The kernel types, layers, and templates you include in this file must be part of your Wind River Linux installation. Missing layers and templates, or misspelled kernel type, layer, and template names, will return an error and halt the **configure** script.

4. Configure a new platform project, and:
 - Use the **--with-layer=** configure option to point to the new layer.

- Refer to the new name you gave your rootfs in the `--enable=rootfs=` configure option.

For example:

```
$ configDir/configure \  
--with-layer=/home/user/myLayer \  
--enable-board=gemux86-64 \  
--enable-rootfs=glibc-custom
```

5. Build the platform project, and verify that the options, such as layers, templates, and kernel types, are included in the build.

As long as you specify the location of the layer that contains the `rootfs.cfg` file, you can reuse the new custom rootfs in any platform project configuration.

EGLIBC File Systems

Embedded GLIBC (EGLIBC) is a variant of GLIBC which is designed with embedded systems in mind.

EGLIBC strives to be source and binary compatible with GLIBC with as few changes as possible. The goals of EGLIBC include reduced footprint, configurable components, and better support for cross-compilation.

See <http://www.eglibc.org> for details on the project.

The GLIBC package used in the Wind River Linux build system is, in fact, EGLIBC (eglibc 2.15).

EGLIBC can be configured to provide the smallest functional configuration of EGLIBC, while being comparable to the feature set and footprint uCLibc project (<http://www.uclibc.org>), while being more compatible with GLIBC. By default, the supported version of EGLIBC is a pre-built version that comes with the pre-built toolchain, and is Linux Standard Base (LSB)- and GLIBC-compatible.

About the EGLIBC Default Platform Project Configuration

Before you can build and deploy an EGLIBC-based file system, you must configure your platform project to enable EGLIBC features, once configuration is complete. With EGLIBC platform projects, there are two relevant variables located in the `projectDir/local.conf` file:

- The `DISTRO` = variable selects which distribution configuration file to use, which determines the default EGLIBC configuration. By default, the selection is set to `"wrlinux"`, which refers to the `projectDir/layers/wrlinux/conf/distro/wrlinux.conf` file.

To create your own custom EGLIBC configuration, you want to replace the `DISTRO="wrlinux"` with a custom configuration file. Wind River Linux provides the `projectDir/layers/wrlcompat/scripts/custom-distro.conf` file for you to use for this purpose. To use this file, you have two options:

Automated

Configure a new platform project using the `--with-custom-distro=distroName` option.

Manual

Copy it to your `projectDir/layers/wrlinux/conf/distro/` directory, rename it, and set your platform project to use this file.

For instructions on creating an EGLIBC platform project image, see [Creating and Customizing EGLIBC Platform Project Images](#) on page 86.



NOTE: While it is possible to modify and use the `wrlinux.conf` file for this purpose, it is not recommended. The `wrlinux.conf` file is used as a basis for all project configuration in Wind River Linux, and modifying it in this manner could cause your builds to fail and your platform projects to become corrupt.

- The `WRL_GLIBC_MODE` = variable, when used with the standard configuration, should be set to its default value of `"wrl-glibc-prebuilt"`. If you wish to change the EGLIBC configuration, it should be set instead to `"wrl-glibc-rebuild"`; this is set automatically by the `custom-distro.conf` file.

Creating and Customizing EGLIBC Platform Project Images

You can create an EGLIBC image at configure time or from an existing platform project image, and customize it using the information in this section.

To create and customize an EGLIBC platform project image:

Step 1 Select one of the following options for creating an EGLIBC platform project image:

Options	Description
New platform project	<p>Configure the platform project using the <code>--with-custom-distro=<i>distroName</i></code> option. For example, a minimal configure command might be:</p> <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_std \ --with-custom-distro=<i>distroName</i></pre> <p>Once the configure command completes, a custom distro configuration file with the name you specified as <code><i>distroName.conf</i></code> is created in the <code><i>projectDir</i>/layers/local/conf/distro</code> directory.</p>
Existing platform project	<ol style="list-style-type: none">1. Copy the <code><i>projectDir</i>/wrlcompat/scripts/custom-distro.conf</code> file to the <code><i>projectDir</i>/layers/local/conf/distro</code> directory and rename the file. In this example, we will change the name to <code>my-eglibc.conf</code>2. Edit and save the <code><i>projectDir</i>/bitbake_build/conf//local.conf</code> file to enable a custom EGLIBC build by changing the <code>DISTRO</code> variable to match the name of the <code><i>projectDir</i>/wrlcompat/scripts/custom-distro.conf</code> from the previous step. For example, if you named the file <code>my-eglibc.conf</code>, edit the variable as follows: <pre>DISTRO = "my-eglibc"</pre> <p>Notice that you do not need to specify the exact file location, or even use the <code>.conf</code> filename extension.</p>

Performing either of these actions enables the customization features for EGLIBC platform project images, and creates an alternate EGLIBC distro configuration file in your `projectDir/layers/conf/distro` directory for you to use to set your EGLIBC features.

Step 2 Optionally, choose the features to add to, or remove from, your EGLIBC distribution, by editing the `projectDir/layers/local/conf/distro/distroName` file's EGLIBC `DISTRO_FEATURES_LIBC` variables. Once you open the file, the default configuration includes the following options:

```
#
# These available eglibc features may be conditionally disabled simply
# by commenting out the lines below. Note that some packages may fail
# to build or fail to work correctly at runtime without required libc
# functionality.
#
# Note that there are dependencies between features that are not
# captured at this level. Needed features may be re-enabled even if
# they are commented out here. See
# layers/oe-core/meta/recipes-core/eglibc/eglibc-options.inc (and
# libc/option-groups.def in the eglibc source) for details.
#
# The selections below correspond to the features needed to build and
# boot the WRLinux "glibc-core" rootfs.
#
DISTRO_FEATURES_LIBC = ""
#DISTRO_FEATURES_LIBC += "ipv6"
#DISTRO_FEATURES_LIBC += "libc-backtrace"
#DISTRO_FEATURES_LIBC += "libc-big-macros"
DISTRO_FEATURES_LIBC += "libc-bsd"
#DISTRO_FEATURES_LIBC += "libc-cxx-tests"
#DISTRO_FEATURES_LIBC += "libc-catgets"
#DISTRO_FEATURES_LIBC += "libc-charsets"
DISTRO_FEATURES_LIBC += "libc-crypt"
DISTRO_FEATURES_LIBC += "libc-crypt-ufc"
#DISTRO_FEATURES_LIBC += "libc-db-aliases"
#DISTRO_FEATURES_LIBC += "libc-envz"
DISTRO_FEATURES_LIBC += "libc-fcvt"
#DISTRO_FEATURES_LIBC += "libc-fmtmsg"
#DISTRO_FEATURES_LIBC += "libc-fstab"
DISTRO_FEATURES_LIBC += "libc-fttraverse"
DISTRO_FEATURES_LIBC += "libc-getlogin"
#DISTRO_FEATURES_LIBC += "libc-idn"
DISTRO_FEATURES_LIBC += "ipv4"
#DISTRO_FEATURES_LIBC += "libc-inet-anl"
DISTRO_FEATURES_LIBC += "libc-libm"
DISTRO_FEATURES_LIBC += "libc-libm-big"
#DISTRO_FEATURES_LIBC += "libc-locales"
DISTRO_FEATURES_LIBC += "libc-locale-code"
#DISTRO_FEATURES_LIBC += "libc-memusage"
DISTRO_FEATURES_LIBC += "libc-nis"
#DISTRO_FEATURES_LIBC += "libc-nsswitch"
DISTRO_FEATURES_LIBC += "libc-rcmd"
DISTRO_FEATURES_LIBC += "libc-rtld-debug"
DISTRO_FEATURES_LIBC += "libc-spawn"
#DISTRO_FEATURES_LIBC += "libc-streams"
DISTRO_FEATURES_LIBC += "libc-sunrpc"
DISTRO_FEATURES_LIBC += "libc-utmp"
DISTRO_FEATURES_LIBC += "libc-utmpx"
#DISTRO_FEATURES_LIBC += "libc-wordexp"
DISTRO_FEATURES_LIBC += "libc-posix-clang-wchar"
DISTRO_FEATURES_LIBC += "libc-posix-regexp"
DISTRO_FEATURES_LIBC += "libc-posix-regexp-glibc"
DISTRO_FEATURES_LIBC += "libc-posix-wchar-io"
```

Using a text editor, uncomment and tailor the features include in your distribution. To specify individual items, refer to the mapping table at: [EGLIBC Option Mapping Reference](#) on page 88.



NOTE: This list of options has been tested to create a reasonably small footprint platform project file system. Not all option combinations are tested or supported. It is possible to uncomment features and create a platform project file system that does not function as a result. You may need to test feature option combinations as a result.

Step 3 Save the file if you made changes to it.

Step 4 Rebuild the platform project file system.

```
$ make
```

Once the build completes, the platform's system images and kernel will be located in the `projectDir/export` directory.

For information on testing your platform image on a target system, see [QEMU Targets](#) on page 241.

EGLIBC Option Mapping Reference

Use this reference to map components in the `option-groups.def` file to facilitate component selection for an EGLIBC build.

Feature to Map	Option to Map to
ipv4	OPTION_EGLIBC_INET
ipv6	OPTION_EGLIBC_ADVANCED_INET6
libc-big-macros	OPTION_EGLIBC_BIG_MACROS
libc-bsd	OPTION_EGLIBC_BS
libc-tests	OPTION_EGLIBC_CXX_TESTS
libc-catgets	OPTION_EGLIBC_CATGETS
libc-charsets	OPTION_EGLIBC_CHARSETS
libc-crypt	OPTION_EGLIBC_CRYPT
libc-crypt-ufc	OPTION_EGLIBC_CRYPT_UFC
libc-db-aliases	OPTION_EGLIBC_DB_ALIASES
libc-envz	OPTION_EGLIBC_ENVZ
libc-fcvt	OPTION_EGLIBC_FCVT
libc-fmtmsg	OPTION_EGLIBC_FMTMSG
libc-fstab	OPTION_EGLIBC_FSTAB
libc-straverse	OPTION_EGLIBC_FTRAVERSE

Feature to Map	Option to Map to
libc-getlogin	OPTION_EGLIBC_GETLOGIN
libc-idn	OPTION_EGLIBC_IDN
libc-inet-anl	OPTION_EGLIBC_INET_ANL
libc-libm	OPTION_EGLIBC_LIBM
libc-libm-big	OPTION_EGLIBC_LIBM_BIG
libc-locales	OPTION_EGLIBC_LOCALES
libc-locale-code	OPTION_EGLIBC_LOCALE_CODE
libc-memusage	OPTION_EGLIBC_MEMUSAGE
libc-nis	OPTION_EGLIBC_NIS
libc-nsswitch	OPTION_EGLIBC_NSSWITCH
libc-rcmd	OPTION_EGLIBC_RCMD
libc-rtld-debug	OPTION_EGLIBC_RTLD_DEBUG
libc-spawn	OPTION_EGLIBC_SPAWN
libc-streams	OPTION_EGLIBC_STREAMS
libc-sunrpc	OPTION_EGLIBC_SUNRPC
libc-utmp	OPTION_EGLIBC_UTMP
libc-utmpx	OPTION_EGLIBC_UTMPX
libc-wordexp	OPTION_EGLIBC_WORDEXP
libc-posix-clang-wchar	OPTION_POSIX_C_LANG_WIDE_CHAR
libc-posix-regex	OPTION_POSIX_REGEX
libc-posix-regex-glibc	OPTION_POSIX_REGEX_GLIBC
libc-posix-wchar-io	OPTION_POSIX_WIDE_CHAR_DEVICE_IO

6

Localization

About Localization 91

About Localization

Localization support allows you to develop projects for speakers of different languages.

Locales make geographic and language specific settings available to software users via its user interface. These include character set, number format, date and time formats, currency, collation rules, paper size, phone number format and others.

Wind River Linux provides varying levels of support for locales depending on the file system selected for your build. You can use the **locale** command to view details about the current locale.

See Also:

- <http://en.wikipedia.org/wiki/Locale>
for a general discussion of locales.
- http://www.loc.gov/standards/iso639-2/php/English_list.php
for a list of language codes.
- http://www.iso.org/iso/country_codes/iso_3166_code_lists/country_names_and_code_elements.htm
for a list of 2-digit country codes.

Determining which Locales are Available

Different file system options offer varying support for language locales. Knowing which are available allows you to plan internationalization support for your project.

A list of locales supported by your file system is provided in your project.

Step 1 View the list of supported locales in the file ***installDir/build/eglibc/eglibc-2.18/lib/localedata/SUPPORTED***

For example:

```
$ more installDir/build/eglibc/eglibc-2.18/libc/localedata/SUPPORTED
```

Output will look similar to the following:

```
# This file names the currently supported and somewhat tested locales.  
# If you have any additions please file a glibc bug report.  
SUPPORTED-LOCALES=  
aa_DJ.UTF-8/UTF-8 \  
aa_DJ/ISO-8859-1 \  
aa_ER/UTF-8 \  
aa_ER@saaho/UTF-8 \  
aa_ET/UTF-8 \  
af_ZA.UTF-8/UTF-8 \  
af_ZA/ISO-8859-1 \  
am_ET/UTF-8 \  
an_ES.UTF-8/UTF-8 \  
an_ES/ISO-8859-15 \  
ar_AE.UTF-8/UTF-8 \  
ar_AE/ISO-8859-6 \  
...  
en_US.UTF-8/UTF-8 \  
en_US/ISO-8859-1 \  
en_ZA.UTF-8/UTF-8 \  
en_ZA/ISO-8859-1 \  
en_ZM/UTF-8 \  
en_ZW.UTF-8/UTF-8 \  
en_ZW/ISO-8859-1 \  
es_AR.UTF-8/UTF-8 \  
es_AR/ISO-8859-1 \  
...
```

Step 2 Locate the locale code you want to support.

The typical format of a locale is `xx_XX`, where the first two characters represent the language and the second two represent the country. For example, `af_ZA` represents South African (ZA) Afrikaans (af). In a few cases language codes are three characters long.

Encoding is also indicated for each locale. For example, the entry:

```
es_AR.UTF-8/UTF-8 \  
...
```

indicates 8 bit Universal Transformation Format (UTF) support for Argentinian Spanish.

```
es_AR/ISO-8859-1 \  
...
```

indicates International Standards Organization 8859-1 support for Argentinian Spanish.

See the following for code look-up resources:

- http://www.loc.gov/standards/iso639-2/php/English_list.php
for a list of language codes.
- http://www.iso.org/iso/country_codes/iso_3166_code_lists/country_names_and_code_elements.htm
for a list of 2-digit country codes.
- <http://lh.2xlibre.net/locales/>
provides additional information about locales such as paper sizes, currencies, numeric formats, etc.

Setting Localization

Adding a locale to your project provides internationalization support for speakers of different languages.

Step 1 Add support for the locale to your *projectDir/local.conf* file.

For example, to add UTF-8 British English support, add:

```
GLIBC_GENERATE_LOCALES += "en_GB.utf8"  
IMAGE_LINGUAS = "en-gb.utf8"
```



NOTE: Observe the differences in case and use of underscore versus dash to construct the values for *GLIBC_GENERATE_LOCALES* and *IMAGE_LINGUAS*.

The += operator in the example above keeps us from preventing any default locales from being generated for glibc. To include additional locales in the image, use the += operator when assigning to *IMAGE_LINGUAS* as well.

Step 2 Rebuild the file system.

```
$ make fs
```

Support for the locale has been added when the build completes successfully.

7

Portability

[About Platform Project Portability](#) 95

[Copying or Moving a Platform Project](#) 96

[Updating a Platform Project to a New Wind River Linux Installation Location](#) 96

About Platform Project Portability

It is possible to move a platform project for comparison or development, or configure it for stand-alone portability.

Basic Portability

In this context, basic portability refers to the functionality included for moving or copying platform projects by default, with no special **configure** script options.

When you configure and build a platform project, the project's contents reside in the project directory (**projectDir**). Aside from the content in the **projectDir/layers/local** directory, much of the project contents are actually symbolic links to relevant git repositories located in the development environment. This creates a requirement for the development environment to know the location of the **projectDir**, so that it can populate the directories in alignment with the project configuration options.

Wind River Linux uses the `WRL_TOP_BUILD_DIR` variable to define the platform project's location, and make it possible to copy or move a platform project on the same build host to another location to meet your development needs. This variable is defined in the **projectDir/bitbake_build/conf/bblayers.conf** file. For additional information, see [Configuration Files and Platform Projects](#) on page 33.



NOTE: If you relocate the product install directory you must reconfigure any platform projects created using the original installation location. For additional information, see [Updating a Platform Project to a New Wind River Linux Installation Location](#) on page 96.

If you move a platform project, you must clear up all temporary files that comprise absolute paths. For additional information, see [Copying or Moving a Platform Project](#) on page 96.

Stand-alone Portability

In this context, stand-alone portability refers to a platform project that is not dependant on the Wind River Linux installation, or development environment, for project build and development.

To create a stand-alone platform project, add the **--enable-stand-alone-project=yes configure** script option when you configure your project.

When you configure a platform project with this option, the symbolic links between the development environment and *projectDir* are replaced with copies of the directories and files from the git repositories. This can consume a significant amount of disk space.

Copying or Moving a Platform Project

Learn how to copy or move a platform project.

The following procedure requires a previously configured platform project, or a project configured as a stand-alone project using the **--enable-stand-alone-project=yes configure** script option.

Step 1 Copy or move the top-level *projectDir* to a new location.

Step 2 Remove the *projectDir/bitbake_build/tmp* directory.

Run the following command from the new *projectDir* location:

```
$ rm -rf bitbake_build/tmp
```

This is required because the OE-core performs a sanity check to verify the physical location of the *projectDir/bitbake_build/tmp* directory. If it fails, the build process will halt, and you will receive an error message. Removing the directory causes the build system to update the path to the new location.

Step 3 Build the file system.

```
$ make
```

Updating a Platform Project to a New Wind River Linux Installation Location

Learn how to update a platform project when the Wind River Linux installation (*installDir*) changes.

The following procedure requires a previously configured platform project, and a new or changed *installDir*.

Step 1 Obtain the **configure** script command used to create the project.

This information is located in the `projectDir/config.log` file. For example:

```
configDir/configure --enable-board=qemux86-64 --enable-rootfs=glibc_small --enable-  
kernel=standard  
--enable-bootimage=iso
```



NOTE: In this example, `configDir` refers to the path to your Wind River Linux configure script, for example, `/home/user/WindRiver/wrlinux-5/wrlinux/`.

Step 2 Enable reconfiguration and rerun the configure script.

- a) Open a terminal window and navigate to the `projectDir`.
- b) Copy the `configure` script command from *Step 1* into the terminal.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-rootfs=glibc_small \  
--enable-kernel=standard \  
--enable-bootimage=iso
```

- c) Modify the path to the `configure` script to match the new Wind River Linux installation (`configDir`) location.
- d) Add the `--enable-reconfig=yes` option to the script command, and rerun it.

```
$ new_configDir/configure \  
--enable-board=qemux86-64 \  
--enable-rootfs=glibc_small \  
--enable-kernel=standard \  
--enable-bootimage=iso \  
--enable-reconfig=yes
```

The `configure` script will reconfigure the platform project to use the new installation location.

Step 3 Build the file system.

```
$ make
```

Once complete, the platform project will be symbolically linked to the new installation's location.

Step 4 Optionally, verify that the platform project is symbolically linked to the new installation location.

If the build from the previous step fails, or you want to verify the location of the git repositories that your platform project point to, perform this step.

Run the following command from the platform project directory:

```
$ ls -l git
```

The system should return:

```
$ lrwxrwxrwx 1 user user 40 Aug 16 17:29 git ->  
/home/user/new_installDir/wrlinux-5/git
```

where `new_installDir` represents the path to the new installation location.

8

Layers

About Layers	99
Layers Included in a Standard Installation	100
Installed Layers vs. Custom Layers	102
Layer Structure by Layer Type	103
About Layer Processing and Configuration	105

About Layers

Layers provide a mechanism for separating functional components of the development environment as described in this section.

Layers are multiple independent collections of recipes, templates, code, configuration files, and packages, typically contained in a layer directory. Multiple layers may be included in a single project, and each layer can provide any combination of features, ranging from kernel patches to new user space packages.

A layer allows the addition of new files, such as the recipes that define a specific package or packages, and machine configuration files that define a board for a new BSP, without modifying the original development environment. You can create your own layers and organize the content to better suit your development needs, and include or exclude the layers from the project configure and build.

In Wind River Linux, layers reside in the installation (development) environment and the build environment, in the platform project directory *projectDir*. When you configure and build a platform project image, the layers in the installation provide configuration information depending on you platform project configuration settings to configure your project.

Once the platform project configuration completes, a new set of layers specific to the platform project are created in the *projectDir/layers* directory.


The list and order of the platform project layers are maintained in the *projectDir/bitbake_build/conf/bblayers.conf* file. This file includes the list of layers used to create the target platform image.

Each layer has a **layerDir/conf/layer.conf** file that the BitBake build system uses to process the layer on project configuration and build. See [Configuration Files and Platform Projects](#) on page 33 .

Layers Included in a Standard Installation

The Wind River Linux standard installation provides a subset of layers as described in this section that are necessary for the build system and may also be used for development purposes.

See [Directory Structure for Platform Projects](#) on page 43 for a pictorial representation of the layer structure. In a default installation, these layers reside in the **projectDir/layers** directory, and include:

 **CAUTION:** The following layers are part of the Wind River Linux installation and are required by the build system. It is recommended that you do not modify these layers, as it may render your installation and development environment unusable. For additional information, see [Installed Layers vs. Custom Layers](#) on page 102.

 **CAUTION:** Depending on your platform project configuration options, some of these layers may not be present.

examples

Contains fully-working layer examples that provide functionality to a platform project image. These layer examples include:

fs-final

Provides an example of how a template can impact on the generation process of the file system.

hello-world

Adds the classic “Hello World” application to your platform project image.

kernel-config-example

Provides an example of how to use a template to change a global kernel parameter.

lemon_layer

Adds a simple multi-threaded web server called the `lemon_server` to your platform project image. You can use the examples provided in the **README** file located in this directory to use the `lemon_server` to analyze memory leaks and debug the `lemon_server` application.

To add the layer functionality described above to your platform project image, refer to the **README** file located in the directory containing the layer. For information on viewing **README** files, see [README Files in the Development Environment](#) on page 39 and [About README Files in the Build Environment](#) on page 56.

meta-downloads/

Contains copies of components referenced from external layers. The items are provided in a way to avoid having to download them from the network. An associated configuration file is also provided to inform the build system to use this as a pre-mirror.

meta-networking/

Contains networking-related packages and configuration. It should be useful directly on top of **oe-core** and compliments **meta-openembedded**.

meta-selinux/

Enables SELinux support when used with Poky. The majority of this layer's work is accomplished in **.bbappend** files, used to enable SELinux support in existing Poky packages.

meta-webserver/

Provides support for building web servers, web-based applications, and related software.

git/oe-core/

Contains the core metadata for current versions of OpenEmbedded. It is distro-less (can build a functional image with **DISTRO = ""**) and contains only emulated machine support. The Yocto Project has extensive documentation about OpenEmbedded included a reference manual, which can be found at <http://yoctoproject.org/community/documentation>.

oe-core-dl/

Contains downloaded packages and configuration files that comprise the package offerings from the Yocto Project. The **conf/layer.conf** file defines the mirror sites and order of locations that packages are retrieved from.

python-rhel5/

Contains Python binaries and source required by BitBake and Poky for the build process.

wr-base/

Contains the recipes and other configuration files that comprise the Wind River Linux base offering and make it possible to use the **configure** script to generate a platform project image. See *About the Configure Script* on page 63.

wr-bsps/

Contains recipes and machine configuration information for Wind River-supplied BSPs. See the *Wind River Linux Release Notes* for a list of available BSPs.

wr-features/

Provides many of the base components for Wind River Linux. It is required by most other layers that Wind River provides.

wr-fixes/

Provides bug fixes for other layers in the system. This layer requires **oe-core**.

wr-freescale_qoriq_dpaa/

Contains recipes and configuration information for using the Freescale User Space Datapath Acceleration architecture.

wr-installer/

Contains a target-based installed for Wind River Linux. This layer requires most of the layers that Wind River Linux provides.

wr-intel-support/

Contains software support in Wind River Linux for Intel technologies such as Intel QuickAssist, DPDH, and AMT.

wr-kernel/

Contains recipes and machine configuration information for Wind River-supplied kernels and kernel features. This includes a git repository for the Wind River Linux kernel and kernel tools.

wr-kvm-binary-guest-images/

Contains configuration information and directories for guest images.

wr-kvm-compile-guest-kernels/

Contains configuration information and directories for guest kernels.

wrlcompat/

In a typical Yocto Project build environment, the build output creates a specific directory structure. This structure is different than the Wind River Linux structure from previous releases. The wrlcompat layer ensures that build output is consistent with previous Wind River Linux (4.x) releases.

wrlinux/

Contains the recipes, configuration information, and files that support Wind River Linux tools and enhance development with the Yocto Project build system.

wr-simics/

Contains configuration information for using the Wind River Linux Simics system simulator.

wr-prebuilts/

Contains configuration information and files to support using pre-built binaries to create platform project images.

wr-toolchain/

Contains files, recipes, configuration information and documentation to support the GNU toolchain supplied by Wind River for development.

wr-toolchain-shim/

Provides configuration glue to allow selection of an automatically-integrated toolchain layer, which in turn contains both rules for building the toolchain from source, and rules for using the prebuilt binaries. This layer also contains tuning files and configuration overrides for those layers.

wr-tools-debug/

Contains configuration information and files to support debugging and ptrace with Workbench and Wind River tools.

wr-tools-profile/

Contains configuration information and files to support Wind River Linux development tools, including: analysis, boot time, code coverage, Valgrind, and lttng.

See [Templates in the Development Environment](#) for additional information.

Installed Layers vs. Custom Layers

When the installed layers do not meet your development needs, you can customize them or create new layers.

[Layers Included in a Standard Installation](#) on page 100 provides a description of the installed layers that comprise your Wind River Linux build system. These layers include support for creating and developing platform project images and applications with Wind River Linux, and also include support for add-on products, such as Wind River Simics.

It is possible to modify these layers, but is not recommended. If you discover that the supplied layers do not meet your development needs, you can create a custom layer that does. One example might be to add support for new hardware that is not included in the **wr-bsps** layer.

Custom layers let you add additional functionality and extend the capabilities of your development environment, as well as your build environment and the platform target you are developing for. The Wind River Linux build system makes it possible to do the following:

- Create a new, custom layer and include it by default with all new platform project image creation, or specify it for a single project.

See [Creating a New Layer](#) on page 106 for additional information.

- Extend the capabilities of an existing layer with append files (**.bbappend**) and save those extensions as a new layer.

See [About Recipes](#) on page 109 for additional information.

- Include or exclude layers as necessary to meet your development needs.

See [Enabling a Layer](#) on page 107 and [Disabling a Layer](#) on page 108 for additional information.

Layer Structure by Layer Type

Layers all have a similar structure, but include additional directories in their structure depending on their intended usage.

Layers can include any combination of recipes, templates, code, configuration files, and packages. In the Wind River Linux development environment, there are three specific layer types:

basic, or application-specific

This is used to manage applications and packages required by your project. For each newly configured and built platform project, Wind River Linux automatically creates a **projectDir/layers/local** layer for managing project-specific changes. See [About the layers/local Directory](#) on page 54 for additional information.

machine-specific

This is used to maintain BSP and kernel-related modifications and/or requirements.

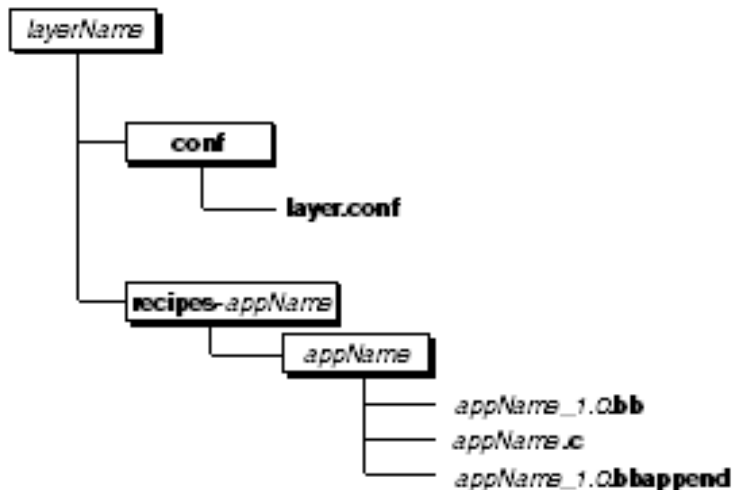
distribution-specific

This is used to maintain policies related to your platform project distribution (distro).

Layers make the development and build environments highly configurable. Layers are replaceable—if you have a different kernel layer, for example, you could specify it as an option to the **configure** script and override the default kernel layer.

Application-specific Layers

The following figure provides an example of a minimum layer requirements for an application-specific layer:

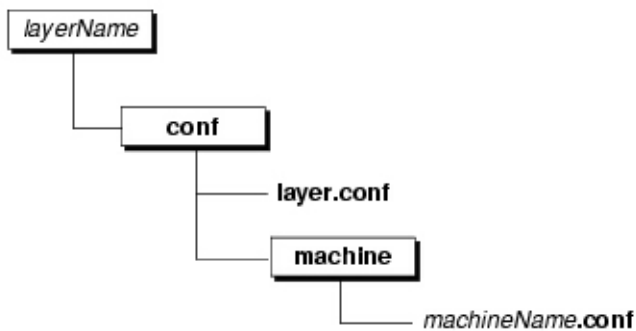


In this example, the layer includes a **conf/** directory with the **layer.conf** file, and a **recipes-appName** folder with a recipe file (**.bb**), application source (**.c**), and an append file (**.bbappend**) that extends the capability of an existing recipe file.

Note that the only minimum requirements for an application-specific layer are the **conf/layer.conf** file and a recipe file (**.bb**). You can organize the information and content in any manner you need to meet your development requirements.

Machine-specific Layers

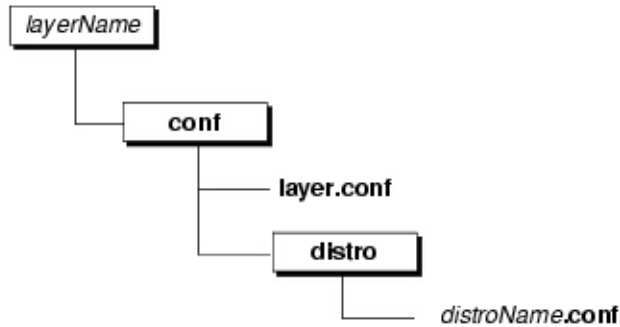
The following figure provides an example of a minimum layer requirements for a machine-specific layer:



A machine-specific layer requires the same **layerDir/conf/layer.conf** file as all layers do, but also includes the **layerDir/conf/machine/machineName.conf** file to differentiate this as a machine (BSP) layer type.

Distro-specific Layers

The following figure provides an example of a minimum layer requirements for a distro-specific layer:



A distribution-specific layer requires the same **conf/layer.conf** file as all layers do, but also includes the **conf/distro/*distroName.conf*** file to differentiate this as a distribution (distro) layer type.

- [Creating a New Layer](#) on page 106
- [Creating a Recipe File](#) on page 112

About Layer Processing and Configuration

Wind River Linux includes layers, and some optional products from Wind River are implemented as layers. In addition, you can create your own custom layers, and include layers created by others. This section describes how the build system configures layers in a hierarchical relationship.

When you create a project with the configure script, you do so using the available templates and layers. The configuration process creates a list of available layers, and then searches them to obtain any required templates. If a required template is not found, it is an error.

Layers provide templates and packages, while templates provide configuration. For example, a new package becomes available to the build system when you add it to a layer, but it only becomes part of a given project when you configure in the template that selects it. The template does not contain the package, it merely marks the package for inclusion.

The terms “higher” and “lower” are used to describe the priority layers or templates have. A higher-level template (or layer) takes precedence over a lower-level one, and is thus more specific, rather than less specific. When configure searches for components, it selects higher-level components first. When configure applies multiple components, it applies lower-level components first; this design allows higher-level components to override lower-level components.

For example, a kernel configuration fragment for a given BSP is at a higher level than the generic “standard” kernel configuration. The BSP-specific kernel configuration settings can then override more generic kernel configuration settings.

Combine layers with the configuration of templates, and the addition of the **fs_final*.sh** script, and the **changelist.xml** file located in the ***projectDir*/layers/local/conf/image_final** directory to build a complete run-time system that you provide the configuration details for.

About Processing a Project Configuration

Learn how a project configuration is processed when you run the **configure** and **make** commands.

In Wind River Linux, processing your project configuration occurs in the following manner:

1. You run the **configure** script to specify your target platform options. See [Introduction](#) on page 61. This script automatically includes certain default layers in your project configuration, along with any configure options that you specify.
2. You run the **make** command to build your project. This process takes into account all the metadata of your platform project, including default and specified layers, templates, and the recipes and **.conf** files that are associated with the metadata. For additional information, see:
 - [About Layers](#) on page 99
 - [Metadata](#) on page 32
 - [Configuration Files and Platform Projects](#) on page 33

Part of the build process is processing the **bblayers.conf** file (see [The bblayers.conf File](#)). Layers are processed from top to bottom in this file. Additionally, the **layers.conf** file has a variable called `BBFILE_PRIORITY`, which sets the processing priority of the layer, so that a developer can specify a higher priority to layers that other layers depend on.

3. After all of the layers and metadata are processed, and just before the actual file system image for the platform project is created, the build system processes the **fs_final*.xml** and the **changelist.xml** files. These files specify additional features or actions to be done on the target. See [Introduction to Managing Target Platforms](#).

Creating a New Layer

Learn how to create a layer for your platform project image.

- Step 1** Create a new layer directory.

The Yocto Project uses the meta- prefix (**meta-layerName**) for all layers, while Wind River uses the **wr-** prefix (**wr-bsps**, for example) to denote Wind River Linux-supplied layers.

- Step 2** Create a **conf/layer.conf** file in your layer directory.

You can simply copy an existing one (see [Configuration Files and Platform Projects](#) on page 33) and update it to reflect the new layer content, and also set the priority as appropriate.

- Step 3** Add layer-specific content, depending on the layer type.

For additional information, see [Layer Structure by Layer Type](#) on page 103.

Options	Description
Machine support	If the layer is adding support for a machine, add the machine configuration in conf/machine/
Distro policy	If the layer is adding distro policy, add the distro configuration in conf/distro/
New recipes	If the layer introduces new recipes, put the recipes you need in recipes-* subdirectories of the layer directory.

Enabling a Layer

To enable a layer, add your layer's path to the `BBLAYERS` variable in your `bblayers.conf` file .

Step 1 Open the `projectDir/bitbake_build/conf/bblayers.conf` file in an editor.

For additional information, see [Directory Structure for Platform Projects](#) on page 43.

Step 2 Add your layer's name to the `BBLAYERS` variable.

In this example, a new layer is added to the end of the list:

```

LCONF_VERSION = "5"

BBPATH = "${TOPDIR}"
BBFILES ?= ""
WRL_TOP_BUILD_DIR ?= "${TOPDIR}/.."
# resolve WRL_TOP_BUILD_DIR immediately with a canonical path
# to satisfy the bitbake logger
WRL_TOP_BUILD_DIR := "${@os.path.realpath(d.getVar('WRL_TOP_BUILD_DIR', True))}"

BBLAYERS = " \
  ${WRL_TOP_BUILD_DIR}/layers/wrlinux \
  ${WRL_TOP_BUILD_DIR}/layers/wrlcompat \
  ${WRL_TOP_BUILD_DIR}/layers/wr-toolchain \
  ${WRL_TOP_BUILD_DIR}/layers/oe-core/meta \
  ${WRL_TOP_BUILD_DIR}/layers/oe-core-dl \
  ${WRL_TOP_BUILD_DIR}/layers/meta-downloads \
  ${WRL_TOP_BUILD_DIR}/layers/wr-kernel \
  ${WRL_TOP_BUILD_DIR}/layers/wr-bsps/qemux86-64 \
  ${WRL_TOP_BUILD_DIR}/layers/wr-base \
  ${WRL_TOP_BUILD_DIR}/layers/wr-features \
  ${WRL_TOP_BUILD_DIR}/layers/wr-tools-profile \
  ${WRL_TOP_BUILD_DIR}/layers/wr-tools-debug \
  ${WRL_TOP_BUILD_DIR}/layers/meta-networking \
  ${WRL_TOP_BUILD_DIR}/layers/meta-webserver \
  ${WRL_TOP_BUILD_DIR}/layers/wr-prebuilts \
  ${WRL_TOP_BUILD_DIR}/layers/local \
  ${WRL_TOP_BUILD_DIR}/layers/myLayer

```

Step 3 Save the file.

Step 4 Reconfigure the platform project.

Run the following command from the platform project directory:

```
$ make reconfig
```

This command reconfigures your platform project to include the new layer changes. The build system parses each `conf/layer.conf` file as specified in the `BBLAYERS` variable. During the processing of each `conf/layer.conf` file located in the path of the layer, the build system adds the recipes, classes and configurations contained within the particular layer to your platform project image (see [Directory Structure for Platform Projects](#) on page 43). Once the command completes, the newly-updated target file system located in the `projectDir/export/dist` directory will include the newly added layer

Step 5 Rebuild the target file system.

```
$ make
```

After the command completes, the newly-updated target file system located in the `projectDir/export/dist` directory will include the newly added layer.

Disabling a Layer

To disable a layer, remove the associated path from the **bblayers.conf** file

Step 1 Open the *projectDir/bitbake_build/conf/bblayers.conf* file in an editor.

For additional information, see [Directory Structure for Platform Projects](#) on page 43.

Step 2 Comment out or remove the code line with the layer's name.

Step 3 Save the file.

Step 4 Reconfigure the platform project.

Run the following command from the platform project directory:

```
$ make reconfig
```

Step 5 Rebuild the target file system.

```
$ make
```

9

Recipes

[About Recipes](#) 109

[Creating a Recipe File](#) 112

[Identifying the LIC_FILES_CHKSUM Value](#) 113

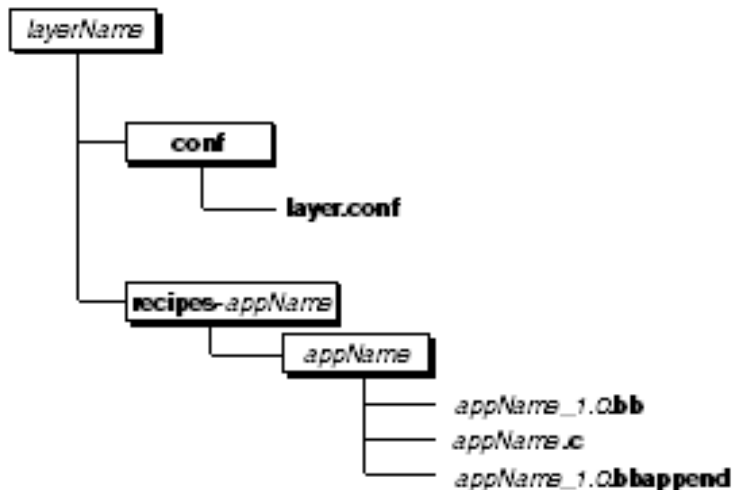
About Recipes

The Wind River Linux installation and build environments include recipes for use in specifying build instructions, similar to a Makefile.

A recipe file provides instructions for building required packages. It describes:

- library dependencies
- where to get source code
- what patches to apply
- dependencies on other recipes
- configuration and compilation options
- the logical sequence of execution events
- what software and/or images to build

Recipes use the **.bb** file extension, and are typically located in an a **recipes-*appName*** directory:



If you want to include your application or package in your platform project build, you must have a recipe file associated with it. You can copy and modify an existing recipe file, or create one from scratch.

See also:

- [Creating a Recipe File](#) on page 112
- [Metadata](#) on page 32
- [The Yocto Project Development Manual: Using .bbappend Files](#)
<http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html#using-bbappend-files>
- [The Yocto Project Wiki Recipe and Patch Style Guide](#)
https://wiki.yoctoproject.org/wiki/Recipe_%26_Patch_Style_Guide
- [Creating a Recipe File](#) on page 112
- [Metadata](#) on page 32

A Sample Application Recipe File

Learn how recipes work by examining a sample file.

In the *Wind River Linux Getting Started Guide*, the procedure for adding the Hello World (**hello.c**) sample application is explained. Once the application has been added to your platform project, the following recipe file is created automatically for you. See [About the layers/local Directory](#) on page 54 for the recipe file location in the platform project build environment.

```
DESCRIPTION = "This package contains the simple Hello World program."  
# PD = Public Domain license  
LICENSE = "PD" LIC_FILES_CHKSUM =  
"file://hello.c;beginline=1;endline=3;md5=0227db6a40baae2f3f41750645be145b"  
  
SECTION = "sample"  
  
PR = "r2"  
  
SRC_URI = "file://hello.c"  
  
S = "${WORKDIR}" do_compile() {  
    ${CC} ${CFLAGS} -o hello hello.c  
}  
do_install() {  
    install -d ${D}${bindir}
```

```
install -m 0755 hello ${D}${bindir}
}
```

Note that the `SRC_URI = element` defines the location of the application's source file. You would need to modify this location to match your own applications source, in relation to the location of the recipe file.

See [Creating a Recipe File](#) on page 112 for details on required recipe elements and licensing information.

About Recipe Files and Kernel Modules

When a recipe produces a kernel-module, it is necessary to configure the recipe to prefix the name of the generated packages with `kernel-module-`. The easiest way to do this is by specifying:

```
PKG_name = "kernel-module-name"
```

The following works in most cases, where only one module is generated in a package that has the same name as the recipe:

```
PKG_name = "kernel-module-${PN}"
```

In this example, `name` is the package name being generated by the recipe. If the package name already begins with `kernel-module-`, then no further changes are required.

Extending Recipes with .bbappend Files

You can extend, rather than replace, recipes using `.bbappend` files

It is not always necessary to recreate entire recipe files from scratch. Instead, you , providing that the original information (recipe, or `.bb` file) resides in an existing layer.



NOTE: `.bbappend` files do not just relate to recipes, they also relate to layers, and existing `conf/layer.conf`, `conf/machine/machineName.conf`, and `conf/distro/distroName.conf` files.

For an example of using a `.bbappend` file to extend kernel features, see [Configuring Kernel Modules With Fragments](#)

Step 1 Create `.bbappend` files to supplement an existing recipe file with new information.

For `.bbappend` files to work successfully, they must have the same name as the original recipe file. See [Layer Structure by Layer Type](#) on page 103 for an example of a recipe (`.bb`) filename and an append (`.bbappend`) filename for the same application.

Step 2 Configure the project.

Step 3 Build the project.

The build system compiles a list of `conf/local.conf` files, recipe files (`.bb`), and append (`.bbappend`) files, analyzes them, and creates the complete “picture” of your intended platform project image.

Creating a Recipe File

Learn how to create a recipe file for an application package in your platform project.

Recipe files are necessary if you want your application package to be included in your platform project image each time the project is configured, reconfigured, or built. See [About Recipes](#) on page 109.

After a new package is added to a layer in your platform project, it requires a recipe file to match the new package contents.

Step 1 Copy an existing recipe file into the same layer and directory where your package resides.

Step 2 Change the name of the recipe file to match the package name for the application.

For example, if your application is named **my-app.1.0**, name the recipe file to match it, **my-app_1.0.bb** for example.



NOTE: If you use the Package Importer tool to add a package to your platform project, it generates a recipe automatically for you. See [About the Package Importer Tool \(import-package\)](#) on page 144.

See [About Recipes](#) on page 109 for the required minimum recipe file contents.

Step 3 Open the recipe file in an editor.

```
$ vi layers/local/recipes-local/my-app/my-app_1.0.bb
```

Step 4 Update the md5 checksum to match your packages

a) Locate the following code line:

```
LIC_FILES_CHKSUM = "file://LICENCE.TXT;md5="
```

b) Update the md5 checksum to match your packages.

```
LIC_FILES_CHKSUM = "file://LICENSE;md5=f27defe1e96c2e1ecd4e0c9be8967949"
```



NOTE: If you do not know your packages md5 checksum value, see [Identifying the LIC_FILES_CHKSUM Value](#) on page 113.

Step 5 Change **my_bin** to match the name of the application.

a) Locate the following code line:

```
install -m 0755 ${S}/my_bin ${D}${bindir}
```

b) Change **my_bin** to match the name of the application, **my-app**.

Step 6 Save the file.

Step 7 Add your package to the platform project build.

```
$ make -C build my-app.addpkg
```


Step 8 Build the package.

```
$ make -C build my-app
```

The shell will display the build output. If you receive build errors for incorrect license checksum, see [Identifying the LIC_FILES_CHKSUM Value](#) on page 113.

In this case, you may need to perform steps 2 through 4 to update the `LIC_FILES_CHECKSUM` value.

Identifying the LIC_FILES_CHKSUM Value

Each time you add a new package to your platform project image either using the Package Importer tool or manually, you must update the package's recipe file `LIC_FILES_CHKSUM` to match the value of the package.

The syntax for the `LIC_FILES_CHKSUM` value is follows:

```
LIC_FILES_CHKSUM="file://license_info_location;md5=md5_value"
```

license_info_location

This is the name of the file that contains your license information. This could be a separate license file, the application's Makefile, or even the application's source file itself, for example, **my-app.c**.

md5_value

The numerical checksum value of the file called out in *license_info_location*.

When you add an application package to the system, or build a platform project that includes applications with recipe files, this value is checked, and returns a build failure if the md5 checksum value does not match the value that the build system expects.

If you do not know this value, or your build fails with the following warning, you must obtain the correct checksum value, and update the recipe's `LIC_FILES_CHKSUM` variable with it.

```
ERROR: Licensing Error: LIC_FILES_CHKSUM
```

- Choose an option to determine the `LIC_FILES_CHKSUM` value.

In this procedure, the examples reference a license file named **LICENSE** located in the **my-app** directory.

Options	Description
Use the md5sum command	<ol style="list-style-type: none"> 1. Run the md5sum command on the license file. <pre>\$ md5sum layers/local/recipes-local/my-app/LICENSE</pre> <p>The system returns the checksum value, for example:</p> <pre>2ebc7fac6e1e0a70c894cc14f4736a89 LICENSE</pre> 2. Enter the numerical value only in the md5= section. For example: <pre>LIC_FILES_CHKSUM = "file:// LICENSE;md5=f27defe1e96c2e1ecd4e0c9be8967949"</pre>

Options	Description
Use the build system	<ol style="list-style-type: none"><li data-bbox="599 258 1421 285">1. Run the make command to build the package recipe. <pre data-bbox="639 310 1421 359">\$ make -C build recipeName</pre><p data-bbox="639 380 1421 407">The build will fail. This is expected.</p><li data-bbox="599 415 1421 443">2. Scan the build output for the license checksum value. <p data-bbox="639 464 1421 491">For example:</p><pre data-bbox="639 516 1421 625">ERROR: my-app: md5 data is not matching for file:// LICENSE;md5=8e7a4f4b63d12edc5f72e3020a3ffae8 ERROR: my-app: The new md5 checksum is 2ebc7fac6e1e0a70c894cc14f4736a89</pre><ul data-bbox="639 646 1421 772" style="list-style-type: none"><li data-bbox="639 646 1421 709">- The first line states that the md5 checksum from the package's recipe file is incorrect.<li data-bbox="639 718 1421 772">- The second line provides the correct md5 checksum value. Use this value to update the LIC_FILES_CHKSUM md5= <i>value</i>.

10

Templates

[About Templates](#) 115

[Adding Feature Templates](#) 116

[Adding Kernel Configuration Fragments](#) 117

About Templates

The Wind River Linux installation and build environments include templates for specifying system-wide configuration values.

Templates are Wind River extension to the Yocto build system. The template feature is implemented in a bitbake compatible way, using Yocto compatible syntax, but you will not find templates in other Yocto based distributions.

By enabling one or more settings, templates can do the following:

- define the supporting files and/or packages required by a specific architecture
- add new functionality, such as a web server or debugging tools to your platform project image
- add or modify hardware support to the kernel or BSP
- many more, depending on your end-system requirements

At its most basic form, a template is simply a directory containing a collection of text configuration files. Templates can include other templates, be included in a layer, and also supplement an existing template or layer.

Wind River Linux provides feature and kernel templates to simplify development. Once you configure a platform project, templates are added to your platform project directories. See [Feature Templates in the Project Directory](#) on page 47 and [Kernel Configuration Fragments in the Project Directory](#) on page 51

A template may contain any of the following types of configuration files:

recipe (**.bb**) files

See [About Recipes](#) on page 109

class (**.bbclass**) file

A file that defines the inheritance of build logic as denoted by the class type, and packaging requirements.

configuration (**.conf**) files

See [Configuration Files and Platform Projects](#) on page 33

extension (**.bbappend**) files

See [About Recipes](#) on page 109.

In addition, the directory for template configuration files may contain:

- packages required by the template
- git repository
- source and binaries
- other files required by the feature

Adding Feature Templates

Use feature templates to add additional functionality to your platform project.

Use this example procedure to add feature templates when you initially configure a platform project using the **--with-template=** configure option.

Step 1 Configure the project to add the template.

In this example, the platform project will be configured and built with the **feature/debug** template.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-parallel-pkgbuilds=4 \  
--enable-jobs=4 \  
--with-template=feature/debug
```

To add multiple templates, append the additional template names:

--with-template=feature/templateName1, templateName2

The resulting configure command would look like:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-parallel-pkgbuilds=4 \  
--enable-jobs=4 \  
--with-template=feature/debug,feature/analysis
```

For additional information, see [Configure Options Reference](#) on page 66.

Step 2 Optionally, rebuild the platform project image.

```
$ make
```

For additional information on the available kernel configuration fragments, see [Feature Templates in the Project Directory](#) on page 47.

Adding Kernel Configuration Fragments

Use kernel configuration fragments, (similar to templates) to add additional functionality to your platform project.

You can add kernel configuration fragments when you initially configure a platform project using the `--enable-kernel=kernelType+type/templateName.scc` configure option.

The type in this example refers to the kernel option, either `cfg`, `features`, or `small`.

Step 1 Configure the platform project to include the kernel option.

In this example, the platform project will be configured and built with the `features/debugfs/debugfs-config.scc` template.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-rootfs=glibc_small \  
--enable-parallel-pkgbuilds=4 \  
--enable-jobs=4 \  
--enable-kernel=standard+features/debugfs/debugfs-config.scc
```

For additional information, see [Configure Options Reference](#) on page 66

Step 2 Optionally, rebuild the platform project image.

```
$ make
```

For additional information on the available kernel configuration fragments, see [Kernel Configuration Fragments in the Project Directory](#) on page 51.

11

Finalizing the File System Layout with changelist.xml

About File System Layout XML Files	119
About File and Directory Management with XML	119
Device Options Reference	120
Directory Options Reference	121
File Options Reference	122
Pipe Options Reference	123
Symlink Options Reference	123
The Touched/Accessed touch.xml Database File	124

About File System Layout XML Files

The file system layout feature has been designed to allow you to view the contents of the target file system in Workbench as it will be generated by the platform project build system.

You can add custom files to a file system. These files are then placed in the resulting filesystem RPM. See *Wind River Workbench by Example (Linux Version)* for using the File System Configuration Layout tool in Workbench to do the following:

- Examine file meta properties
- Add files and directories to the file system
- View parent packages and remove packages
- Add devices to `/dev` and change their ownership
- View files and directories that have been “touched” or accessed on the target

About File and Directory Management with XML

You can manage your file system with the `changelist.xml` file.

Managing Files and Directories with XML

The `projectDir/layers/local/conf/image_final/changelist.xml` file is an XML file that is managed by Workbench but can be edited or processed by command line tools. Opening the User Space Configuration tool in Workbench and making modifications creates the optional `changelist.xml` file. You can also create this file manually in the same location.

The script `projectDir/layers/wrlinux/scripts/fs_changelist.lua` processes this file immediately before the optional finalization script `fs_final.sh` as part of the `image-fs_finalize.bbclass` recipe (see [Options for Adding an Application to a Platform Project Image](#) on page 138). The file system finalization is recorded in the log found at `projectDir/bitbake_build/tmp/work/bsp-wrs-linux/wrlinux-image-filesystem-version/temp/do_rootfs/log.do_rootfs.timeStamp`. Any additions to the file system using `fs_final*.sh` and `changelist.xml` can be seen in `projectDir/export/dist` once the platform project image is built.

The `changelist.xml` file is processed in the pseudo environment, so the only named account and group that is valid is `root`. You should use numeric entries for alternate users and groups, and then subsequently create the accounts and groups in `fs_final.sh` with the numbers you used in the `changelist.xml` file, or with the `EXTRA_USERS_PARAMS`.

Device Options Reference

Any device added in `changelist.xml` in the default filesystem will be ignored. By default, Linux devices are now created in a dynamic file system, `devtmpfs`, mounted at runtime on `/dev`. As a result, any device nodes created statically are ignored.

The following tables show required and optional fields for adding a device to the system using the `filesystem/changelist.xml` file.



NOTE: Adding a device in this manner currently does not work.

Required Fields

Field and Value	Description
<code>action=addbdev</code> or <code>action=addcdev</code>	Use the <code>addbdev</code> action to add a block device. <code>umode</code> does not work with this action. Use the <code>addcdev</code> action to add a char device. <code>umode</code> does not work with this action.
<code>name=deviceName</code>	Name of the device directory added to the target file system
<code>umode=permissions</code>	The user/group/other permissions of the target symlink, in octal.
<code>major=majorNumber</code>	The major number for this device
<code>minor=minorNumber</code>	The minor number for this device

Optional Fields

Field and Value	Description
uid = <i>username</i>	The user name, in text or in numeric form (as with the chown command).
gid = <i>groupname</i>	The group name, in text or in numeric form (as with the chgrp command).

Examples

```
<cl action="addbdev" name="/usr/share/f_bdev1" major="3" minor="4" />
<cl action="addbdev" name="/usr/share/f_bdev2" major="3" minor="4" />
<cl action="addbdev" name="/usr/share/f_bdev3" major="3" minor="4" size="10" />
<cl action="addbdev" name="/usr/share/f_bdev4" major="3" minor="4" uid="user4"
gid="group4"/>
<cl action="addcdev" name="/usr/share/f_cdev1" major="1" minor="2" />
<cl action="addcdev" name="/usr/share/f_cdev2" major="1" minor="2" />
<cl action="addcdev" name="/usr/share/f_cdev3" major="1" minor="2" size="10" />
<cl action="addcdev" name="/usr/share/f_cdev4" major="1" minor="2" uid="user4"
gid="group4"/>
```

Directory Options Reference

Required and optional fields for adding a directory to the file system using the *filesystem/changelist.xml* file.

Required Fields

Field and Value	Description
action = adddir	Use the adddir action to add a directory to the file system
name = <i>dirname</i>	Name of the directory added to the target file system

Optional Fields

Field and Value	Description
umode = <i>permissions</i>	The user/group/other permissions of the directory, in octal, if the source= field is not present, to modify or override the permissions of an existing directory.
uid = <i>username</i>	The user name, in text or in numeric form (as with the chown command).
gid = <i>groupname</i>	The group name, in text or in numeric form (as with the chgrp command).

Examples

```
<cl action="adddir" name="/usr/share/f_dir1" />  
<cl action="adddir" name="/usr/share/f_dir2" umode="777" />  
<cl action="adddir" name="/usr/share/f_dir3" size="10" />  
<cl action="adddir" name="/usr/share/f_dir4" uid="1001" gid="1001" />
```

About The source= Field

If the **source=** field is not present, then a new empty directory is created on the target file system.

If the **source=** field is present, then the source directory name and attributes are copied from that source location. This command will not copy the contents of the source directory. Each file or sub-directory is expected to be iterated explicitly with the respective file or directory add directive.

File Options Reference

Required and optional fields for adding a file to the file system using the **filesystem/changelist.xml** file.

Required Fields

Field and Value	Description
action=addfile	Use the addfile action to add a file to the file system
name=filename	Name of the file added to the target file system
source=fullPath	The name and path of the file on the host file system. If present, the source file is to be copied into the target file system with the supplied permissions. If not present, then this entry is used to modify the permissions of an existing file.

Optional Fields

Field and Value	Description
umode=permissions	The permissions of the target file, in octal (as with the chmod command).
size=size	Where <i>size</i> is the pre-calculated size of the file used if the source= field is present, saving a size lookup by the tools that process this file (Workbench or command line tools).
uid=username	The user name, in text or in numeric form (as with the chown command).

Field and Value	Description
gid = <i>groupname</i>	The group name, in text or in numeric form (as with the chgrp command).

Examples

```
<cl action="addfile" name="/usr/share/f_foo1" source="/tmp/layout/f_foo1" />
<cl action="addfile" name="/usr/share/f_foo2" source="/tmp/layout/f_foo1" umode="777" />
<cl action="addfile" name="/usr/share/f_foo3" source="/tmp/layout/f_foo1" size="10" />
<cl action="addfile" name="/usr/share/f_foo4" source="/tmp/layout/f_foo1" uid="1001" gid="1001" />
```

Pipe Options Reference

Required and optional fields for adding a pipe to the file system using the **filesystem/changelist.xml** file.

Required Fields

Field and Value	Description
action = addpipe	Use the addpipe action to add a pipe to the file system
name = <i>pipeName</i>	Name of the directory added to the target file system

Optional Fields

Field and Value	Description
uid = <i>username</i>	The user name, in text or in numeric form (as with the chown command).
gid = <i>groupname</i>	The group name, in text or in numeric form (as with the chgrp command).

Examples

```
<cl action="addpipe" name="/dev/f_pipe1" />
<cl action="addpipe" name="/dev/f_pipe2" />
<cl action="addpipe" name="/dev/f_pipe3" size="10" />
<cl action="addpipe" name="/dev/f_pipe4" uid="user4" gid="group4"/>
```

Symlink Options Reference

Required and optional fields for adding a symlink to the file system using the **filesystem/changelist.xml** file.

Required Fields

Field and Value	Description
action=addsymbk	Use the addsymbk action to add a symlink to the file system
name=symlinkName	Name of the symlink added to the target file system
target=targetName	Name of the target within the target file system.

Examples

```
<cl action="addsymbk" name="/usr/share/f_syml" target="/usr/share/f_fool" />  
<cl action="addsymbk" name="/usr/share/f_syml2" target="/usr/share/f_fool" />  
<cl action="addsymbk" name="/usr/share/f_syml3" target="/usr/share/f_fool" size="10" />  
<cl action="addsymbk" name="/usr/share/f_syml4" target="/usr/share/f_fool" uid="user3" gid="group3" />
```

The Touched/Accessed touch.xml Database File

This section provides information on the XML for the file system layout **touched.xml** file. You can custom-generate and import such a formatted file using any name.

There is only one entry definition and one action (*action="touched"*). If a file name is in this list, then that file was directly or indirectly touched or accessed on the target. This file is generated by a script that runs on the target, and by default populated with all files touched since first boot.

```
<?xml version="1.0" encoding="UTF-8"?>  
<layout_change_list version="1">  
  <change_list>  
    <cl action="touched" name="/bin"/>  
    <cl action="touched" name="/bin/busybox"/>  
    <cl action="touched" name="/fetch-footprint.sh"/>  
    <cl action="touched" name="/lib"/>  
    <cl action="touched" name="/lib/libnss_files-2.8.so"/>  
    <cl action="touched" name="/lib/libc-2.8.so"/>  
    <cl action="touched" name="/lib/ld-2.8.so"/>  
    <cl action="touched" name="/etc"/>  
    <cl action="touched" name="/etc/fstab"/>  
    <cl action="touched" name="/etc/init.d/rcS"/>  
    <cl action="touched" name="/etc/profile"/>  
    <cl action="touched" name="/etc/passwd"/>  
    <cl action="touched" name="/etc/inittab"/>  
    <cl action="touched" name="/etc/nsswitch.conf"/>  
    <cl action="touched" name="/touched.xml"/>  
  </change_list>  
</layout_change_list>
```

See and the *Wind River Linux Workbench by Example, Linux Version* for more information on the use of this tool.

PART III

Userspace Development

Developing Userspace Applications.....	127
Understanding the User Space and Kernel Patch Model.	153
Patching Userspace Packages.....	159
Modifying Package Lists.....	167

12

Developing Userspace Applications

Introduction	127
Creating a Sample Application	132
Exporting the SDK	136
Adding Applications to a Platform Project Image	138
Importing Packages	144
Listing Package Interdependencies	151

Introduction

About Application Development

You can use the Wind River Linux SDK to develop applications and cross-compile them for the intended target.

The Wind River Linux SDK is a development environment which provides all the necessary tools to cross-compile and deploy your application in the intended target. The SDK is generated from a working Wind River Linux Platform project and is therefore associated with the particular architecture and build options of the project.

In a typical scenario you cross-compile your application and run it on an emulator such as QEMU first to verify its core functionality. As your development process proceeds, it is likely that you will need to cross-compile and deploy the application binaries to a NFS root file system, which is mounted by the real target, to proceed with further testing. Finally you integrate your application binaries with the build system so that they are automatically deployed on the target's image.



NOTE: Creating a build environment on the target does install some GPLv3 software. If you want to produce a GPLv2 target with the new applications you developed, you could build your applications on the target, and then add the binaries to subsequent builds that do not include the build tools.

Also note that while Wind River does support the target compiler for product development, it does not support the compiler on shipped products.

See also:

- *Wind River Workbench by Example, Linux Version*

Cross Development Tools and Toolchain

Use the GNU toolchain to cross-compile applications for your target system.

Wind River Linux 5 is based on the Yocto Project (<http://www.yoctoproject.org>) implementation of the OpenEmbedded Core (OE-Core) metadata project. The Yocto Project uses build recipes and configuration files to define the core platform project image, as well as the applications and functionality it provides.

This build system uses metadata contained in the recipes and configuration files to define and create a Linux kernel and a root file system with all necessary configuration and initialization files for a deployed Linux platform. You can add or remove source RPM and traditional tar archive packages for customized solutions. You can also add or remove RPM binary packages from the target file system, automatically checking dependencies, flagging missing libraries, components, or version mismatches. The build system provides a version-controllable development environment, separate from the host file system which is protected from inadvertent damage.

Cross-development is supported by the inclusion of the GNU cross-toolchain, and enhanced by the addition of Wind River Workbench.



NOTE: The build toolchain required to cross-compile programs for your target system is located in the `projectDir/host-cross` directory. See *Directory Structure for Platform Projects* on page 43.

Workbench supports kernel mode debugging through the Kernel GNU Debugger (KGDB), and user mode debugging through the `ptrace` agent.

For detailed information on using Wind River Workbench, see the *Wind River Workbench User's Guide*, and the *Wind River Workbench by Example, Linux Version*.

About Sysroots and Multilibs

Application developers use sysroots provided by the platform developer to build applications.

Once the application is built, it can be incorporated into platform project images.

What are Sysroots?

Wind River Linux provides sysroots, which are a prototype target directory which contains the necessary library, header, and other files as they would appear on the target. Sysroots also include toolchain wrappers for each of the supported development hosts.

In general pre-built libraries and toolchain wrappers are not provided for application development because they may not well reflect the actual platform prepared by the platform developer. Instead, the platform developer generates and exports a sysroot for the configured platform project.

Architecture-specific sysroots are generated with the **make export-sdk** command. See [Exporting the SDK](#) on page 136. To use these sysroots, you must generate the SDK, uncompress it, and source the sysroot file as described in that section.

Once sourced, these sysroots enable application developers to get started developing for the platform target configuration.

About Exporting Sysroots

To produce an exported sysroot environment from a configured build directory, use the **make export-sdk** command as described in [Exporting the SDK](#) on page 136.

What are Multilibs?

CPU and multilib templates encode information about a particular supported target, such as compiler flags that are needed or desirable when building for that target. These templates exist so that you should never have to manually specify any CPU-specific or architecture-specific compiler flags to build software correctly for a BSP.

A CPU template defines specific performance tuning flags for a given CPU. A multilib template defines the flags which control ABI compatibility. It is possible to have several CPU templates which share one multilib template.

Multilib Targets

Wind River supports multiple libraries on certain targets. With these multilib targets, it is possible, for example, to compile an application against both 32- and 64-bit libraries, and not just one or the other.

In cases where a board supports multilibs, a reasonable default library has been chosen, but you may need a different library. For example, `qemux86-64` targets may include the `x86_64` or `x86_32` CPU types, with `x86_64` being the default. If you want to provide for development with the `x86_32` CPU type on a `qemux86-64` target, you need to take additional action to be sure the appropriate packages are included in the sysroot you export.

Enabling Multilib Support in Platform Projects

Learn how to enable multilib support for your platform project.

Multilibs are enabled by adding the **MULTILIB=** declaration to your project's **local.conf** file (see [Configuration Files and Platform Projects](#) on page 33).

The potential multilibs are listed in the **AVAILTUNES** declaration. For `qemu86-64` BSP, that value equals the following.

```
AVAILTUNES=" x86 x86-64 x86-64-x32"
```

In this example, **x86** (32-bit), **x86-64** (64-bit), and **x86-64-x32**, a combination of both, are available tuning options for multilib support.



NOTE: Not all available tunings are necessarily supported in projects, for example **x86-64-x32**.

To perform this procedure, you must have a previously configured platform project.

Step 1 Open the **projectDir/local.conf** file in an editor.

Step 2 Add the **MULTILIBS** variable to enable 32-bit support.

The value that you enter must be listed in the **AVAILTUNES** declaration. For example, for the **qemux86-64** BSP you would add:

```
MULTILIBS = "multilib:lib32"  
DEFAULTTUNE_virtclass-multilib-lib32 = "x86"
```

Step 3 Save the file.

The platform project now supports the addition of 32-bit packages.

Step 4 Optionally add packages to the file system.

For additional information, see [Adding Multilib Packages](#) on page 130.

Step 5 Build the platform project image.

```
$ make
```

Adding Multilib Packages

Once you have enabled multilib support in a platform project, you can add multilib packages.

This procedure requires that you have previously configured a platform project that has been enabled to support multilib packages. For additional information, see [Enabling Multilib Support in Platform Projects](#) on page 129.

Step 1 Navigate to the **projectDir**.

```
$ cd projectDir
```

Step 2 Add the package.

```
$ make -C build lib32-zlib.addpkg
```

Step 3 Build the package.

```
$ make -C build lib32-zlib.build
```

The build system will automatically add the multilib variants of the dependencies for this package, for example the **lib32-glibc** and other libraries, and include them in the final file system image. All packages for all default and alternate multilibs will have links in the project's build directory, using the naming convention to distinguish the variants.

Step 4 Optionally verify the architecture (32- or 64-bit) of the package.

Perform this step to view the flags for the package that determine the architecture.

```
$ make -C build lib32-zlib.env | grep "^TARGET_CC_ARCH="
```

The system provides the following output:

```
TARGET_CC_ARCH="-m32"  
$
```

In this example, the `TARGET_CC_ARCH="-m32"` flag indicates a 32-bit package.

Adding Multilib Support for All Libraries to the SDK

Use this procedure to add multilib support to a SDK.

Multilib libraries are not installed to the SDK by default when you build a platform project and generate the SDK. This is because the target file system does not typically require any multilib images, and adding additional libraries that increase the memory footprint, but cannot run on the target file system, is not considered a good use of resources. However, it may be necessary to add multilib support for a specific development need, as described in the following procedure.



NOTE: Performing this procedure will only add support to the SDK for use on the development host, and not to the target file system.

Step 1 Update the `projectDir/local.conf` file.

Add the following lines and save the file.

```
TOOLCHAIN_TARGET_TASK = "task-core-standalone-sdk-target task-core-standalone-sdk-  
target-dbg"  
TOOLCHAIN_TARGET_TASK += "${@} '.join([x + "-" + y for y in  
    ((d.getVar("LIBC_DEPENDENCIES", True) or "") +  
     ' libgcc libgcc-dev libstdc++ libstdc++-dev').split() for x in  
    (d.getVar('MULTILIB_VARIANTS',  
             True) or "").split())}]}"
```

The first line is the default setting, and the second line adds the specific multilib libraries to the image.

Step 2 Create the SDK.

```
$ make export-sdk
```



NOTE: For additional information on creating SDK images, see [Exporting the SDK](#) on page 136.

After the command finishes processing, it creates a shell script for extracting the SDK in the `projectDir/export/` directory. The exact name includes components of the project settings. Look for a file name that ends with `-sdk.sh`, for example:


```
projectDir/export/wrlinux-5.0.1.0-glibc-x86_64-qemux86_64-wrlinux-image-glibc-small-  
sdk.sh
```

Adding Static Library Support to the SDK

Use this procedure to add static library support to a SDK.


When you build the file system and generate the SDK, the SDK does not include support for static libraries. This is intentional in that most applications should be using the shared libraries for linking. If you require specific static libraries for your development needs, you can add a single static library, or all static libraries.

Step 1 Choose an option for adding static libraries.

Options	Description
Single library	Update the <i>projectDir/local.conf</i> file with the following information: <pre>TOOLCHAIN_TARGET_TASK_append = " lib_recipeName"</pre> <hr/> <p> NOTE: The leading space prior to <i>lib_recipeName</i> is required by the build system.</p> <hr/> <p>In this example, <i>lib_recipeName</i> refers to the recipe name for the specific static library.</p>
All static libraries	Update the <i>projectDir/local.conf</i> file with the following information: <pre>SDKIMAGE_FEATURES = "staticdev-pkgs dev-pkgs dbg-pkgs"</pre>

Step 2 Save and close the file.

Step 3 Generate the SDK.

 **NOTE:** For additional information on creating SDK images, see [Exporting the SDK](#) on page 136.

```
$ make export-sdk
```

After the command finishes processing, it creates a shell script for extracting the SDK in the *projectDir/export/* directory. The exact name includes components of the project settings. Look for a file name that ends with *-sdk.sh*, for example:

projectDir/export/wrlinux-5.0.1.0-glibc-x86_64-qemux86_64-wrlinux-image-glibc-small-sdk.sh

Creating a Sample Application

Learn to develop a sample C application that calculates a specified number of terms in the Fibonacci series.

The Fibonacci series takes the number of terms as a command-line parameter.

This application provides a good example of a typical C application, and includes the following:

- Two source files
- A header file
- A Make file to provide guidance for building the final binary and clean up the working directory
- A license file for association with the platform project

This example shows how to work with applications that are not part of your existing platform project image, and do not automatically build when you run the **make** command in the platform project directory. You can use this procedure for any stand-alone application that you want to develop and test apart from your platform project image.

If you wish to add an application project to your platform project image that builds automatically with each subsequent project build, you can use the Package Importer tool. See [About the Package Importer Tool \(*import-package*\)](#) on page 144.

To develop and compile an application to match your platform project's architecture, you must first export the SDK and source the sysroot. See [Exporting the SDK](#) on page 136.



NOTE: If you have previously exported the SDK, you can simply **source** the associated **env.sh** file located in the sysroot directory.

Step 1 Create a working directory for your application project.

You can create a working directory anywhere on your host workstation. There are no restrictions for location or directory name. In this example, you will create a directory in your platform project's directory. For example:

```
$ mkdir ~/Builds/qemux86-64_small/Fibonacci
$ cd ~/Builds/qemux86-64_small/Fibonacci
```

Step 2 Set up the **main.c** file.

a) Create the **main.c** file with **vi**.

```
$ vi main.c
```

b) Enter or copy the following text.

```
/*
 * Copyright 2012 Wind River Systems, Inc.
 */

#include <stdio.h>
#include "math.h"

int main(int argc, char *argv[])
{
    int i, count=0;

    if (argc >= 2)
        count = atoi(argv[1]);

    for (i=0; i < count; i++) {
        printf("%d\n", fibonacci(i));
    }

    return 0;
}
```

c) Save the file.

Step 3 Set up the **math.c** file.

a) Create the **math.c** file with **vi**.

```
$ vi math.c
```

b) Enter or copy the following text.

```
/*
 * This is public domain software
```

```
*/  
  
int fibonacci(int n)  
{  
    if (n <= 0)  
        return 0;  
    else if (n == 1)  
        return 1;  
    else  
        return (fibonacci(n-1) + fibonacci(n-2));  
}
```

c) Save the file.

Step 4 Create the **math.h** file.

a) Create the header **math.h** file with vi.

```
$ vi math.h
```

b) Enter or copy the following text.

```
/*  
 * This is public domain software  
 */  
  
int fibonacci(int n);
```

c) Save the file.

Step 5 Create the **makefile** file.

a) Create the header **math.h** file with vi.

```
$ vi makefile
```

b) Enter or copy the following text.

```
#  
# Copyright 2012 Wind River Systems, Inc.  
#  
VERSION := 1.0  
DEPS := math.h  
SRC := main.c math.c  
OBJ := $(SRC:.c=.o)  
  
all: fibonacci  
  
archive: fibonacci.tar.bz2  
  
fibonacci: $(OBJ)  
    $(CC) -o $@ $^ $(LIBS)  
  
%.o: %.c $(DEPS)  
    $(CC) -c -o $@ $<  
  
fibonacci.tar.bz2: $(SRC) $(DEPS) Makefile LICENSE  
    tar --transform 's,^,fibonacci-${VERSION}/,' -cjf $@ $^  
  
.PHONY: clean  
  
clean:  
    @rm -f fibonacci *.o *~
```

c) Save the file.

Step 6 Create the **LICENSE** file.

a) Create the **LICENSE** file with vi.

```
$ vi LICENSE
```

- b) Enter or copy the following text.

```
The Fibonacci application is licensed under the GPL v3.  
For the purposes of this example we keep these statements in this file  
but you should include the full text of the license here instead.
```

- c) Save the file.

Step 7 Compile the program.

This step requires that you already created the SDK and sourced the development environment, as detailed in [Exporting the SDK](#) on page 136.

- a) Navigate to the project directory for the application.
b) Compile the application.

```
$ make
```

After the **make** command completes, it creates a **fibonacci** binary file in the working directory, compiled to match the sourced development environment.

Step 8 Optionally, test the program on the host.

If your host workstation is compatible with the target architecture of your platform, you can run the program with the following command:

```
$ ./fibonacci 5
```

The program should output:

```
0  
1  
1  
2  
3
```

If your host workstation is not compatible, go to the next step to test your application on the target.

Step 9 Test the program on the target.

- a) Copy the program binary to the platform project target file system **usr/bin** directory.

```
$ fibonacci ~/Builds/qemux86-64_small/export/dist/usr/bin
```

- b) Navigate back to the platform project directory.

```
$ cd ..
```

- c) Launch the platform project in an emulator.

```
$ make start-target
```

- d) After the system finishes booting, log in.

Step 10 Optionally, add your application project to the platform project image

This step is recommended if you plan to include your application as part of your platform project image. See [Options for Adding an Application to a Platform Project Image](#) on page 138.

Exporting the SDK

Exporting the SDK

After you have successfully configured and built a platform project, you can export the SDK for application development.

The following procedure requires that:

- You have a previously configured and built platform project—see [Introduction](#) on page 61 and the *Wind River Linux Getting Started Guide: Developing a Platform Project Image Using the Command-line*.
- You have read/write privileges to the `/opt` directory on your host workstation.



NOTE: These privileges are necessary for the project toolchain to install and work properly.

Step 1 Generate the SDK.

- a) Navigate to the platform project directory that you wish to create the SDK for.
- b) Create the SDK.

```
$ make export-sdk
```

After the command finishes processing, it creates a shell script for extracting the SDK in the `projectDir/export/` directory. The exact name includes components of the project settings. Look for a file name that ends with `-sdk.sh`, for example:

`projectDir/export/wrlinux-5.0.1.0-glibc-x86_64-qemux86_64-wrlinux-image-glibc-small-sdk.sh`

This script will install the SDK with the toolchain for your project that you need for cross-compiling applications. Note that the file is named after the architecture and file system of your configured project. This example uses a qemux86-64 BSP and glibc_small root file system.

Step 2 Run this script to install the SDK.

In this example, you first navigate to the `export` directory, and then install the SDK.

```
$ cd export
$ ./wrlinux-5.0.1.0-glibc-x86_64-qemux86_64-wrlinux-image-glibc-small-sdk.sh
```

Step 3 Enter the target directory path, or accept the default `/opt/windriver/wrlinux/5.0-qemux86-64`.

Step 4 When prompted, press `Y` and `ENTER` to install the SDK.

The SDK will extract and install to the directory specified in step 3 on page 136. Once complete, the required environment variables for the target architecture are set up to allow you to immediately begin application cross-compiling and development. Specifically, the environment variables `CC`, `CXX`, and `CFLAGS` are set to cross-compile C and C++ programs using the corresponding cross-compilers.

Step 5 Rebuild the file system to incorporate the SDK for development.

Run the following command from the **projectDir**:

```
$ make fs
```

Exporting the SDK for Windows Application Development

After you have successfully configured and built a platform project, you can export the SDK for application development on a Windows host.

Use the following instructions create and extract a SDK suitable for developing applications on a Microsoft Windows host.

Step 1 There are three possibilities for generating the SDK:

Option	Description
Configure a new project using the <code>--enable-sdk-wins=yes</code> configure option.	<ol style="list-style-type: none">1. Enter the following command from the projectDir to configure the platform project:<pre>\$ configDirconfigure \ --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_std \ --enable-win-sdk=yes \ --enable-parallel-pkgbuilds=4 \ --enable-jobs=4</pre>2. Once the configuration completes, create the SDK.<pre>\$ make export-sdk</pre>
Override <code>EXPORT_SYSROOT_HOSTS</code> in the projectDir/local.conf file. See Configuration Files and Platform Projects on page 33	<ol style="list-style-type: none">1. Edit the platform project's local.conf file. With a previously configured platform project, open the projectDir/local.conf file and edit the EXPORT_SYSROOT_HOSTS option as follows, and save the file:<pre>EXPORT_SYSROOT_HOSTS ?= "x86-linux2 x86-win32"</pre>2. Once complete, create the SDK.<pre>\$ make export-sdk</pre>
Override <code>EXPORT_SYSROOT_HOSTS</code> on the command line.	<p>With a previously configured platform project, enter the following EXPORT_SYSROOT_HOSTS the projectDir:</p> <pre>\$ make export-sdk EXPORT_SYSROOT_HOSTS="x86-linux2 x86-win32"</pre>

After the command finishes processing, it creates an archive in the **projectDir/export/** folder containing the SDK. The exact name includes components of the project settings, such as the architecture and file system. Look for a file name that ends with a **.zip** extension. For example:

projectDir/export/wrlinux-5.0.1.0-glibc-x86_64-qemux86_64-wrlinux-image-glibc-small-sdk-win32.zip

This compressed file contains the toolchain for your project that you need for cross-compiling applications.

Step 2 Copy the **.zip** file containing the SDK to your Windows host.

Step 3 On the Windows host, extract the SDK ***.zip** file.

You can extract all of your SDKs into the same parent directory, as each core directory is unique and named after the project architecture name and file system.

Step 4 Import the SDK in Workbench.

From the Workbench main menu, select **File > Import > Wind River Linux > Import SDK**, then click **Next**.

The Import Wind River Linux SDK window opens.

Step 5 Navigate to the location of the SDK you extracted previously, and select the **sysroots** sub-directory.

Once you have selected the directory, the SDK Information fields will populate with related information.

Step 6 Click **Finish** to import the SDK.

Once the SDK import is complete, the build spec is made available in Workbench for application development. See the *Wind River Workbench by Example (Linux version)* for additional information.

Adding Applications to a Platform Project Image

Options for Adding an Application to a Platform Project Image

There are several options available for you to add an application to an existing platform project image.

The following options are available in Wind River Linux for adding an application to your platform project image:

Use the **make** command

This option lets you specify the addition of a single package using the **make** command. For example:

```
$ make -C build recipeName.addpkg
```

For information on using this command, see [Adding New Application Packages to an Existing Project](#) on page 139.

Import the application tree as a package

This option imports the application tree as a package, and creates a recipe file for it, thereby including it as part of the platform project image. See [About the Package Importer Tool \(import-package\)](#) on page 144.

Use a **fs_final*.sh** script

This option automatically installs the application's binary to the root file system each time the platform project is built. While the application is automatically built and installed in the root file system, it is not linked to the platform project. See [Adding an Application to a Root File System with fs_final*.sh Scripts](#) on page 141.

Use the **changelist.xml** file

This option automatically adds the application, but the **changelist.xml** file has many features that might be useful, depending on your platform project requirements. See [Adding an Application to a Root File System Using **changelist.xml**](#) on page 140 and [About File System Layout XML Files](#) on page 119.

Use the **configure** command

This option adds the package to the platform project image when you configure, or reconfigure the platform project image. See [Configuring a New Project to Add Application Packages](#) on page 142.

Adding New Application Packages to an Existing Project

Learn how to add the **gdb** package to an existing, previously configured and built, platform project.

The following procedure provides instructions to add **gdb** to an existing, previously configured and built, platform project. This procedure uses the following example **configure** script command to create the platform project that the **gdb** package will be added to:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small
```

This example assumes that you do not already have **gdb** included with your platform project.

For additional information, see: [Introduction to Configuring and Building Platform Projects](#).

Step 1 Add the **gdb** package to the platform project build.

a) Navigate to the *projectDir*.

```
$ cd projectDir
```

b) Add the **gdb** package.

```
$ make -C build gdb.addpkg
```

The system will return the following output:

```
make: Entering directory `./Builds/qemux86-64_small/build'  
==Checking ../layers/local/recipes-img/images/wrlinux-image-glibc-small.bb==  
==Checking for valid package for gdb==  
...  
=== ADDED gdb to ../layers/local/recipes-img/images/wrlinux-image-glibc-small.bb ===
```

Package changes like this are added to the *projectDir*/**layers/local/recipes-img/images/wrlinux-image-file-system.bb** recipe file, where *file-system* represents the name of the root file system used to configure the platform project.

c) Verify that **gdb** was added successfully.

```
$ cat layers/local/recipes-img/images/wrlinux-image-glibc-small.bb
```

The system will return the following output, after the line that declares **#### END Auto Generated by configure ####**:

```
#### END Auto Generated by configure ####
```

```
IMAGE_INSTALL += "gdb"
```

This indicates that the package will be included in the build.

Step 2 Build the **gdb** package.

```
$ make -C build gdb
```

Building the package takes a couple of minutes, during which you will see the progress on your terminal.

Step 3 Rebuild the root file system.

```
$ make
```

Rebuilding the file system should take just a couple of minutes this time, because only the newly added elements need to be built.

Step 4 Verify that the package was added successfully.

See [Verifying the Project Includes the New Application Package](#) on page 143.

Adding an Application to a Root File System Using changelist.xml

After you have created an application, you can use the **changelist.xml** file to add it to a platform project image root file system.

If you place a file named **changelist.xml** in the **projectDir/layers/local/conf/image_final** directory, then the contents of the file are executed after all the other root file system packages have been processed, but before the final root file system's tar file is created. Though this section provides instructions for adding an application to a platform project root file system, there are many more possibilities with the **changelist.xml** file. See [Managing Files and Directories with XML](#).

The location of the script file is inside the **projectDir/layers/local** directory (see [About the layers/local Directory](#) on page 54). This location is meant to simplify development by keeping your project configuration changes and additions in one location.

This approach provides the option of running commands that impact the target file system only, and not the platform project build.

In the following procedure, you will create a **changelist.xml** file to add the Fibonacci binary created in [Creating a Sample Application](#) on page 132.

Step 1 Create the **changelist.xml** file.

- a) Navigate to the platform project top-level directory.

```
$ cd ~Builds/qemux86-64_small
```

- b) Create the **changelist.xml** file with **vi**.

```
$ layers/local/conf/image_final/changelist.xml
```

- c) Enter or copy the following text.

```
<?xml version="1.0" encoding="UTF-8"?>
<layout_change_list version="1">
<change_list>
<cl action="addfile" name="/usr/bin/fibonacci"
source="/home/revo/Builds/qemux86-64_small/Fibonacci/fibonacci"/>
</change_list>
```

```
</layout_change_list>
```



NOTE: In the example above, substitute `/home/revo/Builds/qemux86-64_small` for the location of your platform project directory.

If your application project has more specific requirements, such as setting permissions, and so on, you can add those requirements to your script file. See *Managing Files and Directories with XML* for additional information.

d) Save the file.

Step 2 Rebuild the root file system.

Run the following command from the platform project directory:

```
$ make
```

After the project rebuilds, the `changelist.xml` file will run automatically to update the `fibonacci` binary.

Step 3 Verify that the binary was added successfully.

Note that there are no provisions in the XML syntax to run arbitrary commands, such as can be done with `fs_final*.sh` scripts (see *Adding an Application to a Root File System with `fs_final*.sh` Scripts* on page 141). The result is that your binary — in this example, the `fibonacci` binary — must exist prior to rebuilding the root file system.

```
$ ls export/dist/usr/bin/fibonacci
```

If the `fibonacci` binary exists, the system displays:

```
export/dist/usr/bin/fibonacci
```

Adding an Application to a Root File System with `fs_final*.sh` Scripts

After you have created an application, you have a number of options to add it to the root file system of your platform project image.

When you place a script file named `fs_final*.sh` (`fs_final_script_use.sh`) in the `projectDir/layers/local/conf/image_final` directory, the contents of the script are executed after all the other root file system packages have been processed, but before the final root file system's tar file is created.

The location of the script file is inside the `projectDir/layers/local` directory (see *About the layers/local Directory* on page 54). This location is meant to simplify development by keeping your project configuration changes and additions in one location.

This approach provides the option of running script commands that impact the target file system only, and not the platform project build.

In the following procedure, you will create a `fs_final*.sh` script for the `fibonacci` binary created in *Creating a Sample Application* on page 132, to add the application to the target's root file system.

Step 1 Create the `fs_final*.sh` script for the application.

- a) Navigate to the platform project top-level directory. For example:

```
$ ~Builds/qemux86-64_small
```

- b) Create the **fs_finalfibonacci.sh** script with **vi**.

```
$ vi layers/local/conf/image_final/fs_final_fibonacci.sh
```

- c) Enter or copy the following text.

```
#  
# Add the fibonacci binary to the root file system  
#  
make -C /home/revo/Builds/qemux86-64_small/Fibonacci  
cp /home/revo/Builds/qemux86-64_small/Fibonacci/fibonacci usr/bin
```



NOTE: In the example above, substitute **/home/revo/Builds/qemux86-64_small** for the location of your platform project directory.

Take a deeper look at what the script accomplishes:

- Line 1 (begins with **make**) builds the project
- Line 2 (begins with **cp**) copies the binary to the **/usr/bin** directory of the target system.
Note that this is a single line of code, and is only split in this example for display purposes.

If your application project has more specific requirements, such as setting permissions, and so on, you can add those requirements to your script file.

- d) Save the file.

Step 2 Rebuild the root file system.

Run the following command from the platform project directory:

```
$ make
```

After the project rebuilds, the **fs_final_fibonacci.sh** script will run automatically to update the **fibonacci** binary.

Step 3 Verify that the binary was added successfully.

- a) List the contents of the platform project root file system.

From the project directory, enter:

```
$ ls export/dist/usr/bin/fibonacci
```

The system should return the following output to confirm the **fibonacci** binary exists:

```
export/dist/usr/bin/fibonacci
```

Configuring a New Project to Add Application Packages

Learn how to add a package to a project at configure time.

The following procedure illustrates how to add the **gdb** (GNU Debugger) to a project.

➔ **NOTE:** The functionality (the target-resident debugger) added in this example is currently only supported on targets with the x86 architecture (32- and 64-bit), but the workflow for adding a non-default layer and templates is the same with all architectures and BSPs.

Step 1 Configure the project.

To configure a `glibc_small` platform project that includes the `gdb` package, use the `--with-package=gdb` configure option. For example:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-parallel-pkgbuilds=4 \  
--enable-jobs=4 \  
--with-package=gdb
```

➔ **NOTE:** To add an application to a previously configured platform project, add the `--enable-reconfigure` option to your configure command.

In addition to the standard configuration arguments of a board, kernel, and file system, this configuration adds the `gdb` package.

When you configure a project with a specific package, that package is added to the platform project, and available for use once the project is built and deployed to a target.

Step 2 Build the project

Enter the `make` command:

```
$ make
```

This will take from minutes to hours, depending on your development resources. When it is finished, you will have a kernel and file system that includes `gdb`.

Step 3 Verify that the package was added successfully.

See [Verifying the Project Includes the New Application Package](#) on page 143

Verifying the Project Includes the New Application Package

Learn how to verify that a package was added successfully to the platform project.

The procedure in this section illustrates how to verify that a package was added successfully to the platform project.

This procedure assumes you previously added the `gdb` package from [Configuring a New Project to Add Application Packages](#) on page 142 or [Adding New Application Packages to an Existing Project](#) on page 139.

Step 1 Verify that the `gdb` is available now by looking at the generated root file system.

From the project directory, enter:

```
$ ls export/dist/usr/bin/gdb
```

The system should return the following output to confirm that the **gdb** executable exists:

```
export/dist/usr/bin/gdb
```

Step 2 Verify that the **gdb** command is available on the running target.

a) Deploy the platform project on a target.

For additional information, see *Using QEMU from the Command Line*.

b) Run the **gdb** command on the target:

```
# gdb
```

The system should return the following and display the (**gdb**) prompt to confirm **gdb** is working:

```
GNU gdb (Wind River Linux Sourcery CodeBench 4.6-60) 7.2.50.20100908-cvs
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-wrs-linux-gnu".
For bug reporting instructions, please see:
<support@windriver.com>.
(gdb)
```

c) Use the debugger or type **quit** to return to the command prompt.

Importing Packages

About the Package Importer Tool (*import-package*)

Use the Package Importer tool to add application packages in various forms to your platform project image.

For concepts and information on the interface for the Package Importer tool, see the *Wind River Workbench by Example, Linux 5 version: About the Package Importer Tool (import-package)*.

There are three approaches to importing a package detailed in this guide:

- Importing an existing sample project
- Importing a source package from the web (**wget**)
- Importing a SRPM package from the web

Importing a Sample Application Project as a Package

Use this Package Importer Tool example procedure to learn how to import a Wind River sample application package.

In this procedure, you will learn to import the **pthread** sample application.

This procedure requires a previously configured and built platform project. If you do not have an existing platform project, this procedure was created using the following **configure** options:

```
$ configDir/configure \
--enable-board=qemux86-64 \
```



```
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-parallel-pkgbuilds=4 \  
--enable-jobs=4
```

Step 1 Launch the Package Importer tool.

```
$ make import-package
```

The GUI of the tool displays and the Progress field displays the *installDir* and *projectDir* locations in the host file system.

For additional information, see the *Wind River Workbench by Example, Linux 5 version: About the Package Importer Tool* (import-package).

Step 2 Import the package contents.

The following sub-steps illustrate importing an application's source tree, or directory, to a package in your platform project.

- a) Set the **Package Type** selection to **Application Source Tree**.
- b) Update the Package Location field.

In the Package Location field, enter:

```
$ installDir/wrlinux-5/samples/mthread
```

- c) Click **Update**.

The Package Name, Package Version, and Progress fields automatically populate.

- d) Click **Import** to import the package to your platform project.

The Progress field will indicate that the import is successful.

- e) Click **Close** to close the Package Importer tool.

Step 3 Update the recipe file license checksum and build the package.

After a new package is imported into your platform project, you must update the recipe file to match the new package contents.

- a) Open the recipe file for the package.

In the project directory, open the recipe file located at: *projectDir/layers/local/recipes-local/mthread/mthread_1.0* in an editor. For example:

```
$ vi layers/local/recipes-local/mthread/mthread_1.0.bb
```

- b) Locate the following code line:

```
LIC_FILES_CHKSUM = "file://LICENCE.TXT;md5="
```

This value is checked at build time, and will cause a build error if it is not correct. As a result, you need to obtain it.

- c) Update the *LIC_FILES_CHKSUM* value.

For purposes of this example, the **mthread** application does not include a Makefile, so this example uses the application's name:

```
LIC_FILES_CHKSUM = "file://mthread.c;md5=numerical_checksum_value".
```

- d) Change **my_bin** to match the name of the application.

For example, change:

```
install -m 0755 ${S}/my_bin ${D}${bindir}
```

to:

```
install -m 0755 ${S}/mthread ${D}${bindir}
```

- e) Update the compiler options.

This step is only required for applications that do not have a Makefile in the application tree. Just before the line that reads # **You must populate the install rule**, enter the following lines code in the recipe file:

```
do_compile() {  
    ${CC} ${CFLAGS} -lpthread -o mthread mthread.c  
}
```

The **-lpthread** option is required for multi-threaded packages, and must be placed prior to the output (**-o**) option.

- f) Save the file.
g) Build the package.

```
$ make -C build mthread
```

The shell displays the build output. If you receive a build error for an incorrect license checksum, see [Identifying the LIC_FILES_CHKSUM Value](#) on page 113 to obtain the packages md5 checksum value.

- h) Update the md5 checksum value.

After you have the new md5 checksum value, perform [3.c](#) on page 145, above, again, this time entering the new md5 checksum in the `LIC_FILES_CHECKSUM` value. For example:

```
LIC_FILES_CHKSUM = "file://Makefile;md5=2ebc7fac6e1e0a70c894cc14f4736a89"
```

- i) Save the file.
j) Rebuild the package.

```
$ make -C build mthread
```

With the correct md5 checksum, the package should build successfully.

Step 4 Verify that the package was added to the build.

```
$ ls build/mthread-1.0-r0/image/usr/bin/
```

If the package was added to the build, the list of files includes the **mthread** file.

Importing a Source Package from the Web (wget)

Use this Package Importer Tool example procedure to learn how to import a source application package.

In this procedure, you will learn to import the **bc** application package. Note that the **bc** package is used as an example only, and that you may use this procedure to import other source packages required for your project.

This procedure requires a previously configured and built platform project. If you do not have an existing platform project, this procedure was created using the following configure options:

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_small \
--enable-parallel-pkgbuilds=4 \
--enable-jobs=4
```

Step 1 Launch the Package Importer tool.

From the project directory, enter the following:

```
$ make import-package
```

The GUI of the tool displays and the **Progress** field displays the *installDir* and *projectDir* locations in the host file system.

See the *Wind River Workbench by Example, Linux 5 version: About the Package Importer Tool* (import-package), for additional information.

Step 2 Import the package contents.

- a) Set the **Package Type** selection to **Source Package**.
- b) Update the Package Location field.

In the Package Location field, enter <http://autobuilder.yoctoproject.org/pub/sources/bc-1.06.tar.gz>, then click **Update**.

The update tool adds the values of the Package Name, Package Version, and Progress fields.

- c) Click **Import**.
The Progress field displays the progress of the update.
- d) Click **Close** to close the Package Importer tool.

Step 3 Update the recipe file and build the package.

- a) Open the recipe file in an editor.

In the platform project directory, open the recipe file located at *projectDir/ayers/local/recipes-local/bc/bc_1.06.bb* in an editor. For example:

```
$ vi layers/local/recipes-local/bc/bc_1.06.bb
```

- b) Locate the following code line:

```
LIC_FILES_CHKSUM = "file://LICENCE.TXT;md5="
```

This value is checked at build time, and will cause a build error if it is not correct. As a result, you need to obtain it.

- c) Modify the **LIC-FILES_CHKSUM** value.

From:

```
LIC_FILES_CHKSUM = "file://LICENCE.TXT;md5="
```

to:

```
LIC_FILES_CHKSUM = "file://COPYING;md5="
```

For purposes of this example, the application was retrieved from the web, and does not include a **LICENSE.txt** file. Instead, this application uses the **COPYING** file, which stores the license information for all the source code files for the application. This is why you changed the name from **LICENSE.TXT** to **COPYING**.

In addition, you are leaving the md5 checksum value empty. This will cause the build to fail in the next step, but that is okay. When it does, the build system will provide the correct md5 checksum value to enter here.

- d) Build the package.

```
$ make -C build bc
```

The shell displays the build output. Since we left the checksum value empty, you will receive a build error for an incorrect license checksum. Scan the build output for the correct checksum value. See [Identifying the LIC_FILES_CHKSUM Value](#) on page 113.

- e) Update the `LIC_FILES_CHKSUM` value.

Enter the md5 value the build system provides in the previous step, for example:

```
LIC_FILES_CHKSUM = "file://COPYING;md5="94d55d512a9ba36caa9b7df079bae19f"
```

- f) Modify the `#inherit` code line:

Locate the line that reads:

```
#inherit autotools
```

and remove the comment character (`#`) so that the line reads:

```
inherit autotools
```

- g) Change the `install` code line.

Locate the line that reads:

```
install -m 0755 ${S}/my_bin ${D}${bindir}
```

and change it to:

```
install -m 0755 ${S}/bc/bc ${D}${bindir}
```

- h) Build the package.

```
$ make -C build bc
```

The shell displays the build output. With the correct md5 checksum, the package builds successfully.

Step 4 Verify the package was added to the build.

```
$ ls build/bc-1.06-r0/image/usr/bin/
```

Importing a SRPM Package from the Web

Use this Package Importer Tool example procedure to learn how to import a source RPM (SRPM) package.

If you have standardized on SRPM for your integration of applications into Wind River Linux, this procedure provides a way to migrate that infrastructure into Wind River Linux. SRPMs were the preferred package format in Wind River Linux 4.x. In Wind River Linux 5.0.1, we suggest that you use whatever format the upstream source provides, which is typically not SRPM.

In this procedure, you will learn to import the **dos2unix** SRPM package and integrate it into your platform project. Note that the **dos2unix** package is used as an example only, and that you may use this procedure to import other SRPM packages required for your project.

This procedure requires a previously configured and built platform project. If you do not have an existing platform project, this procedure was created using the following configure options:

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_small \
--enable-parallel-pkgbuilds=4 \
--enable-jobs=4
```

Step 1 Launch the Package Importer tool.

From the project directory, enter the following:

```
$ make import-package
```

Note that the GUI of the tool displays and the Progress field displays the *installDir* and *projectDir* locations in the host file system.

See the *Wind River Workbench by Example, Linux 5 version: About the Package Importer Tool* (import-package), for additional information.

Step 2 Import the package contents.

- a) Set the **Package Type** selection to **Source Package**.
- b) Update the Package Location field.

In the Package Location field, enter **ftp://ftp.muug.mb.ca/mirror/fedora/linux/development/20/source/SRPMS/d/dos2unix-6.0.3-3.fc20.src.rpm** then click **Update**.

The update tool adds the values of the Package Name, Package Version, and Progress fields.

- c) Click **Import**.
- The Progress field displays the progress of the update.
- d) Click **Close** to close the Package Importer tool.

Step 3 Update the recipe file and build the package.

- a) Open the recipe file in an editor.

In the platform project directory, open the recipe file located at *projectDir/layers/local/recipes-local/dos2unix/dos2unix_6.0.3-2.fc19.bb* in an editor. For example:

```
$ vi layers/local/recipes-local/dos2unix/dos2unix_6.0.3-2.fc19.bb
```

- b) Append the **SRC_URI** line with the name of the embedded tar ball in the SRPM package.

From:

```
SRC_URI = http://www.your_company_here.com/downloads/
dos2unix-6.0.3-2.fc19.src.rpm;extract="
```

to:

```
SRC_URI = http://www.your_company_here.com/downloads/  
dos2unix-6.0.3-2.fc19.src.rpm;extract=${BPN}-6.0.3.tar.gz"
```

- c) Confirm that the **S=** macro definition also matches the embedded tar ball.

In this example, the default is correct: and does not require updating.

```
S="${WORKDIR}/${BPN}-6.0.3"
```

- d) Save the recipe file.
e) Build the package:

```
$ make -C build dos2unix
```

The shell displays the build output. Since we left the `LIC_FILES_CHKSUM` md5 checksum value in the recipe file empty, you will receive a build error for an incorrect license checksum. Scan the build output for the correct checksum value. See [Identifying the LIC_FILES_CHKSUM Value](#) on page 113.

- f) Update the `LIC_FILES_CHKSUM` value.

Enter the md5 value the build system provides in the previous step, for example:

```
LIC_FILES_CHKSUM = "file://Makefile;md5=1ba513be50142c911e7971a6f4d47e89"
```

- g) Save the file.
h) Enter the following command to build the package:

```
$ make -C build dos2unix
```

The shell displays the build output. With the correct md5 checksum, the package builds successfully.

Step 4 Verify the package was added to the build.

To view the packages source files, from the command line, enter:

```
$ ls build/dos2unix-6.0.3-2.fc19-r0/dos2unix-6.0.3  
  
bcc.mak      dos2unix.c   mingw.mak    test         wccdos16.mak  
BUGS.txt    dos2unix.h   NEWS.txt     TODO.txt     wccdos32.mak  
Changelog.txt dos2unix.o   po          unix3dos     wcc.mif  
common.c    emx.mak     pod2htmd.tmp unix2dos.c   wcc.mif  
common.h    INSTALL.txt pod2htmli.tmp unix2dos.h   wccwin32.mak  
common.o    mac2unix     qerycp.c     unix2dos.o  
...
```

Step 5 Optionally update the `projectDir/layers/local/recipes-local/dos2unix/dos2unix_6.0.3-2.fc19.bb` recipe file.

This step ensures that other basic information about the package is correct.

For additional information, see the *Wind River Linux Migration Guide: Updating a BitBake Recipe from a SRPM *.spec File*.

Step 6 Optionally relocate the package.

If you want to make the package available to other platform projects, you can move it from the local layer at `projectDir/layers/local/recipes-local/dos2unix` to a custom layer directory. If you

choose to move the package's directory, you must also relocate the `projectDir/layers/local/downloads/dos2unix/dos2unix-6.0.3-2.fc19.src.rpm` source package that it processes.

Listing Package Interdependencies

The `list-packageconfig-flags` utility displays interdependencies between packages.

Many of the packages compiled by the bitbake build system have interdependencies. If one package is present another is required, or is compiled differently. It can be difficult to find these interdependencies, as the package recipes can be spread over several files in multiple layers. In order to help you understand the interdependencies a script is available to list the package configuration of the current project. It is not necessary to build the project to observe these dependencies.

The script runs in the bitbake build environment, so in order to use it you must first enter the bitbake shell.

Step 1 Start the bitbake shell.

In the base of your platform project enter:

```
$ make bbs
```

Step 2 Run the script.

```
$ ../layers/oe-core/scripts/contrib/list-packageconfig-flags.py -a -p | less
```

The command options in this example list preferred versions of all packages. Run `list-packageconfig-flags.py` with the `-h` option for a full list of options:

```
Usage: list-packageconfig-flags.py [-f|-a] [-p]
       list available pkgs which have PACKAGECONFIG flags

OPTION:
  -h, --help      display this help and exit
  -f, --flag      list available PACKAGECONFIG flags and all affected pkgs
  -a, --all       list all pkgs and PACKAGECONFIG information
  -p, --prefer    list pkgs with preferred version
```



NOTE: Running the script with no arguments outputs basic package dependency information similar to what can be obtained with `make fs-expand` for the `IMAGE_INSTALL` contents.

The script may take several minutes to parse the recipe information before giving any output.

From the output in this example we can observe that the `cups-1.6.3` recipe is built differently because the `acl` and `avahi` packages are present in the project.

```
Setting up packages link
Creating export directory
Creating project properties
Parsing recipes..done.
=====
cups-1.6.3
/opt/Builds/glib_std/layers/oe-core/meta/recipes-extended/cups/cups_1.6.3.bb
PACKAGECONFIG None
PACKAGECONFIG[avahi] --enable-avahi,--disable-avahi,avahi
PACKAGECONFIG[acl] --enable-acl,--disable-acl,acl

lib32-quagga-0.99.21
virtual:multilib:lib32:/opt/Builds/glib_std/layers/meta-networking/recipes-protocols/
quagga/quagga_0.99.21.bb
```

```
PACKAGECONFIG None
PACKAGECONFIG[cap] --enable-capabilities,--disable-capabilities,libcap

lib32-coreutils-8.21
virtual:multilib:lib32:/opt/Builds/glib_std/layers/oe-core/meta/recipes-core/coreutils/
coreutils_8.21.bb
PACKAGECONFIG None
PACKAGECONFIG[acl] --enable-acl,--disable-acl,acl,

gtk+3-3.8.2
/opt/Builds/glib_std/layers/oe-core/meta/recipes-gnome/gtk+/gtk+3_3.8.2.bb
:
```


13

Understanding the User Space and Kernel Patch Model

[Patch Principles and Workflow](#) 153

[Patching Principles](#) 154

[Kernel Patching with scc](#) 155

Patch Principles and Workflow

Understanding the patch workflow will help you resolve patch-related issues if and when they arise.

Understanding the patch principles and workflow used for development helps facilitate changes to your project as they arise. This section discusses the workflow from a command line perspective, using command line tools. For an example of how to use the Workbench patch manager GUI, refer to *Wind River Workbench by Example, Linux Version*.

There are two main principles Wind River Linux uses in applying patches:

- Wind River Linux keeps its source code pristine. Patches are only applied to project code, when building a project.
- Patch lists are rigorously maintained.

Patch workflow for Wind River developers follows this pattern:

1. Product designers first decide on where—which template or layer—to insert the patch.
2. The individual developer configures a project for the specific product, specifying the relevant layer or template in the configure command.
3. The individual developer then works locally, developing new code and new patches to extend existing code.
4. The developer then validates their local work with the central code base before folding back changes and patches. The more general the layer in which the patch is placed, the greater the scope of testing required to justify the acceptance of these changes. Automated test tools and procedures for the individual contributor help in keeping the code base correct.
5. After successful validation, the developer checks in the changes.

Patch Deployment

Kernel patches and package patches can be deployed in:

- custom layers
- custom templates
- the installed development environment.

Wind River suggests that custom patches be deployed within a custom template or layer, thereby leaving the development environment intact. For more information and examples, see [About Kernel Configuration and Patching](#) on page 177 and [Introduction to Patching Userspace Packages](#) on page 159

Patching Principles

Learn about patching and patch troubleshooting principles.

Patch Application and Resolution

During patch development, apply patches within a project created for that purpose.

Simple Reject Resolutions

Simple reject resolutions include resolving path names, fuzz factor, whitespace, and patch reverse situations.

Some hunk rejects can be resolved by simple adjustments, including:

Leading Path Names

The leading path directory names in the patch may not match the directory names of your targets. By removing some or all of the patch's leading path names, you may then match the local environment.

Fuzz Factor

Each hunk has a leading and following number of lines around changes to provide a validating context for the hunk. If these leading or following lines do not exactly match the target file, the so-called "fuzz factor" can be loosened from an exact match (0) to a looser match (> 0).

White Space

Sometimes the only difference in the leading and following context lines is in the exact whitespace. The patch apply can be adjusted to ignore white space differences when attempting to apply the patch.

Patch Reversal

Sometimes the patch file was created backwards, meaning that it reflects the differences from the new version to the original, instead of the normal direction of the original to the new version. Reversing the patch will fix this and allow the patch to apply.

Preserving the Patch File, Fixing the Source

If a patch almost, but not quite applies, it can sometimes be fixed by adjusting the source target so that the context matches what the patch is looking for.

If the patch file must be maintained exactly as it was received, this is the preferred method.

After rejects are resolved in this manner, you can always introduce an intermediate patch that takes the source to this adjusted state, allowing the original source and the acquired patch to be preserved, if that is required or desired.

Preserving the Source File, Fixing the Patch

Alternatively, you can adjust the patch file itself. This is more complicated because it involves modifying the patch file using the patching syntax.

This method is preferred if the patch file is unlikely to be externally updated, and thus a localized version is acceptable. It also removes the need for any intermediate patch, as described in the previous section, or the undesirable situation of a patch to a patch.

Placing Unresolved Rejects into Files

Some rejects require study and so cannot be immediately resolved using the methods in this section. You should be able to accept the patch hunks that apply cleanly, and preserve a copy of the hunks that do not. These reject hunks can be saved to a file for analysis.

Placing Unresolved Rejects into the Source (Inline)

Alternatively, you may wish to place the rejected hunks directly in the target source file, so that they can be seen within the context in which they do (or should) apply. This reduces the potential clutter of multiple reject files (which might otherwise be lost or forgotten).

Kernel Patching with scc

Wind River Linux uses the **scc** script to patch the kernel.

The **scc** script is the logic that controls the selection of the kernel patches passed to the build system during the kernel patching phase. The following describes basic **scc** functionality.

The patches are largely self documenting. The **.scc** files document the overall patch strategy for the kernel or feature. The patches themselves have a header that describes the specifics of the patch.

Normally all interactions with the **scc** script are handled by the build system and it should rarely be invoked from the command line. The rich feature set of **scc** is primarily used in constructing the git tree from the kernel cache. Typical end users will, at most, simply list some of their custom add-on patches and configuration changes in a simple **.scc** file they create in their project or custom template.

To patch a kernel with an **.scc** file, see [Patching the Kernel With SCC Files](#) on page 193.

Kernel Patching Design Philosophy

Unlike other packages in the build system, the kernel is not single purpose or targeted at a particular piece of hardware. It must perform the same tasks and offer the same APIs across many architectures and different pieces of hardware.

The key to managing a feature-based patching of the Linux kernel is to remove both the distributed control of the patches (subdirectory-based **patches.list** files) and hand editing of the patch files.

Replacing these two characteristics with script-based patch list generation and a method to control and describe the desired patches with a top-down approach eases the management of

kernel patching complexity. Additionally, a direct mapping between BSPs and profiles can be easily made, increasing maintainability.

The **scc** script has been implemented to control the process of patch list generation and feature-based patching.

In the most simple example, **.scc** files look very similar to the **patches.list** of earlier releases. One notable difference is that the metadata concerning the license, source and reviewers of the patch are contained inside the patch itself and not in the **.scc** file. This information can be in the **.scc** file, but only as a secondary source of information.

About scc Facilities

scc provides the following facilities:

- Top down, feature-based control of patches. This approach allows a feature and profile-based global view of functionality and compatibility to dictate which patches should be applied. It also allows feature- and architecture-specific patch context modifications to be created by each individual feature.
- Feature inheritance and shared patches means that each feature may explicitly include the features and inherit their patches. Each feature can then modify the inherited patches list and substitute slightly different patches to work in their context. This allows the sharing and reuse of patches by only changing the minimum amount and context of existing feature patches.
- Allows upstream, feature-based patches to be logically grouped and used in many different patch stacks. This allows isolation and combination testing of features and allows a single set of patches to be used in multiple platforms.

Modifications to a feature patch set are contained in the modifying top level feature's directory, leaving the original patch in it is pristine form. These are called patch context mods and can be architecture-, platform-, or feature-based.

Patch context mods can be identified by the name of the original patch which they are based plus a suffix of the feature name which required the modification of the original patch.

- Associates kernel configuration directly with the patches that comprise a kernel feature.

About .scc Files

An **.scc** file is a small, sourced shell script. Not all shell features should be used in these scripts, and in particular no output should be generated, because the script is interpreted by the calling framework. You can use conditionals and any other shell commands, but you should be careful to use only basic, standard commands.

A feature script may denote where it should be located in the link order. This is only used by scripts that are not being included by a parent or entry point script and that you wish to be executed. The following declaration denotes the section names in a **.scc** file:

```
# scc.section section_name
```

Any variable passed to **scc** with the **-D=macro** option is available in individual feature scripts. To see what variables are available, locate the invocation of **scc** and search for defines.

The following built-in functions are available:

dir

Changes the current working patch directory, and subsequent calls to patch use this as their base directory.

patch

Outputs a patch to be included in the feature's patch set. Only the name of the patch is supplied, and the path is calculated from the currently set patch directory.

include

Indicates that a particular feature should be included and processed in order. There is an optional parameter **after** *feature_name* to indicate that the order of processing should not be used and a feature must be included **after** *feature_name*. Include paths are relative to the root of the directories passed with **-I**.

Note that changing the default order of large feature stacks by forcing a different order with **after** can result in a significant work effort in order to rebase the patches of the features, if they are touching the same source files.

set_kernel_version

Takes a new kernel version as its argument. This allows a feature to change the effective kernel version and allows other features to test this value with the *KERNEL_VERSION* variable.

scc File Examples

The following presents an example on the use of the **scc** command. Note that you can get detailed help with:

```
$ scc --help=scc
```

For an example of an **scc** file that specifies a normal node, refer to the following code example: Configuration files and patches in a **.scc** file are specified as shown (with comments):

```
#           +---- name of file to read
#           v
kconf hardware common_pc.cfg
# ^       ^
# |       +-- 'type: hardware or non-hardware
# |
# +---- kernel config

# patches
patch 0002-atl2-add-atl2-driver.patch
patch 0003-net-remove-LLTX-in-atl2-driver.patch
patch 0004-net-add-net-poll-support-for-atl2-driver.patch
```


14

Patching Userspace Packages

- [Introduction to Patching Userspace Packages 159](#)
- [Patching with Quilt 160](#)
- [Create an Alias to exportPatches.tcl to save time 161](#)
- [Preparing the Development Host for Patching 161](#)
- [Patching and Exporting a Package to a Layer 162](#)
- [Verifying an Exported Patch 164](#)
- [Incorporating a Patch into a Platform Project Image 165](#)

Introduction to Patching Userspace Packages

Learn about the requirements for patching userspace packages for your platform projects.

Patches for packages are delivered with the recipe in a directory with the same name as the package. For example, for the **busybox** package, there are a few patches in *projectDir/layers/oe-core/meta/busybox/busybox-version*, and couple of additional patches in another layer in *projectDir/layers/wr-base/recipes-core/busybox/busybox*

This is just a convention; the recipe specifies the location of the patches, for example:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"  
SRC_URI += "file://umount-make-d-always-active-add-D-to-suppress-it.patch \  
           file://move-ip-to-sbin-to-make-it-more-FHS-and-LSB.patch"
```

The build system identifies the files as patches by their **.patch** extension and applies them in the patch build stage when you use the default **do_patch** build rule.

When multiple patches need to be applied; the order in which they are applied can be important, to manage the order you will find in the package build directory a subdirectory called **patches**. This contains all the patches gathered from all the layers in the project and file named **series**, which lists the order in which they will be applied.

For example, listing the contents of `projectDir/build/busybox-1.19.4-r17/busybox-1.19.4/patches/` would show patches in the BusyBox `patches` directory as follows:

```
B921600.patch
busybox-appletlib-dependency.patch
busybox-cross-menuconfig.patch
busybox-mkfs-minix-tests_bigendian.patch
busybox-udhcp-no_deconfig.patch
fix-for-spurious-testsuite-failure.patch
get_header_tar.patch
move-ip-to-sbin-to-make-it-more-FHS-and-LSB.patch
move-ip-to-sbin-to-make-it-more-FHS-and-LSB.patch~
run-parts.in.usr-bin.patch
series
sys_resource.patch
umount-make-d-always-active-add-D-to-suppress-it.patch
watch.in.usr-bin.patch
wget_dl_dir_fix.patch
```

The tool that allows you to manage patches and add additional patches to a package is called **quilt**. It is not required that you use **quilt**, but recommended. In Wind River Linux, **quilt** is configured to use the package's `projectDir/build/packageName-revision/packageName/patches` directory and `series` file, by the `.pc/.quilt_patches` and `.pc/.quilt_series` hidden build director.

Create an Alias to exportPatches.tcl to save time

If you frequently work with patches, a common command you will run is `exportPatches.tcl`, which is found in `installDir/wrlinux-5/scripts/exportPatches.tcl`. Rather than constantly type out the full pathname to that Tcl script, you can set a command-line alias. Refer to your host documentation for setting an alias.

Note that `exportPatches.tcl` sources the `wish` interpreter that is provided by Workbench. For information on preparing your host for patching, see [Preparing the Development Host for Patching](#) on page 161.

About Patching Toolchain-related Packages

With Wind River Linux, pre-built toolchain components cannot be modified, so long as you are using the toolchain provided with your installation. This includes the following packages, grouped by package name:

- `eglibc-source-dbg`
- `gdb, gdbserver`
- `libgcc, libgcc-dev`
- `libgomp, libgomp-staticdev, libgomp-dev, libgomp-dbg`
- `libstdc++, libstdc++-dev, libstdc++-staticdev`
- `linux-libc-headers, virtual/linux-libc-headers`

One exception is EGLIBC, which you can rebuild using the `build-libc` feature, or using the custom-distro functionality described in [EGLIBC File Systems](#) on page 85.

Patching with Quilt

Quilt is a general-purpose patching mechanism that you can use whenever you are working with patches—with packages as well as custom applications. The Wind River Linux build system uses it for patching packages as described in this section.

Quilt is especially useful for dealing with a series of patches and with patches that contain multiple files. Open source packages typically contain the package source as well as multiple patches to be applied to that source to produce the binary in a package. In addition, you may be modifying the source to make your own changes.

The proper way to modify the source is to add one or more patches, rather than modifying original source files or patches. This keeps your changes distinct and allows you to carry your custom patches with you as you upgrade to newer versions of Wind River Linux.

Use of the quilt tool facilitates your work with patches. As detailed in this manual, you can patch a package using quilt, and then export the patches from your project directory to a custom layer, for use in other platform projects.

Create an Alias to exportPatches.tcl to save time

Setting an alias for the **exportPatch** command can save you time.

If you frequently work with patches, a common command you will run is **exportPatches.tcl**, which is found in *installDir/wrlinux-5/scripts/exportPatches.tcl*.

- Set a command-line alias.

Refer to your host documentation for setting an alias.



NOTE: Note that **exportPatches.tcl** sources the **wish** interpreter that is provided by Workbench. For information on preparing your host for patching, see [Preparing the Development Host for Patching](#) on page 161.

Preparing the Development Host for Patching

Before you begin patching userspace packages, use this procedure to prepare your host.

Using the **exportPatches.tcl** script to patch packages requires that the **wish** interpreter, part of the **tk** package, be installed and working properly on the development host.

Step 1 Prepare the host for patching packages with Workbench.

```
$ cd installDir
$ wrenv -p workbench
```

Step 2 Prepare the host for patching packages from the command line.

If your installation does not include Workbench, run one of the following commands to install the **tk** package, required to run the **wish** interpreter and **exportPatches.tcl** script:

Options	Description
Ubuntu/Debian-based host	<pre>\$ sudo apt-get install tk</pre>
RedHat/RPM-based host	<pre>\$ sudo yum install tk</pre>

Patching and Exporting a Package to a Layer

Use this procedure to understand the steps necessary to patch a package in your platform project image.

The following steps use the **which** package to describe how to create and export a patch with Wind River Linux.

If you do not already have a platform project, the following **configure** script command was used to create the project in this procedure:

```
$ mkdir qemux86-64_quilt-prj && cd qemux86-64_quilt-prj
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-rootfs=glibc_std \
--enable-kernel=standard \
--enable-build=production \
--enable-parallel-pkgbuilds=4 \
--enable-jobs=4
```

Step 1 Navigate to the top-level folder in the *projectDir*.

```
$ cd projectDir
```

Step 2 Build the **which** package.

Run the following command to create the **which** build directory and apply the current patches to it:

```
$ make -C build which
```

Step 3 Add the toolchain to your path, and make **quilt** available to your platform project.

```
$ export PATH=$PATH:$PWD/bitbake_build/tmp/sysroots/x86_64-linux/usr/bin
```

In this example, **x86_64** is specific to the qemux86-64 BSP used to create the platform project image. This may change for your specific project, depending on your BSP and architecture.

Step 4 Navigate to the new build directory for the patch and display its contents.

```
$ cd build/which-2.20-r3/which-2.20
$ ls patches

remove-declaration.patch  series
```

Step 5 Create a new patch:

```
$ quilt new example.patch

Patch patches/example.patch is now on top
```

Note that **quilt** provides a response for each command.

Step 6 Add a file to the patch.

```
$ quilt add which.c

File which.c added to patch patches/example.patch
```

Step 7 Modify the file's content to patch it.

Open the **which.c** file in a text editor, and change the following line:

```
fprintf(out, "Usage: %s ...);
```

to read:

```
fprintf(out, "USAGE: %s ...);
```

to essentially capitalize the word **USAGE**.

Step 8 Save the **which.c** file.

Step 9 Optionally examine your current local changes.

While optional, you can see the changes made to the original package source by running the following command at any time:

```
$ quilt diff

Index: which-2.20/which.c
=====
--- which-2.20.orig/which.c
+++ which-2.20/which.c
@@ -27,7 +27,7 @@ static const char *programe;

static void print_usage(FILE *out)
{
- fprintf(out, "Usage: %s [options] [--] COMMAND [...] \n", programe);
+ fprintf(out, "USAGE: %s [options] [--] COMMAND [...] \n", programe);
  fprintf(out, "Write the full path of COMMAND(s) to standard output. \n \n");
  fprintf(out, "  --version, -[vV] Print version and exit successfully. \n");
  fprintf(out, "  --help,          Print this help and exit successfully. \n");
}
```

Step 10 Refresh the patch and save all current changes to the patch file created in 5 on page 162.

```
$ quilt refresh

Refreshed patch patches/example.patch
```

Until you perform this refresh step, none of your changes are written to your patch file, so this refresh step is normally your last step before finally writing your patch file to a custom layer for later use.

Step 11 Export the patch.

After you have made all the changes you want, and have “refreshed” your patch with those changes, you can now export your patch to a location of your choice for future inclusion in this or other platform projects. Run the following command to export your patch:

```
$ installDir/wrlinux-5/scripts/exportPatches.tcl \
EXPORT_PATCH_PATCH=full_path_to_patch \
EXPORT_PATCH_LAYER=path_to_layer \
EXPORT_PATCH_DESCR=" text "
```

Where:

EXPORT_PATCH_PATCH

Represents the full path to your generated patch file:

projectDir/build/ package /wrlinux_quilt_patches/ patch_name.

EXPORT_PATCH_LAYER

Represents the path to an existing or new layer directory. The layer infrastructure for the patch and the patch itself will be created for you if it does not already exist.

EXPORT_PATCH_DESCR

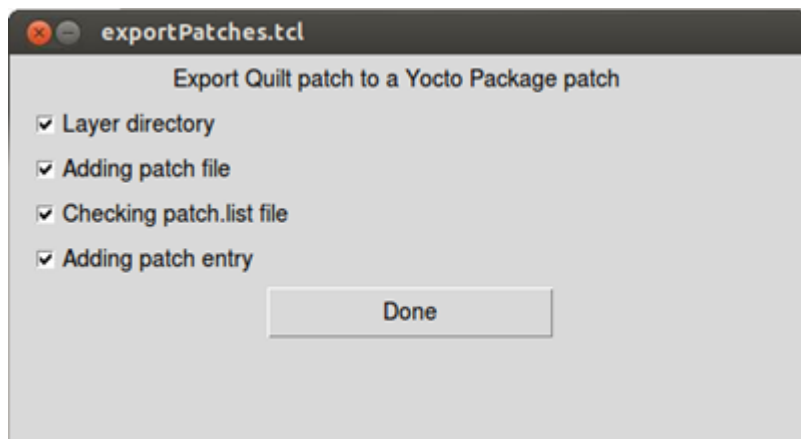
Your description to be included in the `patches.list` file

For example, the following command:

```
$ ~/WindRiver/wrlinux-5/scripts/exportPatches.tcl \  
EXPORT_PATCH_PATCH=$PWD/patches/example.patch \  
EXPORT_PATCH_LAYER=~/.layers/which-test-layer/ \  
EXPORT_PATCH_DESCR="This is a test"
```

will create a new layer for your patch in the `layers/which_test_layer`, in your home directory.

The `exportPatches.tcl` script displays a confirmation dialog to indicate the export results:



Step 12 Rebuild the package with the new patches:

```
$ make -C build which
```

Verifying an Exported Patch

Once you have patched a package, use this procedure to verify the content and structure of the patch.

Perform the following steps to verify the contents of your patch.

Step 1 Navigate to the file system location where you exported your patch to from [Patching and Exporting a Package to a Layer](#) on page 162.

Step 2 View the content and structure of your patch layer.

```
$ tree
```

The output displays the relevant patch structure:

```
which-2.205 tree ~/layers/which-test-layer
/home/revo/layers/which-test-layer
├── conf
│   └── layer.conf
├── recipes-local
│   └── which
│       ├── files
│       │   ├── example.patch
│       │   └── patches.list
│       └── which_2.20.bbappend
```

4 directories, 4 files

Step 3 Display the contents of the recipe information created to apply the patch.

```
$ cat ~/layers/which-test-layer/recipes-local/which/which_2.20.bbappend

# which-2.20-r3: local patches
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"

# preserve this SRC_URI formatting to support patch update tools
SRC_URI += "\
    file://example.patch \
"
```

Note that this recipe append file includes the new **example.patch** file.

Step 4 View the contents of the patch itself:

```
$ cat ~/layers/which-test-layer/recipes-local/which/files/example.patch

Index: which-2.20/which.c
=====
--- which-2.20.orig/which.c
+++ which-2.20/which.c
@@ -27,7 +27,7 @@ static const char *progname;

static void print_usage(FILE *out)
{
- fprintf(out, "Usage: %s [options] [--] COMMAND [...]\n", progname);
+ fprintf(out, "USAGE: %s [options] [--] COMMAND [...]\n", progname);
  fprintf(out, "Write the full path of COMMAND(s) to standard output.\n\n");
  fprintf(out, "  --version, -[vV] Print version and exit successfully.\n");
  fprintf(out, "  --help,          Print this help and exit successfully.\n");
}
```

Note that this is similar to the output of the **quilt diff** command, if you ran the command prior to refreshing and exporting the patch.

Incorporating a Patch into a Platform Project Image

Once you create and export a patch to a layer, you can include the layer (and patch) in an existing or new platform project.

The following procedure requires that you have previously created and exported a patch to a layer as described in [Patching and Exporting a Package to a Layer](#) on page 162.

Step 1 Choose a platform project option to add the layer (and patch) to:

Options	Description
Existing platform project	<p>To add the layer to an existing project:</p> <ol style="list-style-type: none">1. Navigate to the platform project directory: <pre>\$ cd projectDir</pre>2. Run the following commands to enable project reconfiguration and run that configuration to include the new patch layer: <pre>\$ echo `tail -1 config.log` --enable-reconfig --with-layer=~path_to_layer > newconfig \$ chmod +x newconfig \$./newconfig</pre> <p>The project will configure itself to include the new layer.</p>
New platform project	<p>To add the layer to a new platform project, include the --with-layer=path_to_layer configure option when you create your project. For example:</p> <pre>\$ configDir/configure --enable-board=qemux86-64 \ --enable-rootfs=glibc_std \ --enable-kernel=standard \ --enable-build=production \ --enable-parallel-pkgbuilds=4 \ --with-layer=path_to_layer \ --enable-jobs=4</pre>

Step 2 Optionally, build the platform project:

```
$ make
```

15

Modifying Package Lists

[About the Package Manager 167](#)

[About Modifying Package Lists 172](#)

About the Package Manager

Use the Package Manager tool to manage the analysis and removal of packages in your platform project image.

Overview

When you configure and build a platform project image, a number of packages are included automatically, depending on your project configuration. Including additional layers and templates in your configuration often adds additional packages and their dependencies to your platform project image.

As you continue to develop your target system, you may want to remove packages to help decrease your platform's file system footprint. The Package Manager helps simplify the process of package removal with an interface that displays a package's state and any other dependencies the package requires.

The package removal feature works by creating a modified package list and appending it to the `projectDir/default-image.bb` file. The tool uses the build system's tools that identify the dependent packages and generate the revised package list.

About Package States and Information

Packages that are built as part of building the platform project's file system provide full dependency information. Before you use the Package Manager to display status and remove packages, you should ensure your platform project image is built.

Packages that not yet built but are known to be required are presented in an preliminary state. The Package manager can retrieve some package information, but it is not complete.

Packages that are available for the image, but are attempted only if they are needed, generally provide no information about the package's dependencies.

Launching the Package Manager

Use this procedure to start the Package Manager and obtain package information from your platform project image.



WARNING: The content of this chapter must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionality.

This procedure requires a previously built platform project image.



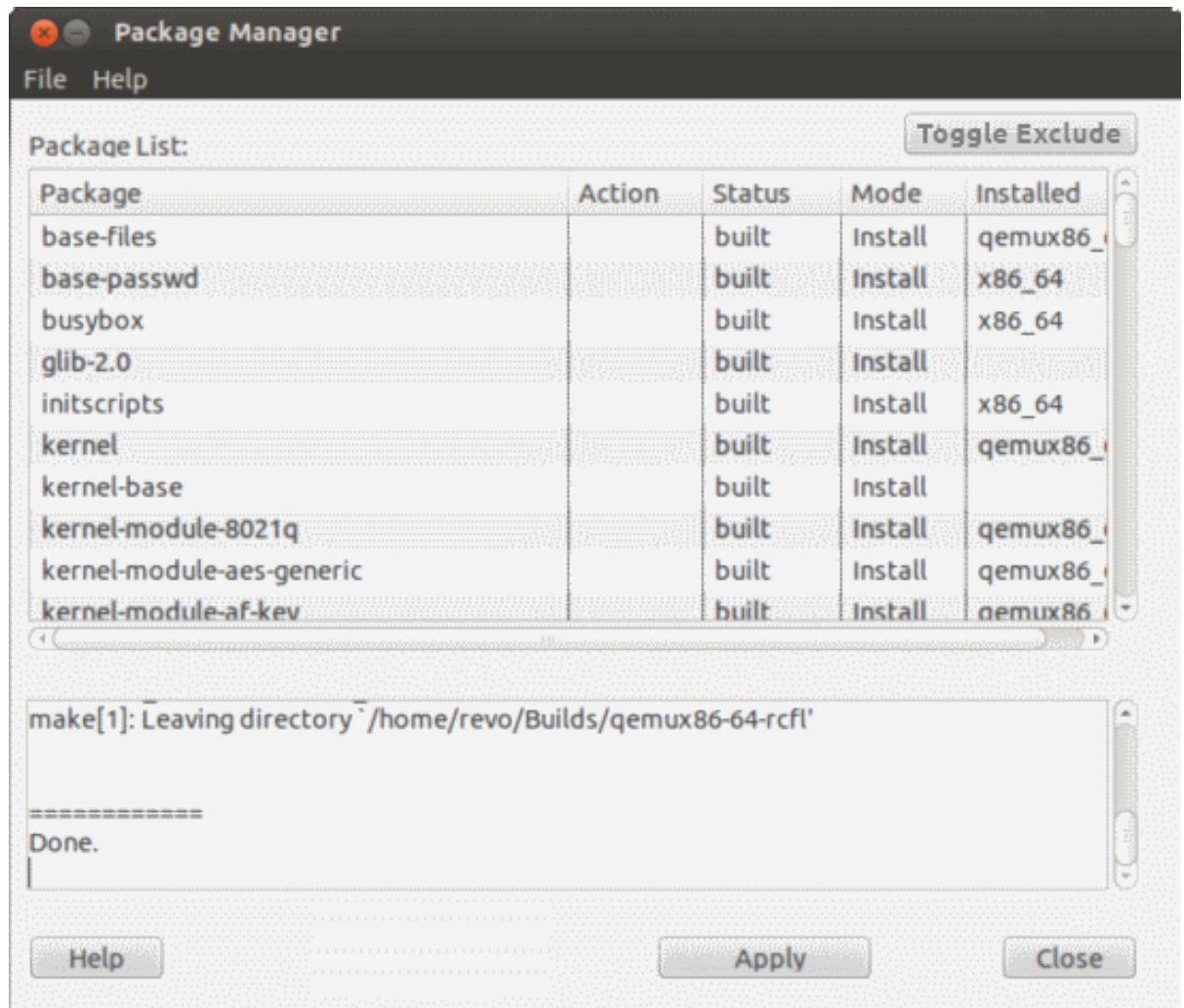
NOTE: You may launch the Package Manager on a configured, but not built, platform project, but doing so will provide limited package data, as this information is compiled by the build system as it builds the platform project image.

- Launch the Package Manager.

Run the following command from the *projectDir*:

```
$ make package-manager
```

The Package Manager tool displays.



When the tool initially displays, it will read the package state for the platform project and update the **Status**, **Mode**, and **Installed** columns accordingly.

Removing Packages

Remove a package and its dependencies after launching the tool.

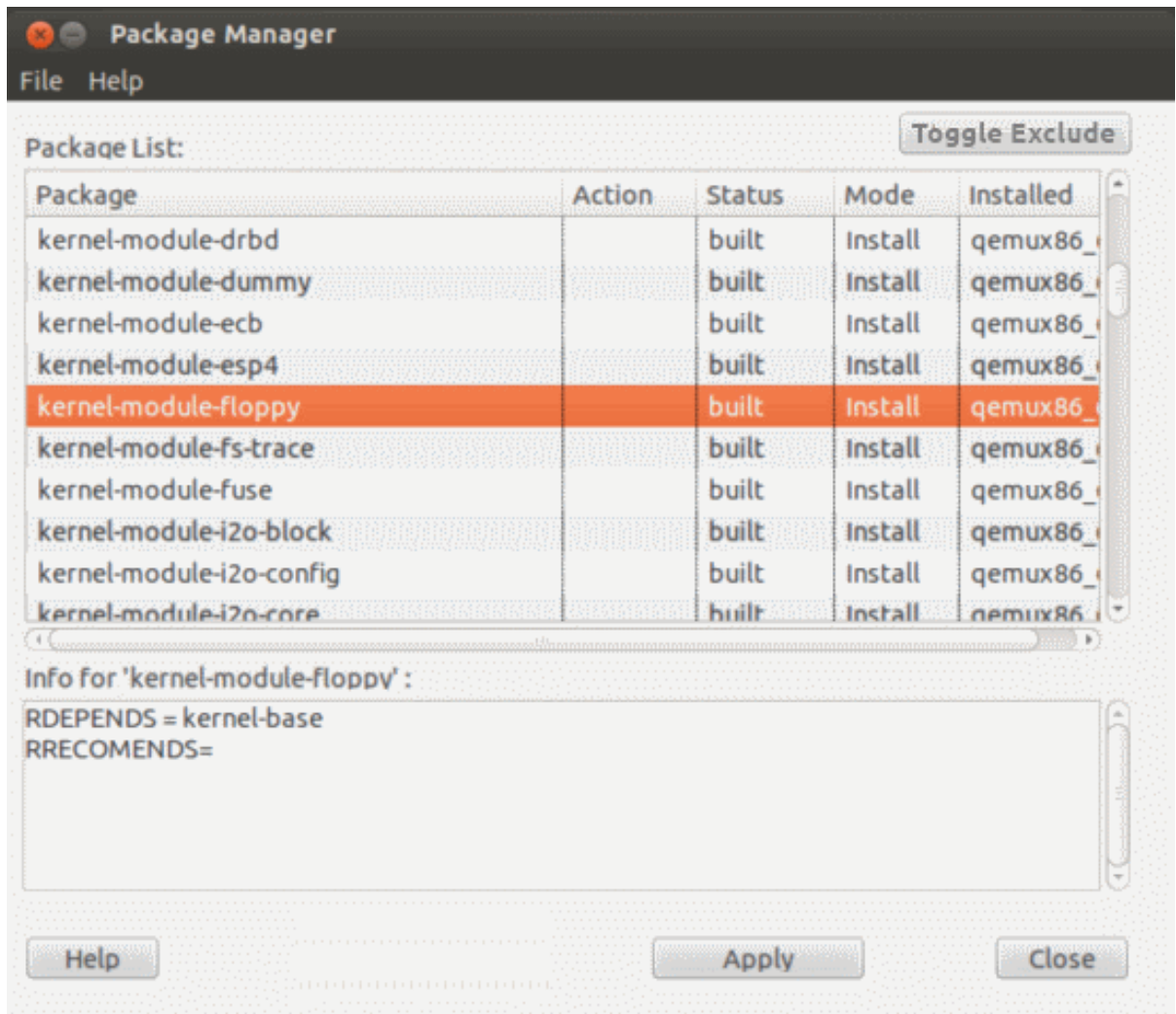


WARNING: The content of this chapter must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionality.

Once the Package Manager is running, you can select packages and remove them.

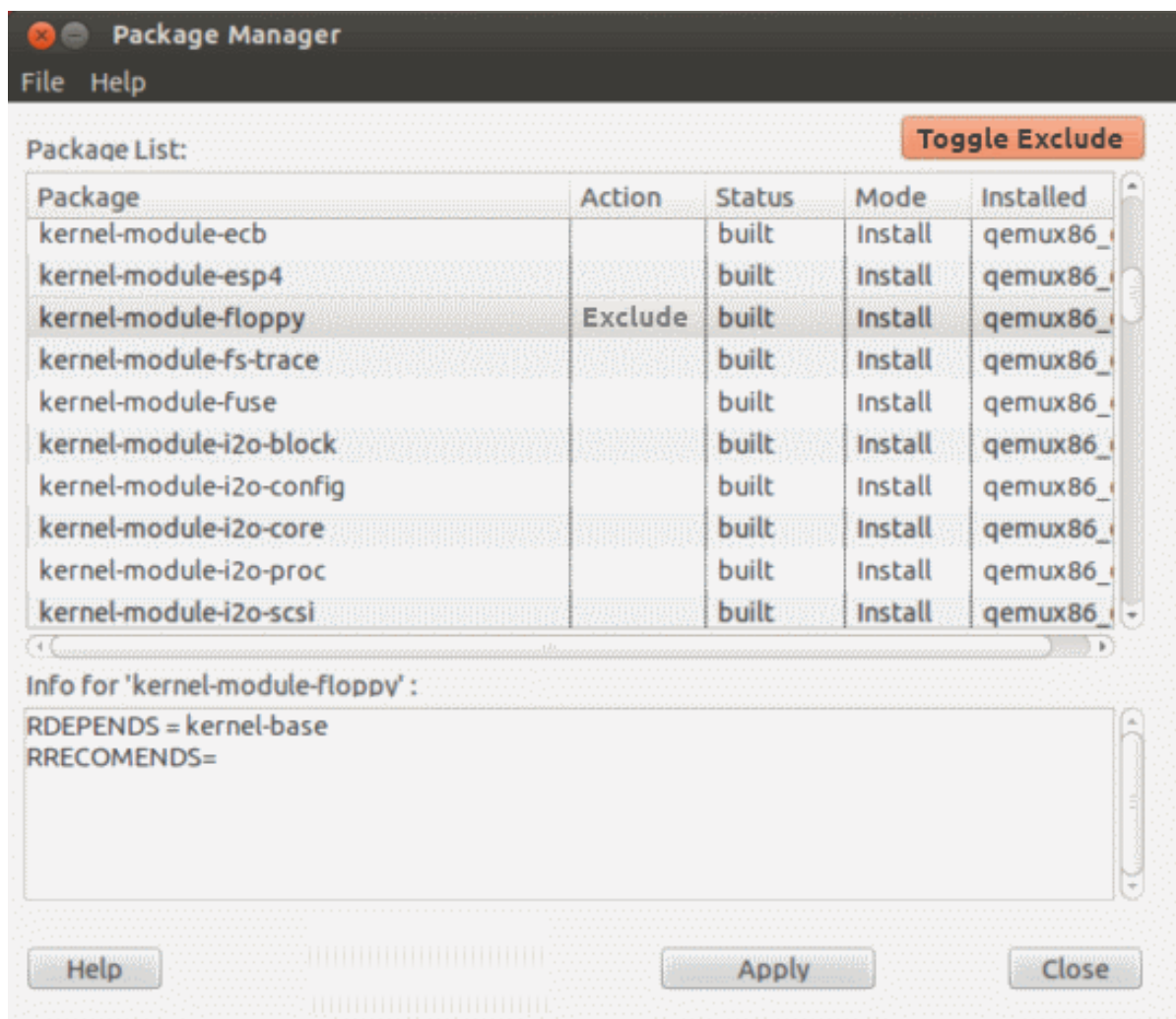
Step 1 Select a package in the **Package** column.



Information for the package displays in the bottom field. In this example, the **kernel-module-floppy** package depends on the **kernel-base** package.

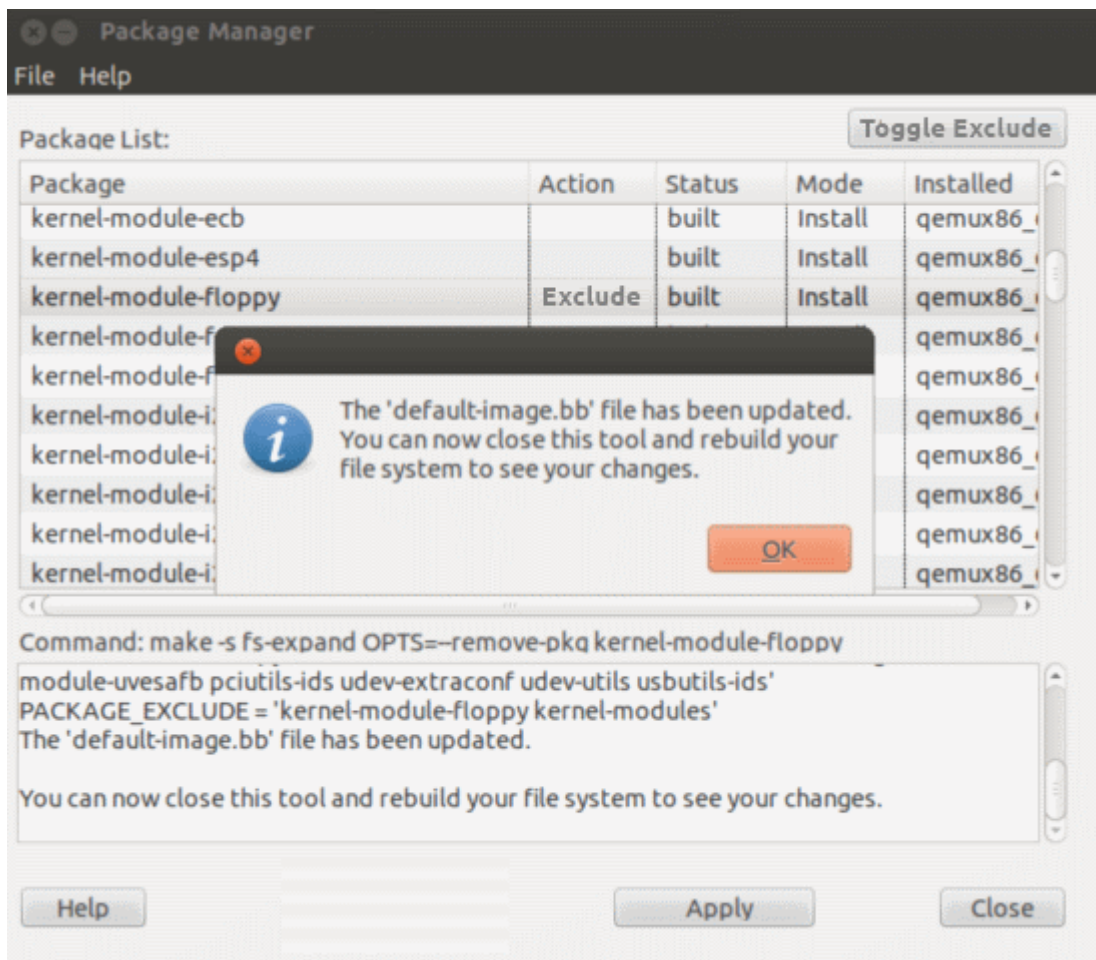
Step 2 Click **Toggle Exclude**.

Notice that the **Action** column displays **Exclude** for the selected package.



Step 3 Click **Apply**, then click **Yes** to confirm package removal.

The Package Manager processes the removal and displays a confirmation when complete.



If the package selected has a hard dependency from another package in the file system, you will receive a warning in the bottom field. If this occurs, you will not be able to remove the package with the Package Manager.

Step 4 Click **Close** to close the Package Manager.

Step 5 Rebuild the platform project image.

Run the following command from the *projectDir*:

```
$ make
```

Step 6 Optionally open the Package Manager to display the new package information.

See [Launching the Package Manager](#) on page 168 for additional information.

About Modifying Package Lists

You can modify the package list for a project.

This section describes scaling a platform project to add or remove functionality by adding and removing packages. For a more fine-grain approach of modifying the contents of the file system

see As described in *Analyzing and Optimizing Runtime Footprint* on page 307 and the section *Finalizing the File System Layout with changelist.xml*.

The Wind River front-end or wrapper to the Yocto/bitbake build system implements the concept of templates and individual packages that can be added (and some cases removed) from a project. This is in addition to the packages specified in the bitbake recipes in the various layers which make up the project. This section discusses the use of these features in customizing a platform project configuration and as a result, scaling your file system to include the minimal set of components for your application.

The `glibc_small` and `glibc_core` file systems are designed to contain a minimal amount of packages, for a busybox and bash based file systems respectively. The suggested workflow is to start with one of these file systems and add packages and templates as needed.

Removing packages specified by a layer is more problematic, as often their removal breaks some basic functionality required to boot the board. There are many package dependencies within the base filesystem layer, and while some packages may be removed, often it is necessary to modify the configuration of other components in order to keep them functional. The tools to visualize and understand the complete package dependencies are still under development, so the procedure described later in this chapter is provided as an interim solution until the bitbake build system evolves and richer set of tools is developed.

Adding a Package

Two methods are available for adding packages.

In this procedure, you will learn two methods for adding a package.

- Complete one of the following:
 - Add the parameter `--with-package=` to the `configure` command at project creation time.
 - Use the command `make -C build packageName.addpkg` any time after package creation.
 - Edit the `projectDir/layers/local/image.bb` file and add the following line:

```
IMAGE_INSTALL += "packageName"
```

The recipe for the package is consulted and any dependent and optional packages listed there are added as well.

About Adding Templates

Feature templates are a mechanism that appends the project recipe with additional packages or other configuration options.

In many cases feature templates will add one or more packages to the project by defining an include file which is a list of added packages in a text file with the file extension `.inc`. Thus, every package mentioned in the template is included with all the dependent and optional packages in its recipe.

Feature templates are added at project creation time using the `--with-template argument` to the `configure` command. Layers, including BSP layers, may include a default template that adds packages to your project as a side effect of including the layer.

Removing a Package

Unneeded packages can be removed from a project.

Packages that have been added individually or by a feature template may be removed using the project build command.



NOTE: If the package you wish to remove has been included as part of a layer, or is a package that is included by package dependency specified in another package recipe, this procedure will not work. Wind River Linux provides the Package Manager tool for removing these types of packages. For additional information, see [About the Package Manager](#) on page 167.

- Use the following **make** command to remove a package:

```
$ make -C build packageName.rmpkg
```

All package dependencies are reviewed and revised by this command.

PART IV

Kernel Development

Patching and Configuring the Kernel.....	177
Creating Alternate Kernels from kernel.org Source.....	199
Exporting Custom Kernel Headers.....	201
Using the preempt-rt Kernel Type.....	205

16

Patching and Configuring the Kernel

[About Kernel Configuration and Patching](#) 177

About Kernel Configuration and Patching

Use the procedures in this section to learn how to configure the Linux kernel to add or remove options by applying patches directly to the source code.

Customizing the Linux kernel to better fit the particular details of a hardware implementation is almost always a required step in an embedded software development cycle.

Kernel customization can take the form of simply enabling or disabling kernel configuration options; this is typically done to enable specific drivers and to shrink the final kernel image and run-time load by removing unneeded functionality. Customization can also come in the form of patches applied to the source code, either in-house or third party patches, to modify specific areas of kernel behavior.

You can reconfigure kernels in one of the following manners:

- [Configuring the Linux Kernel with menuconfig](#) on page 185
- [Configuring Kernel Modules With Make Rules](#) on page 183
- [Kernel Module Configuration and Patching with Fragments](#) on page 179

Alternatively, you can patch and/or extend kernel capabilities:

- [Patching the Kernel](#) on page 195
- [Patching the Kernel With SCC Files](#) on page 193

Example Platform Project Configure for the Examples in this Chapter

The examples in this chapter are based on the qemux86-64 target board, and assume you have a platform project configured and built with no errors. The tasks in this chapter use the example configure script command from [Configuring and Building a Complete Run-time](#) on page 77.

Note that the content should apply equally to other target boards.

Kernel Source Locations in a Platform Project Directory

Inside your platform project directory the kernel sources will be located in:

build/linux-windriver-*version*-r0/linux

The configuration file of the kernel will be located in:

build/linux-windriver-*version*-r0/linux-qemux86-64-standard-build/.config, where **version** is the current kernel revision, for example 3.4.

Configuration

Use kernel configuration options to add or remove features to your platform project.

Depending on end-user requirements for your platform project image, you may need to make changes to the existing kernel configuration. Use the examples in this section to configure a kernel module with fragments or **make** rules.

The Initial Creation of the Kernel Configuration File

In Wind River Linux, kernel configuration fragments determine a platform's features and comprise the kernel configuration file.

Wind River aims to provide uniformity across BSPs of a given platform and across given architectures. To do so, non-hardware specific kernel options (for example, supported file systems) are generally chosen on a per-platform basis, and then the hardware specific options (for example, device drivers) are chosen on a per-BSP basis.

To achieve this, fragments of kernel configuration files (called **config** files or config fragments) are placed among the other files that determine the content of a particular platform, architecture, feature, or BSP. These fragments contain just the relevant kernel settings that pertain to that area where they are placed.

When you configure your project and make an initial selection of a platform and a BSP (board), you implicitly choose a subset of the various layers and feature templates that are available to be included in your build. Config files that are found in these layers and templates are collected together, and this concatenation of fragments form the initial input to the Linux Kernel Configurator (LKC).

The kernel configuration fragments are collected, starting from the generic and proceeding to the specific, to assemble platform- and board-specific kernel configuration options into a format that is suitable for the LKC.



NOTE: Specifying a particular setting in a configuration fragment does not automatically guarantee that the option appears in the final **.config** file. Wind River still uses the built-in part of the default kernel.org configuration (usually referred to as the *LKC*) to process the fragments and produce the final **.config**, and the final dependency check may discard or add options as required, for example, due to dependency reasons.

The configuration file that is used to generate a new kernel is **projectDir/build/linux-windriver-*version*-r0/linux-qemux86-64-standard-build/.config**. It is created from default kernel.org option settings, plus the options settings from all the kernel configuration files in the distribution and build environments.

The path to the `.config` file is based on the selection of your BSP and kernel type when you configure your platform project, and will change based on your selections.

Kernel Module Configuration and Patching with Fragments

Kernel configuration can be done conveniently using configuration fragments, which are small files that contain kernel configuration options in the syntax of the original kernel's `.config` configuration file.

Kernel fragments capture specific changes to the kernel's configuration file. They are enabled by creating a basic infrastructure inside the local layer, or any other layer included in the platform project.

Once the infrastructure is in place, the BitBake build system will incorporate the kernel fragments into the kernel configuration process to build the corresponding kernel image and associated kernel modules.

In this section, you will learn to reconfigure the Linux kernel to make some changes on the kernel modules which are installed by default.

The changes you will make include:

- Removing the **floppy** and **parport** (parallel port) modules, assuming that they are not necessary for the intended target.
- Turning the **minix** kernel module into a static kernel feature, so that its functionality is provided by the kernel image itself.
- Add the **pcspkr** (PC speaker) module.

Once complete, you will rebuild the kernel and file system, reboot the emulated target, and verify that your changes have been applied.

Populate the Local Layer with the Required Subdirectories

Populate the local layer as part of configuring the Linux kernel with fragments

To perform this procedure, you must have a previously configured and built platform project. For additional information, see [Configuring and Building a Complete Run-time](#) on page 77.

- Populate the local layer with subdirectories.

From the platform project's main directory, enter the following command to create the required directories to maintain your kernel fragments:

```
$ mkdir -p layers/local/recipes-kernel/linux/linux-windriver
```

The basic directory structure necessary to support configuration fragments is dictated by the content of the `BBFILES` variable inside the `projectDir/layers/local/conf/layer.conf` file. See [Directory Structure](#) on page 29.

More specifically, the element `${LAYERDIR}/recipes-*/**/*.bbappend` in this variable determines where the `.bbappend` files will be searched for. The part of the command line above that reads: `recipes-kernel/linux` complies with this pattern.

The `linux-windriver` subdirectory is used to further localize kernel configuration files for the kernels provided by Wind River and it is named after the Linux kernel package itself.

Create the Kernel's BitBake Append (.bbappend) File

Create a BitBake append file as part of configuring the Linux kernel with fragments

This procedure requires that you have populated the **projectDir/layers/local** directory with the subdirectories required for patching the kernel as described in [Populate the Local Layer with the Required Subdirectories](#) on page 179.

Step 1 Create the BitBake append (**.bbappend**) file of the kernel.

Run the following command to create the kernel's **.bbappend** file:

```
$ vi layers/local/recipes-kernel/linux/linux-windriver_3.4.bbappend
```

Step 2 Add the following default lines of code:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"  
SRC_URI += "file://config_baseline.cfg"
```

The variable `FILESEXTRAPATHS_prepend` extends the search path of BitBake to include a directory named after the package being processed, **PN** for package name under the current directory, **THISDIR**. In this example, **PN** is **linux-windriver** and this explains why we originally created a subdirectory with this name.

The name of the kernel fragment is added to the BitBake variable `SRC_URI`, which holds the list of configuration files, of any kind, to be processed when building the project.

The syntax `file://config_baseline.cfg` is used to tell BitBake that the configuration fragment is to be found as a regular text file inside the layer, and not for example, through a source version control system somewhere else. This file is where you will add fragments that make changes to the kernel.

Create the Kernel's Configuration Fragment

Create a kernel fragment as part of configuring the Linux kernel with fragments

This procedure requires that you have created a **.bbappend** file for patching the kernel as described in [Create the Kernel's BitBake Append \(.bbappend\) File](#) on page 179.

Step 1 Create the configuration fragment for the kernel.

Create the kernel's **config_baseline.cfg** file.

```
$ vi layers/local/recipes-kernel/linux/linux-windriver/config_baseline.cfg
```

Step 2 Configure kernel fragments for this example.

The following lines add fragment configuration:

```
# CONFIG_BLK_DEV_FD is not set  
# CONFIG_PARPORT is not set  
CONFIG_MINIX_FS=y  
CONFIG_INPUT_MISC=y  
CONFIG_INPUT_PCSPKR=m
```

The configuration fragment(s) in this example have the same syntax as the **.config** file for the kernel. Note that we added the statement:

```
CONFIG_INPUT_MISC=y
```

which is a prerequisite for the option **CONFIG_INPUT_PCSPKR** to become available.

Also note that lines starting with the **#** character are not comments but indicate instead that a particular kernel feature is to be disabled.

At this moment the layer structure to support kernel configuration fragments should look like this:

```
layers/local/recipes-kernel/ |- linux |- linux-windriver |- config_baseline.cfg |-  
linux-windriver_3.4.bbappend
```

Clean up the Linux Kernel Package and Optionally Configure the Package

Clean up the Linux kernel package as part of configuring the Linux kernel with fragments

This procedure requires that you have previously created a kernel configuration fragment as described in [Create the Kernel's Configuration Fragment](#) on page 180.

Step 1 Clean up the Linux kernel package.

```
$ make -C build linux-windriver.clean
```

Cleaning up the **linux-windriver** kernel package first is a necessary step to force the build system to subsequently reload all associated configuration files. You should do this every time you make changes to your kernel configuration fragments and prior to rebuilding the kernel package.

Step 2 Configure the Linux kernel package if necessary.

```
$ make -C build linux-windriver.configure
```

Configuring the Linux kernel package is an optional step in that configuration will happen automatically when later on you get to rebuild the package. However, configuration is done much faster than rebuilding. Therefore the advantage of doing the configuration step manually is that you can verify very quickly that the changes specified in the configuration fragment are correct by inspecting the generated configuration file of the kernel located at:

projectDir/build/linux-windriver-version/linux-qemux86-64-standard-build/.config

Rebuild the Linux Kernel Package and File System

Rebuild the kernel and file system as part of configuring the Linux kernel with fragments

Once the kernel configuration fragment has been created, and the **linux-windriver** kernel package has been cleaned and configured as described in [Clean up the Linux Kernel Package and Optionally Configure the Package](#) on page 181, follow this procedure to rebuild the kernel.

Step 1 Rebuild the Linux kernel package and file system.

Run the following command from the platform project's directory to rebuild the kernel package:

```
$ make -C build linux-windriver.rebuild
```

Once complete, the new **linux-windriver** package is available containing the modified kernel image to be used in the target.

Step 2 Rebuild the file system.

```
$ make
```

This command updates the root file system to include the new structure of kernel modules to be loaded on the target. Note that if your configuration fragments do not modify the current or default kernel modules then you do not need to rebuild the root file system.

For example, if the only line in the configuration fragment above had been **CONFIG_PRINTK_TIME=y** then only the kernel image would have been modified when rebuilding the kernel package but the root file system would have remained the same.

Run the Emulated Target

Run the emulated target after you have configured the Linux kernel with fragments

This procedure tests whether the kernel configuration fragments created in [Create the Kernel's Configuration Fragment](#) on page 180 function as expected on a simulated target platform.

To complete this procedure, you must have previously configured the kernel package and rebuilt the file system as described in [Rebuild the Linux Kernel Package and File System](#) on page 181.

Step 1 Verify that the **pcspkr** module still exists on the target.

Run the following command from the platform project directory:

```
$ make start-target
```

Step 2 Log in to the system.

Use the user name **root** with password **root**

Step 3 Verify that status of the **floppy**, **parport**, and **pcspkr** modules:

```
root@qemux86-64: ~# lsmod
```

The system should return the following:

```
Not tainted  
pcspkr 2030 0 - Live 0xffffffffa0002000
```

The module list shows that the **floppy** and **parport** modules are no longer present and that the **pcspkr** module is active now.

Step 4 Review how the **pcspkr** module loads.

Run the following command on your qemux86-64 target:

```
$ cat /usr/lib64/udev/rules.d/60-persistent-input.rules | grep pcspkr
```

The system should return the following:

```
DRIVERS=="pcspkr",ENV{.INPUT_CLASS}="spkr"
```

The automatic loading of modules is handled by the udev infrastructure.

Step 5 Verify that the minix file system is still supported.

```
root@qemux86-64: ~# cat /proc/filesystems |grep minix
```

The system should return the following:

```
minix
```

Support for the minix file system is still available but this time it is built into the Linux kernel image itself.

Step 6 Shut down the target.

Enter the following command in the emulator console:

```
root@qemux86-64: ~# halt
```

Configuring Kernel Modules With Make Rules

The **make** command provides a simplified means to add or remove selected kernel modules as needed.

After you build your project, all configured kernel modules become available for use with the **make** command, as detailed in this section.

In this topic, the **pcspkr** module will be added as a project package and not directly as a kernel option as was done before.

To perform this procedure, you must have the following pre-requisites met:

A working platform project

If you can configure and build a platform project successfully, then the **make** command is working and you have a platform project to perform this procedure. For an example of the platform project configure options used to create this procedure, see [Configuring and Building a Complete Run-time](#) on page 77.

The **pcspkr** module

The following procedure assumes the **pcspkr** module is available at this point because it was built previously in [Kernel Module Configuration and Patching with Fragments](#) on page 179

Understanding What Modules are Already Available

After an initial build of the file system completes, all configured kernel modules become available as pre-compiled binaries inside your project's working space. The first thing to do is to determine which modules are available, and then use the make rule options to add or to remove selected modules.

In this example, once you determine which kernel modules are available, you are going to add the **pcspkr** module, verify that it is loaded in the target, and then remove it, using make commands.

You may have added the **pcspkr** module already using the **menuconfig** method as described in [Configuring the Linux Kernel with menuconfig](#) on page 185. In this exercise, that module will be added as a project package and not directly as a kernel option as was done before.

Step 1 Determine the kernel modules available in your platform project.

Run the following command from the platform project directory to list and sort these files:

```
$ find bitbake_build/tmp/deploy/rpm | grep kernel-module- | \  
perl -p -i -e 's/(kernel-module-.*)-3.*/$1/' | sort
```

The result will be an alphabetically sorted list of all available modules already pre-compiled and ready to be used. Kernel modules are packaged as individual files in the following directory:

***projectDir*/bitbake_build/tmp/deploy/rpm**

Their file names all start with the **kernel-module-** prefix.

Step 2 Add the **pcspkr** module to the build.

As stated in *Pre-requisites*, above, the following command assumes the **pcspkr** module is available at this point because it was built previously in [Kernel Module Configuration and Patching with Fragments](#) on page 179

- a) Add the module.

```
$ make -C build kernel-module-pcspkr.addpkg
```

- b) Verify that the kernel module package has been added.

```
$ cat layers/local/recipes-img/images/wrlinux-image-file-system.bb
```

In this example, *file-system* refers to the rootfs used to configure your platform project. If you used the instructions from [Configuring and Building a Complete Run-time](#) on page 77, the command would be:

```
$ cat layers/local/recipes-img/images/wrlinux-image-glibc-small.bb
```

The system will return the following output, after the line that declares **#### END Auto Generated by configure ####**:

```
#### END Auto Generated by configure ####  
IMAGE_INSTALL += "kernel-module-pcspkr"
```

This indicates that the package will be included in the build.

- c) Rebuild the root file system.

```
make
```

Step 3 Verify the **pcspkr** module exists on the target.

- a) Launch the platform project image in an emulator.

```
make start-target
```

- b) Once the emulator finishes booting, login as user **root** with password **root**.
c) Verify that the **pcspkr** module was added to the target.

```
root@qemux86-64: ~# lsmod
```

The system should return the following, indicating that the **pcspkr** module was added:

```
Not tainted  
pcspkr 2030 0 - Live 0xfffffffffa0002000
```

- d) See how the **pcspkr** module loads.

```
$ cat /usr/lib64/udev/rules.d/60-persistent-input.rules | grep pcspkr
```

The system should return the following:

```
DRIVERS=="pcspkr",ENV{.INPUT_CLASS}="spkr"
```

Step 4 Remove the **pcspkr** module from the build.

- a) Remove the **pcspkr** module package.

Run the following command in the *projectDir*:

```
$ make -C build kernel-module-pcspkr.rmpkg
```

- b) Clean and rebuild the kernel image.

```
$ make -C build linux-windriver.clean
```


This updates the set of available kernel modules, removing the **pcspkr** module in the process.

Step 5 Verify the **pcspkr** module is removed from the build.

a) Launch the platform project in an emulator.

Run the following command in the *projectDir*:

```
$ make start-target
```

b) After the emulator finishes booting, login as user **root** with password **root**.

c) Verify that the **pcspkr** module is removed from the target.

Run the following command on the target:

```
root@qemux86-64: ~# lsmod
```

The system should return the following, indicating that the **pcspkr** module was added:

```
Not tainted
```

Notice that the **pcspkr** module no longer loads or is present.

d) Shut down the target.

```
root@qemux86-64: halt
```

Configuring the Linux Kernel with menuconfig

Perform the procedure in this section to use **menuconfig** to access and change the different kernel options.

menuconfig is a basic configuration mechanism provided by the Linux kernel build system that provides a menu-based access to the different kernel options.

In this section, you will learn to reconfigure the Linux kernel to make some changes on the kernel modules which are installed by default.

The changes you will make include:

- Removing the **floppy** and **parport** (parallel port) modules, assuming that they are not necessary for the intended target.
- Turning the **minix** kernel module into a static kernel feature, so that its functionality is provided by the kernel image itself.
- Add the **pcspkr** (PC speaker) module.

Once complete, you will rebuild the kernel and file system, reboot the emulated target, and verify that your changes have been applied.

The following procedures require a configured and built platform project. See [About Kernel Configuration and Patching](#) on page 177.

Step 1 Boot the emulated **qemux86-64** target that you have built in the previous sections.

a) Launch the target.

From the platform project's directory, run the following command:

```
$ make start-target
```

b) Log in as user **root**, and password **root**.

You should have now access to the command line shell on the target.

Step 2 List the kernel modules installed on the target.

Run the following command from the target console:

```
root@gemux86-64:~# lsmod
```

The console should return the following output:

```
Not tainted
parport 23894 1 parport_pc, Live 0xffffffffa0017000
floppy 60578 0 - Live 0xffffffffa0022000
parport_pc 18367 0 - Live 0xffffffffa0038000
minix 29971 0 - Live 0xffffffffa0042000
```

This output represents the list of kernel modules loaded in the system. In this example, we will assume that **floppy** and **parport** (parallel port) modules are not required, so we will remove them. We will also integrate the **minix** module into the kernel image itself.

Step 3 Verify support for the minix file system.

Run the following command from the target console:

```
root@gemux86-64:~# cat /proc/filesystems |grep minix
```

The console should return the following output to indicate support for the minix file system:

```
minix
```

Note that since the **minix** module is already loaded, it is expected that the kernel supports it.

Step 4 Shutdown the emulated target.

Run the following command from the target console:

```
root@gemux86-64:~# halt
```

This will cleanly shutdown the console window so you can make changes to the kernel's configuration.

Step 5 Launch the **menuconfig** configuration tool for the kernel

Note that to use **xconfig** or **config**, listed in the commands, below, you must have the QT toolkit and QT development tools installed on your host. For example, with a Debian-based workstation, you could use the command: **sudo apt-get install qt4-dev-tools qt4-qmake** to install QT.

Enter one of the following commands from the platform project directory to launch the kernel configuration menu:

Options	Description
Run <code>menuconfig</code> in a separate terminal window:	<pre>\$ make -C build linux-windriver.menuconfig</pre>
Run the graphical <code>xconfig</code> interface to <code>menuconfig</code> :	<pre>\$ make -C build linux-windriver.xconfig</pre>
Run the graphical <code>gconfig</code> interface to <code>menuconfig</code> :	<pre>\$ make -C build linux-windriver.gconfig</pre>

After a few seconds, a new terminal window displays with the kernel configuration menu.

Step 6 Remove the **floppy** and **parport** modules from the kernel configuration.

a) From the top kernel configuration menu, select **Device Drivers > Block devices > Normal floppy disk support** .

Normal floppy disk support should be listed with a letter **M** marker indicating that it is to be compiled as a module.

b) Press **SPACE** twice to remove this module from the build.

The marker is now blank indicating that the floppy module is not selected.

c) Select **Exit** at the bottom menu twice, using **TAB** or left/right arrow keys, to return to the top level list of configuration options.

d) From the top kernel configuration menu, select **Device Drivers > Parallel port support** .

Parallel port support should be listed with a letter **M** marker indicating that it is to be compiled as a module.

e) Press **SPACE** twice to remove this module from the build.

f) Select **Exit** at the bottom menu to return to the top level list of configuration options.

Step 7 Turn the **minix** module into a static kernel feature.

a) From the top kernel configuration menu, select **File systems > Miscellaneous filesystems > Minix file system support**.

Minix file system support should be listed with a letter **M** marker indicating that it is to be compiled as a module.

b) Press **SPACE** once to turn it into a static kernel feature.

The marker is now a ***** indicating that the minix option is configured as a static kernel feature.

c) Select **Exit** at the bottom menu twice, pressing **TAB** to return to the top level list of configuration options.

Step 8 Add the **pcspkr** module.

By default, miscellaneous device support is disabled. To add PC speaker support, you must enable it.

a) From the top kernel configuration menu, select **Device Drivers > Input device support**, then press **SPACE**.

- b) Select **Miscellaneous devices**, then press SPACE to enable the sub-menu.
- c) Select PC Speaker support and press the space bar to add the PC Speaker support option as a module.

The marker is now a letter **M** indicating that the **pcspkr** option is a module.

- d) Select **Exit** three times to return to the top level list of configuration options.

Step 9 Save the new kernel configuration and rebuild the image and modules.

- a) From the top kernel configuration menu, select **Exit**.
- b) When prompted to save your new configuration, select **Yes** to finish the configuration session.
- c) Run the following command from the platform project directory to rebuild the kernel image and modules:

```
$ make -C build linux-windriver.rebuild
```

- d) Run the following command from the platform project directory to rebuild the root file system and update the kernel modules as necessary:

```
$ make
```

Step 10 Boot the emulated target to test your new kernel configuration.

- a) Launch the target.

Run the following command from the platform project directory:

```
$ make start-target
```

- b) After the target window boots, login as user **root** with password **root**.
- c) Verify that status of the **floppy**, **parport**, and **pcspkr** modules:

```
root@qemux86-64:~# lsmod
```

```
Not tainted  
pcspkr 2030 0 - Live 0xffffffffa0002000
```

The module list shows that the **floppy** and **parport** modules are no longer present and that the **pcspkr** module is active now.

- d) See how the **pcspkr** module loads.

The udev infrastructure manages automatic module loading.

On your qemux86-64 target, type the following command:

```
$ cat /usr/lib64/udev/rules.d/60-persistent-input.rules | grep pcspkr
```

The system should return the following:

```
DRIVERS=="pcspkr",ENV{.INPUT_CLASS}="spkr"
```

- e) Verify that the minix file system is still supported:

```
root@qemux86-64:~# cat /proc/filesystems |grep minix
```

The system should return the following:

```
minix
```

Support for the minix file system is still available but this time it is built into the Linux kernel image itself.

Patching

Use kernel patching to apply kernel changes directly to the kernel source code.

Depending on end-user requirements for your platform project image, you may need to customize the kernel source code, either to make changes to the Wind River kernel, or third-party modules or patches. Use the examples in this section to patch a kernel.

Kernel Configuration Fragment Auditing

Use audit reporting to identify potential issues with your Linux kernel configuration.

Wind River provides an informational audit that takes place when the configuration (**.config**) file is generated that looks for the following:

1. Non-hardware specific settings in the BSP fragments.
2. Settings specified in the BSP fragments that it is necessary to change or remove in the final **.config** to satisfy the dependency information of LKC.
3. Settings that were duplicated in more than one fragment.
4. Settings that simply do not match any currently available option.

The intent of this on-the-fly-audit of the fragment content and the generated **.config** file is to warn you when it looks like a BSP may be doing things it should not be doing.

For example, filtering is performed to identify duplicate entries, and warnings are issued when options appear to be incorrect due to being unknown or being ignored for dependency reasons.

Because there are many kernel options available and many kernel configuration fragments, the auditing mechanism provides summary output to the screen and collects detailed information in a folder relevant to kernel configuration fragment processing. The warnings are captured in files in the audit data directory

projectDir/build/linux-windriver/linux/.meta/cfg/kernel_type/bsp_name/*.

For example, the following directory would apply to an Intel BSP with a standard kernel type:

projectDir/build/linux-windriver/linux/.meta/cfg/standard/intel-x86-64

Kernel options are all sourced from Kconfig files placed in various directories of the kernel tree that correspond to the locations of the code that they enable or disable. The logical grouping has the effect of making each the content of each Kconfig either primarily hardware specific (for example, options to enable specific drivers) or non-hardware specific (for example, options to choose which file systems are available.)

Auditing is implemented by the `kconf_check` script, from the Yocto Project kernel tools recipes. The auditing takes place in two steps, since the input first needs to be collated and sanitized, and then the final output in the **.config** file from the LKC must be compared to the original input in order to produce warnings about dropped or changed settings. This script is responsible for assembling the fragments, filtering out duplicates, and auditing them for hardware and non-hardware content.

The files of interest under the ***projectDir/build/linux-windriver/linux/.meta/cfg/kernel_type/bsp_name/*** directory include the following:

hardware.kcf

The list of hardware Kconfig files.

non-hardware.kcf

The list of non-hardware Kconfig files.

By the end of this process, Wind River has sorted all the existing Kconfig files into hardware and non-hardware, and this forms the basis of the audit criteria.

Audit Reporting

The audit takes place at the Linux configuration step and reports on the following:

- Items in the BSP that do not look like they are really hardware related.

Having a non-hardware item in a BSP is not treated as an error, since there may be applications where something like a memory-constrained BSP wants to turn off certain non-hardware items for size reasons alone.

- Items in one fragment that are re-specified again in another fragment or even in the same fragment later on.

Again this is not treated as an error, since there are several use cases where an over-ride is desired (e.g. the customer-supplied fragment described below). Normally there should be no need for doing this -- but if someone does this, the usual rule applies, that is, the last set value takes precedence.

- Hardware-related items that were requested in the BSP fragment(s) but not ultimately present in the final **.config** file.

Items like this are of the highest concern. These items output a warning as well as a brief pause in display output to enhance visibility.

- Invalid items that do not match any known available option.

This is for any **CONFIG_OPTION** item in a fragment that is not actually found in any of the currently available Kconfig files. Usually this reflects a use of data from an older kernel configuration where an option has been replaced, renamed, or removed.

See the following section for a commented example of auditing output.

Example of Kernel Fragment Auditing Output

Once you have configured your platform project, you can use the **.config** build rule to generate the initial audit directory contents.

```
$ make -C build linux-windriver.config
make: Entering directory `/mnt/Linux_build/Build/intel-x86-64-glibc_std/build'
...
Setting up host-cross and links
Setting up packages link
Setting up packages link
Creating export directory
Creating project properties
NOTE: Tasks Summary: Attempted 346 tasks of which 346 didn't need to be rerun and all
succeeded.
make: Leaving directory `/mnt/Linux_build/Build/intel-x86-64-glibc_std/build'
```

Once the build completes, you can examine the **projectDir/build/linux-windriver/temp/log.do_kernel_config_check** file to see a summary of the kernel configuration audit process.

```
$ cat build/linux-windriver/temp/log.do_kernel_configcheck
```

```
DEBUG: Executing python function do_kernel_configcheck
NOTE: validating kernel config, see_log.do_kernel_configcheck for details
DEBUG: The following new/unknown Kconfig files were found:
  arch/arm/mach-keystone/Kconfig
  drivers/clk/davinci/Kconfig
  drivers/clk/keystone/Kconfig
  drivers/dma/dw/Kconfig
  drivers/gpu/drm/rcar-du/Kconfig
  drivers/hwqueue/Kconfig
  drivers/staging/lttng2/Kconfig

[non-hardware (25)]: .meta/cfg/standard/intel-x86-64/specified_non_hdw.cfg
  This BSP sets config options that are possibly non-hardware related.

[invalid (5)]: .meta/cfg/standard/intel-x86-64/invalid.cfg
  This BSP sets config options that are not offered anywhere within this kernel

[errors (1)]: .meta/cfg/standard/intel-x86-64/fragment_errors.txt
  There are errors withing the config fragments.

[mismatch (9)]: .meta/cfg/standard/intel-x86-64/mismatch.cfg
  There were hardware options requested that do not
  have a corresponding value present in the final ".config" file.
  This probably means you aren't getting the config you wanted.

DEBUG: Python function do_kernel_configcheck finished
```

Duplicate instances of options, whether across fragments or in the same fragment, will generate a warning. You can view the indicated **fragment_errors.txt** file to see the specific options.

```
$ cat build/linux-windriver/linux/.meta/cfg/standard/intel-x86-64/fragment_errors.txt

Warning: Value of CONFIG_I2C_I801 is defined multiple times within fragment .meta/cfg/
kernel-cache/bsp/intel-x86/intel-x86.cfg:
CONFIG_I2C_I801=m
CONFIG_I2C_I801=y
```

Whenever duplicate options are encountered, only the last instance is included in the final configuration file.

The contents of the **invalid.cfg** file indicate which options are not being recognized. It may be that the options are incorrect or obsolete. An option that is spelled incorrectly may also trigger this warning. Note that any mis-spelled syntax, for example **CONFIG_OPTION=y** is ignored and unreported.

```
$ cat build/linux-windriver/linux/.meta/cfg/standard/intel-x86-64/invalid.cfg
CONFIG_MULTICORE_RAID456
CONFIG_SND_HDA_ENABLE_REALTEK_QUIRKS
CONFIG_SND_HDA_POWER_SAVE
CONFIG_WIRELESS_EXT_SYSFS
CONFIG_USB_SUSPEND
```

The non-hardware options are meant to be in the domain of the platform, not the BSP. The provided BSP options are found to be non-hardware-related and so they are reported here.

```
$ cat build/linux-windriver/linux/.meta/cfg/standard/intel-x86-64/invalid.cfg
CONFIG_MULTICORE_RAID456
CONFIG_SND_HDA_ENABLE_REALTEK_QUIRKS
CONFIG_SND_HDA_POWER_SAVE
CONFIG_WIRELESS_EXT_SYSFS
CONFIG_USB_SUSPEND
```

The **mismatch.cfg** file will indicate the option(s) causing this message. An example of a mismatch is a case where you have requested **CONFIG_OPTION=y** and you get the message:

```
$ cat build/linux-windriver/linux/.meta/cfg/standard/intel-x86-64/mismatch.cfg

Value requested for CONFIG_ACPI_CONTAINER not in final ".config"
```

```
Requested value: "CONFIG_ACPI_CONTAINER=m"
Actual value set: "CONFIG_ACPI_CONTAINER=y"

Value requested for CONFIG_ACPI_IPMI not in final ".config"
Requested value: "CONFIG_ACPI_IPMI=m"
Actual value set: ""

Value requested for CONFIG_ACPI_PCI_SLOT not in final ".config"
Requested value: "CONFIG_ACPI_PCI_SLOT=m"
Actual value set: "# CONFIG_ACPI_PCI_SLOT is not set"

Value requested for CONFIG_ASYNC_TX_DISABLE_PQ_VAL_DMA not in final ".config"
Requested value: "CONFIG_ASYNC_TX_DISABLE_PQ_VAL_DMA=y"
Actual value set: ""

Value requested for CONFIG_ASYNC_TX_DISABLE_XOR_VAL_DMA not in final ".config"
Requested value: "CONFIG_ASYNC_TX_DISABLE_XOR_VAL_DMA=y"
Actual value set: ""

Value requested for CONFIG_DCA not in final ".config"
Requested value: "CONFIG_DCA=m"
Actual value set: ""

Value requested for CONFIG_HOTPLUG_PCI_ACPI not in final ".config"
Requested value: "CONFIG_HOTPLUG_PCI_ACPI=m"
Actual value set: ""

Value requested for CONFIG_IGB_DCA not in final ".config"
Requested value: "CONFIG_IGB_DCA=y"
Actual value set: ""

Value requested for CONFIG_IXGBE_DCA not in final ".config"
Requested value: "CONFIG_IXGBE_DCA=y"
Actual value set: ""
```

In most cases, the option is not used because it is not valid for the input you provided.

The first example provides a case where you have an option **CONFIG_OPTION=m**, but you have not enabled modules. In this case, LKC would provide **CONFIG_OPTION=y**, assuming that was a valid option.

If you make changes to the layer content in your Linux platform project and wish to rerun the kernel audit portion of the build, it is first necessary to rerun the patch stage of the build to gather the kernel fragments from the layers.

```
$ BBOPTS="-f" make -C build linux-windriver.patch
make: Entering directory `/mnt/Linux_build/Build/intel-x86-64-glibc_std/build'
Inferred task 'patch' for recipe 'linux-windriver'.
Setting up host-cross and links

...

NOTE: Tainting hash to force rebuild of task /mnt/Linux_build/Build/intel-x86-64-glibc_std/layers/wr-kernel/recipes-kernel/linux/linux-windriver_3.10.bb, do_patch
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks Setting up host-cross and links Setting up packages link
link

Creating export directory Creating project properties
NOTE: Tasks Summary: Attempted 19 tasks of which 18 didn't need to be rerun and all succeeded.
make: Leaving directory `/mnt/Linux_build/Build/intel-x86-64-glibc_std/build'
```

After fragments are gathered into the kernel build directory, you can rerun the audit process with the `kernel_configme` command, and observe the results in log files as shown earlier.

```
$ make -C build linux-windriver.kernel_configme
make: Entering directory `/mnt/Linux_build/Build/intel-x86-64-glibc_std/build'
Inferred task 'kernel_configme' for recipe 'linux-windriver'.
Setting up host-cross and links
```



```
...  
NOTE: Tasks Summary: Attempted 20 tasks of which 19 didn't need to be rerun and all  
succeeded.  
make: Leaving directory `/mnt/Linux_build/Build/intel-x86-64-glibc_std/build'
```

Contents of the Audit Data Directory

The audit data directory contains the following files:

- all.cfg
- all.kcf input_non_hardware_configs.cfg
- always_nonhardware.cfg
- avail_hardware.cfg
- config_frag.txt
- config.log merge_log.txt
- fragment_errors.txt
- hardware.kcf
- hdw_frags.txt
- input_hardware_configs.cfg
- intel-x86-64-standard-config-3.10.19 redefined_as_board_specific.txt
- invalid.cfg redefinition.txt
- known_current.kcf required.cfg
- known.kcf required_configs.cfg
- mismatch.cfg
- mismatch.txt
- non-hardware.kcf
- non_hdw_frags.txt
- optional_configs.cfg always_hardware.cfg
- specified_hdw.cfg
- specified_non_hdw.cfg
- unknown.kcf
- verify.cfg

Patching the Kernel With SCC Files

Use this procedure to conveniently maintain kernel configuration changes and patches in a local layer in a single `.scc` file.

This procedure:

- Requires a configured and built platform project. See [Example Platform Project Configure for the Examples in this Chapter](#) on page 177.
- Requires the use of a single `.scc` script file to instruct BitBake to apply the kernel configuration fragment and the patch that were applied in [Patching the Kernel](#) on page 195.
- Assumes that you already have the required directory structure in the `projectDir/layers/local` directory.

An `.scc` file is a script file that provides information to the BitBake build system about what kernel changes to apply, and how to apply them. Use `.scc` files for grouping kernel changes, configuration fragments and patches. `.scc` files are a convenient way to track changes to the stock kernel provided by Wind River.

In this procedure, you will use a single `.scc` script file to instruct BitBake to apply the kernel configuration fragment and the patch that were applied in [Patching the Kernel](#) on page 195. Note that the instructions assume that you already have the required directory structure in the `projectDir/layers/local` directory.

For concepts related to using series configuration compiler (`.scc`) files, see [Kernel Patching with scc](#) on page 155.

Step 1 Update the `.bbappend` file to tell BitBake about the the location of the `.scc` file.

a) Open the `linux-windriver_3.4.bbappend` file in an editor, for example:

```
$ vi layers/local/recipes-kernel/linux/linux-windriver_3.4.bbappend
```

b) Update the following line, beginning with `SRC_URI` declaration, to add the location and name of the new patch file:

```
SRC_URI += "file://kernel_baseline.scc"
```

Here we modified the `SRC_URI` variable to include the file `kernel_baseline.scc`, and to let BitBake know that this file has to be processed at build time.

c) In `projectDir/layers/local/recipes-kernel/linux/linux-windriver` layer directory, use a text editor to create the `.scc` file.

```
$ vi layers/local/recipes-kernel/linux/linux-windriver/kernel_baseline.scc
```

d) Add the following lines and save the file:

```
kconf non-hardware config_baseline.cfg  
patch 0001-init-add-WR-to-the-boot-label.patch
```

The file `kernel_baseline.scc` contains the instructions needed to apply the kernel configuration changes and the `0001-init-add-WR-to-the-boot-label.patch` kernel patch previously applied in [Patching the Kernel](#) on page 195.

`.scc` files provide you with more control on how the kernel changes are applied. For additional information, see [Kernel Patching with scc](#) on page 155.

Step 2 Clean up and rebuild the Linux kernel package and file system.

a) Clean up the package.

```
$ make -C build linux-windriver.clean
```

Do this every time you make changes to the patches that you want to apply to the Linux kernel. This step forces the build system to subsequently reload all associated configuration files.

b) Rebuild the Linux kernel package.

```
$ make -C build linux-windriver
```

Once complete, a new `linux-windriver` package is available containing the modified kernel image to be used in the target.

c) Rebuild the file system.

```
$ make
```

This command updates the root file system to include the new structure of kernel modules to be loaded on the target.

Step 3 Verify that the `.scc` file changes were successfully applied to the target.

```
$ make start-target
```

Note the early boot message from the kernel console. It should read something similar to the following:

```
WR Linux version 3.4.6-WR5.0+snapshot-20120807_standard (revo@my-workstation-11)
  (gcc version 4.6.3 (Wind River Linux Sourcery CodeBench 4.6-60) ) #1 SMP PREEMPT
Tue Aug 7 12:33:23 EDT
```

Patching the Kernel

Use this procedure to maintain patches in a local layer that will be applied to the Linux kernel at build time.

- The following procedures require a configured and built platform project. See [Example Platform Project Configure for the Examples in this Chapter](#) on page 177.
- This exercise is an extension of the configuring kernel modules with fragments procedure in [Kernel Module Configuration and Patching with Fragments](#) on page 179 and assumes that you already have the required directory structure in the `projectDir/layers/local` directory.
- The following procedure assumes that you have initialized your git environment already using the command `git config --global` to setup the `user.name` and `user.email` variables. For additional information, see <http://git-scm.com/book/en/Getting-Started-First-Time-Git-Setup#Your-Identity>.

Maintaining individual kernel patches in a layer is one way in which you can track changes to the stock kernel provided by Wind River. In this procedure, you will apply a single patch to the kernel to modify the label used to display the kernel version during early booting.

Step 1 Update the `.bbappend` file to tell BitBake about the patch file.

a) Open the `linux-windriver_3.4.bbappend` file in an editor, for example:

```
$ vi layers/local/recipes-kernel/linux/linux-windriver_3.4.bbappend
```

b) Update the following line to add the location and name of the new patch file:

```
SRC_URI += "file://config_baseline.cfg \
file://0001-init-add-WR-to-the-boot-label.patch"
```

Here we modified the `SRC_URI` variable to include the file `0001-init-add-WR-to-the-boot-label.patch`, and to let BitBake know that this file has to be processed at build time.

c) Save the file.

Step 2 Edit the source code.

a) Open and edit the `version.c` file:

Run the following commands from the top-level platform project directory:

```
$ cd build/linux-windriver-3.4-r0/linux
```

```
$ vi init/version.c +48
```

b) Change this line to read:

```
"WR Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
```

c) Save the file.

Step 3 Commit the change.

Enter the following to commit the change:

```
$ git commit -m "init: add 'WR' to the boot label" init/version.c
```

Step 4 Create the patch.

a) Enter the following to create the patch:

As stated in Prerequisites, above, the following command assumes that you have initialized your git environment already using the command **git config --global** to setup the `user.name` and `user.email` variables. For additional information, see <http://git-scm.com/book/en/Getting-Started-First-Time-Git-Setup#Your-Identity>.

```
$ git format-patch -s -n \
```

```
-o projectDir/layers/local/recipes-kernel/linux/linux-windriver \
```

```
origin/standard/common-pc-64/base
```

Once applied, this patch modifies the banner message displayed in the Linux console early in the boot process. Instead of displaying:

```
Linux version ...
```

it will now display:

```
WR Linux version ...
```

b) Return to the top of your platform project directory.

```
$ cd ../../..
```

Step 5 Clean up and rebuild the Linux kernel package.

a) Clean up the package.

```
$ make -C build linux-windriver.clean
```

Do this every time you make changes to the patches that you want to apply to the Linux kernel. This step forces the build system to subsequently reload all associated configuration files.

b) Rebuild the Linux kernel package:

```
$ make -C build linux-windriver.rebuild
```

Once complete, a new **linux-windriver** package is available containing the modified kernel image to be used in the target.

Step 6 Verify that the patch has been applied.

You can look directly at the source file located in: **projectDir/build/linux-windriver/linux/init/version.c** to verify that the patch has been applied.

Additionally, it is important to note that the git repository of the kernel has been updated accordingly.

- a) View the git log.

Run the following commands from the platform project's directory:

```
$ cd build/linux-windriver/linux
```

```
$ git log
```

The system will return the following:

```
commit 4250412525031d95c3d60f4ccac00ea098ce6920
Author: Revo User <revo.user@windriver.com>
Date:   Wed Sep 26 10:16:35 2012 -0400

    init: add 'WR' to the boot label

Signed-off-by: Revo User <revo.user@windriver.com>
```

- b) Enter **q** to exit the **git log** command.
c) Return to the top of your platform project directory.

```
$ cd ../../..
```

The Linux kernel is deployed as a git repository in your working directory. BitBake has therefore committed the change using the message you entered in 3 on page 196, above, to commit the change.

Step 7 Launch the target to test the patch.

- a) Start the target.

```
$ make start-target
```

Note the early boot message from the kernel console. It should read something similar to the following:

```
WR Linux version 3.4.6-WR5.0+snapshot-20120807_standard (revo@my-workstation-11)
(gcc version 4.6.3 (Wind River Linux Sourcery CodeBench 4.6-60) ) #1 SMP PREEMPT
Tue Aug 7 12:33:23 EDT
```


17

Creating Alternate Kernels from kernel.org Source

Wind River provides the capability to build arbitrary git-based kernel sources using a development-only recipe. This recipe uses the Yocto infrastructure to clone and build directly from the desired kernel repository, starting from a user-specified tag and complete configuration.



NOTE: Only the kernel version supplied with Wind River Linux is validated and supported. Using any other kernel version is not covered by standard support.

This procedure is therefore suitable only for projects that are not under Wind River standard support, such as a Proof of Concept. It is expected that this procedure will build without errors with most BSPs, but it is unlikely the resulting kernel will boot without further configuration and patches.

Step 1 Update the platform project's **bblayers.conf** file to add kernel development support.

In a previously created Wind River Linux Platform project based on a standard kernel, (that is without CGL, RT-Linux or similar profile), add **projectDir/layers/wr-kernel/kernel-dev** to the file **projectDir/bitbake_build/conf/bblayers.conf**. This makes the **linux-windriver-custom** recipe available to the build.

For example:

```
$ echo 'BBLAYERS += "${WRL_TOP_BUILD_DIR}wr-kernel/wr-kernel-dev" ' \
>> projectDir/bitbake_build/conf/bblayers.conf
```

Step 2 Create a **.bbappend** file in the local layer of your build.

```
$ cd projectDir/layers/local
$ mkdir -p recipes-kernel/linux
$ cd recipes-kernel/linux/
$ echo 'FILESEXTRAPATHS := "${THISDIR}/${PN}"' >> linux-windriver-custom.bbappend
```

Step 3 Update the **SRCREV** for the kernel version being built.

This is the git hash of a tag in the kernel.org tree. The kernel will be cloned from the kernel.org git repository so it is necessary to have downloading enabled in your **local.conf** file.

For example, to build the Linux 3.6 kernel the tag is:

```
$ echo 'SRCREV = "a0d271cbfed1dd50278c6b06bead3d00ba0a88f9"' >> linux-windriver-  
custom.bbappend
```

Step 4 Make the recipe compatible with your machine.

```
$ echo 'COMPATIBLE_MACHINE = "${MACHINE}"' >> linux-windriver-custom.bbappend
```

Step 5 Add kernel-specific configuration and patches.

When building a particular kernel version, you will also need to add kernel configuration and patches that are specific to the new kernel version. These additional files can be placed in the local layer recipe directory you have just created.

For example, to add a **config/defconfig** fragment for the board to the **SRC_URI**:

```
$ mkdir -p linux-windriver-custom  
$ cp /path/to/my/custom_defconfig linux-windriver-custom/defconfig  
$ echo 'SRC_URI += "file://defconfig"' >> linux-windriver-custom.bbappend
```

Step 6 Move to the root of your project and edit your **local.conf** file.

Change your **PREFERRED_PROVIDER_virtual/kernel_BSP_name** definition so it selects your custom kernel. For example:

```
PREFERRED_PROVIDER_virtual/kernel_qemuppc = "linux-windriver-custom"
```

Step 7 Build the kernel.

Run the following command from the top-level folder in the **projectDir**:

```
$ make -C build linux-windriver-custom
```

If you are reasonably sure your kernel is compatible, you can build it into your file system image using:

```
$ make fs
```



NOTE: This procedure only replaces the kernel and not the file system components. Most notably the kernel-headers package that is exported to the SDK sysroot remains unchanged.

18

Exporting Custom Kernel Headers

[About Exporting Custom Kernel Headers for Cross-compile](#) 201

[Adding a File or Directory to be Exported when Rebuilding a Kernel](#) 201

[Exporting Custom Kernel Headers](#) 202

About Exporting Custom Kernel Headers for Cross-compile

It is possible to export custom kernel headers for application development cross-compilation using a built-in task for the Linux kernel.

The Wind River Linux kernel includes the `linux-windriver.install_kernel_headers` task, which enables developers to export their custom kernel headers to the sysroot for use in cross-compiling user space code. This task is provided by default and does not require any specific platform project configuration option. This task runs after `linux-windriver.do_install()` and before `linux-windriver.do_populate_sysroot`. Any header files and directories listed in the global variable `KERNEL_INSTALL_HEADER` are copied to the sysroot.

Each entry in `KERNEL_INSTALL_HEADER` is expected to exist in the Linux kernel source `include/` directory. If a file already exists in the destination, the build system will not overwrite it, but instead issue a warning. For example, to include a header file named `myfile.h`, the file must exist in the `projectDir/build/linux-windriver-3.4-r0/linux/include` directory, or a subdirectory of it. See [Adding a File or Directory to be Exported when Rebuilding a Kernel](#) on page 201 for examples of using `add KERNEL_INSTALL_HEADER_append`.

For instructions on exporting a custom kernel header, see [Exporting Custom Kernel Headers](#) on page 202.

Adding a File or Directory to be Exported when Rebuilding a Kernel

Append files or directories to the `KERNEL_INSTALL_HEADER` variable each time the kernel is rebuilt as shown in these examples

This procedure is a supplement to [Exporting Custom Kernel Headers](#) on page 202.

Each entry in the `KERNEL_INSTALL_HEADER` variable is expected to exist in the Linux kernel source **include/** directory. To add a file or directory to be exported each time you rebuild the kernel, use `KERNEL_INSTALL_HEADER_append` to add to the variable as illustrated in the following example.

This variable is not configuration file-specific, and can be added to any of your layer configuration files, such as

`projectDir/layers/local/conf/layers.conf`

Step 1 Open the **`projectDir/layers/local/conf/layers.conf`** file in a text editor.

Step 2 Update the `KERNEL_INSTALL_HEADER` variable.

- To add a single file, such as **myfile.h**:

```
KERNEL_INSTALL_HEADER_append += "myfile/myfile.h"
```

- To add all files in a directory:

```
KERNEL_INSTALL_HEADER_append += "myfile"
```

Step 3 Save the file.

Exporting Custom Kernel Headers

Use this procedure to export custom kernel headers for application development cross-compilation

This procedure requires a previously configured platform project. For additional information, see [About Configuring a Platform Project Image](#) on page 62. It also requires that any custom kernel header files that you want to export be located in the **`projectDir/build/linux-windriver-3.4-r0/linux/include`** directory or a subdirectory. For additional information, see [About Exporting Custom Kernel Headers for Cross-compile](#) on page 201.

Step 1 Unpack the Linux kernel.

Run the following command in the root of the **`projectDir`**.

```
$ make -C build linux-windriver.patch
```

Step 2 Navigate to the source directory of the kernel build.

```
$ cd build/linux-windriver-3.4-r0/linux
```

Step 3 Optionally create a file to test this procedure.

If you do not have a file in the **`projectDir/build/linux-windriver-3.4-r0/linux/include`** directory, you can use the following line to create one for testing purposes:

```
$ echo "#define my file" > include/myfile.h
```

Step 4 Add and commit your file to the git repository for the kernel.

```
$ git add include/myfile.h  
$ git commit -m "new #define my file"
```

For the commit message, you can enter anything you like, specific to your custom header file.

Step 5 Open the `projectDir/ayers/local/conf/ayers.conf` file in a text editor.

Step 6 Add the header file to the `projectDir/ayers/local/conf/ayers.conf` file and save the file.

```
$ KERNEL_INSTALL_HEADER_append += "myfile.h"
```

This will include your custom header file in the build. For additional information on adding header files, see [Adding a File or Directory to be Exported when Rebuilding a Kernel](#) on page 201.

Step 7 Rebuild the kernel:

```
$ make -C build linux-windriver
```

This can take some time to complete. When it finishes, your custom header file will be located in the `projectDir/bitbake_build/tmp/sysroots/BSP_name/usr/include` directory.

For a `qemux86-64` BSP, the path would be `projectDir/bitbake_build/tmp/sysroots/qemux86-64/usr/include/myfile.h`. This places your custom header file in the appropriate directory for user space cross-compiling.

19

Using the preempt-rt Kernel Type

[Introduction to Using the preempt-rt Kernel Type](#) 205

[Enabling Real-time](#) 207

[Configuring preempt-rt Preemption Level](#) 207

Introduction to Using the preempt-rt Kernel Type

Wind River Linux provides a conditional real-time kernel type, **preempt-rt**, for certain board and file system combinations.

The default scheduler for **preempt-rt** is the Completely Fair Scheduler (CFS). For information on configuring preemption levels, see *About Preemption Model Configuration*.



NOTE: Conditional real-time support is not available for all boards.

The preempt-rt kernel type provides four levels of preemption to suit most platform project requirements, as described in this section. These options are available in the **Kernel Configuration > Processor type and features > Preemption model (Fully Preemptible Kernel (RT))** menu as described in *Configuring preempt-rt Preemption Level* on page 207.



NOTE: The **Processor type and features** selection is specific to using an x86 architecture. Different architectures may have different wording for this selection. Refer to Kernel Configuration documentation specific to your architecture for additional information.

No Forced Preemption (Server)

The text kernel configuration entry is **CONFIG_PREEMPT_NONE**. This is the traditional Linux preemption model geared towards throughput. It will provide reasonable overall response latencies but there are no guarantees and occasional long delays are possible. This configuration will maximize the raw processing throughput of the kernel irrespective of scheduling latencies.

Voluntary Kernel Preemption (Desktop)

The text configuration entry is `CONFIG_PREEMPT_VOLUNTARY`. This configuration reduces the latency of the kernel by adding more explicit preemption points to the kernel code. The new preemption points break long non-preemptive kernel paths, minimizing rescheduling latency and providing faster application reactions, at the cost of slightly lower throughput. This offers faster reaction to interactive events by enabling a low priority process to voluntarily preempt itself during a system call. Applications run more smoothly even when the system is under load. A desktop system is a typical candidate for this configuration.

Preemptible Kernel (Low-latency Desktop)

This configuration applies to embedded systems with latency requirements in the milliseconds range.

The text configuration entry is `CONFIG_PREEMPT_LL`. This configuration further reduces kernel latency by allowing all kernel code that is not executing in a critical section to be preemptible. This offers immediate reaction to events. A low priority process can be preempted involuntarily even during syscall execution. This is similar to `CONFIG_PREEMPT_VOLUNTARY`, but allows preemption anywhere outside of a critical (locked) code path.

Applications run more smoothly even when the system is under load, at the cost of slightly lower throughput and a slight run-time overhead to kernel code. (According to profiles when this mode is selected, even during kernel-intense workloads the system is in an immediately preemptible state more than 50% of the time.)

Preemptible Kernel (Basic RT)

This configuration applies to embedded systems with latency requirements in the milliseconds range.

The text configuration entry is `CONFIG_PREEMPT_RT_B`. This configuration is similar to `CONFIG_PREEMPT_LL`, but it enables changes that are considered the preliminary configuration for `CONFIG_PREEMPT_RT_FULL`.

With this mode selected, a system can be in an immediately preemptible state more than 70% of the time, even during kernel-intense workloads.

Fully Preemptible Kernel (RT)

This configuration applies to time-response critical embedded systems, with guaranteed latency requirements of 100 usecs (microseconds) or lower.

The text configuration entry is `CONFIG_PREEMPT_RT_FULL`. This configuration further reduces the kernel latency by replacing virtually every kernel spinlock with preemptible (blocking) mutexes, and allowing all but the most critical kernel code to be involuntarily preemptible. The remaining low-level, non-preemptible code paths are short and have a deterministic latency of a few tens of microseconds, depending on the hardware. This enables applications to run smoothly irrespective of system load, at the cost of lower throughput and run-time overhead to kernel code.

Selecting the fully preemptible kernel automatically includes the preemptible RCU configuration parameter. The text configuration entry is `CONFIG_PREEMPT_RCU`. This option reduces the latency of the kernel by making certain RCU sections preemptible. Normally RCU code is non-preemptible. If this option is selected, read-only RCU sections become preemptible. This helps latency, but may expose bugs due to now-naive assumptions about each RCU read-side critical section remaining on a given CPU through its execution.

Testing indicates that with this mode selected, a system can be in an immediately preemptible state more than 95% of the time, even during kernel-intense workloads.

Applications running on a `CONFIG_PREEMPT_RT_FULL` kernel need to be aware that in some cases they may be competing with kernel services running in scheduled task context. Various legacy test suites exercising privileged real-time scheduling policies at high priorities have also been found to fail, and in some cases have caused system lockup due to the changed scheduling dynamics in the kernel.

These conditions are a result of kernel code which had been running in hard-exception context now running in task-scheduled context. The cause of this issue is the ability of a privileged application or test task to elevate its scheduling priority above system daemons. The potential exists for such a task to halt system scheduling if it does not relinquish the CPU.

The work-around is to assure system daemons schedule with a priority greater than any application task. This may be accomplished by either a `chrt` of the system daemons above the expected priority range of application usage, or constraining the application to use priorities below that of system daemons.

Enabling Real-time

To enable the preemptible real-time feature, configure your project with the `preempt-rt` kernel option.

This procedure requires that you have previously created a platform project build directory. For additional information, see [About Creating the Platform Project Build Directory](#) on page 62.

Step 1 Configure the platform project with the `--enable-kernel=preempt-rt` option.

For example, to configure a `qemux86-64` board with a standard file system and conditional real-time, enter:

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=preempt-rt \
--enable-rootfs=glibc_std
```

See [About Configure Options](#) on page 65 for additional information on `configure` script options.

Step 2 Build the project.

```
$ make
```

Configuring preempt-rt Preemption Level

You may configure the real-time kernel to run in one of four levels of increasingly aggressive preemption behavior.

This section explains how to launch `menuconfig` to configure preempt-rt kernel parameters, and make changes to your preemption levels.



NOTE: These instructions describe command-line procedures for configuring your preemption levels. See the *Wind River Workbench by Example Guide (Linux version)* for instructions on using Workbench to configure preemption.

To perform the following procedure, you must have a platform project image configured and built using the `--enable-kernel=preempt-rt configure` option. See [Enabling Real-time](#) on page 207.

Step 1 Build and open a kernel development shell (kds).

Run the following command from the `projectDir`:

```
$ make kds
```

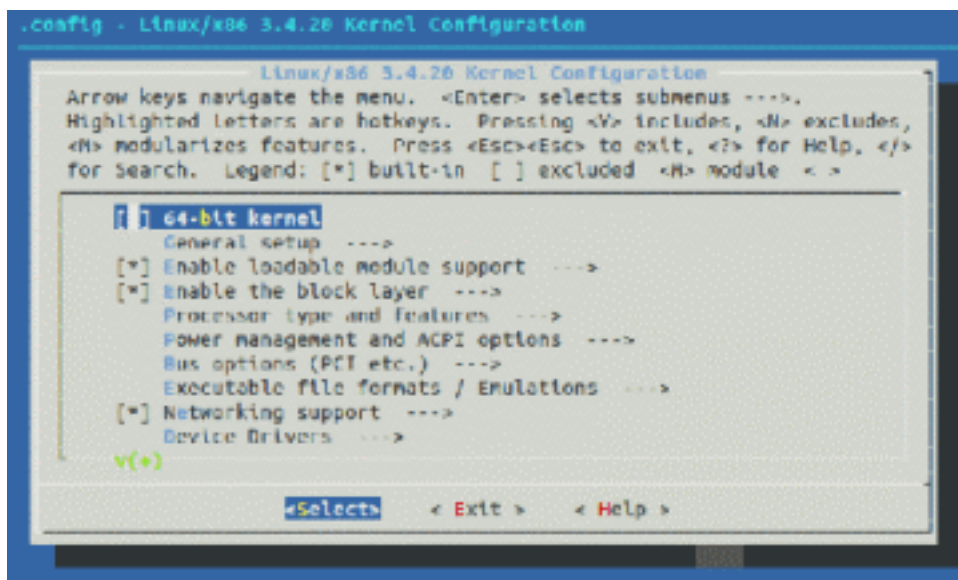
Once the command completes, it will launch a new kds in a separate window.

Step 2 Open the platform project's kernel configuration menu.

Run the following command in the kds terminal:

```
$ make menuconfig
```

The Kernel Configuration graphical interface launches:



Step 3 Navigate to the **Preemption Models** selection options.

Select **Processor type and features > Preemption model (Fully Preemptible Kernel (RT))** to view the Preemption Models options.



NOTE: The first selection, **Processor type and features**, is specific to using an x86 architecture. Different architectures may have different wording for this selection. Refer to Kernel Configuration documentation specific to your architecture for additional information.

Step 4 Select a preemption model.

Use the arrow keys to select a preemption model, then press **SPACE** to select and automatically return to the previous menu.

See *About Preemption Model Configuration* for additional information on the different preemption models.

Step 5 Press **ESC** twice to return to the main menu.

Step 6 Optionally, set the debug functionality you want to include in your kernel.

Select **Kernel hacking** from the main menu, then highlight **Debug preemptible kernel** and press **SPACE** to select it. Press **ESC** twice to return to the main menu.

This option enables the kernel to detect preemption count underflows, track critical section entries, and emit debug assertions should an illegal sleep attempt occur. Unsafe use of `smp_processor_id()` is also detected. The text configuration entry for this option is `CONFIG_DEBUG_PREEMPT`.

Step 7 Press **ESC** and select **Yes** at the prompt to save your configuration.

Step 8 Enter **exit** to close the kds window.

Step 9 Rebuild the kernel with the new configuration.

Run the following command in the *projectDir*:

```
$ make -C build linux-windriver.rebuild
```

Step 10 Rebuild the platform project and add the new kernel module changes.

```
$ make
```

PART V

Debugging and Enabling Analysis Tools Support

Kernel Debugging.....	213
Userspace Debugging.....	219
Analysis Tools Support.....	231

20

Kernel Debugging

[Kernel Debugging](#) 213

[Debugging with KGDB Using an Ethernet Port \(KGDBOE\)](#) 214

[Debugging with KGDB Using the Serial Console \(KGDBOC\)](#) 216

[Disabling KGDB in the Kernel](#) 217

[Kernel Debugging with QEMU](#) 218

Kernel Debugging

Understand the limitations for using **gdb** to perform KGDB debugging.

This information in this section is specific to using **gdb** from the command line. To perform KGDB debugging with Workbench, see the *Wind River Workbench by Example, Linux Version*.

You may find it useful to make a KGDB connection from the command line using **gdb** for several reasons:

- You are more familiar with **gdb** for particular types of debugging,
- You wish to automate some KGDB tests.
- You are having problems with your KGDB connection from Workbench.

Known KGDB Limitations

Before you begin, there are some known limitations with using KGDB. The following is not supported:

- Serial console (KGDBoC) over USB interface through one USB-serial gadget. See [Debugging with KGDB Using the Serial Console \(KGDBOC\)](#) on page 216.
- Ethernet (KGDBoE) on SMP systems with threaded interrupts.

Always refer to the BSP **README** for the most up to date info regarding any limitations or restrictions.

Debugging with KGDB Using an Ethernet Port (KGDBOE)

KGDBOE permits KGDB debugging operations over an Ethernet port. By default, KGDBOE is available as a module in the WR Linux kernel.

Perform the procedure in this topic to use KGDB debugging operations over an Ethernet port using **gdb**.

Step 1 Launch a platform project image on a hardware target.

Step 2 Install the kernel module.

Run the following command in the target's console:

```
# modprobe kgdboe kgdboe=@/,@host_ip_adress/
```

See the kernel documentation for the full details on the option syntax.



NOTE: KGDBOE is only available on Ethernet drivers that support the Linux Netpoll API. If this is not the case for your board the **modprobe** command above will fail with the message:

```
kgdboe: netpoll_setup failed kgdboe failed
```

Step 3 Run the cross-compiled version of **gdb** on your **vmlinux** image.

a) Go to your project directory.

```
$ cd projectDir
```

This makes it easier to provide the path to the **vmlinux** symbol file

b) Run the cross-compiled version of on your **vmlinux** image.

```
$ ./scripts/gdb export/images/vmlinux-symbols-qemux86-64
```

For some boards, you need to assert the architecture for **gdb**.

- For the 8560, for example, it is necessary to specify:

```
(gdb) set architecture powerpc:common
```

- For a MIPS-64 CPU board with a 32-bit kernel, it is necessary to specify:

```
(gdb) set architecture mips
```



NOTE: Without this setting, **gdb** may continually respond with errors such as the following and other errors.

```
Program received signal SIGTRAP, Trace/breakpoint trap. 0x00000000 in ?? ()
```

Step 4 In the **gdb** session, connect to the target.

Port 6443 is reserved for KGDB communication.

```
(gdb) target remote udp:target IP:6443
```

➔ **NOTE:** You may see various warnings and exceptions that you can ignore. If, however, **gdb** informs you that the connection was not made, review your configuration, command syntax, and the IP addresses used.

Note that after issuing this command, and if the connection to the target is successful, the target will halt.

Step 5 Enter the **where** command, and note the output.

```
(gdb) where
```

You should see a backtrace stack of some depth. If you see only one or two entries, or a **??**, then you are observing an error.

Step 6 Enter the **info registers** command, and note the output.

a) Type the following **gdb** command:

```
(gdb) info registers
```

You should see the list of registers.

b) Examine the list of registers.

If, for example, the program counter is zero or otherwise unreasonable, then you are observing an error.

Step 7 Enter a breakpoint command for **do_fork**.

```
(gdb) break do_fork
```

➔ **NOTE:** If the **do_fork** location as a breakpoint does not work on this installation, note it and choose another kernel entry point. See *Wind River Workbench by Example, Linux Version: Debugging Kernel Space* for more information.

Step 8 Continue the target execution.

```
(gdb) c
```

➔ **NOTE:** Note that the target resumes normal operation.

Step 9 On the host, verify that the program stopped at the **do_fork** breakpoint.

a) On the target, type **ls** then press RETURN.

The host displays the status of the **do_fork** breakpoint:

```
Breakpoint 1, do_fork ... linux/kernel/fork.c:16011601 if (clone_flags &
CLONE_NEWUSER) {
```

b) Perform additional debugging operations as necessary.

- You can resume the target's operation using the **gdb** command **c** (for continue) again.
- You can press **CTRL+C** to send a break, set breakpoints, view the stack, view variables, and so on.



NOTE: You may wish to build the kernel with `CONFIG_DEBUG_INFO=y` if you want more debugging info.

Step 10 Release the KGDB connection.

When you are finished debugging, enter the following command to disconnect `gdb` from the target:

```
(gdb) disconnect
(gdb) quit
```



WARNING: If you quit `gdb` without first disconnecting from the target, you may have to reboot the target before you can reconnect.

You may also lose Telnet and other communication, especially if the target was stopped at a breakpoint.

Debugging with KGDB Using the Serial Console (KGDBOC)

The following procedure illustrates the use of KGDB debugging operations from a serial console.

KGDBOC permits KGDB debugging operations using the serial console. The serial console operates in two modes—the usual mode in which you use the serial console to login and so on, and a mode that allows you to enter the KGDB debugger.

If your hardware does not support the line break sequence or agent-proxy is connected to your target as a debug splitter, you will have to start the agent-proxy with the `-s003` option. (Workbench users set the **Work Bench Linux KGDB Connection** properties to select **Use character based break**.) If your target continues to run after sending a break command, you most likely need to employ one of these methods.

The following example assumes that the board's console port is connected to the development workstation through a serial-to-USB adapter which can be accessed through the `/dev/ttyUSB0` device:

Step 1 Launch the agent-proxy from within your project's directory.

```
$ host-cross/usr/bin/agent-proxy 2223^2222 localhost /dev/ttyUSB0,115200
```

Step 2 Connect to the board's console port.

```
$ telnet localhost 2223
```

To display the `#` prompt of the target, press ENTER.

Step 3 Find the device file used as console by inspecting the kernel's boot command line.

Run the following command on the target:

```
# telnet localhost 2223
```


The command returns the following output:

```
... console=ttyO2,115200n8 mpurate=auto ...
```

Step 4 Configure kgdboc to use the console device.

```
# echo ttyO2 > /sys/module/kgdboc/parameters/kgdboc
```

The console returns a confirmation:

```
kgdb: Registered I/O driver kgdboc.
```

Step 5 Enter kdb mode by sending the **sysrq-g** magic sequence:

```
# echo g > /proc/sysrq-trigger
```

The console returns:

```
SysRq : DEBUG  
Entering kdb (current=0xde63da40, pid 543) due to Keyboard Entry  
kdb>  
kgdb: Registered I/O driver kgdboc.
```

Step 6 Enter kgdb mode from the **kdb** prompt.

```
kdb> kgdb
```

The console returns a confirmation:

```
Entering please attach debugger or use $D#44+ or $3#33
```

Step 7 Launch the **gdb** debugger.

Run the following command on the host workstation:

```
$ ./scripts/gdb export/images/vmlinux-symbols-beagleboard
```

The host console displays the **gdb** prompt.

Step 8 Connect **gdb** to the target:

```
(gdb) target remote localhost:2222
```

You can start now your debugging session using all available **gdb** commands.

You can use the **gdb** command **c** (for continue) followed by **CTRL+C** to resume and stop execution on the target.

Disabling KGDB in the Kernel

Learn how to disable KGDB to begin a transition to production builds.

By default, KGDB is enabled in the pre-built and generated Wind River Linux kernels, but can be disabled if necessary. Typically, production-level builds no longer require kernel debugging support, so you can use the following procedure to disable it..

Step 1 Set up the configuration support files of the kernel.

Run the following command in the **projectDir**:

```
$ make -C build linux-windriver.menuconfig
```

This command launches the Configuration tool.

Step 2 Disable debugging info in the kernel.

Select **Kernel hacking > Compile the kernel with debug info** and press **SPACE** to disable.

Step 3 Exit the configuration tool.

Tab to the bottom menu and select **Exit > Exit**. Select **Yes** when prompted to save your changes.

Step 4 Rebuild the kernel.

```
$ make -C build linux-windriver.rebuild
```

A new kernel is built and **vmlinux** symbol table file created in the **export** directory.

Remember these files for Workbench and the command line testing.

Kernel Debugging with QEMU

Learn how to start QEMU from the command line and load the KGDB kernel modules.

The following example procedure assumes you have built a platform project for one of the supported boards.

When you have created the platform project, you can start QEMU from the command line and load the KGDB kernel modules shown in the following procedure.

After the module is loaded you can, for example, connect to the kernel using Workbench as described in *Workbench by Example, Linux Version*.

Step 1 Launch a QEMU target.

Run the following command from the **projectDir**:

```
$ make start-target
```

See [QEMU Targets](#) on page 241 for more information.

Step 2 Start **gdb** on your workstation.

```
$ ./scripts/gdb export/images/vmlinux-symbols-qemux86-64
```

Step 3 Connect to the emulated target.

```
(gdb) target remote :1234
```

From this moment on, you can use the **gdb** command **c** (for continue) followed by **CTRL+C** to resume and stop execution on the target.



NOTE: Refer to *Wind River Workbench by Example, Linux Version* for details on loading Ethernet as well as Serial KGDB target modules on physical targets.

21

Userspace Debugging

- [Adding Debugging Symbols to a Platform Project](#) 219
- [Adding Debugging Symbols for a Specific Package](#) 220
- [Dynamic Instrumentation of User Applications with uprobes](#) 221
- [Debugging Individual Packages](#) 228
- [Debugging Packages on the Target Using gdb](#) 228
- [Debugging Packages on the Target Using gdbserver](#) 229

Adding Debugging Symbols to a Platform Project

Learn how to add debugging symbols to your binaries for debugging on the target.

Whether you are debugging on the target directly or remotely from the development workstation, you will want to have debugging symbols for your binaries. The easiest method to use to add debugging symbols is to use the `--enable-build=debug configure` script option when you create your platform project.

Once you build the project with the `make` command, this option installs debug symbols in the form of a `*.debuginfo` archive file, located in the `projectDir/export` folder. For example:

```
projectDir/export/qemux86-64-glibc-small-standard-dist-debuginfo.tar.bz2
```

If you use this option, the `*.debuginfo` file and symbols are automatically added to the target root file system, so there is no need to extract this file on the target to perform local debugging.

These symbols are located in `.debug` subdirectories, along with the location of the corresponding binaries. For example, the debug information for binaries in `/usr/bin` is in the directory

`/usr/bin.debug`

If you used the `--enable-build=profiling configure` script option, note that this also adds symbols in your file system for debugging, but requires you to manually extract the symbols to the target file system to perform debugging.

Step 1 Verify whether debugging symbols have been built for your platform project.

Navigate to the **projectDir/export** folder to see whether the ***.debuginfo** archive exists. For example:

```
projectDir/export/qemux86-64-glibc-small-standard-dist-debuginfo.tar.bz2
```

The full name of the archive is based on the BSP, root file system, and kernel type, and may differ depending on your platform project configuration. If your platform project includes this archive, it is ready to perform userspace debugging. If not, you can add them in the following step.

Step 2 Add debugging symbols.

Options	Description
Previously built platform project	<ol style="list-style-type: none"> 1. Perform a distclean on the root file system. Run the following command from the projectDir: <pre>\$ make -C build wrlinux-image-filesystem.distclean</pre>where make -C build wrlinux-image-filesystem refers to the name of the projectDir/layers/local/recipes-img/images/wrlinux-image-file-system.bb recipe file, for example, make -C build wrlinux-image-glibc-std.distclean. This can take a few moments to complete. 2. Create the *.debuginfo archive. <pre>\$ make fs-debug</pre>
Previously configured platform project	<p>Run the following command in the projectDir:</p> <pre>\$ make fs-debug</pre>

Once the command completes, it creates the ***.debuginfo** archive in the project **export** directory.

Adding Debugging Symbols for a Specific Package

Depending on your development needs, you may only need to create debugging symbols for a specific package, and not the entire platform project.

For platform projects with the **glibc_small roots**, you can pre-configure or add symbols after the root file system has built.

- To add symbols to the root file system, take one of the following actions:

Options	Description
--with-package=busybox-dbg	Use the configure option at configuration time
make -C build busybox-dbg.addpkg	Use the command after the project is built

Dynamic Instrumentation of User Applications with uprobes

There are number of tracing options provided with Wind River Linux. This example focuses on **uprobes**. The 'u' indicates "user," and **uprobes** are designed to trace applications and user libraries, whereas many other types of Linux instrumentation are focused on the kernel.

Other instrumentation libraries provided with Wind River Linux that examine applications are:

ptrace

Used by GDB and Wind River user mode agent for debugging

LTTng (The Linux trace toolkit)

Records system calls for the workbench System Viewer

Wind River Profiler

Periodically records the contents of both user and kernel stacks

In the following examples, we will be doing profiling. It is far more convenient to use the Wind River Profiler than the following method, and the Profiler is functional with striped binaries on the target file system because it obtains debug information through object path mapping on the host. However, the **uprobes** method has the advantage of being entirely target based and has no dependence on development host tool connectivity.

The **uprobe** library provides a mechanism for a kernel function to be invoked whenever a process executes a specific instruction location. An interface to **uprobes** is provided through the **perf** events subsystem, accessed from the shell with the **perf probe** command.

When a **uprobe** is inserted in a program, a special copy is made of the page containing the probe. In that copy the instruction is replaced by a breakpoint. When the breakpoint is hit by a running process, the event is recorded and the program continues normal operation.

While the kernel event tracing system is the default user of **uprobes**; there is also a published interface you can use for your own custom tools. At the core of **uprobes** is this function:

```
#include <linux/uprobes.h>

int uprobe_register(struct inode *inode, loff_t offset, struct uprobe_consumer *uc);
```

The **inode** structure points to an executable file; the probe is placed at offset bytes from the beginning. The **uprobe_consumer** structure provides the callback mechanism for when the process encounters the probe; it looks like:

```
struct uprobe_consumer {
    int (*handler) (struct uprobe_consumer *self, struct pt_regs *regs);
    bool (*filter) (struct uprobe_consumer *self, struct task_struct *task);
    struct uprobe_consumer *next;
};
```

The **filter()** function is optional; if it exists, it determines whether **handler()** is called for each specific hit on the probe. The handler returns an **int**, but the return value is ignored in the current code.

uprobe Syntax

The **uprobe** syntax is similar the **kprobe** syntax, but because only minimal process symbol information is available to the kernel, it is typical to specify the probe location with an offset.

Table 6 Synopsis of uprobe_tracer

Parameter	Definition
p[:[GRP/]EVENT] PATH:SYMBOL[+offs] [FETCHARGS]	Sets a probe
GRP	Group name. If omitted, use "uprobes" as default.
EVENT	Event name. If omitted, the event name is generated based on SYMBOL[+offs] .
PATH	Path to an executable of a library.
SYMBOL[+offs]	Symbol+offset where the probe is inserted.
FETCHARGS	Arguments. Each probe can have up to 128 arguments.
%REG	Fetch register REG .

This comes from the Linux kernel documentation directory of your platform project in the file:

projectDir/build/BSP_name-wrs-linux/linux-windriver-version/linux/ Documentation/ trace/uprobracer.txt

The format of the **perf probe** command is not consistent across versions of Linux. The version in Wind River Linux supports the following options:

```
usage: perf probe [<options>] 'PROBEDEF' ['PROBEDEF' ...]
or: perf probe [<options>] --add 'PROBEDEF' [--add 'PROBEDEF' ...]
or: perf probe [<options>] --del '[GROUP:]EVENT' ...
or: perf probe --list

-v, --verbose      be more verbose (show parsed arguments, etc)
-l, --list         list up current probe events
-d, --del <[GROUP:]EVENT>
                  delete a probe event.
-a, --add <[EVENT=]FUNC[+OFF|%return] [[NAME=]ARG ...]>
                  probe point definition, where
                  GROUP:      Group name (optional)
                  EVENT:      Event name
                  FUNC:       Function name
                  OFF:        Offset from function entry (in byte)
                  %return:    Put the probe at function return
                  ARG:        Probe argument (kprobe-tracer argument format.)

-f, --force       forcibly add events with existing name
-n, --dry-run     dry run
--max-probes <n> Set how many probe points can be found for a probe.
-F, --funcs       Show potential probe-able functions.
--filter <[!]FILTER>
                  Set a filter (with --vars/funcs only)
                  (default: "!__k???tab_* & !__crc_*" for --vars,
                   "!*" for --funcs)
-x, --exec <executable|path>
                  target executable name or path
```

Configuring uprobes with perf

Install **perf** to enable **uprobe** debugging

uprobes requires the **perf** package. By default, **uprobes** are already included in your kernel configuration if your BSP supports this functionality in the **Kernel Hacking > Tracers** section of the kernel configuration.

Option	Name	y/n/m
Tracers	FTRACE	y
Kernel Function Tracer	FUNCTION_TRACER	y
Interrupts-off Latency Tracer	IRQSOFF_TRACER	n
Preemption-off Latency Tracer	PREEMPT_TRACER	n
Scheduling Latency Tracer	SCHED_TRACER	n
Trace syscalls	FTRACE_SYSCALLS	y
Branch Profiling		No branch
Trace max stack	STACK_TRACER	n
Support for tracing block IO actions	BLK_DEV_IO_TRACE	y
Enable kprobes-based dynamic events	KPROBE_EVENT	y
Enable uprobes-based dynamic events	UPROBE_EVENT	y
enable/disable ftrace tracepoints dynamically	DYNAMIC_FTRACE	y
Kernel function profiler	FUNCTION_PROFILER	n
Perform a startup test on ftrace	FTRACE_STARTUP_TEST	n
Memory mapped IO tracing	MMIOTRACE	n

Figure 5: CONFIG_UPROBE_EVENT=y

- Enter the following commands from the root of your project directory tree.

```
$ make -C build perf.addpkg
$ make -C build perf
$ make fs
```

These commands have no negative side effect if the package is already present in your platform project.

Dynamically Obtain User Application Data with uprobes

Use **uprobe** to dynamically obtain application and library data on the **perf** application.

This procedure requires the following for successful completion:

- Previously configured and built platform project with debugging symbols. See [Adding Debugging Symbols to a Platform Project](#) on page 219.
- The **perf** package included in your platform project build. For additional information on adding packages, see [Adding Debugging Symbols to a Platform Project](#) on page 219.
- The CONFIG_UPROBE_EVENT=y kernel configuration parameter enabled in the kernel.

By default, uprobes are included in your kernel configuration, but may not be supported by all BSPs. To see whether your BSP supports this functionality, refer to the **Kernel Configuration > Kernel Hacking > Tracers** section of the utility. For example:

Option	Name	y/n/m
Tracers	FTRACE	y
Kernel Function Tracer	FUNCTION_TRACER	y
Interrupts-off Latency Tracer	IRQSOFF_TRACER	n
Preemption-off Latency Tracer	PREEMPT_TRACER	n
Scheduling Latency Tracer	SCHED_TRACER	n
Trace syscalls	FTRACE_SYSCALLS	y
Branch Profiling		No branch
Trace max stack	STACK_TRACER	n
Support for tracing block IO actions	BLK_DEV_IO_TRACE	y
Enable kprobes-based dynamic events	KPROBE_EVENT	y
Enable uprobes-based dynamic events	UPROBE_EVENT	y
enable/disable ftrace tracepoints dynamically	DYNAMIC_FTRACE	y
Kernel function profiler	FUNCTION_PROFILER	n
Perform a startup test on ftrace	FTRACE_STARTUP_TEST	n
Memory mapped IO tracing	MMIOTRACE	n

- The platform project is launched on a hardware target and you are logged in.

Step 1 Confirm that the virtual debug file system is mounted.

```
$ ls /sys/kernel/debug/
bdi          kprobes      memblock     sched_features  usb
hid          ltt           powerpc      tracing
```

Step 2 Mount it if it is not.

```
$ mount -t debugfs nodev /sys/kernel/debug
```

Step 3 Observe the current **perf** functions using the CPU.

```
$ perf top
```

Step 4 Press **q** to stop and return to the prompt once you have seen enough:

Step 5 Choose a symbol to examine in a user library.

In the previous step, a list of symbols displayed in the terminal. In this step, you will examine one of them.

```
$ perf probe -x /lib/libc-2.15.so strstr
Added new event:
probe_libc:strstr (on 0x8dcd4)
```

You can now use it in all **perf** tools, such as:

```
$ perf record -e probe_libc:strstr -aR sleep 1
```

Step 6 Obtain some data.

Run the following command in the background for an extended period of time to obtain data:

```
$ perf record -e probe_libc:strstr -aR sleep 60 &
```

Step 7 Enter some random commands to generate data.

```
$ top  
tar -czf this.tar /etc/*
```

When the background task is complete, a message displays in the console, for example:

```
[ perf record: Woken up 1 times to write data ]  
[ perf record: Captured and wrote 0.111 MB perf.data (~4844 samples) ]  
  
[1]+  Done                  perf record -e probe_libc:strstr -aR sleep 60
```

Step 8 Review your results.

```
$ perf report
```

```
Events: 602 probe_libc:strstr  
70.43%  tar   libc-2.15.so  [.] strstr  
28.74%  perf  libc-2.15.so  [.] strstr  
0.83%   top   libc-2.15.so  [.] strstr
```

Step 9 Press **q** to exit the **perf** application's interactive mode.

Run this command once your tracing operations are complete.

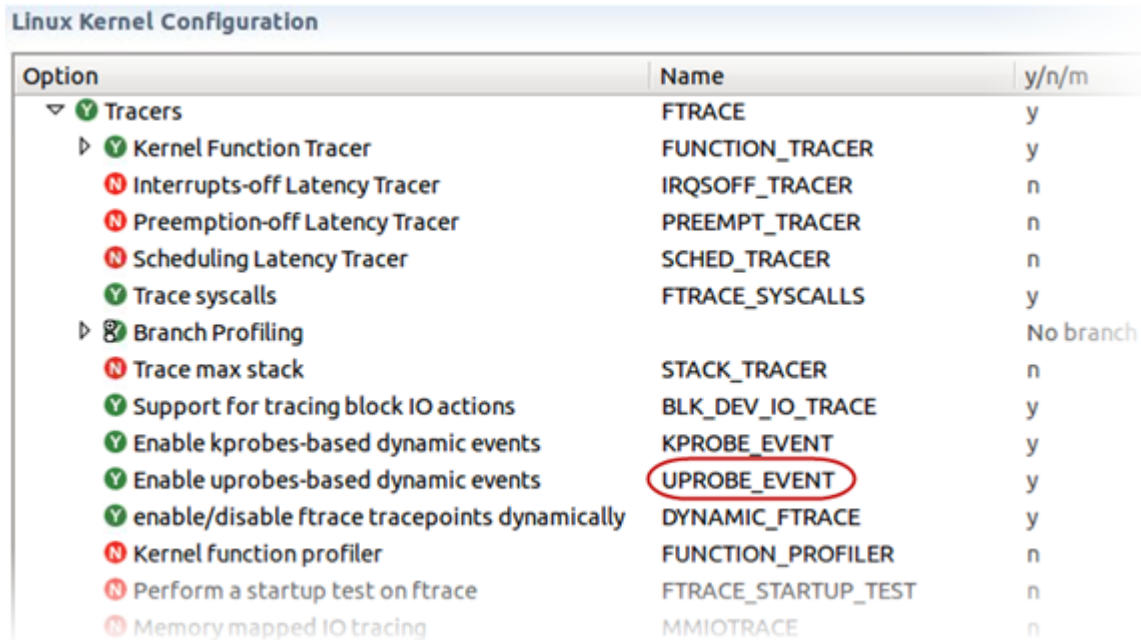
Dynamically Obtain Object Data with uprobes

Use **uprobe** to dynamically obtain data on a specific object, created as a new probe, in the **perf** application.

This procedure requires the following for successful completion:

- Previously configured and built platform project with debugging symbols. See [Adding Debugging Symbols to a Platform Project](#) on page 219.
- The **perf** package included in your platform project build. For additional information on adding packages, see [Options for Adding an Application to a Platform Project Image](#) on page 138.
- The **CONFIG_UPROBE_EVENT=y** kernel configuration parameter enabled in the kernel.

By default, uprobes are included in your kernel configuration, but may not be supported by all BSPs. To see whether your BSP supports this functionality, refer to the **Kernel Configuration > Kernel Hacking > Tracers** section of the utility. For example:



Option	Name	y/n/m
Tracers	FTRACE	y
Kernel Function Tracer	FUNCTION_TRACER	y
Interrupts-off Latency Tracer	IRQSOFF_TRACER	n
Preemption-off Latency Tracer	PREEMPT_TRACER	n
Scheduling Latency Tracer	SCHED_TRACER	n
Trace syscalls	FTRACE_SYSCALLS	y
Branch Profiling		No branch
Trace max stack	STACK_TRACER	n
Support for tracing block IO actions	BLK_DEV_IO_TRACE	y
Enable kprobes-based dynamic events	KPROBE_EVENT	y
Enable uprobes-based dynamic events	UPROBE_EVENT	y
enable/disable ftrace tracepoints dynamically	DYNAMIC_FTRACE	y
Kernel function profiler	FUNCTION_PROFILER	n
Perform a startup test on ftrace	FTRACE_STARTUP_TEST	n
Memory mapped IO tracing	MMIOTRACE	n

- The platform project is launched on a hardware target and you are logged in.

To debug an object file effectively, you can use the following procedure to examine the symbols visible to **perf** probe.

Step 1 Confirm which symbols are available to **perf** in a specific object file.

This step uses the **F** option, for example:

```
$ perf probe -F -x /lib/libc.so.6 | grep mal
```

```
malloc  
malloc@plt  
malloc_info  
memalign@plt
```

Step 2 Create a probe for an interesting function.

```
$ perf probe -x /lib/libc.so.6 malloc
```

```
Added new event:  
probe_libc:malloc (on 0x88914)
```

You can now use it in all **perf** command-line tools, such as:

```
$ perf record -e probe_libc:malloc -aR sleep 1
```

Step 3 Obtain some data.

Run it in the background for an extended period of time:

```
$ perf record -e probe_libc:malloc -agR sleep 60 &
```

Step 4 In this example we have added the **g** option to the command; this enables a call tree in the recorded results. Enter some random commands to generate data. For example:

```
$ top
tar -czf this.tar /etc/*
```

Step 5 Observe the results.

Once the console reports the run as complete, enter **perf report** to observe the results. They will look similar to the following:

```
Events: 3K probe_libc:malloc
86.37%   tar   libc-2.15.so  [.] malloc
 5.33%   perf  libc-2.15.so  [.] malloc
 4.73%   tcf-agent libc-2.15.so  [.] malloc
 2.30%   top   libc-2.15.so  [.] malloc
 1.28%   gzip  libc-2.15.so  [.] malloc
```

Step 6 Select a line and press **ENTER** to observe the call tree if **perf report** is interactive.

Not all functions are displayed. On a typical embedded system, the libraries will be stripped of debug information and only public APIs will be shown.

```
Events: 3K probe_libc:malloc
- 86.49%   tar   libc-2.15.so  [.] malloc
  - malloc
    - 57.67% vasprintf
      - 99.36% 0x10005a
        - 0x100792
          + 89.68% 0x10045e
            + 10.32% 0x100793
              + 0.64% 0x100049
                + 12.75% 0x100054
                  + 10.17% 0x100053
                    + 6.65% 0xfefdff
                      + 3.14% 0xff4bbd
                        + 2.22% tsearch
                          + 2.22% __nss_lookup_function
                            + 2.03% __nss_database_lookup
                              + 0.55% 0xfec07e
                                + 4.45%   perf  libc-2.15.so  [.] malloc
                                + 3.94%   tcf-agent libc-2.15.so  [.] malloc
                                + 3.84%   top   libc-2.15.so  [.] malloc
                                + 1.28%   gzip  libc-2.15.so  [.] malloc
```

Figure 6: perf report Call Tree

Step 7 Press **q** to exit the interactive mode of **perf**.

Step 8 List the probes you have created.

```
$ perf probe -l |more
```

```
probe_libc:malloc   (on 0x00088914)
probe_libc:strstr   (on 0x0008dcd4)
```

Step 9 Remove the probes with the **d** option.

```
$ perf probe -d probe_libc:malloc
```

```
Removed event: probe_libc:malloc
```

Debugging Individual Packages

Use these debug options to debug your packages.

See the *Wind River Linux Getting Started Guide: Debugging an Executable*.

Table 7 Synopsis of debugging individual packages

Debugging method or option	Description
Default Package Options	Packages may have configuration or compile-time options that are not used in the default build of the package. You may have specific needs that require that the default package be built with different options. Refer to the particular packages for customizable options.
Using gcore	When using gdb with Wind River Linux, note that gdb has an internal gcore command that provides functionality that in other systems is provided by a separate gcore executable.

Debugging Packages on the Target Using gdb

This topic explains how to add the **gdb** package to your build.

In order to debug programs on the target, you will need to add the **gdb** package to your build. This can be done at configuration time with the **--with-package=gdb configure** option (see [Configuring a New Project to Add Application Packages](#) on page 142) or after the project is built with the **make -C build gdb.addpkg** command (see [Adding New Application Packages to an Existing Project](#) on page 139).

You will also want the debugging symbol files on your target using one of the methods described in [Adding Debugging Symbols to a Platform Project](#) on page 219.

The following is an example of on-target debugging commands on a glibc-small system. The debugging target is **/bin/busybox**, and specifically the **ls** command implementation.

Step 1 Start **gdb**.

```
# gdb
```

Step 2 Set the debug directory.

```
(gdb) set debug-file-directory /bin/.debug
```

Step 3 Select the **busybox** binary.

```
(gdb) file /bin/busybox
```

Step 4 Set `ls` as the `busybox` command to debug.

```
(gdb) set args ls /
```



NOTE: To debug other functionality implemented by `busybox`, change the arguments of the `set args option` command.

Step 5 Set a breakpoint.

```
(gdb) break main
```

Step 6 Run under the debugger until you reach the breakpoint.

```
(gdb) run
```



NOTE: You can then step over the implementation details of the `ls` command within the `busybox` binary.

Debugging Packages on the Target Using gdbserver

Use this procedure to debug a package by running `gdbserver` on the target and `gdb` on the development host.

You can debug target binaries remotely by running `gdbserver` on the target and `gdb` on the development host.

The following example illustrates how to debug the `ls` command remotely on a `qemux86-64` target with a `glibc_small` root file system.

In order to debug programs on the target using `gdbserver`, you will need to add the `gdb` package to your build. This can be done at configuration time with the `--with-package=gdbserver` configure option (see [Configuring a New Project to Add Application Packages](#) on page 142) or after the project is built with the `make -C build gdbserver.addpkg` command (see [Adding New Application Packages to an Existing Project](#) on page 139).

This procedure assumes that you are already connected to the target.

Step 1 Launch the `gdbserver`.

```
# gdbserver :23 /bin/ls
```

Step 2 Perform a debugging session.

a) Begin the debug session on the development host.

```
$ cd projectDir
$ ./scripts/gdb
(gdb) file bin/busybox
(gdb) target remote localhost:4441
(gdb) break main
(gdb) continue
```

You can then proceed with the debugging session as if it were being performed locally.

b) Change the `gdbserver` command arguments to debug other BusyBox programs.

For example:

```
# gdbserver :23 /usr/bin/less /etc/hosts
```

The command initiates debugging of the **less** command implementation.

- c) Use the **make** command to change the default TCP ports.

In this example, TCP ports 23 and 4441 are the default values used by QEMU.

To customize them, use the **make config-target** command, then modify option **39 TARGET0_QEMU_TELNET_RPORT**.

- d) To debug remotely on a hardware target, connect the host **gdb** session directly to the target.

At the (**gdb**) prompt, type the following command:

```
(gdb) target remote target-ip-address:23
```

Note that in this case, the selection of port number **23** is entirely arbitrary. Any value will do.

22

Analysis Tools Support

[About Analysis Tools Support](#) 231

[Using Dynamic Probes with ftrace](#) 231

[Analysis Tools Support Examples](#) 237

About Analysis Tools Support

Use analysis tools with Workbench as documented in online Analysis Tools and Workbench documentation.



NOTE: Analysis tools are primarily used with Workbench as documented in online *Analysis Tools and Workbench* documentation.

Like other Wind River Linux configuration commands, you can perform the following through Workbench or the command line. Note that if you create projects through the command line, you then have to import them into Workbench for them to become visible.

Backtracing, which is used by the analysis tools, is performed differently by MIPS boards than by non-MIPS boards, so the following presents two examples of configuring builds for analysis tools.

If you are not interested in memory allocations invoked in libraries called by your application, then you can use the production stripped versions of the library object files and simply build your application in Workbench with the Debug build specification. Workbench Memory Analysis will resolve addresses to functions, files and line numbers for addresses in your main application object file but report the addresses in your samples that reside in stripped library objects as “unknown” functions.

Using Dynamic Probes with ftrace

The dynamic **kprobes** feature is an extension of the Linux kernel **ftrace** function tracer.

Currently, x86 is the only platform supported, and the format is instruction-dependent. For more information about supported instructions, please refer to the document

arch/x86/lib/x86-opcode-map.txt

Unlike the function tracer, the **kprobes** tracer can probe instructions inside of kernel functions. It allows you to check which instruction has been executed. And unlike the Tracepoint-based events tracer described in **tracepoints.txt** the **kprobes** tracer can add new probe points on the fly.

One of the design goals of **kprobes** is to allow their insertion and deletion from the command-line without the need for any specialized user tools. So the manipulation of **kprobes** is done via the **proc** and **sys** virtual file systems. The syntax is complex and probably best implemented in a script for processes that are more complex than this example.

The kernel tracing infrastructure is documented in the **../Documentation/trace/** directory found at <http://www.kernel.org/doc/Documentation/trace/> and in your platform project **projectDir/build/BSP_name-wrs-linux/linux-windriver-version/linux/Documentation/trace** directory. The **ftrace.txt** file describes the basic kernel tracing facility. The dynamic **kprobes** extension is described in the **kprobetrace.txt** file.

Configuration

There are no user packages required to use dynamic **kprobes**. The required features are already enabled in the kernel on supported Wind River Linux BSPs in the **Kernel Hacking > Tracers** section of the kernel configuration.

There are two kernel options:

- CONFIG_KPROBE_EVENT=y
- CONFIG_DYNAMIC_FTRACE=y

Linux Kernel Configuration

Option	Name	y/n/m
Tracers	FTRACE	y
Kernel Function Tracer	FUNCTION_TRACER	y
Interrupts-off Latency Tracer	IRQSOFF_TRACER	n
Preemption-off Latency Tracer	PREEMPT_TRACER	n
Scheduling Latency Tracer	SCHED_TRACER	n
Trace syscalls	FTRACE_SYSCALLS	y
Branch Profiling		No branch
Trace max stack	STACK_TRACER	n
Support for tracing block IO actions	BLK_DEV_IO_TRACE	y
Enable kprobes-based dynamic events	KPROBE_EVENT	y
Enable uprobes-based dynamic events	UPROBE_EVENT	y
enable/disable ftrace tracepoints dynamically	DYNAMIC_FTRACE	y
Kernel function profiler	FUNCTION_PROFILER	n
Perform a startup test on ftrace	FTRACE_STARTUP_TEST	n
Memory mapped IO tracing	MMIOTRACE	n

Figure 7: kprobe Configuration

kprobe Syntax Parameters Definition

The following table provides a list of the **kprobe** syntax parameters and their definitions.

Table 8 kprobe Syntax Parameters

Parameter	Definition
p[:[GRP/]EVENT] SYMBOL[+offs] MEMADDR [FETCHARGS]	Set a probe
r[:[GRP/]EVENT] SYMBOL[+0] [FETCHARGS]	Set a return probe
-:[GRP/]EVENT	Clear a probe
GRP	Group name. If omitted, "kprobes" is used as the default.
EVENT	Event name. If omitted, the event name is generated based on SYMBOL[+offs] or MEMADDR
SYMBOL[+offs]	Symbol+offset where the probe is inserted.
MEMADDR	Address where the probe is inserted.
FETCHARGS	Arguments. Each probe can have up to 128 arguments.
%REG	Fetch register REG .
@ADDR	Fetch memory at ADDR (in kernel text segment)
@SYM[+ -offs]	Fetch memory at SYM + - offs (SYM should be a data symbol)
\$stackN	Fetch <i>N</i> th entry of stack (<i>N</i> >= 0)
\$stack	Fetch stack address
\$retval	Fetch return value ³
+ -offs(FETCHARG)	Fetch memory at FETCHARG + - offs address ⁴
NAME=FETCHARG	Set NAME as the argument name of FETCHARG

Resources

Documentation/trace/kprobetrace.txt in your installation.

Documentation/trace/ftrace.txt in your installation.

<http://lwn.net/Articles/343766/>

Preparing to use a kprobe

Complete these steps to prepare to debug with a kprobe.

³ Only for return probe

⁴ This is useful for fetching a field of data structures

Step 1 Check that your BSP supports kprobes.

For information on supported boards, see [Bootloaders and Board README Files](#) on page 18

Step 2 Create a platform project based on the BSP.

Step 3 Verify the correct kernel options are enabled.

Open the Wind River Workbench Kernel Configuration editor and review your settings.

Step 4 Build your project and deploy to your target.

Step 5 Mount the **debugfs** file system if it is not already available.

```
# mount -t debugfs nodev /sys/kernel/debug
```

Ftrace uses the **debugfs** file system to hold the control files as well as the files to display output.

Step 6 Enable **/proc/sys/kernel/ftrace_enabled** if it is not already.

```
# echo 1 > /proc/sys/kernel/ftrace_enabled  
echo > /sys/kernel/debug/tracing/trace  
1
```

Setting up a kprobe

The steps in this procedure show you how to set up a **kprobe** for kernel debugging and confirm that it is working.

Step 1 Determine the correct address(es) to set probe points at.

You can find the related value of a function in your project's kernel in **System.map**; a link to this file is found at:

projectDir/export/BSP_name-System.map-WRversion_standard

For example, to find the address of `do_fork`, you can enter the following command:

```
# export$ grep "do_fork" gemux86-System.map-WR5.0.1.0_standard  
c102e680 T do_fork  
c167b905 t do_fork_idle
```

The example above would yield an address similar to **0xc102bac0**.

Step 2 Set probe points.

Echo the points to **/sys/kernel/debug/tracing/kprobe_events**, replacing the numeric value **0xc102bac0** with the correct value for your project.

```
# echo 'p:doforkprobe 0xc102bac0 clone_flags=%ax stack_start=%dx regs=%cx  
parent_tidptr=+4($stack) child_tidptr=+8($stack)' \  
>> /sys/kernel/debug/tracing/kprobe_events
```



NOTE: Ensure that the entire quoted section of the command is on the same line when you enter it at the terminal prompt.

Additional examples.

Check the four parameters of `do_sys_open`

```
# echo 'p:myprobe do_sys_open dfd=%ax filename=%dx flags=%cx mode=+4($stack)' > /sys/
kernel/debug/tracing/kprobe_events
```

Check the return value of `do_sys_open`, `__dentry_open`, and `do_fork`.

```
# echo 'r:myretprobe do_sys_open $retval' >> /sys/kernel/debug/tracing/kprobe_events
```

```
# 'r:fork_retprobe do_fork $retval' >> /sys/kernel/debug/tracing/kprobe_events
```

```
# 'r:dentry_openprobe __dentry_open $retval' >> /sys/kernel/debug/tracing/kprobe_events
```

This sets a **kprobe** on the top of the `do_fork()` function with recording 1st to 5th arguments as **doforkprobe** event. Note also that whichever register/stack entry is assigned to each function argument depends on arch-specific ABI.

Step 3 Confirm that the **kprobe** is working.

You can **cat** the **kprobe** and review output to determine if it is working correctly.

a) Generate output

For example:

```
# cat /sys/kernel/debug/tracing/kprobe_events
p:kprobes/doforkprobe 0xc102e680 clone_flags=%ax stack_start=%dx regs=%cx
parent_tidptr=+4($stack) child_tidptr=+8($stack)
```

b) Review the output.

A successfully inserted probe will also appear in the tracing directory. If there is one **kprobe** in `kprobe_events`, **kprobe** will be in the directory of `/sys/kernel/debug/tracing/events`. For example:

```
root@d610:/sys/kernel/debug/tracing> ls -l events/kprobes/
total 0
drwxr-xr-x 2 root root 0 May 26 13:44 doforkprobe
-rw-r--r-- 1 root root 0 May 26 13:44 enable
-rw-r--r-- 1 root root 0 May 26 13:44 filter
root@d610:/sys/kernel/debug/tracing> ls -l events/kprobes/doforkprobe/
total 0
-rw-r--r-- 1 root root 0 May 26 13:45 enable
-rw-r--r-- 1 root root 0 May 26 13:44 filter
-r--r--r-- 1 root root 0 May 26 13:44 format
-r--r--r-- 1 root root 0 May 26 13:44 id
```

Enabling and Using a kprobe

The steps in this procedure show you how to enable and use a **kprobe** to trace kernel debugging data.

Step 1 Enable the **kprobe**.

a) Set the value of `/sys/kernel/debug/tracing/events/kprobes/doforkprobe/enable` to 1.

```
# echo 1 > /sys/kernel/debug/tracing/events/kprobes/doforkprobe/enable
```

There will now be some capture counts in `<kprobe_profile>.`:

b) Review the capture counts..

```
# root@d610:/sys/kernel/debug/tracing> cat kprobe_profile
```

You should see results similar to the following:

```
doforkprobe          26          0
```

The first column is the event; the second is the number of probe hits; and the third is the number of probe miss-hits.



NOTE: Substitute the value appropriate to your environment for @d610.

Step 2 Enable tracing.

```
# echo 1 > /sys/kernel/debug/tracing/tracing_enabled
```

Step 3 View the trace.

```
# cat /sys/kernel/debug/tracing/trace#
```

```
# tracer: nop
#
# entries-in-buffer/entries-written: 2/2   #P:1
#
#              -----=> irqs-off
#              /-----=> need-resched
#              | /-----=> hardirq/softirq
#              || /-----=> preempt-depth
#              ||| /-----=> delay
#
#           TASK-PID  CPU#  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
#           |  |      |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
#           sh-518   [000]  d..2 1679.804579: doforkprobe: (do_fork+0x0/0x310)
clone_flags=1200011 stack_start=bfc6c7d0 regs=d6689fb4 parent_tidptr=0 child_tidptr=0
#           sh-518   [000]  d..2 2068.342909: doforkprobe: (do_fork+0x0/0x310)
clone_flags=1200011 stack_start=bfc6c7d0 regs=d6689fb4 parent_tidptr=0 child_tidptr=0
```

Disabling a kprobe

The steps in this procedure show you how to disable a kprobe after using it to debug a kernel.

- Remove your probe from `kprobe_events`..

```
# echo 0 > /sys/kernel/debug/tracing/events/kprobes/doforkprobe/enable
```

```
# echo 0 > /sys/kernel/debug/tracing/tracing_enabled
```

```
# echo '-:doforkprobe' >> /sys/kernel/debug/tracing/kprobe_events
```

`doforkprobe` will be removed from `kprobe_events`, `kprobe_profile`, and `events/kprobes`. If there is no kprobe in `kprobe_events`, `events/kprobes` will be deleted too.

Analysis Tools Support Examples

Use the examples in this section to configure a platform project to add analysis tools support from the command line.

Additional Reading

Refer to the analysis tools documentation for specifics on using the Wind River Analysis Tools.

Adding Analysis Tools Support for MIPS Targets

These instructions show you how to add analysis tool support to MIPS targets.

Step 1 Configure the platform project for MIPS targets.

The following **configure** example command adds analysis tools support using the **--with-template=feature/analysis** option:

```
$ ../configure --enable-board=qemumips \  
--enable-rootfs=glibc_std \  
--enable-kernel=standard \  
--with-template=feature/analysis
```



NOTE: MIPS boards use a different method for backtracing. A production build along with a **make fs-debug** (which places the symbolic information on the host) is fine for use with **oprofile**, but if you are using **mpatrol**, you should specify the **--enable-build=profiling** argument to your configure command. This is necessary because **mpatrol** requires the presence of additional symbols to analyze target memory on the target. Note that **oprofile** can fetch these symbols from the host.

Step 2 Build the MIPS target file system.

```
$ make
```

Adding Analysis Tools Support for Non-MIPS Targets

These instructions show you how to add analysis tool support to non-MIPS targets.

Step 1 Configure the target.

Use the following **configure** example command to add analysis tools support to, for example, a **qemux86-64** target:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-rootfs=glibc_std \  
--enable-kernel=standard \  
--with-template=feature/analysis \  
--enable-build=profiling
```



NOTE: The `--enable-build=profiling` option enables frame pointers for the backtrace code. (The `--enable-build=debug` option also enables frame pointers which enables backtrace functionality.)

Step 2 Build the target file system.

```
$ make
```

PART VI

Using Simulated Target Platforms for Development

QEMU Targets.....	241
Wind River Simics Targets.....	251

23

QEMU Targets

[QEMU Targets](#) 241

QEMU Targets

QEMU is a processor simulator for supported boards. (Refer to your Release Notes for a list of supported boards.) Using QEMU for simulated deployment, no actual target boards are required, and there are no networking preliminaries.

QEMU and Workbench are compatible both in User Mode and Kernel Mode. QEMU deployment, for the supported boards, offers a suitable environment for application development and architectural level validation. User-space and kernel binaries are compatible with the real hardware.

When started, QEMU runs in a pseudo-root environment and starts the NFS server with alternate RPC ports. The simulated target is given a hard-coded IP address of 10.0.2.15, and localhost is visible from the simulated target as 10.0.2.2.

See [QEMU Target Deployment Options](#) on page 241.

QEMU Prerequisites

To deploy a QEMU simulation, you must have built a platform project for one of the QEMU-enabled BSPs, which are named after their architecture. For example, `qemux86-64` represents a 64-bit x86 board. See the *Wind River Linux Release Notes* for a list of QEMU-enabled BSPs.

QEMU Target Deployment Options

Use the examples in this section to configure, monitor, and specify launch options for QEMU from the command line.

The *Wind River Linux Getting Started Guide: Deploying a Platform Project Image* provides an example of how to deploy a QEMU target for user mode debugging. You can also use QEMU to perform kernel mode debugging (KGDB) of supported Wind River Linux targets as described in this section.

After you have built a platform project for one of the QEMU-supported boards and then built the file system (**make**), you can start an instance of QEMU for that target.

Note that after a building a platform project using the **make** command, the pre-built kernel is automatically copied to the **export** subdirectory of the project directory. The QEMU simulator loads and executes the kernel found within the **export** subdirectory, and NFS-mounts the **export/dist** subdirectory as its root file system.

Setting QEMU Configuration Options

This procedure shows you how to access and change QEMU configuration options

Step 1 Start the QEMU configuration tool.

Issue the following command to enter an interactive QEMU configuration tool.

```
$ make config-target

===QEMU and or User NFS Configuration===
1: TARGET_QEMU_BOOT_TYPE=usernfs
2: NFS_EXPORT_DIR=/home/user/WindRiver/workspace/qemux86-64_prj
3: NFS_MOUNTPROG=21111
4: NFS_NFSPROG=11111
5: NFS_PORT=3049
6: TARGET_QEMU_BIN=qemu
7: TARGET_QEMU_AUTO_IP=yes
8: TARGET_QEMU_USE_STDIO=yes
9: TARGET_QEMU_BOOT_CONSOLE=ttyS0
10: TARGET_QEMU_GRAPHICS=no
11: TARGET_QEMU_KEYBOARD=en-us
12: TARGET_QEMU_PROXY_PORT=4442
13: TARGET_QEMU_PROXY_LISTEN_PORT=4446
14: TARGET_QEMU_DEBUG_PORT=1234
15: TARGET_QEMU_AGENT_RPORT=udp:4444::17185
16: TARGET_QEMU_KGDB_RPORT=udp:4445::6443
17: TARGET_QEMU_TELNET_RPORT=tcp:4441::23
18: TARGET_QEMU_SSH_RPORT=tcp:4440::22
19: TARGET_QEMU_MEMSCOPE_RPORT=tcp:5698::5698
20: TARGET_QEMU_PROFILESCOPE_RPORT=tcp:5678::5678
21: TARGET_QEMU_KERNEL=bzImage
22: TARGET_QEMU_INITRD=
23: TARGET_QEMU_HARD_DISK=
24: TARGET_QEMU_CDROM=
25: TARGET_QEMU_BOOT_DEVICE=
26: TARGET_QEMU_KERNEL_OPTS=
27: TARGET_QEMU_OPTS=
Enter number to change (q quit)(s save):
```

Step 2 Make configuration changes.

Enter the corresponding number and press **ENTER** to change the value of an option.

For example, enter **10** to turn graphics on or off.



NOTE: You need to build graphics support into your kernel, change the bootline, or use the `TOPTS="-gc"` option, which does both for you.

Step 3 Save your changes.

Type **S** and press **ENTER**.

Accessing the QEMU Monitor

Learn how to access the QEMU Monitor to manipulate QEMU from within a running simulation. The QEMU Monitor provides QEMU-specific commands from within the simulation.

Step 1 Start QEMU.

```
$ make start-target
```

Step 2 Enter the monitor.

Press **CTRL+A C** to access the QEMU Monitor.

The Monitor appears:

```
(qemu) help
help|? [cmd] -- show the help
commit device|all -- commit changes to the disk images (if -snapshot is used) or
backing files
info subcommand -- show various information about the system state
q|quit -- quit the emulator
.
.
(qemu) CTRL+A,C
root@localhost:/root>
```

Step 3 Quit the Monitor.

When you are done using the Monitor, type one of the following to exit QEMU.

- `q`
- `quit`

Viewing QEMU Command Line Options

Learn how to access QEMU command line options

- Display QEMU command line options.
Passing the **-h** option to the **TOPTS** parameter displays the options available to set.

```
$ make start-target TOPTS="-h"
```

Output similar to the following is displayed:

```
Usage ./scripts/config-target.pl [Options] <command>
Options:
-c      Use text console
-gc     Use graphics console
-p      Use telnet proxy as console
-i #    Increment the remote port offsets by #
        typically used when starting more than
        one target
-d      Extra script debug output
-w      Wait until debugger attaches to QEMU
-x      Use an external console defined by
        TARGET_VIRT_EXTERNAL_CONSOLE
        and go into the background
```

```
-o      Output the target start command which you
        could use to start a debugger with
-m #    Number of megabytes of RAM to use on the target
-su     Use "su -c" instead of "sudo" for root access
-t      Use tuntap
-cd <iso_file>      Boot from CD (QEMU Only)
-disk <disk_image>  Boot kernel with disk image
-cow <cow_file>     COW file for (UML Only)
-no-kqemu           Do not use the kqemu accelerator

Commands:
start      Start target, NFS server and proxy (if needed)
stop      Stop the target and NFS server...
nfs-start  Start the NFS server
nfs-stop  Stop the NFS server
net-start  Start the network server (TUN/TAP)
net-stop  Stop the network server (TUN/TAP)
kqemu-start Load the KQEMU kernel module
kqemu-stop unload the KQEMU kernel module
allstop   Stop target, NFS server and proxy
config    Display or change the default configuration
```

QEMU Targets

Starting a QEMU Session

Use this information as a guideline for starting a new QEMU session.

After you build a QEMU-enabled project, you can run it in a QEMU session. You can perform typical operations as if you are running on an actual hardware target. You can communicate to the host using IP address 10.0.2.2. The IP address of the simulation is 10.0.2.15.

Step 1 Run the following command:

```
$ make start-target
```

The emulated system boots.

Step 2 Log in with the user name **root**, password **root**.

The session is now running.

Resolving QEMU Start Errors

Use this information as a guideline for resolving errors that occur while starting QEMU.

If there is an error because you have a former session still running that you no longer want, you will have to close those sessions.

- Enter the following command:

```
$ pkill rpc
```

The previous session is ended.

Running Multiple QEMU Sessions

Use this information as a guideline for running multiple QEMU sessions simultaneously.

To run an additional QEMU session, you could also use `-i` option to automatically increment port numbers by the specified amount.

- Issue the following command to start a new session with the port number incremented by 2:

```
$ make start-target TOPTS="-i 2"
```

Starting a QEMU Session From a .iso File

Use this information as a guideline for starting a QEMU session from an ISO file.

In some circumstances you may need to boot a CDROM image in QEMU.

- To boot a CDROM image (.iso file) in QEMU, enter:

```
$ make start-target TOPTS="-cd projectDir/export/image.iso"
```

See [About Configuring and Building Bootable Targets](#) on page 265.

Starting a QEMU Session From a Disk Image

Use this information as a guideline for starting a QEMU session from a disk image.

Under some circumstances you may need to start a QEMU session from a disk image.

- To boot a USB or hard disk image in QEMU, enter:

```
$ make start-target TOPTS="-disk Hard_Disk_Image"
```

See [About Configuring and Building Bootable Targets](#) on page 265 for more on creating and booting .iso images.

Starting a QEMU Session With a Graphics Console

Use this information as a guideline for starting a QEMU session with a graphics console.

Your development efforts may involve a graphics console such as X Windows.

- To boot with a graphics console in your simulation, enter:

```
$ make start-target TOPTS="-gc"
```

Passing Boot Options to QEMU

Use this information as a guideline for passing boot options to QEMU.

You may want to pass multiple boot options to a QEMU session. You can do this using the environment variable `TARGET_QEMU_KERNEL_OPTS`.

- Enter a command similar to the following example:

```
$ make TARGET_QEMU_KERNEL_OPTS="init=/bin/bash" make start-target
```

Using Multiple QEMU Options

Use this information as a guideline for applying multiple QEMU options.

You may need to combine multiple QEMU options on the command line.

- Increment the port count and boot a **.iso** image.

Enter the following example command:

```
$ make start-target TOPTS="-i 2 -cd projectDir/export/image.iso"
```

Port Mappings for Accessing the QEMU Target Simulation

Use this information as a guideline for using port mapping to access a QEMU target simulation.

By default QEMU is launched with NAT (Network Address Translation) by default. QEMU with NAT does not require root privileges on the host. The tap option avoids network routing issues associated with NAT but requires root privileges. The usual host ports are mapped to new port numbers so that you can access the features through the new port numbers. For example, KGDB is usually accessed at port 6443, but you use port 4445 when you connect to the simulation.

Telnet port 23 has been mapped to port 4441, and ssh port 22 has been mapped to port 4440. You can access the running simulation through those ports with the appropriate tools.

- Log in to the running simulation.

Enter the following command to use **ssh** to log in to the running simulation from another terminal window on the same host:

```
$ ssh -p 4440 root@localhost
```

Ending a QEMU Session

Use this information as a guideline for ending a QEMU session.

It is good practice to end your QEMU session when you are finished with it.

- Use one of the following options to end your QEMU session:.

- Enter the following in the terminal window

```
CTRL+A,X
```

- Run the **halt** command.

```
# halt
```

This will cleanly perform a system shutdown on the emulated target.

TUN/TAP Networking with QEMU

Use the information in this section to enable virtual networking in QEMU.

TUN and TAP are virtual network kernel drivers used to implement network devices that are supported entirely in software, making them ideal for use with a QEMU deployment.

TAP, for network “tap”, simulates an Ethernet device and works with layer 2 packets such as Ethernet frames.

TUN, short for network “tunnel”, simulates a network layer device. It works with layer 3 packets, such as IP packets. Once enabled, TAP creates a network bridge while TUN provides the routing.

You can use TUN/TAP networking to configure a network on your host that connects to the QEMU target simulation. If you wish to connect two or more QEMU simulations for testing and debugging, TUN/TAP lets you specify the networking parameters for each simulation.



NOTE: Configuring TUN/TAP networking on the host requires root privileges—you can start the emulation as the root user, or start it as another user and you will be prompted for the root password.

TUN/TAP Settings from Workbench

If you used Workbench to create the QEMU target connection, TUN/TAP is enabled by default. It is possible to make changes to the default settings when you create a new target connection or from the **Target Connection Properties** dialog.

The default settings include:

TARGET_TAP_DEV

The device number of the software network tap. The default setting is **auto**, but you may specify a number for the tap. For example, **tap0**, **tap1**, and so on.

TARGET_TAP_UID

The user ID name of the tap device. The default setting is **auto**.

TARGET_TAP_IP

The IP address of the tap interface. The default setting is **auto**.

TARGET_TAP_ROOTACCESS

The root access command for starting or making changes to TAP settings. The default setting is **sudo**, but **su -c** is also acceptable.

TARGET_TAP_HOST_DEV

The host Ethernet interface. The default is **eth0**.



NOTE: You must configure the TUN/TAP interface once for each system boot.

Configure TUN/TAP in the Wind River Workbench New Target Wizard

Use this information as a guideline for configuring TUN/TAP using the Wind River Workbench New Target Wizard.

TUN/TAP can be set up from within Wind River Workbench.

Step 1 Start Wind River Workbench.

Step 2 Click the **New Connection** button in the Remote Systems window to launch the New Connection Wizard.

Step 3 Select the connection type.

Since we are accessing TUN/TAP settings for a QEMU deployment, choose **Wind River QEMU Connection**, then click **Next**.

Step 4 Update TUN/TAP settings.

In the QEMU Simulator Configuration section of the New Connection dialog, , make changes as necessary to the default TUN/TAP settings.

Step 5 Complete the wizard process in accordance with your target connection requirements.

Configure TUN/TAP settings for an existing target connection

Use this information as a guideline for configuring TUN/TAP for use with an existing target connection.

In some circumstances, you may need to configure TUN/TAP for use with an existing target connection.

Step 1 Start Wind River Workbench.

Step 2 Access Remote Systems properties.

In the Remote Systems window, right-click on the QEMU target connection you want to make changes on, then click **Properties**.

Step 3 Update TUN/TAP settings.

In the Target Connection dialog, **QEMU Simulator Configuration** tab, make changes as necessary to the default TUN/TAP settings.

Step 4 Click **OK** to save the settings.

Configure TUN/TAP from the Command Line

Use this information as a guideline for configuring TUN/TAP from the command line.

In some circumstances, you may need to configure TUN/TAP from the command line.

Step 1 Enter the following at the command line:

```
$ make net-start TOPTS="-t"
```



NOTE: This command must be run as **root**. If you are not logged in as **root**, **sudo** will automatically run and prompt you for the root password.

Step 2 When your simulation is running, view the routing information on the simulation:

```
root@localhost:/root> route
```

```
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
192.168.200.0 * 255.255.255.0 U 0 0 0 eth0
default 192.168.200.1 0.0.0.0 UG 0 0 0 eth0
```



```
root@localhost:/root>
```

Step 3 View routing information on the host:

```
host_# route
```

```
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
192.168.200.15 * 255.255.255.255 UH 0 0 0 tap0
192.168.200.0 * 255.255.255.0 U 0 0 0 tap0
190.0.2.123 * 255.255.255.0 U 0 0 0 eth0
default gateway-02 0.0.0.0 UG 0 0 0 eth0
host_#
```

Notice that **192.168.200.1** is assigned to the host and **192.168.200.15** is assigned to the target.

Wind River Simics Targets

[Wind River Simics Targets](#) 251

Wind River Simics Targets

Use Wind River Simics to simulate real world hardware target platforms.

Wind River Simics is a fast, functionally-accurate, full system simulator. Simics creates a high-performance virtual environment in which any electronic system – from a single board to complex, heterogeneous, multi-board, multi-processor, multicore systems – can be defined, developed and deployed.

Simics enables companies to adopt new approaches to the product development life cycle resulting in dramatic reduction in project risks, time to market, and development costs while also improving product quality and engineering efficiency. Simics allows engineering, integration and test teams to use approaches and techniques that are simply not possible on physical hardware.

To purchase Wind River Simics, contact your Wind River sales representative.

To use Simics as a platform project image target, see [Using Simics from the Command Line](#) on page 251.

Using Simics from the Command Line

Use the basic procedures in this section to launch and/or configure a Simics target from the command-line.

You can enter **make help** at any time from your project directory to view a full listing of Simics options.

Where to Find Additional Information

For detailed information on using Simics, see:

- *Wind River Simics Hindsight Installation Guide*
- *Wind River Simics Hindsight Getting Started Guide*
- *Wind River Simics Hindsight User's Guide*

Meeting Simics Prerequisites

To use Wind River Simics successfully, ensure the prerequisites in this section are met.

Before you can use Simics, you must first install it. See the *Wind River Simics Installation Guide* for detailed instructions.

Step 1 Set up the environment.

Add the following lines of code to your **.bashrc** file or the equivalent configuration file for your shell environment. Adjust the paths and Simics version number to match your installation.

```
export SIMICS_BIN_HOME=customer_path_to_simics_install/simics-x.x/bin
export SIMICS_LICENSE_FILE=customer_path_to_license_file
```

Step 2 Save the file and close the terminal.

The next time you open a terminal window, the path is set up and Simics will be ready for use.

Launching the Simics Basic Target Console

Use this information as a guideline for launching the Simics Basic Target console.

This procedure requires a previously configured and built platform project and a Wind River Simics installation. For additional information, see [Meeting Simics Prerequisites](#) on page 252.

Step 1 Change to your project directory.

```
$ cd projectDir
```

Step 2 Boot the Simics target.

```
$ make start-simics
```

The target starts, and you can perform debugging tasks.

Step 3 Stop the target simulation.

Issue the following command from a second terminal window to stop Simics:

```
$ make stop-simics
```

Launching the Simics Graphics Target Console

Use this information as a guideline for launching the Simics Graphics Target console.

To have access to the full set of Simics capabilities, you need to run the Simics emulation in a graphics console.

This procedure requires a previously configured and built platform project and a Wind River Simics installation. For additional information, see [Meeting Simics Prerequisites](#) on page 252.

Step 1 Change to your project directory.

```
$ cd projectDir
```

Step 2 Boot the target in a Simics graphics console.

Run the following from the platform project directory:

```
$ make start-simics TOPTS=" -g"
```

Step 3 Stop the target simulation.

Issue the following command from a second terminal window to stop Simics:

```
$ make stop-simics
```

Refer to the *Wind River Simics Hindsight User's Guide* for additional information on using the full capabilities of Simics.

Enabling Simics Acceleration for x86 BSPs

Use this information as a guideline for using Simics acceleration for x86 BSPs.

Simulated targets can be accelerated for x86 BSPs with Simics.

This procedure requires a previously configured and built platform project based on an x86 BSP and a Wind River Simics installation. For additional information, see [Meeting Simics Prerequisites](#) on page 252.

Step 1 Change to your project directory.

```
$ cd projectDir
```

Step 2 Boot the Simics target.

Run the following command as root to use the Simics Accelerator on x86 BSPs:

```
# make start-simics-vmp
```



NOTE: You only need to run this command once on the development host.

Refer to the *Wind River Simics Hindsight User's Guide* for additional information on the VMP acceleration feature.

Step 3 Stop the target simulation.

Issue the following command from a second terminal window to stop Simics:

```
$ make stop-simics
```

Configuring a Simics Target

Use this information as a guideline for configuring a Simics target.

Once your target is ready, you need to configure it for use with Simics.

- Configure Simics.

Issue the following command from the platform project's directory:

```
$ make config-target-simics
```

For additional information on configuring Simics, see the *Wind River Simics Hindsight User's Guide*.

PART VII

Deployment

Managing Target Platforms.....	257
Deploying Flash or Disk Target Platforms.....	265
Deploying initramfs System Images.....	283
Deploying KVM System Images.....	287

25

Managing Target Platforms

[Customizing Password and Group Files](#) 257

[About Idconfig](#) 259

[Connecting to a LAN](#) 260

[Adding an RPM Package to a Running Target](#) 261

[Adding Reference Manual Page Support to a Target](#) 262

[Using Pseudo](#) 263

Customizing Password and Group Files

You can modify the password and group file construction process to produce custom password and group files using several methods as described in this section.

Overview

The default contents of the Wind River Linux file system contain only one login for the root user. In any secure system you will want to create additional logins for the various roles that are implemented in your security policies. Creating these logins can be problematic as the operations require privileged access.

It is important to note that there is a preferred method for making these modifications. The other methods are described in case the preferred method does not suit your needs, but you should take care when using these other methods for reasons discussed below.



CAUTION: This caution applies if you are not using the preferred method ([Adding an Application to a Root File System with fs_final*.sh Scripts](#) on page 141) of modifying these files.

Individual package configurations, file system owners and groups, and other items may be affected if the numeric IDs do not match the password and group files as originally installed. Modifying or removing password or group file entries may cause adverse behavior to occur within the system.

It is important to look at the file system installation logs, as RPM automatically sets the user and group ID to **root** for files that have an owner or group ID that is not in the password or group files. This can introduce a security flaw in specific cases, but will more likely cause an application to not work as intended.

You should also be aware of the fact that some configuration files may have specific user ID or group ID numbers defined in them. Changing these numbers in the password or group files could then cause application behavior to be affected.

The following topics provide different options for customizing system password and group files:

Using an fs_final.sh Script to Edit the Password and Group File

Use this information as a guideline for editing password and group files using an fs_final.sh script.

This method of modifying the password and group files is preferred because it preserves the default password and group settings, along with any additions for individual packages that may have occurred during file system generation. This is important because file owner and group ID numbers are defined as individual files are created in the target file system.

Step 1 Create an **fs_final.sh** script in a layer that contains your desired modifications.

Step 2 Add your desired modifications.

See [Adding an Application to a Root File System with fs_final*.sh Scripts](#) on page 141 for an example of what is required to get the **fs_final*.sh** script working with your platform project.

For example, the following **fs_final.sh** script adds a new **wrs1** user with password **wrs1** to a **glibc_small rootfs**, only if the user does not already exist:

```
# Example of preferred methodology to add to password and group files
grep -q "^wrs1:" etc/passwd if [ $? -ne 0 ] ; then
#
# If the user does not already exist:
# Add the user
#
echo "Adding wrs1 default user..."
echo 'wrs1:$1$JUYoDU8h$tFUwFwPWO4tfE24KJBEOB/:500:500:Default non-root user
account:/home/wrs1:/bin/sh'>> etc/passwd
#
# Add a group for the user
#
echo 'wrs1:x:500:'>> etc/group
#
# Create a home directory for the user
#
cp -r etc/skel home/wrs1
chown -R 500:500 home/wrs1
fi
```

Once set up, the script will run each time the platform project is built using the **make** command.

`projectDir/layers/local/recipes-local/myusermods/fs_final.sh`

Using an `fs_final_sh` Script to Overwrite the Password and Group File

Use this information to explicitly generate password and group files from an `fs_final.sh` script.

You can use an `fs_final.sh` script that explicitly creates password and group files in the script. This method overwrites the default password and group files after the filesystem has been constructed.

Step 1 Create an `fs_final.sh` script in a layer that contains your desired modifications.

Step 2 Add your desired modifications.

See [Adding an Application to a Root File System with `fs_final*.sh` Scripts](#) on page 141 for an example of what is required to get the `fs_final*.sh` script working with your platform project.

For example, you could use these commands in an `fs_final.sh` script to create a new `passwd` file:

```
...
cat <<EOF > etc/passwd
custom passwd file contents
EOF
...
```

About `ldconfig`

`ldconfig` is a utility that indexes shared object names to simplify loading on shared object libraries by executables.

It scans standard directories and those found in the `ld.so.conf` configuration file and stores its index in `ld.so.cache`. Although not generally used on embedded systems, there are a couple of circumstances where it may be useful as a work-around on Wind River Linux:

- In situations where an executable binary lacks an `RPATH`.
- When a library such as `libfoo.so` provides a shared object name (soname) of `libbar.so`, the filesystem convention requires a symbolic link from `libbar.so` to `libfoo.so`. However, broken applications, filesystems, or images can fail to create the link.

`ldconfig` is not enabled by default. See [Enabling `ldconfig` Support](#) on page 259 for installation instructions.

Enabling `ldconfig` Support

The `ldconfig` utility may be required for executables to successfully load shared objects under unusual circumstances.

`ldconfig` support is not enabled by default. In *exceptional circumstances*, it may be required for executables to load properly.

Step 1 Run `configure` to create your project.

You will need to include the option `--enable-ldconfig=yes` in the project definition.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
```

```
--enable-rootfs=glibc_small \  
--with-template=feature/debug,feature/analysis \  
--enable-ldconfig=yes
```



NOTE: If you are adding **ldconfig** support to an existing project, you must also specify the **--enable-reconfig** option.

Step 2 Build your project.

```
$ make
```

After rebuilding, **ldconfig** and an empty **ld.so.conf** file will be included in your project. The **USE_LDCONFIG** environment variable is automatically enabled (set to **USE_LDCONFIG=1**). You can disable **ldconfig** at any time by setting it to **0** in your project's **local.conf** file.

Connecting to a LAN

Use this procedure to connect your target platform to a local area network.

To perform the following procedure, you will need

- An IP address on your local network
- A platform built with the **glibc_std** file system
- Optionally, a **resolv.conf** file for name service

Step 1 Assign an IP address on your running target.

Step 2 Configure routing.

Step 3 Optional. Enable name service.

Step 4 Test the configuration of the system's network connection.

a) Check the status of the network connection.

```
bash-3.2# ifconfig
```

```
eth0    Link encap:Ethernet  HWaddr 00:1D:09:B7:DF:A7  
        inet addr:192.168.1.18  Bcast:147.11.152.255  Mask:255.255.255.0  
        inet6 addr: fe80::21d:9ff:feb7:dfa7/64 Scope:Link  
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
        RX packets:738 errors:0 dropped:0 overruns:0 frame:0  
        TX packets:362 errors:0 dropped:0 overruns:0 carrier:0  
        collisions:0 txqueuelen:1000  
        RX bytes:361276 (352.8 KiB)  TX bytes:37878 (36.9 KiB)  
        Interrupt:7
```

The interface should be **UP**.

b) Display the connection's routing information.

```
bash-3.2# route
```

```
Kernel IP routing table  
Destination Gateway Genmask Flags Metric Ref Use Iface  
147.11.152.0 * 255.255.255.0 U 0 0 0 eth0  
default 192.168.1.1 0.0.0.0 UG 0 0 0 eth0
```

Your routing table should include the default gateway and, if you configured name service, you should be able to access hosts by their hostname.

c) Test connectivity to an external network.

```
bash-3.2# ping -c 1 google.com

PING google.com (74.125.67.100) 56(84) bytes of data:
64 bytes from gw-in-f100.google.com (74.125.67.100): icmp_seq=1 ttl=51 time=82.0 ms

--- google.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 84ms
rtt min/avg/max/mdev = 82.007/82.007/82.007/0.000 ms
bash-3.2#
```

You should be able to reach external targets.

Adding an RPM Package to a Running Target

You must install packages with the correct architecture for your target when installing on the running target.

The following examples assumes your target system has:

- A qemux86 architecture
- A network connection and a valid `/etc/resolv.conf` file

Step 1 Get an RPM from a network location:

```
bash-3.2# rpm -ivh dash-0.5.4-1.el5.rf.i386.rpm \
http://dag.wieers.com/rpm/packages/dash/dash-0.5.4-1.el5.rf.i386.rpm
```

The system displays the progress of the package download.

```
--17:24:45--
http://dag.wieers.com/rpm/packages/dash/dash-0.5.4-1.el5.rf.i386.rpm
=> `dash-0.5.4-1.el5.rf.i386.rpm'
Resolving dag.wieers.com... 62.213.193.164
Connecting to dag.wieers.com|62.213.193.164|:80...
connected.
HTTP request sent, awaiting response... 302 Found
Location:
http://rpmforge.sw.be/redhat/el5/en/i386/rpmforge/RPMS/dash-0.5.4-1.el5.rf.i386.rpm
[following]
--17:24:46--
http://rpmforge.sw.be/redhat/el5/en/i386/rpmforge/RPMS/dash-0.5.4-1.el5.rf.i386.rpm
=> `dash-0.5.4-1.el5.rf.i386.rpm'
Resolving rpmforge.sw.be... 130.133.35.16
Connecting to rpmforge.sw.be|130.133.35.16|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 85,914 (84K) [application/x-rpm]

100%[=====>] 85,914
98.07K/s

17:24:47 (97.63 KB/s) - `dash-0.5.4-1.el5.rf.i386.rpm' saved
```

```
[85914/85914]
```



NOTE: If there are problems with the URL to access the RPM, you can also retrieve the RPM with `wget` first, then install it using the following:

```
$ wget http://dag.wieers.com/rpm/packages/dash/dash-0.5.4-1.el5.rf.i386.rpm  
rpm -ivh dash-0.5.4-1.el5.rf.i386.rpm dash-0.5.4-1.el5.rf.i386.rpm
```

Step 2 Use the `rpm` command to install the package on the running target.

```
bash-3.2# rpm -ivh dash-0.5.4-1.el5.rf.i386.rpm
```

The system displays the progress of package installation.

```
dash-0.5.4-1.el5.rf.i386.rpm  dash 0.5.5.1-2_i386.deb.1  
bash-3.2# rpm -ivh dash-0.5.4-1.el5.rf.i386.rpm  
warning: dash-0.5.4-1.el5.rf.i386.rpm: Header V3 DSA  
signature: NOKEY, key ID 6b8d79e6  
Preparing...  
##### [100%]  
 1:dash  
##### [100%]  
bash-3.2#
```

Step 3 Verify that the installation of the dash shell was successful.

```
bash-3.2# dash
```

```
#
```

You get the **dash** shell prompt.

Step 4 Exit the **dash** shell.

```
# exit
```

You are returned to the **bash** shell.

```
bash-3.2#
```

Adding Reference Manual Page Support to a Target

This topic illustrates the installation of the packages for the `man` command onto a running target, then demonstrates the use of the installed `man` command

Reference manual pages can enhance the usability of your platform project image. Use the following procedure to add the `man` command for manual page support.

Step 1 Configure the project.

Add the option `--enable-doc-pages=target` to the project



NOTE: The `--enable-doc-pages` configure option only applies to the `glibc_std` file system.

The option automatically adds the required `man` package to your platform project image.

Step 2 Build and deploy the platform on a target.

```
$ make
$ make start-target
```

Step 3 Test the results.

If you have configured your project correctly, the **man** pages should be available in **/usr/share/man/**. For example, view the section 1 pages on the installed target:

a) List the man pages

```
# ls /usr/share/man/man1/

:.1.gz          make_win_bin_dist.1.gz      pkcs8.1ssl.gz
CA.pl.1ssl.gz  make_win_src_distribution.1.gz  pkill.1.gz
Mail.1.gz      man.1.gz                    pl2pm.1.gz
```

b) View a man page

```
# man man

man(1)                                                    man(1)

NAME
  man - format and display the on-line manual pages
  manpath - determine user's search path for man pages

SYNOPSIS
  man [-acdfFhkKtwW] [--path] [-m system] [-p string] [-C config_file]
```

Using Pseudo

About Using Pseudo (fakestart.sh)

TBD

Concept definition.

Examining Files using Pseudo

Use this information as a guideline for examining files on the target file system using Pseudo.

To examine a file with its settings as they will appear on the target, you can supply the **ls** command to pseudo using **fakestart.sh**. For example:

- `$ scripts/fakestart.sh ls -l export/dist/bin/sh`

The system responds with output similar to the following:

```
lrwxrwxrwx 1 root root 4 Mar 11 11:11 export/dist/bin/sh -> busybox
```

Navigating the Target File System with Pseudo

Use this information as a guideline for viewing target file systems with Pseudo.

You can enter a pseudo shell to move around the target file system and view multiple settings.

Step 1 Start the pseudo shell.

```
$ scripts/fakestart.sh sh
```

Notice that the shell prompt changes to **sh 4.2** to indicate you are using the pseudo shell.

Step 2 Navigate to the platform project bin (binary) directory.

```
sh 4.2# cd export/dist/bin
```

Step 3 List the directory's contents.

```
sh 4.2# ls -l s*
```

The system responds with output similar to the following:

```
lrwxrwxrwx 1 root root 11 Mar 11 11:12 sh -> busybox  
lrwxrwxrwx 1 root root 11 Mar 11 11:12 sleep -> busybox
```

Step 4 Exit the pseudo shell.

```
sh 4.2#exit
```

The **exit** command closes the pseudo shell and return to your normal shell.

26

Deploying Flash or Disk Target Platforms

- About Configuring and Building Bootable Targets 265
- About Configuring a Boot Disk with a USB/ISO Image (Two Build Directories) 266
- Host-Based Installation of Wind River Linux Images 266
- Booting and Installing from a USB or ISO Device 268
- Booting and Installing with QEMU 269
- Configuring and Building the Host Install (Self-Contained Option) 273
- Configuring and Building the Host Install (Two Build Directories Option) 274
- Creating Bootable USB Images 275
- Creating ubifs Bootable Flash Images 278
- Enforcing Read-only Root Target File Systems 279
- Installing a Bootable Image to a Disk 279
- Installing or Updating bzImage 280

About Configuring and Building Bootable Targets

To create a single, bootable image from which to boot a target device, specify the **enable-bootimage** option.

A bootimage option will construct bootable disk or DVD image with the selected file system type, and if necessary multiple partitions, and place the image as single file in the export directory of the project. When you create a single, bootable image from which to boot a target device, the platform project configure options must include **--enable-bootimage=bootimageType**, where *bootimageType* is one of **ext3**, **iso**, **jffs2**, **ubifs**, **tar.gz** or **tar.bz2** (with **tar.bz2** being the default).

For information on creating USB boot images, see [Creating Bootable USB Images](#) on page 275.

For additional information on configuring ubifs boot images, see [Creating ubifs Bootable Flash Images](#) on page 278.

The following is a sample configure command to create an ISO-enabled image:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-bootimage=iso
```

If you need to use additional bootable image options, you can use a comma to add and separate them. For example:

--enable-bootimage=iso,jffs2

After the platform project is configured, use the **make** command to build the target platform image and create a single image with which to boot a target device. The image is placed in the `projectDir/export` directory.

About Configuring a Boot Disk with a USB/ISO Image (Two Build Directories)

To create a self-contained, host-installable image from a reference platform project, you must first have a previously built platform project, or create one.

The following sections describe how to boot from a USB or ISO image so that you can configure a disk that will be your boot disk.

To create a bootable USB directly from a platform project, see [About Configuring and Building Bootable Targets](#) on page 265.

Preliminaries

The following example shows how to use Wind River Linux to configure a common PC to boot Wind River Linux from hard disk. With this method you do not use the server installer and must configure the hard drive manually.

(See [Host-Based Installation of Wind River Linux Images](#) on page 266 for using the installer to automate this procedure).

You can boot and configure your target using either an ISO or USB image as described in this section. The basic procedure is the following:

1. *Configure and Build the Standalone Image*
2. *Prepare the Target's Hard Drive*
3. *Place the File System and Kernel on the Hard Disk*
4. *Configure the Target System Files and Boot*

Host-Based Installation of Wind River Linux Images

Wind River Linux provides options for creating a distribution for installing a platform project image on a host (server) hard disk.



WARNING: The content of this section must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionality.

These options allow for a flexible configuration in which you can create a standalone platform project image or specify a different Wind River Linux platform project image to create your installation disk from. Once complete, the installer lets you specify or accept default sizes for the boot, swap, and root partitions.

Using the Wind River Linux build system, you can create an image to burn to a CD, DVD, or USB device, and then use that image to boot up the target, format the local disk, and install the runtime on the disk. At that point, you can remove the CD, DVD, or USB device and boot the target directly from the local disk. You can also test your build using QEMU as shown in [Booting and Installing with QEMU](#) on page 269.

There are two options, and three methods, to create a host-based installation:

- Self-contained in a single build directory.
In the self-contained installation, the build creates a `/RPM` directory in the root file system, where it puts all the RPM packages that will be used to install the runtime on the target from the same platform project configure options. This installation type requires a single build.
- Using two build directories with a reference platform project for the runtime to install on the target, and one for the install image itself.

Use the `--with-installer-target-build=configure` script option to specify the location of a built platform project's `projectDir/export/images/*.ext3` root file system image file that will be used to create an installable Wind River Linux distribution. So, you first build the project you want to use as a basis to create the installer, specify the location of the project's `*.ext3` file using the `--with-installer-target-build=configure` option, and then perform another build to build everything in your project directory.

This installation type uses one or two builds, depending on whether you have previously configured a platform project image using the `--enable-bootimage=iso` or `(ext3)` and `--enable-target-installer=yes` configure options.



NOTE: Wind River Linux only supports `*.ext3` root file system files for the target installer feature.

Use this option for a minimal installer that creates an installable image from an existing platform project image.

The related options for the `configure` command are as follows:

```
--enable-bootimage=iso
--enable-target-installer=yes
--with-installer-target-build=otherprojectDir/export/images/*.ext3
```

Once the build is finished, you will find the ISO image in, for example, `projectDir/export/images/wrlinux-image-glibc-small-qemux86.iso`. This is the file that contains the bootable system image that you will copy to a USB memory stick.

The `--enable-bootimage=iso` option builds an image that you can use to boot from CD, DVD, or USB devices. With this image, you boot directly from a read-only root on the device. Whether you specify `iso` or `usb` (or both) to the `--enable-bootimage` option, the result is a hybrid image that supports booting from both ISO and USB.

The difference between the build types is just a question of where those packages come from—either the same build in which you create the bootable image, or from another build. If you do not specify the `--with-installer-target-build` option, the build system will use the RPMs that are in `export/RPMS/` in your project directory for the installer image.

Booting and Installing from a USB or ISO Device

Use this procedure to place an installable image on a USB or ISO device and install it on a target.

⚠ WARNING: The content of this section must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionality.

In the following example, you will place the installer on a USB device such as a thumb drive, or on an ISO device such as a CD-ROM, and then install on a target.

Step 1 Create a boot image of the desired type.


For information about creating boot images, see:

- [Configuring and Building the Host Install \(Self-Contained Option\)](#) on page 273
- [Configuring and Building the Host Install \(Two Build Directories Option\)](#) on page 274

Step 2 Burn the image to the USB or ISO device.

For information about burning boot images, see *Burning and Installing the ISO or USB Image*

Step 3 Insert the device in the target to be booted.

Options	Description
USB	Insert the device and determine the USB device assigned to the drive.  NOTE: If there is not an operating system already installed on the target to enable the determination of the device associated with the USB thumb drive, use the boot menu selections, select Drive 0 (sda) as the USB device when prompted in step 7 on page 269 of this procedure, below. In the (likely) event that <code>/dev/sda</code> is not the correct device, you can use the kernel log information immediately above the panic message to identify the available devices on the target.
ISO	Insert the device in the CD-ROM or DVD drive.

Step 4 Reboot the target from the appropriate device.

For example, on a laptop, set the boot settings to boot from USB or CD-ROM.

Step 5 Select **Graphics console** then press **ENTER**.

Step 6 Select one of the following device-specific options.

Options	Description
USB	Choose USB Memory Stick (or disk or SCSI disk)
ISO	Choose USB DVD-ROM (or SCSI DVD-ROM)

a) Press **ENTER** to apply your choice.

Step 7 Select **Drive 0 (sr0)** then press **ENTER**.

Step 8 Follow the rest of the procedure as described in [Installing a Bootable Image to a Disk](#) on page 279.

Booting and Installing with QEMU

Once you create an installable Wind River Linux disk image, you can test it using QEMU before you burn it to disk or install it on a hardware device.

⚠ WARNING: The content of this section must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionally.

After building the file system and boot image, perform the following procedure to use QEMU to create, install to, and then boot from a virtual disk.

Step 1 Create the virtual QEMU disk.

Use the **qemu-img** host tool to create and size the virtual disk.

```
$ cd projectDir
$ host-cross/usr/bin/qemu-img create -f qcow hd0.vdisk 5000M
```

Step 2 Boot the ISO image and install Wind River Linux.

a) Boot the **.iso** image you created in [Configuring and Building the Host Install \(Self-Contained Option\)](#) on page 273.

In this case we will use the graphics console (option **-gc**) with QEMU. Omit this option if you want to use the serial console.

```
$ make start-target \
TOPTS="-cd export/qemux86-64-glibc-small-standard-dist.iso -no-kernel -disk
hd0.vdisk -gc"
```

➔ NOTE: The **-gc** options starts the QEMU session in graphics mode. Omit this option to perform a serial-based, non-graphical installation.

Press **CTRL+ALT** at any time to exit from the boot window. You can click in the window to return control to it.

You can exit the menus entirely by pressing the **ESC** key at the initial menu.

At the boot prompt (boot:) you can get help by pressing **F1** (or **CTRL+F1**). The help information documents the commands available at the prompt.

When the boot loader starts, it presents a series of menus which you use to select a console type, a boot device type, and a specific boot device.

- b) Select **Graphics console** and press ENTER.



➔ **NOTE:** If you did use the `-gc` option in step 2 on page 269, above, select **Serial console**, and refer to the instructions at [Installing a Bootable Image to a Disk](#) on page 279. Once the hard disk installation completes, go to step 3 on page 272, below.

- c) Select **USB DVD-ROM (or SCSI DVD_ROM)** and press ENTER.



- d) Select **Drive 0** and press **ENTER**.



Once you select the drive option, the installation will begin automatically. For instructions on the installation itself, see [Installing a Bootable Image to a Disk](#) on page 279.

Step 3 Boot the installed disk.

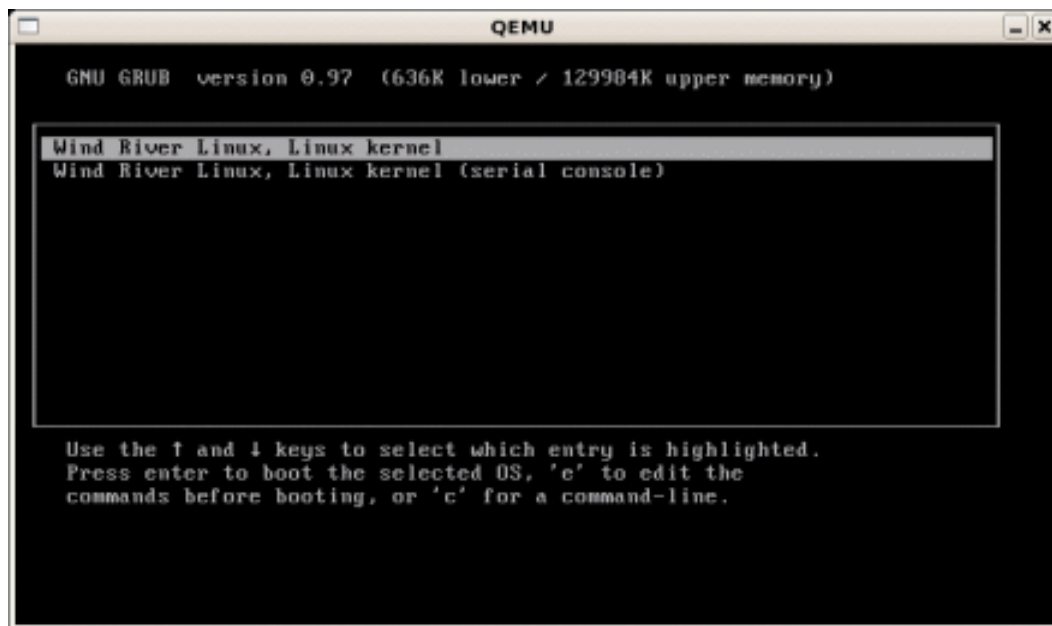
In this example, the installation was performed on the virtual disk you created in step 1 on page 269, above.

- a) Boot from the disk that you installed Wind River Linux on in the previous step.

```
$ make start-target TOPTS=" -no-kernel -disk hd0.vdisk -gc"
```

Do not enter the **-gc** option if you are using the serial console.

- b) Press any key if you are prompted to do so, and then you see the **grub** bootloader menu.



Step 4 Press ENTER to boot from the installed Wind River Linux.

Step 5 At the `localhost login:` prompt, enter `root` and press ENTER to continue.

Configuring and Building the Host Install (Self-Contained Option)

To create a self-contained, host-installable image, you must first configure and build the platform project.

⚠ WARNING: The content of this section must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionality.

Perform the following procedure to configure and build a single project that will furnish the boot image and the packages for your target(s).

To test the installation, you can use QEMU to create, configure, and boot the installation from a virtual disk as described in [Booting and Installing with QEMU](#) on page 269.

Step 1 Configure the bootable image of the server installation.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-bootimage=iso \  
--enable-target-installer=yes
```

Step 2 Create the boot image.

```
$ make fs
```

When you build the file system after specifying the **--enable-bootimage=iso** configure option, it creates a bootable iso image in the platform project's **/export** directory, named after the platform project name, for example: *projectDir/export/qemux86-64-glibc-small-standard-dist.iso*. This is the image you will use to boot and install Wind River Linux on a target system.

The resulting installer will contain the packages from and match the settings of this project's configuration.

Once the build system creates your image, you will want to burn it to a disk. See *Burning and Installing the ISO or USB Image*

Configuring and Building the Host Install (Two Build Directories Option)

To create a self-contained, host-installable image from an existing platform project, you can use an existing platform project, or create one to specify the installable image's configuration and packages.



WARNING: The content of this section must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionality.

The two build directories option requires two builds to create the image:

- One platform project to provide the packages you want to install on your target systems. If you already have an existing reference platform project to use, you can go to step 3 on page 274, and specify the path to your existing platform project directory using the **--with-installer-target-build** configure script option. If not, you will need to create one.
- One project that will furnish the boot image for your targets

Step 1 Configure and build a platform project.

The packages assembled in this project are the ones you will install on your target(s).

For example:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-bootimage=iso \  
--enable-rootfs=glibc_std
```

Step 2 Build the project.

```
$ make
```

Step 3 Configure and build your bootable image.

Specify the location of the packages in your first project with the **--with-installer-target-build** configure option where *otherprojectDir* is the path to the *projectDir* directory of your previous build, or existing reference platform project.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-bootimage=iso \  
--enable-target-installer=yes \  
\
```

```
--with-installer-target-build=otherprojectDir/export/images/wrlinux-image-glibc-small-  
qemux86-64-2013807210107.rootfs.ext3
```

In this example, you must provide the actual file name of the ***.ext3** file above. This name differs depending on your BSP and root file system **configure** script options, and the date the image was built.



NOTE: If you do not specify a project directory with the **--with-installer-target-build** option, the project itself will be used to create the target file system.

Step 4 Create the boot image.

```
$ make
```

The result is a bootable image in the **export** subdirectory of your project build directory, for example

```
projectDir/export/qemux86-64-glibc-small-standard-dist.iso
```


Creating Bootable USB Images

Wind River Linux provides **make** command options for creating bootable USB images.

The following procedure describes how to create a bootable USB image and launch it on a target system. Wind River Linux provides three methods for creating bootable USB images to meet most development needs.

See [About Configuring and Building Bootable Targets](#) on page 265 for concepts related to deploying target images.

Step 1 Choose an option to create a bootable image.

Options	Description
Method 1: ISO Hybrid Boot with make fs	<p>The image you create with this option is called an ISO hybrid because you can burn it to a CDR, DVD, or USB flash for the purpose of booting an x86 system. The boot menu that displays on the target provides two options: serial or graphics.</p> <p>To use this option, you must configure a platform project image with the --enable-bootimage=iso option. For example:</p> <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-rootfs=glibc_small \ --enable-kernel=standard \ --enable-bootimage=iso</pre>
	<p> NOTE: For additional information on configuring platform projects, see About Configuring a Platform Project Image on page 62.</p>
	<p>The --enable-bootimage=iso option adds several packages to the target file system, including syslinux; applies the read-only rootfs template, and changes the kernel configuration options to provide generic ISO boot support.</p>
Method 2: Two-Partition USB Boot with make usb-image	<p>This option creates a bootable USB image from any existing platform project image. The image includes two partitions:</p> <p>16 FAT</p> <p>The first is a small 16 FAT file system for syslinux, the kernel, and a static BusyBox initrd</p> <p>ext2</p> <p>The second is an ext2 file system to mount the root partition for the operating system</p> <p>This method has the following advantages over <i>Method 1: ISO Hybrid</i>:</p> <ul style="list-style-type: none">• Does not require any specific configure options, so you can create a USB file system from any x86-based platform project.• Does not require extra packages on the target file system• You may mount the file system as read/write to provide persistent changes.• You can later update the file system, instead of entirely re-writing it because you can simply mount it on your host.• Works with older BIOS which do not support the ISO hybrid boot. <p>Additionally, it is not necessary to configure the kernel to support USB flash devices, or the ext2 file system in order to use this feature, since this is generally the default for all BSPs.</p>

Options	Description
Method 3: Two-Partition USB Boot Directly to Disk with make usb-image-burn	This option works in the same manner as <i>Method 2: Two-Partition USB Boot</i> , but saves you a step by writing the image directly to the USB device. This method allows you to avoid reformatting the USB device if you already have a partition layout you are happy with.

Step 2 In the platform project directory, enter one of the following commands to create the boot image:

Options	Description
For Method 1:	<pre>\$ make fs</pre> <p>When you build the file system after specifying the --enable-bootimage=iso configure option, it creates a bootable iso image in the platform project's /export directory, named after the platform project name, for example:</p> <pre>projectDir/export/qemux86-64-boot.iso</pre>
For Method 2:	<pre>\$ make usb-image</pre> <p>When you run this command, it creates a bootable image file in the following location:</p> <pre>projectDir/export/usb.img</pre>
For Method 3:	<pre>\$ make usb-image-burn</pre> <p>This option requires that you have inserted a USB device with no mounted partitions. For an example output of this script, see <i>make usb-image-burn Example Output</i>. Once you run this command, go to <i>Step 5</i>, below.</p>

With the Method 1 and 2 commands, you have the option of pressing **ENTER** at each prompt to select the default. For Method 2, you may optionally choose to mount the file system read/write, to provide persistent storage to the USB flash device.

The following example output displays **make usb-image** options once you run the command:

```
Location to write image file [/Builds/qemux86-64-glibc_std/export/usb.img]:
Size of FAT16 boot <#MEGS> [64]:

    The size of export dist is: 37MB
    The creation program automatically adds 100MB
    NOTE: You can make size of the ext2fs partition as large as you like
          so long as it does not exceed the size of the target device.
Size of ext2 fs <#MEGS> [137]:
Use ext 2, 3, 4 [2]:
Location of bzImage [/Labs/qemux86-glibc_std/export/qemux86-bzImage-
WR5.0.1.0_standard]:
Make root file system readonly? <y/n> [y]:
```

Step 3 Find the device name of your USB device and make sure it is unmounted. For this example we'll assume it is **/dev/sdc**.

Step 4 Run one of the following commands to write the image to the USB character device:

Options	Description
For Method 1:	<pre>\$ dd if=export/qemux86_64-boot.iso of=/dev/sdc bs=1M</pre>
For Method 2:	<pre>\$ dd if=export/usb.img of=/dev/sdc bs=1M</pre>

➔ **NOTE:** For the *Method 3: Two-Partition USB Boot Directly to Disk* option, once the command finishes, the disk is written to automatically in the same manner as in this step for the other options.

Step 5 Run the following command to unmount and eject the USB:

```
$ sudo eject /dev/sdc
```

Step 6 Optionally launch the USB flash drive on a target. You can use the new USB flash device to boot a hardware device. To test your new image using QEMU, enter one of the following commands:

- **For Method 1:**

```
$ make start-target \  
TOPTS="-disk export/qemux86_64-boot.iso" \  
TARGET_QEMU_KERNEL=""
```

- **For Method 2:**

```
$ make start-target \  
TOPTS="-disk export/usb.img" \  
TARGET_QEMU_KERNEL=""
```

When the QEMU session begins, you will be prompted to choose a console option. Select **serial** to continue loading the image.

Creating ubifs Bootable Flash Images

Learn about the options available for creating bootable ubifs images.

Wind River Linux provides the option to create a **ubifs** target image.

Step 1 Create a **ubifs** target image using the following platform project configure options:

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-bootimage=ubifs \  
--with-mkubifs-args="-m 2048 -e 129024 -c 1996"
```

In this configure example, the **--with-mkubifs-args=** option specifies the following ubifs parameters:

-m

The minimum-I/O-unit-size.

-e

The logical-eraseblock-size.

-c

The maximum-logical-eraseblock-count.

Step 2 Build the target platform image and create a single image with which to boot a target device.

```
$ make
```

The image is placed in the `projectDir/export` directory.

Postrequisites

See [About Configuring and Building Bootable Targets](#) on page 265 for concepts related to deploying target images.

Enforcing Read-only Root Target File Systems

Learn how to build your target with a read-only root file system.

This feature causes the root filesystem to be mounted read-only with writable data (directories under `/var`) located on a ram disk. It is designed to be used on `glibc_small` and `glibc_standard` root file systems. Read only file systems are useful in situations in which you want the environment to revert to a pristine state at each boot, such as a terminal or consumer electronics. Read only root file systems can also be faster to boot since they don't need to be checked. Packages added either by SATO or manually may need to be updated by hand to work with a read-only root filesystem.

In the following example, we will create an ISO boot image.

Step 1 Configure the platform project to make the root file system read-only.

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-bootimage=iso \  
--with-template=feature/readonly-root
```

Step 2 Build the file system.

```
$ make fs
```

Postrequisites

Depending on the type of image you create, additional steps are required. Refer to the instructions for each and add `--with-template=feature/readonly-root` to the `configure` command as illustrated above.

Installing a Bootable Image to a Disk

Once you create an installable Wind River Linux image using the `--enable-target-installer=yes` configure option, use this procedure to install it.

This procedure works to install either to a hard disk, or when using QEMU to test an installation using a virtual disk, as described in [Booting and Installing with QEMU](#) on page 269.

This procedure assumes that you have previously created an installable **.iso** disk image as described in [Creating Bootable USB Images](#) on page 275 or [About Configuring a Boot Disk with a USB/ISO Image \(Two Build Directories\)](#) on page 266.

Step 1 Press **ENTER**, or modify the values to meet your specifications for the following.

```
Disk to format (default is sda) Enter
What disk do you want to format? (sda) [sda]
MB for the /boot partition (default is 100) Enter
MB for the swap partition (default is 240) Enter
MB for the root partition (default is rest of disk) Enter
...
If you wish to modify the default list of packages to be installed enter
"yes" at the prompt: Enter
...
Would you like to view or modify what will be installed? [no] Enter
...
Press ENTER to continue to the installation. Enter
```

Press **CTRL-C** at any time to exit to the shell.

Once you press **ENTER** to continue the installation, the installation progress displays graphically in the terminal window. Depending on the size of your installation and its required packages, installation may take some time.

Step 2 Enable the serial console if prompted.

```
Do you want login enabled on the serial console? [no] Enter
```

Accept the default **no** here.

The following message will display when the step completes:

```
Initial installation is now complete.
Please remove the install media and reboot from the hard disk.
```

Step 3 Remove the boot device (USB or ISO) from the server and reboot from the hard disk.

If you are using a virtual disk such as in the example described in [Booting and Installing with QEMU](#) on page 269, just close the graphics window (or press **CTRL-A, X** if you are using the serial console).

Installing or Updating bzImage

Add `INHERIT_append = "kernel-grub"` to your **local.conf** file to allow fallback boot options.

This procedure assumes that the boot partition is writable and has at least 5 MB of free space.



WARNING: The content of this section must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionality.

Complete the following steps to allow your current kernel to be preserved as a fall-back boot option during updates.

Step 1 Download a new kernel image rpm file to the target.

Step 2 Examine the **boot** directory.

```
# ls /boot/
grub          vmlinuz
```

Step 3 Review your grub configuration.

```
# cat /boot/grub/grub.cfg
```

If you are using Grub 0.97, substitute **menu.lst** for **grub.cfg**

You should see content similar to the following:

```
menuentry "Linux" {
    set root=(hd0,1)
    linux /vmlinuz root=/dev/hdb2 rw console=tty0 quiet
```

Notice that only one boot entry exists.

Step 4 Install or update **bzImage**.

For example:

```
# rpm -i kernel-image-3.10.11-yocto-standard-3.10.12+git0+285f93bf94_702040ac7c-r0.qemux86_64.rpm
```

Installing the same rpm more than once with the **--force** option will result in multiple kernel images in the boot directory and grub menu.



WARNING: Updating the **bzImage** file can adversely affect compatibility with the kernel-module.

Step 5 Confirm the update or install:

```
# ls /boot/ -al
```

You should see a listing similar to the following:

```
drwxr-xr-x  4 root    root          1024 Sep 18 06:58 .
drwxr-xr-x 17 root    root          4096 Sep 18 06:41 ..
lrwxrwxrwx  1 root    root           30 Sep 18 06:58 bzImage -> bzImage-3.10.11-
yocto-standard
-rw-r--r--  1 root    root          5601808 Sep 18 06:45 bzImage-3.10.11-yocto-standard
drwxr-xr-x  4 root    root           1024 Sep 18 06:58 grub
-rwxr-x---  1 root    root          5601776 Sep 18 06:38 vmlinuz
```

Step 6 Review your grub configuration.

```
# cat /boot/grub/grub.cfg
```

If you are using GRUB 0.97, substitute **menu.lst** for **grub.cfg**

You should see a new boot entry similar to the following:

```
menuentry "Update bzImage-3.10.11-yocto-standard-3.10.12+gitAUTOINC
+285f93bf94_702040ac7c" {
```

```
set root=(hd0,1)
linux /bzImage-3.10.11-yocto-standard root=/dev/hdb2 rw console=tty0 quiet
}
menuentry "Linux" {
set root=(hd0,1)
linux /vmlinuz root=/dev/hdb2 rw console=tty0 quiet
}
```

Notice that both the new and original boot entries exist.

Step 7 Reboot the target.

You will see a new option on the boot menu similar to the following:

```
Update bzImage-3.10.11-yocto-standard-3.10.12+gitAUTOINC+285f93bf94_702040ac7c
```

Deploying *initramfs* System Images

[About *initramfs* System Images](#) 283

[Creating *initramfs* Images](#) 284

[Adding Packages to *initramfs* Images](#) 285

About *initramfs* System Images

Wind River Linux expands on the *initramfs* support found in the Yocto Project by providing the ability to specify the contents of the image and also bundle the image with a kernel image.

With Wind River Linux, you have the option of creating a basic *initramfs* image, or an image that is bundled with the kernel image. The idea behind using a bundled *initramfs* image is that a `cpio` archive can be attached to the kernel image itself. At boot time, the kernel unpacks the archive into a RAM-based disk, which is then mounted and used as the initial root filesystem. The advantage of bundling (building) an image into the kernel is that the rootfs is already provided in the archive, so you do not require one.



NOTE: Bundling the *initramfs* may cause boot issues if the image size becomes too large.

When you create an *initramfs* image, it is important to note that because the image does not include a kernel, you cannot have packages that depend on the kernel in the image, otherwise it will create a circular dependency when you run `make` to build the platform project image. The circular dependency occurs because the kernel is waiting for the image and vice-versa. In this case, the build output will display the following error message, in a loop:

```
Aborted dependency loops search after 10 matches
Aborted dependency loops search after 10 matches
Aborted dependency loops search after 10 matches
...
```

For information on troubleshooting circular dependencies, refer to the `projectDir/ayers/wr-kernel/Documentation/initramfs.txt` file.

Regardless of the image type, creating an *initramfs* image is done by including the `feature/initramfs` template when you configure the project. For additional information, see [Creating *initramfs* Images](#) on page 284.

Creating initramfs Images

You can create a basic image, or one bundled with a kernel, depending on your development needs.

Use the procedures in this section to create an initramfs platform project image. For information on initramfs images in general, see: [About initramfs System Images](#) on page 283.

After the build is complete, your image will be available in the `projectDir/export/images` directory. For example:

`projectDir/export/images/wrlinux-image-initramfs-qemux86-64.cpio.gz`

Step 1 Create a platform project directory and navigate to it.

For example:

```
$ mkdir qemux86-64-glibc-small-initramfs
$ cd qemux86-64-glibc-small-initramfs
```

Step 2 Configure a platform project using the initramfs feature template.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-rootfs=glibc_small \
--enable-kernel=standard \
--with-template=feature/initramfs
```

To create an initramfs image bundled with a kernel, replace the `--with-template=feature/initramfs` configure option with `--with-template=feature/initramfs-integrated`.

You may also substitute the `--with-template=feature/initramfs` option by adding `+initramfs` to the end of the `--enable-rootfs` option, for example: `--enable-rootfs=glibc_small+initramfs` for a basic image and `--enable-rootfs=glibc_small+initramfs-integrated` for a bundled kernel image.



NOTE: It is important that you specify `glibc_small` as the root file system. Using `glibc_core` in this example will cause boot failures.

Step 3 Build the file system and create the initramfs image(s).

```
$ make
```

This command can take some time to complete, depending on your host system configuration. Once complete, your image(s) are available in the `projectDir/export/images` directory. These images include the main initramfs image and a separate rootfs image. For example:

- `wrlinux-image-initramfs-qemux86-64.cpio.gz`
- `wrlinux-image-initramfs-qemux86-64-20130219204726.rootfs.cpio.gz`

Images bundled with a kernel include the kernel image with initramfs, and one without. For example:

- `bzImage-initramfs-x86-64.bin`
- `bzImage-x86-64.bin`

Adding Packages to initramfs Images

Once you create an initramfs image, you may need to add packages to it to meet your specific development requirements.

Use the procedures in this section to add packages to an initramfs image created using the procedures in [Creating initramfs Images](#) on page 284. For information on initramfs images in general, see: [About initramfs System Images](#) on page 283.

Step 1 Open the `projectDir/layers/local/recipes-img/images/wrlinux-image-file-system.bb` file in an editor.

In this example, `file-system` represents the name of the rootfs used to configure the platform project. If you configured your platform project using the instructions in [Creating initramfs Images](#) on page 284, the file name would be

`projectDir/layers/local/recipes-img/images/wrlinux-image-glibc-small.bb`

Step 2 Add the package to the file.

For each package you want to add, add the following line to the file, after the line that reads `###
END Auto Generated by configure ###`:

```
### END Auto Generated by configure ###  
IMAGE_INSTALL_pn-wrlinux-image-initramfs += "packageName1"  
IMAGE_INSTALL_pn-wrlinux-image-initramfs += "packageName2"
```

When adding packages, it is important to keep track of the package's footprint, since adding too many, or too large a package may make the initramfs image too large to function on your hardware. Refer to your hardware manufacturer's documentation for memory footprint limitations.

Step 3 Save the file once you are finished.

Step 4 Rebuild the platform project.

```
$ make
```

Once the project build completes, the initramfs images located in the `projectDir/export/images` directory will include the new packages. For specific information the initramfs image names, see [Creating initramfs Images](#) on page 284.

28

Deploying KVM System Images

[About Creating and Deploying KVM Guest Images](#) 287

[Create the Host and Guest Systems](#) 289

[Deploying a KVM Host and Guest](#) 290

About Creating and Deploying KVM Guest Images

Learn about the requirements for deploying and networking a Linux guest system using the kernel virtual machine (KVM)-enabled `x86_64_kvm_guest` BSP and virtio.

About Multiple Environment Systems

In an environment where code is running in multiple OS environments, it is easy to get confused about what we are building and running on. To minimize this confusion, this document specifically refers to three general environments:

Build host

The computer/OS where all code is compiled.

KVM host

This is the primary OS running on the hardware machine of the target.

KVM guest

This is the OS that is running in (in the case of the example in this guide) a QEMU emulation running on the KVM host.

KVM refers to QEMU (userspace) with KVM acceleration enabled, with KVM kernel extensions, and hardware support.

In this guide, the KVM procedure uses two separate operating systems running on the same target hardware. These systems are referred to as the host and guest OS. The host environment, is assumed to be implemented as a 64-bit BSP, on a hardware target board. The guest can be either 32- or 64-bit, but for this example we will use a 64-bit BSP.

About Networking with MacVTap, virtio and netperf

virtio

To provide networking between a host and a guest operating system, KVM provides several emulated device and para-virtualized device options for host to guest network communications. This is implemented using **virtio**, which is a series of efficient, well-maintained Linux drivers designed for that purpose. This includes a simple extensible feature mechanism for each driver, a ring buffer transport implementation called vring, and more. The procedure in this guide uses **virtio** paravirtualization because of its ease of setup and low-latency characteristics.

VhostNet

To improve network latency and throughput, you can optionally use VhostNet as part of your system configuration. VhostNet replaces QEMU interaction in the transmission and receipt of packets with a kernel module. To implement VhostNet, the host and guest kernel must be configured with the following parameters and components:

Host

The **CONFIG_VHOST_NET=y** option is included automatically when you configure your host using the **--with-template=feature/kvm** configure option.

Guest

- **CONFIG_PCI_MSI=y**
- **CONFIG_VIRTIO_PCI**
- **CONFIG_VIRTIO_NET**

These guest options are added automatically when you create the guest using the x86-64-kvm-guest BSP as explained in this guide.

In addition, VHostNet requires x86-64 support as well as **AMD_IOMMU** for the guest. For additional information on using VhostNet, see <http://www.linux-kvm.org/page/VhostNet>.

MacVTap

To create virtual MAC address on the host for the guest to communicate with, MacVTap is used. MacVTap is a device driver designed to simplify virtualized bridged networking. It replaces the combination of the TUN/TAP and bridge drivers with a single module based on the MacVLan device driver. A MacVTap endpoint is typically a character device that largely follows the TUN/TAP **ioctl** interface and can be used directly by KVM/QEMU and other hypervisor implementations that support the TUN/TAP interface. The endpoint extends an existing network interface, the lower device, and has its own MAC address on the same Ethernet segment, or the host in our example.

MacVTap operates in several modes, but for the purposes of the procedure in this guide - for host to guest/guest to host communications - MacVTap is configured to use *bridge* mode. This is necessary because some modes actually rely on external hardware to route packets, which slows communication from host to guest (and back), and requires external hardware to be configured correctly, which is out of scope for this procedure.

netperf

To test network performance between the host and guest, the netperf package is used. Due to licensing restrictions, it is necessary for you to obtain the netperf package at <http://www.netperf.org>.

Create the Host and Guest Systems

Use the following procedure to create a KVM host and guest system for deployment.

The following procedure creates QEMU KVM host and guest target platforms ready for deployment. Before you begin, if you want to test the network performance between the host and guest, you will need to obtain the **netperf 2.6.0** package from <http://www.netperf.org>.

Step 1 Configure a platform project image for the KVM host system.

Run the following **configure** command on the build host:

```
$ configDir/configure \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-board=intel-xeon-core \  
--with-template=feature/kvm
```

The configure command may take a couple moments to complete.

Step 2 Install the **netperf** package.

a) Obtain the **netperf** package (**netperf-2.6.0.tar.bz2**).

This package is available online from <http://www.netperf.org>.

b) Place the **netperf** package in the **projectDir/layers/local/download** directory.

c) Update the **projectDir/local.conf** file, **LICENSE_FLAGS_WHITELIST** option to read:

```
LICENSE_FLAGS_WHITELIST += "non-commercial"
```

d) Build and add the package.

```
$ make -C build netperf.addpkg
```

Step 3 Add the **bridge-util** package.

```
$ make -C build bridge-utils.addpkg
```

Step 4 Add the **wrs-kvm-helper** package.

```
$ make -C build wrs-kvm-helper.addpkg
```

Step 5 Build the KVM host platform project image.

```
$ make fs
```

The build process can take some time. Once it completes, your KVM host is ready for deployment.

Step 6 Configure a platform project image for the KVM guest system.

```
$ configDir/configure \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-board=x86-64-kvm-guest
```

The **configure** command may take a couple moments to complete.

Step 7 Install the **netperf** package.

- a) Obtain the **netperf** package (**netperf-2.6.0.tar.bz2**). from and

This package is available online from <http://www.netperf.org>.

- b) Place the **netperf** package in the **projectDir/layers/local/download** directory.
c) Update the **projectDir/local.conf** file, **LICENSE_FLAGS_WHITELIST** option to read:

```
LICENSE_FLAGS_WHITELIST += "non-commercial"
```

- d) Build and add the package.

```
$ make -C build netperf.addpkg
```

Step 8 Build the KVM guest platform project image.

```
$ make fs
```

The build process can take some time. Once it completes, your KVM host is ready for deployment.

Step 9 Create a USB image to boot the KVM guest.

```
$ make usb-image
```

Accept the defaults to create the image, with the exception of the ext type. Choose **ext 3** when prompted.

Also, if you wish to make changes to the root file system on the guest, enter **n** (no) when prompted to make the root file system readonly.

Step 10 Copy the KVM guest image to the root directory of the KVM host platform project.

For example, you would copy the guest **projectDir/export/usb.img** image file to the host platform project's **projectDir/export/dist** directory.

You now should have a KVM host and KVM guest platform project image ready for deployment.

Deploying a KVM Host and Guest

Once you have configured and built KVM host and guest images, use this procedure to deploy them and set up networking.

To perform this procedure, you must have a previously built KVM host and guest platform project image as described in [Create the Host and Guest Systems](#) on page 289.

Step 1 Boot the KVM host platform and log in as root.

This process differs, depending on your hardware target board setup. Refer to your board manufacturer's procedures for additional information.

Step 2 Insert the KVM kernel mod.

```
root@host0 # modprobe kvm-intel
```

Step 3 Configure the MacVTap interface.

```
root@host0 # ip link add link eth0 name macvtap0 type macvtap
root@gemu0 # ip link set macvtap0 address 1a:46:0b:ca:bc:7b up
root@gemu0 # ip link show macvtap0
```

Step 4 Verify that the MacVTap driver is included in the KVM host kernel.

The expected result displays immediately after the command:

```
root@host0 #dmesg | grep macv
macvtap0: no IPv6 routers present
```

With the virtio device nodes set up and MacVTap configured, the KVM host is ready to boot the KVM guest.

Step 5 Verify that the tap device is available.

The expected result displays immediately after the command:

```
root@qemu0 #ls /dev/tap*
/dev/tap7
```

Step 6 Boot the KVM guest.

Run the following command in the root (/) directory on the KVM host:

```
root@host0 #qemu-system-x86_64 -nographic -k en-us -m 1024 \
-net user,hostname="kvm-guest" \
-enable-kvm \
-net nic,macaddr=1a:46:0b:ca:bc:7d,model=virtio \
-net tap \
-drive file=usb.img,if=virtio
```

Step 7 Select the boot option and log in.

When the guest starts to boot, select **Serial console** at the graphical prompt. Once the guest boots, enter **root** for the user name and password to log in.

Step 8 Start the guest network interface.

Run the following command on the KVM guest:

```
root@kvm-guest~# ifconfig eth0 10.0.2.15
```

Step 9 Start **netserver** and **netperf**.

a) Log into the KVM host.

On the build host, log into the KVM host using **ssh**:

```
$ ssh root@qemu0
```

When prompted, enter **root** for password.

b) Start **netserver**.

```
root@qemu0 #netserver
Starting netserver at port 12865
Starting netserver at hostname 0.0.0.0 port 12865 and family AF_UNSPEC
```

Step 10 Start **netperf** to display network throughput between the KVM host and guest.

Run the following command on the KVM guest:

```
root@kvm-guest~# netperf -H 10.0.2.2 -l 60
```

```
TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to host_ip port 0 AF_INET
```

```
Recv  Send  Send  
Socket Socket Message Elapsed  
Size  Size  Size  Time  Throughput  
bytes bytes bytes secs.  10^6bits/sec  
87380 16384 16384 60.08 91.87
```

When the **netperf** command runs successfully, the very presence of bytes sent and received indicates that the KVM host and guest are networked correctly.

Testing

Running Linux Standard Base (LSB) Tests.....	295
--	-----

29

Running Linux Standard Base (LSB) Tests

[About the LSB Tests](#) 295

[Testing LSB on Previously Configured and Built Target Platforms](#) 296

[Disabling Grsecurity Kernel Configurations on CGL Kernels](#) 297

[Running LSB Distribution Tests](#) 297

[Running LSB Application Tests](#) 299

About the LSB Tests

Understand the requirements and preparation necessary to run the LSB tests on your hardware target platform.

LSB Distribution Tests Overview

The goal of the LSB is to develop and promote a set of open standards that will increase compatibility among Linux distributions and enable software applications to run on any compliant system, even in binary form. In addition, the LSB will help coordinate efforts to recruit software vendors to port and write products for Linux Operating System.

Wind River Linux provides the **lsbtesting** feature template (**--with-template=feature/lsbtesting** configure option) to prepare a target platform to run the LSB Distribution Checker (**/usr/bin/LSB_Test.sh** script). This template automatically installs the packages required to support the LSB tests on your hardware target platform. In addition, you can also use the **hello-world** application to test LSB application compliance. For additional information, see [Running LSB Application Tests](#) on page 299.

When you run the **/usr/bin/LSB_Test.sh** script on the target system, it starts a HTTP web server that automatically retrieves and installs the files required to run the distribution tests. A full installation requires approximately 1.2 GB of available disk space. If you include the application battery packages as part of running the tests, this will increase the disk space requirement even more. Before running the tests, the target platform should have three to five GB of available space.

There are some prerequisites for the test suites to be able to run as listed in the `/opt/lsb-test/manager/README` file. If they are not met, an error message will display when you try to run the test(s). If this occurs, you must resolve the problem(s) and run the tests again.

LSB Test Requirements

In addition to the disk space requirements described in the previous section, successful execution of the LSB distribution tests requires the following:

- A hardware target platform with Wind River Linux 5.0.1 installed.
- A host development system for cross-development
- A sound device with compatible drivers installed for the ALSA tests
- Root access to the target system
- Internet access on the target system with DNS
- Access to a FTP and/or NFS boot server, preferably with the same geo-location for the host and target
- Three to five GB local storage on the target system



NOTE: Local storage is necessary, as running the LSB tests over a NFS-based root file system it is not recommended.

- (Optional) An X server if you wish to run the graphical tests

In addition, the target system requires a network subsystem with the following:

- Ability to resolve its hostname and localhost. You can accomplish this using the `ip=dhcp` flag at boot time.
- Available ports, including port 80 for the LSB Apache tests, and others required by specific tests, such as port 5000. Stop any applications or processes that run on port 80, such as `boa`.



NOTE: Running `sshd` on the standard port should not cause any conflicts.

Testing LSB on Previously Configured and Built Target Platforms

Learn how to download and install LSB tests directly to a pre-built target

If your hardware target platform is already up and running, and you do not want to reconfigure and build it to add the LSB tests with the `lsbtesting` feature template, you can download the tests directly to your platform.

Step 1 Download and install tests.

Wind River recommends using the `--with-template=feature/lsbtesting` configure option over downloading it directly. This option adds extra, required packages to the target platform image's root file system, and configures other packages as necessary to enable LSB testing.

- a) Download the tests.

Download the tests directly to your platform at <http://ftp.linuxfoundation.org/pub/lsb/snapshots/distribution-checker>

- b) Install for the architect of your test platform.

Run the included `install.pl` script as `root`.

Step 2 Start the interface.

Run the following command on the target platform:

```
root@target>~# /opt/lsb/test/manager/bin/dist-checker-start.pl
```

This starts an HTTP web server with a web interface where you can select and run the various tests from.

Disabling Grsecurity Kernel Configurations on CGL Kernels

Learn how to check if Grsecurity is enabled for your kernel, and disable it if necessary for LSB testing

For target platforms with a standard kernel type, Grsecurity must be disabled for the LSB core tests to run successfully.

Step 1 Start the Kernel Configuration tool.

```
$ make -C build linux-windriver.menuconfig
```

Step 2 Disable Grsecurity.

a) Deselect **Grsecurity**

In the Kernel Configuration tool, select **Kernel Configuration > Security options** and deselect **Grsecurity** if necessary.

b) Rebuild the kernel if you made changes.

```
$ make -C build linux-windriver.rebuild
```

Step 3 Disable **tpe** on the target.



NOTE: This step only applies to target platforms with a CGL-enabled kernel type.

Run the following command on the target platform to disable **tpe**:

```
root@target>~# sysctl -w kernel.grsecurity.tpe=0
```

Running LSB Distribution Tests

Use the following procedure to run the LSB distribution tests on your hardware target platform.

Before you begin, make sure that your target system meets the requirements in [About the LSB Tests](#) on page 295.

Step 1 Launch the hardware target platform from the host system.

Step 2 Log into the target as **root**.

Step 3 Start the tests.

On the target platform console, enter the following:

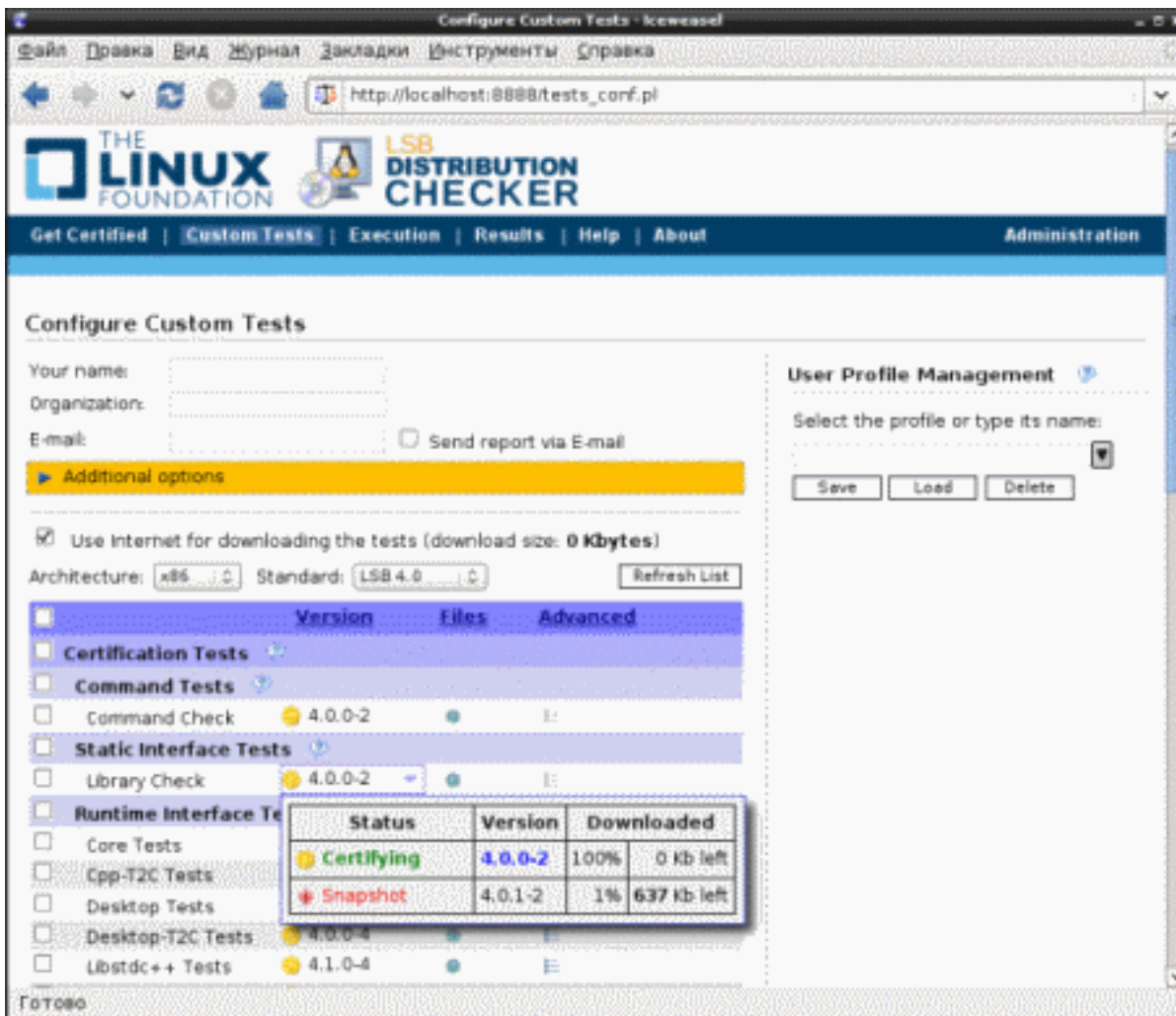
```
root@target~# sh /usr/bin/LSB_Test.sh
```

The script will begin downloading the required files, and will start the tests once complete.

Step 4 Connect to the test interface.

The test interface at **http://localhost:8888** on the local machine, or from a remote machine with network connectivity at **http://targetIP:8888**.

The main page of the LSB Distribution Checker interface lets you choose whether you want to perform certification tests, or select a specific set of custom tests. If you select **Custom Tests**, a page displays where you can select the specific test, or tests, that you want to run.



Step 5 Select the tests you want to run and click **Run Selected Tests**.

The Execution page loads to provide details for each test as it is run. Required components are downloaded to facilitate test completion. To see test results, select the **Results** link. You can toggle between the Execution and Results page as much as you. This does interrupt the tests.

Step 6 View the results.

When the tests complete, the Results page loads automatically. It may be necessary to run some tests manually, depending on the results and information provided by the LSB Distribution Checker.

Should a failure occur in a given test, the LSB Distribution Tester provides summaries and logs to help you troubleshoot the problem. In addition, a history of previous LSB tests can be retrieved, as each test run is stored as a tar file on the target platform.

For additional information on the LSB tests, go to the Linux Foundation's website located at: <http://www.linuxfoundation.org/collaborate/workgroups/lfb>.

Running LSB Application Tests

In addition to the LSB distribution tests, you can use the LSB application tests to verify that an application meets LSB requirements.

This procedure uses the Hello World application included with Wind River Linux to demonstrate how to verify LSB application compliance. Before you begin, make sure that your target system meets the requirements in *About the LSB Tests* on page 295. In addition, you must download and install the LSB Application Checker that matches your architecture from the Linux Foundation website at: http://ispras.linuxbase.org/index.php/Linux_App_Checker_Releases on the target platform. You can use the installable or the local all-in-one version.



NOTE: For PPC 64-bit target platforms, the default userspace is 32-bit. As a result, the correct architecture to download is **ppc32**.

Refer to the **README** file located in the main directory where the LSB Application Checker is extracted or installed to. This file provides information on the prerequisites for running each test. If these prerequisites are not met, the tests will most likely fail.

Step 1 Add the Hello World application to your target system's root file system.

For instructions, see the *Wind River Linux Getting Started Guide: Creating and Deploying an Application*.

Step 2 Launch the hardware target platform from the host.

Step 3 Log into the target as **root**.

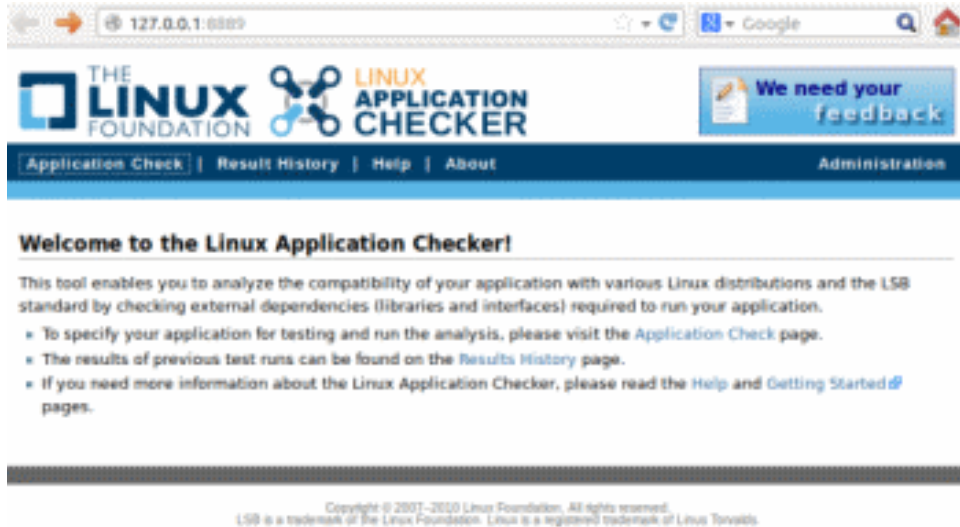
Step 4 Start the test interface.

On the target platform console, enter the following:

```
root@target~# ./app-checker/bin/app-checker-start.pl port_number
```

Where *port_number* is optional. The default port is 8889. If another application or process uses the default, you may specify another port.

The script will locate your system's web browser and launch the interface.



If the script cannot locate the system's web browser, you may start the browser manually and specify the address **http://localhost:8889**. If you specified a different port to start the interface, replace **8889** with that port number.

Optionally, you can connect the LSB Application Checker from any remote computer with network connectivity to the target system. To do so, enter the URL to the target system in the remote computer's web browser:

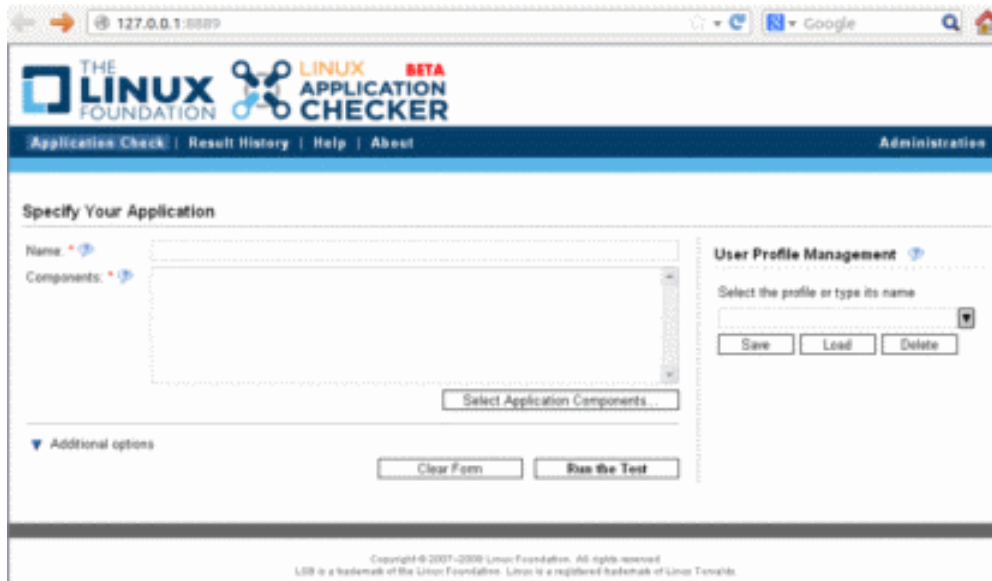
```
http://targetIP:8889
```

To enable this remote feature, modify the **AcceptConnections** option in the server configuration file and restart the server. This configuration file is located in one of the following locations:

- **/etc/opt/lsb/app-checker/app-checker-server.conf** for the installable version running system-wide
- **./config/app-checker-server.conf** for the non-installable version

Step 5 Select the **Application Check** link.

The Specify Your Application page loads.

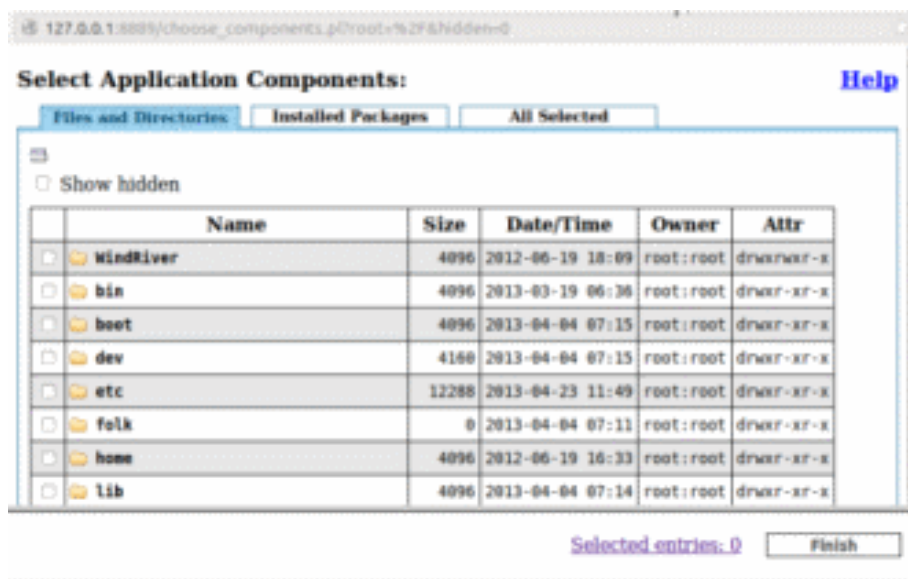


Step 6 Enter application-specific information for the application you want to test.

- a) In the Name field, enter a name for your test report.
- b) In the Components field, enter the file path to the application, for example: `/usr/bin/hello`
- c) Click **Select Application Components** to specify the components associated with the application. This includes:

- Individual files
- Whole directories
- Installed RPM packages (preended with **pkg**)
- RPM and DEB package files
- TAR.GZ and TAR.BZ2 archives

The tool will unpack archive files, but only test the following file types: ELF executables and shared libraries, Perl, Python, and Shell scripts.



Navigate to the location of your application's components and select each component to add it.



NOTE: The Hello World (**hello**) application is a standalone binary. As a result, there is nothing additional to select.

- d) Once component selection is complete, click **Finish** to add them to the Components field.
- e) Optionally, select additional options.

Click **Additional options** to display the advanced test options where you can specify the LSB version and enter comments about the test.

- f) To save the test for future use, enter a name in the User Profile Management section of the page and click **Save**.

Step 7 Once all test parameters are set, click **Run the Test**.

Once the test completes, the results will display on the Test Report page. Review your results, and take appropriate action to correct any issues that arise. In addition, if your test does not identify any compatibility issues, click the **Apply for Certification** link to be directed to the Certification System.

PART IX

Optimization

About Optimization.....	305
Analyzing and Optimizing Runtime Footprint.....	307
Reducing the Footprint.....	313
Analyzing and Optimizing Boot Time.....	317

30

About Optimization

This section provides information on optimizing your platform project image to prepare it for product deployment.

Reducing Footprint

Reducing the footprint of the platform image to fit on smaller memory footprint devices. See:

- [About BusyBox](#) on page 313
- [About devshell](#) on page 314

Analyzing Boot-time

Analyzing boot-time (system startup) and using the data to reduce it. See:

- [Analyzing and Optimizing Boot Time](#) on page 317
- [About Reducing Early Boot Time](#) on page 321
- [Reducing Network Initialization Time with Sleep Statements](#) on page 321
- [Reducing Device Initialization Time](#) on page 323
- [Removing Unnecessary Device Initialization Times](#) on page 323

Reducing File System Size

Reducing file system size by statically linking required libraries to their binaries, and optimizing them. See: [About Static Linking](#) on page 314

31

Analyzing and Optimizing Runtime Footprint

[Analyzing and Optimizing Runtime Footprint](#) 307

[Collecting Platform Project Footprint Data](#) 308

[Footprint \(fetch-footprint.sh\) Command Option Reference](#) 311

Analyzing and Optimizing Runtime Footprint

Use the **fetch-footprint.sh** tool to gather information that can help you reduce the size of your target image (footprint).

The **fetch-footprint.sh** tool is a **bash** script that recurses through a file system and captures runtime information on file and directory access and modification times. It outputs results based on the kind of information you request and the output format you desire, including output that can be used by Workbench or other tools.

You can build your runtime from Workbench or the command line. But note that if you want to use the output to analyze your platform project in Workbench, you must view the output from the project in which you created the runtime. If you build the project from the command line and want to view **fetch-footprint.sh** XML output in Workbench to analyze that project, you must import the project into Workbench. Note that **fetch-footprint.sh** is designed to ignore the **/proc/** and **/sys/** directories.

The **fetch-footprint.sh** script recognizes touched files based on the clock minute and second. If you are using an NFS file system or a QEMU session, you may notice that the target and its file system may be slightly out-of-sync with system time. If the time difference is small, you may be missing the most immediately changed files, and if the difference is radical, then you may not get any samples at all. If this occurs, obtaining an accurate time difference between the host and the target system, plus extending your sample session, will help you adjust timestamps accordingly.

Collecting Platform Project Footprint Data

Learn how to configure a platform project to collect footprint data, and to optionally use Workbench to analyze that data.

Collecting footprint data requires that you configure a platform project with the **--with-template=feature/footprint** configure option, and run the **fetch-footprint.sh** script once the project's file system is built. Optionally, you can create an XML footprint file for use with Workbench, to view the results and manage your target platform accordingly.

Step 1 Configure and build a platform project with the footprint feature template.

a) Create a platform project directory and navigate to it if needed.

If you do not already have a platform project to check the footprint on, run the following command on your development host to create a platform project directory and navigate to it

```
$ mkdir -p qemu86-64_glibc-footprint && cd qemu86-64_glibc-footprint
```

b) Configure the platform project.

```
$ configDir/configure \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std \  
--enable-board=qemu86-64 \  
--with-template=feature/footprint
```



NOTE: To configure an existing platform project, add the following configure options to your existing platform project's configure command:

```
--enable-reconfig --with-template=feature/footprint
```

c) Build the file system.

```
$ make fs
```

This command may take some time to complete.

Step 2 Boot your target to run the **fetch-footprint.sh** script.

For a hardware board, refer to your board's recommended instructions. For the QEMU target created in step 1 on page 308, above, run the following command:

```
$ make start-qemu
```

Step 3 Log in to the target.

Once the target boots, enter **root** for the user name and password to log in.

Step 4 Run the **fetch-footprint.sh** script with options on the target.

For a list of available options, see [Footprint \(fetch-footprint.sh\) Command Option Reference](#) on page 311, or run the script without any options for online help. For example:

```
root@qemu0:~# cd /  
root@qemu0:~# /opt/windriver/footprint/fetch-footprint.sh
```

```
fetch-footprint assists in identifying files that were accessed/modified/changed during  
a
```

```

given time interval      Usage: fetch-footprint [OPTION] [FORMAT] <directory>

Options:
[Mandatory - use one]
-a      Display all files
-b      Select files accessed/modified/changed since last boot time
-f      Display files that have future timestamps
[Optional]
-s      Print disk usage statistics
-d      Include directories also
-S      Start date should follow this option <yyyy.mm.dd-HH:MM:SS>
-E      End date should follow this option <yyyy.mm.dd-HH:MM:SS>
       Note: Start Date can not be greater than the end date

Format:
[Optional]
-I      Prints files in a format that fetch_host_footprint understands [Execute
from the target's /]
-R      Displays accessed, modified & changed timestamps
-X      Displays the data in an XML format
-T      Run in Busybox mode [Display all timestamps in seconds from Epoch]

Usage Scenarios:
  Display timestamps of files ONLY                `fetch-footprint.sh -a`
  Display timestamps of files & directories        `fetch-footprint.sh -a -d`
  Display Files AND Directories inside            `fetch-footprint.sh -b -d ../
tempDir/ -R`
  ../tempDir/ directory that were accessed
  /modified/changed since boottime
  List all files in the input mode                `fetch-footprint.sh -a -d ../..
-s -I
  [Execute this from target's /]
  Display files between timestamps                `fetch-footprint.sh -a -S
2009.05.29-21:49:01 -E 2009.05.30-02:50:01 -s`

(c) WindRiver Systems

```



NOTE: When you run the script, you may see more files than you expect in the output—for example, running daemons may touch files which will then appear in command output.

Step 5 Create an XML file for Workbench to use to help you view and managed touched files.

This step is optional.

a) Create XML output.

To produce XML output, use the **-X** option. To produce output on files touched since boot time, use the **-b** argument. For example, the following command creates a XML file named `touchfile.xml` that includes all files touched since boot time:

```
root@qemu0:/# /opt/windriver/footprint/fetch-footprint.sh -b -X > touchfile.xml
```

Once created, you can use Workbench to view the **touchfile.xml** file and list of all files "touched" while the target boots. Use this information to determine which files are not being used on the target.

b) Copy the output to your development host where you built the runtime.

```
$ scp root@targetIP:/touchfile.xml /tmp
```

Alternately, when using the NFS mounted file system, (which is the default for simulators), the file will be found locally, in the NFS root, typically: **projectDir/export/dist/touchfile.xml**

- c) Launch Workbench.

If you created the platform project using Workbench, go to the next step. If you created the platform project from the command line, you must import it to Workbench.

To import the platform project, right-click in the Project Explorer view and select **File > Import > Wind River Linux > Existing Wind River Linux Platform Project into Workspace** and click **Next**.

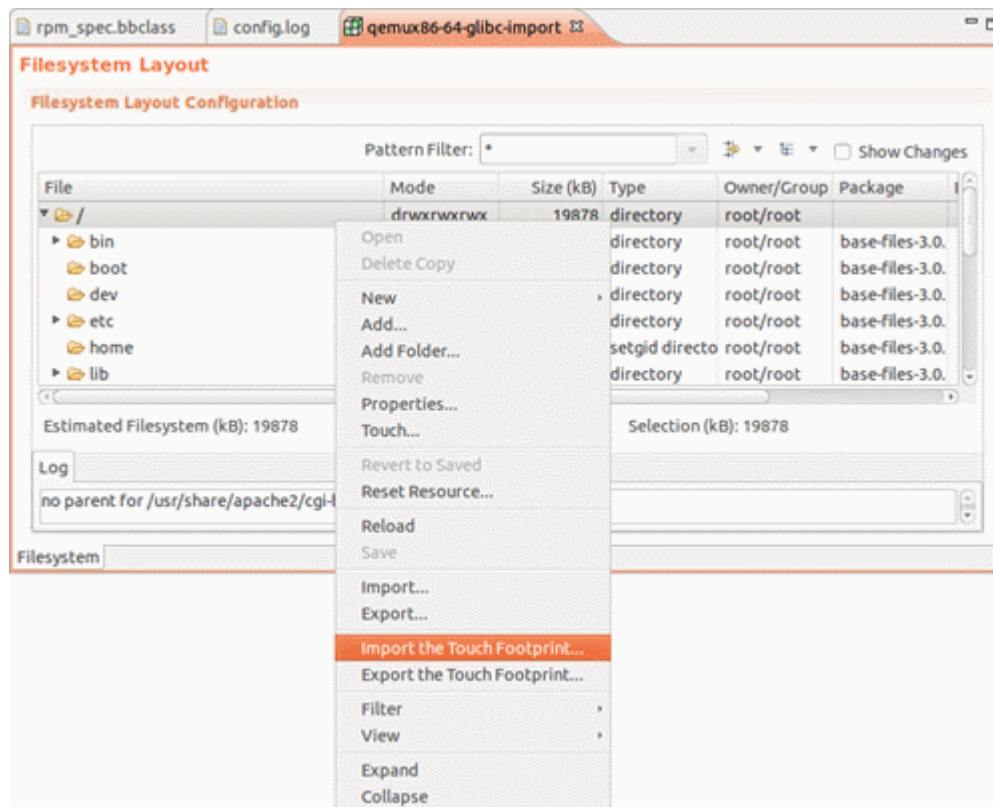
Select the directory for the platform project (where you created the footprint's XML file) and click **OK**.

- d) Expand the platform project in Project Explorer and double-click on **User Space Configuration**.


If prompted, click **OK** to rebuild the package database.

- e) Import the XML file.

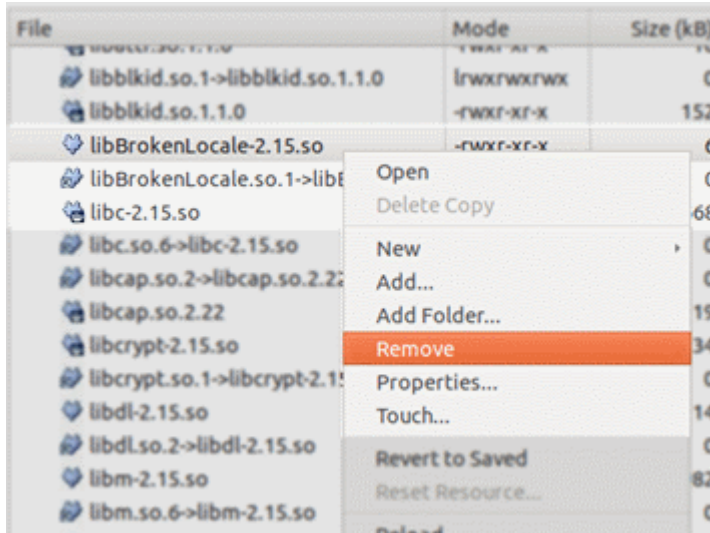
In the **Filesystem** tab, right-click and select **Import the Touch Footprint**. Browse to the XML file you copied to your development host in step 5.b on page 309 and click **OK**.



- f) View touched files in the Filesystem Layout view.

The files that were accessed since the last boot (and during any subsequent coverage testing you might perform), now have a hand icon  decorator indicating that they were accessed.

The remaining files in the system may be removed to reduce the footprint of your file system. For example, in the **lib** directory, you might decide after viewing the touch, like the one below, that you will need **libc-2.15.so**, but not **libBrokenLocale-2.15.so**. Right click the file you wish to remove, and select **Remove**.




Perform this action for any and all files you wish to remove, to help reduce your platform project's footprint.

Footprint (fetch-footprint.sh) Command Option Reference

Use this command reference to learn about the available options for running **fetch-footprint.sh** to obtain platform project footprint data. Run the script without any options to view online help.

Table 9 Options for Running **fetch-footprint.sh**

Command	Option	Description
Mandatory Options- Use only one mandatory option when you run the command		
fetch-footprint.sh	-a	Display all files
	-b	Select files accessed/modified/changed since last boot time
	-f	Display files that have future timestamps
Optional Command Options - Not required to run the command, these options let you specify report information and output		
fetch-footprint.sh	-s	Print disk usage statistics
	-d	Include directories
	-S <i>yyyy.mm.dd-HH:MM:SS</i>	Use to specify the start date of the files to begin gathering footprint data on.
	-E <i>yyyy.mm.dd-HH:MM:SS</i>	Use to specify the end date of the files to gather footprint data on.

Command	Option	Description
		 NOTE: The specified start date must occur before the end date.
	-I	This command is executed from the target's file system. Prints data in a format that the fetch_host_footprint understands.
	-R	Displays accessed, modified and changed timestamps
	-X	Displays the data in XML format
	-T	Runs in BusyBox mode to display all timestamps in seconds from Epoch

32

Reducing the Footprint

[About BusyBox 313](#)

[Configuring a Platform Project Image to Use BusyBox 313](#)

[About devshell 314](#)

[About Static Linking 314](#)

[About the Library Optimization Option 315](#)

About BusyBox

You can use BusyBox to reduce the footprint of your platform project image.

BusyBox merges tiny versions of standard Linux utilities into a single small executable. These utilities include a shell, compression utilities, a DHCP server, login utilities, archiving utilities like **tar** and **rpm**, core utilities like **cat**, **df** and **ls**, networking utilities like **ping** and **tftp**, system administration utilities like **mount** and **more**, and process utilities like **free**, **ps**, and **kill**.

These utilities have reduced functionality compared to their standard Linux counterparts, but they also have a much smaller footprint, and merging them into a single executable results in a smaller footprint still.

Configuring a Platform Project Image to Use BusyBox

Add BusyBox to your platform project image to reduce its footprint.

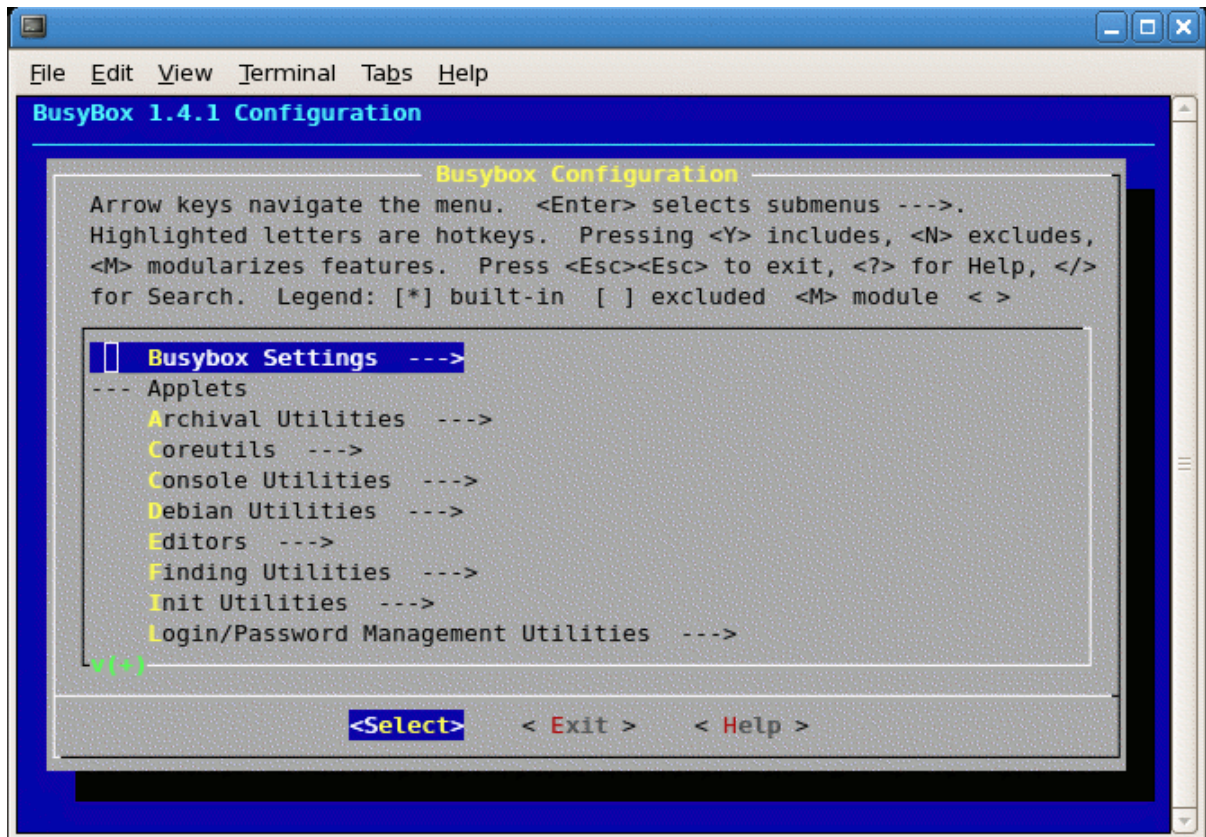
You may add or remove commands supported by the BusyBox executable in much the same way as you configure the Linux kernel. By removing commands you do not intend to use, you reduce the size of the executable even further.

Step 1 Open the BusyBox Configuration tool.

In the ***projectDir/build*** directory, enter the following command:

```
$ make -C build busybox.menuconfig
```

The configuration utility displays:



Step 2 Make changes to your configuration as necessary.

The BusyBox **menuconfig** program functions in exactly the same way as the kernel **menuconfig**. You can access help for each command, and discard or save your changes.

Step 3 Rebuild BusyBox.

```
$ make -C build busybox
```

About devshell

The **devshell** is a terminal shell that runs in the same context as the BitBake task engine.

You can run **devshell** directly, or it may spawn automatically, depending on your development activity. The **devshell** is automatically included when you configure and build a platform project.

For additional information on using **devshell**, see: *The Yocto Project Reference Manual: Development Within a Development Shell*.

About Static Linking

Static linking can improve the performance and portability of your code by including all library files required by a package or application into a single executable module.

Library references are more efficient because the library procedures are statically linked to the program, instead of dynamically linked to library or libraries located elsewhere within the system. Unlike dynamic linking, the overhead required to manage links from applications and packages to their library sources and any network-related issues is not required.

Because the library files are no longer shared by other applications or packages, using static linking can increase file size and project footprint, but offers significant gains by simplifying development and deployment. If your project uses a few core applications, then static linking can save resources and simplify development and portability.

See <http://www.yoctoproject.org/docs/current/dev-manual/dev-manual.html#including-static-library-files> for details on enabling static linking with your platform project image.

About the Library Optimization Option

Use this feature to optimize **Glibc** and **libm** by removing library functions not required by the applications installed into the run-time file system.

Library optimization works only with `glibc_small` file systems. Library optimization also rebuilds libraries to relink them with only the object files necessary for chosen applications.

You enable library optimization with the **enable-scalable=mklibs** option in the platform project **configure** command. See *Configure Options Reference* on page 66.

Like static linking (see *About Static Linking* on page 314), library optimization offers the greatest savings in run-time file system size when very few applications are included. Although library optimization may not offer as great initial savings in size as static linking, the savings should not tail off quite so rapidly as more applications are added. As with static linking, the savings realized will depend on the run-time file system.

33

Analyzing and Optimizing Boot Time

[Analyzing and Optimizing Boot Time](#) 317

[Creating a Project to Collect Boot Time Data](#) 318

[Analyzing Early Boot Time](#) 319

[About Reducing Early Boot Time](#) 321

Analyzing and Optimizing Boot Time

Understanding the factors that impact your platform project image's boot process can help you make decisions to improve and optimize it.

There are two distinct phases of system boot time:

Early boot time

The time from when the kernel is launched to the time the init process (usually `/sbin/init`) is launched

Late boot time

The time from when the init process is launched until the last start-up script is executed.

For Wind River Linux 5, this manual focuses on early boot time analysis and optimization.

The **bootlogger** script collects data on both of these phases of boot time. The script uses the Linux kernel's `ftrace` feature to capture profiling data from the Wind River Linux boot sequence.

The **bootlogger** script overrides the regular `/sbin/init` as the first process and copies the early boot time data in `/debug/tracing/trace` to `/home/root/kernel-init.log` and then configures `ftrace` to trace init processes. When the final init process is executed (`/etc/rcS.d/S999stop-bootlogger`), **bootlogger** copies the late boot time data to `/var/log/post-kernel-init.log`. The names and locations of these files are configurable in the `/export/dist/sbin/bootlogger/bootlogger.conf` file of the target (`projectDir/export/dist/sbin/bootlogger/bootlogger.conf`). As a final step, **bootlogger** launches the regular init process.



NOTE: The **bootlogger** script is designed to be used in development and is not intended to be deployed in production systems.

Creating a Project to Collect Boot Time Data

Before you can collect boot-time data with **bootlogger**, you must configure a platform project to include the feature.

To collect boot time data with **bootlogger**, do the following:

Step 1 Configure and build your platform project for boot logging.

- a) Create a project directory and navigate to it.
- b) Run the configure command.

To configure your platform project for boot logging, specify the feature/boottime template, for example

```
$ configDir/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_small \  
--enable-jobs=4 \  
--enable-parallel-pkgbuilds=4 \  
--with-template=feature/boottime
```

- c) Build the project by entering the following in the project directory:

```
$ make
```

When you build your project, you will have an **/sbin/bootlogger** script, an **/etc/bootlogger.conf** configuration file, and a **stop-bootlogger** script configured as the last init script to run.

Step 2 Configure your boot sequence to use bootlogger.

Configure your kernel boot command line to pass **initcall_debug ftrace=initcall init=/sbin/bootlogger**. This is typically done by passing commands to the bootloader, or as a compilable kernel option.

If you are using QEMU to emulate your target, you can enter `make config-target` and then replace the current options or insert the new **bootlogger** options before the current options for **TARGET0_QEMU_KERNEL_OPTS**.

For example, add the options **initcall_debug ftrace=initcall init=/sbin/bootlogger** as follows:

```
$ make config-target  
....  
53: TARGET0_QEMU_KERNEL=bzImage  
54: TARGET0_QEMU_CPU=  
55: TARGET0_QEMU_INITRD=  
56: TARGET0_VIRT_DISK=  
57: TARGET0_VIRT_DISK_UNIT=  
58: TARGET0_VIRT_CDROM=  
59: TARGET0_VIRT_ROOT_MOUNT=rw  
60: TARGET0_QEMU_BOOT_DEVICE=  
61: TARGET0_QEMU_KERNEL_OPTS=clock=pit oprofile.timer=1  
62: TARGET0_VIRT_UMA_START=yes  
63: TARGET0_QEMU_OPTS=  
64: TARGET0_VIRT_EXT_WINDOW=no  
65: TARGET0_VIRT_EXT_CON_CMD=xterm -T Virtual-WRLinux -e  
66: TARGET0_VIRT_CONSOLE_SLEEP=5
```

```
67: TARGET0_QEMU_HOSTNAME=
68: TARGET0_VIRT_DEBUG_WAIT=no
69: TARGET0_VIRT_DEBUG_TIMEOUT_DEFAULT=40
Enter number to change (q quit)(s save): 61

New Value:initcall_debug ftrace=initcall init=/sbin/bootlogger other-options
Enter number to change (q quit)(s save): s
Enter number to change (q quit)(s save): q
```

Step 3 Boot the target to collect the data.

Boot your target or emulation. When it has finished the complete boot sequence there will be boot logs for both the early and late phases of the boot process in **/root/** on the target.

Once you have collected the data, you will want to analyze it. The following sections describe how to use the data collected by **ftrace** and **bootlogger**:

- [Analyzing Early Boot Time](#) on page 319
- [About Reducing Early Boot Time](#) on page 321

Analyzing Early Boot Time

Use the **analyze-boottime** command to analyze early boot time data.

A platform project, configured and built with the boot time parameters set as described in [Creating a Project to Collect Boot Time Data](#) on page 318.

Perform the following steps to analyze early boot-time data:

Step 1 Boot the target.

Step 2 Copy **/home/root/kernel-init.log**.

Copy the log from the target to your development host for analysis or, if you are using QEMU, you can analyze **export/dist/var/bootlog/kernel-init.log** on the host.

Step 3 Run the **analyze-boottime** command

Run **analyze-boottime** on the early boot log data. (For command-line help, run the command without arguments.) For example, to display results on the console (**-c**), sorted (**-s**) by the time they take, and identify (**-i**) a log file, enter:

```
$ host-cross/usr/bin/analyze-boottime -c -s -i \
export/dist/home/root/kernel-init.log

Time Delta      Context Switch      Function Name
-----
0.007607
0.007950
0.009109
0.009660
0.009781
raid_init
...
0.171444      *
0.223721
0.266659      *
0.275421      *
0.292454
1.384742      *
1.597498      *
ip_auto_config

Total      : 325 Functions
```

```
Boottime: 13.434060 sec
```

➔ **NOTE:** If you are using QEMU and run **analyze-boottime** in your *projectDir*, you do not have to enter the **-i path_to/kernel-init.log** portion of the command.

With **analyze-boottime** you can find which functions are taking most of the time:

Time Delta

This column provides the time each function takes. For example, in the output shown, **ip_auto_config** takes the most time.

Context Switch

This column indicates where the kernel takes control of a function, performs its task, and returns control to the function.

Function Name

This column provides the name of the kernel function. Totals are summarized at the bottom. The example shown is for a QEMU boot.

Refer to the following sections for information on optimizing your project based on the two most time-consuming aspects of the example output:

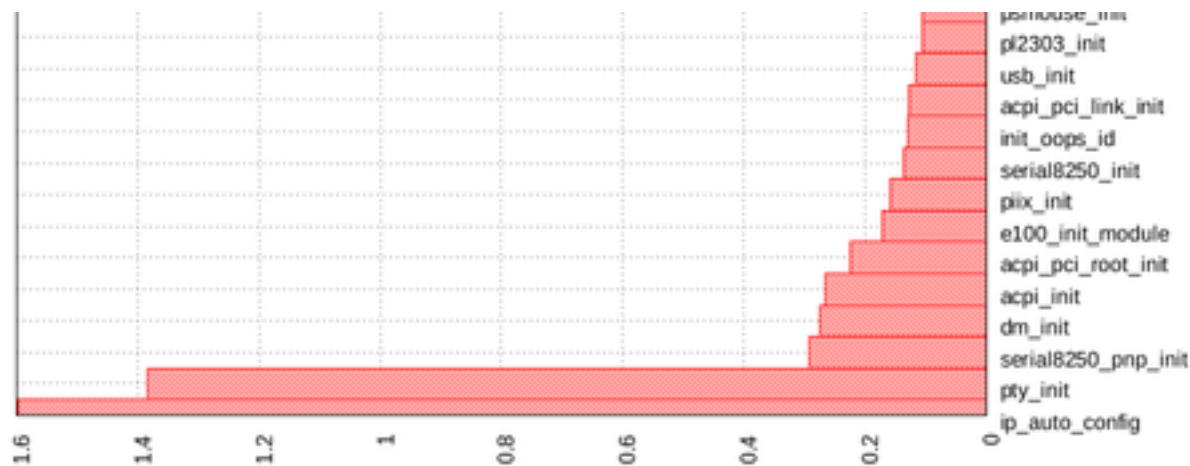
- **ip_auto_config**—see [Reducing Network Initialization Time with Sleep Statements](#) on page 321
- **pty_init**—see [Reducing Device Initialization Time](#) on page 323

Step 4 Review the results of the **analyze-boottime** command.

To view the results graphically, run the command with following options:

```
$ host-cross/usr/bin/analyze-boottime -g early_bootgraph.png
```

This produces a ping file, a portion of which is provided for example:



➔ **NOTE:** The **analyze-boottime** command uses **gnuplot**, which requires the *GDFONTPATH* and *GNUPLOT_DEFAULT_GDFONT* environment variables to be set appropriately.

Step 5 Optionally set the environment variables for **gnuplot** for bash shell users.

Run the following commands in the *projectDir*, or add them to your *.bashrc* file:

```
$ export GDFONTPATH=/usr/share/fonts/liberation
$ export GNUPLOT_DEFAULT_GDFONT=LiberationSans-Regular
```

About Reducing Early Boot Time

Once you've analyzed your platform project image's boot-time, the next step is to reduce it.

The process of reducing early boot time is an iterative one. It includes identifying where time can be reduced on a kernel function, making modifications, examining the results, and then repeating the process for some new function.

Where you can reclaim time from particular functions depends on the requirements of your particular end-system. For an example, the instructions here refer to analyzing a single user system with known hardware and a fixed IP address. The results for a multi-user system with more complex networking requirements will be different, and also pose different challenges to overcome.

Reducing early boot time is accomplished by analyzing and optimizing the following aspects:

- Reducing network initialization time

In the example output from *Analyzing Early Boot Time* on page 319, it shows that the largest single amount of time was spent in the `ip_auto_config()` function. One way to reduce this time is to assign a static IP address to the device so that it does not spend time in DHCP negotiation.

You can assign a static IP address in QEMU with `make config-target` and setting the `TARGET_VIRT_IP` setting to, for example, 10.0.2.15.

With known hardware that does not require time to stabilize, it is possible to remove sleeps from the startup time. To do so, remove sleep statements in `build/linux-windriver-3.4-r0/linux/net/ipv4/ipconfig.c`. See *Reducing Network Initialization Time with Sleep Statements* on page 321.

- Reducing device initialization time
- Removing unnecessary device initialization times

Reducing Network Initialization Time with Sleep Statements

Once you have analyzed your boot-time, you can improve network boot time using the information in this section.

The steps in this section require the following prerequisites:

- A platform project image has been configured and built with the `feature/boottime` template—see *Creating a Project to Collect Boot Time Data* on page 318
- You have run the `analyze-boottime` command to establish a baseline for how long your network initialization (`ip_auto_config`) is taking—see *Analyzing Early Boot Time* on page 319.

Once you have established a baseline, perform the following steps to set a fixed IP address for your QEMU session to help reduce network initialization time:

Step 1 Extract the kernel source:

```
$ make -C build linux-windriver.config
```

Step 2 Edit `build/linux-windriver/linux/net/ipv4/ipconfig.c` and comment out the sleep states:

Find the three lines with sleeps:

```
msleep(CONF_PRE_OPEN);  
...  
ssleep(1);  
...  
ssleep(CONF_POST_OPEN);
```

and change them to:

```
//msleep(CONF_PRE_OPEN);  
...  
//ssleep(1);  
...  
//ssleep(CONF_POST_OPEN);
```

Step 3 Rebuild the kernel and file system:

```
$ make -C build linux-windriver.rebuild; make all
```

Step 4 Reboot.

Step 5 Analyze the new boot logs as you did the first time in [Analyzing Early Boot Time](#) on page 319.

For example, look at the difference between the first QEMU early boot, and the second which used a fixed IP address and with sleep states removed:

```
$ host-cross/usr/bin/analyze-boottime -c -s \  
-i /tmp/initial/kernel-init.log |tail  
0.223721          acpi_pci_root_init  
0.266659          *          acpi_init  
0.275421          *          dm_init  

```

Note the significant improvement in total boot time, as well as the fact that `ip_auto_config` is now not taking the most time. Time to look at the next function that takes the most time, `pty_init`.

Reducing Device Initialization Time

Once you have analyzed your boot-time, you can improve device-related initialization times using the information in this section.

The steps in this section require the following prerequisites:

- A platform project image has been configured and built with the **feature/boottime** template—see [Creating a Project to Collect Boot Time Data](#) on page 318.
- You have run `analyze-boottime` command to obtain metrics on your device initialization time—see [Analyzing Early Boot Time](#) on page 319

In the following procedure, we are going to help you reduce the **pty_init** time. This initialization includes busywaits, locks and mutexes for PTY initialization that can be greatly reduced in this single-user system.

In this example, you will learn to modify a kernel option as follows:

Step 1 Change the value of the **kernel config** option to 5.

Use the Workbench Kernel Configuration tool, a command-line tool such as **make -C build linux-windriver.menuconfig**, or edit **build/linux-windriver-3.4-r0/linux-qemux86-64-standard-build/.config** directly and change the value of the **kernel config** option **CONFIG_LEGACY_PTY_COUNT** from 256 to 5.

Step 2 Rebuild the kernel and file system:

```
$ make -C build linux-windriver.rebuild; make
```

Step 3 Reboot the target.

Step 4 Analyze the new boot logs as you did the first time in [Analyzing Early Boot Time](#) on page 319.

For example, here is a difference between the **pty_init** time for two QEMU early boots with the **CONFIG_LEGACY_PTY_COUNT** set from 256 to 5:

```
$ host-cross/usr/bin/analyze-boottime -c -i \  
/tmp/w_fixedIP_no_sleeps/kernel-init.log|grep pty_init  
0.397953 pty_init  
  
$ host-cross/usr/bin/analyze-boottime -c -i \  
/tmp/w_pty_5/kernel-init.log|grep pty_init  
0.018961 pty_init
```

Note the significant reduction in function time, from 0.397953 to 0.018961.

Removing Unnecessary Device Initialization Times

To optimize your platform project image on an embedded device with known hardware, you may want to remove device initializations for devices that you do not use.

In this example, you remove ATA, MD, and USB device initialization as follows:

Step 1 Edit **quirks.c**.

Edit **build/linux-windriver-3.4-r0/linux/drivers/pci/quirks.c** and comment out the following two lines:

Change:

```
DECLARE_PCI_FIXUP_FINAL(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82441,  
quirk_passive_release);  
DECLARE_PCI_FIXUP_RESUME(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82441,  
quirk_passive_release);
```

to:

```
//DECLARE_PCI_FIXUP_FINAL(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82441,  
quirk_passive_release);  
//DECLARE_PCI_FIXUP_RESUME(PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82441,  
quirk_passive_release);
```

Step 2 Edit kernel configuration options.

Use the Workbench Kernel Configuration tool, a command-line tool such as **make -C build linux-windriver.menuconfig**, or edit **build/linux-windriver-version/.config** directly and turn off the following kernel config options as shown:

```
# CONFIG_ATA is not set  
# CONFIG_MD is not set  
# CONFIG_USB is not set  
# CONFIG_USB_SUPPORT is not set
```

Step 3 Rebuild the kernel and file system:

```
$ make -C build linux-windriver.rebuild; make
```

Step 4 Add some kernel boot arguments.

An example would be turning off power management. For QEMU, run **make config-target** and set the following:

```
TARGET0_QEMU_DEBUG_PORT=0  
TARGET0_QEMU_KERNEL_OPTS="initcall_debug ftrace=initcall \  
init=/sbin/bootlogger quiet acpi=off lpj=11501568"
```

The additional arguments keep all messages from being displayed (**quiet**), turn off the advanced configuration and power interface (**acpi=off**), and presetting the loops per jiffy value (**lpj=11501568**) turns off the loops per jiffy calibration with each boot. If, of course, you want to see the messages, use power management, or determine your loops per jiffy, only turn off the features that you do not need.

Step 5 Reboot the target.

Step 6 Analyze the new boot logs as you did the first time in [Analyzing Early Boot Time](#) on page 319.

For example, here is the difference in total boot time showing the improvement given by this last set of optimizations versus the initial boot:

```
$ host-cross/usr/bin/analyze-boottime -c -i /tmp/initial/kernel-init.log \  
|tail -3  
Total : 325 Functions  
Boottime: 13.434060 sec  
  
$ host-cross/usr/bin/analyze-boottime -c -i /tmp/lessdevs/kernel-init.log \  
|tail -3  
Total : 301 Functions  
Boottime: 2.751994 sec
```

Note the reduction in total time as well as total number of kernel functions.

PART X

Target-based Networking

About Target-based Networking.....	329
Setting Target and Server Host Names.....	331
Connecting a Board.....	333

About Target-based Networking

When you deploy Wind River Linux on a networked board, the boot loader on the board can get a kernel and file system from the network, providing you have a properly configured boot loader and network server setup.

For this to work properly, you must configure your network server(s) to supply the kernel and file system to your board through its network connection. The steps throughout this section assume that you have previously built a file system and have either built a kernel or are using the default kernel provided when you built your platform project.

The Network Boot Process

If you are booting your target board over the network you will typically use the following resources in this order:

1. a bootloader—this is software on the board that you configure to access the network appropriately.
2. an IP configuration—you can configure an IP network address into your bootloader, or you may get your IP address from the network.
3. a kernel to boot—a network server provides a kernel for download.
4. a root file system to mount—the downloaded kernel mounts the root file system from the network.

Board-specific details for the boot loaders are provided in the **README** files in your *projectDir/READMEs* directory. See [README Files in the Development Environment](#) on page 39 for instructions on making these **README** files visible in your platform project.



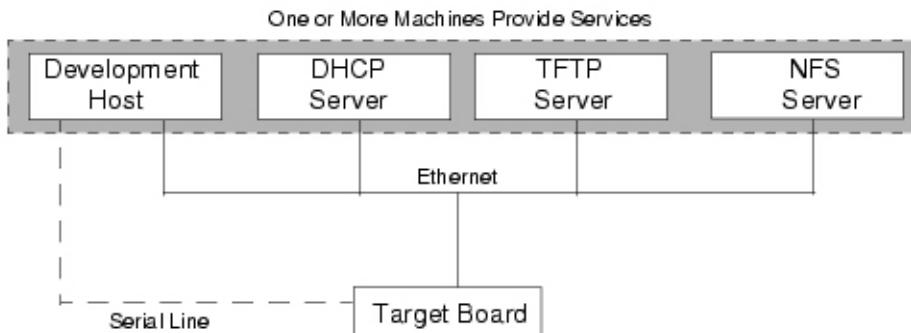
NOTE: Boot loader and network configuration are somewhat different for boards that use the PXE boot protocol. Refer to [About Configuring PXE](#) on page 334 for details on network booting with PXE.

Network Servers During Boot

The typical network deployment boot process uses network servers as follows:

1. The boot loader on the board gets its IP address, either locally or from the network. If from the network, a DHCP server supplies the IP address.
2. With its IP address, the boot loader connects to a TFTP server and downloads a compressed kernel file.
3. The boot loader uncompresses and boots the kernel, which takes control and then mounts its root file system from an NFS server on the network.

The following image depicts these servers in an networked, embedded development environment. Note that the DHCP, TFTP, and NFS servers that you must configure may physically reside on one machine, or may be on different machines depending on your network environment.



Network Configuration on a Different Host

Different network servers provide different GUI and command-line tools for network service configuration. Configuration file specifics may also vary. The information and referenced topics in this manual can only make suggestions on how to configure the different services—refer to your server documentation for specifics on your host and services.



NOTE: You will typically need root (superuser) privileges when configuring network services.

35

Setting Target and Server Host Names

You may want to map your target and server IP addresses to host names for ease of reference.

Context for the current task

Step 1 Configure `/etc/hosts` file of your server to include both the host name and IP address of the target and the server

```
192.168.10.1 server1.lab.org server1
192.168.10.2 target7.lab.org target7
```

Step 2 Add this information to the `fs_final*.sh` script on the target.

see [Adding an Application to a Root File System with `fs_final*.sh` Scripts](#) on page 141

The resulting file system will include the hosts file when downloaded from the server.

Step 3 Build the target file system.

```
$ make
```


36

Connecting a Board

[Configuring a Serial Connection to a Board](#) 333

[About Configuring PXE](#) 334

[Configuring PXE](#) 336

[Configuring DHCP](#) 337

[Configuring DHCP for PXE](#) 338

[Configuring NFS](#) 340

[Configuring TFTP](#) 341

Configuring a Serial Connection to a Board

Setting-up cu and UUCP

Configure a **cu** terminal emulator to connect to the board over a serial connection.

This procedure provides access to the board so you can set boot loader parameters.

Step 1 Edit server configuration files.

The settings in these files must match the device name and baud rate of your serial port.

a) Edit the `/etc/uucp/port` file.

For example:

```
port serial0_38400
type direct
device /dev/ttyS0
speed 38400
hardflow false
```

b) Edit the `/etc/uucp/sys` file.

For example:

```
system S0@38400
```

```
port serial0_38400  
time any
```

You can find instructions on the serial port device name and baud rate in the **README** for each board.

Step 2 Test the terminal connection.

For example:

```
# cu S0@38400
```

Step 3 Disconnect the terminal.

Type the escape character (~), followed by a period (.).

Setting up the Workbench Terminal

Configure a Workbench terminal emulator to connect to the board over a serial connection. This procedure provides access to the board so you can set boot loader parameters.

Step 1 Set the port.

Within the Workbench Terminal view, click the Settings icon. Set the port number.

Step 2 Set the baud rate.

You make this change from the same screen as the port number.

Step 3 Save your changes.

About Configuring PXE

You can configure the Pre-boot Execution Environment (PXE) boot loader on most x86 boards with Wind River Linux board support packages (BSPs).

Boot Process Overview

A PXE boot-enabled NIC supports the Bootstrap Protocol (BOOTP). This protocol, provided by a DHCP server, allows a diskless target to obtain its own IP address, the IP address and name of a server, and the name of the boot loader file on that server that it can download to boot.

Booting the target server follows these steps:

1. The PXE-enabled NIC of the target broadcasts its MAC address, requesting an IP address from a BOOTP/DHCP server.
2. The DHCP/BOOTP server, configured with the MAC address of the target and other options, returns the IP address of the target, along with the name of the TFTP server and the name of the PXELinux boot loader file, which resides on the TFTP server.
3. The target downloads, using TFTP, the PXELinux boot loader, which provides the name of the Linux kernel image to load. The PXELinux boot loader downloads the kernel. The target runs the kernel, which rediscovers its IP address from the DHCP server.
4. The DHCP server provides the location for the NFS root file system; the kernel mounts it and completes system initialization.

To use PXE, you must copy the kernel and root file system to their download and export directories.

The default TFTP download directory is **/tftpboot**. In the examples provided here, the NFS export directory for the root file system is

/home/nfs/export

You may configure TFTP and NFS to use the same directory if you prefer.

The PXELinux Boot Loader File

The PXELinux boot loader file is **pxelinux.0**. This file is part of the Syslinux package. Installing Syslinux installs **pxelinux.0** into the **/usr/lib/syslinux** directory.

The PXELinux Configuration File

The PXELinux configuration file resides in the **/tftpboot/pxelinux.cfg** directory. There can be separate configuration files for separate targets. To enable this, a filename convention is used that identifies a configuration file by the hardware type and MAC address of its specific target, or its IP address.

The following example demonstrates how the PXE bootloader searches for the correct configuration file. The example assumes that the PXE bootloader is looking for the configuration file for the **target.lab.org** of the scenario, which has been assigned an IP address of 192.168.10.2, and which has an Ethernet card with a MAC address of 00-20-ED-6E-82-3D.

First, the bootloader will look for a configuration file corresponding to its MAC address, with the first two digits representing its ARP code. This filename, all in lowercase, would be:

01-00-20-ed-6e-82-3d

(Note the **01-** preceding the MAC address.)

If that filename cannot be found in the **/tftpboot/pxelinux.cfg/** directory, the bootloader will search for a file named after its IP address in hexadecimal. The filename for this example, all in uppercase, would be:

C0A80A01

Not finding that, the bootloader will search for files in the following order:

C0A80A0

C0A80A

C0A80

C0A8

C0A

C0

C

Finally, not finding any of these files, it will look for a file named **default**.

Configuring PXE

This procedure illustrates the procedure needed to configure the Pre-boot Execution Environment (PXE) boot loader on most x86 boards with Wind River Linux board support packages (BSPs)

The information referenced here describes a typical development example of bringing up a board using PXE, TFTP, and NFS.

For DHCP and PXE boot, the following components are required:

- DHCP
- NFS
- TFTP
- The Syslinux package, which contains the PXELinux boot loader
- The TFTP and PXELinux packages must be installed.

Step 1 Copy the kernel and root file system to their download and export directories.

The default TFTP download directory is **/tftpboot**. In the examples provided here, the NFS export directory for the root file system is

/home/nfs/export

You may configure TFTP and NFS to use the same directory if you prefer.

Step 2 Copy the PXELinux boot loader file **/usr/lib/syslinux/pxelinux.0** to the TFTP download directory. **/tftpboot** is the default TFTP download directory.

Step 3 Set up the PXELinux configuration file.

a) Create the directory and file.

In this example, the filename **default** is used.

```
# mkdir /tftpboot/pxelinux.cfg
# touch /tftpboot/pxelinux.cfg/default
```

b) Populate the file.

For example:

```
netboot
prompt 1
display
pxeboot.msg
timeout 300
label netboot
kernel bzImage
append ip=dhcp root=/dev/nfs nfsroot=/home/nfs/export
```

bzImage represents the kernel's actual filename. It has been given the label **netboot**, which is also the default kernel to load.

Step 4 Configure the target to use PXE boot.

Setting up the target requires that network boot using PXE is enabled. This is generally done within the BIOS setup routine. Configure the boot parameters and sequence in your BIOS to enable the PXE boot loader and to boot from it first (or only).



NOTE: You may be able to find PXE boot images on the Web, for example at <http://www.rom-omatic.net/>.

When your target boots you should see the target go through the following sequence:

1. broadcast MAC address and receive IP address
2. download PXE Boot Loader and configuration file
3. download bzImage
4. boot bzImage
5. get IP address again
6. mount NFS file system



NOTE: If you cannot get through the first two steps in this sequence, verify your **dhcpd.conf** file settings. If you cannot download the bzImage file, verify that your TFTP server is enabled and **xinetd** has been restarted. If your bzImage boots but cannot mount the file system, verify that the NFS daemon (nfsd) is running and that the targets root file system exists in

/usr/nfs/export

Configuring DHCP

This procedure illustrates how to configure a DHCP server to provide your board with an IP address at boot time.

On the DHCP/BOOTP server, you must configure the **dhcpd.conf** configuration file, and you must create a **dhcpd.leases** file if one does not already exist, as described in this section.

The DHCP configuration file is **/etc/dhcpd.conf**. A sample file is presented below. It provides a basic example of this file for a DHCP server called **server1.lab.org**. The IP address of the server is 192.168.10.1. The configuration file identifies **server1.lab.org** as the TFTP server and the target is assigned a static IP addresses.

In this example the DHCP server is version 3.0.1 from the Internet Software Consortium's (ISC). Refer to the documentation for your DHCP server for specific configuration file settings.

Step 1 Edit the **/etc/dhcpd.conf** file.

It should look similar to the following:

```
authoritative;

ddns-update-style ad-hoc;
default-lease-time 21600;
max-lease-time 21600;
option routers 192.168.10.1;
option subnet-mask 255.255.255.0;
option broadcast-address 192.168.10.255;
option domain-name "lab.org"; # Substitute your domain name
option domain-name-servers 192.168.10.1; # substitute your DNS server address

# Subnet and range of IP addresses for dynamic clients
subnet 192.168.10.0
netmask 255.255.255.0 {
```

```
range 192.168.10.3 192.168.10.40;
}
host server1.lab.org { # substitute your board's host name
  hardware Ethernet XX:XX:XX:XX:XX:XX; # Substitute your board's hardware address
  fixed-address 192.168.10.1;
}
}
```

Notice that the static IP address of the target is within the subnet of the DHCP server, but outside the range of the dynamic IPs.

- a) Provide your board's MAC address as the value for **hardware Ethernet**.
- b) Adjust values in the sample `/etc/dhcpd.conf` to match your environment.

Edits may include values for:

- **domain-name**
- **domain-name-servers**
- the board's **host** name and **fixed-address**

Step 2 Create a `/var/lib/dhcp/dhcpd.leases` file if needed.

- a) Determine if the file exists.

```
ls /var/lib/dhcp/dhcpd.leases
```

- b) Create the file if necessary.

```
touch /var/lib/dhcp/dhcpd.leases
```

Step 3 Start the dhcp server.

The command to start a service varies across Linux distribution. Consult your server's documentation for specifics.

On Red Hat Linux, you would run:

```
$ service dhcpd start
```

Configuring DHCP for PXE

This topic describes the correct configuration changes to support DHCP for PXE

Step 1 Edit the `/etc/dhcpd.conf` file.

- a) Add the PXE boot additions.

```
allow booting;
allow bootp;
```

- b) Set up static IP booting for the target.

```
host target.lab.org {
  hardware ethernet 00:20:ED:6E:82:3D;
  fixed-address 192.168.10.2;
  next-server 192.168.10.1;
  filename "pxelinux.0";
  option root-path "192.168.10.1:/home/nfs/export";
}
```



NOTE: Substitute the correct MAC address for the sample hardware Ethernet address provided.

A correctly configured `/etc/dhcpd.conf` had the following contents:

```
authoritative;

ddns-update-style ad-hoc;
default-lease-time 21600;
max-lease-time 21600;
option routers 192.168.10.1;
option subnet-mask 255.255.255.0;
option broadcast-address 192.168.10.255;
option domain-name "lab.org";
option domain-name-servers 192.168.10.1;

# Next two lines PXE boot additions

allow booting;
allow bootp;

# Subnet and range of IP addresses for dynamic clients
subnet 192.168.10.0
netmask 255.255.255.0 {
range 192.168.10.3 192.168.10.40;
}
  host server1.lab.org {
    hardware Ethernet XX:XX:XX:XX:XX:XX;
    fixed-address 192.168.10.1;
  }

# Next section PXE boot static IPs for the target; an example MAC address
# (hardware ethernet address) is provided.

host target.lab.org {
hardware ethernet 00:20:ED:6E:82:3D; # Replace this address as appropriate
fixed-address 192.168.10.2;
next-server 192.168.10.1;
filename "pxelinux.0";
option root-path "192.168.10.1:/home/nfs/export";
}
```

In this example, `dhcpd.conf` has been configured to support BOOTP, and the PXE target is configured with a static IP address and supplied the following:

fixed address

The address of the PXE server.

filename

Provides the file name of the PXE file in `/tftpboot` to download, `pxelinux.0` in this case .

next-server

The address of the NFS server.

option root-path

Provides the path on the NFS server for the exported PXE files.

Step 2 Restart the DHCP server to apply the changes.

The command to start a service varies across Linux distribution. Consult your server's documentation for specifics.

On Red Hat Linux, you would run:

```
$ service dhcpd start
```

Configuring NFS

This procedure explains how to configure NFS to provide a root file system for your board when the target kernel boots.

Refer to your host documentation for details on installing and enabling NFS if it is not already available.

You will need to have root permission and have created an export directory such as **/nfsroot/**

Step 1 Make the root file system available for export.

Copy and uncompress the compressed run-time file system file to the NFS export directory.

For example, you could use the command line as root as follows:

```
$ su -  
Password:  
# mkdir /nfsroot  
# cd /nfsroot  
# tar -xjvpf projectDir/export/*dist.tar.bz2
```

Step 2 Configure the **/etc/exports** file.

The NFS configuration file is a plain-text file, **/etc/exports**. You must configure it to export the run-time file system to the target.

For example, if your target had the IP address of 192.168.10.2, the **/etc/exports** file might appear as shown in the following example.

```
/nfsroot 192.168.10.2/255.255.255.0(rw,sync,no_subtree_check,no_root_squash)
```

This makes **/nfsroot** available for mounting to the machine with network address 192.168.10.2 only.

Step 3 Restart the server.

```
# exportfs -ra
```

Step 4 Restart NFS.

The command to start a service varies across Linux distribution. Consult your server's documentation for specifics.

On Red Hat Linux, you would run:

```
$ service nfs start
```

Configuring TFTP

This procedure illustrates how to provide your board with a kernel at boot time by configuring a TFTP server on your network. When the board boots, it downloads the kernel.

Context for the current task

Step 1 Make the kernel available for download.

The default TFTP download directory is typically **/tftpboot**. If a download directory for TFTP is not already created, you must create it. Refer to your server documentation for the name of your TFTP download directory and for instructions if you want to change the default.

For example, assuming a TFTP download directory of **/tftpboot** you could copy the kernel to the TFTP download directory as follows:

```
# cd projectDir/export
# cp -L *uImage* /tftpboot/uImage
```

This copies the kernel from your export directory to the file with the shorter name (for convenience) of **uimage** in the TFTP download directory. The **-L** option covers both a prebuilt kernel and scenario a symlink to a kernel you have explicitly built.

Step 2 Enable TFTP for **xinetd**.

For many Linux systems, the TFTP server is automatically started upon request with **inetd** or **xinetd**. The following provides some general instructions for enabling the TFTP server with **xinetd**. Refer to your system documentation for details on how to enable TFTP.

Edit the file **/etc/xinetd.d/tftp** and change the setting **disable=yes** to **disable=no**.

Alternately, you can avoid manual editing by using the **setup** program at the command line to enable the service.

Step 3 Restart **xinetd**.

The command to restart a service varies across Linux distribution. Consult your server's documentation for specifics.

On Red Hat Linux, you would run:

```
$ service xinitd restart
```

PART XI

Reference

Additional Documentation and Resources.....	345
Common make Command Target Reference.....	351
Build Variables.....	361
Package Variable Listing.....	367
Lua Scripting in Spec Files.....	369
Kernel Audit Directory Contents.....	371

37

Additional Documentation and Resources

[Document Conventions](#) 345

[Wind River Linux Documentation](#) 346

[Additional Resources](#) 347

[Open Source Documentation](#) 348

[External Documentation](#) 350

Document Conventions

Use this information to understand the formatting used throughout Wind River Linux documentation.

In this document, placeholders that you must substitute a value for 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 Wind River Linux. By convention, this is typically **C:\WindRiver** on Windows hosts and **/opt/WindRiver** on Linux hosts.

The placeholder *projectDir* refers to the project directory in which much of your work takes place. For example, if you maintain your project files in **/home/user/workspace/qemux86-64** (**qemux86-64_prj** in Workbench), this manual uses *projectDir* to represent that file location.

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 also shown in **bold** and system output is shown in typewriter text, for example:

```
$ pwd
/home/mary/Builds/qemux86-64_prj
$
```

Long command lines that would normally wrap are shown using the backslash (\) followed by ENTER, which produces a secondary prompt, at which you may continue typing. The secondary prompts are not shown to make it easier to cut and paste from the examples.

In the following example you would enter everything literally except the \$ prompt:

```
$  
  
$ /opt/WindRiver/wrlinux-5/wrlinux/configure \  
--enable-board=qemux86-64 \  
--enable-kernel=standard \  
--enable-rootfs=glibc_std
```

If a command requires root privileges to run, the prompt is displayed as #.

Wind River Linux Documentation

Understanding the full range of documentation available is invaluable to helping you find the information you need to work effectively in Wind River Linux.

The following is a list of the documentation provided by Wind River that supports development of Linux targets. Much of this documentation is available through the “start” menu of the installation host, for example under **Applications > Wind River > Documentation** in the Gnome desktop for Linux.

Product Installation and Licensing Guides

For information on installing Wind River Linux and other products from Wind River, refer to the online product installation and licensing guides at <http://www.windriver.com/licensing/documents/>

Wind River Linux User's Guide (this document)

This guide describes Wind River Linux—how to configure it, and customize it for your needs. It is primarily oriented toward command line usage, but it is also useful to Workbench developers who want to understand some of the underlying design and implementation of the build system. It provides both explanatory and procedural use case material.

Wind River Linux Getting Started

The Getting Started guide provides a few brief procedures that you can perform on the command line or with Workbench. Its primary purpose is to orient you in the primary ways to use Wind River Linux and point to the documentation areas that focus most on the way you will be using the product.

Wind River Linux Migration Guide

This guide provides background information, instructions, and procedures for migrating Wind River Linux 4.x projects to Wind River Linux 5.0.1.

Wind River Workbench User's Guide

This guide describes how to use Workbench to develop projects, manage targets, and edit, compile, and debug code.

Wind River Workbench by Example, Linux Version

This guide is for Linux-specific use of Workbench, and provides examples on how to configure and build application, platform, and kernel module projects

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 [Document Conventions](#) on page 345 for details on the help key.

Wind River Workbench User Interface Reference

This guide describes the Workbench user interface for all supported Wind River products. Look here for specific information such as interface dialogs, fields and their uses.

Wind River Analysis Tools documentation

This is a set of documents that describe how to use the Wind River Analysis Tools that are provided with Workbench. The tools include a memory use analyzer, an execution profiler, and System Viewer, a logic analyzer for visualizing and troubleshooting complex embedded software. The Wind River System Viewer API Reference is also included.

Wind River Workbench Host Shell User's Guide

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

Most of the documentation is available online as PDFs or HTML accessible through Wind River Workbench online help. Links to the PDF files are available by selecting **Wind River > Documentation** from your operating system start menu.

From a Workbench installation you can view the documentation in a Web browser locally (**Help > Help Contents**) or by launching the help browser independently of Workbench with the **wrhelp.sh** script as described in Wind River Workbench, By Example, Linux Version.

The documentation is also available below your installation directory (called *installDir*) through the command line as follows:

- PDF Versions—To access the PDF, point your PDF reader to the *.pdf file, for example:
`installDir/docs/extensions/eclipse/plugins/com.windriver.ide.doc.wr_linux_version/wr_linux_users_guide/wr_linux_users_guide_5.pdf`
- HTML Versions—To access the HTML, point your web browser to the **index.html** file, for example:
`installDir/docs/extensions/eclipse/plugins/com.windriver.ide.doc.wr_linux_version/wr_linux_users_guide/html/index.html`

Additional Resources

Refer to these additional resources to help facilitate your development needs.

Online Support

Wind River Online Support provides updates and enhancements to packages as they become available, which can be downloaded and added to Wind River Linux.

Tutorials designed to illustrate Wind River integration with Workbench, as well as sample configuration files to simplify the target board boot process, are also available.

Developer Web Site

Wind River has a public web site <http://developer.windriver.com> that you can both monitor and contribute to. It contains discussions, documents, and additional information of general use of Wind River Linux.

Workbench Tutorials

Detailed Workbench tutorials are available in the *Wind River Workbench User's Guide* and *Wind River Workbench by Example, Linux Version*.

Open Source Documentation

Use the links and information in this section to learn about Linux open source development.

The main source for information referenced throughout this section is the Linux Documentation Project (<http://tldp.org>). As with all documentation, proprietary or otherwise, open source documentation, while valuable, must always be scrutinized for relevance. It is sometimes written specifically for a certain Linux distribution (which may not always be obvious), and sometimes even for a specific version. It is often out-of-date. It is a good idea to compliment, where possible, the provided resources with resources from vendors, mailing lists, and from the maintainers themselves.

Linux Standards Base

Wind River Linux 5 has been designed to support the Linux Standards Base, allowing you to more easily use applications from other LSB-conforming distributions to Wind River Linux.

See <http://www.linuxfoundation.org/collaborate/workgroups/lsb> for details on the Linux Foundation LSB workgroup and related documentation.

Carrier Grade Linux

The Carrier Grade Linux page on the Linux Foundation website is a repository for articles, white papers and projects devoted to developing Carrier Grade-compliant Linux distributions and applications.

See <http://www.linuxfoundation.org/collaborate/workgroups/cgl>.

Linux Development

Reference Title	Link
<i>Building and Installing Software Packages for Linux</i>	http://www.tldp.org/HOWTO/Software-Building-HOWTO.html
<i>Program Library HOWTO</i>	http://www.tldp.org/HOWTO/Program-Library-HOWTO/index.html
<i>Linux Loadable Kernel Module HOWTO</i>	http://www.tldp.org/HOWTO/Module-HOWTO/index.html
<i>Linux Parallel Processing HOWTO</i>	http://www.tldp.org/HOWTO/Parallel-Processing-HOWTO.html

Reference Title	Link
<i>Secure Programming for Linux HOWTO</i>	http://www.tldp.org/HOWTO/Secure-Programs-HOWTO/index.html
<i>RPM HOWTO</i>	http://www.tldp.org/HOWTO/RPM-HOWTO/index.html
<i>Maximum RPM</i> Additional, useful information on using RPMs	http://www.rpm.org/max-rpm/

Networking

Reference Title	Link
<i>The Linux Networking Overview HOWTO</i>	http://www.tldp.org/HOWTO/Networking-Overview-HOWTO.html
<i>The Linux Networking HOWTO</i>	http://www.tldp.org/HOWTO/NET3-4-HOWTO.html
<i>The PPP HOWTO</i>	http://www.tldp.org/HOWTO/PPP-HOWTO/index.html
<i>ADSL Bandwidth Management HOWTO</i>	http://www.tldp.org/HOWTO/ADSL-Bandwidth-Management-HOWTO/index.html
<i>Traffic Control HOWTO</i>	http://www.tldp.org/HOWTO/Traffic-Control-HOWTO/
<i>Netfilter/Iptables HOWTO</i> This includes a good deal of documentation on packet filtering, NAT, and tutorials.	http://www.netfilter.org/documentation
<i>VPN HOWTO</i>	http://www.tldp.org/HOWTO/VPN-HOWTO/index.html

Security

Reference Title	Link
<i>Netfilter/Iptables HOWTO</i> This includes a good deal of documentation on packet filtering, NAT, and tutorials.	http://www.netfilter.org/documentation
<i>SSL Certificates HOWTO</i>	http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/index.html
<i>OpenSSH</i>	http://www.openssh.com

External Documentation

Use external documentation to enhance your developer knowledge and capabilities. The following table lists open source documentation related to the Yocto Project and Wind River Linux:

Table 10 Wind River Linux External Documentation

External Information source	Location
The Yocto Project	Online: http://www.yoctoproject.org
OpenEmbedded Core (OE-Core)	Online: http://www.openembedded.org/wiki/OpenEmbedded-Core
QEMU	Online: http://wiki.qemu.org
GNU Toolchain Documentation	<p>A full set of GNU compiler-related development documentation is included with every Wind River Linux 5 installation in the following installed locations, based on your target architecture:</p> <p><i>installDir/wrlinux-5/layers/wr-toolchain/4.6a-99/share/doc/wrs-linux-arm-wrs-linux-gnueabi</i> – for ARM-based target systems</p> <p><i>installDir/wrlinux-5/layers/wr-toolchain/4.6a-99/share/doc/wrs-linux-i686-wrs-linux-gnu</i> – for x86-based target systems</p> <p><i>installDir/wrlinux-5/layers/wr-toolchain/4.6a-99/share/doc/wrs-linux-mips-wrs-linux-gnu</i> – for MIPS-based target systems</p> <p><i>installDir/wrlinux-5/layers/wr-toolchain/4.6a-99/share/doc/wrs-linux-powerpc-wrs-linux-gnu</i> – for PowerPC-based target systems</p>

38

Common make Command Target Reference

This section provides a summary of common build arguments for the **make** command from the command-line and with Workbench.

Platform Project Development

make Command in <i>projectDir</i>	Workbench Build Target	Description
make make fs make all	fs	Each of these commands builds a new file system from RPMs where available, use source otherwise. Note that there is no difference and the commands are interchangeable. For information on using make , see About the make Command on page 72.
make build-all	build-all	Performs the same action as make . For information on forcing a source build, see About the make Command on page 72.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make fs-debug	fs-debug	This produces an additional file system image in the <i>projectDir/export</i> directory named similarly to the file system image but with -debuginfo.tar.bz2 at the end. This additional image contains only debug information and source. This file can be used for cross-debugging with Workbench, or <i>gdb-server</i> , or alternately deployed on with the file system for on-target debug with <i>gdb</i> . This build target is only supported with production builds; all other build types include debug information in the default file system image.
make help		View command-line help information for the make command
	delete	Remove the <i>project_prj</i> contents and folder.
make reconfig	reconfig	Re-process templates and layers. Recreates list files and makefiles but does not support changes to <i>config.sh</i> (which require a new configuration). Once run, this command locks the platform project to the latest product update (RCPL) available and saves the release number to the <i>projectDir/config.log</i> file as a reference if you need to recreate the project at a later date.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make upgrade		<p>Upgrades the platform project build to the latest product update. Each time you update Wind River Linux, run this command in the <i>projectDir</i> to ensure your platform project is built with the latest available features.</p> <p>Once this command is run, the <i>projectDir/config.log</i> file will indicate the RCPL (product update) in use, and automatically select the latest RCPL with the highest number.</p>
make -C build busybox.menuconfig		<p>Menu-based tool to configure busybox</p> <p>This is the equivalent of bitbake -c menuconfig busybox within the BitBake environment.</p>
make export-dist		<p>Configures the <i>projectDir/export/dist</i> directory and builds the file system when necessary.</p>
make host-tools		<p>Builds all the native sstate packages required to build glibc-small, core, std and std-sato file systems and saves them to an exportable <i>projectDir/export/host-tools.tar.bz2</i> archive.</p>


make Command in <i>projectDir</i>	Workbench Build Target	Description
make bbs		<p>Sets up the BitBake environment, such as the variables required, before you can run BitBake commands.</p> <p>This command executes a new shell environment and configures the environment settings, including the working directory and <i>PATH</i>.</p> <p>To return to the previous environment, simply type exit to close the shell.</p> <hr/> <p> NOTE: This command is the equivalent of source <code>layers/oe-core/oe-init-buildenv</code> bitbake_build.</p>

Image Deployment

make Command in <i>projectDir</i>	Workbench Build Target	Description
make start-qemu		Start a QEMU simulation. Use make start-target TOPTS="--help" for a list of options.
make start-target		Start a QEMU simulation. Use make start-target TOPTS="--help" for a list of options.
make usb-image		<p>Use to create a bootable USB image from any existing platform project image. The image includes two partitions:</p> <p>16 FAT</p> <p>The first is a small 16 FAT file system for syslinux, the kernel, and a static BusyBox initrd</p> <p>ext2</p> <p>The second is an ext2 file system to mount the root partition for the operating system</p>

make Command in <i>projectDir</i>	Workbench Build Target	Description
make usb-image-burn		Use to create a bootable USB image from any existing platform project image, and burn the image directly to a USB drive.

Application Deveopment

make Command in <i>projectDir</i>	Workbench Build Target	Description
make export-sdk	export-sdk	<p>Creates a SDK suitable for application development in the export/ directory, which can be used for providing build specs in Workbench.</p> <p>This option first builds the root file system, then populates the SDK.</p> <p>This includes the sysroot and toolchain, and is the preferred method to use to set up the environment for application development.</p>
make export-toolchain	export-toolchain	Performs the same tasks as make export-sdk .
make sysroot		<p>Forces the sysroot population in the <i>projectDir</i>.</p> <p>If the contents of your platform project build originates from the sstate-cache, the system knows that the sysroot is not necessary for any activities and never populates it. Run make sysroot to populate the sysroot if you require it.</p>
make export-sysroot	export-sysroot	<p>This performs the same action as make export-sdk above, but does not export the toolchain.</p> <p>This is designed to simplify application development overhead, for when updates occur for the sysroot, but not the (generally unchanging) toolchain.</p>

make Command in <i>projectDir</i>	Workbench Build Target	Description
make host-tools		Builds all the native sstate packages required to build glibc-small, core, std and std-sato filesystems and saves them to an exportable <i>projectDir/export/host-tools.tar.bz2</i> archive.
make populate-sdk		Same as make sysroot , above.
make populate-sysroot		Same as make export-sysroot , above. This is the equivalent of bitbake -c populate_sdk <i>imageName</i> within the BitBake environment.

Package and Recipe Management

make Command in <i>projectDir</i>	Workbench Build Target	Description
make -C build <i>recipeName</i>		Build the recipe <i>recipeName</i> This is the equivalent of bitbake <i>recipeName</i> within the BitBake environment.
make -C build <i>recipeName.addpkg</i>		Add a recipe's package and any packages it is known to require, and reconfigure the Makefiles as appropriate. See Yocto Project Equivalent make Commands on page 73 for a Yocto Project equivalent to this command.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make -C build <i>recipeName.clean</i>		<p>Clean the package identified by <i>recipeName</i></p> <p>Using this command will undo any source file changes made in your package directory, consistent with Yocto Project and OpenEmbedded package build target rules.</p> <p>Note that this behavior differs from Wind River Linux 4.x, which ran the package's Makefile clean rule and typically did not remove source files from the project directory.</p> <p>This is the equivalent of bitbake -c clean <i>recipeName</i> within the BitBake environment.</p>
make -C build <i>recipeName.compile</i>		<p>This will only do the compile. If you just specify <i>recipeName</i> (with no .compile suffix), the top level dependency of <i>.sysroot</i> will trigger and the build system will compile the package, generate an RPM, and install it to the <i>sysroot</i>.</p> <p>This is the equivalent of bitbake -c compile <i>recipeName</i> within the BitBake environment.</p>
make -C build <i>recipeName.distclean</i>		<p>Clean the package identified in the recipe and the package patch list. This deletes the existing build directory of the package as well as .stamp files.</p> <p>This is the equivalent of bitbake -c distclean <i>recipeName</i> within the BitBake environment.</p>
make -C build <i>recipeName.install</i>		<p>This is the equivalent of bitbake -c install <i>recipeName</i> within the BitBake environment.</p>
make -C build <i>recipeName.patch</i>		<p>Copy package source into the build area and apply patches.</p> <p>This is the equivalent of bitbake -c patch <i>recipeName</i> within the BitBake environment.</p>

make Command in <i>projectDir</i>	Workbench Build Target	Description
make -C build <i>recipeName.rebuild</i>		Clean, then build a package. This is the equivalent of bitbake -c rebuild <i>recipeName</i> within the BitBake environment.
make -C build <i>recipeName.rebuild_noddep</i>		Rebuild <i>recipeName</i> without rebuilding dependent packages. For example, enter make -C build linux-windriver.rebuild_noddep to rebuild the kernel without rebuilding dependent userspace packages. This is the equivalent of bitbake -c rebuild_noddep <i>recipeName</i> within the BitBake environment.
make -C build <i>packageName.rmpkg</i>		Remove a package and any packages it is known to require, and reconfigure the makefiles as appropriate. This command only removes packages that were added with the make -C build <i>packageName.addpkg</i> command. It does not remove packages added using templates or the inclusion of layers as part of your platform project image build. See Yocto Project Equivalent make Commands on page 73 for a Yocto Project equivalent to this command.
make -C build <i>recipeName.unpack</i>		Unpack the packages source but stop before patching phases. This is the equivalent of bitbake -c unpack <i>recipeName</i> within the BitBake environment.
make import-package	import-package	Starts a GUI applet that assists the developer in adding external packages to a project

make Command in <i>projectDir</i>	Workbench Build Target	Description
<code>make package-manager</code>	<code>package-manager</code>	Starts a GUI applet for managing packages and package dependencies.

Kernel Development

make Command in <i>projectDir</i>	Workbench Build Target	Description
	Kernel Configuration	Wind River Workbench tool for kernel configuration
<code>make -C build linux-windriver</code>	<code>kernel_build</code>	Build the Wind River Linux kernel recipe
<code>make -C build linux-windriver.build</code>	<code>kernel_build</code>	Build the Linux kernel recipe. If you have made changes to the kernel in your project (for example with make -C build linux-windriver.menuconfig), run the linux-windriver.rebuild target, not this one, to get those changes to take effect. This is the equivalent of bitbake linux-windriver within the BitBake environment.
<code>make -C build linux-windriver.clean</code>		Clean the kernel build
<code>make -C build linux-windriver.config</code>	<code>kernel_config</code>	Extract and patch kernel source for kernel configuration. This is the equivalent of bitbake -c config linux-windriver within the bitbake environment.
<code>make -C build linux-windriver.menuconfig</code>	<code>kernel_menuconfig</code>	Extract and patch kernel source and launch menu-based tool for kernel configuration. This is the equivalent of bitbake -c menuconfig linux-windriver within the bitbake environment.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make -C build linux-windriver.reconfig		Regenerates the kernel configuration by reassembling the config fragments. This is the equivalent of bitbake -c reconfig linux-windriver within the bitbake environment.
make -C build linux-windriver.xconfig	kernel_xconfig	Extract and patch kernel source and launch X Window tool for kernel configuration. This is the equivalent of bitbake -c xconfig linux-windriver within the bitbake environment.
make DTSbasename.dtb		This supersedes make -C build linux-windriver.DTSbaseName.dtb . It must be run from a kds shell.

PR Server Usage

39

Build Variables

The list and description of `config.sh` build variables shown in the following table is provided for informational purposes only—you would not typically change `config.sh` files directly. These are constructed and inherited during the configure process from the templates.

Note that many of the items are also copied into the `config.properties` file which is used to initialize Workbench with its project information, and a few of the fields are also copied into the toolchain wrappers. Therefore, even if you modify `config.sh`, your modifications may not be carried forward to other components using the fields.

Table 11 Build Variables and Descriptions

Variable	Description
BANNER	Informational message printed when <code>configure</code> completes. Can be used in any template.
TARGET_TOOLCHAIN_ARCH	Specifies the generic toolchain architecture: arm , i586 , mips , powerpc . Must match toolchain. Generally specified in the <code>templates/arch/...</code> item. Only set in an arch template.
AVAILABLE_CPU_VARIANTS	These are all of the available CPU variants for a configuration. For example, in a Power PC 32-bit/64-bit install, both <code>ppc</code> and <code>ppc64</code> would be listed. A value from this variable is substituted for the VARIANT prefix in the following variables.

The following items should be prefixed with the VARIANT name as specified in **AVAILABLE_CPU_VARIANTS**. *VARIANT* is replaced with the specific variant, for example `VARIANT_TARGET_ARCH=powerpc` becomes `ppc_TARGET_ARCH=powerpc`.

<code>VARIANT_COMPATIBLE_CPU_VARIANT</code>	Specifies all of the CPU variants that are compatible with the specific variant. For example <code>ppc</code> is compatible with <code>ppc_750</code> .
<code>VARIANT_TARGET_ARCH</code>	The architecture used by GNU configure to specify that variant.
<code>VARIANT_TARGET_COMMON_CFLAGS</code>	CFLAGS that are beneficial to pass to an application but not required to optimize for

Variable	Description
	a multilib. Equivalent of CFLAGS=... in the environment or in a makefile.
<code>VARIANT_TARGET_CPU_VARIANT</code>	Name of a variant. Also used as the RPM architecture.
<code>VARIANT_TARGET_ENDIAN</code>	BIG or LITTLE
<code>VARIANT_TARGET_FUNDAMENTAL_ASFLAGS</code>	Flags to be passed to the assembler when using the toolchain wrapper to assemble with a given user space. These are hidden from applications.
<code>VARIANT_TARGET_FUNDAMENTAL_CFLAGS</code>	Flags to be passed to the compiler when using the toolchain wrapper to compile for a given user space. These are hidden from applications.
<code>VARIANT_TARGET_FUNDAMENTAL_LDFLAGS</code>	Flags to be passed to the linker when using the toolchain wrapper. These are hidden.
<code>VARIANT_TARGET_LIB_DIR</code>	The name of the library directory for the ABI - lib , lib32 , lib64 .
<code>VARIANT_TARGET_OS</code>	linux-gnu or linux-gnueabi
<code>VARIANT_TARGET_RPM_PREFER_COLOR</code>	The preferred color when installing RPM packages to the architecture: <ul style="list-style-type: none"> • 0—No preference • 1—ELF32 • 2—ELF64 • 4—MIPS ELF32_n32 <p>"Color" is RPM terminology for a bitmask used in resolving conflicts. If RPM is going to install two files, and they have conflicting md5sum or sha1, it uses the color to decide if it can resolve the conflict. Two files of color 0 cause a conflict and the install fails. Otherwise, the system's "preferred" color takes precedence for the install. If the file is outside of the permitted colors, then again it is an error (if it causes a conflict).</p>
<code>VARIANT_TARGET_RPM_TRANSACTION_COLOR</code>	The colors that are allowed when installing RPM packages to that architecture. A bitmask of the above. For example, on a 32-bit system, generally 1. On a 64/32 bit system, 3. On a mips64 system, 7.
<code>VARIANT_TARGET_RPM_SYSROOT_DIR</code>	The internal gcc directory prefix to get to the sysroot information.
<code>VARIANT_TARGET_USERSPACE_BITS</code>	Bitsize of a word, 32 or 64 .
BSP-Specific Variables	
<code>BOOTIMAGE_JFFS2_ARGS</code>	For targets that support JFFS2 booting, these values will be passed when creating the JFFS2 image.

Variable	Description
	Endianness (-b/-l), erase block size (-e), and image padding (-p) are commonly passed.
KERNEL_FEATURES	Features to be implicitly patched into the kernel independent of the configure command line and options.
LINUX_BOOT_IMAGE	Name of the image used to boot the board, used to create the export default image symlink.
TARGET_BOARD	BSP name as recognized by the build system.
TARGET_LINUX_LINKS	List of images is created by the kernel build.
TARGET_PLATFORMS	Mainly used for compatibility reasons. Indicates which platform(s) a particular board supports.
TARGET_PROCFAM	Internal Wind River use only.
TARGET_SUPPORTED_KERNEL	The list of kernels supported by a particular board.
TARGET_SUPPORTED_ROOTFS	List of root file systems supported by a particular board.
TARGET_TOOLS_SUBDIRS	Additional host tools that should be built to support this board.
QEMU-related variables	
Refer to the release notes for details on QEMU-supported targets. Enter make config-target in <i>projectDir</i> for additional information.	
TARGET_QEMU_BIN	The QEMU host tool binary to use, if this BSP can be simulated by QEMU.
TARGET_QEMU_BOOT_CONSOLE	The console port the target uses. This is BSP specific. For example, for qemux86-64 it is ttyS0 .
TARGET_QEMU_ENET_MODEL	Some BSPs such as the qemux86 and qemux86-64 use a different Ethernet type. This parameter can be used to select a different Ethernet type to override the default that is hard coded in the QEMU host binary.
TARGET_QEMU_KERNEL	The “short” name of the boot image to search for in the export directory inside the <i>BUILD_DIR</i> . For qemux86-64 it would be set to bzImage or for the arm_versatile_926ejs it would be set to zImage . The specific image that is used is based on the boot loader that is hard-coded into the QEMU binary. This image is different than the boot image the real target might use in some cases. If you specify a full path to a binary kernel image it will not search the export directory and will instead use the image you specified.

Variable	Description
TARGET_QEMU_KERNEL_OPTS	These are any extra options you might want to pass to the kernel boot line to override the defaults.
TARGET_QEMU_OPTS	These are any additional options you need to pass to the QEMU binary to get it to run correctly. In the case of the ARM Versatile and MTI Malta boards, the -M argument is passed so that the QEMU host binary will be configured with the correct simulation model since each host binary supports multiple simulation models within the same architecture.
Feature or Root File System Specific Items	
TARGET_LIBC	Value should be glibc or uclibc . No value defers to the default value glibc .
TARGET_LIBC_CFLAGS	Additional flag to add to the fundamental cflags (in the toolchain wrapper) for the libc being used. Normally this is blank except for the uclibc case where it is -muclibc . This is hidden from the application space.
TARGET_ROOTFS_CFLAGS	An additional CFLAG that needs to be used when a feature or rootfs is specified. Again hidden from the application space
TARGET_ROOTFS	Name of the configured ROOTFS.
Generic Optimizations	
TARGET_COPT_LEVEL	These are all optional optimizations that override defaults in configure. Generally you use these if you want to change the optimizations for -Os and not -O2 . See the glibc_small rootfs for an example.
TARGET_COMMON_COPT	
TARGET_COMMON_CXXOPT	

Additional Notes on Build Variables

multilib templates are designed to match the multilibs as defined by the compiler and libcs. The cpu templates are expected to include a multilib template and either use it “as-is” or augment it with additional optimizations.

Only multilib templates are allowed to specify **TARGET_FUNDAMENTAL_*** flags. cpu templates can only specify:

- **TARGET_COMMON_CFLAGS**
- **TARGET_CPU_VARIANT**
- **AVAILABLE_CPU_VARIANTS**
- **COMPATIBLE_CPU_VARIANTS**

Everything else is expected to be inherited from multilib templates.

For all of the items in the **multilib/cpu** templates, they should be prefixed with the variant name. The following items are required to be prefixed with a variant:

- **TARGET_COMMON_CFLAGS**

- TARGET_CPU_VARIANT
- TARGET_ARCH
- TARGET_OS
- TARGET_FUNDAMENTAL_CFLAGS
- TARGET_FUNDAMENTAL_ASFLAGS
- TARGET_FUNDAMENTAL_LDFLAGS
- TARGET_SYSROOT_DIR
- TARGET_LIB_DIR
- TARGET_USERSPACE_BITS
- TARGET_ENDIAN
- TARGET_RPM_TRANSACTION_COLOR
- TARGET_RPM_PREFER_COLOR COMPATIBLE_CPU_VARIANTS
- TARGET_ROOTFS—only specify in a ROOTFS template
- TARGET_COPT_LEVEL, TARGET_COMMON_COPT, TARGET_COMMON_CXXOPT - specify either ROOTFS or board template, do not specify CPU or **Multilib**.

The best way to determine what to do in a custom template is use wrll-wrlinux as an example, with the information provided here in order to create custom templates.

40


Package Variable Listing

The tables in this section provide a list of the package variables in Wind River Linux.

The following variables are fixed, and identify specific information about the package:

Variable	Description
<i>pkg_RPM_DEFAULT</i>	Lists all of the produced binary packages that should be installed on the target file system (usually excludes development packages.)
<i>pkg_RPM_ALL</i>	Lists all of the packages produced (does not inherit from any other list.) This is used as a validation that the package is being produced properly. If this (and RPM_IGNORE) do not match what RPM tells the build system will be produced, a warning message is generated telling you that you should update your makefile.
<i>pkg_TYPE=</i>	For an SRPM package, this must be set to SRPM . For classic packages (ordinary compressed source files) for which you have added a spec file, it must be set to spec . For adding a classic package without a spec file, leave this field empty.
<i>pkg_VERSION=</i>	Version-release of the src.rpm package
<i>pkg_ARCHIVE=</i>	Complete file name of the src.rpm package
<i>pkg_MD5SUM=</i>	The MD5 checksum of the src.rpm package
<i>pkg_UPSTREAM=</i>	The download site of the src.rpm package

The following variables may be defined, if necessary:

Variable	Description
<code>pkg_RPM_NAME=</code>	Necessary if the produced RPM name is different from <code>pkg_NAME</code> , or if more than one binary is produced.
<code>pkg_DEPENDS</code>	A list of dependencies which must be built before this package is built.
<code>pkg_RPM_DEVEL</code>	Lists all of the development packages. These plus the <code>pkg_RPM_DEFAULT</code> list are installed into the sysroot for development purposes. This is only required if the package produces development RPMs, that is, binary RPMs that contain information that must be installed into the sysroot for other programs to build properly
	<hr/>  NOTE: The sysroot is populated by installing both <code>pkg_RPM_DEFAULT</code> and <code>pkg_RPM_DEVEL</code> . <hr/>
<code>pkg_RPM_IGNORE</code>	In a few cases, the RPM program reports it will generate a package, that it doesn't actually generate. This is a way to capture those situations.

41

Lua Scripting in Spec Files

Lua is a scripting language with an interpreter built into rpm. This allows you to write `%pre` and `%post` lua scripts to be run at pre- and post-installation.

The `wrs` library is included in the lua interpreter from Wind River. It consists of three functions:

- `wrs.groupadd()`
- `wrs.useradd()`
- `wrs.chkconfig()`

The following provides an example of a post-install section that creates a group and user named `named`:

```
%wrs_post -p <lua>
wrs.groupadd('-g 25 named')
wrs.useradd('-c "Named" -u 25 -g named -s /sbin/nologin -r -d /var/named named')
```

Each function takes one argument, which is the string you would enter at the shell prompt if you were running the Linux command of the same name.

Spec file macros are expanded within the string, so the following works as expected:

```
%wrs_pre -p <lua>
wrs.groupadd('-g %{uid} -r %{gname}')
wrs.useradd('-u %{uid} -r -s /sbin/nologin -d /var/lib/heartbeat/cores/hacluster -M -c
"heartbeat user" -g %{gname} %{uname}')
```

As can be seen from the file names, when the lua script executes, the “root” directory is the root of the target file system.

The `base`, `table`, `io`, `string`, `debug`, `loadlib`, `posix`, `rex`, and `rpm` libraries are also built-in to the lua interpreter. Their use, and general lua programming is not covered here. For more information on the Lua scripting language, see <http://www.lua.org>.

Kernel Audit Directory Contents

Audit data is stored in the `projectDir/build/linux-windriver/linux/meta/cfg/kernel_type/BSP_alias/` directory. The contents of this directory are refreshed for every `linux-windriver.config` or `linux-windriver.reconfig`.

The table below describes the contents of the files that appear in the audit data directory.

For more information on kernel auditing, see [Kernel Configuration Fragment Auditing](#) on page 189.

File	Description
<code>all.kcf</code>	Alphabetical listing of all Kconfig files found in this kernel.
<code>known_current.kcf</code>	List of previously categorized Kconfig files present in the patched linux tree about to be used for compilation.
<code>known.kcf</code>	List of Kconfig files for which the build system has already information on whether to be classified as hardware or not
<code>non-hardware.kcf</code>	List of Kconfig files known to contain non-hardware related items.
<code>hardware.kcf</code>	Kconfig files that are to be treated as containing hardware options.
<code>unknown.kcf</code>	List of Kconfig files present in the about-to-be-used linux tree that are not known by the build system to be either hardware or non-hardware items.
<code>all.cfg</code>	Alphabetical listing of all the <code>CONFIG_</code> items found in this kernel
<code>always_hardware.cfg</code>	<code>CONFIG_</code> items that are to be treated as always hardware, regardless of what Kconfig file they are in.

File	Description
always_nonhardware.cfg	As above, but non-hardware.
avail_hardware.cfg	All the options from all the hardware-related Kconfig files, less those options found in always_nonhardware.cfg
specified.cfg	List of the CONFIG_ items specified by the BSP.
specified_hdw.cfg	List of the CONFIG_ items specified by the BSP which are hardware (ideally this should be almost all of them).
specified_non_hdw.cfg	List of the CONFIG_ items specified by the BSP which are non-hardware (ideally this should be almost always empty).
fragment_errors.txt	Settings which are specified multiple times within a single fragment.
redefinition.txt	List of options that are set in one fragment and then reset in another later on.
invalid.cfg	Configuration options specified in the BSP that do not match any known valid option, that is, this item is not in any Kconfig file.
<i>BSP-kernel_type-kernel_version</i>	A concatenation of all the file fragments. The file of the same name in projectDir is a symlink to this file.
config.log	The output of the LKC processing as it creates the final .config file.
