

W1ND™

WIND RIVER® LINUX

USER'S GUIDE

7.0



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

22 May 2015

Contents

PART I: INTRODUCTION

1 Wind River Linux Overview	17
Wind River Linux	17
Kernel and File System Components	18
Supported Run-time Boards	21
Optional Add-on Products	22
Product Updates	22
Identifying the Installed RCPL Version	23
Updating Wind River Linux	23
Updating Wind River Linux Using the git-based Installer	24
About Obtaining BSPs	27
Licensing New BSPs for Wind River Linux 7.0	27
Installing New BSPs	28
Documentation Updates	29
Update your installation's documentation to the latest version	30
Get the Latest Documentation from Wind River Online Support	30
Creating Platform Project Build Error Reports	30
2 Run-time Software Configuration and Deployment Workflow	33
3 Development Environment	35
Directory Structure	35
Metadata	40
Configuration Files and Platform Projects	41
Assigning Empty Values in BitBake Configuration Files	46
README Files in the Development Environment	47
Viewing a Specific README File in the Installation	47
Cloning a Layer to View Installation README Files	47
Viewing All Installation README Files in a Web Browser	48
Creating Subset Repositories of meta-openembedded Layers	48
Using the Repository Subset Script	50
Repository Subset Script Options Reference	53
4 Build Environment	55
About the Project Directory	55

Creating a Project Directory	56
Directory Structure for Platform Projects	57
Feature Templates in the Project Directory	61
Kernel Configuration Fragments in the Project Directory	67
Viewing Template Descriptions	70
About the layers/local Directory	71
About README Files in the Build Environment	72
Adding a Layer to a Platform Project to View README Files	73
Adding All Layers to a Platform Project to View All README Files	73

PART II: PLATFORM PROJECT IMAGE DEVELOPMENT

5 Configuration and Build	77
About Configuring a Platform Project Image	77
About Creating the Platform Project Build Directory	79
Initializing the Wind River Linux Environment	79
About the Configure Script	80
About the Platform Project Configure Tool	96
About Building Platform Project Images	97
About the make Command	98
Yocto Project Equivalent make Commands	100
About Build Logs	102
About Managing Builds with the Yocto Project Toaster	102
Running Toaster	103
Sample Toaster Build Output	103
Shutting Down Toaster Web Server	107
Restarting Toaster Web Server	107
Collecting Toaster Data for Multiple Build Project Directories	108
Changing Toaster Servers	108
Recapturing Complete Toaster Data for Subsequent Builds	109
Build-Time Optimizations	109
About Accelerating Builds with a Remote Shared State Cache Server	110
Setting Up a Remote Shared State Cache Server	111
Accelerating Builds with Shared State Cache	113
Creating and Using a Shared State Cache with Multiple Host Operating Systems	113
Maintaining Shared State Cache	114
Securing Shared State Cache	114
Using the Read-only Template and Whitelist to Secure Shared State Cache	116
Using Verification and Signing Options to Secure Shared State Cache	119
Examples of Configuring and Building	122

Configuring and Building a Complete Run-time	123
Commands for Building a Kernel Only	123
Configuring and Building a Flash-capable Run-time	124
Configuring and Building a Debug-capable Run-time	124
Building a Target Package	125
About Verifying Builds with the Image Manifest	126
Verifying Builds Using Image Manifest and the Audit Tool	127
About Maintaining Open Source License Compliance	128
Exporting Using the Archiver Feature	129
Archiver Options Reference	131
About Creating Custom Configurations Using <code>rootfs.cfg</code>	132
About the <code>rootfs.cfg</code> File	133
About New Custom <code>rootfs</code> Configuration	135
Glibc File Systems	136
Creating and Customizing Glibc Platform Project Images	138
Glibc Option Mapping Reference	140
6 Localization	143
About Localization	143
Determining which Locales are Available	143
Setting Localization	145
7 Portability	147
About Platform Project Portability	147
Copying or Moving a Platform Project	148
Updating a Platform Project to a New Wind River Linux Installation Location	149
8 Layers	151
About Layers	151
About BSP Layers and Sublayers	152
Layers Included in a Standard Installation	152
Installed Layers vs. Custom Layers	156
Layer Structure by Layer Type	156
About Layer Processing and Configuration	158
About Processing a Project Configuration	159
Creating a New Layer	159
Enabling a Layer	160
Disabling a Layer	161
About Exporting Local Layer Changes	161
Exporting Local Layer Changes	162
Combining Platform Project Layers	164
9 Recipes	167

About Recipes	167
A Sample Application Recipe File	168
About Recipe Files and Kernel Modules	169
Extending Recipes with .bbappend Files	169
Creating a Recipe File	170
Identifying the LIC_FILES_CHKSUM Value	171
10 Templates	173
About Templates	173
Adding Feature Templates	174
Adding Kernel Configuration Fragments	174
Template Processing	175
11 Finalizing the File System Layout with changelist.xml	177
About File System Layout XML Files	177
About File and Directory Management with XML	178
Device Options Reference	178
Directory Options Reference	179
File Options Reference	180
Pipe Options Reference	181
Symlink Options Reference	182
The Touched/Accessed touch.xml Database File	182

PART III: USERSPACE DEVELOPMENT

12 Developing Userspace Applications	187
About Application Development	187
Cross Development Tools and Toolchain	188
About Sysroots and Multilibs	188
About Obtaining Package Source not Provided by Wind River	192
Creating a Sample Application	196
Exporting the SDK	199
Exporting the SDK	199
Sourcing the SDK	201
Exporting the SDK for Windows Application Development	202
Sourcing the SDK for Windows Application Development	203
Using the SDK	204
About Using the SDK	204
Compiling Programs Using the SDK Environment Variables	205
SDK Environment make Command Reference	206
SDK Environment Variables Reference	207

Adding Applications to a Platform Project Image	209
Options for Adding an Application to a Platform Project Image	209
Adding New Application Packages to an Existing Project	209
Adding an Application to a Root File System Using changelist.xml	211
Adding an Application to a Root File System with fs_final*.sh Scripts	212
Configuring a New Project to Add Application Packages	213
Verifying the Project Includes the New Application Package	214
Importing Packages	215
About the Package Importer Tool (import-package)	215
Importing a Sample Application Project as a Package	215
Importing a Source Package from the Web (wget)	217
Importing a SRPM Package from the Web	219
Listing Package Interdependencies	221
About the BitBake Build Environment Display Tool	222
Displaying the BitBake Environment Details	226
BitBake Environment Display Tool Reference	227
13 Understanding the User Space and Kernel Patch Model	229
Patch Principles and Workflow	229
Patching Principles	230
Kernel Patching with scc	231
14 Patching Userspace Packages	235
Introduction to Patching Userspace Packages	235
Patching with Quilt	237
Create an Alias to exportPatches.tcl to save time	237
Preparing the Development Host for Patching	237
Patching and Exporting a Package to a Layer	238
Verifying an Exported Patch	240
Incorporating a Patch into a Platform Project Image	241
15 Modifying Package Lists	243
About the Package Manager	243
Launching the Package Manager	244
Removing Packages	245
About Modifying Package Lists	247
Adding a Package	248
About Adding Templates	248
Removing a Package	249
16 Maintaining Package Recipe Revisions	251
About Package Revision Management	251
Enabling Package Build History	253

Querying the PR Server Database	254
Comparing Build History with Previous Revisions	255

PART IV: KERNEL DEVELOPMENT

17 Understanding the Wind River Linux Kernel	261
About the Wind River Linux Kernel	261
About the Baseline Kernel	262
About Branching Features and Kernel Types	263
Kernel Branching and Tagging Strategy	264
Kernel Development Tools	265
Kernel Tree Construction and Workflow	266
About Creating and Maintaining Custom Kernel Branches	267
Creating a Custom Kernel Repository	269
Updating a Hybrid Kernel Branch	270
Building a Custom Kernel Branch	271
Viewing Changes Applied to a Kernel Tree	272
About Saving Kernel Modifications	272
About Saving Incremental Kernel Changes	274
Using Bulk Export to Save Kernel Modifications	275
Exporting Committed Changes Internally with Git	276
Exporting Committed Changes Internally as Patches	277
Exporting Committed Changes Upstream	278
Working with -dirty Kernel Strings	278
18 Patching and Configuring the Kernel	281
About Kernel Configuration and Patching	281
Configuration	282
Patching	293
19 Creating Optimized Custom Kernel Builds	303
About Optimized Custom Kernel Builds	303
Creating a Platform Project with a Dummy Kernel	305
Building the Kernel Using the Custom Kernel Recipe	306
Building the Kernel from External Source	308
Extracting the Kernel Build Output	309
20 Creating Alternate Kernels from kernel.org Source	311
21 Exporting Custom Kernel Headers	315
About Exporting Custom Kernel Headers for Cross-compile	315

Adding a File or Directory to be Exported when Rebuilding a Kernel	315
Exporting Custom Kernel Headers	316
22 Using the preempt-rt Kernel Type	319
About Using the preempt-rt Kernel Type	319
Enabling Real-time	321
Configuring preempt-rt Preemption Level	321
PART V: DEBUGGING AND ENABLING ANALYSIS TOOLS SUPPORT	
23 Kernel Debugging	327
Kernel Debugging	327
Debugging with KGDB Using an Ethernet Port (KGDBOE)	328
Debugging with KGDB Using the Serial Console (KGDBOC)	330
Disabling KGDB in the Kernel	331
Kernel Debugging with QEMU	332
24 Userspace Debugging	333
Adding Debugging Symbols to a Platform Project	333
Adding Debugging Symbols for a Specific Package	334
Dynamic Instrumentation of User Applications with uprobes	335
Configuring uprobes with perf	337
Dynamically Obtain User Application Data with uprobes	337
Dynamically Obtain Object Data with uprobes	339
Debugging Individual Packages	342
Debugging Packages on the Target Using gdb	342
Debugging Packages on the Target Using gdbserver	343
25 Analysis Tools Support	345
About Analysis Tools Support	345
Using Dynamic Probes with ftrace	345
Preparing to use a kprobe	348
Setting up a kprobe	348
Enabling and Using a kprobe	349
Disabling a kprobe	350
Analysis Tools Support Examples	351
Adding Analysis Tools Support for MIPS Targets	351
Adding Analysis Tools Support for Non-MIPS Targets	351

PART VI: USING SIMULATED TARGET PLATFORMS FOR DEVELOPMENT

26 QEMU Targets	355
About QEMU Targets	355
QEMU Target Deployment Options	356
Setting QEMU Configuration Options	356
Accessing the QEMU Monitor	357
Viewing QEMU Command Line Options	357
Managing QEMU Targets	358
Starting a QEMU Session	358
Resolving QEMU Start Errors	359
Running Multiple QEMU Sessions	359
Starting a QEMU Session From a .iso File	359
Starting a QEMU Session From a Disk Image	359
Starting a QEMU Session With a Graphics Console	360
Passing Boot Options to QEMU	360
Using Multiple QEMU Options	360
Port Mappings for Accessing the QEMU Target Simulation	360
Ending a QEMU Session	361
TUN/TAP Networking with QEMU	361
Configure TUN/TAP in Workbench for a New Connection	362
Configure TUN/TAP in Workbench for Existing Target Connections	362
Configure TUN/TAP from the Command Line	363
27 Wind River Simics Targets	365
About Wind River Simics Targets	365
Using Simics from the Command Line	365
Meeting Simics Prerequisites	366
Launching the Simics Basic Target Console	366
Launching the Simics Graphics Target Console	366
Enabling Simics Acceleration for x86 BSPs	367
Configuring a Simics Target	367

PART VII: DEPLOYMENT

28 Managing Target Platforms	371
Customizing Password and Group Files	371

Using an <code>fs_final.sh</code> Script to Edit the Password and Group File	374
Using an <code>fs_final_sh</code> Script to Overwrite the Password and Group File	374
About <code>ldconfig</code>	375
Enabling <code>ldconfig</code> Support	375
Connecting to a LAN	376
Adding an RPM Package to a Running Target	377
Adding Reference Manual Page Support to a Target	378
About Compressing Documentation on Targets	379
Compressing Target Documentation	379
Using <code>Pseudo</code>	380
About Using <code>Pseudo</code>	380
Examining Files using <code>Pseudo</code>	381
Navigating the Target File System with <code>Pseudo</code>	381
29 Deploying Flash or Disk Target Platforms	383
About Configuring and Building Bootable Targets	383
Host-Based Installation of Wind River Linux Images	384
Configuring and Building the Host Install	387
Booting and Installing from a USB or ISO Device	388
Booting and Installing with QEMU	389
Creating u-boot RAM Disk Images	392
Creating Bootable USB Images	394
make <code>usb-image-burn</code> Example Output	397
Creating ubifs Bootable Flash Images	399
Enforcing Read-only Root Target File Systems	400
Installing with the GUI Installer	400
Installing with the Serial Console Installer	408
Installing or Updating <code>bzImage</code>	409
About Manually Configuring a Boot Disk with Files from a USB/ISO Image	411
Prepare the Target's Hard Drive	411
Place the File System and Kernel on the Hard Disk	414
Configure the Target System Files and Boot	414
About Deploying an Image with a Virtual Machine Manager	416
Using VMware with <code>vmdk</code> Boot Images	416
Using VirtualBox with <code>vmdk</code> Boot Images	418
30 Deploying initramfs System Images	421
About <code>initramfs</code> System Images	421
Creating <code>initramfs</code> Images	422
Adding Packages and Kernel Modules to <code>initramfs</code> Images	423
31 Deploying KVM System Images	425

About Creating and Deploying KVM Guest Images	425
Create the Host and Guest Systems	427
Deploying a KVM Host and Guest	428

PART VIII: TESTING

32 Running Linux Standard Base (LSB) Tests	433
About the LSB Tests	433
Testing LSB on Previously Configured and Built Target Platforms	434
Disabling Grsecurity Kernel Configurations on CGL Kernels	435
Running LSB Distribution Tests	435
Running LSB Application Tests	437
33 Validating Platform Project Images with ptest	441
Testing With ptest	441
Building a Platform Project and Running ptest to Collect Results	442
Running ptest-diff.sh	446
Running ptest-summary.sh	447
Processed ptest-run Log File Example Results	448
ptest-summary.sh and ptest-diff.sh Option Reference	450
34 Running Open POSIX Tests	451
About the Open POSIX Test Suite	451
Running the Open POSIX Tests	452

PART IX: OPTIMIZATION

35 About Optimization	459
36 Analyzing and Optimizing Runtime Footprint	461
Analyzing and Optimizing Runtime Footprint	461
Collecting Platform Project Footprint Data	462
Footprint (fetch-footprint.sh) Command Option Reference	465
37 Reducing the Footprint	467
About BusyBox	467
Configuring a Platform Project Image to Use BusyBox	467
About devshell	468
About Static Linking	469
About the Library Optimization Option	469

38 Analyzing and Optimizing Boot Time	471
Analyzing and Optimizing Boot Time	471
Creating a Project to Collect Boot Time Data	472
Analyzing Early Boot Time	473
About Reducing Early Boot Time	475
Reducing Network Initialization Time with Sleep Statements	475
Reducing Device Initialization Time	476
Removing Unnecessary Device Initialization Times	477

PART X: TARGET-BASED DEVELOPMENT

39 Building a Self-Hosting Target System	481
About Building Self-Hosting Target Systems	481
Building a Self-Hosting Platform Project Image	482
40 Developing on the Target	483
Building a Wind River Linux Platform Project	483
Building Software Packages on the Target	485
Building a Modified Kernel on the Target	487

PART XI: TARGET-BASED NETWORKING

41 About Target-based Networking	491
42 Setting Target and Server Host Names	493
43 Connecting a Board	495
Configuring a Serial Connection to a Board	495
Setting-up cu and UUCP	495
Setting up the Workbench Terminal	496
About Configuring PXE	496
Configuring PXE	498
Configuring DHCP	499
Configuring DHCP for PXE	500
Configuring NFS	502
Configuring TFTP	503

PART XII: REFERENCE

44 Additional Documentation and Resources	507
Document Conventions	507
Wind River Linux Documentation	508
Additional Resources	508
Open Source Documentation	509
External Documentation	511
45 Common make Command Target Reference	513
46 Build Variables	525
47 Platform Kernel Versions by Product Release	531
48 Lua Scripting in Spec Files	533

PART I

Introduction

Wind River Linux Overview.....	17
Run-time Software Configuration and Deployment Workflow.....	33
Development Environment.....	35
Build Environment.....	55

1

Wind River Linux Overview

Wind River Linux	17
Product Updates	22
About Obtaining BSPs	27
Documentation Updates	29
Creating Platform Project Build Error Reports	30

Wind River Linux

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

Wind River Linux 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
- package revision management through the implementation of the package revision server.
- a custom USB image tool for platform project images

Wind River Linux supports all Yocto Project build commands, but also offers simplified configure and build commands based on the Wind River Linux LDAT 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>.

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 7.0.

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.

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 [About Using the preempt-rt Kernel Type](#) on page 319.

tiny

This kernel profile is intended for use with the [*glibc_tiny*](#) file system. To keep the footprint small, the **tiny** kernel's default configuration is not POSIX-compliant. As a result, it does not support Linux Test Project (LTP) tests.



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 281

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 Tiny (`glibc_tiny`)

A minimal file system you can use as a starting point for developing very small distributions for resource constrained devices. This includes the tiny kernel, a tiny BusyBox, and a custom init.

The file system removes many common tools and features. Because of its minimal nature, you may want to consider customizing your kernel, BusyBox and system behavior. Even basic features like **shutdown** have been removed. If you need to add a feature to the system, you should review existing features to learn how they are implemented as a starting point.

See the [Poky-Tiny](#) project and [About Glibc Tiny](#) for more information about **Glibc Tiny**.

wr-installer

A Wind River Linux-supplied root file system option for creating a target platform with installer support. Once the platform project image is complete and transferred to a disk, the installer feature provides a graphics-based menu for installing the distribution.

See [Host-Based Installation of Wind River Linux Images](#) on page 384 for additional information.

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	Tiny
glIBC_core	Yes	Yes	No
glIBC_std	Yes ¹	Yes	No
glIBC_std_sato	Yes	Yes	No
glIBC_small	Yes	Yes	No
glIBC_cgl	No	No	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 7.0 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 67.

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.

Supported Run-time Boards

Supported run-time boards define the target platforms supported by Wind River Linux.

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 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. 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/READMES** directory after you configure a project. To view these files, see [About README Files in the Build Environment](#) on page 72

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 383 and [About Target-based Networking](#) on page 491.

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 7.0. The following table provides correlation between Wind River Linux 4.x and 7.0.

Processor Family	4.x BSP Name	7.0 BSP Name
MIPS	cav_octeon_cn63xx	cav-octeon3
	cav_octeon_cn68xx	cav-thunderx
	cav_octeon2	
PPC	fsl_p204x	fsl-t2xxx
	fsl_p3041	fsl-ls10xx
	fsl_p4080	

Processor Family	4.x BSP Name	7.0 BSP Name
x86	intel_atom_eg20t_pch intel_atom_z530 kbc_km2m806 intel_atom_n4xx_d5xx_82801h m	intel_x86
QEMU MIPS	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 its capabilities to meet your development needs.

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

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.

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.

Identifying the Installed RCPL Version

Before you perform a product update, check to see which version you currently have installed using **git branch -r**.

Since your Wind River Linux installation is a git repository, you can use git commands to determine the installed product version.

Step 1 Navigate to the Wind River Linux installation directory.

```
$ cd installDir/wrlinux-7
```

Step 2 Display the installed product information.

```
$ git branch -r
origin/LB21_7.0
origin/LB21_7.0_RCPL0002
```

Note that all of the installed versions display. In this example, the original version and RCPL 2 (**RCPL0002**) are installed.

Updating Wind River Linux

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

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.

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
```

Updating Wind River Linux Using the git-based Installer

You can use the git-based installer, **install-WRL7.pl**, to download Wind River Linux software releases.

This procedure explains how to use the git-based installer to copy Wind River Linux software updates from the WindShare portal. If you wish to perform a minimal installation, you must use the git-based installer script.

To complete this procedure, you will need your Wind River WindShare credentials, including your user name and password. Once the installation script verifies your credentials, the software is downloaded.

Step 1 Navigate to your Wind River Linux installation directory.

During installation of Wind River Linux, the git-based installer script **install-WRL7.pl** is placed in the installation directory.

Step 2 Run the script.

To perform a full installation, run the following:

```
$ ./install-WRL7.pl
```

To perform a minimal installation, run the following:

```
$ ./install-WRL7.pl --minimal \
--url https://windshare.windriver.com/remote.php/webdav
```

You will be prompted for a user name and password, unless you use the command line options to provide this information. For information about the command line options, see: [Installer Script Option Reference](#) on page 25.

The installation or update begins immediately once the credentials are verified, and provides progress status in the terminal.

Installer Script Option Reference

Use **install-WRL7.pl** script options to customize your installation.

Options

--help -h		Brief help message
--man		Display full documentation
--dest <i>destdir</i>	file path	Location for product installation. The file path must only include alphanumeric characters with no spaces or special characters.
--download <i>number</i>	number from 1 to 5	Sets the number of connections to download the release
--insecure		Disable SSL root CA validation
--local-dir <i>localdir</i>	file path	Local copy of the install tree
--password <i>password</i>	alphanumeric	Wind Share password
--user <i>username</i>	alphanumeric	Wind Share login name
--verbose -v		Enable more logging
--version -V		Print version information

Advanced Options

--branch <i>branch</i>	alphanumeric	Specify a branch name to install from
------------------------	--------------	---------------------------------------

--map <i>location</i>	URL	URL for a valid map file
--maxtime	numeric	Maximum time for curl downloads
--minimal		Specifies for a minimal installation. When enabled, only a minimal number of downloads will occur during the installation. Additional downloads are performed during the build process.
		For a product update, updates the packages in your minimal installation. If you want to use the full installation of Wind River Linux instead, it is recommended that you reinstall the product.
--noexec -n		Do not perform the installation from clone repositories
--shared <i>cloneDir</i>	file path	Use a share clone from a local installation.
		NOTE: Use for testing only, and not for production use.
--silent		Do not write to stdout
--skipto	numeric	Skip to a particular step in the install
--timeout	numeric	Timeout value for initial curl server connection
--use-srv <i>server</i>	server name or IP address	Specify a release git server for the install
--userpass	alphanumeric	Accept user name and password from stdin user:password

About Obtaining BSPs

To support customers' development needs, Wind River continually releases new BSPs for additional boards and processor families.

Contact your Wind River representative to learn about newly available architecture support.

You obtain and install Wind River Linux BSPs by electronic software delivery (ESD). (This does not apply to other products such as VxWorks or earlier releases of Wind River Linux.)

Licensing New BSPs for Wind River Linux 7.0

Depending on the product(s) you have purchased, you may need new installation keys or license updates for BSPs that become available post-release, referred to as *async* BSPs.

Existing Installation Keys and Licensing Terms Apply

For the following products, you can use your existing Product Activation File (PAF, also known as the `install.txt` file):

- Wind River Linux 7.0.0 Project Type 4, Time Based
- Wind River Linux 7.0.0 Project Type 4, Time Based

No New Installation Keys or Licensing Updates Required

For the following products, no additional install keys or licensing updates are required because these products do not deliver BSPs:

- Add Wind River Linux 7.0.0 Tools with Wind River Workbench 3.3, Time Based
- Add Wind River Linux 7.0.0 Tools with Wind River Workbench 3.3, Subscription
- Add Runtime Open Source Software Disclosure for Wind River Linux 7.0.0, Time Based
- Add Runtime Open Source Software Disclosure for Wind River Linux 7.0.0, Subscription
- Add Wind River Linux 7.0.0 Rolling Cumulative Feature Layer (RCFL), Subscription

Updates Based on Processor Family (ProcFam) Licensing (Standard Model Only)

Ask your Wind River representative if a new BSP is included in the ProcFam you have licensed. If it is, request an update order to add the keys needed for the New BSP (and go get a new PAF). If it is not, you would need to license the additional ProcFam.

If the new BSP is contained within the ProcFam you have licensed, it would apply to these products:

- Add Processor Family for Wind River Linux 7.0.0 Single Project, Perpetual
- Add Processor Family for Wind River Linux 7.0.0 with Wind River Workbench 3.3, Platform Developer, Perpetual

If the new BSP is in a different ProcFam than any you have licensed, it would apply to these products:

- Add Processor Family for Wind River Linux 7.0.0 Single Project, Perpetual
- Add Processor Family for Wind River Linux 7.0.0 with Wind River Workbench 3.3, Platform Developer, Perpetual

Updates Based on BSP (New Model Only)

- Replacing an Existing BSP with a New BSP

If you would like to replace a previously purchased BSP with a new BSP, contact your Wind River representative to inquire about this possibility. The following products are delivered by BSP basis and may be applicable in this scenario:

Project-Based Business Model

- Wind River Linux 7.0.0 Project Type 0, Time Based
- Wind River Linux 7.0.0 Project Type 0, Subscription
- Wind River Linux 7.0.0 Project Type 1, Time Based
- Wind River Linux 7.0.0 Project Type 1, Subscription
- Wind River Linux 7.0.0 Project Type 2, Time Based
- Wind River Linux 7.0.0 Project Type 2, Subscription
- Wind River Linux 7.0.0 Project Type 3, Time Based
- Wind River Linux 7.0.0 Project Type 3, Subscription



NOTE: If you replace a BSP with another one on your Customer license, (as opposed to adding a new BSP), you are no longer entitled to support or updates on the replaced BSP(s).

- Adding BSPs

To purchase additional BSPs, you must order one of the following products:

- Add Wind River Linux 7.0.0 Board Support Package, Time Based
- Add Wind River Linux 7.0.0 Board Support Package, Subscription

Acquiring a New Product Activation File (PAF)

After your update order has been processed and finalized, you can go to the licensing portal and retrieve your new PAF (this is the `install.txt` file). Alternatively, the Wind River Licensing Support team may help you retrieve a new PAF.



NOTE: Even if your order has been finalized, you will not be able to install new BSP(s) until you acquire the PAF.

Once you have the PAF you can install the new BSP(s) as described next.

Installing New BSPs

After retrieving your new Product Activation File (the `install.txt` file) as needed, you can install the async BSP(s).



NOTE: Each BSP is supported by a toolchain architecture, and if you do not have the corresponding toolchain architecture installed, your new BSP will not be usable. For example, you will not be able to configure a project with the BSP, and it will appear incomplete in the Workbench platform project wizard.

You can install the toolchain before or after installing the BSP. To do so, re-insert each of your Wind River Linux discs in succession. At the activation screen, point the installer to your new Product Activation File and proceed to install the new architecture support.

Step 1 Prepare for installation.

- a) Shut down all product processes, including any license servers.
- b) If you have altered any Wind River-supplied files in your installation tree (not a recommended practice), archive them manually before installing any updates, as these files may be overwritten during the installation process.

Step 2 Install the BSPs.

- a) Launch the installer.

Go to *installDir/maintenance/wrInstaller/x86-linux2* and run **wrInstaller**.

- b) Select **Product updates**.

The installer will automatically search all available updates and verify your entitlement. (This may take a few minutes.)

Step 3 Continue through the installer screen selections, until you get to the Choose Activation Type screen.

- If your existing keys are up-to-date, select **Use the existing product activation file**.
- If you have ordered new BSPs, other products, or received upgrades and have retrieved an updated Product Activation File (PAF), select **Permanent install key file** and point to your new PAF.

Step 4 Continue your selections through the installation process and complete by clicking **Finish**.

The installer program automatically finds, downloads, and installs applicable BSPs for you.

Step 5 Restart any license servers after installation (if applicable).

Documentation Updates

Documentation is available at Wind River Online Support, through Workbench help, and also on the start menu on your installation host.

You can also access the documentation locally with an independent help browser as follows:

- Linux:

```
$ installDir/workbench-3.3/x86-linux2/bin/wrhelp.sh &
```

- Windows:

```
$ installDir\workbench-3.x\x86-win32\bin\wrhelp.bat
```

- Or start an infocenter that provides remote access:

```
$ installDir/workbench-3.3/x86-linux2/bin/wrhelp.sh -start &
```

Refer to *Wind River Workbench by Example, Linux Version: Where to Find Information* for details on using **wrhelp.sh**.

You can also use a web browser and browse to the ***.pdf** or **index.html** files in locations under *installDir/docs/extensions/eclipse/plugins/*

To access GNU toolchain documentation from the command line, see the subdirectories under the **share/doc/** directory for your toolchain, for example, under:

Update your installation's documentation to the latest version

You can install the latest documentation using ESD with the **wrInstaller** maintenance tool.

Step 1 From the Workbench main menu, select **Help > Update Wind River Products**.

Step 2 From the command line, run the Maintenance Tool:

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

 **NOTE:** Running the git-based installer also updates the documentation. For more information, see: [Updating Wind River Linux Using the git-based Installer](#) on page 24

Get the Latest Documentation from Wind River Online Support

You can access and download the latest documentation without installing it on your host system.

There are two options to obtain the latest documentation.

OPTION 1: Run the maintenance tool to add the latest released documentation to your installation. For additional information, see [Update your installation's documentation to the latest version](#) on page 30.

OPTION 2 : Get the latest documentation from OLS:

Step 1 Go to the Wind River Online Support site at:

<http://www.windriver.com/support>

Step 2 Log in to Wind River Online Support.

Select **Access Online Support** to login. You will need your Wind River login information to continue.

Step 3 Navigate to the product page.

Select **Products > Current Products > Wind River Linux**.

Step 4 Select the **Manuals** link that aligns with the product version you want to obtain information on.

Creating Platform Project Build Error Reports

When build errors occur, you can create a report and send the results by email to Wind River Customer Support to help identify the problem.

The error reporting tool, disabled by default, allows you to create a report of build errors, and save them locally or submit the information to a central database on an error reporting server. If there is no error server that you want to send the build errors to, you can send the error report manually using email.

When you submit the data to a central error server, you can use the web interface to browse errors from outside the build environment. In addition to browsing the error(s), you can view statistics, and query the database. This includes submitting data to the Yocto Project error reporting server located at <http://errors.yoctoproject.org>. For additional information on the error reporting tool, see <http://www.yoctoproject.org/docs/1.6.1/dev-manual/dev-manual.html#using-the-error-reporting-tool>.

Step 1 Enable error reporting.

You can enable error reporting for a new platform project at configure time, or to an existing platform project.

Options	Description
New platform project	Use the --enable-error-report configure option. For example:
Existing platform project	Edit the projectDir/local.conf file to add the following line: <pre>INHERIT = += "report-error"</pre>

Step 2 Optionally specify a directory for the error report.

By default, the error reporting feature stores information in **projectDir/bitbake_build/tmp/log/error-report**. However, you can specify a directory for the report.

To do so, edit the **projectDir/local.conf** file to add the following line:

```
ERR_REPORT_DIR = "path"
```

In this example, *path* is the location of the error report directory.

Step 3 Build the platform project to generate the report.

```
$ make
```

This command may take some time to run for a new platform project. For project experiencing build errors, it typically completes much quicker.

Once the build completes, the error report is available at **projectDir/bitbake_build/tmp/log/error-report**, or at the location specified in the previous step. For example:

```
$ ls bitbake_build/tmp/log/error-report
error-report.txt
```

The error report feature creates a text file for sending to an error-report server or to use as an email attachment.

Step 4 Optionally send the error report to an error reporting server.

This step requires an error reporting server. This may be a local server, or a community server, such as the one located at <http://errors.yoctoproject.org>. Instructions for setting up your own server are located with the server source code in the README at <http://git.yoctoproject.org/cgit/cgit.cgi/error-report-web/>.

Run the following command from the project directory to send the error report.

Options	Description
Specify a server	<pre>\$ make send-error-report SERVER=server_location</pre> <p>Using this command, you must specify a server location.</p>
Send to default Yocto Project server	<ol style="list-style-type: none">1. Start a BitBake shell. <pre>\$ make bbs</pre>2. Send the error report. <pre>\$ send-error-report error-report_location</pre><p>For example, to send the report created in this procedure:</p><pre>\$ send-error-report projectDir/bitbake_build/tmp/log/error-report/error_report.txt</pre><p>You may also specify a server location with this command, for example:</p><pre>\$ send-error-report error-report_location server_location</pre><p>In this example, <i>server_location</i> is the location of the error reporting server. If you do not specify a location, the error report is sent to the Yocto Project server at http://errors.yoctoproject.org. Note that entries to this database are available to the public.</p>

Step 5 When prompted, enter your name and email address.

The first time the script is run, it prompts you for your name and email address. This information is appended to the report.

Step 6 View the report.

Once the report is received, a link to the newly created error report will be printed to the console, for example:

<http://localhost:8000/Errors/Search/1/96>

Select this link to view information about the build error(s) in a web browser.

2

Run-time Software Configuration and Deployment Workflow

Understanding the general workflow for creating and deploying a complete run-time system provides useful context for specific development decisions.

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 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 79

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

- [Directory Structure for Platform Projects](#) on page 57
 - [Directory Structure](#) on page 35
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:

- [About Configuring a Platform Project Image](#) on page 77
- [About the make Command](#) on page 98

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:

- [About QEMU Targets](#) on page 355
- [About Wind River Simics Targets](#) on page 365
- [About Target-based Networking](#) on page 491

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 199. 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 187.
- Debug applications and the kernel. See [Kernel Debugging](#) on page 327 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 459.

3

Development Environment

Directory Structure	35
Metadata	40
Configuration Files and Platform Projects	41
README Files in the Development Environment	47
Creating Subset Repositories of meta-openembedded Layers	48

Directory Structure

The development environment provides functionality through layers, recipes and templates provided by Wind River Linux and the BitBake build system.

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 57.

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 include any of the following characters:

Character	Description
+	Plus symbol
space	Space
tab	Tab
@	'At' symbol
~	Tilde symbol

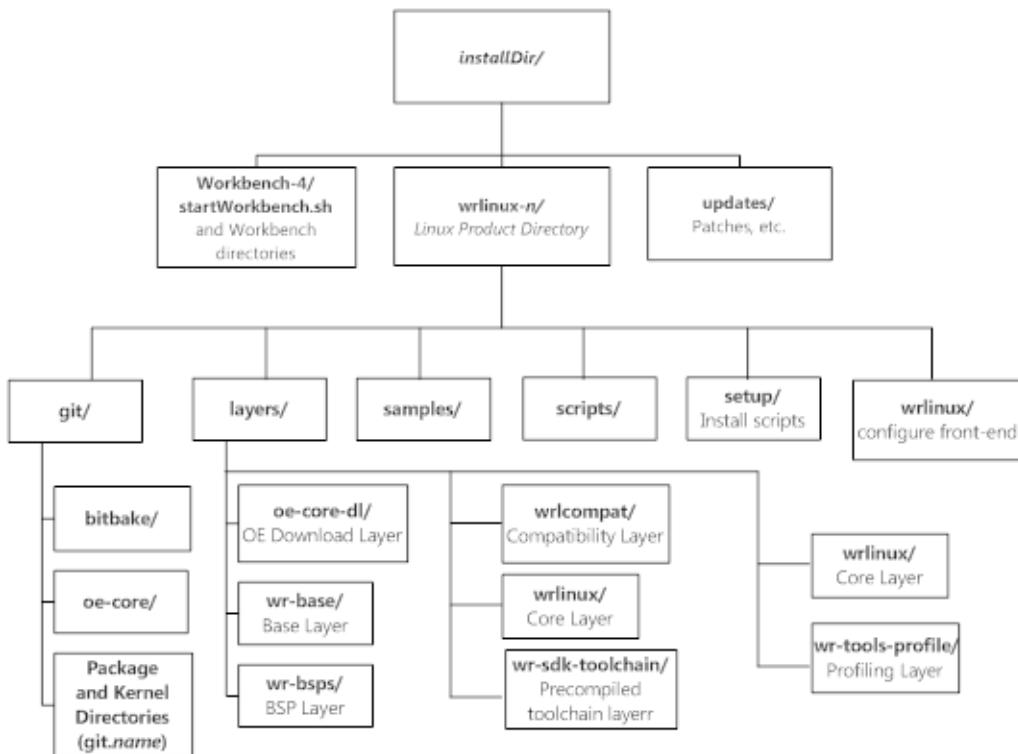
Character	Description
^	Carat symbol
#	Pound symbol

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 57, 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 *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 Workbench.

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

workbench-4/

Contains the Workbench installation and the **startWorkbench.sh** shell script.

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 508 for additional information.

The **updates** Directory

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

The **wrlinux-7** Directory

The **wrlinux-7** directory contains the Wind River Linux development environment, with the contents as shown in and discussed in this section. Sub-directories include:

addons

Though not shown in *Development Environment Directories*, this directory contains the files required for using the Wind River Linux add-on profiles. For additional information, see *Optional Add-on Products* on page 22.

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. 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 57.

layers/

Contains layers required by the development environment. For additional information on layers in general, see *About Layers* on page 151.

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 152 for a breakdown of the layers included with your installation.

scripts

Contains scripts useful for enhancing your development.

Git Subset Repository tool

The `installDir/wrlinux-7/scripts/git-repo-subset/git-repo-subset.sh` script lets you create subset repositories of the **meta-openembedded** upstream repository. For additional information, see [Creating Subset Repositories of meta-openembedded Layers](#) on page 48.

Host Package Installation script

The `host_package_install.sh` script automatically downloads and installs the required development libraries for using Wind River Linux.

Package Importer tool

The `importPackage.sh` script launches a GUI-based tool for adding applications to your platform project image. For additional information, see [About the Package Importer Tool \(import-package\)](#) on page 215.

Platform Project Configure tool

The `project-configure.sh` script launches a GUI-based tool for configuring a platform project. For more information, see: [About the Platform Project Configure Tool](#) on page 96.

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.

moduledbug_3

A sample 3.x kernel module project.

moduledbug_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.

SDK

Contains pre-built SDKs for Windows application development for the following architectures:

- ARM
- ia32
- MIPS
- POWER

These SDKs are ready-to-use directly from the command line, or to be imported into Workbench and are available in the *installDir/wrlinux-7/SDK/wr-sdk-version/arch-quickstart-686-mingw32/arch* directory. For additional information, see [About Using the SDK](#) on page 204.

setup

Contains installation-related scripts that setup the development environment. To use these scripts, run them from the *installDir/setup* directory, for example:

```
$ cd installDir/setup  
$ ./prebuiltck.sh
```

This example runs the **prebuiltck.sh** script, which checks for the validity of the prebuilt items shipped with the product.

postinst_lx_ckpackages.sh

Verifies that the required host packages for using Wind River Linux are installed.

postinst_lx_symlinks.sh

Verifies that the required symlinks for using Wind River Linux are setup .

postinst_wrlinux.bat

For windows-based installations, this verifies the installation directory.

postinst_wrlinux.sh

Checks for the required host packages, and prompts you to install them if necessary.

prebuiltck.sh

Checks for the validity of the md5 checksum for prebuilt items shipped with the product in the *installDir/layers/wr-prebuilt*s directory..

preuninst_wrlinux.bat and **preuninst_wrlinux.sh**

Clears the installation cache and runs the **uninstall_mgr.sh** script.

uninstall_mgr.sh

Runs the tool to uninstall Wind River Linux.

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 80 and [About Configure Options](#) on page 81.

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 41

Recipes (**.bb**) files

See [About Recipes](#) on page 167.

Classes (**.bbclass**) files.

Appends (**.bbappend**) files to existing layers and recipes. See [About Recipes](#) on page 167.

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 [About Configuring a Platform Project Image](#) on page 77, and [About Building Platform Project Images](#) on page 97.
- Custom layers and/or templates with their own configuration, recipes, classes, and append files. [About Layers](#) on page 151.
- 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 209.

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 41
- [Creating a New Layer](#) on page 159
- [Creating a Recipe File](#) on page 170

Configuration Files and Platform Projects

Various **.conf** files 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 160 and [Disabling a Layer](#) on page 161 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(dgetVar('WRL_TOP_BUILD_DIR', True))}"

BBLAYERS = " \
    ${WRL_TOP_BUILD_DIR}/layers/wrlinux \
    ${WRL_TOP_BUILD_DIR}/layers/wrcompat \
    ${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-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-prebuilt \
    ${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-7/wrlinux/configure --enable-board=qemux86-64 --enable-
```

```
rootfs=glibc_small --enable-rm-oldimgs --with-template=feature/debug --enable-jobs=3 --  
enable-parallel-pkgbuilds=3 --enable-reconfig --with-rcpl-version=0000  
  
CONF_VERSION = "1"  
  
#  
# Distribution choice. Normally "wrlinux" but can be customized by  
# the user with --with-custom-distro=<name>  
#  
DISTRO = "wrlinux"  
WRLINUX_RCPLVERSION = ".0"  
  
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:///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.
# exmaple: 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-separated 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_SYSLINKMENU = "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
```

You can use this file to configure your platform project to fit the requirements of your design. The following are some additional variables:

BB_ALLOWED_NETWORKS

This variable specifies a list of trusted hosts from which to download files. A warning will be logged and network access will be denied if network access is attempted by a host that is not on the trusted hosts list. The value of this variable is in the form of a space-separated list.

PREMIRRORS_append

If you are using a minimal installation of the Wind River Linux product, then this variable will include additional source mirrors as locations to fetch the required packages. During the build process, the tool performs additional downloads since only the necessary configuration and recipes were installed.

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 71) that is generated when you build a platform project.

```

#
# Copyright (C) 2013 Wind River Systems, Inc.
#

BBPATH ?= ""
# 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}/meta-virtualization/recipes*/**/*.bb \
${LAYERDIR}/meta-virtualization/recipes*/**/*.bbappend \
${LAYERDIR}/recipes-*/*/*.bbappend \
${LAYERDIR}/classes/*/*.bbclass"

BBFILE_COLLECTIONS += "wr-kernel"
BBFILE_PATTERN_wr-kernel := "^${LAYERDIR}/"
BBFILE_PRIORITY_wr-kernel = "6"

BBMASK ?= "/(linux-yocto)"

PREFERRED_PROVIDER_virtual/kernel = "linux-windriver"
PREFERRED_VERSION_linux-windriver ?= "3.14%"

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

# This should only be incremented on significant changes that will
# cause compatibility issues with other layers
LAYERVERSION_wr-kernel = "1"

LAYERDEPENDS_wr-kernel = "core wr-userspace-base networking-layer"

```

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 specified with this variable.

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 156), specifies the BSP configuration for your platform project image.

Assigning Empty Values in BitBake Configuration Files

Assigning an empty value in a BitBake platform project **.conf** file requires a space between the declaration's double quotes.

Assigning an empty value in a platform project **.conf** file is different in BitBake than it is in 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 projectDir :
Edit local.conf file	<ol style="list-style-type: none">1. Open the projectDir/local.conf 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
```

Step 3 Retrieve the empty string.

Run the following commands from the *projectDir/bitbake_build* directory.

```
$ make bbs  
$ 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/examples/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-7/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-7/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-7/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-7/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.

For Wind River Linux 7.0, you must navigate to the top-level installation directory.

```
$ cd installDir/wrlinux-7
```

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.

Creating Subset Repositories of meta-openembedded Layers

Wind River Linux provides a script, `git-repo-subset.sh`, to clone specific layers from the **meta-openembedded** core development git repository branch.

The upstream **meta-openembedded** git repository located at <http://git.openembedded.org/meta-openembedded/tree> includes thousands of development objects, including layers, recipes, packages, and source material. The layers in this repository contain useful applications for developing your platform project image, in addition to the layers included with your Wind River Linux installation.

While you could just copy the contents of a layer to "clone-and-own" the recipe, it can be helpful to make a git repository of a subset of the upstream. The subset script lets you pick only what you need from the upstream layer while keeping the git history and upstream commit IDs. At a later

time, when there are upstream changes that you wish to use in your project, you can again use this script to regenerate the subset of the layer.

An example of the upstream repository structure is as follows:

```
$ ls -L 1 /path-to/meta-openembedded
+
├── contrib
├── COPYING.MIT
├── meta-efl
├── meta-filesystems
├── meta-gnome
├── meta-gpe
├── meta-initramfs
├── meta-multimedia
├── meta-networking
├── meta-oe
├── meta-perl
├── meta-python
├── meta-ruby
├── meta-systemd
├── meta-webserver
└── meta-xfce
├── README
└── toolchain-layer
```

Note that each directory that begins with **meta-** or the **toolchain-layer** is an actual Open Embedded layer, with the required structure and files for use with the Wind River Linux build system, based on the Yocto Project.

Wind River Linux includes the entire content from several layers and subsets of other layers using this script. Please examine these layers, and if you wish to create additional layers, you can use this script. Some caution should be exercised to avoid adding libraries that interact with existing packages through the **PACKAGECONFIG** BitBake syntax, since these interactions have not been confirmed to work well by Wind River Linux developers.

The script, **git-repo-subset.sh**, located at *installDir/wrlinux-7/scripts/git-repo-subset/*, can help you retrieve only the repository content you need. It runs with the following options:

```
git-repo-subset.sh -OPTIONS repository_location subset_name
```

OPTIONS

For a list of script options, see [Repository Subset Script Options Reference](#) on page 53.

repository_location

This represents the path to the meta-openembedded repository, either at a Web location or local path, that you want to clone.

subset_name

The name of the generated subset repository or branch.

For examples of using the **git-repo-subset.sh** script, see [Using the Repository Subset Script](#) on page 50.

It is important to note that the script is designed to clone valid BitBake or Open Embedded layer repositories that include recipes. While layers may also include configuration files, the script will not process them.

Because the script creates subsets of active git repositories, it is also possible to share your work with the community by pushing changes back to the development community at your discretion.

Using the Repository Subset Script

Use the `git-repo-subset.sh` script to create subsets of the **meta-openembedded** repository in your development environment.

To perform the script examples, you will need a valid Wind River Linux installation that includes the `git-repo-subset.sh` script.

- Step 1** Navigate to a location on the host system to clone the subset repository or files to.

In the following examples, the path to the `git-repo-subset.sh` script is removed for convenience.

Options	Description
Create a repository from a meta-openembedded layer	This example creates a local repository from the meta-openembedded/meta-xfce layer, a layer not included with your Wind River Linux installation.. 1. Run the following command: <pre>\$ git-repo-subset.sh -s meta-xfce -w my_workdir git://git.openembedded.org/meta-openembedded meta-xfce</pre> Output will display as the command runs. Once the command completes, a new repository named meta-xfce.git , that includes only the meta-xfce layer will be created at the location specified as <i>my_workdir</i> . To create a branch, instead of a new repository, use the -b option. For example: <pre>\$ git-repo-subset.sh -b -s meta-xfce -w my_workdir git://git.openembedded.org/meta-openembedded meta-xfce</pre> 2. View the contents of the repository: <pre>\$ tree -L 1 my_workdir/meta-xfce └── conf ├── COPYING.MIT ├── README ├── recipes-apps ├── recipes-art ├── recipes-bindings ├── recipes-multimedia ├── recipes-panel-plugins └── recipes-thunar-plugins └── recipes-xfce</pre>

Notice that it includes the contents of the **meta-openembedded/meta-xfce** repository, which is only a subset of the actual repository.

Options	Description
Create a repository with a single recipe, located in a sub-folder of the meta-open-embedded repository	<p>This example creates a local repository from the meta-openembedded/meta-oe layer, that includes a single recipe for the meta-oe/recipes-benchmark/iperf package, which is not included with your Wind River Linux installation..</p> <ol style="list-style-type: none"> 1. Create a filter-list file. <p>Because your Wind River Linux installation also includes the meta-oe layer, it is possible to have competing git repositories with the same content. This may cause collision issues with git pulls, and possible file corruption. To keep this from happening, create a filter-list text file that contains the paths to the content you wish to add to your local subset repository. For the meta-oe/recipes-benchmark/iperf content, the filter-list file contains the following content:</p> <pre style="background-color: #f0f0f0; padding: 10px;">meta-oe/classes/ meta-oe/conf/ meta-oe/COPYING.MIT meta-oe/licenses/ meta-oe/README meta-oe/site/ meta-oe/recipes-benchmark/iperf/</pre>

Save the file as **filter-list-iperf.txt**.

2. Run the following command:

```
$ git-repo-subset.sh -s meta-oe -w my_workdir -l path_to/
filter-list-iperf.txt git://git.openembedded.org/meta-
openembedded meta-oe-mysubset
```

Once the command completes, a new repository named **meta-oe-mysubset.git**, that includes only the contents specified in the **filter-list-iperf.txt** file.

3. View the contents of the repository:

```
$ tree -L 1 my_workdir/meta-oe
├── classes
├── conf
├── COPYING.MIT
├── licenses
├── README
└── recipes-benchmark
```

Notice that it includes the contents of the **filter-list-iperf.txt**.

Options	Description
Create a repository from recipes in different layers in the meta-openembedded repository	<p>To create a repository from recipes in different layers, the layers themselves must be added to preserve the recipe content and usage. As a result, the -s (sub-directory) option must not be used.</p> <p>1. Create a filter-list file.</p> <p>In this example, you will create a repository with content from three different layers within the meta-openembedded repository. Use the following content to create the filter-list file:</p> <pre>meta-oe/classes/ meta-oe/conf/ meta-oe/COPYING.MIT meta-oe/licenses/ meta-oe/README meta-oe/site/ meta-oe/recipes-benchmark/iperf/ meta-perl/conf/ meta-perl/COPYING.MIT meta-perl/README meta-perl/recipes-perl/libtest/ meta-ruby/</pre> <p>Save the file as filter-list-multi-layers.txt.</p> <p>2. Run the following command:</p> <pre>\$ git-repo-subset.sh -w my_workdir -l path_to/filter-list- multi-layers.txt git://git.openembedded.org/meta- openembedded meta-openembedded-mysubset</pre> <p>Once the command completes, a new repository named meta-openembedded-mysubset.git, that includes only the layer contents specified in the filter-list-multi-layers.txt file.</p> <p>3. View the contents of the repository:</p> <pre>\$ tree -L 1 my_workdir/meta-openembedded-mysubset └── meta-oe ├── meta-perl └── meta-ruby</pre> <p>Notice that it includes the layers and contents specified in the filter-list-multi-layers.txt file.</p>

Step 2 Use the repository layer with a platform project.

To include your new repository layer in a platform project build, use the **--with-layer= configure** script option.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glIBC_small \
--with-layer=my_workdir/meta-oe
```

In this example, the repository with the layer is located at **my_workdir/meta-oe**. If the repository has multiple layers, you will need to specify the path to the valid layer, for example:

```
--with-layer=my_workdir/meta-openembedded-mysubset/meta-ruby
```

Step 3 Build the platform project.

```
$ make
```

Once the command completes, the contents of the layer will be part of your platform project image.

Repository Subset Script Options Reference

The **configure** script can be run with a large number of options.

You can display a complete option and syntax list with the following command:

```
$ ./installDir/wrlinux-7/scripts/git-repo-subset./git-repo-subset.sh -h
```

Option	Description
-b	Clone a branch only, do not create a new repository.
-h	Display script help text.
-l	Use to specify a filter list, which is a text file that lists only the objects that you want to add to a repository. Because your Wind River Linux installation also includes subsets from the meta-openembedded repository, filter lists are necessary to ensure that there is no collision between competing repositories, which may cause file corruption.
-s	Use to specify a sub-directory in the meta-openembedded repository. The content in the specified subdirectory will split out into the root directory of the repository created by the script.
-w	Specify the work directory to clone and create a subset repository from.

4

Build Environment

[About the Project Directory](#) 55

[Creating a Project Directory](#) 56

[Directory Structure for Platform Projects](#) 57

[About README Files in the Build Environment](#) 72

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*.

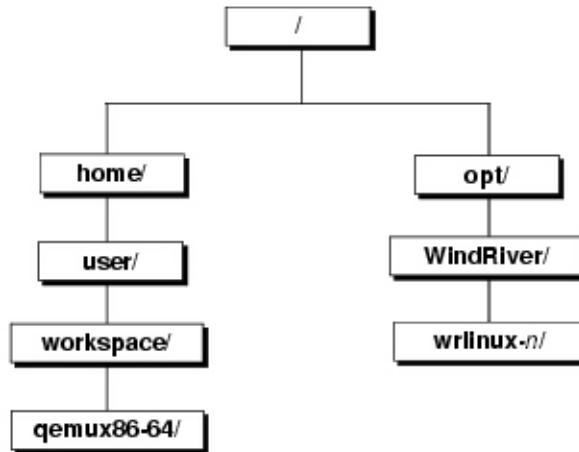
The main reason for this is that it is possible to corrupt your development environment by running the `configure` script from inside it (see [About the Configure Script](#) on page 80).



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
```

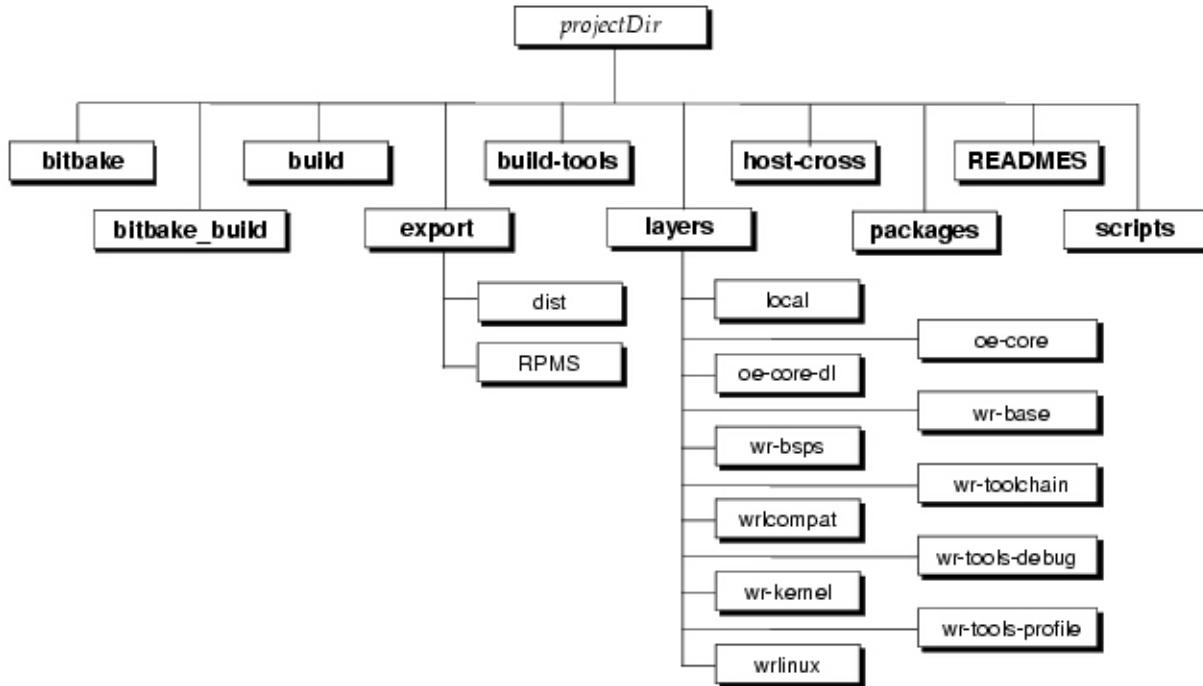
Options	Description
Workbench	If you are using Workbench, the project directory is created for you when you click Finish in the configuration wizard. It is created as a folder in your workspace folder (by default, <code>\$HOME/WindRiver/workspace/</code>) with the name you assign and a <code>_prj</code>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

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.

build-tools

A directory that hosts soft links to development tool directories for native packages.

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-qemux86-64-datetimeStamp.tgz

A compressed tar file containing the Linux kernel modules. A soft link to this file named **modules-qemux86-64.tgz** is created for your convenience.

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 71.

layers/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**.

layers/meta-webserver

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

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-7/git` directory in the installation development environment. See [Directory Structure](#) on page 35 .

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.

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 80.

Additionally, this layer contains templates for adding additional features to your platform project image. See [Feature Templates in the Project Directory](#) on page 61 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 67 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-version

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 80. Additionally, this layer contains templates for adding additional features to your platform project image. See [Feature Templates in the Project Directory](#) on page 61.

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 67.

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 533.

rsim

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

layers/wr-sdk-toolchain

Precompiled toolchain layer 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 61.

layers/wr-tools-profile

Contains configuration information and files to support Wind River Linux development tools, including: analysis, boot time, code coverage, Valgrind, and Itng. You can add these features to your platform project using templates. See *Feature Templates in the Project Directory* on page 61.

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/layers</i> sub-directory	Sub-directory contents
wr-base/templates/feature/	
acl	Adds the Linux acl (Access Control Lists) as a distribution feature to the platform project image.
archiver	Adds an additional step to recipe building using the ARCHIVER_MODE option in the <i>projectDir/local.conf</i> or another recipe file. This provides the ability to create an archive of the sources of a given package and the recipe file or files associated with it.
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.
installer-support	Provides support for creating an installable version of a Wind River Linux platform project image. For additional information, see <i>Configuring and Building the Host Install</i> on page 387.
ipv6	Configuration file for enabling ipv6.
LAMP	Appends the following images to the build: mysql5 , modphp , phpmyadmin , apache2 , and xdebug
lsbtesting	Prepares the platform project target image to run the LSB Distribution Checker for LSB testing. For additional information, see <i>About the LSB Tests</i> on page 433.
mysql	Adds the mysql5 package.

<i>projectDir/layers sub-directory</i>	Sub-directory contents
	mysql-odbc
	Adds packages necessary to use MySQL over ODBC.
	package-management
	This template adds the package-management feature to install package management tools and preserve the package manager database on a target for platform project images with root file systems that do not support the PR server by default, such as glibc-small and glibc-core .
	Alternately, you can add PR Server support using the following configure options:
	glibc-small
	--enable-rootfs=glibc-small/package-management
	glibc-core
	--enable-rootfs=glibc-core/package-management
	For additional information, see Package Management Configure Option .
	ptest
	Adds ptest (package test) functionality for validating ptest-enabled packages on your platform project image.
	readonly-root
	Use to enforce a readonly-root target file system for your platform project target image. For additional information, see Enforcing Read-only Root Target File Systems on page 400.
	self-hosted
	Adds development packages to allow on-target development, creating a self-hosted image with the ability to build applications, kernels, and platform projects on itself.
	target-toolchain
	Adds binutils , the gcc compiler, and the core development libraries libc-dev and libgcc-dev to enable development on the target.
	This option provides a smaller development footprint than feature/self-hosted for on-target development.
	Supports x86 and x86-64 targets only.
	test
	Adds a collection of tests from the Linux Test Project (LTP) package.

<i>projectDir/layers sub-directory</i>	Sub-directory contents
wr-kernel/templates/feature/	custom-kernel Prepares the platform project to use a custom kernel, specified as linux-windriver-custom , for use with Wind River Linux make targets.
	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-dev Prepares the platform project for kernel development, such as using a custom kernel as described in <i>About Optimized Custom Kernel Builds</i> .
	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 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.
	lttng2 Configuration file to configure the kernel for lttng2 and disable ltt .

<i>projectDir/layers sub-directory</i>	Sub-directory contents
	lxc Provides a set of tool for managing Linux kernel containers (lxc) and cgroups on a platform project root file system image.
	msa Enables the Microstate Accounting kernel feature.
	nfc Provides support for developing target platforms for hardware, such as smart phones, that is considered a near fielded communication (nfc) device. This includes data exchanges without contact on supported devices.
	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.
	yaffs2-utils Provides a set of tool for managing YAFFS2 (Yet another flash file system) system images.
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. libwrsqtxhelper Adds a library of Qt object wrapper functions that can help display Qt objects that are otherwise opaque on Qt 4, X-11-based systems, when used with Wind River Workbench.

<i>projectDir/layers sub-directory</i>	Sub-directory contents
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	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).
wrlcompat/templates/feature/	
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/deploy/tar) to be used or copied over to other directories.

<code>projectDir/layers</code> sub-directory	Sub-directory contents
	image-manifest
	Creates a text file with a manifest of the basic contents of the platform project image, named after the image name, for example: <code>projectDir/export/images/wrlinux-image-glibc-std-qemux86-64-date.rootfs.manifest</code> . This file is used to compare archived images against a new build that uses the same configuration as the archived image, to verify that the contents of each build match.
	Once a platform project image is created using this template, you can use the <code>projectDir/layers/wrlcompat/scripts/wrl-audit-image.py</code> script to compare image files. For additional information, see About Verifying Builds with the Image Manifest on page 126.
	ip-report
	Adds an IP report to the BitBake parser environment.
	package-report
	Adds a package report to the BitBake parser environment

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 to the board/target-specific system kernel being built.

Currently, Wind River Linux 7.0 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

Configuration Fragment Name	Description
<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
<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

Configuration Fragment Name	Description
<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
<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-e1xxxx/intel-e100.scc</code>	Enable Intel E100 and E1000 support
<code>features/ipmi/ipmi.scc</code>	Enable core ipmi support
<code>features/iwlagn/iwlagn.scc</code>	Enable iwlagn 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

Configuration Fragment Name	Description
<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
<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 173 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.

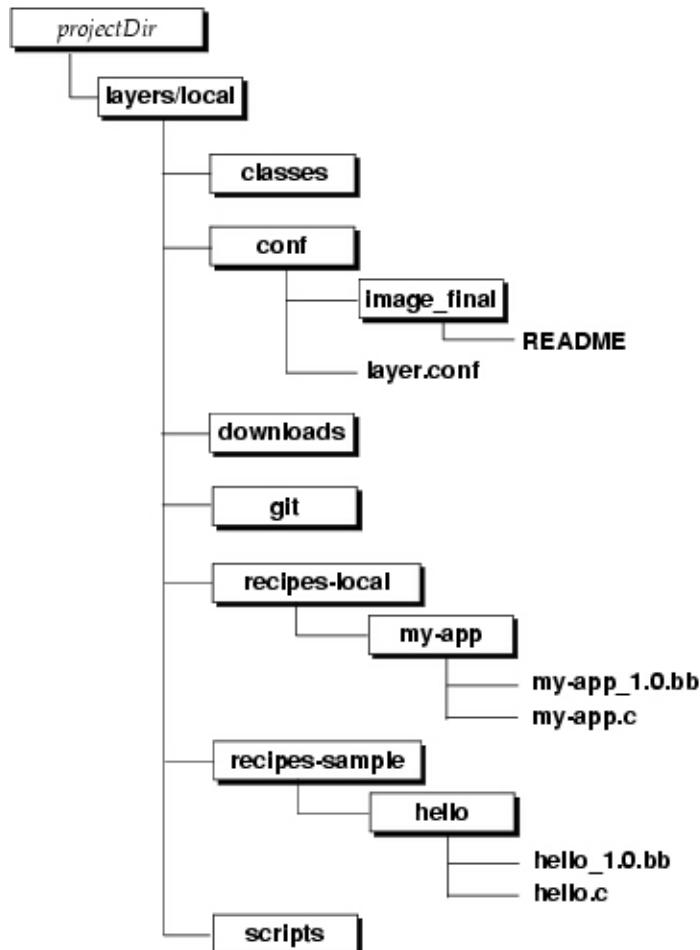
Click Add to the right of the **Templates** field. A list of available templates will display. Select a template in the list to view its description.

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** directory 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 215.

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 35
- [About Layers](#) on page 151
- [Creating a New Layer](#) on page 159

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 170.
- 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 159.
- 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 187

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

See also:

[Directory Structure for Platform Projects](#) on page 57

[Directory Structure](#) on page 35

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-7/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.....	77
Localization.....	143
Portability.....	147
Layers.....	151
Recipes.....	167
Templates.....	173
Finalizing the File System Layout with changelist.xml....	177

5

Configuration and Build

About Configuring a Platform Project Image	77
About Building Platform Project Images	97
About Managing Builds with the Yocto Project Toaster	102
Build-Time Optimizations	109
About Accelerating Builds with a Remote Shared State Cache Server	110
Securing Shared State Cache	114
Examples of Configuring and Building	122
About Verifying Builds with the Image Manifest	126
About Maintaining Open Source License Compliance	128
About Creating Custom Configurations Using rootfs.cfg	132
Glibc File Systems	136

About Configuring a Platform Project Image

Before creating a platform project image you must configure the platform project to match your target system's hardware and software requirements.

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 82, 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](#) on page 97.

Before You Begin

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 57
- [Run-time Software Configuration and Deployment Workflow](#) on page 33

Design Benefits

The design of the Yocto Project BitBake build system with enhanced Wind River Linux configuration tools 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 57 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 71
- By using custom layers and templates (see [About Layers](#) on page 151 and [About Templates](#) on page 173, 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 80. 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-LineCreating and Configuring a Platform Project*, for information on setting these variables.

For information on the project directory, see

- [About the Project Directory](#) on page 55
- [Directory Structure for Platform Projects](#) on page 57

Initializing the Wind River Linux Environment

Initializing the development environment and creating the ***WIND_LINUX_CONFIGURE*** environment variable simplifies 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-7
```

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 as 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-7/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-7/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 122 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 a **.** (period) in your **PATH** environment variable. In addition to being a security issue, having a period in your **PATH** can cause problems with the build. Remove any periods 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 79.

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=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 109.

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](#) on page 97 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](#) on page 97.



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 122 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

Many options to the **configure** script are available to customize your Wind River Linux projects.

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

Option	Description
--enable-rootfs= <i>rootfs</i>	Specifies the file system. This option is equivalent to specifying --with-template=rootfs/rootfs 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
To create a specific boot image: --enable-bootimage= <i>option1,option2...</i>	Specifies the creation of a boot image using a comma-separated list containing one or more of the following image types: ext3 , ext4 , hdd , iso , jffs2 , tar.gz , tar.bz2 , ubifs , vmdk , and -tar.bz2 .
To create u-boot images without compression: --enable-bootimage= <i>imageType1.u-boot</i>	Note that tar.bz2 is the default image type. To disable this, you must select -tar.bz2 , or use tar.gz to set it as the default.
To create u-boot images with specified compression: --enable-bootimage= <i>imageType1.compressionType1.u-boot</i>	The vmdk option provides a bootable image for a virtual machine (VM) target, such as VMWare or Virtual Box. For additional information, see About Deploying an Image with a Virtual Machine Manager on page 416.
	To create u-boot images, you specify the image and compression type from the following options: <i>imageType</i> cpio , ext2 , ext3 , or ext4 <i>compressionType</i> gz , bz2 , lzma , or leave blank
	For example: --enable-bootimage=ext4.bz2.u-boot creates a u-boot image with an ext4 root file system with bz2 compression. To run a u-boot image on a target, certain kernel options must be set. For additional information, see Creating u-boot RAM Disk Images on page 392. Note that after the build completes, you must run the make usb-image or make iso-image command to actually create a

Option	Description
	bootable image. Once complete, the image resides in the <code>projectDir/export/</code> directory. See About Configuring and Building Bootable Targets on page 383.
<code>--enable-build=[debug production productiondebug profiling]</code>	To specify the amount of free space that you want your bootimage to have in kilobytes, use the <code>--with-bootimage-space= configure</code> option.
	When building or rebuilding a platform project (<code>make</code>), use the following options to specify your build optimization:
	debug Use to compile and install binaries and libraries with debugging information (<code>-g</code>). 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: <code>-O0 -fno-omit-frame-pointer</code>
	The platform build produces a single file system image that includes <code>feature/dbg</code> template packages.
	The debug items are installed into a <code>.debug</code> subdirectory. For example, the <code>/bin/bash</code> debug file is located in <code>/bin/.debug/bash</code> , with corresponding sources in
	<code>/usr/src/debug/bash</code>
	production
	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.
	Subsequently building the <code>make fs-debug</code> target will produce an additional file system image with <code>debuginfo</code> in the filename, containing only debug information

Option	Description
	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 flow may be reordered, and only limited stack tracing information is available.
productiondebug	
	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.
profiling	
	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:
	<ul style="list-style-type: none">• -fno-omit-frame-pointer• -fvisibility=default
--enable-build-tools=[auto yes]	auto or default
	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 .

Option	Description
	yes
	<p>Use this option to skip the sanity check and force a build tools installation.</p>
--enable-ccache=[yes no]	<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 scenario. Your development workstation must have ccache installed.</p>
	<p>For additional information on using ccache, see the <i>Wind River Linux Getting Started Guide: Configuring a Platform Project</i>.</p>
--enable-checkout-all-layers=[yes no] option	<p>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.</p>

Option	Description
<code>--enable-doc-pages=target</code>	<p>The specified path must be an absolute path, and not a relative path as compared to the build directory and tree.</p> <p>For additional information, see Enabling Package Build History on page 253.</p>
<code>--enable-dedicated-prserver=[yes no]</code>	<p>Use this option to include the documentation man/info pages in the file system of the target.</p> <p>NOTE: This option only applies to the glibc_std file system.</p>
<code>--enable-error-report=[yes no]</code>	<p>Use this option to create a dedicated build that acts only as a shared PR service, for build acceleration purposes.</p> <p>NOTE: Since this option skips certain checks during configure, a project built and deployed with this feature is not supported for building Wind River Linux images.</p>
<code>--enable-internet-download=[yes no]</code>	<p>Use to enable error reporting in your platform project build. For additional information, see Creating Platform Project Build Error Reports on page 30.</p>
<code>--enable-jobs= number</code>	<p>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.</p> <p>For additional information, see About Obtaining Package Source not Provided by Wind River on page 192.</p>
	<p>Specifies the maximum number of parallel jobs that make should perform. This should be set to the number of CPUs your system has available.</p> <p>Note that this setting is optional, as the BitBake build system automatically sets this value depending on your host system requirements.</p>

Option	Description
--enable-ldconfig=[yes no]	<p>This option places a post-installation script in RPMs to update the ldconfig cache during installation, and also generates the <code>etc/ld.so.conf</code> and <code>etc/ld.so.conf.d</code> files on the image at the system level.</p> <p>For additional information, see About ldconfig on page 375.</p>
--enable-multilib=[lib32 lib64]	<p>Sets the default multilib for the image.</p> <p>lib32</p>
	<p>Use to enable 32-bit libraries if you require them for development on a 64-bit target platform. By adding this option, when you generate an SDK, it will include both 64- and 32-bit libraries.</p>
	<p>lib64</p>
	<p>Use to enable 64-bit libraries if you require them for development on a 32-bit target platform. By adding this option, when you generate an SDK, it will include both 32- and 64-bit libraries.</p>
--enable-package-manager=[rpm ipk opk]	<p>Specifies the package type used by the root file system. If not specified using this option, the default package type is <code>rpm</code>.</p>
--enable-parallel-pkgbuilds= number	<p>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 <code>configure</code> option above sets</p>
--enable-prelink=[yes no]	<p>Pre-links target binaries and libraries. If unused, defaults to yes.</p>
--enable-prserver=[yes local hostname:port no]	<p>Sets the <code>PRSERV_HOST</code> variable in the <code>projectDir/local.conf</code> file to enable a PR service for monitoring package revisions.</p>
	<p>yes, local or not specified</p> <p>Sets the value to <code>localhost:0</code> to use the local BitBake PR service.</p>

Option	Description
hostname:port	If you set a shared PR service for your development environment, you would specify the hostname or IP address and port, for example: 192.256.0.2:82.
	NOTE: There are limitations when using shared PR server. For additional information, see <i>About Package Revision Management</i> on page 251.
no	Use this option to disable the PR service.
--enable-reconfig	Enables the configure script to run with different or added options on a previously configured platform project.
--enable-rm-oldimgs=[yes no]	Removes old root file system files located in the projectDir/bitbake_build/tmp/deploy/images directory, and retains only the latest image. Each time you run make 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.
--enable-rm-work=[yes no]	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

Option	Description
	you consider a few packages in the glibc-std build take between 200MB to 300MB each to compile.
--enable-rpm-prefer-elf-arch=[1 2 3]	<p>For use with a multilib (32- and 64-bit userspace applications) systems, this sets the RPM_PREFER_ELF_ARCH option in the projectDir/local.conf file to configure the preferred ABI when an RPM-based root file system backend detects an ELF file conflict.</p> <p>A conflict occurs when two or more packages try to install an ELF file to the same path. If this value is not set, the RPM root file system backend will try to use heuristics to determine the ABI of each package if there is a conflict.</p> <p>Options include:</p>
1	ELF32 configuration resolves conflict.
2	ELF64 configuration resolves conflict.
3	ELF64 N32 configuration resolves conflict.
	Use for mips64 and mips64el BSPs only
--enable-scalable-mklibs=[yes no]	Specifies whether target binaries are to be optimized by removing some unused functions.
--enable-sdkimage-staticlibs=[yes no]	Use to install static libraries in your SDK images. By default, static libraries are not installed.
--enable-sdkmachine=[auto i686 x86_64 i686-mingw32 x86_64-mingw32]	<p>Use to specify the SDKs that will be created when you run make export-sdk in the projectDir once the platform project is configured.</p> <p>To specify more than one architecture, use commas with no spaces, for example:</p> <p>--enable-sdkmachine=auto, i686,x86_64-mingw32</p>

Option	Description
auto	The default setting if no options are specified. Generates an SDK based on the build system. For example, if the build system is x86_64, then the target will be.
i686	Generates an SDK for a 32-bit Linux host.
x86_64	Generates an SDK for a 64-bit Linux host.
i686-mingw32	Generates an SDK for a 32-bit Windows host.
x86_64-mingw32	Generates an SDK for a 64-bit Windows host.
--enable-stand-alone-project=[yes no]	Creates a platform project that is completely stand-alone, and not dependant on the Wind River Linux installation (<i>installDir</i>) for project build and development. 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 147.
--enable-target-installer=[yes no]	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..
NOTE: Wind River Linux only supports *.ext3 root file system files for the target installer feature.	

Option	Description
	When you use this configure option, you must also use the --enable-rootfs=wr-installer option.
	If you are building a platform project to be used as a reference for the --with-installer-target-build option, you must append the --enable-rootfs= option with +installer-support for that platform project build. For example:
--enable-rootfs=glibc-std+installer-support	
	This ensures root file system support for the target installer.
--enable-test=[yes no]	Includes the standard suite of test packages for the file system and kernel.
--enable-toaster=[yes no]	This includes the Wind River Linux integration of the Yocto Toaster in the project, enabling use of the Toaster web interface to view build data.
--enable-win-sdk=[yes no]	Generate an SDK that can be used on a Windows host to generate target binaries (standalone or through Workbench) in addition to the standard Linux SDK. For additional information, see Exporting the SDK for Windows Application Development on page 202.
--help	Prints an option summary similar to this table and exits without creating a project.
--with-bootimage-space=Additional_free_space_in_kilobytes	Use this option to define the amount of additional free space in kilobytes the bootimage created with the --enable-bootimage= configure option will have. If you do not specify an amount, the default is zero.
--with-doc-compress=[xz gz bz2]	Use this option to specify compression settings for documentation, such as info and man pages, on the target. For additional information, see About Compressing Documentation on Targets on page 379.
--with-dl-dir=directoryPath	When using a minimal install, specifies the directory in which to store the downloaded packages. This option

Option	Description
	updates the BitBake variable DL_DIR in the projectDir/local.conf file. The default directory is installDir/cached_downloads . However, if write permissions are not set up properly in the installDir , then the default directory is projectDir/bitbake_build/downloads . Or, you can use this option to set a different directory than the default.
--with-init=[sysvinit systemd]	Use this option to replace the default systemd with sysvinit as a system and service manager for your platform project image.
	NOTE: The systemd system and service manager may not be compatible with all platform builds. If you experience issues, use the sysvinit manager.
--with-installer-target-build= <i>full_path_to_target_system_*.ext3_file</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-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>licenseFlagType1, licenseFlagType2, licenseFlagType3...licenseTypeN</i>	Use this option to set a comma-separated list of license flag types that are included automatically in the

Option	Description
--with-rcpl-version= 000x	<p>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. For additional information, see <i>About Obtaining Package Source not Provided by Wind River</i> on page 192.</p>
--with-ro-sstate-mirror=uri1,uri2,uri3...	<p>Once you have updated the product, new platform projects default to the latest installed RCPL release when you run configure script commands.</p> <p>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.</p>
--with-sstate-dir= directoryPath	<p>Use this option to set a comma-separated list of locations where a read-only shared cache mirror is available for use in accelerating builds. For additional information, see <i>Accelerating Builds with Shared State Cache</i> on page 113.</p>
--with-template=feature/sstate_signature --with-sstate-sign-format= [gpg md5sum sha1sum sha256sum]	<p>Specify a directory for the sscache files for a remote sstate server at project configuration. For additional information, see <i>Setting Up a Remote Shared State Cache Server</i> on page 111.</p>
--with-sstate-sign-mode= [sign_only verify_only sign_verify disable]	<p>Provides a signature option.</p>
--with-sstate-check-level= [error warn]	<p>Selects the encryption format. md5sum is the default.</p>
	<p>Mode signing options. For example, <i>sign_verify</i> means it must be verified and signed before being allowed. <i>sign_verify</i> is used by default</p> <p>Sets the security level, which is set to error by default.</p> <p>error</p> <p>If a signature file is not available or a signature verification failed, it issues an error and halts the build.</p>

Option	Description
	warn
--with-sstate-secret-key= <i>Secret_Key_Path</i>	If a signature file is not available or the signature verification failed, it issues a warning and continues the build.
--with-sstate-gpg-passphrase= <i>Password</i>	The location in the file system to the secret key path, which is typically a directory where the secret key is maintained.
--with-sstate-public-key= <i>Public_Key_Path</i>	The secret key and its passphrase (password) is used for the signature of shared state cache items. It is combined with --with-sstate-sign-format=gpg and --with-sstate-sign-mode=[sign_only sign_verify] .
--with-template= <i>template1,template2,template3...</i>	The public key is used to verify shared state cache items. It is combined with the --with-sstate-sign-format=gpg and with --with-sstate-sign-mode=[verify_only sign_verify] . Appends the specified templates to the usual template list created by the configure options.
--with-template=feature/package-management	This template adds the package-management feature to install package management tools and preserve the package manager database on a target. Alternately, it can be added as the following configuration option: --enable-rootfs=glibc-[small core]/package-management It is used because package-management is not part of glibc-small or glibc-core by default.
--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.)

Option	Description
--without-layer= <i>layer1,layer2,layer3...</i>	Specifies the layers to exclude from the generated bblayer.conf file.
--with-urlmap= <i>locationOfKernelSource</i>	Specifies the URL from which to download the kernel source code.

About the Platform Project Configure Tool

The Platform Project Configure tool is a GUI-based alternative to the **configure** script that also provides for the entire platform project configuration process.

Like the **configure** script, the Platform Project Configure tool populates the project directory in which it is run with a subdirectory structure 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.

To configure a platform project using the **Platform Project Configure** tool, you create a project directory and then launch the tool from that directory. The project is always created in the directory from which you run the tool.

The Platform Project Configure dialog box appears when the tool starts. Use it to configure all required and optional parts of your platform project. At a minimum, you must specify the board, kernel, and file system for the project.

You can configure the build type by selecting from the **Build:** list box.

Check the **Enable reconfigure** check box to enable reconfiguration of the project.

You can also add layers, templates, and options. Click on the associated buttons to select from the layers, templates, and options. The **Configure Command:** text box shows the analogous command line that you would use with the **configure** script.

Note that when adding a layer, the **Rescan Layers** button in the Platform Project Configure dialog box turns red. Click the button after adding layers to rescan the layers configuration before clicking **Finish**.

When you click on the **Add Options** button, notice that the option string appears in the text box, help appears in the description box, and, if there is a default value, it appears below the description box. For information about the configuration options, see: [Configure Options Reference](#) on page 82.

Click on the **Add template** button to add kernel and rootfs templates. When you add a template, the selections you make for the kernel and rootfs are shown in the **Configure Options:** box. If you see you have selected a wrong option, you can click the **Add template** button again and deselect the option and choose another. The **Configure Options:** box will reflect the changes in options. For more information about templates, see: [About Templates](#) on page 173.

For information about layers, see: [About Layers](#) on page 151.

Using the Parse Configure Command

Use the **Parse Configure Command** dialog from the **Platform Project Configure** tool to quickly change a platform project.

Step 1 Access the **Platform Project Configure** tool from Workbench or from the command-line.

For more information about the **Platform Project Configure** tool, see: *Wind River Linux 7 Command-Line Tutorials* and *Wind River Linux 7 Workbench Tutorials*.

Step 2 Select **File Parse Configure Command** in the Platform Project Configure dialog box.

The Parse Configure Command dialog box appears.

You can apply configure commands or command fragments to the current platform project or to a selected platform project.

Options	Description
Run commands on the current configuration	Use the Parse Configure Command dialog box to enter configure commands to apply to the current configuration.
Run commands on a selected configuration	Click the Browse button to locate the project directory and the configure.log file for the project that you want to work with.

About Building Platform Project Images

The **make** command and build logs are important resources when building platform project images.

The **make** command

After you have configured a project as described in [About Configuring a Platform Project Image](#) on page 77, 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.

If you are using a minimal install, package sources based on your platform project's configuration are downloaded during build time. You can download the package sources before starting the build process. This is useful for running a build in an offline environment. The following lists the different Makefile targets for package fetching:

- **make fetchall**
- **make world.fetchall**
- **make universe.fetchall**

For detailed information, refer to [Common make Command Target Reference](#) on page 513.

When you run **make**, it builds (or rebuilds) the platform project using the specified configure options. If source changes are detected, the binary packages associated with those changes are automatically rebuilt.

NOTE: Build times will differ depending on the particular configuration you are building and on your development host resources.

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 109 for more information on improving project build times.

See also:

- [Examples of Configuring and Building](#) on page 122
- [QEMU Targets for Deployment and Testing](#)
- [Wind River Simics Targets for Deployment and Testing](#)

About Build Logs

When you build packages, the build system creates symlinks to separate build output logs for each package in

projectDir/build/package_name/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.

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 77, 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** , 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 100.



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 109 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 513.

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 513.

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

```
make -C build recipeName
```

Or simply:

```
make 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 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 513.

Wind River Linux make command	Yocto Project BitBake equivalent
make bbs	Source layers/oe-core/oe-init-buildenv bitbake_build
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.	
make	bitbake <i>imageName</i>
	For example, bitbake wrlinux-image-glibc-std . To determine the correct <i>imageName</i> , you can either:
	<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>
	NOTE: make also extracts the image and makes it ready for make start-target .
	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.

Wind River Linux make command	Yocto Project BitBake equivalent
make recipeName	bitbake recipeName
Build the package's recipe <i>recipeName</i>	
make recipeName.rebuild	bitbake -c rebuild recipeName
Rebuild the package's recipe <i>recipeName</i>	
make linux-windriver	bitbake linux-windriver
Build the Wind River Linux kernel's recipe.	
make 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 packageName.distclean	bitbake -c cleansstate recipeName
make -C build packageName.config	bitbake -c configure recipeName
make -C build packageName.download	bitbake -c fetch recipeName
make -C build packageName.rebuild and make -C build packageName.rebuild_nodep	bitbake -c compile recipeName
<hr/>	
make -C build packageName.quilt and make -C build packageName.quiltprep	bitbake -c quiltprep recipeName
<hr/>	
make -C build packageName.addpkg	Edit <i>layers/local/image.bb</i> and add the following line: <code>IMAGE_INSTALL += "packageName"</code>
<hr/>	
NOTE: You must specify the package name for this command option.	

Wind River Linux make command	Yocto equivalent
make -C build packageName.rmpkg	Edit <code>layers/local/<i>image.bb</i></code> and remove or comment out the following line: <code>IMAGE_INSTALL += "packageName"</code>
NOTE: You must specify the package name for this command option.	
make -C build packageName.env	bitbake -e recipeName

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.

About Managing Builds with the Yocto Project Toaster

Toaster is a web interface-based build analysis tool for the Yocto Project build system that is integrated into Wind River Linux.

Toaster collects data about build processes and build outcomes, presenting the data in a web interface that lets you browse, search, and query the information in different ways.

The toaster integration consists of a configuration option and three `make` targets. For information on running Toaster, see: [Running Toaster](#) on page 103

For the documentation for the Yocto Project Toaster web interface, see: <https://www.yoctoproject.org/documentation/toaster-manual-161>.

Running Toaster

You must configure a project and launch Toaster in order to make use of the web interface.

Configure your platform project with the option **--enable-toaster=yes** to enable the use of Toaster with your project.

Step 1 Configure your project in your project directory. The configure script applies the option **--enable-toaster=yes** by default.

NOTE: To maximize the data captured and presented by Toaster, you must not configure the project with a saved state cache; you should not use the option **--with-sstate-dir=**.

Step 2 Start the web server and database.

```
$ make start-toaster-webserver
```

This creates a SQLite database that stores the build data and starts the django web server. The web server is accessed by pointing a browser to **localhost:8000**.

Step 3 Launch the platform project build.

```
$ make fs-toaster
```

This launches a platform project build that enables a bitbake client to intercept and store bitbake events into the database, which at the end of the build and after flushing of events, creates the web-based Toaster views.

Step 4 Open a browser and enter **http://localhost:8000** for the URL address.

The home page for the Toaster interface opens in the browser. For examples of build details available in Toaster, see: [Sample Toaster Build Output](#) on page 103.

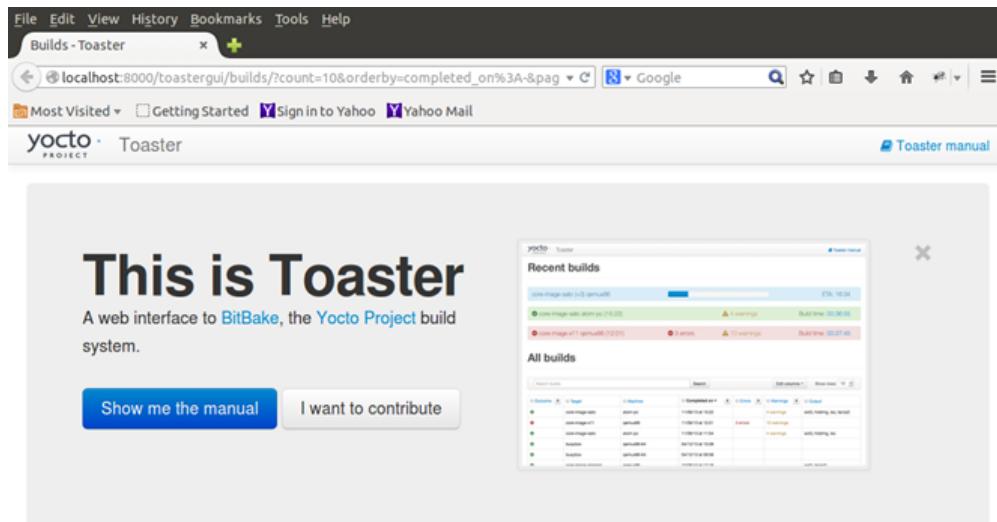
Sample Toaster Build Output

Several screen shots illustrate the interface and the build data that is viewable using Toaster.

Toaster Home Page

When you open Toaster in your browser by pointing to **localhost:8000**, the Toaster home page appears, similar to the following image.

Note that the estimated time to complete the build is a rough estimate based on the number of unfinished Bitbake tasks; that time does not account for the actual time that the remaining tasks may take and the ETA may be significantly underestimated.



Recent Builds

wrlinux-image-glibc-small qemux86-64
(14 minutes ago) ETA: in 5 minutes from now

Sample Output From a Completed Build

Immediately after the build completes, the following page indicates that the ETA is "now," but there will be a lag time of a few minutes after the make command exits but before the data has been completely written to the Toaster database. Once the data is written, a refresh of the page will show output similar to the following.

The screenshot shows the Yocto Project Toaster web interface. At the top, there's a navigation bar with links for File, Edit, View, History, Bookmarks, Tools, and Help. Below the navigation bar is a browser-like header with tabs for 'Builds - Toaster' and 'localhost:8000/toastergui/builds/?count=10&orderby=completed_on%3A-&'. The address bar shows 'localhost:8000/toastergui/builds/'. The page title is 'yocto PROJECT Toaster' and there's a link to 'Toaster manual'.

The main content area has a section titled 'Recent Builds' which lists one build entry:

✓ wrlinux-image-glibc-small qemux86-64 (3 seconds ago)	Build time: 00:30:36
---	----------------------

Below this is a section titled 'All builds' containing a table with the following data:

Outcome	Target	Machine	Completed on	Failed tasks	Errors	Warnings	Output
✓	wrlinux-image-glibc-small	qemux86-64	17/10/14 06:11				ext3, tar.bz2

At the bottom of the table, it says 'Showing 1 to 1 out of 1 entries.' and 'Show rows: 10'.

Subsequent builds in the same project directory are appended to the build list. As described in the Yocto on-line Toaster documentation, clicking on a cell under the Target column initiates a drill-down into the data collected by Toaster, with the first page appearing similar to the screen below.

The screenshot shows a web browser window for the 'Toaster' application. The URL is 'localhost:8000/toastergui/build/1'. The main content area displays the build details for 'wrlinux-image-glibc-small qemux86-64'.

Completed on 17/10/14 06:11 Build time: **00:30:36**

Images

wrlinux-image-glibc-small

Packages included: 359
Total package size: 39.5 MB
License manifest
Image files: wrlinux-image-glibc-small-qemux86-64-20141017104021.rootfs.ext3 (64.0 MB)
wrlinux-image-glibc-small-qemux86-64-20141017104021.rootfs.tar.bz2 (13.4 MB)

Build summary

Configuration	Tasks	Recipes & Packages
Machine: qemux86-64 Distro: wrlinux-small Layers: extra-downloads local meta meta-downloads meta-networking meta-oe-subset meta-per-subset meta-python-subset meta-websERVER oe-core-d-1.5 qemux86-64 wr-base wr-fixes wr-kernel	Total number of tasks: 1876 Tasks executed: 1686 Tasks not executed: 190 Reuse: 10 %	Recipes built: 169 Packages built: 3192

You can drill down by clicking on various fields on the screen. For example, by successive selections of **Packages**, then **busybox**, then **wrlinux-image-glibc-small**, you can view the set of files that are installed into the target file system image, similar to the screen below.

The screenshot shows a web browser window with the Toaster interface. The URL is `localhost:8000/toastergui/build/1/package_included_detail/1/2540?count=25&c=`. The page title is "busybox_1.22.1-r32.0 (wrlinux-image-glibc-small)". On the left, there's a table of files in the root file system. On the right, there's a "Package information" panel with details like size, license, recipe, version, layer, and layer branch.

File	Size
/bin/busybox	14 B
/bin/busybox.nosuid	506.4 KB
/bin/busybox.suid	51.5 KB
/bin/sh	14 B
/etc/busybox.links.nosuid	2.1 KB
/etc/busybox.links.suid	91 B

Package information

Size ⓘ 560.2 KB
 License ⓘ GPLv2 & bzip2
 Recipe ⓘ busybox
 Recipe version ⓘ 1.22.1-r32
 Layer ⓘ meta
 Layer branch ⓘ wr-7.0-20141008
 Layer commit ⓘ e4c78e4ee6ba01380838b893f12bbfbfcff231ca
 Layer directory ⓘ /ord-dlerner-d81/dlerner/70/toast/layers
 /oe-core/meta

Shutting Down Toaster Web Server

You can safely shut down Toaster web server while preserving the database.

After stopping Toaster, the web pages will no longer be served on your host by the Django web server. The SQLite database is preserved and you can restart the server to view the data.

- Run the following command to shut down Toaster web server.

```
$ make stop-toaster-webserver
```

Restarting Toaster Web Server

You can restart Toaster web server to view stored database information about a build.

Re-starting Toaster web server in the same project directory where it was previously running will re-enable the views. The SQLite data remains in the `projectdir/bitbake_build/toaster.sqlite` database after the server is stopped.

- Step 1** Run the following command to restart the web server.

```
$ make start-toaster-webserver
```

Collecting Toaster Data for Multiple Build Project Directories

You can configure Toaster to collect data from multiple build project directories.

The **start-toaster-webserver** build targets create a SQLite database in the current project's **bitbake_build** directory. To use a SQLite database that can be shared for more than one project, do not use the `make-start-toaster-webserver` command, but instead use the following procedure.

- Step 1** Copy a project directory bitbake subdirectory to a non-project location, such as `$HOME/all-builds` as shown below.

```
$ cp -r projectDir/bitbake $HOME/bitbake
```

- Step 2** Run the database setup and web server startup command in the new bitbake directory.

```
$ cd $HOME/bitbake
$ installDir/scripts/toaster-webserver.sh -b $PWD -c start
```

- Step 3** In this bitbake directory, export the path to the common sqlite directory into the Toaster shell variable `DATABASE_URL`

```
$ export DATABASE_URL=sqlite3:///$PWD/toaster.sqlite
```

- Step 4** With the export above set, then run `make fs-toaster` command in any project directory configured with the option `-enable-toaster=yes`.

Changing Toaster Servers

The configure and build Yocto Toaster/Wind River Linux integration uses the Toaster out-of-box database server, SQLite, and the Toaster out-of-box web server built into Django.

This is adequate for a small number of users running a limited set of concurrent builds. Use of other database engines such as MySQL or PostgreSQL can be done using the following procedure.

- Step 1** Copy the project bitbake directory to create a separate, out-of-project-directory bitbake directory.

```
$ cp -r projectDir/bitbake $HOME/bitbake
```

- Step 2** From this point, follow the Yocto Toaster documentation instructions for setting up a different SQL database server. For the documentation for the Yocto Project Toaster web interface, see: <http://yoctoproject.org/bulk/devday-eu-2014/ypdd14-reyna-toaster.pdf>.

Postrequisites

 **NOTE:** Consult the Toaster documentation for setting up Apache HTTP Server as a replacement for the Django web server: <http://yoctoproject.org/bulk/devday-eu-2014/ypdd14-reyna-toaster.pdf>.

For Yocto Project Toaster documentation, see: <https://www.yoctoproject.org/documentation/toaster-manual-161>.

Recapturing Complete Toaster Data for Subsequent Builds

The build system is optimized to reuse previous build data in order to optimize build speed.

Running **make fs-toaster** for subsequent builds in the same directory will not provide much information, since the bitbake functions that capture the data are bypassed in favor of already built artifacts. In order to maximize Toaster data recaptured in subsequent builds, you must manually remove build artifacts from the previous project as follows:

Step 1 Go to your project directory.

```
$ cd projectDir
```

Step 2 Run the following commands.

```
$ rm -rf sstate-cache/*  
$ cd tmp  
$ rm -rf sstate-control stamps sysroots buildstats deploy
```

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*.

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** 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.

About Accelerating Builds with a Remote Shared State Cache Server

A remote shared state cache server helps you accelerate builds in a shared development environment.

In addition to using a local build directory shared state cache, as described in the *Wind River Linux Getting Started Guide Configuring a Platform Project*, you can also specify a remote server to provide a read-only shared state cache. This approach lets you share the cache with other projects and/or developers to provide a network resource to accelerate platform project builds.

To avoid filesystem problems caused by too many files in one directory, the sstate files are distributed across directories named with the first two characters of the sstate checksum.

To avoid incompatibilities with native sstate, the shared sstate is located in a separate directory. The name of this directory is based on the LSB release naming. For example: Ubuntu 12.04 is "Ubuntu-12.04" and OpenSuSE 12.3 is "OpenSuSE-12.3". This means that native sstate files produced on a host like Ubuntu-12.04 are by default not compatible and will not be used on other hosts.

The name is available as a Bitbake variable `NATIVE LSB STRING`, and is displayed each time `configure`, `make` or `bitbake` is run.

Native sstate Compatibility

By default native sstate can only be reused on a host OS with the same `NATIVE LSB STRING`. The following compatibility guidelines will help you determine which shared sstate caches can be used with which platforms.

- RedHat/CentOS 5.x distributions are compatible. The `NATIVE LSB STRING` for all RedHat/CentOS 5.x host distributions is "RedHat-5". For example the native sstate generated on CentOS 5.8 can be used on a RedHat 5.9 machine.
- RedHat/CentOS 6.x distributions are compatible and their `NATIVE LSB STRING` is "RedHat-6". For example native sstate built on RedHat 6.3 can be reused on a CentOS 6.4 machine.



NOTE: To override default behavior and make native sstate from another host OS (that has a `NATIVE LSB STRING` of `OTHERLSB`) available to a build, add the following to `local.conf`:

```
SSTATE_MIRRORS += "file://.*/(.*)/(.*) (file|http)://path/to/sstate/  
SLED-11.2/1/2"
```

If the host operating systems are not compatible, this will cause build failures.

Setting Up a Remote Shared State Cache Server

Before you can accelerate builds with a remote shared state cache server, you must build a platform project to seed the shared state cache directory.

This procedure provides an alternative to using the local shared state cache. Use this approach to create a remote shared state cache as described in [About Accelerating Builds with a Remote Shared State Cache Server](#) on page 110.

To perform this procedure, you will need a server on your network with access to the platform project build directory and access for other users that intend to use the remote shared cache from their own project build directories.

Step 1 Configure a platform project using the `--with-sstate-dir= configure` script option.

This option lets you specify a directory that the build system will use to seed and share build-related files, to help accelerate future builds.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel-standard \
--enable-rootfs=glibc_std \
--with-sstate-dir=path_to_ssccache_dir
```

To set the shared state cache directory for an existing platform project, add the `--enable-reconfig=yes` option to the `configure` script command.

In this example, the `path_to_ssccache_dir` is the location on your development host that you will synchronize with the remote server.

Step 2 Build the platform project to populate the cache seed files.

```
$ make
```

This process can take some time to complete.

Step 3 Optionally verify that the shared state cache directory is populated.

Once the project build completes, the directory specified in `--with-sstate-dir=` will be populated with a number of related subdirectories.

```
$ cd path_to_ssccache_dir
$ ls
00/ 10/ 21/ 32/ 40/ 53/ 64/ 7e/ 8f/ a0/ af/ c0/ d9/ ef/
02/ 11/ 22/ 33/ 41/ 54/ 65/ 7f/ 90/ a1/ b0/ c1/ db/ f1/
03/ 13/ 23/ 34/ 44/ 55/ 66/ 81/ 91/ a2/ b3/ c2/ dd/ f2/
04/ 14/ 24/ 36/ 47/ 58/ 69/ 82/ 92/ a5/ b7/ c7/ e0/ f7/
06/ 15/ 26/ 37/ 49/ 59/ 71/ 85/ 93/ a6/ b8/ c8/ e2/ f8/
08/ 16/ 27/ 39/ 4b/ 5a/ 73/ 87/ 95/ a7/ b9/ c9/ e3/ fb/
0a/ 17/ 28/ 3a/ 4d/ 5b/ 76/ 89/ 97/ a8/ bb/ d1/ e4/ fe/
0b/ 18/ 29/ 3b/ 4e/ 5e/ 79/ 8b/ 98/ a9/ bc/ d4/ e5/ ff/
0c/ 1a/ 2b/ 3c/ 4f/ 60/ 7a/ 8c/ 99/ aa/ bd/ d5/ e9/ Ubuntu-12.04/
0e/ 1d/ 2f/ 3d/ 51/ 62/ 7c/ 8d/ 9a/ ab/ be/ d7/ ea/
0f/ 20/ 30/ 3e/ 52/ 63/ 7d/ 8e/ 9e/ ac/ bf/ d8/ ed/
```

Step 4 Set up a server on the network.

To use the cache with other builds or projects, you will require a server that can provide read-only access to the cache created in the previous steps. This server must have access to the cache directory, and also the ability to be accessed by other platform project build locations that will use the server's shared cache.

The shared state cache option supports the NFS, HTTP, and FTP server protocols, although NFS on a fast network is recommended. While HTTP and FTP are supported, you should avoid using them. HTTP limits the shared state cache to read-only and FTP provides poor performance.

Step 5 Sync the network server to the cache location. You have three options:

Options	Description
Using make host-tools	<p>Use the make host-tools command to build a portable archive containing only native shared state cache files.</p> <ol style="list-style-type: none">1. Make the compressed archive for exporting. <pre>\$ make host-tools</pre><p>Once the command completes, the archive is located at <i>projectDir/export/host-tools.tar.bz2</i>.</p><ol style="list-style-type: none">2. Copy the archive to the location on the server where you want the cache to be located that is accessible to the next build.3. Uncompress the archive in a network location <pre>\$ tar -xjf host-tools.tar.bz2</pre>
Using rsync	<p>The rsync method requires a user on the server with write access to the server's document root directory, and the server IP address.</p> <ol style="list-style-type: none">1. Run the following command from the shared state cache directory: <pre>\$ rsync --progress --archive --stats \ > user@serverIP_address:read_only_sstate_cacheDir</pre>In this example, you would substitute <i>user</i>, <i>serverIP_address</i>, and <i>read_only_sstate_cacheDir</i> for the server's actual values. For example: <pre>\$ rsync --progress --archive --stats . \ > mary@128.221.1.55:/var/www/ro-sstate-cache</pre>Using the values above, the cache directory would be located in the server's /var/www/ro-sstate-cache directory. You can optionally include the --remove-source-files <i>cacheDir</i> variable to remove source files from the platform project build directory and place them on the server. For example:
Using NFS	<p>NOTE: If you do not include this option, the build system will try to retrieve cache data from the original cache directory, and you will not be able to observe the level of cache fetching that the server provides.</p> <p>Rsync the files to a shared NFS drive. This requires using the instructions directly above in the rsync option, but pointing to the location of the NFS share.</p>

Once the server setup is complete, you can configure platform projects on the network to use the read-only cache to accelerate project builds.

Accelerating Builds with Shared State Cache

Configure a platform project to use a read-only shared state cache on a network server.

This procedure requires a network server configured to use a read-only shared state cache as described in [Setting Up a Remote Shared State Cache Server](#) on page 111.

Step 1 Configure a platform project to point to the network server's shared state cache.

Use the **--with-ro-sstate-mirror= configure** script option to specify the server's IP address and cache location.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_std \
--enable-buildstats=yes \
--with-ro-sstate-mirror=[http | ftp]://serverIP_address:read_only_sstate_cacheDir #use
this to mirror on remote host)
--with-ro-sstate-mirror=file:///read_only_sstate_cacheDir #use this line to mirror on
localhost
```

In this example, you would substitute *serverIP_address* and *read_only_sstate_cacheDir* for the actual IP address and cache directory, for example: **128.221.1.55:/var/www/ro-sstate-cache**.

Once configuration is complete, the platform project will use the cache specified.

Step 2 Build the platform project.

```
$ make
```

The build looks in the local sstate, and then the remote mirror for cached items to accelerate the build. If it doesn't find them, it builds from scratch.

When **--enable-buildstats=yes** is used, output as follows is appended to build output:

```
NOTE: Build completion summary:
NOTE:  From shared state: 288
NOTE:  From scratch: 11
```

In this case, 288 tasks were not rerun because the build system was able to pull the required data from the shared cache.

Step 3 Optionally perform the previous steps on other platform projects to leverage the shared cache.

Creating and Using a Shared State Cache with Multiple Host Operating Systems

Use the **--with-ro-sstate-mirror** configure script option to use a single shared sstate cache with multiple host operating systems.

Reusing shared state cache among multiple developers is an effective way to shorten the time required for builds. However, the separation of native shared sstate cache and non-native shared state cache files makes creating a shared sstate cache more complicated. More information is available at [About Package Revision Management](#) on page 251.

The recommended steps are:

1. Build and update the non-native shared sstate cache separately.

2. Create a separate native shared state cache for each host OS.
3. Run the **configure** command with the **--with-ro-sstate-mirror** for each location.

Maintaining Shared State Cache

Shared state cache can be maintained with the **sstate-cache-management.sh** script.

- Run **sstate-cache-management.sh** regularly to remove duplicates from an sstate cache.

```
$ cd buildDir/layers/oe-core/scripts;
./sstate-cache-management.sh --yes --remove-duplicated --cache-dir=sstateDir
```

Securing Shared State Cache

Depending on your development environment, or company policy, you may want to secure objects in your shared state cache.

You can secure shared state cache items at build time with a variety of security options, such as:

Using Read-only to Restrict Non-authorized Builds from Source

This method uses the **--with-template=feature/sstate_READONLY configure** script option to set the shared state cache to read-only. Once set, with the cache enabled, the **SSTATECACHE_WHITELIST** setting in the **projectDir/local.conf** file specifies the packages that may be built from source and used in the build.

As a result, for objects to be built from source, you must add them to the **SSTATECACHE_WHITELIST** setting, for example:

```
SSTATECACHE_WHITELIST = "recipe1 recipe2 recipe3 recipe4"
```

From a security standpoint, this method provides the greatest security, because it only allows objects placed into the shared state cache and included in the whitelist to be built from source in your build.

By default, a recipe that is built from source but not in the whitelist will cause the build to fail. This prevents unwanted packages from being built.

If you do not want to halt builds, due to items not being whitelisted, or changed in the source, use the following setting:

```
SSTATE_CHECK_LEVEL = "warn"
```

This may also be added using the **--with-sstate-check-level=warn configure** script option. For additional information, see [Using the Read-only Template and Whitelist to Secure Shared State Cache](#) on page 116.

To disable any build from source, which means only the build from existed shared state cache is allowed, include this entry in your **projectDir/local.conf** file, but leave the list empty, for example:

```
SSTATECACHE_WHITELIST = ""
```

Once you update the setting, you must run the **make** command to rebuild the platform project image to update the cache. Items that were initially included in the cache will return an error.

It is recommended to use the read-only shared state cache feature with a shared cache specified using the **--with-sstate-dir= configure** script option. This ensures the following:

- If all recipes are built from cache, there are no build errors.
- If any recipe is built from source, it halts the build, or returns an error warning, depending on your **--with-sstate-check-level=** setting.
- To allow a recipe to build from source, add it to the whitelist.

Signing and Verifying to Restrict Non-authorized Builds from Existing Shared State Cache

Signing and verifying cache objects ensures that only signed items are allowed in your builds from the shared state cache. This method requires a pair of public and secret (private) keys to sign and verify objects. Once you have this key information, you can add it with the **configure** command as an option, or to the **projectDir/local.conf** file, setting.

The **--with-template=feature/sstate_signature configure** script option enables signature verification for your platform project image, where your shared state cache resides. Once enabled, you use other **configure** script options to specify the following:

--with-sstate-sign-format=

Format in which the system verifies cache objects, including **gpg**, **md5sum**, **sha1sum**, and **sha256sum**. The **projectDir/local.conf** setting is:

```
SSTATE_SIGN_FORMAT = "option"
```

--with-sstate-sign-mode=

The mode in which the implements signature verification, including **sign_only**, **verify_only**, or **sign_verify**. The **projectDir/local.conf** setting is:

```
SSTATE_SIGN_MODE = "option"
```

--with-sstate-check-level=

The verification level, either **error** or **warn**. The **projectDir/local.conf** setting is:

```
SSTATE_CHECK_LEVEL = "option"
```

--with-sstate-secret-key=Secret_Key_Path

The location in the file system to the secret key path, which is typically a directory where the secret key is maintained. The **projectDir/local.conf** setting is:

```
GPG_SECRET_KEY = "path_to_key_directory"
```

--with-sstate-gpg-passphrase=Password

Used with the **--with-sstate-secret-key=** option, this provides the password necessary to verify the GPG signature. The **projectDir/local.conf** setting is:

```
GPG_PASSPHRASE = "password"
```

--with-sstate-public-key=Public_Key_Path

The location in the file system to the public key path, which is typically a directory where the public key is maintained. This is required for GPG security, and is used with **--with-**

`sstate-sign-format=gpg` and `--with-sstate-sign-mode=[verify_only | sign_verify]` options.
The `projectDir/local.conf` setting is:

```
GPG_PUBLIC_KEY = "path_to_key_directory"
```



NOTE: It is beyond the scope of this guide to provide information on the steps required to establish your security credentials.

A warning is issued at build time if there is any mismatch in the security settings. For additional information, see [Using Verification and Signing Options to Secure Shared State Cache](#) on page 119.

When you add security settings to the shared state cache, this adds another verification step to the build system for each object that it tries to retrieve from cache, which processes thousands of objects during a typical build. Depending on the complexity of your security settings, this can cause the build to take longer than it may when building from source.

As a result, security settings are disabled by default so that there is no impact on performance. To enable, and set shared state cache security, see [Using Verification and Signing Options to Secure Shared State Cache](#) on page 119.

Using the Read-only Template and Whitelist to Secure Shared State Cache

Access for shared state objects can be set using `configure` script options.

In this procedure, you will use the `--with-template=feature/sstate_READONLY` `configure` option to make the shared state cache read-only, and add recipe names to the `SSTATECACHE_WHITELIST` setting in the `projectDir/local.conf` file.

In addition, you will optionally update the `--with-sstate-check-level=` settings to provide warnings, but allow build completion.



NOTE: You can combine read-only shared state cache security with signing and verification as described in [Using Verification and Signing Options to Secure Shared State Cache](#) on page 119.

Step 1 Create a platform project using the `--with-sstate-dir=` `configure` script option.

This option lets you specify a directory that the build system will use to seed and share build-related files, to help accelerate future builds.

- a) Create a platform project directory and navigate to it.
- b) Configure the project.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_std \
--with-sstate-dir=path_to_ssccache_dir
```

To set the shared state cache directory for an existing platform project, add the `--enable-reconfig=yes` option to the `configure` script command.

In this example, the `path_to_ssccache_dir` is the location on your development host that you will share the cache directory with.

- c) Build the platform project to populate the cache seed files.

```
$ make
```

This process can take some time to complete.

Step 2 Configure a new platform project to set read-only shared state cache.

- a) Create a platform project directory and navigate to it.
- b) Configure the project.

```
$ configDir configure \
--enable-board=qemux86-64 \
--enable-rootfs=glibc_std \
--enable-kernel=standard \
--with-template=feature/sstate_READONLY \
--with-sstate-dir=path_to_project1_ssccache_dir
```

Set the **--with-sstate-dir**= option to the same directory specified for the first platform project created in step 1.

Step 3 Build a package to seed the shared state cache.

In this example, a native (host) application is used to demonstrate read-only shared state cache operation.

Run the following command from the platform project directory configured in the previous step.

```
$ make db-native
```

The package builds successfully from the shared state cache, which has been previously populated.

Step 4 Clean the package's shared state cache and rebuild the package.

- a) Clean the package shared state cache items.

Use the **cleansstate** **make** option to clear shared state cache package items.

```
$ make db-native.cleansstate
```

- b) Rebuild the package.

```
$ make db-native
```

This command triggers an error:

```
|ERROR: In sstatecheck hook:sstate_READONLY_check Read-only
|sstate-cache is enabled, the build of "db-native quilt-native
|autoconf-native automake-native libtool-native
|gnu-config-native m4-native texinfo-dummy-native"
|did not come from sstate-cache. Only the recipe listed in
|SSTATECACHE_WHITELIST is allowed to build from source
|ERROR: Hash validation failed in RunQueueExecuteScenequeue
...
...
```

Notice that the build fails, because there are no objects allowed in the shared state cache that the package requires, because the platform project is set to use a read-only shared state cache.

This is expected, because by default, the **db-native** package is not in the **SSTATECACHE_WHITELIST** setting in the **projectDir/local.conf** file, and also because of the default setting in the **SSTATE_CHECK_LEVEL** setting in the **projectDir/local.conf**, which defaults to **error**, a setting that halts the build.

Step 5 Update the shared state cache whitelist.

In the previous step, the build failed because the items that the build system tried to add to the shared state cache could not be added because the cache is set to read-only. To complete the build, you must add the build items in the output to the shared state cache whitelist.

Options	Description
Use the make command	Add each package, one at a time, using the following command: <pre>\$ make db-native.sstate_whitelist</pre> You must perform this command for each of the following, additional packages: quilt-native , autoconf-native , automake-native , libtool-native , gnu-config-native , m4-native , and texinfo-dummy-native .
Update the <i>projectDir</i>/<i>local.conf</i> file	<ol style="list-style-type: none">1. Open the <i>projectDir</i>/local.conf file in an editor, and add the following content: <pre>SSTATECACHE_WHITELIST += "db-native quilt-native autoconf-native \ automake-native libtool-native gnu-config-native m4-native \ texinfo-dummy-native"</pre>2. Save the file.

Step 6 Build the **db-native** package again.

```
$ make db-native
```

Now that the required items are whitelisted, the build succeeds. From this point forward, when you build the **db-native** package, the build system will use the package information in the cache to speed up build times.

If you make changes to the package source or recipe, this will trigger a change by the build system to build from source, rather than the shared state cache. Since the source content is different, this will cause the build to fail, because the source does not match what is in the cache. To correct this, clear the shared state cache and rebuild.

Step 7 Optional: Optionally change the **SSTATE_CHECK_LEVEL** setting to provide warnings rather than halt the build.

During early stages of development, you may not want to halt builds due to shared state cache objects not being whitelisted.

Update platform project configuration to set the shared state cache check level.

Options	Description
Use the configure script.	Update the platform project by rerunning the configure script with additional objects. <pre>\$ configDir configure \ --enable-board=qemux86-64 \ --enable-rootfs=glibc_std \ --enable-kernel=standard \ --with-template=feature/sstate_READONLY \ --with-sstate-check-level=warn \ --enable-reconfig=yes</pre>
Update the <i>projectDir</i>/local.conf file	Open the <i>projectDir</i> /local.conf file in an editor, and add the following content: <pre>SSTATE_CHECK_LEVEL = "warn"</pre>

If there are changes to the shared state cache whitelist, or the source that whitelisted packages are built from, instead of halting the build, this setting will provide a warning instead.

In this case, the package should rebuild without errors because it has been previously whitelisted, and no changes were made to the source.

Using Verification and Signing Options to Secure Shared State Cache

Signing and verifying shared state cache objects provides security that can be used to ensure product validity throughout the development process.

To perform this procedure, you must first have the security credentials required, including:

- GPG public key directory, with a public key located in it
- GPG secret key directory, with a secret key located in it
- A valid GPG passphrase

 **NOTE:** Generating security keys and passphrases is beyond the scope of this document.

This information is necessary to use GPG to sign and verify shared state cache objects.

 **NOTE:** You can combine signing and verification with read-only shared state cache security as described in [Using the Read-only Template and Whitelist to Secure Shared State Cache](#) on page 116.

Step 1 Create a platform project to create a signed shared state cache to be used by other, local platform projects.

This platform project uses the `--with-sstate-dir=path_to_ssccache_dir` option to specify a directory that the build system will use to seed and share build-related files, to help accelerate future builds.

- a) Create a platform project directory and navigate to it.

b) Configure the project.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_std \
--with-template=feature/sstate_signature \
--with-sstate-sign-format=gpg \
--with-sstate-sign-mode=sign \
--with-sstate-secret-key=Gpg_Secret_Key_Dir \
--with-sstate-gpg-passphrase=Gpg_Passphrase \
--with-sstate-dir=path_to_ssccache_dir
```

To set the shared state cache directory for an existing platform project, add the **--enable-reconfig=yes** option to the **configure** script command.

In this example, you would need to substitute *Gpg_Secret_Key_Dir*, and *Gpg_Passphrase* with your own site-specific security information. For additional information on signing options, see [Securing Shared State Cache](#) on page 114.

In this example, the *path_to_ssccache_dir* is the location on your development host that you will share the cache directory with.

c) Build the db-native package to populate the shared state cache.

```
$ make db-native
```

This process can take some time to complete.

Step 2 Configure a new platform project to use shared state cache verification.

This new platform project will make use of the **--with-sstate-public-key=** option to use the gpg public key to verify shared state cache items in the platform project created in the previous step.

- Create a platform project directory and navigate to it.
- Configure the project.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_std \
--with-template=feature/sstate_signature \
--with-sstate-sign-format=gpg \
--with-sstate-sign-mode=verify \
--with-sstate-public-key=Gpg_Public_Key_Dir \
--with-sstate-dir=path_to_project1_ssccache_dir
```

In this example, you would need to substitute *Gpg_Public_Key_Dir* with your own site-specific security information. For additional information on signing options, see [Securing Shared State Cache](#) on page 114.

Set the **--with-sstate-dir=** option to the same directory specified for the first platform project created in step 1.

Step 3 Build a package and sign shared the generated state cache objects.

The following example uses the **db-native** package as an example.

```
$ make db-native
```

The application builds successfully from the shared state cache in project 1, which has been previously populated.

Step 4 Optional: Test the verification signature by opening and editing the file and then rebuild to see if it fails.

The following examples are used for reference only. In this step, making changes to the platform project's gpg files or public key are used to simulate tampering with the system, to trigger an error.

a) Locate the signature file.

```
$ find bitbake_build/sstate-cache/signature/ -name "*db-native*"
bitbake_build/sstate-cache/signature/17/sstate:db::
6.0.30:r0::3:17877fdf57814e1e49008ae9e0eba3b1.gpg

bitbake_build/sstate-cache/signature/Ubuntu-14.04/e5/sstate:m4-native:x86_64-linux:
1.4.17:r0:x86_64:3:e5da94c56279dd35f317cff7c25beb52.g
```

In this example, the signature file is located in `projectDir/bitbake_build/sstate-cache/signature/sstate` and `projectDir/bitbake_build/sstate-cache/signature/Ubuntu-14.04/e5/sstate`. This information may be different on your development host.

b) Change the signature file.

In this example you will use the `diff --git` command demonstrates recommended changes to modify the file gpg key.

```
$ diff --git
a/bitbake_build/sstate-cache/signature/17/sstate:db::
6.0.30:r0::3:17877fdf57814e1e49008ae9e0eba3b1.gpg
b/bitbake_build/sstate-cache
index 5c29439..ab74f81 100644
---
a/bitbake_build/sstate-cache/signature/17/sstate:db::
6.0.30:r0::3:17877fdf57814e1e49008ae9e0eba3b1.gpg
+++
b/bitbake_build/sstate-cache/signature/17/sstate:db::
6.0.30:r0::3:17877fdf57814e1e49008ae9e0eba3b1.gpg
@@ -1,7 +1,7 @@
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1
-iQEcbAAQAgAGBQJUG+ivAAoJEUwjEYU/Q/RiWQH/jIAVzikqUGYDAMHDicUPZWL
+1QEcbAAQAgAGBQJUG+ivAAoJEUwjEYU/Q/RiWQH/jIAVzikqUGYDAMHDicUPZWL
hadbaHdUTPjnUSSMxeYv355FAIsYMPOY9YC5uzCfqemxhBR26TzA+9ozFeDhEwEq
A6CMNJ7xb8dRdM41pUZHmwJfMyRfO9aphcq7KOPCCVca9oW3/AQw6su7SfWw2caF
40EAdMsTsry9MaWzoVJ+AJcvIh75fkhoZOC PegVGZW6LkLd1MW5B+hFDMOxaK1ZA
```

With the gpg signature file changed, it no longer matches the generated signatures required by the shared state cache.

c) Remove the temp directory in project 2.

This clears the staging directory used by the build system, to simulate a new project build, keeping the existing shared state cache intact.

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

d) Build the **db-native** package again.

```
$ make db-native
```

This time it triggers an error:

```
...
ERROR: In sstatecheck hook:sstate_verify gpg verify sig state failed
sign
$SSTATE_DIR/signature/17/sstate:db::
6.0.30:r0::3:17877fdf57814e1e49008ae9e0eba3b1.gpg
```

```
sstate
$SSTATE_DIR/17/sstate:db::
6.0.30:r0::3:17877fdf57814e1e49008ae9e0eba3b1_populate_lic.tgz

gpg verify sig state failed
sign
$SSTATE_DIR/signature/Ubuntu-14.04/e5/sstate:m4-native:x86_64-linux:.
4.17:r0:x86_64:3:e5da94c56279dd35f317cff7c25beb52.gpg
sstate
$SSTATE_DIR/Ubuntu-14.04/e5/sstate:m4-native:x86_64-linux:.
1.4.17:r0:x86_64:3:e5da94c56279dd35f317cff7c25beb52_populate_sysroot.tgz
ERROR: Hash validation failed in RunQueueExecuteScenequeue ...
```

In this example, the build fails because the signature was changed manually, and no longer matches what is expected by the build system. This ensures that the shared cache is protected by signature verification.

- Step 5** Optional: Optionally change the `SSTATE_CHECK_LEVEL` setting to provide warnings rather than halt the build.

During early stages of development, you may not want to halt builds due to shared state cache objects not matching signature verification.

Update platform project configuration for project 2 to set the shared state cache check level.

Options	Description
Use the configure script.	Update the platform project by rerunning the configure script with additional objects. <pre>\$ configDir configure \ --enable-board=qemux86-64 \ --enable-rootfs=glIBC_std \ --enable-kernel=standard \ --with-template=feature/sstate_signature \ --with-sstate-sign-format=gpg \ --with-sstate-sign-mode=sign_verify \ --with-sstate-public-key=Gpg_Public_Key_Dir \ --with-sstate-check-level=warn \ --with-sstate-dir=path_to_project1_ssCache_dir \ --enable-reconfig=yes</pre>
Update the <code>projectDir</code>/local.conf file	Open the <code>projectDir/local.conf</code> file in an editor, and add the following content: <pre>SSTATE_CHECK_LEVEL = "warn"</pre>

If there are changes to the shared state cache objects signatures or verification, instead of halting the build, this setting will provide a warning instead.

Examples of Configuring and Building

Various strategies are available 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

Speed your platform project image development with a **configure** script example.

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

Several sample **make** commands provide a basis for configuring and building 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: <code>\$ make linux-windriver</code>

To achieve this	Do this
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 linux-windriver.menuconfig</pre> See About Kernel Configuration and Patching on page 281 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 linux-windriver.rebuild</pre>

Configuring and Building a Flash-capable Run-time

Use an 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 (journalling 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 109.

Step 2 Build the project.

```
$ make
```

For additional supported images, see [About Configuring and Building Bootable Targets](#) on page 383.

Configuring and Building a Debug-capable Run-time

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

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 82.

--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**.

To get debug information for binaries of libC6 and libGCC, it is necessary to use **--with-template=feature/build_libc**, or you can use the included debuginfo file.

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-7/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 recipeName.patch
```

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

Step 2 Build the package source.

```
$ make recipeName.rebuild
```

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

About Verifying Builds with the Image Manifest

You can use the **feature/image-manifest** template to create a platform project **.manifest** file for use in comparing platform project builds to ensure that their components match.

About the Image Manifest File

The image manifest file, also referred to as an audit manifest, provides a list of the core contents of a platform project image. The purpose of the file is to create a list that can be used as a basis to determine if the build has been corrupted, changed, or compromised when compared against the manifest.

For regular files in the build image, the image manifest includes a checksum, and the file size. For all other file types, such as directories, devices, and so on, the image manifest includes the file type and path name associated with it.

The image manifest is not created by default. To generate an image manifest file, add the **--with-template=feature/image-manifest** option to the platform project **configure** command, for example:

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_std \
--with-template=feature/image-manifest
```

Once you build the platform project, symlinks to the platform project image and manifest files are placed in the **projectDir/export/images** folder:

```
wrlinux-image-glibc-std-qemux86-64-date.rootfs.ext3
wrlinux-image-glibc-std-qemux86-64-date.rootfs.manifest
```

NOTE: The actual location of the files is **projectDir/bitbake_build/tmp/deploy/images/qemux86-64**.

The image manifest file is used with the **projectDir/layers/wrlcompat/scripts/wrl-audit-image.py** audit tool to compare the manifest build against another project build.

About the **wrl-audit-image.py** Audit Tool

The **projectDir/layers/wrlcompat/scripts/wrl-audit-image.py** audit tool script compares an image manifest file against a file system directory tree. To verify the contents of an image, it must be accessible as a file system directory tree. In a typical build, which produces a file system, for example, ***.ext3**, encapsulated in a single file, that file must be mounted in a location accessible by the host.

This is accomplished by using a loop device bound to the image file, as described in [Verifying Builds Using Image Manifest and the Audit Tool](#) on page 127. The tool runs with two arguments:

Image manifest file

This is the ***.manifest** file from the platform project configured and built with the **feature/image-manifest** template.

Location of mounted root file system

This is the image root directory of the mounted file system. The type of file system depends on your project configuration. If you do not specify the root file system type, the default is ***.ext3**.

Once invoked, the tool runs silently and exits with a **status 0**, indicating that the image manifest file matches the mounted root file system.

If there are differences, information about the mismatch is displayed, including the path name and mismatch type. Once the tool verification completes, any files found in the image hierarchy, but not in the image manifest, will display. Any error results in the script exiting with a non-zero status.

Verifying Builds Using Image Manifest and the Audit Tool

Compare an image manifest file to a mounted root file system to verify any differences in the core components that comprise the build(s).

This procedure requires a previously built platform project image, configured using the same options as the platform project used to create the image manifest file. The root file system file, for example, **wrlinux-image-glibc-std-qemux86-64-date.rootfs.ext3**, is required for comparison against the image manifest.

For additional information on the image manifest and what it is used for, see [About Verifying Builds with the Image Manifest](#) on page 126.

Step 1 Configure a platform project to create the image manifest.

Run the following command from the platform project directory.

```
$ configDir/configure \
--enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glibc_std \
--with-template=feature/image-manifest
```

The **--with-template=feature/image-manifest** **configure** option provides the necessary information to generate the image manifest once the project is built.

Step 2 Build the project.

```
$ make
```

This process can take some time, depending on your development host resources. Once the build completes, the image manifest file is located at **projectDir/export/images/wrlinux-image-glibc-std-qemux86-64-date.rootfs.manifest**

Step 3 Mount the platform project root file system image using a loop device.

This step uses the root file system file from a previous platform project image. In this example, the file is located at **/storage/wrlinux-image-glibc-std-qemux86-64.ext3**.

a) Bind a loop device to the root file system image.

```
$ sudo losetup --show --read-only --find \
/storage/wrlinux-image-glibc-std-qemux86-64.rootfs.ext3
```

When prompted, enter the super user password. When the command completes, it will display the name of the allocated loop device:

```
$ /dev/loop0
```

In this example, `/dev/loop0` represents the first loop allocation. You will need this information in the following step. The number may be different on systems with other loop allocations.

- b) Create a mount point.

Create a directory to mount the root file system.

```
$ mkdir /img
```

- c) Mount the loop device to mount the image.

Run the following command to mount the root file system image from the loop allocation created in Step 3a, above.

```
$ sudo mount -o ro /dev/loop0 /img
```

Once the command completes, the image's file system hierarchy is available in the `/img` directory.

```
$ ls /img
bin      dev      home      lib64      media      proc      sys      usr
boot     etc      lib       lost+found   mnt       root      tmp      var
```

Step 4 Run the `wrl-audit-image.py` audit tool to compare the image manifest with the mounted root file system.

The `projectDir/layers/wrlcompat/scripts/wrl-audit-image.py` audit tool is run with two arguments - the location of the image manifest file, and the mounted root file system.

```
$ projectDir/layers/wrlcompat/scripts/wrl-audit-image.py \
/storage/wrlinux-image-glibc-std-qemux86-64.manifest \
/img
```

The tool will run in the background and exit silently if there are no differences between the image manifest and mounted root file system. If there are differences, these will be displayed in the output, including different checksum values and file size differences.

About Maintaining Open Source License Compliance

Maintain compliance with various open source licensing requirements during the life cycle of the product by exporting a list with the archiver feature template.

The **archiver** feature template provides a useful way for customizing, creating, and providing an archive of the metadata layers (recipes, configuration files, and so forth). It therefore enables you to meet any requirements to include the scripts to control compilation, as well as any modifications to the original source. This output can be used for verification assurance for a number of items such as source code, licensing text, along with compilation scripts and source code modifications.

Depending on your requirements from your customer, you might be required to release any layers that patch, compile, package, or modify any open source software included in your released images under section 3 of GPLv2 or section 1 of GPLv3.

See [Feature Templates in the Project Directory](#) on page 61.

The **ARCHIVER_MODE** options are set from the **local.conf** file.

Refer to the [Archiver Options Reference](#) on page 131.

See <http://www.yoctoproject.org/docs/1.6/dev-manual/dev-manual.html#maintaining-open-source-license-compliance-during-your-products-lifecycle> for general information.

Exporting Using the Archiver Feature

Use the **archiver** feature template to create a compressed file that includes build information, licensing information, kernel information, and toolchain sources.

Run the **archiver** template feature as a configuration option. You first either build a new platform project and add the archiver configuration option or you can add the archiver option to an existing platform project as shown below in Step 1. Then you add any packages that you need to your build and rebuild the project with the archiver options set in the **local.conf** file to meet your requirements.

Step 1 Configure a platform project to export the archived content.

Options	Description
Add the archiver feature to a new platform project.	Run the following command from the platform project directory. <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_small \ --with-template=feature/archiver</pre>
Enable the archiver feature on an existing platform project.	Run the following command from the platform project directory. <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_small \ --enable-reconfig \ --with-template=feature/archiver</pre>

The **--with-template=feature/archiver** **configure** option provides the necessary information to generate an accompanying archive once the project is built.

Step 2 Add a package.

This step is optional, to provide a reference package to compare archiver settings. The following command installs the **gdb** package.

```
$ make gdb.addpkg
```

You have added the **gdb** package to your build.

Step 3 Build the platform project.

```
$ make
```

Once the build completes, the archived contents of the **gdb** package, which are based on the **archiver** options set in the **projectDir/local.conf** file, are copied to the **export/sources** or **tmp/deploy/sources** directory.

Step 4 View the archived results of the package source.

Running the following command from the project directory:

```
$ ls export/sources/x86_64-wrs-linux/gdb-* -al
```

Note the contents of the directory:

```
total 31552
-rw-rw-r-- 2 wruser wruser 32244142 Oct  8 15:31 gdb-7.7.1.tar.gz
-rw-rw-r-- 2 wruser wruser      980 Oct  8 15:31 gdbserver-cflags-last.diff
-rw-rw-r-- 2 wruser wruser      634 Oct  8 15:31 include_asm_ptrace.patch
-rw-rw-r-- 2 wruser wruser    1128 Oct  8 15:31 kill_arm_map_symbols.patch
-rw-rw-r-- 2 wruser wruser   42982 Oct  8 15:31 renesas-sh-native-support.patch
-rw-rw-r-- 2 wruser wruser      127 Oct  8 15:31 series
```

Step 5 Configure the **local.conf** file to set the **archiver** options.

If you have already configured and built a platform project and the project is built with the package added, you can set the options that you need by appending the options to the **projectDir/local.conf** file.

```
# Set Archiver Options
ARCHIVER_MODE[src] = "configured"
ARCHIVER_MODE[diff] = "1"
```

This example sets the **ARCHIVER_MODE[src]** and **ARCHIVER_MODE[diff]** options. For additional **archiver** configuration options, see [Archiver Options Reference](#) on page 131.

Step 6 Rebuild the **gdb** package.

Run the **cleansstate** command.

```
$ make gdb.cleansstate
```

Rebuild the package.

```
$ make gdb
```

View updated archived contents by running the following command:

```
$ ls export/sources/x86_64-wrs-linux/gdb-* -al
```

```
total 31724
drwxrwxr-x 2 dev1  dev1      4096 Oct  9 16:00 .
drwxrwxr-x 3 dev1  dev1      4096 Oct  9 16:00 ..
-rw-rw-r-- 2 dev1  dev1  32475140 Oct  9 16:02 gdb-7.7.1-r0-configured.tar.gz
```

Note that the output is dependent on your **archiver** option settings.

Postrequisites

You can use the various options provided by the **archiver** feature template and *projectDir/local.conf* file to specify the source change results necessary to fulfill your software build requirement for licensing and compliance issues.

Archiver Options Reference

The options that determine the **archiver** template output provide licensing and patching options are described.

ARCHIVER_MODE[src]	"original"	<p>This is the default. It also includes the original patch files, but the tar ball contains patches between do_unpack and do_patch and are no longer provided. In this situation ARCHIVER_MODE[diff] = "1" doesn't work.</p> <p>NOTE: The changes to the archiver are made by appending the options in the <i>local.conf</i> file. Usage example:</p> <pre>ARCHIVER_MODE[src] = "original"</pre>
ARCHIVER_MODE[src]	"patched"	This value provides the patched source.
ARCHIVER_MODE[src]	"configured"	This value provides the configured source.
ARCHIVER_MODE[diff]	"1"	<p>The patches between do_unpack and do_patch are created.</p> <p>You can set the one that you'd like to exclude from the diff as shown below:</p> <pre>ARCHIVER_MODE[diff-exclude] ?= ".pc autom4te.cache patches"</pre>
ARCHIVER_MODE[dumpdata]	"1"	The environment data, similar to what is included in the bitbake -e recipe .
ARCHIVER_MODE[recipe]	"1"	Sets the output of the recipe to the .bb file and the .inc file.
ARCHIVER_MODE[srpm]	"1"	Sets whether to output the .src.rpm package.
COPYLEFT_LICENSE_INCLUDE		Filters the license to be included (in the recipe where the license resides).
COPYLEFT_LICENSE_EXCLUDE		Filters the license to be excluded (in the recipe where the license resides).

COPYLEFT_LICENSE_INCLUDE	'GPL* LGPL*'	The GPL and LGPL license are included.
COPYLEFT_LICENSE_EXCLUDE	'CLOSED Proprietary'	The CLOSED Proprietary license are excluded.
COPYLEFT_RECIPE_TYPES	"target"	The recipe type that is archived.

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 18, 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

Several general steps are involved in creating a custom rootfs.

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.

Glibc File Systems

Wind River Linux provides glibc for the root file system used in your platform project builds and images. It is possible to create custom distributions and leverage additional glibc features based on your project needs.

About Glibc Default Platform Project Configuration

Before you can build and deploy a glibc-based file system, you must configure your platform project to enable glibc features, once configuration is complete. 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 glibc 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 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 a glibc platform project image, see [Creating and Customizing Glibc Platform Project Images](#) on page 138.



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 `USE_SDK_GLIBC` option is used to enable a glibc rebuild. To change the default glibc configuration and rebuild it, use the following setting in the `projectDir/layers/local/conf/distro/distroName.conf` file:

```
USE_SDK_GLIBC = "0"
```

where `distroName` is the name chosen with the `--with-custom-distro= configure` option. For additional information, see [Creating and Customizing Glibc Platform Project Images](#) on page 138.

About Modifying Wind River Linux Glibc



WARNING: While it is possible to make changes to glibc, modifying or patching the contents provided by Wind River is not recommended. Doing so will void your warranty. Wind River is not responsible for any damage(s) that stem from changes made to glibc.

When using the supported Wind River Linux toolchain, the `glibc` package is usually installed from prebuilt binaries. You can compile it manually, and if you do so, you can patch it using a `.bbappend` file.

The following workflow provides an overview of the process.

- Configure, or reconfigure, your platform project using the `feature/build_libc` template. This is required to make any changes to glibc.
- Create a layer in your platform project directory to maintain the changes. For additional information, see [About the layers/local Directory](#) on page 71.
- In the new layer, create a `glibc.bbappend` file.
- Add the location of the patches to the `glibc.bbappend` file, using the `FILESEXTRAPATHS_prepend()` statement, for example:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"  
SRC_URI_append = " file://my1st.patch  
                  file://my2nd.patch"
```

This example uses a directory named `files` in the layer directory, where the contents of the files reside.

5. Rebuild glibc once your changes are complete.

```
$ make glibc.rebuild
```

6. Deploy your platform project image to a target to test your changes.

Creating and Customizing Glibc Platform Project Images

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

To create and customize a glibc platform project image:

- Step 1** Select one of the following options for creating a glibc platform project image:

Options	Description
New platform project	Configure the platform project using the --with-custom-distro=distroName option. For example, a minimal configure command might be: <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_std \ --with-custom-distro=distroName</pre>
Existing platform project	<ol style="list-style-type: none">1. Copy the projectDir/layers/wrlcompat/scripts/custom-distro.conf file to the projectDir/layers/local/conf/distro directory and rename the file. In this example, we will change the name to: my-glibc.conf2. Edit and save the projectDir/bitbake_build/conf/local.conf file to enable a custom glibc build by changing the DISTRO variable to match the name of the projectDir/wrlcompat/scripts/custom-distro.conf from the previous step. For example, if you named the file my-glibc.conf, edit the variable as follows: <pre>DISTRO = "my-glibc"</pre>

Notice that you do not need to specify the exact file location, or even use the **.conf** filename extension.

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

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

```
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-ftraverse"
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: [Glibc Option Mapping Reference](#) on page 140.

 **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 [About QEMU Targets](#) on page 355.

Glibc Option Mapping Reference

Options for modifying glibc settings include setting database aliases, character sets, cryptographic settings, and more. Once set, these selections map components in the **option-groups.def** file for your specific glibc build.

Feature to Map	Option to Map to
ipv4	OPTION_GLIBC_INET
ipv6	OPTION_GLIBC_ADVANCED_INET6
libc-big-macros	OPTION_GLIBC_BIG_MACROS
libc-bsd	OPTION_GLIBC_BS
libc-tests	OPTION_GLIBC_CXX_TESTS
libc-catgets	OPTION_GLIBC_CATGETS
libc-charset	OPTION_GLIBC_CHARSET
libc-crypt	OPTION_GLIBC_CRYPT
libc-crypt-ufc	OPTION_GLIBC_CRYPT_UFC
libc-db-aliases	OPTION_GLIBC_DB_ALIASES
libc-envz	OPTION_GLIBC_ENVZ
libc-fcvt	OPTION_GLIBC_FCVT
libc-fmtmsg	OPTION_GLIBC_FMTMSG
libc-fstab	OPTION_GLIBC_FSTAB
libc-straverse	OPTION_GLIBC_FTRAVERSE
libc-getlogin	OPTION_GLIBC_GETLOGIN
libc-idn	OPTION_GLIBC_IDN
libc-inet-anl	OPTION_GLIBC_INET_ANL
libc-libm	OPTION_GLIBC_LIBM
libc-libm-big	OPTION_GLIBC_LIBM_BIG
libc-locales	OPTION_GLIBC_LOCALES
libc-locale-code	OPTION_GLIBC_LOCALE_CODE
libc-memusage	OPTION_GLIBC_MEMUSAGE
libc-nis	OPTION_GLIBC_NIS
libc-nsswitch	OPTION_GLIBC_NSSWITCH
libc-rcmd	OPTION_GLIBC_RCMD
libc-rtld-debug	OPTION_GLIBC_RTLD_DEBUG

Feature to Map	Option to Map to
libc-spawn	OPTION_GLIBC_SPAWN
libc-streams	OPTION_GLIBC_STREAMS
libc-sunrpc	OPTION_GLIBC_SUNRPC
libc-utmp	OPTION_GLIBC_UTMP
libc-utmpx	OPTION_GLIBC_UTMPX
libc-wordexp	OPTION_GLIBC_WORDEXP
libc-posix-clang-wchar	OPTION_POSIX_C_LANG_WIDE_CHAR
libc-posix-regexp	OPTION_POSIX_REGEXP
libc-posix-regexp-glibc	OPTION_POSIX_REGEXP_GLIBC
libc-posix-wchar-io	OPTION_POSIX_WIDE_CHAR_DEVICE_IO

6

Localization

About Localization 143

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/libc/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=\naa_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 \  
es_AR/utf8
```

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 \  
es_AR/utf8
```

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

```
es_AR/ISO-8859-1 \  
es_AR/iso8859-1
```

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
```

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

7

Portability

[**About Platform Project Portability**](#) 147

[**Copying or Moving a Platform Project**](#) 148

[**Updating a Platform Project to a New Wind River Linux Installation Location**](#) 149

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 41.



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 149.

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 148.

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.

Project Changes

It is also possible to make the changes to a platform project, such as application and package additions and modifications, and kernel configuration changes, portable and accessible to other platform projects. This is accomplished using the `make export-layer` command. For additional information, see [About Exporting Local Layer Changes](#) on page 161.

Copying or Moving a Platform Project

Copying or moving a platform project involves steps in addition to basic file system commands..

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

Update a platform project when the Wind River Linux installation (*installDir*) changes by setting the **--enable-reconfig** option.

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-7/wrlinux/.

Step 2 Enable reconfiguration and rerun the configure script.

- Open a terminal window and navigate to the *projectDir*.
- 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
```

- Modify the path to the **configure** script to match the new Wind River Linux installation (*configDir*) location.
- 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-7/git
```

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

8

Layers

[About Layers](#) 151

[About BSP Layers and Sublayers](#) 152

[Layers Included in a Standard Installation](#) 152

[Installed Layers vs. Custom Layers](#) 156

[Layer Structure by Layer Type](#) 156

[About Layer Processing and Configuration](#) 158

[About Exporting Local Layer Changes](#) 161

[Combining Platform Project Layers](#) 164

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 your 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 41 .

About BSP Layers and Sublayers

A Wind River Linux platform project automatically includes all of the sublayers in a specified BSP layer.

Wind River Linux provides support for a BSP layer which contains one or more sublayers. Also, each sublayer can have dependent or independent BSPs, for example:

```
intel-x86/
|-- conf
|   |-- layer.conf
|   |-- machine
|       |-- intel-x86-32.conf
[snip]
|-- intel-broadwell
|   |-- conf
|       |-- layer.conf
|       |-- machine
|           |-- intel-broadwell-64.conf
```

As shown above, there are two layers. The top layer is `intel-x86`, and the sublayer is `intel-broadwell`. There are two BSPs contained in these layers:

- The one in the top layer is `intel-x86-32`, defined in the `intel-x86-32.conf` file.
- The one in the sublayer is `intel-broadwell-64`, defined in the `intel-broadwell-64.conf` file.

When you run `configure` with the option `--enable-board=intel-broadwell-64`, both the `intel-x86` and `intel-broadwell` layer are added to `BBLAYERS`.

You can determine if there is a sublayer and BSP by checking for the following:

- There must be a `conf/layer.conf` in the subdirectory, for example, the `intel-broadwell/conf/layer.conf`.
- There must be a `conf/machine/bsp.conf` configuration file in the subdirectory, for example, the `intel-broadwell/conf/machine/intel-broadwell-64.conf`.

In this example, the `intel-broadwell` is a sublayer, and the `intel-broadwell-64` is a BSP in the sublayer.

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 57 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 156.



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 47 and [About README Files in the Build Environment](#) on page 72.

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-mingw-dl

Provides support for building **mingw** applications to support Windows Application Development projects.

meta-networking-dl/

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

meta-oe-subset-dl/

Contains subsets of the **meta-openembedded** upstream git repository, located at <http://git.openembedded.org/meta-openembedded/tree/>, selected by Wind River Linux development.

meta-perl-subset-dl/

Contains subsets of the **meta-openembedded/meta-perl** upstream git repository, located at <http://git.openembedded.org/meta-openembedded/tree/>, selected by Wind River Linux development.

meta-python-subset-dl/

Contains subsets of the **meta-openembedded/meta-python** upstream git repository, located at <http://git.openembedded.org/meta-openembedded/tree/>, selected by Wind River Linux development.

meta-security-dl/

Provides support for building security-related software.

meta-selinux-dl/

Provides support for building SELinux-related software.

meta-webserver-dl/

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/

Provides many of the base components for Wind River Linux. 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 80.

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-fixes/

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

wr-freescale_qoriq_dpa/

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-prebuilt/

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

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 152 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 159 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 167 for additional information.

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

See [Enabling a Layer](#) on page 160 and [Disabling a Layer](#) on page 161 for additional information.

Layer Structure by Layer Type

Layers all have a similar structure, but include additional directories 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 71 for additional information.

machine-specific

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

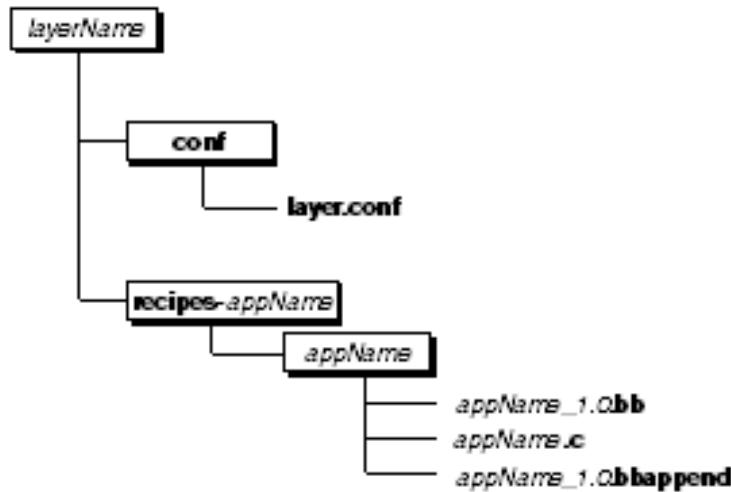
distribution-specific

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:

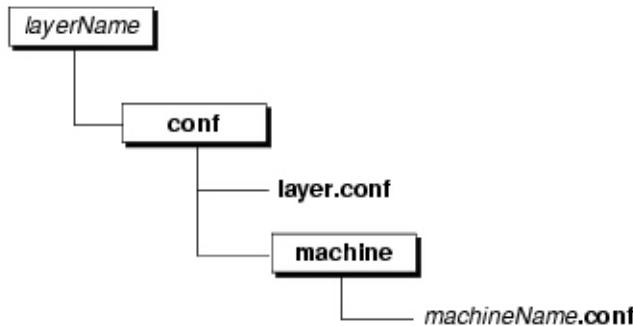


In this example, the layer includes a `conf/` directory with the `layer.conf` file, and a `recipes-appName` folder with a recipe file (`.bb`), application source (`.c`), and an append file (`.bbappend`) that extends the capability of an existing recipe file.

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

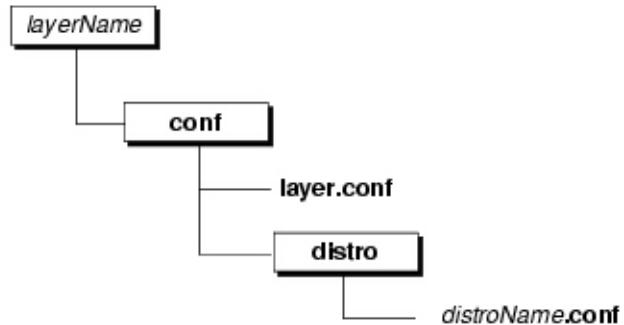
The following figure provides an example of a minimum layer requirements for a machine-specific layer:



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 159
- [Creating a Recipe File](#) on page 170

About Layer Processing and Configuration

Wind River Linux includes layers, including some optional products from Wind River. You can create your own custom layers, and include layers created by others. The build system configures layers 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 the **configure** script searches for components, it selects higher-level components first. When the **configure** script 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

It is important to understand the general project configuration workflow 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 [About Configuring a Platform Project Image](#) on page 77. 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 151
 - [Metadata](#) on page 40
 - [Configuration Files and Platform Projects](#) on page 41

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.

For additional information, see [Adding an Application to a Root File System with fs_final*.sh Scripts](#) on page 212 and [Adding an Application to a Root File System Using changelist.xml](#) on page 211.

Creating a New Layer

Several steps are required 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 41) 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 156.

Options	Description
Machine support	If the layer is adding support for a machine, add the machine configuration in conf/machine/

Options	Description
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.

Related Links

[About Layers](#) on page 151

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

[Metadata](#) on page 40

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

[Configuration Files and Platform Projects](#) on page 41

Various **.conf** files define specific aspects of your platform project build.

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 57.

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:

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 57). 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 57.

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
```

Related Links

[About Layers](#) on page 151

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

About Exporting Local Layer Changes

Changes made in your build environment may be captured for use in another platform project or for source control management.

After making a number of changes in your build environment it is very useful to be able to create a layer capturing those changes. The layer then provides a way to recreate your current customized build. To include the layer in another platform project, you just enter the original **configure** command, and specify the layer to include with the **--with-layer= configure** option.

Similarly, other developers can recreate your customized build environment in the same way, or you can modify your build environment by including their layers.

To export all project-specific changes in a single layer, use the **make export-layer** command. When you run this command from the *projectDir*, it creates the *projectDir/export/export-layer* directory, and populates the directory with the following:

- New layer directory, with a name comprised of the *projectDir* and a time stamp, for example:
qemux86-64-glibc-small-20140424-1110PDT
- Compressed **.tar** file, with the same name as the directory above. You can extract the contents of this file to another location for use with another platform project.

Once exported, the layer and compressed file includes the following platform project changes:

- Package additions and modifications
- System file modifications
- **kernel.config** modifications

Exporting Local Layer Changes

Use the **make export-layer** command to export platform project application and kernel changes for use in another platform project or for source control management.

To export the changes made to an existing platform project, you must have a previously built platform project.

Step 1 Export the platform project changes to a layer.

Run the following commands from the *projectDir*.

Options	Description
Export to default location	<pre>\$ make export-layer</pre> <p>When the command completes, the exported layer, and a compressed .tar file with its contents, with all project modifications will be located in the <i>projectDir/export/export-layer</i> directory.</p>
Export to a specified location	<pre>\$ EXPORT_LAYER_DIR=\$HOME/layerName/ make export-layer</pre> <p>When the command completes, the exported layer, and a compressed .tar file with its contents, with all project modifications will be located in the <i>layerName</i> directory in your home directory. If the directory does not exist, it will be created for you.</p>

After the layer is created, you may either copy its contents to another location, or use the **.tar** file to move the contents. Note that you must extract the contents of the **.tar** file before you use it in another platform project.

Step 2 Optionally use the exported layer in another platform project.

Copy, move, or extract the exported layer content from the previous step. The location should be on your local development host.

Step 3 Configure and build the project to include the exported layer contents.

Options	Description
New platform project	<p>Use the <code>--with-layer=layerPath/layerName</code> configure script option when you initially create the platform project. For example:</p> <ol style="list-style-type: none"> 1. Create the <i>projectDir</i>. <pre>\$ mkdir -p qemux86-64-glibc-small && cd qemux86-64-glibc-small</pre> <ol style="list-style-type: none"> 2. Run the configure command to include the layer in the new platform project. <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-rootfs=glibc_small \ --enable-kernel=standard \ --with-layer=layerPath/layerName</pre> <p>In this example, <i>layerPath/layerName</i> refers to the location and name of the exported layer.</p> <ol style="list-style-type: none"> 3. Build the platform project. <pre>\$ make</pre>
Existing platform project	<p>Include the <code>--enable-reconfig=yes</code> option in your configure command. For example:</p> <ol style="list-style-type: none"> 1. Navigate to the <i>projectDir</i> of the platform project that you want to add the layer to. 2. Run the configure command to include the layer in the new platform project. <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-rootfs=glibc_small \ --enable-kernel=standard \ --with-layer=layerPath/layerName \ --enable-reconfig=yes</pre> <p>In this example, <i>layerPath/layerName</i> refers to the location and name of the exported layer. The <code>--enable-reconfig=yes</code> option allows the platform project to configure the new additions, which is the added layer.</p> <ol style="list-style-type: none"> 3. Build the platform project. <pre>\$ make</pre>

Combining Platform Project Layers

Use the **bitbake-layers flatten** command to combine specified layers into a single layer for use in another platform project or for source control management.

Depending on your development requirements, you can combine any specified layers, or all layers, into a single layer in your platform project using the **bitbake-layers flatten** command. The command syntax is as follows:

```
bitbake-layers flatten layerName1 layerName2 layerName3 destinationLayer
```

In this example, the contents of *layerName1*, *layerName2*, and *layerName3* would be added to a new layer named *destinationLayer*.

To combine layers, you must have a previously built platform project, and the layer must be a valid, Yocto Project layer that is part of the platform project image.

Step 1 Start the bitbake shell.

Run the following commands from the *projectDir*.

```
$ make bbs
```

Once complete, the prompt will run in the *projectDir/bitbake_build* directory.

Step 2 List the available project layers.

Run the following command to list the project layers. This is required, because the **bitbake-layers flatten** command will only combine valid project layers listed in the *projectDir/bitbake_build/conf/bblayers.conf* file.

```
$ bitbake-layers show-layers
```

The output will display the platform project layers:

```
Setting up host-cross and links
Setting up packages link
Creating export directory
Creating project properties
layer          path                                priority
=====
wrlinux        /home/mmorton/Builds/qemux86-64-glibc-small/layers/wrlinux      5
wrlcompat      /home/mmorton/Builds/qemux86-64-glibc-small/layers/wrlcompat    5
wr-sdk-toolchain /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-sdk-toolchain
100
wr-tcwappers   /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-tcwappers  6
meta           /home/mmorton/Builds/qemux86-64-glibc-small/layers/oe-core/meta    5
oe-core-dl-1.7  /home/mmorton/Builds/qemux86-64-glibc-small/layers/oe-core-dl-1.7  0
meta-downloads  /home/mmorton/Builds/qemux86-64-glibc-small/layers/meta-downloads 0
wr-kernel      /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-kernel     6
qemux86-64      /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-bsps/
qemux86-64      7
wr-base         /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-base      6
wr-fixes       /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-fixes     7
wr-tools-profile /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-tools-profile
7
wr-tools-debug   /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-tools-debug  7
meta-networking  /home/mmorton/Builds/qemux86-64-glibc-small/layers/meta-networking 5
meta-oe-subset    /home/mmorton/Builds/qemux86-64-glibc-small/layers/meta-oe-subset  6
meta-python-subset /home/mmorton/Builds/qemux86-64-glibc-small/layers/meta-python-
subset 7
meta-perl-subset  /home/mmorton/Builds/qemux86-64-glibc-small/layers/meta-perl-subset
6
meta-webserver    /home/mmorton/Builds/qemux86-64-glibc-small/layers/meta-webserver  6
wr-prebuilt      /home/mmorton/Builds/qemux86-64-glibc-small/layers/wr-prebuilt     0
```

```
extra-downloads      /home/mmorton/Builds/qemux86-64-glibc-small/layers/extra-downloads 0
local               /home/mmorton/Builds/qemux86-64-glibc-small/layers/local    1000
```

When you use the **bitbake-layers flatten** command, you only need to specify the layer name from the output, and not the path and layer name.

Step 3 Combine platform project layers.

Options	Description
Specified layers	Use the bitbake-layers flatten command and specify the layers you want to combine. For example: <code>bitbake-layers flatten meta-webserver meta-networking meta-user-layer</code> Once the command completes, the projectDir/bitbake_build/meta-user-layer will be a valid layer with content from both the meta-webserver and meta-networking layers.
All layers	Use the bitbake-layers flatten command and specify the output directory only. For example: <code>bitbake-layers flatten all-layers</code> Once the command completes, the projectDir/bitbake_build/all-layers will be a valid layer with content from all project layers.
NOTE: This command may take some time to complete, and requires a considerable amount of disk space.	

Step 4 Optionally relocate your new, combined layer.

Copy or move the layer to another directory for source control management or another platform project for reuse. For additional information, see [Enabling a Layer](#) on page 160.

9

Recipes

[**About Recipes**](#) 167

[**Creating a Recipe File**](#) 170

[**Identifying the LIC_FILES_CHKSUM Value**](#) 171

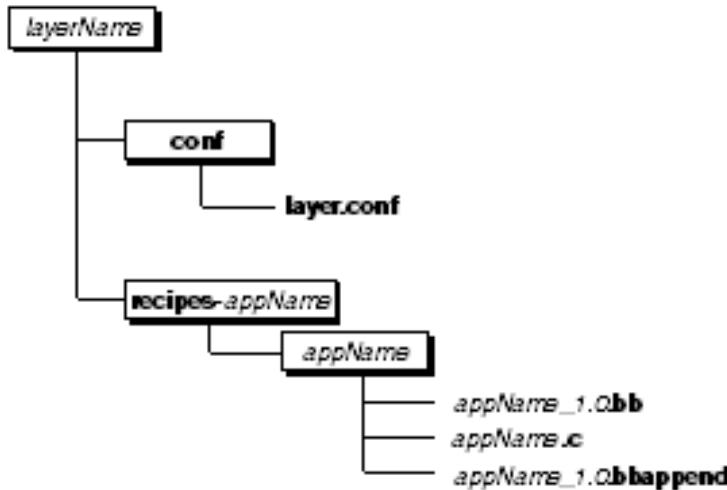
About Recipes

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 170
- [Metadata](#) on page 40
- [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 170
- [Metadata](#) on page 40

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 71 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 170 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 can use **.bbappend** files to supplement an existing recipe file with new information, 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 156 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

Creating a recipe file for an application package in your platform project is necessary if you want your application package to be included in your platform project image each time the project is configured, reconfigured, or built.

After a new package is added to a layer in your platform project, it requires a recipe file to match the new package contents. See [About Recipes](#) on page 167 for general information.

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 215.

See [About Recipes](#) on page 167 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 171.

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 my-app.addpkg
```

Step 8 Build the package.

```
$ make 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 171.

In this case, you may need to perform steps 2 through 4 to update the *LIC_FILES_CHECKSUM* value.

Related Links

[About Layers](#) on page 151

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

https://wiki.yoctoproject.org/wiki/Recipe_%26_Patch_Style_Guide

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">Run the md5sum command on the license file. <pre>\$ md5sum layers/local/recipes-local/my-app/LICENSE</pre>The system returns the checksum value, for example: <pre>2ebc7fac6e1e0a70c894cc14f4736a89 LICENSE</pre>
Use the build system	<ol style="list-style-type: none">Enter the numerical value only in the md5= section. For example: <pre>LIC_FILES_CHKSUM = "file://LICENSE;md5=f27def1e96c2e1ecd4e0c9be8967949"</pre>Run the make command to build the package recipe. <pre>\$ make recipeName</pre>The build will fail. This is expected.Scan the build output for the license checksum value. For example: <pre>ERROR: my-app: md5 data is not matching for file://LICENSE;md5=8e7a4f4b63d12edc5f72e3020a3ffae8 ERROR: my-app: The new md5 checksum is 2ebc7fac6e1e0a70c894cc14f4736a89</pre><ul style="list-style-type: none">The first line states that the md5 checksum from the package's recipe file is incorrect.The second line provides the correct md5 checksum value. Use this value to update the LIC_FILES_CHKSUM md5= value.

Related Links

[The Yocto Project Poky Reference Manual: Track License Change](#)

[About Recipes](#) on page 167

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

10

Templates

[About Templates](#) 173

[Adding Feature Templates](#) 174

[Adding Kernel Configuration Fragments](#) 174

[Template Processing](#) 175

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 61, [Kernel Configuration Fragments in the Project Directory](#) on page 67, and [Template Processing](#) on page 175.

Adding Feature Templates

Use feature templates to add additional functionality to your platform project.

This example procedure shows how 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 \
--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 \
--with-template=feature/debug,feature/analysis
```

For additional information, see [Configure Options Reference](#) on page 82.

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 61.

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-kernel=standard+features/debugfs/debugfs-config.scc
```

For additional information, see [Configure Options Reference](#) on page 82

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 67.

Template Processing

Understanding how Wind River Linux processes templates enables you to make efficient and optimal use of templates.

Templates are processed by the default template first, followed by the order specified on the command line, for example, `--enable-rootfs=rootfs+feature1+feature2+featureN`, followed by `--with-template=feature/3,feature/4,feature/N`.

Creating templates that have order dependencies should not be attempted, as it can lead to confusing output, and because processing order is subject to change in the future.

There are three files that determine how templates are processed:

template.conf

This file can contain any BitBake formatted content.

The `template.conf` files are added to the `bitbake_build/conf/local.conf`, at the end, as `require <path>` statements.

image.inc

This file contains BitBake formatted content that is added to the selected image recipe by the build system. It usually consists of either `IMAGE_FEATURES`, or `IMAGE_INSTALL` variables.

The items from these files are directly copied into the generated `local/recipes-img/images/<image>.bbappend` file.

require

This file allows a template to require another template automatically.

By using the `require` file, we can ensure that the required configurations will be loaded only once, even if multiple templates require them.

The format of this file is either a comment line starting with a #, or a template to include. The template should be in the same format as would be passed to the `configure` command using the `--with-template=` option. One required template per line is expected.



NOTE: The `require` load order is not specified, so it may be loaded either before or after. Therefore, necessary steps should be taken to ensure the correct values in either case.

The directory format of the layers is as follows:

layer/templates/default

This contains the template that is loaded when that specific layer is used. This should contain any required actions, usually documented in the layer's README file, that are required for this layer to function properly.

layer/templates/feature

This contains feature templates organized by *name*. The system is designed such that templates should be small individual pieces of functionality. This way an end user can combine templates to produce a feature rich system. It is suggested to keep features as simple and compact as possible.

11

Finalizing the File System Layout with changelist.xml

About File System Layout XML Files	177
About File and Directory Management with XML	178
Device Options Reference	178
Directory Options Reference	179
File Options Reference	180
Pipe Options Reference	181
Symlink Options Reference	182
The Touched/Accessed touch.xml Database File	182

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
- 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 209). 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**.

The following tables show required and optional fields for adding a device to the system using the **filesystem/changelist.xml** file.

NOTE: Adding a file to **/dev** in this manner currently does not work. Files in **/dev** are managed by the standard Linux **udev** daemon which creates most files at boot time using lists of rules. See **udev** documentation to learn how to write a rule for your device.

Required Fields

Field and Value	Description
action=addbdev or action=addcdev	Use the addbdev action to add a block device. umode does not work with this action.
	Use the addcdev action to add a char device. umode does not work with this action.
name= <i>deviceName</i>	Name of the device directory added to the target file system
umode= <i>permissions</i>	The user/group/other permissions of the target symlink, in octal.

Field and Value	Description
major = <i>majorNumber</i>	The major number for this device
minor = <i>minorNumber</i>	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 <code>chown</code> command).
gid = <i>groupname</i>	The group name, in text or in numeric form (as with the <code>chgrp</code> 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 = <code>addir</code>	Use the <code>addir</code> 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=permissions	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=username	The user name, in text or in numeric form (as with the chown command).
gid=groupname	The group name, in text or in numeric form (as with the chgrp command).

Examples

```
<cl action="addir" name="/usr/share/f_dir1" />
<cl action="addir" name="/usr/share/f_dir2" umode="777" />
<cl action="addir" name="/usr/share/f_dir3" size="10" />
<cl action="addir" 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 <code>chmod</code> command).
size=size	Where <code>size</code> 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 <code>chown</code> command).
gid=groupname	The group name, in text or in numeric form (as with the <code>chgrp</code> 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=pipeName	Name of the directory added to the target file system

Optional Fields

Field and Value	Description
uid=username	The user name, in text or in numeric form (as with the <code>chown</code> command).
gid=groupname	The group name, in text or in numeric form (as with the <code>chgrp</code> 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=addsymlink	Use the addsymlink 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="addsymlink" name="/usr/share/f_sym1" target="/usr/share/f_fool" />
<cl action="addsymlink" name="/usr/share/f_sym2" target="/usr/share/f_fool" />
<cl action="addsymlink" name="/usr/share/f_sym3" target="/usr/share/f_fool"
size="10" />
<cl action="addsymlink" name="/usr/share/f_sym4" target="/usr/share/f_fool"
uid="user3" gid="group3" />
```

The Touched/Accessed touch.xml Database File

You can define and import the file system layout with a custom version of the **touched.xml** file populated with appropriate XML. You can save it with 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"/>
```

```
<c1 action="touched" name="/etc/passwd"/>
<c1 action="touched" name="/etc/inittab"/>
<c1 action="touched" name="/etc/nsswitch.conf"/>
<c1 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.....	187
Understanding the User Space and Kernel Patch Model.	229
Patching Userspace Packages.....	235
Modifying Package Lists.....	243
Maintaining Package Recipe Revisions.....	251

Developing Userspace Applications

About Application Development	187
Creating a Sample Application	196
Exporting the SDK	199
Using the SDK	204
Adding Applications to a Platform Project Image	209
Importing Packages	215
Listing Package Interdependencies	221
About the BitBake Build Environment Display Tool	222

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 us 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 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 57.

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 199. 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 199.

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

Multilibs are enabled by adding the **MULTILIB=** declaration to your project's **local.conf** file.

See [Configuration Files and Platform Projects](#) on page 41 for more information.

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-it), 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 190.

Step 5 Build the platform project image.

```
$ make
```

Adding Multilib Packages

Adding multilib packages involves using `make packageName.addpkg`, and then building and verifying the package.

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 189.

Step 1 Navigate to the `projectDir`.

```
$ cd projectDir
```

Step 2 Add the package.

```
$ make lib32-zlib.addpkg
```

Step 3 Build the package.

```
$ make 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 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

You can add multilib support to the SDK for a specific development need..

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.

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 = "packagegroup-core-standalone-sdk-target packagegroup-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 199.

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-7.0.0.0-glibc-x86_64-qemux86_64-wrlinux-image-glibc-small-sdk.sh`

Adding Static Library Support to the SDK

If you require specific static libraries for your development needs, you can add a single static library, or all static libraries to an 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.

Step 1 Choose an option for adding static libraries.

Options	Description
Single library	Update the <code>projectDir/local.conf</code> file with the following information: <code>TOOLCHAIN_TARGET_TASK_append = " lib_recipeName"</code> NOTE: The leading space prior to <code>lib_recipeName</code> is required by the build system.
All static libraries	In this example, <code>lib_recipeName</code> refers to the recipe name for the specific static library. Update the <code>projectDir/local.conf</code> file with the following information: <code>SDKIMAGE_FEATURES = "staticdev-pkgs dev-pkgs dbg-pkgs"</code>

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 199.

```
$ 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:

About Obtaining Package Source not Provided by Wind River

Due to licensing restrictions, the Wind River Linux build system does not include the source for all packages available.

When you add a package to a platform project image, it is possible for the package to not be added and built if the build system cannot find the package source locally, or if one of the following is not set in the relevant platform project image configuration file(s):

LICENSE_FLAGS

In the package recipe, this setting defines the license types that the build system will process. It is comprised of one or two parts, depending on the license:

```
LICENSE_FLAGS = "license type provider name"
```

- The license type allows the **commercial** or **non-commercial** entries.

A **commercial** license type indicates there is a component of the package that has some type of commercial requirements. These include intellectual property (such as a patent), or some similar license restriction that prevents the package or component from being used without additional review or permission.

A non-commercial license type indicates that there is some clause in the license type that prevents the package or component from being used commercially.

- The provider name is optional, and is designed to allow the recipe creator to group a serial of packages with similar IP restrictions together. For example, in a Wind River Linux installation, some of the recipes for the provided packages are set to:

```
LICENSE_FLAGS = "commercial_windriver"
```

Typically, only one license type is set for a package or component. However, it is possible to set additional license types, depending on your requirements. To set additional license types, add the license type, separated by a space, for example:

```
LICENSE_FLAGS = "commercial_windriver commercial_provider1 non-commercial"
```

To include all commercial and non-commercial license types, for all providers:

```
LICENSE_FLAGS = "commercial non-commercial"
```

Note that excluding the provider name includes all provider licenses with the license type.

LICENSE_FLAGS_WHITELIST

Where the **LICENSE_FLAG** determines the license type applicable to the specific package or component, this setting in the **projectDir/local.conf** file determines which license types are allowed in the platform project build. To enable specific license types, add them to this setting in the **projectDir/local.conf**:

```
LICENSE_FLAGS_WHITELIST += "commercial_provider1 non-commercial_provider1"
```

To include all commercial and non-commercial licenses, for all providers:

```
LICENSE_FLAGS_WHITELIST += "commercial non-commercial"
```

For packages or components not included with your Wind River Linux installation, due to licensing requirements or restrictions, you can still add these types of components to your platform project image:

The **--enable-internet-download configure** option

When you configure a platform project, this setting allows the build system to go to the Internet to retrieve package source. This source is maintained within the layer where the package or component resides in the build system. This retrieved source is specific to the platform project image build. If you choose this option, your build host must have an internet connection.

The **make extra-downloads** command

Once you have built a platform project image, this command allows you to create a layer that includes the source for all packages or components not included with your Wind River Linux installation. The resulting layer is located at **projectDir/layers/extra-downloads/downloads**. The benefit of this approach is that the layer is portable, and may be used in other platform project builds, and by other build hosts that do not have an active Internet connection.

This command looks for entries in the **layer.conf** file, **EXTRA_DOWNLOAD_RECIPES_append** setting, as a confirmation for downloading the package or component source. If the component is not listed, the command will return a warning to ensure that the component is added to the list.

An example setting from a qemux86-64 platform project build with a glibc-small root file system, **projectDir/layers/oe-core-dl-1.7/conf/layers.conf** file is as follows:

```
EXTRA_DOWNLOAD_RECIPES_append = " qmmp libomxil libmad lame libav x264 \
gst-fluendo-mpegdemux gstreamer1.0-plugins-ugly gstreamer1.0-libav \
gst-plugins-ugly gst-ffmpeg gstreamer1.0-omx gst-openmax \
gst-fluendo-mp3 mpeg2dec \"
```

For packages to be built once the source is obtained, the license requirement settings explained in this section must be set.

Creating a Reusable Layer with Downloaded Package Source

Create a layer with package source not provided with your Wind River Linux installation.

The following procedure requires that:

- You have a previously configured and built platform project—see [About Configuring a Platform Project Image](#) on page 77 and the *Wind River Linux Getting Started Guide: Developing a Platform Project Image Using the Command Line*.
- You are familiar with package or component licensing requirements described in [About Obtaining Package Source not Provided by Wind River](#) on page 192. Understanding the build system requirements concerning license-related settings will help you with other packages or applications you may need to add to your platform project image.

Step 1 Create the downloaded package source layer.

- a) Navigate to the platform project directory that you wish to create the package source layer in..
- b) Create the layer.

```
$ make extra-downloads
```

After the command finishes processing, it creates a layer directory at **projectDir/layers/extra-downloads/downloads**. If the command provides a warning that a specific package or packages was not built, you must ensure that the package is added to the **EXTRA_DOWNLOAD_RECIPES_append** option in the relevant **layer.conf** file.

- c) Optionally review the contents of the layer directory.

```
$ ls layers/extra-downloads/downloads
git2
lame-3.99.5.tar.gz qmmp-0.7.7.tar.bz2
libav-0.8.15.tar.xz
libmad-0.15.1b.tar.gz
libomxil-bellagio-0.9.3.tar.gz
gst-ffmpeg-0.10.13.tar.bz2
```

```
gst-fluendo-mp3-0.10.19.tar.bz2
gst-fluendo-mpegdemux-0.10.72.tar.bz2
gst-libav-1.4.1.tar.xz
gst-omx-1.2.0.tar.xz
gst-openmax-0.10.1.tar.bz2
gst-plugins-ugly-1.4.1.tar.xz
gst-plugins-ugly-0.10.19.tar.bz2
mpeg2dec-0.4.1.tar.gz
netperf-2.6.0.tar.bz2
```

Note that the output reflects the package names added to the `projectDir/layers/oe-core-dl-1.5/conf/layers.conf` file, `EXTRA_DOWNLOAD_RECIPES_append` setting, as described in [About Obtaining Package Source not Provided by Wind River](#) on page 192.

Step 2 Optionally install a package from the downloaded source.

In this example, you will install the **lame** MP3 encoder package.

- Update the license setting in the **local.conf** file.

Since the **lame** package has a commercial license, you will need to update the `projectDir/local.conf LICENSE_FLAGS_WHITELIST` option as follows:

```
LICENSE_FLAGS_WHITELIST += "commercial"
```



NOTE: The license type is located in the package's recipe (.bb) file, in the `LICENSE_FLAGS` setting.

- Build the **lame** package.

```
$ make lame
```

Building the package takes a couple of minutes, during which you will see the progress on your terminal.

- Add the **lame** package.

```
$ make lame.addpkg
```

The build system will add the built package to the platform project image. Once the command completes, it will return:

```
==== ADDED lame to ../../layers/recipe-img/images/wrlinux-image-glibc-small.bbappend ===
```

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.

- Rebuild the platform project.

```
$ make
```

Rebuilding the file system should take just a couple of minutes this time, because only the newly added elements need to be built. The **lame** application is now part of your platform project image.

Step 3 Optionally share the layer with another build host.

This step adds downloaded source to a platform project image on a build host without an Internet connection.

To share the contents, simply copy the `projectDir/layers/extra-downloads/downloads` directory to the platform project location on the new build host.

- a) Copy the layer contents to the new build host.

To share the contents, simply copy the `projectDir/layers/extra-downloads/downloads` directory to the platform project location on the new build host.

- b) Rebuild the platform project on the new host.

From the platform project directory on the new host, enter:

```
$ make
```

If you need to build a package from the `projectDir/layers/extra-downloads/downloads` layer, the source is available without requiring an Internet connection, provided that the licensing requirements are set as described in [About Obtaining Package Source not Provided by Wind River](#) on page 192.

Creating a Sample Application

Develop a typical C application that calculates a specified number of terms in the Fibonacci series to illustrate the use of multiple source files, headers, a license file and **make**.

The Fibonacci series takes the number of terms as a command-line parameter.

This application 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 215.

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 199.

 **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.

- Create the header **math.h** file with vi.

```
$ vi makefile
```

- 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 *~
```

- Save the file.

Step 6 Create the **LICENSE** file.

- Create the **LICENSE** file with vi.

```
$ vi LICENSE
```

- 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.
```

- 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 199.

- Navigate to the project directory for the application.
- 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.

- Copy the program binary to the platform project target file system **usr/bin** directory.

```
$ fibonacci ~/Builds/qemux86-64_small/export/dist/usr/bin
```

- Navigate back to the platform project directory.

```
$ cd ..
```

- Launch the platform project in an emulator.

```
$ make start-target
```

- 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 209.

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 [About Configuring a Platform Project Image](#) on page 77 and the *Wind River Linux Getting Started Guide: Developing a Platform Project Image Using the Command-line*.
- You have read/write privileges to the directory (**/opt** in the instructions below) on your host workstation. You should check the ownership of the destination directory, as a lack of privileges could generate an error when you run **make fs** in the exported SDK directory.

If you do not have write permission to install the SDK into the selected directory, then the installation script checks if you have sudo privilege. If you have sudo privilege, then the installation script creates the **/sdk** directory and assigns owner/group as root/root. In this case, to avoid the error, you must use **sudo make fs** instead of **make fs**, or you can change ownership of files as administrator before executing **make fs**.



NOTE: These privileges are necessary for the project toolchain to install and work properly.

Step 1 Optionally specify the type of SDKs that you want to generate.

By default, an SDK is created using the platform project configuration as a basis. Wind River Linux provides the `--enable-sdkmachine= configure` script option to specify the SDK machine, or machines generated. You can use this to create multiple SDKs, such as one for Linux and another for Windows host development.

For additional information on SDK machine options, see [Configure Options Reference](#) on page 82.

Step 2 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-7.0.0.0-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.

To install the SDK to another Linux development host, copy the script file to the host before proceeding.

Step 3 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-7.0.0.0-glibc-x86_64-qemux86_64-wrlinux-image-glibc-small-sdk.sh
```



NOTE: If you specified additional SDK machines, you will need to run the script for each architecture to install the SDK.

Step 4 Enter the target directory path, or accept the default `/opt/windriver/wrlinux/7.0-qemux86-64`.

Step 5 When prompted, press **Y** and **ENTER** to install the SDK.

The SDK will extract and install to the directory specified in step 4 on page 200.

Postrequisites

Once the SDK is installed, you must source it to use it to develop applications based on your platform project's architecture. For additional information, see [Sourcing the SDK](#) on page 201.

Sourcing the SDK

After you have successfully exported the SDK, you can source it for application development.

The following procedure requires that you have completed the steps in [Exporting the SDK](#) on page 199.

Step 1 Source the SDK.

This step sets the required environment variables for the target architecture 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. For additional information, see [Compiling Programs Using the SDK Environment Variables](#) on page 205.

Options	Description
bash	<pre>\$ source /opt/windriver/wrlinux/7.0-qemux86-64/environment-setup-i586-wrs-linux</pre>
POSIX	<pre>\$./opt/windriver/wrlinux/7.0-qemux86-64/environment-setup-i586-wrs-linux</pre>

The script that you run is based on your selected architecture, for example, **environment-setup-arch-wrs-linux.sh**. Wind River also provides the **sdkDir/env.sh** script. Use this script to source the default SDK architecture.

If you installed the SDK to a directory other than the default, substitute the path to the **environment-setup-core2-64-wrs-linux.sh** script.

Step 2 Optionally rebuild the SDK filesystem.

While not required to use the Wind River Linux SDK, building the filesystem in the **sdkDir/export/dist** directory can enhance your development activities by keeping your application-specific testing separate from the platform project's build directory.

 **NOTE:** The filesystem build in the **sdkDir** is created from the platform project build that the SDK was created from, and does not use the BitBake build system. As a result, no upstream changes are included in the build.

- a) Navigate to the **sdkDir**.

```
$ cd /opt/windriver/wrlinux/7.0-qemux86-64
```

- b) Build the filesystem.

```
$ make fs
```

Once the build completes, the **/opt/windriver/wrlinux/7.0-qemux86-64/export/dist** (**sdkDir/export/dist**) directory will contain the target filesystem ready for application-specific development.

In addition, the **sdkDir/export** directory will contain the platform project filesystem images for deployment.

- c) Optionally generate a filesystem image ready for deployment.

Use this step to generate filesystem image to include any changes you may have made to the `sdkDir/export/dist`, such as repackaging an application into the target filesystem after copying the binaries to the `sdkDir/export/dist` directory.

```
$ make fs-image
```

Once the command completes, the updated filesystem is created in the `fs-image.tar.bz2` file, available in the `sdkDir/export` directory.

Exporting the SDK for Windows Application Development

When you configure a platform project, you can choose to also use an exported SDK to enable support for building the SDK for a Windows host.

Use the following instructions create an SDK suitable for developing applications on a Microsoft Windows host.

Step 1 Specify the type of SDK that you want to create.

Option	Description
Single SDK based on the build host	<p>Configure a new, or reconfigure an existing platform project by adding the <code>--enable-sdkmachine</code>= configure option.</p> <pre>--enable-sdkmachine=auto,i686-mingw32</pre> <p>The <code>--enable-sdkmachine=auto,i686-mingw32</code> option enables support for an SDK based on the build host, and another used for 32-bit Windows development. This is the equivalent of using the <code>--enable-win-sdk=yes</code> configure option.</p> <p>If you are reconfiguring an existing platform project, also include the following option:</p> <pre>--enable-reconfigure=yes</pre>
One or more SDKs	<p>Specify the type of SDKs that you want to generate, including a combination of Linux and Windows SDKs, using the <code>--enable-sdkmachine</code>= configure option.</p> <p>Typically, this includes 32- and 64-bit SDKs to meet your application development requirements. For additional information on SDK machine options, see Configure Options Reference on page 82.</p> <pre>--enable-sdkmachine=auto,i686-mingw32,x86_64-mingw32</pre> <p>The <code>--enable-sdkmachine</code>= option above will cause the build system to create three separate SDKs:</p> <ul style="list-style-type: none">One Linux SDK based on the platform project configurationOne 32-bit Windows SDKOne 64-bit Windows SDK

Step 2 Create the SDK.

Run the following command from the *projectDir*:

```
$ make export-sdk
```

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-7.0.0.0-glibc-i686-mingw32-qemux86_64-wrlinux-image-glibc-small-sdk.zip

This compressed file contains the toolchain for your project that you need for cross-compiling applications. Typically this SDK is provided to the application developer's platform team, who makes it available for installation on the Windows host as described in the next step.

Postrequisites

Once the SDK is installed, you must source it to use it to develop applications based on your platform project's architecture. For additional information, see [Sourcing the SDK for Windows Application Development](#) on page 203.

Sourcing the SDK for Windows Application Development

When you configure a platform project, you can choose to source a previously exported SDK, allowing you to build the SDK for a Windows host.

Use the following instructions to create and extract a SDK suitable for developing applications on a Microsoft Windows host.

Before undertaking this task, you must first have completed the steps in [Exporting the SDK for Windows Application Development](#) on page 202.

Step 1 Copy the .zip file you created in [Exporting the SDK for Windows Application Development](#) on page 202 containing the SDK to your Windows host.

Step 2 On the Windows host, extract the SDK *.zip file to a unique directory.

IMPORTANT: Windows has limitations for character path length that could impact SDK usage. For additional information, see [About Using the SDK](#) on page 204.

NOTE: Certain compressed file managers such as WinZip do not support long file names and will report errors. Use an alternative such as the built-in Windows Explorer .zip file support or 7zip.

Step 3 Optionally, setup the SDK environment for command-line application development.

This step is required to use the command-line in Windows to develop your applications with the SDK environment variables that Wind River provides. For additional information, see [Compiling Programs Using the SDK Environment Variables](#) on page 205.

To set up the environment, run the batch file located in the extracted SDK directory.

```
C:\sdkDir\environment-setup-core2-64-wrs-linux.bat
```

Once the batch file completes, your environment is ready for command-line development.

Step 4 Import the SDK in Workbench.

From the Workbench main menu, select **File > Import > Wind River Linux > Import Wind River Linux 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 ***sdkDir/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.

To compile programs from the command line in Windows, see [Compiling Programs Using the SDK Environment Variables](#) on page 205.

Using the SDK

About Using the SDK

Understanding the available features and environment of an installed and sourced SDK will help you develop applications in Windows and Linux using the command line or Workbench.

Command Line Development

The Wind River Linux SDK provides many options to help you develop applications without a Wind River Linux installation or the BitBake build system.

To accomplish this, Wind River provides **make** commands (Linux-only) to help you manage filesystem tasks such as:

- Decompressing the SDK filesystem and debug symbols
- Creating a filesystem image, in the ***sdkDir*** or on a USB device
- Starting a QEMU simulator, or SIMICS if it's installed

For a list of available make commands, see [SDK Environment make Command Reference](#) on page 206. It's important to note that the SDK directory does not invoke the Wind River Linux build system or require access to upstream BitBake repositories.

In addition to the **make** commands for Linux, Wind River also provides options to source the SDK for Linux and Windows hosts, which makes development environment variables available for use in the Linux and Windows command line. For additional information, see [Compiling Programs Using the SDK Environment Variables](#) on page 205.

Workbench Development

Workbench greatly simplifies SDK use in Linux and Windows, and removes the requirement for sourcing the SDK for application development. For Linux development, builds are forward-portable, so even platform projects developed from the command line may be imported to Workbench directly without the need to also import the SDK.

For Windows development, once you import the SDK, you are ready to develop applications based on the SDK architecture.

Windows Development and Installation Path Limitations

When you install the SDK on a Windows host, you must ensure that the installation location is in compliance with Windows path limitations for the SDK to work as intended. Windows has a limitation of a 256-character path length. Depending on the SDK configuration, some internal paths may use up to approximately 130 characters. This leaves approximately 126 characters for the installation path.

Using Windows short paths, each directory path uses up to nine characters, or eight plus the backslash (\) for the path itself. With 126 characters available in the path, a 14-directory deep path should work; however it is recommended to keep the path no greater than 12-directories deep.

A workaround for the path limitation is to use the Windows **subst** command. For additional information on using this command, see the Windows documentation.

If the SDK path is too long, common failure modes include:

```
command not found  
  
error: CreateProcess: No such file or directory fatal  
  
error: -fuse-linker-plugin, but liblto_plugin-0.dll not found
```

Quick Start SDK for Windows Application Developers

For Windows-based Wind River Linux installations, a quick start 32-bit SDK is provided to help application developers get started for the following architectures:

- ARM
- ia32
- MIPS
- POWER

The SDK is available in the ***installDir/wrlinux-7/SDK/wr-sdk-version/arch-quickstart-686-mingw32/arch*** directory. You may use the SDK directly from the command line or imported into the Workbench environment.

It is important to note that the quick start SDKs are installed, by default, 10 directories deep. If you choose an alternative path for the product installation, it may make the quickstart SDK paths too deep, which can cause failures as described in the previous section..

Compiling Programs Using the SDK Environment Variables

Environment variables provided by Wind River can be used to compile applications from the command-line in Linux or Windows.

Before you compile applications from the command-line, you must export the SDK and source the environment, as described in [Exporting the SDK](#) on page 199 and [Exporting the SDK for Windows Application Development](#) on page 202.

Once your application development is ready, you can use the SDK environment variables to develop your programs from the command line as described in this procedure.

Step 1 Compile the application based on the application type and development host.

The following examples use the Hello Linux (**hello.c**) sample application.

Options	Description
Linux host, C application	<code>\$ \$CC hello.c -o hello</code>
Linux host, CXX application	<code>\$ \$CXX hello.c -o hello_cxx</code>
Linux host, configure and compile a GNU autoconf application	<code>\$./configure \$CONFIGURE_FLAGS \$ make</code>
Windows host, C application	<code>C:\sdkDir\%CC% hello.c -o hello</code>
Windows host, CXX application	<code>C:\sdkDir\%CXX% hello.c -o hello_cxx</code>
Windows host, configure and compile a GNU autoconf application	GNU autoconf is not included with the Windows SDK.

SDK Environment make Command Reference

The Wind River Linux SDK folder and environment provides **make** commands that let you create file systems and launch QEMU instances based on the SDK.

Command	Description
make	Run without options to display the list of available make commands with descriptions.
make fs	Decompress the filesystem to the <i>sdkdir/export/dist</i> directory.
make fs-debug	Decompress the debug symbols for the filesystem.
make fs-image	Create a new *.tar.bz2 file system in the <i>sdkdir/export</i> directory.
make usb-image	Supports x86 only. Build a USB image in the <i>sdkdir/images</i> directory.
make usb-image-loop	Supports x86 only. Build a USB image with a loop device.
make usb-image-burn	Supports x86 only. Write a USB image directly to a USB device.
make fakeroot	Enter the fakeroot shell in the filesystem. This simulates root access in the <i>sdkdir/export/dist</i> directory to allow you to make filesystem changes.

Command	Description
make start-qemu	Supports Linux hosts only. Starts a QEMU simulator from the filesystem in the sdkDir .
make stop-qemu	Supports Linux hosts only. Stops the QEMU simulator session started with the make start-qemu command.
make config-target-qemu	Supports Linux hosts only. Displays configuration options for the QEMU simulator launched from the sdkDir .
make start-simics	Starts a SIMICS simulator if installed.
make stop-simics	Stops the SIMICS simulator session started with the make start-simics command.
make config-target-simics	Displays configuration options for the SIMICS simulator if installed.
make reset	Erase and restore the state files in the SDK.
CAUTION: This command will reset the SDK files and remove any changes or additions made. Use with caution.	
OBJCOPY	Minimum command and arguments to run objcopy
OBJDUMP	Minimum command and arguments to run objdump
PKG_CONFIG_PATH	Path to pkg-config files for the target
RANLIB	Minimum command and arguments to run ranlib
SDKROOT	Path to the installed SDK directory
SDKTARGETSYSROOT	Path to the sysroot that will be used for cross-compilation
STRIP	Minimum command and arguments to run strip to strip symbols
TARGET_PREFIX	Toolchain binary prefix for the target tools

SDK Environment Variables Reference

These options are available to you for compiling your applications for the Linux or Windows command-line after setting up the Wind River Linux environment for SDK development.

Variable	Description
AR	Minimum command and arguments to run ar .
AS	Minimum command and arguments to run the assembler.

Variable	Description
CC	Minimum command and arguments to run the C compiler
CFLAGS	Suggested C flags
CONFIG_SITE	A gnu autoconf site file pre-configured for the target
CONFIGURE_FLAGS	Minimum arguments for GNU configure
CPP	Minimum command and arguments to run the C preprocessor
CPPFLAGS	Suggested preprocessor flags
CROSS_COMPILE	Toolchain binary prefix for the target tools (same as TARGET_PREFIX)
CXX	Minimum command and arguments to run the C++ compiler
CXXFLAGS	Suggested C++ flags
GDB	Minimum command and arguments to run the GNU Debugger
LD	Minimum command and arguments to run the linker
LDFLAGS	Suggested linker flags when using CC to link
NM	Minimum command and arguments to run nm
OBJCOPY	Minimum command and arguments to run objcopy
OBJDUMP	Minimum command and arguments to run objdump
PKG_CONFIG_PATH	Path to pkg-config files for the target
RANLIB	Minimum command and arguments to run ranlib
SDKROOT	Path to the installed SDK directory
SDKTARGETSYSROOT	Path to the sysroot that will be used for cross-compilation
STRIP	Minimum command and arguments to run strip to strip symbols
TARGET_PREFIX	Toolchain binary prefix for the target tools

Adding Applications to a Platform Project Image

Options for Adding an Application to a Platform Project Image

There are several methods available for adding 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 recipeName.addpkg
```

For information on using this command, see [Adding New Application Packages to an Existing Project](#) on page 209.

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 215.

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 212.

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 211 and [About File System Layout XML Files](#) on page 177.

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 213.

Adding New Application Packages to an Existing Project

You can add packages to an existing, previously configured and built platform project using **make packageName.addpkg**.

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.

- Navigate to the *projectDir*.

```
$ cd projectDir
```

- Add the **gdb** package.

```
$ make 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.

- 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 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 214.

Adding an Application to a Root File System Using `changelist.xml`

Use the `changelist.xml` file to add an application 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 [About File and Directory Management with XML](#) on page 178.

The location of the script file is inside the `projectDir/layers/local` directory (see [About the layers/local Directory](#) on page 71). 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 196.

Step 1 Create the `changelist.xml` file.

- Navigate to the platform project top-level directory.

```
$ cd ~Builds/qemux86-64_small
```

- Create the `changelist.xml` file with `vi`.

```
$ layers/local/conf/image_final/changelist.xml
```

- 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/wruser/Builds/qemux86-64_small/Fibonacci/fibonacci"/>
</change_list>
</layout_change_list>
```

NOTE: In the example above, replace `/home/wruser/Builds/qemux86-64_small` with the path to 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 [About File and Directory Management with XML](#) on page 178 for additional information.

- 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 212). 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, there are several ways to add it to the root file system of your platform project image.

You need a platform project configured and built using the following configure options before proceeding:

```
--with-layer=examples/fs-final  
--with-template=feature/example-fs-final
```

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 71). 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 196, to add the application to the target's root file system.

Step 1 Create the `fs_final*.sh` script for the application.

- Navigate to the platform project top-level directory. For example:

```
$ ~Builds/qemux86-64_small
```

- Create the `fs_finalfibonacci.sh` script with `vi`.

```
$ vi layers/local/conf/image_final/fs_final_fibonacci.sh
```

- Enter or copy the following text.

```
#  
# Add the fibonacci binary to the root file system  
#  
make -C /home/wruser/Builds/qemux86-64_small/Fibonacci  
cp /home/wruser/Builds/qemux86-64_small/Fibonacci/fibonacci usr/bin
```

NOTE: In the example above, replace `/home/wruser/Builds/qemux86-64_small` with the path to 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 requirement 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

You can add a package to a project when you first configure the project by using the **--with-package** option.

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 214

Verifying the Project Includes the New Application Package

Verify that a package was added successfully to a platform project and can be run on the target.

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 213 or *Adding New Application Packages to an Existing Project* on page 209.

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.

- Deploy the platform project on a target.

For additional information, see the *Wind River Linux Getting Started Guide: Developing a Platform Project Image Using the Command-Line*.

- 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 (GDB) 7.7.1
Copyright (C) 2014 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:<http://www.gnu.org/
software/gdb/documentation/>.
For help, type "help". Type "apropos word" to search for commands related to "word".
(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 **mthread** 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
```

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 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-7/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 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 171 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 216, 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 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)

Import a source application package from the web using the Package Importer Tool.

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
```

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/layers/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 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 171.

- 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 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 the Package Importer Tool 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 7.0, 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
```

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 Version: About the Package Importer Tool (import-package)*, for additional information.

Step 2 Import the package contents.

- Set the **Package Type** selection to **Source Package**.
- 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.

- Click **Import**.

The Progress field displays the progress of the update.

- Click **Close** to close the Package Importer tool.

Step 3 Update the recipe file and build the package.

- 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
```

- 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"
```

- 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"
```

- Save the recipe file.
- Build the package:

```
$ make 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 171.

- 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 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  pod2htmi.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 inter-dependencies between packages.

Many of the packages compiled by the bitbake build system have inter-dependencies. If one package is present another is required, or is compiled differently. It can be difficult to find these inter-dependencies, as the package recipes can be spread over several files in multiple layers. In order to help you understand the inter-dependencies 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
```

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+_3.8.2.bb
:
```

About the BitBake Build Environment Display Tool

The **wrbutil show-env** tool is an external program that is used to extract the environment data without having to load the entire BitBake system.

BitBake recipes specify how a particular package is built. It includes all the package dependencies, locations, configuration, compilation, build, install, and remove instructions. Recipes store the metadata for the package in standard variables. During the build process they are used to track dependencies, performing native or cross-compilation of the package, and package it, so that it is suitable for installation on the local or a target device. Your requirements may include the collection of this information.

Wind River provides the **wrbutil show-env** tool easily output this information with various options to show the environment variables. Run the tool after the platform project image has been built.

Description and Usage

wrbutil show-env can be run from inside the Wind River **BitBake Shell** which is started with the **make bbs** command.

Or alternately, **wrbutil show-env** can be invoked in a single command as an option when running the **make bbc** command from the users shell without the need to use an interactive **BitBake Shell**. See [Displaying the BitBake Environment Details](#) on page 226.

The **wrbutil show-env** command can be run in the same **BitBake Shell** where commands to build the image were issued, but it can work in an entirely different session started by either **make** or another **BitBake Shell** session. As long as the commands are run from the same build directory, they are synchronized.

Environment information in the BitBake database exists in the context of a package generated by a recipe. This context is specified by the use of an option for the **wrbutil show-env** command. See [BitBake Environment Display Tool Reference](#) on page 227.

The **wrbutil show-env** command provides several output types:

sh(1) Variable Assignment Statements

Outputs a series of lines that look like variable assignment statements of the format, depending on the attributes about that particular variable that is recorded in the BitBake database. For example the output may look like any of the following:

```
VAR=value
```

or

```
export VAR=value
```

or

```
unset VAR
```

Recipe Function Definitions

Provides a set of output that represents all of the function definitions that are part of the recipe context being examined. One type is a **sh(1)** function, in which case the output is likely to be several lines as shown in the following example:

```
funcname() {  
line-1-sh-command  
LINE2SHVAR=value  
...  
}
```

Recipe Function Definitions - Python

Another type of output is a Python function definition, in which case the function name is prefixed by **python** and encapsulates Python code as shown in the following example:

```
python pyfuncname() {  
def pyfuncname(arg1):  
    " This is a python function "
```

```
    return True
}
```

Variables specified on the command line but not defined in the context being displayed are silently ignored. There will be informational messages printed during the initialization of **wrbutil show-env** to standard error, but all output related to the environment data will be emitted to standard output.



NOTE: If you run the following example, the tool will display informational data to the terminal while retrieving the environment data, but the file **/tmp/environment** will contain the extracted environment data:

```
$ wrbutil show-env > /tmp/environment
```

Tool Option Examples

See [BitBake Environment Display Tool Reference](#) on page 227.

-r option

To display the entire environment context of another context (that is the context of part of the output), use the **-r recipeName** option to specify the recipe that generates that package. For example, to show the context while building the **zip** package, the command would look like this:

```
$ ../../layers/wrlcompat/scripts/wrbutil show-env -r zip
```

Much of the output will be similar to the global context output, but some variables specific to the **zip** build will be different. For example:

```
...
# PN="${@bb.parse.BBHandler.vars_from_file(dgetVar('FILE'),d)[0] or
'defaultpkgname'}"
PN="zip"
PR="r2"
# PRAUTOINX="${PF}"
PRAUTOINX="zip-3.0-r2"
...
}
```

If the **-r** (or **--recipe**) option is not specified, then the environment data presented by **wrbutil show-env** is taken from the global context. This global context is not specific to any particular package (or image), but rather the default values before being defined by a particular package recipe.



NOTE: If the **-r** option specifies a recipe name, then the environment data is that which was present during the build processing of that particular recipe.

As an example, in the global context, the variable **PN** will have as its value the name **defaultpkgname**. But in the context of a package such as a **zip** file, the **PN** will have the name of the **zip** package. Similarly, the **do_install()** function in the image context will likely be different than the **do_install()** function shown in the **zip** recipe context.

-f option

The **-f** or **--flags** option requests output to include the internal BitBake flag attributes associated with a particular variable. The flags (if present) are added within square brackets following the variable name.

For example in the global context, the *DISTRO* variable has several flags associated with it. With the normal **wrbutil show-env** command, the output will generate a shell unset command because of those internal flags:

```
$ ../../layers/wrlcompat/scripts/wrbutil show-env DISTRO
```

Note the result:

```
...  
unset DISTRO
```

But if you add the **-f** flag you will see the variable displayed as an assignment statement with the flags documented:

```
$ ../../layers/wrlcompat/scripts/wrbutil show-env -f DISTRO
```

Note the result:

```
...  
DISTRO[doc,unexport,defaultval]=wrlinux
```



NOTE: If flags are present for the requested output, the format of the output will no longer be valid shell **sh(1)** syntax with this option.

The values generated by **wrbutil show-env** are by default formatted to be acceptable for use within shell scripts. However, variables in BitBake may contain special or non-printable characters. These may either cause problems when being interpreted by the shell, or it may remove some of the formatting of the value when the assignment is interpreted.

-q option

The **-q** or **--quote** option alters the generation of the value side of assignment statements to use backslash escape sequences for all special or non-printable characters. These escape sequences are the usual C library conventions suitable for use by the **printf()** class of string formatting.

To limit the output of **wrbutil show-env** to one or more specific named elements of the environment, append those names at the end of the command line. For example, in the global context you could display the image name along with the default package name with the following command:

```
$ ../../layers/wrlcompat/scripts/wrbutil show-env DEFAULT_IMAGE PN
```

Which results in the following output:

```
...  
DEFAULT_IMAGE="wrlinux-image-glibc-std"  
# PN="@bb.parse.BBHandler.vars_from_file(dgetVar('FILE'),d)[0] or 'defaultpkgname'"  
PN="defaultpkgname"
```

Or, in the context of the image the same output would be obtained with the following command:

```
$ ../../layers/wrlcompat/scripts/wrbutil show-env -r $DEFAULT_IMAGE DEFAULT_IMAGE PN
```

Which results in the following output:

```
DEFAULT_IMAGE="wrlinux-image-glibc-std"
# PN="${@bb.parse.BBHandler.vars_from_file(dgetVar('FILE'),d)[0] or 'defaultpkgname'}"
PN="wrlinux-image-glibc-std"
```

For the zip package you can refine the output with the following command:

```
$ ./layers/wrlcompat/scripts/wrbutil show-env -r zip DEFAULT_IMAGE PN
```

Which results in the following output:

```
DEFAULT_IMAGE="wrlinux-image-glibc-std"
# PN="${@bb.parse.BBHandler.vars_from_file(dgetVar('FILE'),d)[0] or 'defaultpkgname'}"
PN="zip"
```

Displaying the BitBake Environment Details

View BitBake environment information for a project using the **wrbutil show-env** tool.

To display the context of the image produced by the build, first query BitBake for the value of the **DEFAULT_IMAGE** variable, and then provide that value as the name of the recipe corresponding to the image and run the **wrbutil show-env** command.

You must have built a platform project image before performing the following steps:

Step 1 Enter the BitBake shell to run the command.

Options	Description
make bbs	Enter the BitBake shell from the project directory where you have built the platform project image in order to run the make bbs command. \$ make bbs
make bbc BBCMD	Or, run the make bbc from the project directory where you have built your platform project image. \$ make bbc BBCMD=".../layers/wrlcompat/scripts/wrbutil show-env"

If using this option then you can skip Step 2.

Step 2 Built the **DEFAULT_IMAGE**.

The following command builds the image. Note that the image name can also be found in the **local.conf** file:

```
$ bitbake $DEFAULT_IMAGE
```

Step 3 Run the **wrbutil show-env** command.

Now that you are in the BitBake shell you can run the tool.

```
$ ./layers/wrlcompat/scripts/wrbutil show-env
```

This command will output the environment from the context of the global build. It may look like the following output, depending on your build:

```
# ABIEXTENSION="${@bb.utils.contains(\"TUNE_FEATURES\", \"mx32\", \"x32\", \"\", d)}\"  
ABIEXTENSION=""  
# ALL_MULTILIB_PACKAGE_ARCHS="${@all_multilib_tune_values(d, 'PACKAGE_ARCHS')}"  
ALL_MULTILIB_PACKAGE_ARCHS="all any noarch x86_64 core2-64 qemux86_64 x86"  
# ALL_QA="${WARN_QA}~${ERROR_QA}"  
ALL_QA="texrel_files-invalid incompatible-license xorg-driver-abi libdir  
ldflags installed-vs-shipped rpaths dev-so debug-deps dev-deps debug-files arch  
pkgconfig la perms useless-rpaths staticdev pkgvarcheck already-  
stripped compile-host-path dep-cmp install-host-path packages-  
list perm-config perm-line perm-link pkvg-undefined pn-overrides split-  
strip var-undefined version-going-backwards"  
...  
# PN="${@bb.parse.BBHandler.vars_from_file(dgetVar('FILE'),d)[0] or 'defaultpkgname'}"  
PN="defaultpkgname"  
# PR="${@bb.parse.BBHandler.vars_from_file(dgetVar('FILE'),d)[2] or 'r0'}"  
PR="r0"  
# PRAUTOINX="${PF}"  
PRAUTOINX="defaultpkgname-1.0-r0"  
...  
zip_sdk() {  
    # Create an SDK archive as a zip file  
    mkdir -p ${SDK_DEPLOY}  
    rm -f "${SDK_DEPLOY}/wrlinux-7.0.0.0-eglibc-x86_64-qemux86_64-defaultpkgname-sdk.zip"  
    cd ${SDK_OUTPUT} //opt/windriver/wrlinux/7.0-qemux86-64  
    zip -rqy "${SDK_DEPLOY}/wrlinux-7.0.0.0-eglibc-x86_64-qemux86_64-defaultpkgname-  
    sdk.zip".  
}
```

In the first set of variable assignment output, lines starting with # would be interpreted as **sh(1)** comments, but would document the internal value of the variable prior to substituting references to other variables and function invocations.

```
# Create an SDK archive as a zip file
```

The line following the comment will be an assignment statement with all substitutions completed. In the prior example, at the end of the output are function definitions, with the last definition for the **sh(1)** function **zip_sdk** shown above, would be as follows:

```
mkdir -p ${SDK_DEPLOY}  
rm -f "${SDK_DEPLOY}/wrlinux-7.0.0.0-eglibc-x86_64-qemux86_64-defaultpkgname-sdk.zip"  
cd ${SDK_OUTPUT} //opt/windriver/wrlinux/7.0-qemux86-64  
zip -rqy "${SDK_DEPLOY}/wrlinux-7.0.0.0-eglibc-x86_64-qemux86_64-defaultpkgname-  
sdk.zip".
```

BitBake Environment Display Tool Reference

Additional options to the **wrbutil show-env** command follow this usage format: **show-env options VARNAME**.

wrbutil show-env	-f --flags	Append flag(s) to variable name.
wrbutil show-env	-q --quote	Map all special/unprintable characters to string escapes.

wrbutil show-env	-r <i>recipeName</i> --recipe <i>packageName</i>	Show environment from context of <i>packageName</i> recipe.
wrbutil show-env	<i>VARNAME</i>	Show only the variable named <i>VARNAME</i> .

13

Understanding the User Space and Kernel Patch Model

[Patch Principles and Workflow 229](#)

[Patching Principles 230](#)

[Kernel Patching with scc 231](#)

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 281 and [Introduction to Patching Userspace Packages](#) on page 235

Patching Principles

Understanding patching and patch troubleshooting principles is important for success in the development process.

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, white space, 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 297.

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 its 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
#
#kconf hardware common_pc.cfg
# ^      ^
# |      +- 'type: hardware or non-hardware
# |
# +--- kernel config

# patches
patch 0002-at12-add-at12-driver.patch
patch 0003-net-remove-LITX-in-at12-driver.patch
patch 0004-net-add-net-poll-support-for-at12-driver.patch
```


Patching Userspace Packages

Introduction to Patching Userspace Packages	235
Patching with Quilt	237
Create an Alias to exportPatches.tcl to save time	237
Preparing the Development Host for Patching	237
Patching and Exporting a Package to a Layer	238
Verifying an Exported Patch	240
Incorporating a Patch into a Platform Project Image	241

Introduction to Patching Userspace Packages

Certian requirements must be met 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-udhcpc-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-7/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 237.

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 *Glibc File Systems* on page 136.

Patching with Quilt

Quilt is a general-purpose patching mechanism that you can use whenever you are working with patches to update packages as well as custom applications.

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-7/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 237.

Preparing the Development Host for Patching

tk must be installed to provide the **wish** interpreter before you begin patching userspace packages.

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 the following commands to install the **tk** package, required to run the **wish** interpreter and **exportPatches.tcl** script:

Options	Description
Ubuntu/Debian-based host	\$ sudo apt-get install tk
RedHat/RPM-based host	\$ sudo yum install tk

Patching and Exporting a Package to a Layer

Patch a package in your platform project image and export it to a layer using Quilt..

The following steps use the **which** package to describe how to create and export a patch with Wind River Linux.

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
```

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 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 *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");
```

Step 10 Refresh the patch and save all current changes to the patch file created in 5 on page 238.

```
$ 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-7/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

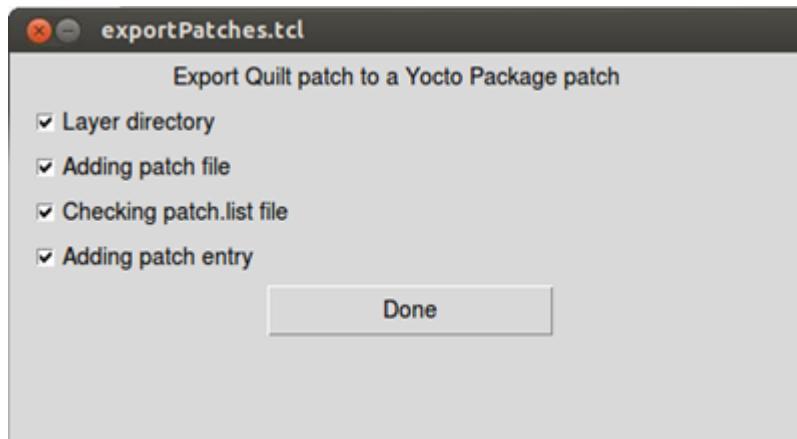
Your description to be included in the **patches.list** file

For example, the following command:

```
$ ~/WindRiver/wrlinux-7/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 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 238.

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 238.

Step 1 Choose a platform project option to add the layer (and patch) to:

Options	Description
Existing platform project	To add the layer to an existing project: <ol style="list-style-type: none">1. Navigate to the platform project directory: <pre>\$ cd <i>projectDir</i></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=~<i>path_to_layer</i> > newconfig \$ chmod +x newconfig \$./newconfig</pre>The project will configure itself to include the new layer.
New platform project	To add the layer to a new platform project, include the --with-layer= <i>path_to_layer</i> configure option when you create your project. For example: <pre>\$ configDir/configure --enable-board=qemux86-64 \ --enable-rootfs=glibc_std \ --enable-kernel=standard \ --enable-build=production \ --enable-parallel-pkgbuilds=4 \ --with-layer=<i>path_to_layer</i> \ --enable-jobs=4</pre>

Step 2 Optionally, build the platform project:

```
$ make
```

Modifying Package Lists

[About the Package Manager](#) 243

[About Modifying Package Lists](#) 247

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 functionally.

This procedure requires a previously built platform project image.



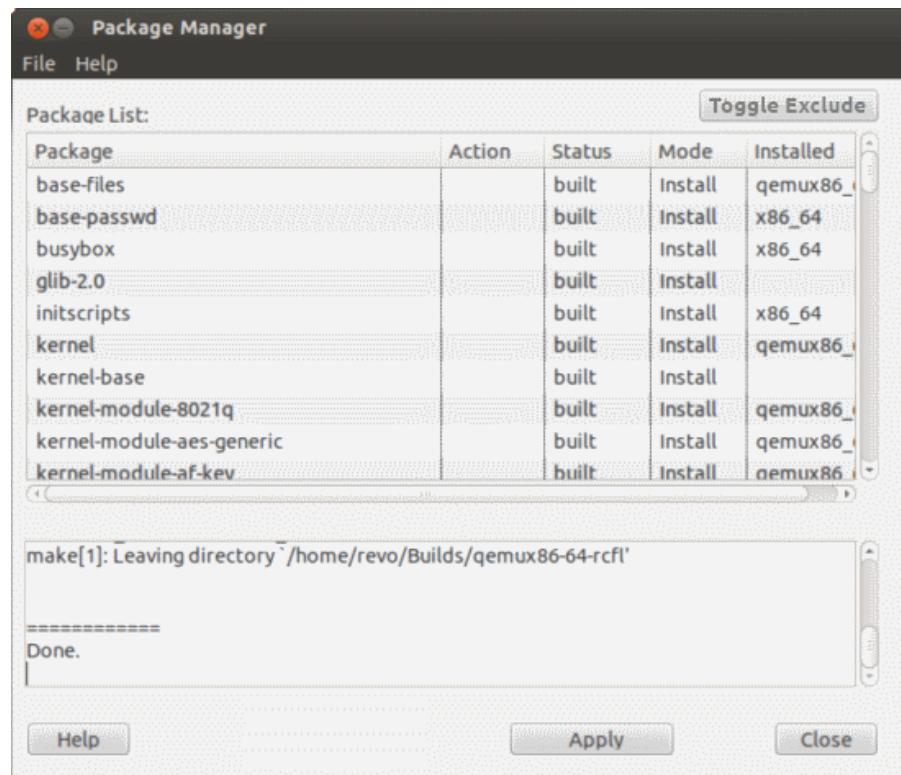
NOTE: You may launch the Package Manager on a configured, but not built, platform project, but doing so will provide limited package data, as this information is compiled by the build system as it builds the platform project image.

- 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.

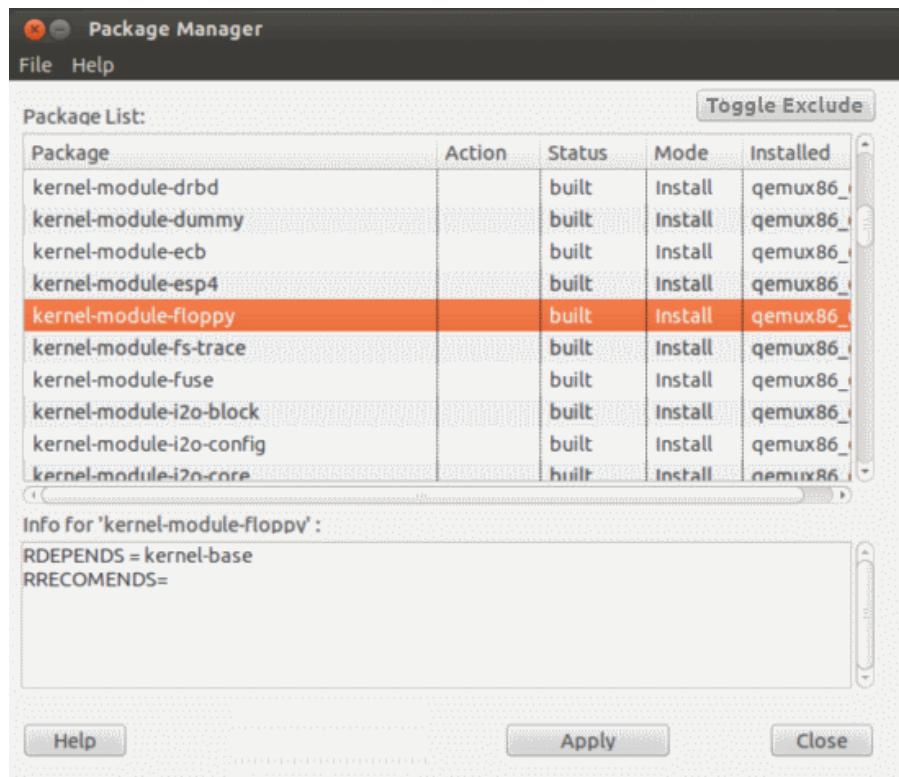


WARNING: The content of this chapter must be considered to be preliminary.

Due to the iterative nature of feature development there may be some variation between the documentation and latest delivered functionally.

Once the Package Manager is running, you can select packages and remove them.

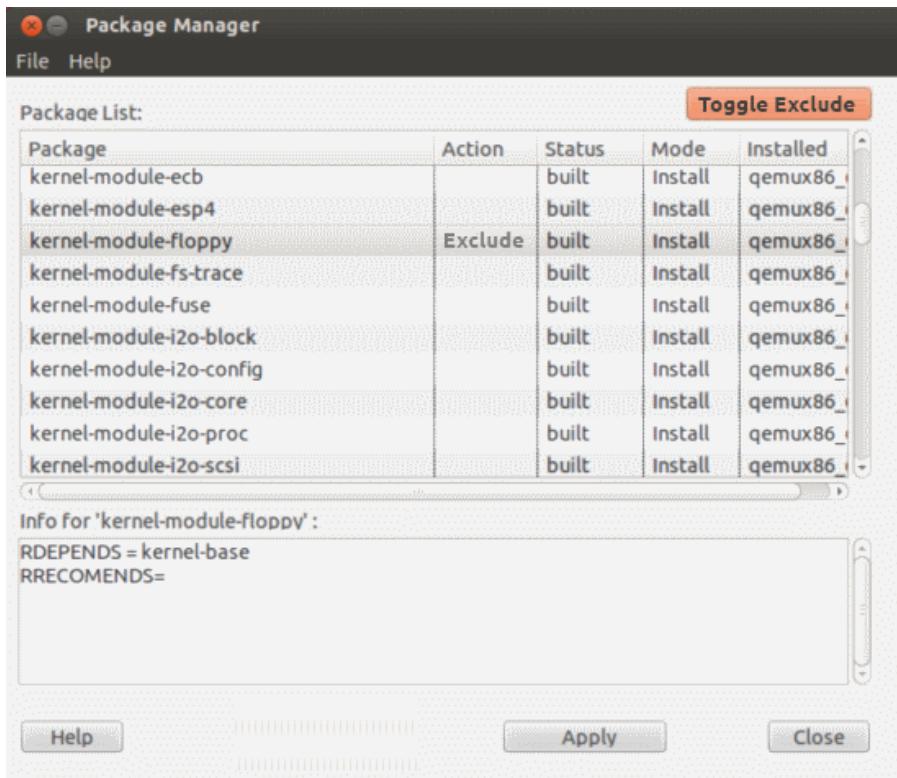
Step 1 Select a package in the **Package** column.



Information for the package appears in the bottom field. In this example, the **kernel-module-floppy** package depends on the **kernel-base** package.

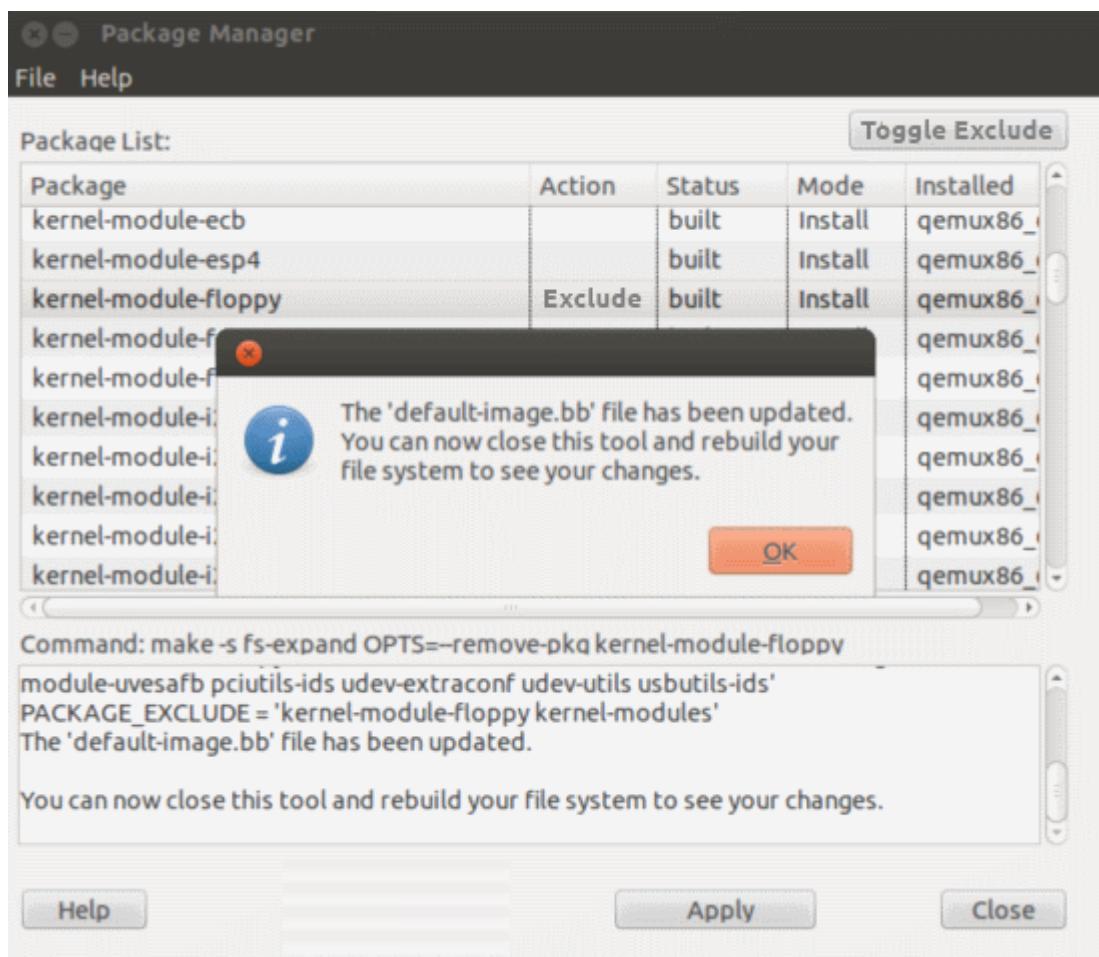
Step 2 Click **Toggle Exclude**.

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 244 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 461 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 there 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** script command at project creation time.
 - Use the following command at any time after package creation:
make packageName.addpkg
 - 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 243.

- Use the following **make** command to remove a package:

```
$ make packageName.rmpkg
```

All package dependencies are reviewed and revised by this command.

16

Maintaining Package Recipe Revisions

About Package Revision Management	251
Enabling Package Build History	253
Querying the PR Server Database	254
Comparing Build History with Previous Revisions	255

About Package Revision Management

Wind River Linux provides convenient tools for using the Yocto PR (package revision) service to manage package revisions.

PR Server Overview

Package Revision (PR) is a vital aspect of embedded software development. A package is comprised of a package name, package version, and package release. During an on-target package upgrade, the system package manager uses those items to determine what items will be upgraded and what items are older versions.

The PR server is responsible for automatically incrementing the package release when a change is made to a recipe. This change may be any change, implicit or explicit, that caused the package to be rebuilt from source. This ensures that the newer version of the package is installed during an upgrade. When you make changes to a package, such as updating the source, adding patches, or changing the project's configuration options, the build system ensures that the package revision is also updated. The initial package revision is set in the recipe file, but the final version is set by the PR server. The PR server starts at whatever value is set by the recipe. The default value is `r0`.



NOTE: The package management template adds the package-management features to the install package management tools, and preserves the package manager database on target. See [Package Management Configure Options](#).

IMPORTANT: When building a product that will be upgraded using a package-based approach be sure to have the PR server enabled and working in a persistent manner for all production builds. Otherwise the package release numbers may not be updated in such a way to facilitate an on-target upgrade.

Depending on your system requirements and development environment, you can specify how a platform project uses the PR server from the following options:

Local PR Server

This is the default method for managing package revisions. When you configure a platform project without any **--enable-prserver**= options, the PR service is automatically set to use a local server. This is the same as using the **--enable-prserver=yes** and **--enable-prserver=local** **configure** script options.

Using a local PR service sets the **projectDir/local.conf** file **PRSERV_HOST** setting as follows:

```
PRSERV_HOST = "localhost:0"
```

Shared PR Server

This option lets you specify a Wind River Linux project for use as a shared PR server (service). Wind River provides the **--enable-dedicated-prserver=yes** configure script option to create a platform project specifically for this purpose. This is the only platform project configuration that you should use to create a shared PR server. Standard platform projects should not be used, and are not supported.

When you configure a platform project with the **--enable-dedicated-prserver=yes** option, this makes the project image a dedicated PR server. As a result, typical configuration and build features for platform projects will not work as expected, and are not supported. Server configuration also requires that you specify the IP address and port for the dedicated server with the **--enable-prserver=IP address:port** **configure** script option. A minimal configure script command is as follows:

```
$ configDir/configure \
--enable-board=intel-atom \
--enable-rootfs=glibc-small \
--enable-prserver=128.224.147.66:31337 \
--enable-dedicated-prserver=yes
```

In this example, the listening PR server's IP address is set to **128.224.147.66** and the port is set to **31337**.

To use this shared server with other platform projects, you must obtain the server's IP address or known host name and port, and configure the platform project to point to the server using the **--enable-prserver=IP address or hostname:port** **configure** script option.

To verify that the dedicated server uses the same IP address and port specified at configure time, you can use the **make start-prserver** command on the target. This command provides specific information for configuring platform projects to point to the PR server.

IMPORTANT: For a shared PR server, the server database only writes data when the service is stopped manually. If the server experiences a crash, all data from the time the service was started will be lost. To save data, it is recommended to periodically stop and restart the service.

Disable PR Service (Manual)

If you wish to maintain package revisions manually, you can disable PR service functionality using the **--enable-prserver=no** configure script option. This is not recommended, but is an available option.

Additional PR Server Features

In addition to monitoring and updating package revisions, the PR server also provides debugging and build acceleration features to help simplify, and speed up, the development process.

The build history feature, enabled when you configure a platform project with the **--enable-build-hist=yes** **configure** script option, creates a **projectDir/bitbake_build/buildhistory** git repository that stores meta-data about the current project. This meta-data includes package dependencies, size, and any generated sub-packages to help aid in debugging package-related issues. For additional information, see [Enabling Package Build History](#) on page 253.

The remote read-only sstate mirror cache feature, enabled when you create a platform project using the **--with-ro-sstate-mirror=IP address:port** **configure** script option. This option appends the specified IP address to the **projectDir/local.conf** file **SSTATE_MIRRORS** setting.

To populate the cache, perform a platform project build, and the PR server syncs the cache to the shared server, either locally, or to a remote, read-only PR server mirror using read-only access over HTTP.

Enabling Package Build History

Enable the package build history feature to aid in package-related debugging.

This procedure requires a platform project that does not disable the PR server using the **--enable-prserver=no** configure script option.

Step 1 Configure a platform project to enable package build history.

Options	Description
Default location	Add the --enable-buildhist=yes option to your configure script, for example: <pre>\$ configDir/configure / --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_std \ --enable-buildhist=yes</pre>
Specify location	Add the --enable-buildhist=yes option to your configure script, and specify the location of your build system git repository using the --enable-buildhist-dir=path option. For example: <pre>\$ configDir/configure / --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_std \ --enable-buildhist=yes \ --enable-buildhist-dir=path_to_build_history_folder</pre>

To enable build history for an existing platform project, add the **--enable-reconfig=yes** option to the **configure** script command.

Step 2 Build the project.

```
$ make
```

Step 3 Verify the folder exists.

Options	Description
Default location	Verify that the <i>projectDir/bitbake_build/buildhistory</i> folder exists.
Specified location	Verify that the <i>path_to_build_history_folder</i> specified in the previous step exists.

Querying the PR Server Database

Use SQL queries to obtain package-related information from the PR server.

This procedure requires a previously built platform project image that does not disable the PR service. The steps provide a simplified example of querying the database for a specific package. Knowledge of SQL queries can help you obtain more specific package-related information.

Step 1 Launch the **sqlite3** binary to begin an **sqlite** session.

Run the following command from the platform project directory (*projectDir*):

```
$ ./host-cross/usr/bin/sqlite3 ./bitbake_build/cache/prserv.sqlite3
SQLite version 3.7.17 2013-05-20 00:56:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
```

Step 2 Obtain information on a specific package.

In this step, the query will return all information in the database on the **zlib** package.

```
sqlite> select * from PRMAIN_nohist where VERSION like "zlib%";
```

The system will return any package with **zlib** in the name that exists in the **PRMAIN_nohist** table in the database, for example:

```
zlib-1.2.8-r0 | x86_64 | e8eda225d7813b120f87f9f0e71cf5bd | 0
```

In this example, there have not been any changes made to the **zlib** package, so there is only one entry in the database.

You may also use a command-line statement to query the database, for example:

```
$ ./host-cross/usr/bin/sqlite3 ./bitbake_build/cache/prserv.sqlite3 \
> 'select * from PRMAIN_nohist where VERSION like "wr-init%";'
wr-init-1.0-r0 | x86_64 | 3332c1c162b4705119c6d490ecf0fb71 | 0
```

The resulting package name for this query would be **wr-init-1.0-r0.0.x86_64.rpm**, located in the *projectDir/bitbake_build/tmp/deploy/rpm/x86_64* directory.

For additional information on querying the Yocto PR Service database, see the Yocto Project website at [Yocto Project PR Service](#)

Comparing Build History with Previous Revisions

Use this procedure as a guideline to compare revisions and debug potential package-related issues.

This procedure requires a platform project configured and built with the `--enable-buildhist=yes` configure script option. For additional information, see [Enabling Package Build History](#) on page 253.

Step 1 Check the build history.

Before you make any changes, check the build history so you will have a reference point for your existing configuration.

- a) Navigate to the build history directory.

Run the following command from the `projectDir`.

```
$ cd bitbake_build/buildhistory
```

This command uses the default build history (`bitbake_build/buildhistory`) directory. If you specified a different directory, navigate to it.

- b) Check the `git log` to view the history.

```
$ git log
```

```
commit a45b55af789902464b7fa298234889734d4d3156
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 09:47:21 2014 -0400

    packages: Build 201307050918 of wrlinux 7.0.0.0 for machine qemux86-64 on
test-01

        cmd: bitbake wrlinux-image-glibc-small

commit d99825687fc9e6ae37d185b66f984bc9acb5621f
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 09:47:20 2014 -0400

    images: Build 201307050918 of wrlinux 7.0.0.0 for machine qemux86-64 on test-01

        cmd: bitbake wrlinux-image-glibc-small
```

For a new platform project with no changes, the output displays the initial commit dates for the packages and images created as part of the build.

Step 2 Add a new package to the platform project.

In this example you will add the `ed` package. For information on adding packages, see [Adding New Application Packages to an Existing Project](#) on page 209

Run the following command from the `projectDir`.

```
$ make ed.addpkg
```

Step 3 Rebuild the platform project.

```
$ make
```

The command should take considerably less time to process when compared to the original platform project build.

- Step 4** Check the build history again to view the new commits created by adding the new package.
- Navigate to the build history directory.

Run the following command from the *projectDir*.

```
$ cd bitbake_build/buildhistory
```

- Check the **git log** to view the history.

```
$ git log
```

```
commit fd7b1affbb479da41cca9480feb00b5c47bfcfae
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 10:12:31 2014 -0400

    packages: Build 201307051010 of wrlinux 7.0.0.0 for machine qemux86-64 on
test-01

        cmd: bitbake wrlinux-image-glibc-small

commit 9a6beb75cd7f9088602c8edd838ae042e388de26
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 10:12:31 2014 -0400

    images: Build 201307051010 of wrlinux 7.0.0.0 for machine qemux86-64 on test-01

        cmd: bitbake wrlinux-image-glibc-small

commit 633bcd638f499e21315bd476c44bf4a7cba2d89
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 10:00:10 2014 -0400

    No changes: Build 201307050956 of wrlinux 7.0.0.0 for machine qemux86-64 on
test-01

        cmd: bitbake wrlinux-image-glibc-small

commit a45b55af789902464b7fa298234889734d4d3156
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 09:47:21 2014 -0400

    packages: Build 201307050918 of wrlinux 7.0.0.0 for machine qemux86-64 on
test-01

        cmd: bitbake wrlinux-image-glibc-small

commit d99825687fc9e6ae37d185b66f984bc9acb5621f
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 09:47:20 2014 -0400

    images: Build 201307050918 of wrlinux 7.0.0.0 for machine qemux86-64 on test-01

        cmd: bitbake wrlinux-image-glibc-small
```

Notice that for the new package additions there is a new commit.

- Check the difference between the initial build and the new commits to see the specific changes.

```
$ git diff --stat=80 HEAD~3
```

```
images/qemux86_64/eglibc/wrlinux-image-glibc-small/build-id      | 2 +-  
.../depends-nokernel-nolibc-noupdate-nomodules.dot           | 1 +  
.../depends-nokernel-nolibc-noupdate.dot                      | 1 +  
.../wrlinux-image-glibc-small/depends-nokernel-nolibc.dot     | 1 +  
.../eglibc/wrlinux-image-glibc-small/depends-nokernel.dot     | 2 ++
images/intel_atom/eglibc/wrlinux-image-glibc-small/depends.dot | 2 ++
.../eglibc/wrlinux-image-glibc-small/files-in-image.txt       | 2 ++
```

```
.../qemux86_64/eglibc/wrlinux-image-glibc-small/image-info.txt | 4 +---  
.../wrlinux-image-glibc-small/installed-package-names.txt | 1 +  
.../wrlinux-image-glibc-small/installed-package-sizes.txt | 1 +  
.../eglibc/wrlinux-image-glibc-small/installed-packages.txt | 1 +  
metadata-revs | 2 +-  
packages/x86_64-wrs-linux/ed/ed-dbg/latest | 10 ++++++++  
packages/x86_64-wrs-linux/ed/ed-dev/latest | 9 ++++++++  
packages/x86_64-wrs-linux/ed/ed-doc/latest | 9 ++++++++  
packages/x86_64-wrs-linux/ed/ed-locale/latest | 9 ++++++++  
packages/x86_64-wrs-linux/ed/ed-staticdev/latest | 9 ++++++++  
packages/x86_64-wrs-linux/ed/ed/latest | 9 ++++++++  
packages/x86_64-wrs-linux/ed/latest | 4 ++++  
19 files changed, 75 insertions(+), 4 deletions(-)
```

From the output you can see how the build history tracks changes to the platform.

Step 5 Make a change to the package dependencies.

Use this step to see how the build history feature records changes to a package's dependencies.

- Navigate to the *projectDir*.
- Modify the package dependencies.

The following command will update the *projectDir/layers/oe-core/meta/recipes-extended/ed/ed_1.9.bb* recipe file to add **inherit autotools pkgconfig**.

```
$ echo "inherit autotools pkgconfig" >> layers/oe-core/meta/recipes-extended/ed/ed_1.9.bb
```

- Rebuild the platform project.

```
$ make
```

NOTE: If you receive an error to update the **LICENSE** field, you will need to add **GPLv3** to the **LICENSE** whitelist.

Step 6 Review the package changes in the build history.

- Navigate to the build history directory.

Run the following command from the *projectDir*.

```
$ cd bitbake_build/buildhistory
```

- Display the latest **git** head revision to see the package changes.

```
$ git show HEAD
```

```
commit 7ecdcd6b60b5430f06729be5677b536540912f65
Author: buildhistory <buildhistory@wrlinux>
Date: Fri Jul 5 10:38:07 2014 -0400

    packages: Build 201307051035 of wrlinux 7.0.0.0 for machine qemux86-64 on
test-01

    cmd: bitbake wrlinux-image-glibc-small

[... edited for readability ...]
diff --git a/packages/x86_64-wrs-linux/ed/latest b/packages/core2-wrs-linux/ed/
latest
index 0a5e232..b42e635 100644
--- a/packages/x86_64-wrs-linux/ed/latest
+++ b/packages/x86_64-wrs-linux/ed/latest
@@ -1,4 +1,4 @@
PV = 1.9
PR = r0
DEPENDS = virtual/i586-wrs-linux-compilerlibs virtual/i586-wrs-linux-gcc virtual/
libc
```

```
+DEPENDS = autoconf-native automake-native gnu-config-native libtool-cross libtool-
native pkgconfig-native virtual/i586-wrs-linux-compilerlibs virtual/i586-wrs-linux-
gcc virtual/libc
PACKAGES = ed-dbg ed-staticdev ed-dev ed-doc ed-locale ed
```

PART IV

Kernel Development

Understanding the Wind River Linux Kernel.....	261
Patching and Configuring the Kernel.....	281
Creating Optimized Custom Kernel Builds.....	303
Creating Alternate Kernels from kernel.org Source.....	311
Exporting Custom Kernel Headers.....	315
Using the preempt-rt Kernel Type.....	319

17

Understanding the Wind River Linux Kernel

[About the Wind River Linux Kernel](#) 261

[About Creating and Maintaining Custom Kernel Branches](#) 267

[About Saving Kernel Modifications](#) 272

[Working with -dirty Kernel Strings](#) 278

About the Wind River Linux Kernel

Wind River Linux presents the kernel through a fully patched, history clean, git repository.

This git tree represents the selected features, board support, and configurations that were extensively tested by Wind River. Presenting the Wind River kernel in this manner allows the end user to leverage community best practices to seamlessly manage the development, build and debug cycles.

The complexity of embedded kernel design has increased dramatically. Whether it is managing multiple implementations of a particular feature, to tuning and optimizing board-specific features, flexibility and maintainability are key concerns. The Wind River Linux kernel is presented with the developer's needs in mind and has evolved to assist in these key concerns.

In particular, prior kernel development methods, such as applying hundreds of patches to an extracted *.tar file, have been replaced with proven techniques that allow easy inspection, bisection and analysis of changes. It also creates a platform for performing integration and collaboration with the thousands of upstream development projects.

Wind River leverages the many features of git to meet the development goals described in this section.. For example, git is in the kernel.org SCM, it continues to grow in popularity, and as a result can support many different work flows, front end development tools, and management techniques. Although it should be noted, that the end user can use as much, or as little, of what git has to offer as is appropriate to their project.

About the Baseline Kernel

Wind River's kernel team monitors many external mailing lists, constantly watching how features are progressing and what significant fixes are being proposed, to deliver the best kernel possible.

Internally we sample the latest external kernels as they are being developed so that we have a vision into what the impact of the upstream changes will be to our product and to our feature and support matrix.

At the beginning of a major development cycle, we consider factors to determine the kernel version that will be released. These factors include release timing based on our intended features, and the release timing and features provided by the kernel.org final kernel version. Typically this will be a kernel that is in the final stages of development by the community, for example is still in the release candidate phase),and not yet a final release.

But by being in the final stages of external development, we know that their final release will clearly land within the early stages of our development window. This balance allows us to deliver to our customers the most up to date kernel as possible, while still ensuring that we have a stable official release as our baseline kernel version.

Once a kernel has been released, Wind River continues to monitor community development to look for significant features of interest, and will consider back porting large features if they have a significant advantage. Customer demand and requirements may also trigger a back port or creation of new functionality in the baseline kernel. Customer requirements and inquiries can also be a driving factor for a feature backport/integration. Generally speaking, every new kernel both adds features and introduces new bugs, this is the basic property of upstream kernel development and has been managed by Wind River's kernel strategy.

Wind River has the policy to not back port minor features to our commercial kernel and only considers significant technological jumps for back porting. This is simply because back porting any small to medium size change from an evolving kernel can easily create mismatches, incompatibilities and very subtle errors. Wind River constantly monitors and participates in upstream development and is aware of significant developments and will arrange to back port or include them in our kernel uprev strategy.

So the maintained kernel is not a particular version, but is a version ++ with a mix of important new mainline development, integration of non-mainline development, BSPs and custom feature integration. If specific functionality is identified in a newer kernel, it can be considered for back port once Wind River performs a gap analysis. It is important to note that the most sustainable and commercial-quality method to include feature development upstream is through a kernel uprev, and not by back porting of hundreds of individual fixes and minor features from various kernel versions.

In an uprev cycle, Wind River takes our ongoing analysis of kernel development, BSP support and release timing to select the best possible kernel version. During the time between full kernel uprevs, Wind River maintains a cutting edge kernel and forward ports significant and critical functionality. The many micro uprevs, or minor updates, gives insight into these new features and allows a focused amount of testing to be done on new versions of the kernel, which prevents surprises when selecting the next major uprev. This kernel is not commercial quality and is used in very special cases for BSP development.

About Branching Features and Kernel Types

Once core kernel features are determined, kernel types based on specific requirements provide a basis for branching kernel development.

As described in [About the Wind River Linux Kernel](#) on page 261, Wind River strives to provide a kernel git tree, that just like the upstream kernel.org tree, has a clear and continuous history. We want to provide our customers with the ability to see the features we've added, and the commits that make up those features, and to be able to seamlessly transition into viewing the history of what made up the baseline kernel underneath our added features as well.

Wind River also strives to deliver a key set of supported kernel types, where each type is tailored to a specific use case, such as for networking, consumer devices, carrier grade, secure, and so on. Features that are applied to that kernel type are those which provide additional value to developers and customers with products in that segment. In contrast, features that do not provide additional value, or are even a possible detriment to that kernel type, are excluded.

Once all the common features and fixes are applied to the standard branch, we are left with the features that are specific to a kernel type, or a subset of kernel types. This is where we adopt the branching strategy used by subsystem maintainers for features that currently are not included with the standard kernel.

We then create a branch off the endpoint of standard (standard kernel with balanced features that apply to all branches) for each of the kernel types. Then for each kernel type branch, we repeat the process used above for applying features to the standard branch, only we are now applying the features that are specific to that kernel type instead. Note that if a feature is to be applied to multiple kernel types, we don't internally store the feature in two different branch locations, but instead only store the unique differences required to apply the feature onto the kernel type in question.

Finally, with all the supported kernel types created and with all their respective features committed to the appropriate kernel type branches, we move to the BSP specific code additions. Some BSPs only make sense to be available on certain kernel types, so for the nominated kernel types, we then create branches off the end of that kernel type for all of the BSPs that are supported on that kernel type.

From the perspective of the tools that create the BSP branch, the BSP is really no different than a feature, and so the same methodology applies to BSPs as it does to features. If a BSP is supported on multiple kernel types, we don't internally store the BSP in two locations, but instead only store the unique differences required to apply the BSP-specific code to the kernel type in question. While the resulting tree may have a significant number of branches, it is important to realize that there is a linear path from the baseline kernel.org kernel, beginning with a select group of features and ending with their BSP-specific commits.

From this perspective, the Wind River Linux kernel represents master branch, and as a result, the existence of any other branches does not impact kernel development. Note that there is value in the existence of these branches in the tree, should a person decide to explore them. For example, a comparison between two BSPs at either the commit level or at the line-by-line code diff level is now a trivial operation, but could definitely help determine development choices and actions.

Kernel Branching and Tagging Strategy

Learn about the kernel branch model to understand how to apply patches to your local kernel.

The work flow of the Wind River kernel follows the recognized community best practices. In particular, the kernel as shipped with the product should be considered an upstream source and viewed as a series of historical, documented modifications (or commits) to the kernel. These kernel modifications represent the development and stabilization done by the Wind River kernel development teams.

These commits only change at significant release points in the product life cycle, which allows the feature or BSP developer to work on a branch created from the last relevant commit in the shipped Wind River kernel. This is transparent to the user, since the kernel tree is left in this state after cloning and building the kernel.

Wind River creates kernel branches at significant functionality sharing points, or where functionality must be kept separate. Reasons for this functionality separation and isolation range from fundamental incompatibilities to keeping board-specific modifications separate.

NOTE: These divisions of the kernel are transparent and should not be relevant to the developer on a daily basis.

Any work on the kernel should be viewed as a series of commits from base of the Wind River kernel to the HEAD of the branch. Branches are transparent and are simply markers in that series of commits. And those markers are shared by many BSPs and features.

The general branching and tagging structure of a Wind River kernel repository is as follows:

```
kernel.org
|
+--- meta
|
+--- standard
|   |
|   |   feature (a)
|   |   feature (b)
|   .
|   |
|   +---- standard/bsp1
|   +---- standard/bsp2
|
|   .
|   +---- standard/cgl (and other kernel types)
|       |
|       |   cgl features: c,d,e, etc
|       |
|       +---- standard/cgl/bsp1
|       +---- standard/cgl/bsp2
|       +---- standard/cgl/bsp3
```

meta

Kernel meta-data. The BSP, feature, and configuration data used to construct the tree, and configure BSPs, is contained within this branch.

standard

Represents branches from kernel.org at a defined point, for example, at version 3.14.22, and where it diverges from kernel.org at this point. Common functionality for all boards is defined in this branch.

feature n1, n2

Features in the standard branch, such as **lttng2** and **yaffs2**.

standard/bsp1 and bsp2

Represents BSP branches at the top of the **standard** branch. Any board-specific changes are contained in these branches.

standard/cgl and other features

Represents feature branches at the top of the standard branch. Any kernel type-specific changes that are part of a specific feature are contained in these branches, inherit the **standard** branch features, and add additional features..

standard/cgl/bsp1, bsp2, and so on.

The BSP branch at the top of the feature kernel type, in this example, the **cgl** kernel type. Although this is a separate branch from the **standard/bsp** branch, the same patches are used to construct the branch to provide identical board-specific functionality.

Kernel Development Tools

Kernel development requires a minimum subset of tools as described in this section.

Since most standard workflows involve moving forward with an existing tree by continuing to add and alter the underlying baseline, the tools that manage Wind River's kernel construction are largely hidden from the developer to present a simplified view of the kernel for ease of use.

The fundamental properties of the tools that manage and construct the kernel include:

- the ability to group patches into named, reusable features
- the ability to allow top down control of included features
- the binding of kernel configuration to kernel patches and features
- the presentation of a seamless git repository that blends Wind River value with the kernel.org history and development

The following tools form the foundation of the Wind River kernel toolkit:

git

The popular distributed revision control system created by Linus Torvalds.

***cfg**

Kernel configuration management and classification through configuration.

kgit*

Wind River kernel tree creation and management tools

scc

Series and configuration compiler

Kernel Tree Construction and Workflow

The Wind River kernel repository, as shipped with the product, is created by compiling and executing the set of feature descriptions for every BSP and product feature.

Overview

Those feature descriptions list all necessary patches, configuration, branching, tagging and feature divisions found in the kernel. The files used to describe all the valid features and BSPs in the Wind River kernel can be found in any clone of the kernel git tree.

The directory `meta/cfg/kernel-cache/` is a snapshot of all the kernel configuration and feature descriptions (`.scc` files) that were used to build the kernel repository. It should however be noted, that browsing the snapshot of feature descriptions and patches is not an effective way to determine what is in a particular kernel branch. Using git directly to get insight into the changes in a branch is more efficient and a more flexible way to inspect changes to the kernel. For examples of using git to inspect kernel commits, see [Viewing Changes Applied to a Kernel Tree](#) on page 272.

As a reminder, it is envisioned that a reconstruction of the complete kernel tree is an action only taken by Wind River staff during an active development cycle. When an end user creates a project, it takes advantage of this complete tree in order to efficiently place a git tree within their project.

Tree Build Workflow

When you build a platform project or the kernel specifically, the general flow of the project specific kernel tree construction is as follows:

1. A top level kernel feature is passed to the kernel build subsystem; typically this is a BSP for a particular kernel type.
2. The file that describes the top level feature is located by searching system directories:
 - the kernel-cache under `linux/*/cfg/kernel-*cache`
 - `.scc` files specified in layer space recipesA `.scc` file that matches the kernel type and the name of the BSP being built is found and becomes the entry point for the build.
3. Extra or additional features are located and appended to the BSP description. These extra features can come from recipes or templates.
4. The complete set of features is compiled to produce a meta-series.

The meta-series is description of all the branches, tags, patches and configuration that need to be applied to the base git repository to completely create the source and configuration for the BSP.

5. The base repository is cloned, and the actions listed in the meta-series are applied to the tree.
6. The git repository is left with the desired branch checked out and any required branching, patching, and tagging has been performed.

The tree is now ready for configuration and compilation.



NOTE: The generated meta-series adds to the kernel specified by the recipe, either Wind River, or a customer kernel. Any addons and configuration data are applied to the end of an existing branch. The full repository generation that is found in `wr-kernel/git/kernel-version.git` is the result of running the above steps for every supported board and kernel combination in the product.

This technique is flexible and allows the seamless blending of an immutable history with additional deployment specific patches. Any additions to the kernel become an integrated part of the branches.

About Creating and Maintaining Custom Kernel Branches

To meet your development needs, you can maintain a specific kernel in a separate repository that inherits the Wind River Linux kernel.

This repository acts as the integration point of your added features and extensions, blended with the supported Wind River baseline kernel. To maintain this tree, you can reuse branches, create new branches, or merge multiple branches into a hybrid solution.

There are three elements to maintaining a kernel in this manner: creating the repository, updating and maintaining it, and integrating it into the build infrastructure.

Creating a Custom Repository

Creating a new, custom repository is a matter of cloning the Wind River kernel repository that you want to work from. For information on creating a repository, see [Creating a Custom Kernel Repository](#) on page 269.

Once the repository is created, you may create custom branches as placeholders for your own kernel changes, or merge existing branches. Exactly what you choose to create or merge depends on your project requirements and the repository you clone from. Two such examples are creating a branch that inherits an existing BSP and creating a new, merged multi-BSP branch, as described in [Creating a Custom Kernel Repository](#) on page 269.

Both of these examples deal primarily with BSP source changes, and not the configuration. The kernel repository's `meta` branch contains the configuration data for each BSP. As a result, you must inspect and merge this data to create a hybrid configuration. Since board and project requirements vary, and Wind River already places common policy configuration data in shared configuration fragments, it is up to you to determine the desired configuration mix.

If you follow the instructions in [Creating a Custom Kernel Repository](#) on page 269, the `meta` branch includes a BSP `.scc` file that describes the new BSP. This BSP fragment can automatically include the configuration of `standard/my-bsp` for single BSP repositories, or multiple other BSPs, as described in [Kernel Branching and Tagging Strategy](#) on page 264.

It can also include new BSP specific configurations that are explicitly created from scratch, or by inspecting existing BSP configurations. If automatic inclusion is used, with overrides in the new BSP enforcing local policy, new configuration changes from Wind River will be automatically inherited when the `meta` branch is updated.

Maintaining a Custom Repository

Once a custom repository has been created, it is maintained separately from the Wind River kernel trees that are updated using RCPLs and RCFLs. This lets you control and inspect changes that are merged into the repository.

Updating the repository to pick up Wind River updates is done using standard git commands and workflows. The general workflow includes:

1. Updating your custom repository with the current Wind River changes.
2. Merging the changes into your relevant custom repository branch, or branches.
3. Resolving any conflicts that you discover in the merge process.

For information on updating a repository in this manner, see [Updating a Hybrid Kernel Branch](#) on page 270. Once all updates have been completed, the resulting branches are pushed into the main custom repository.

Integrating Repository Changes

Assuming that the custom repository has a valid BSP description and configuration, to build the new content, a `linux-windriver.bbappend` file is used to override the `SRC_URI` and `KBRANCH` options, and optionally indicate board compatibility. For additional information, see [Building a Custom Kernel Branch](#) on page 271.

Part of building the new kernel content is comparing and verifying the changes you made to the tree in your custom repository. In previous Wind River releases, there were a collection of directories that contained patches to the kernel. You could use grep to find and inspect those patches to get a general feeling for changes.

This sort of patch inspection is not an efficient way to determine what has been done to the kernel, since there are many optional patches that are selected based on the kernel type and feature description, not to mention patches that are actually in directories that are not being searched. A more effective way to determine what has changed in the kernel is to use git to inspect and search the kernel tree. This is a full view of not only the source code modifications, but the reasoning behind the changes. For additional information, see [Viewing Changes Applied to a Kernel Tree](#) on page 272.

It is important to note that the Wind River git repository does not break existing git functionality. Any commands that you use with git are supported in the Wind River kernel tree, and that kernel.org history is blended with all Wind River changes.

Once changes are made and verified as part of building your custom kernel or BSP repository, you will want to save your modifications, including incremental, and bulk changes to the repository. For additional information, see [About Saving Kernel Modifications](#) on page 272.

Using Other Source Control Management Systems

Wind River recommends using git for all interaction with the Wind River baseline kernel, but understands that not all organizations use git as their primary source control management system. As a result, other source control management systems should support such features as importing and exporting git repositories and commits, and the ability to export commits as patches.

For source control management systems that do, the preferred approach is to import the entire Wind River kernel git repository, branches and all, into the new environment. This approach provides the most flexibility while maintaining the meta data associated with each commit. Once all your changes are made, use the source control management's recommended method for merging the changes back to the Wind River git tree, if applicable.

If you must work in another source control management environment where directly importing the git repository is not supported, it is still possible to export a branch or series of branches using git, and then check them into a new repository. For additional information, refer to your source control management system's documentation.

Creating a Custom Kernel Repository

Learn how to create a local kernel repository for specific development requirements.

In the following procedure, there are two methods for creating a custom repository: creating a repository with a branch that inherits an existing BSP, and a new, merged BSP branch with multiple BSPs.

Step 1 Clone the Wind River kernel repository.

Run the following command on your system where you want the cloned repository to reside as a sibling directory.

```
$ git clone path_to_repo/wr-kernel/git/kernel-version.git custom-repo.git
```

If you clone the repository from your Wind River Linux installation, *path_to_repo* would be *installDir/layers*.

Step 2 Navigate to the repository directory.

```
$ cd custom-repo.git
```

Step 3 Create the repository, depending on your requirements.

Options	Description
Branch that inherits an existing BSP	The following command creates BSP branch based on the intel_x86 BSP: <pre>\$ git checkout -b standard/intel-x86-64/my-board origin/standard/intel-x86-64</pre> <hr/> NOTE: If you merge this branch into a repository that contains a branch named standard/intel-x86-64 , standard/intel-x86-64 would need to be renamed to standard/intel-x86-64/base to avoid file system conflicts.

Options	Description
New, merged multi-BSP branch	<ol style="list-style-type: none">1. Create the new platform branch, from a common feature point in the repository. \$ git checkout -b standard/my-platform origin/standard/base2. Merge BSP support into the branch. \$ git merge origin/standard/intel-x86-643. Merge, or choose specific BSP changes, into the branch. \$ git merge origin/standard/common-pc/base4. Optionally merge, or choose specific BSP changes, for a third BSP into the branch. \$ git merge origin/standard/intel-atom <p>If you receive information concerning multiple, conflicting files as part of the merge, note that the time required to resolve them only happens once. Once resolved, subsequent merges from this BSP branch will no longer conflict. Wind River tries to minimize the number of conflicts, but optimized support libraries, or preview or pending mainline code is a common source of conflicts.</p>

Updating a Hybrid Kernel Branch

Updating a branch with new Wind River Linux changes keeps your custom branch synchronized with the latest development features.

In this procedure, you will update a custom repository with the latest upstream changes, merge and inspect the changes, and resolve any conflicts that arise between the upstream changes and your own development changes.

Step 1 Navigate to the local kernel repository directory.

```
$ cd custom-repo.git
```

Step 2 Fetch the latest upstream changes and add them to your repository.

Run each of the following commands to update all x86-based kernels to the latest RCPL versions.

```
$ git fetch path_to_RCPL_or_RCFL kernel standard/base:standard/base-rcpl$VERSION
$ git fetch path_to_RCPL_or_RCFL kernel standard/intel-x86-64/:standard/intel-x86-64-
rcpl$VERSION
$ git fetch path_to_RCPL_or_RCFL kernel standard/intel-atom/:standard/intel-atom-rcpl
$VERSION
$ git fetch path_to_RCPL_or_RCFL kernel meta:meta-rcpl$VERSION
```

Step 3 Checkout the updates to inspect the changes.

This includes reviewing the changes and resolving conflicts. You will first checkout the branch, and then merge it with your existing repository for each item from the previous step.

```
$ git checkout -b standard/my-platform-merge standard/my-platform
$ git merge standard/base-rcpl$VERSION
```

Resolve any conflicts and merge to the latest RCPL version:

```
$ git merge standard/intel-x86-64-rcpl$VERSION
$ git merge standard/intel-atom-rcpl$VERSION
```

Step 4 Inspect branch content and merge it to the production or build branches.

```
$ git checkout standard/my-platform
$ git merge standard/my-platform-merge
$ branch -D standard/my-platform-merge standard/intel-x86-64-rcpl$VERSION standard/
intel-atom-rcpl$VERSION
```

Step 5 Merge all meta content to the main branch.

```
$ checkout -b meta-merge meta
$ merge meta-rcpl$VERSION
$ checkout meta
$ merge meta-merge
$ branch -D meta-rcpl$VERSION meta-merge
```

Once all updates are complete, the resulting branches are pushed into your main, custom repository.

Building a Custom Kernel Branch

Custom kernel branches make use of a **.bbappend** file to add the feature to the **linux-windriver** kernel.

In *Kernel Configuration and Patching with Fragments* on page 283, you learn how to create a **.bbappend** file to add changes to the kernel. In this example, you will create a **.bbappend** file to override the default kernel settings and use the kernel in your local repository.

Step 1 Create the BitBake append file of the kernel.

Run the following command from the **projectDir**.

```
$ vi layers/local/recipes-kernel/linux/linux-windriver_3.14.bbappend
```

Step 2 Override the default **SRC_URI** and **KBRANCH** locations.

Update the file created in the previous step with the following information:

```
SRC_URI = "git:///opt/wrlinux-x/layers/myBSP/git/
kernel-3.14.x.git;protocol=file;nocheckout=1;branch=${KBRANCH},meta;name=machine,meta"
KBRANCH = "standard/my-platform"
COMPATIBLE_MACHINE_my-platform = "my-platform"
```

In this example, the build system will look to the local custom git repository for kernel source for the build. Note that the **COMPATIBLE_MACHINE** option indicates a specific board or BSP compatibility. This setting is optional.

Another possibility is to point to the location of the local, checked-out repository:

```
KSRC_linux_windriver_3_14 = "/opt/wrlinux-x/layers/myBSP/git/kernel-3.14.x.git"
KBRANCH = "standard/my-platform"
```

Step 3 Rebuild the kernel.

```
$ make linux-windriver.rebuild
```

Viewing Changes Applied to a Kernel Tree

There are many ways to view changes depending on what information you require.

The following examples provide the method to view changes made to the git repository's tree.

- Run each of the example commands from the repository to obtain information on kernel changes.

Options	Description
View a full description of changes	<pre>\$ git whatchanged kernelType..kernelType/BSP</pre>
View a summary of the changes	<pre>\$ git log --pretty=oneline --abbrev-commit kernelType..kernelType/BSP</pre>
View source control changes	<p>The following commands show source control changes as a combined diff.</p> <pre>\$ git diff kernelType..kernelType/BSP \$ git show kernelType..kernelType/BSP</pre>
Dump patched by commit	<pre>\$ git format-patch -o kernelType..kernelType/BSP</pre>
Review a file's change history	<pre>\$ git whatchanged path_to_file</pre>
Identify the commits that touch each line in a file	<pre>\$ git blame path_to_file</pre>

About Saving Kernel Modifications

Once you make changes to the kernel tree to support your development, there are various methods to save them for future use.

Using git for source control management greatly simplifies the process for saving BSP or kernel source changes. The method(s) include:

Incremental, planned sharing

One method for saving changes in a distributed development environment is described in [About Saving Incremental Kernel Changes](#) on page 274. This method is specifically suited for development teams that must be able to keep track of kernel revisions and add to or remove them as development continues.

Bulk export

The idea behind using a bulk export approach is that once you make a lot of local changes that you want to save, you can do that all at once (in bulk) to save the changes to the local source tree. This is not a suitable approach for upstream submission, since the changes are not separated and there are no specific commit messages for each change.

Use the bulk storage approach when changes have not been committed incrementally during development, and you want to gather them at once and add them to your local repository. For information on performing a bulk commit, see [Using Bulk Export to Save Kernel Modifications](#) on page 275.

Internal sharing

This method has two main options: using git commits, or creating patches with git, to save local modifications.

Patch export

Once a patch is created, you can use it for upstream submission as described in [Exporting Committed Changes Upstream](#) on page 278. It can also be placed in a template to automatically patch the kernel.

A patch is a series of commits that you want to use together to create a specific patch. Once the patch is created, the result is a directory with sequentially numbered patches that when applied to a repository, will reproduce the original commit and preserve all related, original information, such as the author, date, commit log, and so on.

Creating a tag or branch prior to creating a patch provides a basis for specifying the tag to create the patch. Alternatively, you can also specify a specific commit or range of commits to create the patch from. For information and options on creating a patch, see [Exporting Committed Changes Internally as Patches](#) on page 277.

Internal git export

Export changes from a working directory by pushing the changes to a master repository. Once pushed to a master repository, you can pull them into a new kernel build at a later time as necessary. For additional information, see [Exporting Committed Changes Internally with Git](#) on page 276.

A pull request entails using the `git request-pull` command to compose an email to the maintainer requesting that a branch be pulled into the master repository. For an example, see <http://github.com/guides/pull-requests>.

External, upstream submission

The most common approach to send changes for upstream submission is to create a patch or set of patches as an email series. This provides a simplified method for reviewing and integrating the changes upstream. Another method is to use a pull request, but doing so depends on the maintainer's preference and whether the patch series is large enough.

When sending a patch as an email series for review, you should be aware of the upstream community in question, and follow their best practices. For example, the standards for submitting kernel patches are available at the following locations:

- <http://userweb.kernel.org/~akpm/stuff/tpp.txt>
- <http://linux.yyz.us/patch-format.html>
- Any Linux kernel source tree, in **Documentation/SubmittingPatches**

The messages used to commit changes are a large part of these standards, so ensure that the headers for each commit have the required information. If the initial commits were not properly documented or do not meet those standards, you can rebase the commits using `git rebase -i`.

This provides an opportunity to manipulate the commits and get them into the required format. Other techniques such as branching and cherry picking commits are also viable options.

Once complete, patches are sent via email to the maintainer(s) or lists that review and integrate changes. The **git send-email** command is commonly used to ensure that patches are properly formatted for easy application and avoid mailer induced patch damage. For an example procedure for sending patches upstream, see [Exporting Committed Changes Upstream](#) on page 278.

About Saving Incremental Kernel Changes

Use git options to facilitate incremental kernel changes in a distributed development environment.

Distributed development with git is possible by having a universally agreed upon unique commit identifier (set by the creator of the commit) mapping to a specific changeset with a specific parent. This ID is created for you when you create a commit, and will be re-created when you amend, alter, or re-apply a commit.

As an individual in isolation, this approach doesn't really add much value, but if you intend to share your tree with normal git push and pull operations for distributed development, you should consider the ramifications of changing a commit that you've already shared with others. A universal identifier greatly simplifies the process of keeping track of a large number of kernel changes.

Assuming that the changes have not been pushed upstream, or pulled into another repository, you can update the commit content and commit messages associated with development using the **git --amend** and **rebase** options. In addition, to remove and reset commits that have not been pushed upstream, you can take advantage of the **git reset** options.

A development team can create branches, cherry-pick specific changes, or perform any number of git operations until the commits are in good order for pushing upstream or for pull requests. After a push or pull, commits are normally considered 'permanent' and should not be modified, only incrementally changed in new commits. This is standard git workflow and Wind River recommends the kernel.org best practices.

NOTE: Wind River recommends to tag or branch before adding changes to a Wind River BSP, or before creating a new BSP. The new branch or tag provides a reference point to help locate and export local changes.

Committing Incremental Changes

Use this approach to commit new incremental changes that have not been pushed upstream.

As described in [About Saving Incremental Kernel Changes](#) on page 274, this approach commits the content and commit messages associated with development.

Step 1 Add a new file to the repository.

```
$ git add path/fileName
```

Step 2 Commit the file to the repository.

```
$ git commit --amend
```

The **--amend** option ensures that the message of the original commit is preserved, and used as the starting point, rather than an empty message.

Step 3 Merge (or stage) the commit for addition to the upstream branch.

```
$ git rebase
```

The **rebase** option merges the commit to the upstream branch associated with the git repository the branch was cloned from. If there is no upstream equivalent, the commit is staged in a temporary area.

Reverting Incremental Kernel Changes

Use these options to revert new incremental changes that have not been pushed upstream.

As described in [About Saving Incremental Kernel Changes](#) on page 274, this approach provides various options for reverting commits associated with development.

- Choose an option that best suits your needs for reverting commit content.

The commands for each option are run from the git repository.

Options	Description
Remove the commit, update working tree, and remove all traces of the change	<code>\$ git reset --hard HEAD^</code>
Remove the commit, but leave files changed and staged for re-commit	<code>\$ git reset --soft HEAD^</code>
Remove the commit, leave files changed and not staged for re-commit	<code>\$ git reset --mixed HEAD^</code>

Using Bulk Export to Save Kernel Modifications

Use bulk export to add all the local changes in a single commit to your local repository.

This approach for saving kernel modifications and BSP changes is best suited for capturing all patches at once and adding them. For additional information, see [About Saving Kernel Modifications](#) on page 272.

Step 1 Navigate to the location of the custom repository.

```
$ cd custom-repo.git
```

Step 2 Add all changes made to the repository source at once.

```
$ git add .
```

Step 3 Commit the changes.

```
$ git commit -s -a -m <commit message>
```

All the changes are now added to the repository.

Exporting Committed Changes Internally with Git

When you have a set number of commits to save for future use, you can export them to a master repository using git.

If you simply want to push local changes, use the following reference command:

```
$ git push ssh://master_server/path_to_repo local_branch:remote_branch
```

or

```
$ git push ssh://openlinux.windriver.com/pub/git/kernel-3.14 standard/common-pc:standard/common-pc
```

However, it is possible to create an environment where changes pushed to a local build host's git repository are shared through a master repository with all development machines in a given environment.

The following example demonstrates how a git pull request from a development build host for a specific BSP can be used to automatically update the builds of all users in a central git repository.

Step 1 Navigate to the master git repository.

```
$ cd linux
```

In this example, the name of the repository is **linux**.

Step 2 Create a tag as a reference prior to pulling the changes to the server.

```
$ git tag -d standard/common-pc-mark
```

Step 3 Create a pull request for the changes committed to the master repository.

```
$ git pull ssh://user@localServer/pub/git/kernel-3.10 standard/common-pc:standard/common-pc
```

NOTE: You will need to change *user@localServer* to match the server location in your development environment.

Step 4 Create a tag as a reference to indicate the point where additions were added to the master repository.

Step 5 Pull the changes to the local development build machine.

a) Navigate to the kernel directory on the local development build host.

```
$ cd wr-kernel/git/kernel-3.14
```

b) Fetch the changes from the master git repository.

```
$ git fetch ssh://user@master_server/pub/git/kernel-3.14
```



NOTE: You will need to change `user@master_server` to match the server location in your development environment.

- c) Perform a Linux kernel build from scratch.

```
$ make linux
```

Since the development changes were added to the master git repo, and then propagated back to the development build hosts, they will automatically be checked out and used to create the build.

Exporting Committed Changes Internally as Patches

When you have a set number of commits to save for future use, you can create a patch from the available options.

Use the following available options, depending on your development setup and needs, to create a patch for future use. Before you begin, you should already identify the commits you plan to use to create the patch from.

The reference command for creating a patch is:

```
$ git format-patch -o save_directory first_commit..last_commit
```

Where `save_directory` is the path and name of the directory where the patch will be saved to.

- Run one of the following patch commands from your repository directory.

Options	Description
Create patch from a tree tagged prior to development	<pre>\$ git format-patch -o save_directory tag_number</pre>
Create patch from last commit	<pre>\$ git format-patch -o save_directory HEAD^</pre>
Create patch from last two commits	<pre>\$ git format-patch -o save_directory HEAD^^</pre>
Create patch from the last commit	<ol style="list-style-type: none">Identify the last commit ID number. <pre>\$ git whatchanged</pre>Create the patch from the commit ID. <pre>\$ git format-patch -o save_directory commit_ID</pre>

Exporting Committed Changes Upstream

Once you have created changes that you feel will benefit upstream users, you can send them for review and approval.

Use this procedure to compile a set of commits as a patch, and send it upstream for external submission. For additional information on exporting local changes, see [About Saving Kernel Modifications](#) on page 272.

Step 1 Create a patch from the last four commits.

The following command creates a patch from the last four commits as an example for this procedure. For additional information on creating patches, see [Exporting Committed Changes Internally as Patches](#) on page 277, or refer to your git manual.

```
$ git format-patch --thread -n -o ~/rr/ HEAD^^^
```

Step 2 Send the email using git.

```
$ git send-email --compose --subject '[RFC 0/N] <patch series summary>' \  
--to foo@windriver.com --to bar@windriver.com \  
--cc list@windriver.com ~/rr
```

The editor is invoked for the 0/N patch. Once complete, the entire series is sent for review using the default email for your development host.

Working with -dirty Kernel Strings

If kernel images are being built with **-dirty** on the end of the version string, this means that there are modifications in the source directory that have not been committed.

Non-committed changes should be committed to the kernel tree, even if you do not plan to save, or export them, for future use. Doing so will remove the **-dirty** appendage from the version string.

Step 1 View the git status.

Run the following command from the git repository or any parent directory to identify modified, removed, or added files.

```
$ git status
```

Step 2 Add and commit all changes.

Run the following commands to bulk commit the changes.

```
$ git add .  
$ git commit -s -a -m "getting rid of -dirty"
```

This example provides a simplified means to quickly add and commit changes. It is also possible to commit specific files or files using git by substituting the first command with:

```
$ git add filename
```

Step 3 Rebuild the kernel.

Run the following command from the platform project directory (*projectDir*).

```
$ make linux-windriver.rebuild
```


Patching and Configuring the Kernel

[About Kernel Configuration and Patching](#) 281

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 289
- [Configuring Kernel Modules With Make Rules](#) on page 287
- [Kernel Configuration and Patching with Fragments](#) on page 283

Alternatively, you can patch and/or extend kernel capabilities:

- [Patching the Kernel](#) on page 299
- [Patching the Kernel With SCC Files](#) on page 297

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 123.

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 3.14.

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 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 123.

- 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 35.

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 283.

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_3.14.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 284.

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_3.14.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 284.

Step 1 Clean up the Linux kernel package.

```
$ make 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 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 285, 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 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 284 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 285.

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@gemux86-64: ~# lsmod
```

The system should return the following:

```
Not tainted
pcspkr 2030 0 - Live 0xfffffffffa0002000
```

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 gemux86-64 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@gemux86-64: ~# 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 123.

The **pcspkr** module

The following procedure assumes the **pcspkr** module is available at this point because it was built previously in [Kernel Configuration and Patching with Fragments](#) on page 283

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 289. 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 Configuration and Patching with Fragments](#) on page 283

- a) Add the module.

```
$ make kernel-module-pcspkr.addpkg
```

- b) Verify that the kernel module package has been added.

```
$ cat layers/local/recipes-img/images/wrlinux-image-file-system.bbappend
```

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 123, the command would be:

```
$ cat layers/local/recipes-img/images/wrlinux-image-glibc-small.bbappend
```

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 kernel-module-pcspkr.rmpkg
```

- b) Clean and rebuild the kernel image.

```
$ make kernel-module-pcspkr.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 281.

Step 1 Boot the emulated quemux86-64 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@qemux86-64:~# lsmod
```

The console should return the following output:

```
Not tainted
  parport 23894 1 parport_pc, Live 0xfffffffffa0017000
  floppy 60578 0 - Live 0xfffffffffa0022000
  parport_pc 18367 0 - Live 0xfffffffffa0038000
  minix 29971 0 - Live 0xfffffffffa0042000
```

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@qemux86-64:~# 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@qemux86-64:~# 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 menuconfig in a separate terminal window:	\$ make linux-windriver.menuconfig
Run the graphical xconfig interface to menuconfig:	\$ make linux-windriver.xconfig
Run the graphical gconfig interface to menuconfig:	\$ make linux-windriver.gconfig

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.
- 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.
 - Press **SPACE** twice to remove this module from the build.
The marker is now blank indicating that the floppy module is not selected.
 - Select **Exit** at the bottom menu twice, using **TAB** or left/right arrow keys, to return to the top level list of configuration options.
 - 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.
 - Press **SPACE** twice to remove this module from the build.
 - 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.
- 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.
 - 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.
 - 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.
- 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 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 0xfffffffffa0002000
```

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_hw.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 within 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

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-3.14.22 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_hdwr_frags.txt
- optional_configs.cfg always_hardware.cfg
- specified_hdwr.cfg
- specified_non_hdwr.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 281.
- 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 299.

- 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 299. 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 231.

Step 1 Update the **.bbappend** file to tell BitBake about the the location of the **.scc** file.

- a) Open the **linux-windriver_3.14.bbappend** file in an editor, for example:

```
$ vi layers/local/recipes-kernel/linux/linux-windriver_3.14.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 299.

.scc files provide you with more control on how the kernel changes are applied. For additional information, see [Kernel Patching with scc](#) on page 231.

Step 2 Clean up and rebuild the Linux kernel package and file system.

- a) Clean up the package.

```
$ make 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 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 3.14.6-WR7.0+snapshot-20120807_standard (wruser@my-workstation-11)
(gcc version 4.9.2 (Wind River Linux Sourcery CodeBench 4.9-20) ) #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 281.
- This exercise is an extension of the configuring kernel modules with fragments procedure in *Kernel Configuration and Patching with Fragments* on page 283 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_3.14.bbappend** file in an editor, for example:

```
$ vi layers/local/recipes-kernel/linux/linux-windriver_3.14.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/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 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 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 <wruser@windriver.com>  
Date:   Wed Sep 26 10:16:35 2012 -0400  
  
    init: add 'WR' to the boot label  
  
Signed-off-by: Wind River User <wruser@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 *Step 3*, 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 3.14.6-WR7.0+snapshot-20120807_standard (wruser@my-workstation-11)  
(gcc version 4.9.2 (Wind River Linux Sourcery CodeBench 4.9-20) ) #1 SMP  
PREEMPT Tue Aug 7 12:33:23 EDT
```


Creating Optimized Custom Kernel Builds

[About Optimized Custom Kernel Builds](#) 303

About Optimized Custom Kernel Builds

Wind River provides the capability to create optimized kernel builds for your platform project image. This includes creating an image without a kernel, building from a custom kernel source, or extracting the output for development purposes.

"Dummy" Kernel, Userspace-only Image Builds

If the time to build, deploy and package the kernel is extending the build time of userspace developers, a "dummy" kernel dependency can be used. The dummy kernel recipe meets the rootfs dependency requirement on **virtual/kernel**, and allows userspace to build the same way as if a kernel was build.



NOTE: When building a rootfs with a dummy dependency, boot functionality is not available, since no kernel is present to be deployed.

To create a platform project image file system only, without a kernel, Wind River Linux provides the **linux-windriver-dummy** kernel recipe. Once the build completes, only the file system **.ext3** and **.tar** image files are available in the **projectDir/export/images** directory.

If you specified additional image types using the **--enable-bootimage=** **configure** option, those images will also be available.

IMPORTANT: If you use a dummy recipe as described in this manual, there can be no userspace application that have explicit dependencies on the kernel in the rootfs. Those applications require kernel source code to build, which cannot be provided by the dummy recipe.

Custom Kernel and External Source Builds

Custom kernel and external source builds allow you to choose the kernel or kernel branch that best meets your project requirements. By specifying a custom kernel, this can help save time by avoiding patching at build time.

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.

To use a custom kernel, a copy of the **linux-windriver** kernel tree is created and hosted on an internal git server. A developer makes the required changes and pushes them to the appropriate branch of the local repository. At build time, the platform project uses the **linux-windriver-custom** kernel recipe, and points to the location of the local repository, where the new custom content resides, creating a platform project image with a new, custom kernel.

Developing a platform project image with a customized kernel requires the following:

Cloned kernel tree

By default, the **SRC_URI** setting in the **linux-windriver.bb** kernel recipe points to the default Linus tree located at: <git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git>. To create a custom kernel, you will need a cloned repository, either from the Linus tree, or another source, on your build host.

If you use a kernel tree other than the Linus tree, you will also require the **SRCREV** of the kernel. To obtain this information, navigate to the local location of the cloned repository and enter:

```
$ git whatchanged kernel_branch
```

For information on cloning a kernel tree, see [About Creating and Maintaining Custom Kernel Branches](#) on page 267.

Kernel source

To configure the kernel, you will need the downloaded or cloned kernel source on your build host.

Whether you just need the cloned kernel tree or the tree and source, depends on your project requirements. Using an external source build with a locally checked-out and configured tree, unmodified by the build system, is useful for a developer who is building and testing local development changes to the kernel.

Installing Kernel Source on the Target

By default, kernel source is not included in the target file system, but is required to build the kernel on the target system. If you plan to perform live target kernel updates as part of system operation, you will need to install the source and target build tools.

The following items are required in your platform project build:

- the **feature/target-toolchain** template to provide the necessary toolchain required for on-target builds
- the **kernel-devsrc** package to provide kernel source

You can add these requirements to an existing platform project, or as part of configuring and building a new project. For example, to add the kernel-devel package, you would run the following from the project directory:

```
$ make kernel-develsrc.addpkg
```

As with any platform project image, the platform project must be built and deployed on a live target to perform on-target updates.

To build a new or updated kernel on the target, use the following example commands:

```
# cd /usr/src/linux
# make oldconfig
# make
```

Extracting Kernel Build Output to Build the Kernel Only

To save time during iterative testing of kernel changes, you may want to save deployment and packaging time by building the kernel directly, without the added time required to build the complete system.

To create a compiled **.tar** file of the kernel build output that includes the kernel image and modules, you can use the **KERNEL_EXTRACTDIR_pn-linux-windriver** setting in the **projectDir/local.conf** file to specify an output directory. Once the platform project image is built, the **.tar** file is available in the location you specified.

Creating a Platform Project with a Dummy Kernel

A dummy kernel recipe is used to build a platform project without a Wind River Linux kernel. This approach only builds userspace, preparing the platform project for the new custom kernel..

This procedure requires a previously configured platform project. For additional information, see [About Configuring a Platform Project Image](#) on page 77.

Step 1 Set the **projectDir/local.conf** file to build the dummy kernel recipe.

Open the **projectDir/local.conf** file in an editor and locate the following line:

```
PREFERRED_PROVIDER_virtual/kernel_arch = "linux-windriver"
```

where *arch* refers to your BSP, for example **qemux86** for a qemu-based BSP.

Change the line to the following, and save the file:

```
PREFERRED_PROVIDER_virtual/kernel_arch = "linux-windriver-dummy"
```

This step creates a platform project image with a placeholder kernel recipe that once built, only includes the userspace portion of the project image.

Step 2 Build the platform project image.

```
$ make
```

Once the project build completes, you will have a platform project image built without the kernel. for example, the **projectDir/export** and **projectDir/export/images** directories will have root file system **.ext3** and **.tar** files, but not kernel files.

Building the Kernel Using the Custom Kernel Recipe

A custom kernel recipe is used to specify a custom kernel repository to use to build a Wind River Linux platform project image kernel.

This procedure requires a previously configured platform project, with the following options:

- **--with-layer=wr-kernel/kernel-dev**
- **--with-template=feature/custom-kernel**

For additional information, see [About Configuring a Platform Project Image](#) on page 77.

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 Create a **.bbappend** file for the custom kernel.

In this example, the new **linux-windriver-custom.bbappend** file is placed in the **projectDir/layers/local** directory. If you are creating a custom kernel for a specific BSP, you would place the file in the BSP folder structure.

a) Create the development directory.

```
$ cd projectDir/layers/local  
$ mkdir -p recipes-kernel/linux && cd recipes-kernel/linux/
```

b) Create the **linux-windriver-custom.bbappend** file.

```
$ echo 'SRC_URI = "custom_repo_location"' >> linux-windriver-custom.bbappend
```

This ensures that the kernel will be built from the source specified. For additional information, see [About Optimized Custom Kernel Builds](#) on page 303.

Step 2 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" '\n>> projectDir/bitbake_build/conf/bblayers.conf
```

Step 3 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 4 Optionally update the **SRCREV** for the kernel version being built.

This step is only required if the Linus source tree is not used.

The **SRCREV** 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 3.11.1 kernel the tag is:

```
$ echo 'SRCREV = "5c68732e7504d2e3745c272979c2eeddd7661e5a"' >> linux-windriver-custom.bbappend
```

Step 5 Make the recipe compatible with your machine.

```
$ echo 'COMPATIBLE_MACHINE = "${MACHINE}"' >> 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"
```

NOTE: For CGL kernels you must add a flag in the **local.conf** file by adding the following line: **KERNEL_FEATURES_CLEAR="t"**

Step 7 Comment out the lines related to **systemd** in the **local.conf** file, as show below:

```
# To use systemd as the init system uncomment the next 4 lines
#VIRTUAL-RUNTIME_init_manager = "systemd"
#DISTRO_FEATURES_append = " systemd"
#DISTRO_FEATURES_BACKFILL_CONSIDERED += " sysvinit"
#KERNEL_FEATURES_append = " cfg/systemd.scc"
```

Step 8 Build the kernel.

Run the following command from the top-level folder in the **projectDir**:

```
$ make linux-windriver-custom
```

If you are reasonably sure your kernel is compatible, you can build it into your file system image using:

```
$ make
```

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.

Building the Kernel from External Source

A custom kernel recipe is used to specify a custom kernel repository to use to build a Wind River Linux platform project image kernel.

To complete this procedure, you will require cloned or downloaded kernel source as described in [About Optimized Custom Kernel Builds](#) on page 303.

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 Configure the kernel source.

The following substeps must be completed before you configure a platform project to use the new externally-sourced kernel.

- Navigate to the location of the cloned or downloaded kernel source tree.

For example"

```
$ cd path_to_external_kernel_source
```

- Run the kernel **menuconfig** tool in the kernel source directory to configure your required parameters.

Use this tool to configure aspects of your kernel for your specific requirement.

Once complete, this creates a kernel **.config** file.

- Copy the **.config** file to the **arch/arch/configs/defconfig** directory.

Step 2 Configure a platform project.

For additional information, see [About Configuring a Platform Project Image](#) on page 77.

Step 3 Set the **projectDir/local.conf** file to point to the external kernel source.

Open the **projectDir/local.conf** file in an editor and add the following lines::

```
INHERIT += "externalsrc"  
EXTERNALSRC_pn-linux-windriver = "path_to_external_kernel_source"
```

Step 4 Comment out the lines related to **systemd** in the **local.conf** file, as show below:

```
# To use systemd as the init system uncomment the next 4 lines  
#VIRTUAL-RUNTIME_init_manager = "systemd"  
#DISTRO_FEATURES_append = " systemd"  
#DISTRO_FEATURES_BACKFILL_CONSIDERED += " sysvinit "  
#KERNEL_FEATURES_append = " cfg/systemd.scc"
```

Step 5 Build the kernel.

Run the following command from the top-level folder in the **projectDir**:

```
$ make linux-windriver
```

If you are reasonably sure your kernel is compatible, you can build it into your file system image using:

```
$ make
```

 **NOTE:** This procedure only replaces the kernel and not the file system components.

Extracting the Kernel Build Output

As part of kernel development, you may need to extract the build output. Wind River Linux provides a **make** command option to simplify that task.

This procedure requires a previously configured platform project. For additional information, see [About Configuring a Platform Project Image](#) on page 77.

Step 1 Update the *projectDir/local.conf* file to specify an extract directory.

Add the following line to the *projectDir/local.conf* file.

```
EXTRACTDIR_pn-linux-windriver = "path_to_extract_directory"
```

In this example, *path_to_extract_directory* is the location of where you want the extracted build output to reside.

Step 2 Compile the kernel and kernel modules.

Run the following command from the platform project directory.

```
$ make linux-windriver.compile
$ make linux-windriver.compile_kernelmodules
```

Step 3 Extract the kernel build output.

```
$ make linux-windriver.extract_kernel_output
```

Once the command completes, the kernel build output will be copied to the location specified in the **EXTRACTDIR_pn-linux-windriver** setting in the *projectDir/local.conf*.

20

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}/layers/wr-kernel/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.

→ **NOTE:** Kernels revisions from 3.16 and greater have different build rules and cannot be built using Wind River Linux recipes.

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 3.15 kernel the tag is available in the first command line. Run each command to add all required SRCREV additions to your **linux-windriver-custom.bbappend** file.

```
$ echo 'SRCREV = "1860e379875dfe7271c649058aeddf5af9d0d"' >> linux-windriver-custom.bbappend
$ echo 'SRCREV_machine = "${SRCREV}"' >> linux-windriver-custom.bbappend
$ echo 'SRCREV_meta = "${SRCREV}"' >> linux-windriver-custom.bbappend
```

To obtain a **SRCREV** for a local repository, you must first clone the repository, as described in [About Creating and Maintaining Custom Kernel Branches](#) on page 267. Once you have a local repository, navigate to the local location of the cloned repository and enter:

```
$ git whatchanged kernel_branch
```

The output will provide the **SRCREV** for the kernel revision in the branch.

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 Optionally add the location of the locally cloned kernel repository.

This is required to build from a local git repository.

```
$ echo 'SRC_URI = "git:///path_to_local_git_repository/
linux.git;protocol=file;nocheckout=1"' >> linux-windriver-custom.bbappend
```

Once all additions are made to the **linux-windriver-custom.bbappend** file, it should contain the following content:

```
$ cat linux-windriver-custom.bbappend
FILESEXTRAPATHS := "${THISDIR}/${PN}"
COMPATIBLE_MACHINE = "${MACHINE}"
LINUX_VERSION = "3.15"
SRCREV = "1860e379875dfe7271c649058aeddf5af9d0d"
SRCREV_machine = "${SRCREV}"
SRCREV_meta = "${SRCREV}"
SRC_URI += " file://defconfig"
```

If you are using a local git repository to build your kernel, the file will also include the following line:

```
SRC_URI = "git:///path_to_local_git_repository/linux.git;protocol=file;nocheckout=1"
```

Step 7 Move to the root of your project and edit your **local.conf** file.

```
$ cd ../../..
$ vi local.conf
```

Change your **PREFERRED_PROVIDER_virtual/kernel_BSP_name="linux-windriver"** setting to refer to the **linux-windriver-custom** recipe, and save the file. For example:

```
PREFERRED_PROVIDER_virtual/kernel_qemuppc = "linux-windriver-custom"
```

→ **NOTE:** For CGL kernels you must add a flag in the **local.conf** file by adding the following line: **KERNEL_FEATURES_CLEAR="t"**

Step 8 Build the kernel.

Run the following command from the top-level folder in the **projectDir**:

```
$ make linux-windriver-custom
```

If you are reasonably sure your kernel is compatible, you can build it into your file system image using:

```
$ make
```

→ **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.

Exporting Custom Kernel Headers

About Exporting Custom Kernel Headers for Cross-compile	315
Adding a File or Directory to be Exported when Rebuilding a Kernel	315
Exporting Custom Kernel Headers	316

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/linux/include` directory, or a subdirectory of it. See [Adding a File or Directory to be Exported when Rebuilding a Kernel](#) on page 315 for examples of using `add KERNEL_INSTALL_HEADER_append`.

For instructions on exporting a custom kernel header, see [Exporting Custom Kernel Headers](#) on page 316.

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 316.

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/layer.conf`

Step 1 Open the `projectDir/layers/local/conf/layer.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 += "myfiles/myfile.h"
```

- To add all files in a directory:

```
KERNEL_INSTALL_HEADER_append += "myfiles"
```

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 77.

It also requires that any custom kernel header files that you want to export be located in the `projectDir/build/linux-windriver/linux/include` directory or a subdirectory. For additional information, see [About Exporting Custom Kernel Headers for Cross-compile](#) on page 315.

Step 1 Unpack the Linux kernel.

Run the following command in the root of the `projectDir`.

```
$ make linux-windriver.patch
```

Step 2 Navigate to the source directory of the kernel build.

```
$ cd build/linux-windriver/linux
```

Step 3 Optionally create a file to test this procedure.

If you do not have a file in the `projectDir/build/linux-windriver/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/layers/local/conf/layer.conf` file in a text editor.

Step 6 Add the header file to the `projectDir/layers/local/conf/layer.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 315.

Step 7 Navigate back to the project directory.

```
$ cd ..
```

Step 8 Rebuild the kernel:

```
$ make 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.

Using the preempt-rt Kernel Type

[About Using the preempt-rt Kernel Type](#) 319

[Enabling Real-time](#) 321

[Configuring preempt-rt Preemption Level](#) 321

About 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.



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 321.



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_RTB**. 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 79.

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 81 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 321.

Step 1 Build and open a kernel development shell (kds).

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

```
$ make kds
```

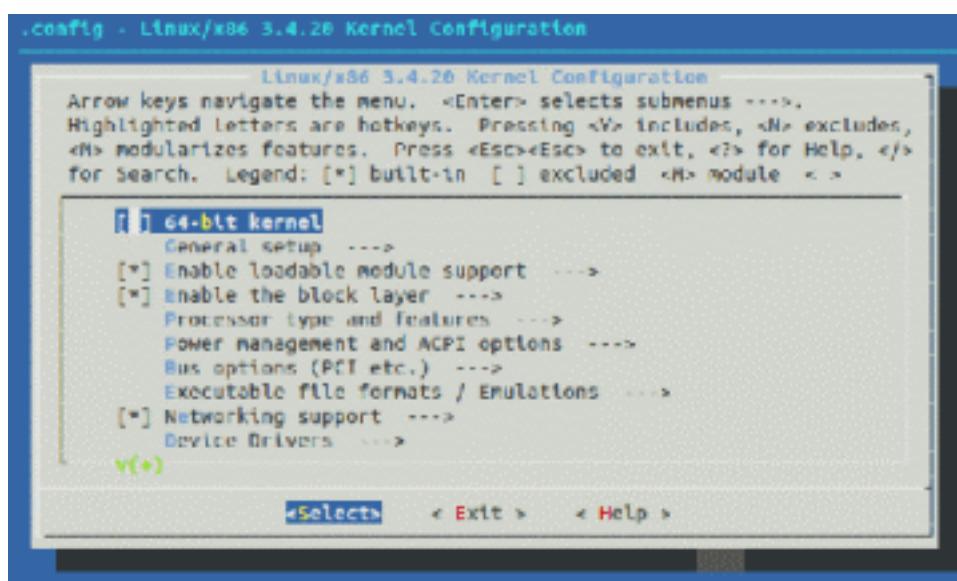
Once the command completes, it will launch a new kds in a separate window.

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.

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 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.....	327
Userspace Debugging.....	333
Analysis Tools Support.....	345

23

Kernel Debugging

[Kernel Debugging](#) 327

[Debugging with KGDB Using an Ethernet Port \(KGDBOE\)](#) 328

[Debugging with KGDB Using the Serial Console \(KGDBOC\)](#) 330

[Disabling KGDB in the Kernel](#) 331

[Kernel Debugging with QEMU](#) 332

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 330.
- 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.

- Type the following **gdb** command:

```
(gdb) info registers
```

You should see the list of registers.

- 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.

- 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) {
```

- 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:

```
# cat /proc/cmdline
```

The command returns the following output:

```
... console=tty02,115200n8 mpurate=auto ...
```

Step 4 Configure kgdboc to use the console device.

```
# echo tty02 > /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 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 linux-windriver.rebuild
```

A new kernel is built and **vmlinu**x 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 [About QEMU Targets](#) on page 355 for more information.

Step 2 Start **gdb** on your workstation.

```
$ ./scripts/gdb export/images/vmlinu-symbols-qemu86-64
```

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.

24

Userspace Debugging

Adding Debugging Symbols to a Platform Project	333
Adding Debugging Symbols for a Specific Package	334
Dynamic Instrumentation of User Applications with uprobes	335
Debugging Individual Packages	342
Debugging Packages on the Target Using gdb	342
Debugging Packages on the Target Using gdbserver	343

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/qemu-x86_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 wrlinux-image-filesystem.distclean</pre>where wrlinux-image-filesystem refers to the name of the projectDir/layers/local/recipes-img/images/wrlinux-image-file-system.bb recipe file, for example, wrlinux-image-glibc-std. This can take a few moments to complete.2. Create the *.debuginfo archive. <pre>\$ make fs-debug</pre>
Previously configured platform project	Run the following command in the projectDir : <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 rootfs**, 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 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

LTTrng (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 though 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/uprobetracer.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.

Figure 5: CONFIG_UPROBE_EVENT=y

Linux Kernel Configuration		
Option	Name	y/n/m
↳ Y Tracers	FTRACE	y
↳ Y Kernel Function Tracer	FUNCTION_TRACER	y
RN Interrupts-off Latency Tracer	IRQSOFF_TRACER	n
RN Preemption-off Latency Tracer	PREEMPT_TRACER	n
RN Scheduling Latency Tracer	SCHED_TRACER	n
Y Trace syscalls	FTRACE_SYSCALLS	y
↳ 8 Branch Profiling		No branch
RN Trace max stack	STACK_TRACER	n
Y Support for tracing block IO actions	BLK_DEV_IO_TRACE	y
Y Enable kprobes-based dynamic events	KPROBE_EVENT	y
Y Enable uprobes-based dynamic events	UPROBE_EVENT	y
Y enable/disable ftrace tracepoints dynamically	DYNAMIC_FTRACE	y
RN Kernel function profiler	FUNCTION_PROFILER	n
RN Perform a startup test on ftrace	FTRACE_STARTUP_TEST	n
RN Memory mapped IO tracing	MMIOTRACE	n

- Enter the following commands from the root of your project directory tree.

```
$ make perf.addpkg
$ make perf
$ make
```

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 333.
- 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 333.
- 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
▽ Y Tracers	FTRACE	y
▷ Y Kernel Function Tracer	FUNCTION_TRACER	y
N Interrupts-off Latency Tracer	IRQSOFF_TRACER	n
N Preemption-off Latency Tracer	PREEMPT_TRACER	n
N Scheduling Latency Tracer	SCHED_TRACER	n
Y Trace syscalls	FTRACE_SYSCALLS	y
▷ B Branch Profiling	STACK_TRACER	n
N Trace max stack	BLK_DEV_IO_TRACE	y
Y Support for tracing block IO actions	KPROBE_EVENT	y
Y Enable kprobes-based dynamic events	UPROBE_EVENT	y
Y Enable uprobes-based dynamic events	DYNAMIC_FTRACE	y
Y enable/disable ftrace tracepoints dynamically	FUNCTION_PROFILER	n
N Kernel function profiler	FTRACE_STARTUP_TEST	n
N Perform a startup test on ftrace	MMIOTRACE	n
N Memory mapped IO tracing		

- 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 Observer 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 333.
- 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 209.
- 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
(N) Interrupts-off Latency Tracer	IRQSOFF_TRACER	n
(N) Preemption-off Latency Tracer	PREEMPT_TRACER	n
(N) Scheduling Latency Tracer	SCHED_TRACER	n
(Y) Trace syscalls	FTRACE_SYSCALLS	y
▷ Branch Profiling	STACK_TRACER	n
(N) Trace max stack	BLK_DEV_IO_TRACE	y
(Y) Support for tracing block IO actions	KPROBE_EVENT	y
(Y) Enable kprobes-based dynamic events	UPROBE_EVENT	y
(Y) Enable uprobes-based dynamic events	DYNAMIC_FTRACE	y
(Y) enable/disable ftrace tracepoints dynamically	FUNCTION_PROFILER	n
(N) Kernel function profiler	FTRACE_STARTUP_TEST	n
(N) Perform a startup test on ftrace	MMIOTRACE	n
(N) Memory mapped IO tracing		

- 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 -aR 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.

Figure 6: perf report Call Tree

```
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
```

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 0x0008dc4)
```

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 213) or after the project is built with the **make gdb.addpkg** command (see [Adding New Application Packages to an Existing Project](#) on page 209).

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 333.

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 213) or after the project is built with the `make gdbserver.addpkg` command (see [Adding New Application Packages to an Existing Project](#) on page 209).

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.

25

Analysis Tools Support

[**About Analysis Tools Support**](#) 345

[**Using Dynamic Probes with ftrace**](#) 345

[**Analysis Tools Support Examples**](#) 351

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**

Figure 7: kprobe Configuration

Linux Kernel Configuration		
Option	Name	y/n/m
▼ Tracers		
▷ Kernel Function Tracer	FTRACE	y
N Interrupts-off Latency Tracer	FUNCTION_TRACER	y
N Preemption-off Latency Tracer	IRQSOFF_TRACER	n
N Scheduling Latency Tracer	PREEMPT_TRACER	n
Y Trace syscalls	SCHED_TRACER	n
▷ Branch Profiling	FTRACE_SYSCALLS	y
N Trace max stack	STACK_TRACER	n
Y Support for tracing block IO actions	BLK_DEV_IO_TRACE	y
Y Enable kprobes-based dynamic events	KPROBE_EVENT	y
Y Enable uprobes-based dynamic events	UPROBE_EVENT	y
Y enable/disable ftrace tracepoints dynamically	DYNAMIC_FTRACE	y
N Kernel function profiler	FUNCTION_PROFILER	n
N Perform a startup test on ftrace	FTRACE_STARTUP_TEST	n
N Memory mapped IO tracing	MMIOTRACE	n

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 ²
+ -offs(FETCHARG)	Fetch memory at FETCHARG + - offs address ³
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/>

² Only for return probe

³ This is useful for fetching a field of data structures

Preparing to use a kprobe

Complete these steps to prepare to debug with a kprobe.

Step 1 Check that your BSP supports kprobes.

For information on supported boards, see *Bootloaders and Board README Files* on page 21

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" qemux86-System.map-WR7.0.0.0_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
-rw-r--r-- 1 root root 0 May 26 13:44 format
-rw-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#  ||||   TIMESTAMP  FUNCTION  
#      | |        |  |||  | | |  
#      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 gemux86-64 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..... 355

Wind River Simics Targets..... 365

QEMU Targets

About QEMU Targets	355
QEMU Target Deployment Options	356
Managing QEMU Targets	358
TUN/TAP Networking with QEMU	361

About 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 356.

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
```

Managing 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 the **-in** 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="-in 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 383.

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 383 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="-in 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 Workbench for a New Connection

Use this information as a guideline for configuring TUN/TAP using the Wind River Workbench Systems Management, New Connection window.

For a new QEMU connection, use the following procedure to set up TUN/TAP from within Wind River Workbench.

Step 1 Start Wind River Workbench.

Step 2 Click the **New Connection** button in the Systems Management toolbar or view to launch the New Connection window.

Step 3 Select the connection type.

Since we are accessing TUN/TAP settings for a QEMU deployment, choose **QEMU**, then click **Advanced**.

Step 4 Select the platform project or SDK root directory.

Click **Browse**, and navigate to and select the platform project or SDK root directory.

Step 5 Click **Advanced** to display the Advanced Settings window.

Step 6 Update TUN/TAP settings.

In the Configuration section of the Advanced Settings window, make changes as necessary to the default TUN/TAP settings.

Once complete, click **OK** to close the Advanced Settings window and return to the New Connection window.

Step 7 Click **Finish** to complete the connection.

If you want to launch the target immediately, select **Connect on finish** prior to clicking **Finish**.

Configure TUN/TAP in Workbench for Existing Target Connections

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 the target connection's details.

In the Systems Management toolbar, click the connection name, for example **QEMU**, and select **Open Connection Details**.

The details appear in a tab in the main Editor view.

Step 3 Click **Advanced** to open Advanced Settings window.

Step 4 Update TUN/TAP settings.

In the Configuration section of the Advanced Settings window, make changes as necessary to the default TUN/TAP settings.

Once complete, click **OK** to close the Advanced Settings window and return to the details tab.

Step 5 Optionally connect to the target to test the settings.

For additional information, see the *Wind River Workbench by Example, Linux Version*.

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.

27

Wind River Simics Targets

[About Wind River Simics Targets](#) 365

[Using Simics from the Command Line](#) 365

About 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 365.

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 366.

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 366.

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 366.

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.....	371
Deploying Flash or Disk Target Platforms.....	383
Deploying initramfs System Images.....	421
Deploying KVM System Images.....	425

Managing Target Platforms

Customizing Password and Group Files	371
About ldconfig	375
Connecting to a LAN	376
Adding an RPM Package to a Running Target	377
Adding Reference Manual Page Support to a Target	378
About Compressing Documentation on Targets	379
Using Pseudo	380

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.

Another possibility is to create user accounts and passwords as part of installing RPMs on a deployed system, as described in [Lua Scripting in Spec Files](#) on page 533. If you have users accounts which are associated with application(s) delivered in the RPM, this is a good choice. This presumes that you have included the RPM package in your target file system build.

The preferred method is to create user accounts and passwords as part of the build. The build system supports creating users and groups with the `EXTRA_USERS_PARAMS` macro. You can add this information to any recipe file, but for a project-specific set of users, a good file to use is `projectDir/layers/local/recipes-img/images/wrlinux-image-filesystem.bb`.

The content of this macro is a series of shell standard commands that add, remove, or modify user, group and password to a file system. Since they are executed by the build in the pseudo

environment, no administrative privileges are required for the build system. The following commands are supported by the macro:

useradd

Add (create) a single user at build time.

userdel

Delete a user at build time.

usermod

Modify a user account at build time.

groupadd

Add (create) a single group at build time.

groupdel

Delete a group at build time.

groupmod

Modify a group at build time.



NOTE: Consult the man pages of any Linux system for detailed, command-specific usage information.



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 212) 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.

Add Users Example

The following example code informs the system that there are users to add, and adds two new users, **tester** and **developer**, to the system. In addition, it provides an example for deleting user **nobody**.

```
# Inform build system we have users to add
inherit extrausers
EXTRA_USERS_PARAMS = " useradd tester; \
                     useradd developer; \
                     userdel nobody;"
```

Set Default Password for User Accounts Example

Depending on your needs, you may want to set a default password for the accounts that you create. The supported commands require encrypted passwords when specified. You can copy the encrypted strings into your recipe.

The command **openssl passwd** is the most convenient as it prefixes the key value to the front of the password it returns, which is what the command expects, and supports the alternative, more secure MD5 encryption format with the **-1** option flag.

The following example sets the passwords **secret1** and **secret2** for users **tester** and **developer**, respectively.

First, use **openssl** to determine the password:

```
# openssl passwd -1 secret1  
$1$9a9Lk2CS$KIj4SYe3mA857sQAg0Npw/
```

Notice that the encrypted password displays. Retain this encrypted output to add to the recipe file.

Then, perform the step again for another password, retaining the output:

```
# openssl passwd -1 secret2  
$1$hBhTGQCh$sG3DBc745SHX4vK/O6G0T/
```

Then, use those values to update the passwords for the recipe file:

```
EXTRA_USERS_PARAMS += " \  
    usermod -p '$1$9a9Lk2CS$KIj4SYe3mA857sQAg0Npw/' tester; \  
    usermod -p '$1$hBhTGQCh$sG3DBc745SHX4vK/O6G0T/' developer;"
```

In this example, the encrypted value for the password **secret 1** was added to the recipe for user **tester**, and the encrypted value for the password **secret 2** was added to the recipe for user **developer**.

Add Groups Example

The following example code adds the groups **test**, **dev**, and **pubs** to the system.

```
EXTRA_USERS_PARAMS = " groupadd test; \  
    groupadd dev; \  
    groupadd pub;"
```

Map Fixed User Name IDs to an Account

The following example code maps existing user IDs from a valid user name to a specific utility account, either **games**, **man**, or **mail**. This example requires the user ID, as displayed when you run the **ls -l** command.

```
EXTRA_USERS_PARAMS += " usermod -u 65 games; \  
    usermod -u 66 man; \  
    usermod -u 68 mail; \  
    usermod -u 1892 not-exist;"
```

If you use the **changelist.xml** file to add and assign ownership to a file or file(s), the ID numbers used in the changelist.xml file must match the IDs set using this example. This is required, because the **changelist.xml** IDs are processes before the user accounts are created. Essentially, a mis-match of IDs can set ownership to a random ID instead of the user (or users) it is intended for. For additional information, see [About File System Layout XML Files](#) on page 177.

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 212 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 `libc_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 212 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 375 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...
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...
#####
1:dash
#####
[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      pkcs8.1ssl1.gz
CA.pl.1ssl1.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]
```

About Compressing Documentation on Targets

Use the **--with-doc-compress** `configure` script option to compress man and info pages to save disk space on the target.

You can enable and specify documentation compression options on your platform project image. On the target, documentation is automatically decompressed when required. The **--with-doc-compress=** option lets you choose from **xz** (LZMA), **gz** (gzip), or **bz2** (bzip2) compression types. For additional information, see [Configure Options Reference](#) on page 82.

Once the `configure` script completes, the following is added to the `projectDir/local.conf` file:

```
INHERIT += "compress_doc"
DOC_COMPRESS = "gz"
```

In this example, the **gz** (gzip) compression type was specified with the **--with-doc-compress=gz** `configure` option. Once you build, or rebuild, the platform project image, document compression will be enabled on the target system image.

To change this setting, you can rerun the `configure` command with a new compression option, or edit the `projectDir/local.conf` file directly.

Compressing Target Documentation

Learn how to add target-based documentation compression on an existing platform project image.

The following procedure requires a previously built platform project. For additional information, see the *Wind River Linux Getting Started Guide: Developing a Platform Project Image Using the Command-Line*.

Step 1 Configure platform project.

```
$ configDir configure --enable-board=qemux86-64 \
--enable-kernel=standard \
--enable-rootfs=glIBC-std \
--enable-parallel-pkgbuilds=4 \
--enable-jobs=4 \
--enable-reconfig \
--with-doc-compress=gz
```

Step 2 Optional: Verify the compression type in the `projectDir/local.conf`.

The compression option produces the following output in the `local.conf` file:

```
INHERIT += "compress_doc"
DOC_COMPRESS = "gz"
```

You can change the compression type at this point or accept the default. For example, you could replace the **gz** with **xz**, or **bz2**.

Step 3 Build the platform project image.

```
$ make
```

Step 4 Add the documentation packages that you require.

```
$ makevim-doc.addpkg
```

You can add other documentation packages, or accompanying documentation for any custom packages that you may need to add to the target by running the command shown above.

Step 5 Optional: View the documentation on the target.

- Start the target.

```
$ make start-target
```

After entering username and password on the target you can open the compressed documentation.

- Open a documentation file to validate that it has been compressed.

Run the following command to open and decompress the man page entry for the VIM program.

```
# man vim
```

The command produces the man page for VIM. Note that it takes some time to decompress it, based on size of the documentation and the target processor speed.

```
VIM(1)           General Commands Manual          VIM(1)
NAME
      vim - Vi IMproved, a programmers text editor
SYNOPSIS
      vim [options] [file ...]
```

Using Pseudo

About Using Pseudo

Pseudo is a utility that provides emulated root permissions for *projectDir/export/dist*.

Pseudo acts as a root console for the target rootfs. It enables you to make file system changes without having to deploy the build on a target. It provides one database of virtual ownerships and permissions per package, and also one for the target rootfs in *projectDir/export/dist*. A wrapper, **fakestart.sh**, uses pseudo to run commands in the emulated root environment for the target rootfs.

You start the pseudo shell using **fakestart.sh**. Once in the shell, you can look at **export/dist** and see the target's permissions and run commands.

Examining Files using Pseudo

You can examine 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 `fakeshell.sh`. For example:

- `$ scripts/fakeshell.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

You can view the target file systems using a pseudo shell.

You can enter a pseudo shell to move around the target file system and view multiple settings.

Step 1 Start the pseudo shell from your project directory:

```
$ scripts/fakeshell.sh sh
```

Notice that the shell prompt changes to # to indicate you are in the pseudo shell.

Step 2 Navigate to the platform project bin (binary) directory.

```
# cd export/dist/bin
```

Step 3 List the directory's contents.

```
# 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.

```
#exit
```

The `exit` command closes the pseudo shell and returns to your normal shell.

29

Deploying Flash or Disk Target Platforms

About Configuring and Building Bootable Targets	383
Host-Based Installation of Wind River Linux Images	384
Booting and Installing from a USB or ISO Device	388
Booting and Installing with QEMU	389
Creating u-boot RAM Disk Images	392
Creating Bootable USB Images	394
Creating ubifs Bootable Flash Images	399
Enforcing Read-only Root Target File Systems	400
Installing with the GUI Installer	400
Installing with the Serial Console Installer	408
Installing or Updating bzImage	409
About Manually Configuring a Boot Disk with Files from a USB/ISO Image	411
About Deploying an Image with a Virtual Machine Manager	416

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**, **vmdk**, **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 394.

For additional information on configuring ubifs boot images, see [Creating ubifs Bootable Flash Images](#) on page 399.

For additional information on creating u-boot images, see [Creating u-boot RAM Disk Images](#) on page 392.

For additional information on creating vmdk images, see [About Deploying an Image with a Virtual Machine Manager](#) on page 416.

The following is a sample **configure** script 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

To specify an amount of additional free space on the bootimage in kilobytes, use the following, additional configure option:

--with-bootimage-space=512

In this example, the resulting bootimage would have an additional 512 kilobytes of free space. You can set this amount to meet your project needs.

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.

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.

About Installation Images

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 389. Once complete, the installer lets you specify or accept default sizes for the boot, swap, and root partitions.

To create an image with an installer for deployment to a host system, see [Configuring and Building the Host Install](#) on page 387.

The related installer options for the **configure** command are as follows:

Reference platform project

The reference platform project provides the build directory that supplies the RPMs that are installed on the target. If this path is an .ext2, .ext3, or .ext4 image file, then the installer will copy this image to the target after the disk has been partitioned and formatted. There is one reference project-specific configure option:

```
--enable-rootfs=glibc_std+installer-support
```

In this example, the **installer-support** template is appended to the root file system selection. This is required to ensure that the reference platform project has the necessary files to create an installable image from.

Installer platform project

The installer platform project creates the bootable image using the packages from the reference platform project. These packages come from the reference platform project in two methods, for example:

- The .ext2, .ext3, or .ext4 image files, located in the **projectDir/export** directory.
- The RPMs located in the directory and sub-directories of the reference platform project.

Either option requires two builds - one for the reference platform project, and another to create the installer image. Installer-specific configure options include:

```
--enable-rootfs=wr-installer  
--enable-bootimage=iso  
--enable-target-installer=yes  
--with-installer-target-build=reference_projectDir/export/images/*.ext3
```

The **--enable-rootfs=wr-installer** option is required to create the bootable image.

Once the build is finished, you will find the ISO image in, for example, **projectDir/export/images/intel-x86-64-installer-standard-dist.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.

About Installer Support

Once you create an installer image, as described in *Configuring and Building the Host Install* on page 387, the installer used to install the image to hardware provides support for a number of features, including Red Hat Kickstart installations, upgrades from existing installations, and support for multiple target builds on a single installation disk. The installer supports grub, version 2.

Performing Red Hat Kickstart installations

The Kickstart installation method provides automatic, unattended operating system installation and configuration. This is made possible using the **/root/anaconda-ks.cfg** file, and specifying this file in the **projectDir/local.conf** file of the installer platform project build. Once you make changes to this file, you update the **local.conf** file to point to your new, updated file location, for example:

```
KICKSTART_FILE = "/path_to_my/anaconda-ks.cfg"
```

This setting will become part of the installer build, and will begin the Kickstart installations by default when you launch the installer image on the target platform. For information on the /root/anaconda-ks.cfg file options, see <http://fedoraproject.org/wiki/Anaconda/Kickstart>.

Upgrading a previously installed product version

It is possible to upgrade a previously installed product version, provided the following conditions are met:

- Only the installer disk created with the RPMs, by setting the **--with-installer-target-build=referenceProjectDir** may be used to perform an upgrade, and not an installer image used by specifying the reference platform project's .ext2, .ext3, or .ext4 image files.
- The previous and current build must use the same PRServer, and have one PRServer instance used when creating the reference and installer platform project images. For additional information on synchronizing PRServers, see [About Package Revision Management](#) on page 251.
- The previous and current reference build must have the same root file system, for example **glibc-std**. It is not possible to upgrade and change root file systems.

To perform an upgrade, boot the target system as described in [Booting and Installing from a USB or ISO Device](#) on page 388. When prompted, select **Upgrade Existing Installation**.

Support for multiple target builds

When you create an installer image as described in [Configuring and Building the Host Install](#) on page 387, you specify the packages to use from a reference platform project in the **--with-installer-target-build=** option, for example:

```
- --with-installer-target-build=reference_projectDir
```

To include additional target builds, create the reference platform projects, and then add the project directories to the same configure option used to create the installer image, for example:

```
- --with-installer-target-build=reference_projectDir1,reference_projectDir2,reference_projectDir3
```

When you build the installer image, it will contain separate installs for each reference platform project. To choose a specific image to install, boot the installer image on the target machine.

You will be prompted to select the target image to install, for example:

```
===== Found the following products =====
1) DISTRO1    target_build1    DISTRO_NAME1    DISTRO_VERION1
2) DISTRO2    target_build2    DISTRO_NAME2    DISTRO_VERION2
3) DISTRO3    target_build3    DISTRO_NAME3    DISTRO_VERION3

Please enter your choice (0 to quit):
```

In this example, **target_buildn** is derived from the name of the reference platform project directory. Depending on the distribution of your installer image, you may want to set reference platform project directory names accordingly.

Multiple target builds also support using Kickstart installations. For this to work properly, the entries in the **projectDir/local.conf** file for and **KICKSTART_FILE=** must include references to a separate *.ks file for each installation, for example:

```
KICKSTART_FILE = "/path_to_my/target1-ks.cfg /path_to_my/target2-ks.cfg \
/path_to_my/target3-ks.cfg"
```

Configuring and Building the Host Install

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 functionally.

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 387, 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.

The platform project that you use for reference must be configured and built using the **installer-support** template. The easiest way to do this is to append the **--enable-rootfs=configure** script option with the template, for example: **--enable-rootfs=glibc-std+installer-support**.

- One project that will furnish the boot image for your targets, as described in this procedure.

Step 1 Configure and build a platform project.

The packages assembled in this project are the ones you will install on your target(s). If you already have a reference platform project build to use, go to 3 on page 387.

For example:

```
$ configDir/configure \
--enable-board=intel_x86-64 \
--enable-kernel=standard \
--enable-bootimage=ext3 \
--enable-rootfs=glibc_std+installer-support
```

The **--enable-bootimage=ext3** option specifies the creation of a platform project **.ext3** image file in the **projectDir/export** directory once the project is built. If you plan to use the reference platform project directory, and not the **.ext3** file to add packages to your installer platform project, this setting is optional.

Step 2 Build the project.

```
$ make
```

Step 3 Create a second platform project to configure and build your bootable image.

This platform project requires a separate build directory. If one does not exist, you will need to create it for this step.

The second platform project is used to create the installer image. It does this by using the packages, or disk images, from the platform project built in the previous step.

When you configure this installer platform project 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=intel_x86-64 \
--enable-kernel=standard \
--enable-rootfs=wr-installer \
--enable-bootimage=iso \
--enable-target-installer=yes \
--with-installer-target-build=otherprojectDir/export/images/wrlinux-image-glibc-std-
intel-x86-64-201407210107.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.

The **--enable-rootfs=wr-installer** and **--enable-target-installer=yes** options are required to create a bootable image.

In this example, the **--with-installer-target-build=** option points to the .ext3 file of the reference project. This will copy the contents of the root file system from the reference project to create the bootable image. You may also specify the root directory of the reference platform project, for example: **--with-installer-target-build=otherprojectDir**.

Step 4 Create the boot image.

Run the following command from the installer **projectDir** to create the bootable image.

```
$ make
```

The result is a bootable image in the **export** subdirectory of your project build directory, for example

projectDir/export/intel-x86-64-installer-standard-dist.iso

For information on installing the image, see:

[Booting and Installing with QEMU](#) on page 389

Use this procedure to test your installation on a virtual disk.

[Booting and Installing from a USB or ISO Device](#) on page 388

Use this procedure to install your image directly to hardware.

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 functionally.

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](#) on page 387

Step 2 Burn the image to the USB or ISO device.

For information about burning boot images, see [Creating Bootable USB Images](#) on page 394

Step 3 Insert the device in the target to be booted.

Options	Description
USB	<p>Insert the device and determine the USB device assigned to the drive.</p> <p>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 389 of this procedure, below.</p> <p>In the (likely) event that /dev/sda 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.</p>
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 with the Serial Console Installer](#) on page 408.

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.

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.

This procedure requires an installable disk image created from [Configuring and Building the Host Install](#) on page 387.

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.

- Boot the **.iso** image you created in [Configuring and Building the Host Install](#) on page 387.

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="-m 2048 -cd export/intel-x86-64-installer-standard-dist.iso \
-no-kernel -disk hd0.vdisk -gc" EX_TARGET_QEMU_OPTS="-vga vmware"
```

NOTE: The **-gc** options starts the QEMU session in graphics mode. Omit this option to perform a serial-based, non-graphical installation.

The **EX_TARGET_QEMU_OPTS** option specifies additional video specific options. In this example, **-vga vmware** specifies a VMWare-capable VGA setting. To start a VNC-capable session, add **-vnc :4** to **EX_TARGET_QEMU_OPTS**.

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.

- Select **Graphics console** and press **ENTER**.



NOTE: If you did use the **-gc** option in step 2 on page 390, above, select **Serial console**, and refer to the instructions at *Installing with the Serial Console Installer* on page 408. Once the hard disk installation completes, go to step 3 on page 391, below.

Once you select the installation option, the installation will begin automatically. For instructions:

Installing with the GUI Installer on page 400

Use this option to perform a GUI-based installation

Installing with the Serial Console Installer on page 408

Use this option to perform a serial console-based installation.

Step 3 Boot the installed disk.

In this example, the installation was performed on the virtual disk you created in step 1 on page 390, above.

- 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.

- 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 Log in to the target console.

At the **localhost login:** prompt, enter **root** for the user name and the password you created when you performed the installation, then press **ENTER** to continue.

Creating u-boot RAM Disk Images

Learn how to create a u-boot RAM disk image with or without compression and set kernel options necessary to support booting u-boot on the target.

Wind River Linux provides the option to create a **u-boot** target image.

Step 1 Create a **u-boot** target image using the following platform project configure options:

Options	Description
Specified compression type	Run the following configure command: <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_std \ --enable-bootimage=imageType.compressionType.u-boot</pre>

Options	Description
No compression type	<p>Run the following configure command:</p> <pre>\$ configDir/configure \ --enable-board=qemux86-64 \ --enable-kernel=standard \ --enable-rootfs=glibc_std \ --enable-bootimage=imageType.u-boot</pre> <p>Note that this is similar to configuring a platform project with compression, and that the option to specify compression is not included.</p>

In these examples, the *imageType* and *compressionType* define the resulting u-boot image that will be created in the **projectDir/export** directory once the project is built. For additional information on all available options, see [Configure Options Reference](#) on page 82.



NOTE: These examples use a **qemux86-64**-based board. To successfully boot an image to the target hardware, you need to use a hardware target BSP.

Step 2 Build the target platform image.

```
$ make
```

The image is created in the **projectDir/export** directory.

Step 3 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 menuconfig in a separate terminal window:	<pre>\$ make -C build linux-windriver.menuconfig</pre>
Run the graphical xconfig interface to menuconfig :	<pre>\$ make -C build linux-windriver.xconfig</pre>
Run the graphical gconfig interface to menuconfig :	<pre>\$ make -C build linux-windriver.gcconfig</pre>

After a few seconds, a new terminal window displays with the kernel configuration menu.

Step 4 Set kernel options for u-boot support.

- a) From the top kernel configuration menu, select **General setup > Initial RAM filesystem and RAM disk (initramfs/initrd) support** to expand the category.

- b) Select all required compression options, based on your **--enable-bootimage=** choices.

For example, if you specified **bzip2** as a *compressionType*, it must be enabled in the kernel. The supported compression options include:

- Support initial ramdisk compressed using gzip.
- Support initial ramdisk compressed using bzip2
- Support initial ramdisk compressed using LZMA

- c) Select **Exit** to return to the main menu.
d) Save the new kernel configuration.

Select **Exit** to return to the main menu, and **Exit** again to close the configuration window. When prompted, select **Yes** to save the new configuration.

Step 5 Rebuild the kernel by running the following command before running the **make** command in Step 6.

```
$ make -C build linux-windriver.rebuild
```

Step 6 Build the file system.

Run the following command from the *projectDir*:

```
$ make
```

Once the build completes, your u-boot platform project image is ready in the *projectDir/export* directory. The file name depends on your choices specified in the **--enable-bootimage=configure** option. For example, an image specified with **ext4** root file system with **bz2** compression, the file name would be:

projectDir/export/qemux86-64-glibc-std-standard-dist.ext4.bz2.u-boot

Step 7 Optionally boot your image on a hardware target.

This procedure varies with your hardware. Refer to your BSP README or documentation for additional information.

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 383 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	<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> <hr/> <p>NOTE: For additional information on configuring platform projects, see About Configuring a Platform Project Image on page 77.</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> <ul style="list-style-type: none"> 16 FAT The first is a small 16 FAT file system for syslinux, the kernel, and a static BusyBox initrd ext2 The second is an ext2 file system to mount the root partition for the operating system <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 <code>make usb-image-burn</code>	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:	<code>\$ make</code> 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 <code>/export</code> directory, named after the platform project name, for example: <code>projectDir/export/qemux86-64-boot.iso</code>
For Method 2:	<code>\$ make usb-image</code> When you run this command, it creates a bootable image file in the following location: <code>projectDir/export/usb.img</code>
For Method 3:	<code>\$ make usb-image-burn</code> This option requires that you have inserted a USB device with no mounted partitions. For an example output of this script, see make usb-image-burn Example Output on page 397. Once you run this command, go to <i>Step 5</i> , below.

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:

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:	<code>\$ dd if=export/qemux86_64-boot.iso of=/dev/sdc bs=1M</code>
For Method 2:	<code>\$ dd if=export/usb.img of=/dev/sdc bs=1M</code>



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.

make usb-image-burn Example Output

Use **make usb-image-burn** to create and write a bootable USB image directly to a USB thumb drive.

When you run **make usb-image-burn** from a platform project directory, the following system output and prompts display. For additional information on related procedures, see [Creating Bootable USB Images](#) on page 394.

```
!!!!!!WARNING This program must run as root!!!!!!
The program writes to raw devices.
Please become root and run the program
Attempting to run: sudo ./scripts/create-usb.pl --usbimg
[sudo] password for WRSuser:
=====
Welcome to the usb disk creation helper
!!WARNING!! Use this program with care as it has the possibility
to DESTROY data on any attached storage on your host system.
=====
Continue [y/n]: y
Detected devices:
/dev/sda - mounted, not an install candidate
/dev/sdb - possible install candidate
Device to install wrlinux on to [/dev/sdb]:
Format device? <y/n> [y]: y
Size of FAT16 boot <#MEGS> [64]: 
Size of ext2 fs <#MEGS OR all> [all]: 128
Location of file system tar.bz2 [/space/jw/4/cpc_small/export/qemux86_64-standard-
glibc_small-dist.tar.bz2]:
Location of bzImage [/space/jw/4/cpc_small/export/qemux86_64-bzImage-
WR4.2.0.0_standard]:
Make root file system readonly? <y/n> [y]: n
RUN: parted -s /dev/sdb mklabel msdos
RUN: parted -s /dev/sdb print
Model: FLASH Drive UT_USB20 (scsi)
Disk /dev/sdb: 4041MB
Sector size (logical/physical): 512B/512B
```

```
Partition Table: msdos

Number Start End Size Type File system Flags
RUN: parted -s /dev/sdb mkpart primary fat16 0 64
Warning: The resulting partition is not properly aligned for best performance.
RUN: fdisk /dev/sdb <<EOF
n
p
2

+128M
w
EOF

WARNING: DOS-compatible mode is deprecated. It is strongly recommended to
switch off the mode (command 'c') and change display units to
sectors (command 'u').

Command (m for help): Command action
  e   extended
  p   primary partition (1-4)
Partition number (1-4): First cylinder (70-4384, default 70): Using default value 70
Last cylinder, +cylinders or +size{K,M,G} (70-4384, default 4384):
Command (m for help): The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
RUN: partprobe
RUN: sync
RUN: dd conv=notrunc bs=440 count=1 if=/space/jw/4/cpc_small/host-cross/share/syslinux/
mbr.bin of=/dev/sdb
1+0 records in
1+0 records out
440 bytes (440 B) copied, 0.0122171 s, 36.0 kB/s
RUN: parted /dev/sdb set 1 boot on
Information: You may need to update /etc/fstab.

RUN: fdisk -l /dev/sdb

Disk /dev/sdb: 4040 MB, 4040748544 bytes
200 heads, 9 sectors/track, 4384 cylinders
Units = cylinders of 1800 * 512 = 921600 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x0000c44a

      Device Boot      Start        End      Blocks   Id  System
/dev/sdb1    *          1         70      62500    e  W95 FAT16 (LBA)
/dev/sdb2            70        216     131899+  83  Linux
RUN: sync
RUN: mkdosfs /dev/sdb1
mkdosfs 3.0.9 (31 Jan 2010)
RUN: syslinux /dev/sdb1
RUN: mke2fs -L wr_usb_boot /dev/sdb2
mke2fs 1.41.9 (22-Aug-2009)
Filesystem label=wr_usb_boot
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
33048 inodes, 131896 blocks
6594 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=67371008
17 block groups
8192 blocks per group, 8192 fragments per group
1944 inodes per group
Superblock backups stored on blocks:
 8193, 24577, 40961, 57345, 73729

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 27 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
```

```
RUN: mount /dev/sdb2 /tmp/tmp.gXLbD10594
==Copying files to media, each . == 1000 files copied, this may take a while==
...
==Unmounting and syncing, may take some time....=
RUN: umount /tmp/tmp.gXLbD10594
RUN: sync
RUN: mcopy -o /space/jw/wr_git/wrlinux-x/layers/ldat-tools/installer/dist/syslinux/
devices.txt /space/jw/wr_git/wrlinux-x/layers/ldat-tools/installer/dist/syslinux/
help.txt /space/jw/wr_git/wrlinux-x/layers/ldat-tools/installer/dist/syslinux/
splash.lss /space/jw/wr_git/wrlinux-x/layers/ldat-tools/installer/dist/syslinux/
splash.txt m:
RUN: mcopy -o /space/jw/4/cpc_small/host-cross/share/syslinux/isolinux.bin /space/jw/4/
cpc_small/host-cross/share/syslinux/vesamenu.c32 /space/jw/4/cpc_small/host-cross/
share/syslinux/menu.c32 /space/jw/4/cpc_small/syslinux.cfg m:
RUN: mcopy -o /space/jw/4/cpc_small/export/qemux86_64-bzImage-WR4.2.0.0_standard
m:vmlinuz
RUN: mcopy -o /space/jw/wr_git/wrlinux-x/layers/ldat-tools/installer/dist/syslinux/
busybox-initrd-static m:initrd
```

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 383 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
```

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.

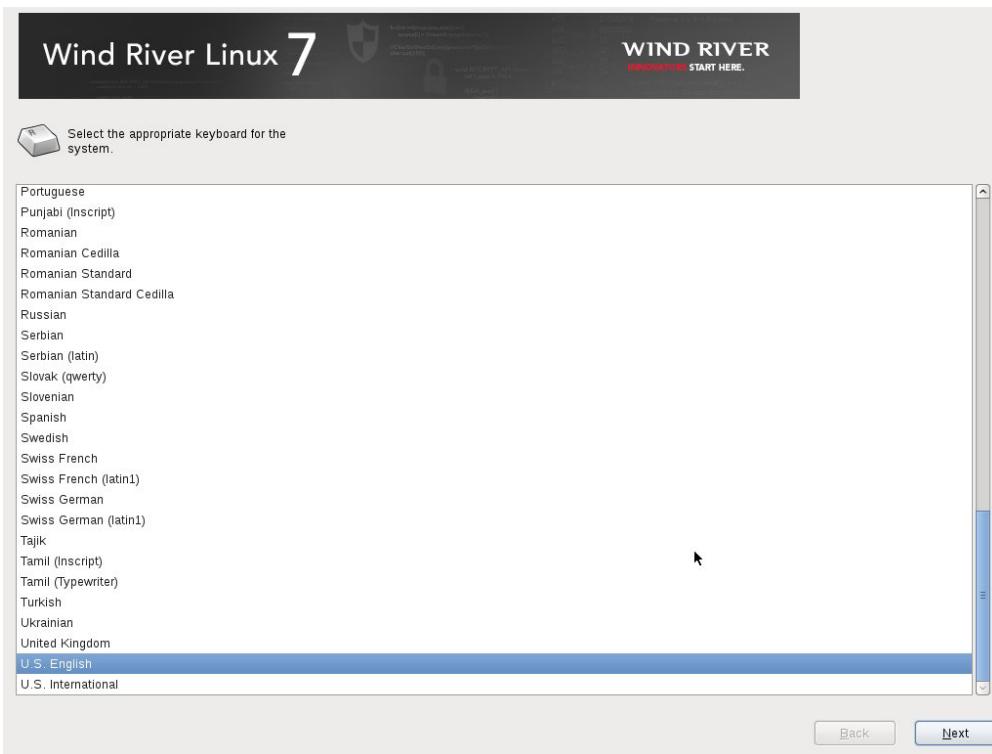
Installing with the GUI Installer

Use this procedure to install an installable Wind River Linux image that you create using the `--enable-target-installer=yes` configure option.

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 389. This example installs the distribution to the entire disk, wiping any existing data on it.

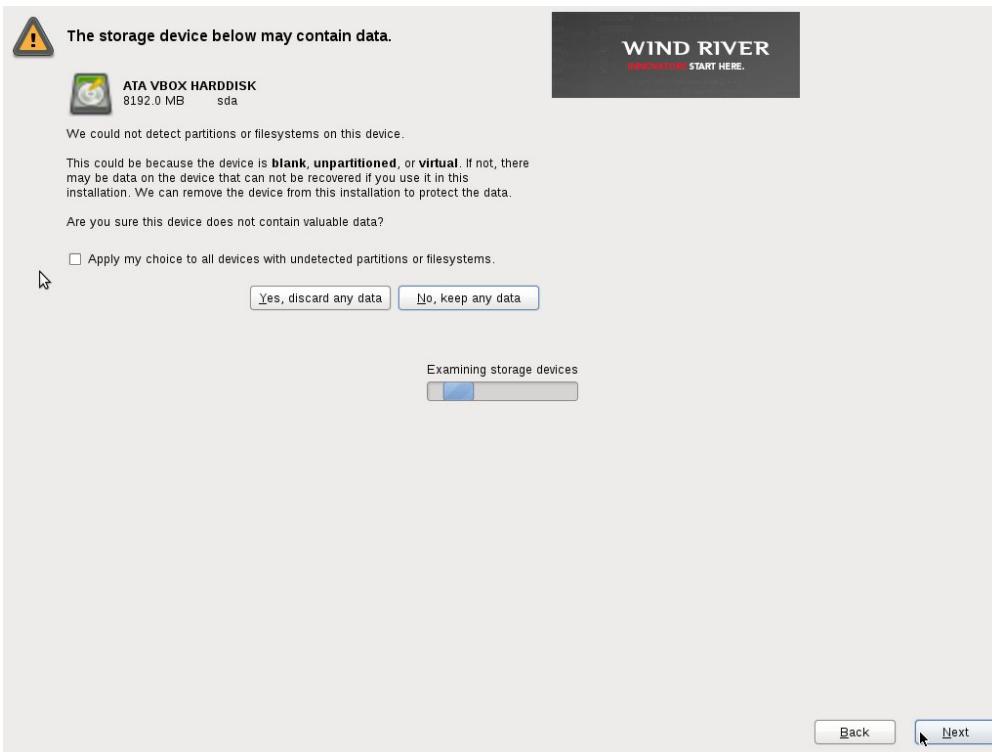
This procedure assumes that you have previously created an installable .iso disk image as described in [Configuring and Building the Host Install](#) on page 387, and started the installer. The first dialog to display lets you select the appropriate keyboard language for the system.

Step 1 Choose the keyboard language, then click **Next**.



For QEMU-based installation, press **CTRL-ALT** at any time to release the mouse from the installer window.

Step 2 Select the storage option.

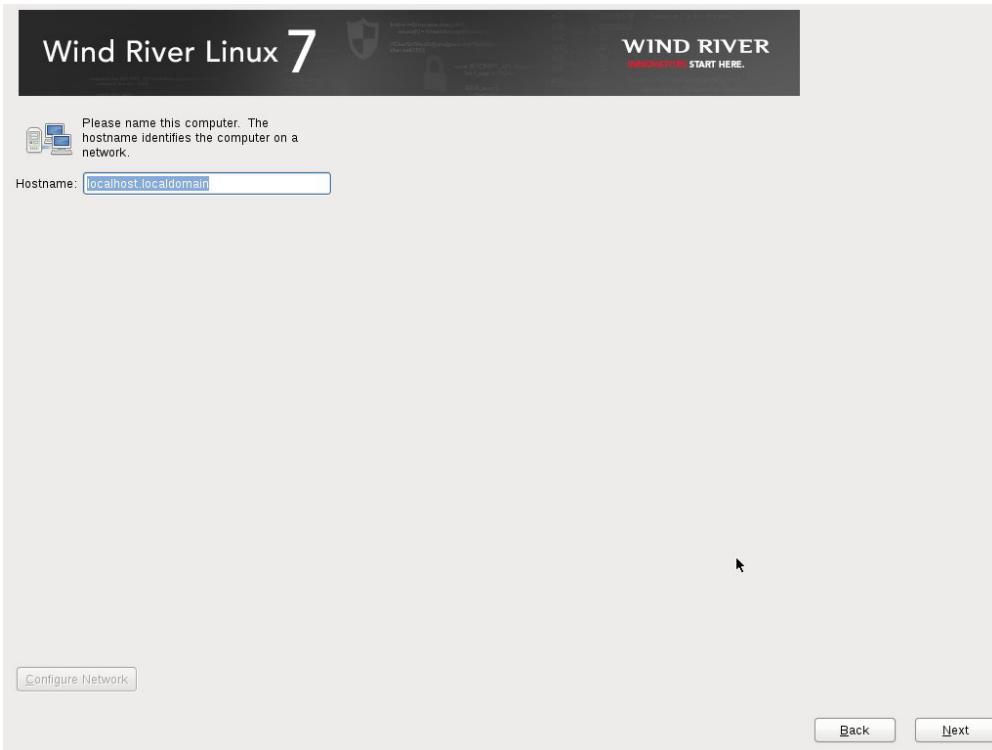


Select **Yes, discard my data** to install the distribution to the disk.

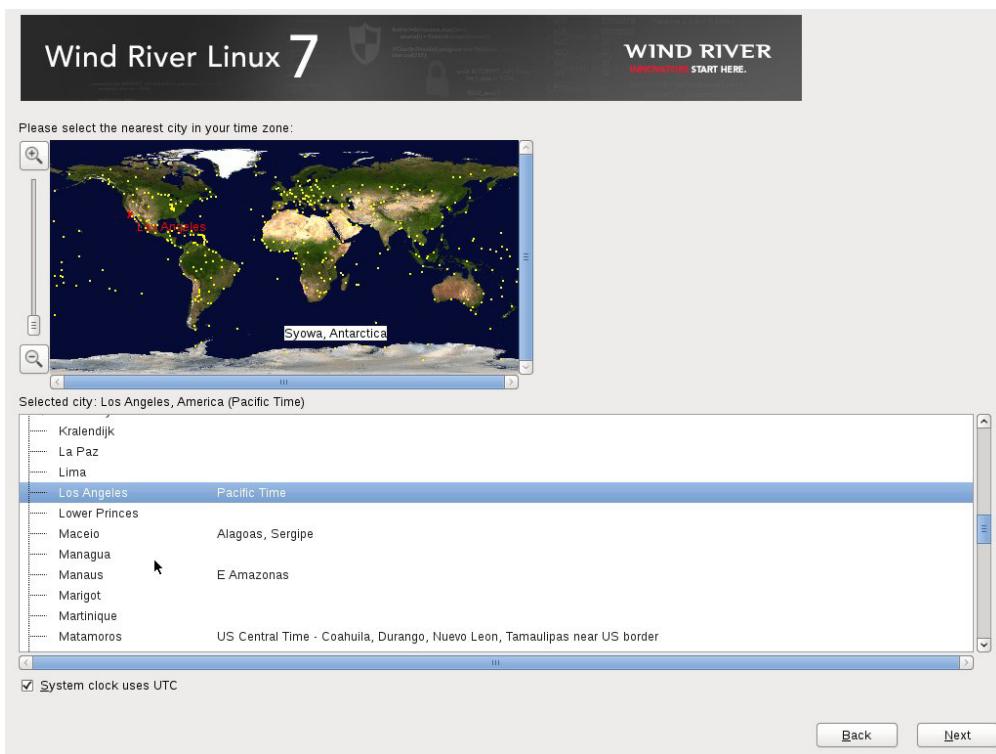


WARNING: This will delete any data on the disk. To keep existing data, select **No, keep my data**. You are prompted to select a location on the disk to perform the installation.

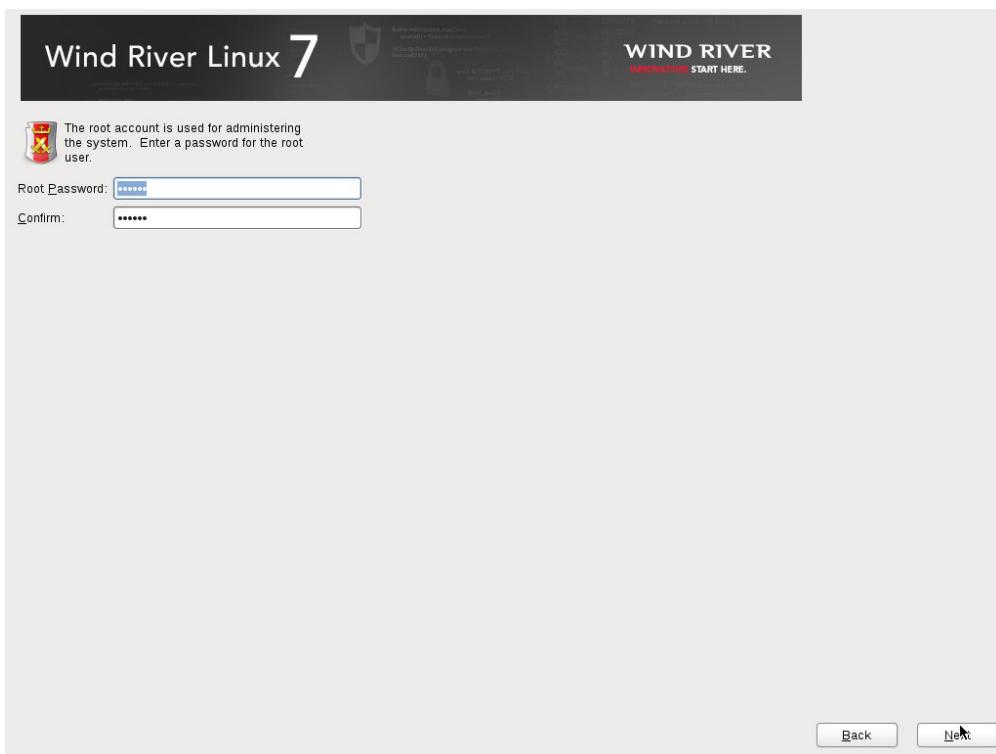
Step 3 Enter a hostname, then click **Next**.



Step 4 Select the city and timezone, then click **Next**.

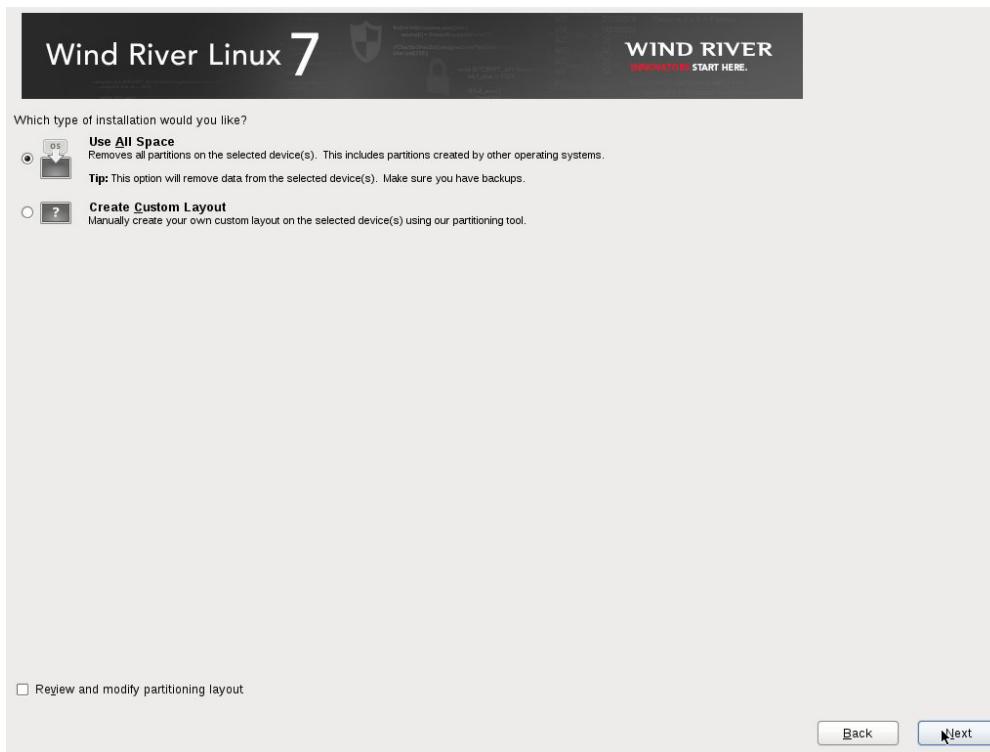


Step 5 Enter a root password, confirm the password, and then click **Next**.



NOTE: The root password must be at least six characters long.

Step 6 Select the type of installation, then click **Next**.



- **Use all space** automatically formats disk partitions.
- **Create custom layout** allows you to specify disk partitions.

Step 7 Select **Write Changes to Disk** to format the disk partitions.

Once selected, you cannot go back to make changes without exiting the installation and starting over.



When formatting completes, you are provided with a list of installation choices, based on your platform project installer configuration created in [Configuring and Building the Host Install](#) on page 387. If you created multiple installations as described in [Host-Based Installation of Wind River Linux Images](#) on page 384, those choices will be available.



Step 8 Select the installation from the list, and optionally, any additional packages.

In this example, select the **wrlinux-image-glibc-std-version 1.0 r5 (glibc-std)** option, created in *Configuring and Building the Host Install* on page 387.

Choose whether you want to install additional software packages:

- To perform the installation with the default packages, select **Customize later**, then click **Next**, then go to *Step 10*.
- To select additional packages to install, perform the next step.

Step 9 Optional: Specify software packages to be included in the installation.

Select **Customize now**, then click **Next**.

Choose one of the following options to add additional software.

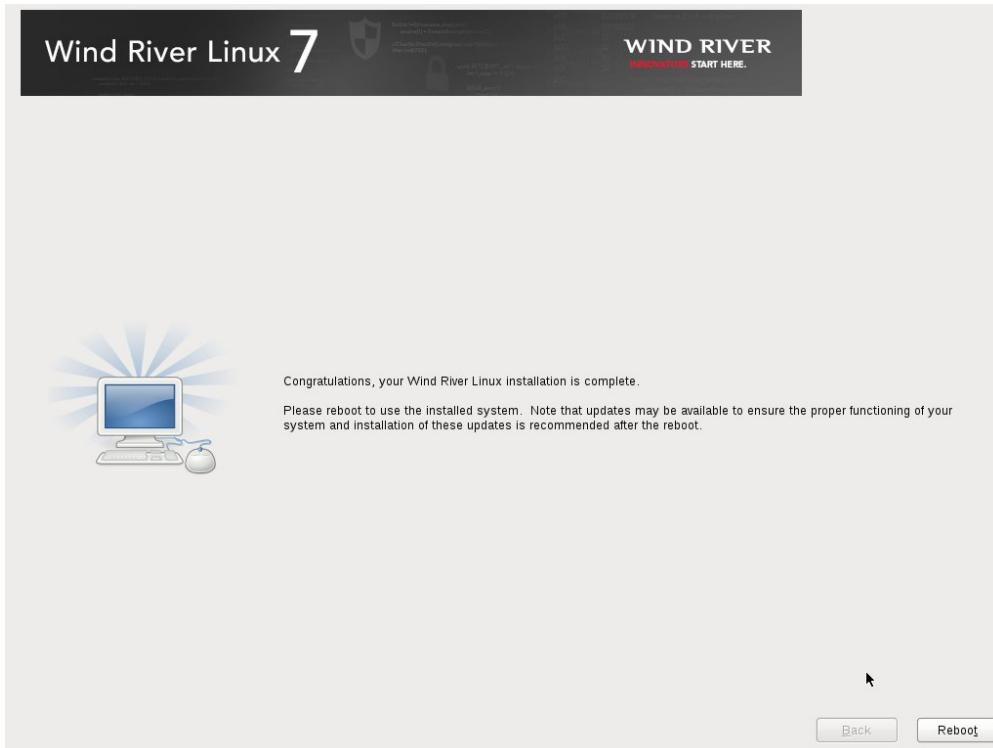
Options	Description
Select software by package group	This window displays when you select Customize now . Make selections in the top left area to choose a specific package group base. The packages included in the base selection display in the top right area, while package dependencies display in the bottom left area. You can select or deselect specific packages in the top right area. To view information on a specific package, click the package name in the top right area and view the information in the bottom right area.
Select software by architecture	1. Select Category by arch to display package options by architecture. 2. Select the architecture in the top left area to display the available packages in the top right area. You can select or deselect specific packages in the top right area. To view information on a specific package, click the package name in the top right area and view the information in the bottom right area.

Once you are finished with package selection, click **Next** to begin the installation. The installer will perform a verification, and add any additional package selections.

Step 10 Install Wind River Linux and any additional package selections.

The installation will begin automatically. A status bar displays progress as the distribution is installed to disk. This may take some time depending on the hardware you are installing to.

Once installation completes, you see a confirmation window.



Step 11 Restart the installed system.

Options	Description
QEMU virtual disk	<ol style="list-style-type: none">1. Close the QEMU window.2. Boot from the disk that you installed Wind River Linux on in <i>Booting and Installing with QEMU</i> on page 389. <pre>\$ make start-target TOPTS="" -no-kernel -disk hd0.vdisk -gc"</pre> <p>Do not enter the -gc option if you are using the serial console.</p>
Physical disk media	<ol style="list-style-type: none">1. Remove the installation media.2. Click Reboot.

Installing with the Serial Console Installer

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 389.

This procedure assumes that you have previously created an installable **.iso** disk image as described in *Creating Bootable USB Images* on page 394 or *About Manually Configuring a Boot Disk with Files from a USB/ISO Image*.

Step 1 Install the host image on the target machine.

- Select the installer boot option.

Once the target machine boots, you will be presented with two options to boot from, the **Graphics console boot** and the **Serial console boot**. Usually you choose the former which gives you control over the installation process directly from the keyboard attached to the target machine. The serial console boot option is useful if you need to run the installer over the serial port.

- Select the installation options.

The installer will present you with a text-based interface and ask you to accept or specify your installation options with a series of questions.

```
What disk do you want to format? (sda sdb) [sda]
```

In this example, the drive **sda** is selected to host the image. This may be different for your particular configuration.

NOTE: If you are building the installer for an OVP guest image, you must use **vda** as the name of the drive, even if this name is not presented as an option.

```
Delete existing partitions and ALL data (yes/no) [no] yes
```

This question asks if you want to delete all existing partitions and use the entire hard drive for the installation. You must answer **yes** to this question, or the installer script will quit and

drop the console to the command prompt. This question is not asked if no partitions are found in the hard drive.

```
How many MB for the /boot partition [500] 200
```

This question lets you specify the size of the **/boot** partition. A minimum of 200 MB should be used.

```
How many MB for the swap partition [3716]
```

The installer suggests a size for the swap partition based on the amount of memory available in the target machine. This value is just a suggestion and is usually safe to leave it as is. Press **ENTER** to accept the default, or provide a new value.

```
How many MB for the root partition (-1 == rest of disk) [-1]
```

The installer asks for the size of the root partition. The default answer of -1 to include the rest of the hard drive is usually adequate.

Once the installation process is complete, the hard drive is fully initialized with three partitions: **swap**, **boot**, and **filesystem**. The hard drive is ready to be used as the boot drive.

- c) Power off the target host system.

Enter **halt -p** at the command prompt or turn the system off using the power switch.



NOTE: The file system in the USB storage device is read only, so there is no harm in directly powering off the target machine.

- d) 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 389, just close the graphics window (or press **CTRL-A, X** if you are using the serial console).

Step 2 Boot the target machine from the new hard drive installation.

It may be necessary to configure the host system's BIOS or UEFI to boot from the hard drive.

Once the target machine finishes booting, you will be presented with the console prompt.

Step 3 Log into the host platform.

Use the user name **root**, with password **root** to log into the system.

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 functionally.

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-3.14.13-yocto-standard-3.14.13+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-3.14.13-  
yocto-standard  
-rw-r--r--   1 root      root      5601808 Sep 18 06:45 bzImage-3.14.13-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-3.14.13-yocto-standard-3.14.13+gitAUTOINC  
+285f93bf94_702040ac7c" {  
    set root=(hd0,1)  
    linux /bzImage-3.14.13-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-3.14.13-yocto-standard-3.14.13+gitAUTOINC+285f93bf94_702040ac7c
```

About Manually Configuring a Boot Disk with Files from a USB/ISO Image

While Wind River Linux provides options for creating an installable image, you may want to use a disk image to provide the root filesystem and kernel bzImage to create a boot disk from.

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 383.

Preliminaries

The following example shows how to use Wind River Linux to configure a common PC to boot Wind River Linux from hard disk, using the root filesystem and kernel bzImage files created when you create and build a Wind River Linux platform project. With this method you do not use the server installer and must configure the hard drive manually.

This requires a previously built platform project with the `--enable-bootimage=iso` `configure` option as described in [Configuring and Building the Host Install](#) on page 387.

If you want to automate this procedure, see [Host-Based Installation of Wind River Linux Images](#) on page 384.

You can boot and configure your target using either an ISO or USB image as described in this section.

Prepare the Target's Hard Drive

Once you have an .iso image with the root file system and kernel images, in order to boot a system using the images, you must partition the hard drive using a bootable CD-ROM or USB device.

These instructions describes how to boot the target and prepare the hard drive using the bootable CD-ROM or USB device you created in [Configuring and Building the Host Install](#) on page 387

Alternatively, you can use some other bootable tool such as the freely-available *Gparted-LiveCD* or *Partition Magic*. All of these allow you to boot the target and then partition and format the hard

disk on the target. You will then have to transfer the target file system (**export/*dist.tar.bz2**) and the kernel (**export/*bzImage***) from the host using some other method because it will not already be on your boot media in the / directory. You may be able to transfer these using an Ethernet connection or by writing them to another CD-ROM or a USB device.

The following procedure assumes that the target has one IDE hard drive, and that it is unpartitioned. The example used is a 30 GB hard drive, to be partitioned with one Linux partition and a swap partition. The particular values you see displayed will differ from those shown here depending on the size of the disk partitions you are creating.



WARNING: Any pre-existing data on the hard drive of the target will be lost when you perform this procedure.

Step 1 Use fdisk to partition the hard drive on the target.

- a) Boot the target from the CD-ROM or USB device.
- b) Enter the **fdisk** command with the device name of the drive you are going to format.

Although the following example does not show it, you may want to create a separate small partition for the grub boot loader (256 MB or less). This is especially useful if you plan to change the root file system or kernel frequently, thus saving the boot sector on the hard disk from frequent rewriting.

In the following example, the hard drive on the target is device **/dev/sda**—your configuration may differ.

At the console on the target, enter the following::

```
root@localhost:/root> fdisk /dev/sda
```

- c) Examine your current partition table.

Use the **fdisk p** command.

```
Command (m for help): p
Disk /dev/sda: 30.0 GB, 30005821440 bytes
16 heads, 63 sectors/track, 58140 cylinders
Units = cylinders of 1008 * 512 = 516096 bytes

Device Boot      Start        End      Blocks   Id  System

```

Command (m for help):

If your disk is already partitioned, in **fdisk** delete the existing partitions with the **d** command.

In the preceding example, there are no existing partitions.

- d) Enter **n** to create a new partition:

```
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
```

- e) Enter **p** to create a primary partition

```
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
```

- f) Enter **1** to create partition **1** (**/dev/sda1**).

```
Partition number (1-4): 1
First cylinder (1-58140, default 1):
```

- g) Enter a number of cylinders for the size of your primary partition.

Since you are only creating one partition, this is the majority of the disk space and the remainder is used for swap space.

```
First cylinder (1-58140, default 1): ENTER
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-58140, default 58140): 50000
Command (m for help):
```

- h) Use the **a** command to toggle the **bootable** flag on the partition **/dev/sda1**.

- i) Create a second partition (**/dev/sda2**) to use as swap space:

```
Command (m for help): n
Command action
  e   extended
  p   primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (50001-58140, default 50001): ENTER
Using default value 50001
Last cylinder or +size or +sizeM or +sizeK (50001-58140, default 58140): ENTER
Using default value 58140

Command (m for help):
```

- j) Change the type of the second partition to swap space (type 82).

```
Command (m for help): t
Partition number (1-4): 2
Hex code (type L to list codes): 82
Changed system type of partition 2 to 82 (Linux swap / Solaris)

Command (m for help):
```

- k) Write your new partition table to disk.

```
w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: Re-reading the partition table failed with error 16: Device or resource
busy.
The kernel still uses the old table.
The new table will be used at the next reboot.
Syncing disks.
```

- l) Leave the CD-ROM or USB device in the drive and reboot the target.

```
# reboot
```

Reboot from the CD-ROM or USB device just as you did before.



NOTE: If the **partprobe** command is available on the Live CD/USB image, the reboot can be avoided by running **partprobe** to make the new partitions available for the next steps.

Step 2 Format and mount the hard drive and swap.

- a) Format the main Linux partition.

```
# mkfs.ext3 -I 128 /dev/sda1
```

- b) Initialize the newly created swap partition.

```
$ mkswap /dev/sda2
```

- c) Create a temporary mount point for the hard disk.

```
# -p /tmp/mnt/sda1
```

- d) Mount the main Linux partition.

```
# /dev/sda1 /tmp/mnt/sda1
```

Place the File System and Kernel on the Hard Disk

To create a bootable target disk manually, once you have formatted the disk, you must copy the file system and kernel to it.

You can now install the file system and place the kernel in the installed file system.

- Step 1** Change directory to the hard disk root and uncompress and extract the file system from the boot device root directory.

This example specifies a file path of `/tmp/mnt/sda1/`. Use the correct value for your system.

```
# cd /tmp/mnt/sda1/  
# tar jxvpf /*dist.tar.bz2
```

- Step 2** Copy the kernel from the device to the boot directory of the hard disk.

```
# cp /*bzImage* /tmp/mnt/sda1/boot/bzImage
```

Note that this example shortens the name to **bzImage** for convenience.

You now have the file system and kernel in place. In the next section you configure the system to boot.

Configure the Target System Files and Boot

Perform the procedure in this section to set up your file system mount table and GRUB boot menu.

Set up your file system mount table (**fstab**) and your GRUB boot menu (**menu.lst**) as described in this section.

NOTE: Note that the following procedure assumes `/dev/sda` is the hard drive: Your configuration may differ.

- Step 1** Edit the `/etc/fstab` File.

Edit `/tmp/mnt/sda1/etc/fstab` to look something like the following:

```
proc /proc proc defaults 0 0 # AutoUpdate  
sysfs /sys sysfs defaults 0 0 # AutoUpdate
```

```
devpts /dev/pts devpts defaults 0 0 # AutoUpdate
relayfs /mnt/relay relayfs defaults 0 0 # AutoUpdate
tmpfs /dev/shm tmpfs defaults 0 0 # AutoUpdate
/dev/hdc /mnt/hdc_cdrom iso9660 noauto,users,exec 0 0 # AutoUpdate
/dev/sda1 / ext3 auto,users,suid,dev,exec 0 0 # AutoUpdate
/dev/sda2 swap swap defaults 0 0 # AutoUpdate
/dev/fd0 /mnt/floppy vfat,msdos noauto,users,suid,dev,exec 0 0 # AutoUpdate
```

Use values appropriate for your target. For example, remove the **/dev/fd0** entry if your target does not have a floppy drive.

Step 2 Install GRUB to the Master Boot Record (MBR).

- Start GRUB.

```
# grub
grub>
```

- Set the root device, and then install GRUB to the MBR.

One way to install GRUB to the MBR is as follows:

```
grub> root (hd0,0)
grub> setup (hd0)
grub> quit
```

Alternatively, you can install GRUB as follows:

```
grub> root (hd0,0)
# /tmp/mnt/sda1/boot/grub
g# # grub-install --root-directory=/tmp/mnt/sda1 /dev/sda
```

Add the **--no-floppy** option if you do not have a floppy drive on your target.

Step 3 Configure GRUB.

- Backup the default **boot/grub/menu.lst** file.

The **orig_menu.lst** file contains useful instructions that can help you understand the GRUB menu entries. It also shows you how to set up a system for dual- or multi-booting if you want to configure target disks that way in the future.

```
# cd /tmp/mnt/sda1/boot/grub
# mv menu.lst orig_menu.lst
```

- Use a text editor to create a new **menu.lst** file that contains the following:

```
default 0
timeout 5
title my Common PC
root (hd0,0)
kernel (hd0,0)/boot/bzImage root=/dev/sda1 rw
boot /boot/bzImage
```

Save the edited file as

menu.lst

Step 4 Reboot, removing the USB or CD-ROM device so that the target reboots from hard disk. (The system may need to be rebooting before you can remove the device.)

Step 5 Log in as user **root**, password **root**.

About Deploying an Image with a Virtual Machine Manager

Wind River Linux provides the **--enable-bootimage=vmdk** **configure** option to create bootable images for use with virtual machine (VM) managers, such as VMWare and VirtualBox.

VM managers support the **vmdk** image type, which you can create as part of your platform project image build. The **--enable-bootimage=vmdk** **configure** option adds the following line to the **projectDir/local.conf** file:

```
IMAGE_FSTYPES += "vmdk"
```

For existing platform projects, you can edit this file directly. Once you run the **make** command to build, or rebuild, the platform project, the **vmdk** image is located in the **projectDir/export/images** directory, for example:

projectDir/export/images/wrlinux-image-glibc-small-intel-x86-64-20140924194516

Note that the file name will include the file system type and BSP, along with a date and time stamp. You will use this file as the image to boot on your preferred VM manager.

For additional information, see [About Configuring and Building Bootable Targets](#) on page 383.

Using VMware with vmdk Boot Images

Learn how to deploy a **vmdk** image on a VMware client.

Some general guidelines for using a VMware client include:

Guest Operating System

Set to **Linux**.

Version

Set to **Other Linux 3.x kernel**, either 32- or 64-bit, depending on your platform project BSP selection.

Hard disk setting

Use a single SATA type hard disk that references your **vmdk** image.

Other settings

Settings such as memory, network connections, and number of processing cores depend on your host capabilities and testing requirement..

The following procedure shows how to deploy a **vmdk** platform project image on a VMware Workstation (build 10.0.3 build-1895310), on a Windows 7, 64 bit machine.

Step 1 Copy the bootable **vmdk** platform project image to the Windows host, where the VMware Workstation client is installed.

Step 2 Create the new VM.

- a) In the VMware Workstation client, select **File > New > Virtual Machine**.
- b) Select **Custom (Advanced)**.
- c) Enter **Next** as the **Type** and click **Next**.
- d) Select **I will install the operating system later**.
- e) Click **Next**.

- f) Select **Guest operating system** to Linux and **version** to other Linux 3.x kernel, 32-bit or 64-bit, depending on your platform project image BSP selection. Click **Next**.
- g) Enter a name for the virtual machine, then click **Next**.
- h) Set the **Processor Configuration** input setting to the number your processor supports, then click **Next**.

Step 3 Configure the memory size, network settings, and I/O controller type for the VM.

- a) **Memory size**: Set 256 MB or higher according to your host and target requirements. Click **Next**.
- b) Select any of the available choices as your **Network connection** setting and then click **Next**.
- c) Select **LSI Logic (Recommended)** as the I/O controller type. Click **Next**, then select **SATA** and click **Next**.

NOTE: The vmdk image must be used as SATA drive.

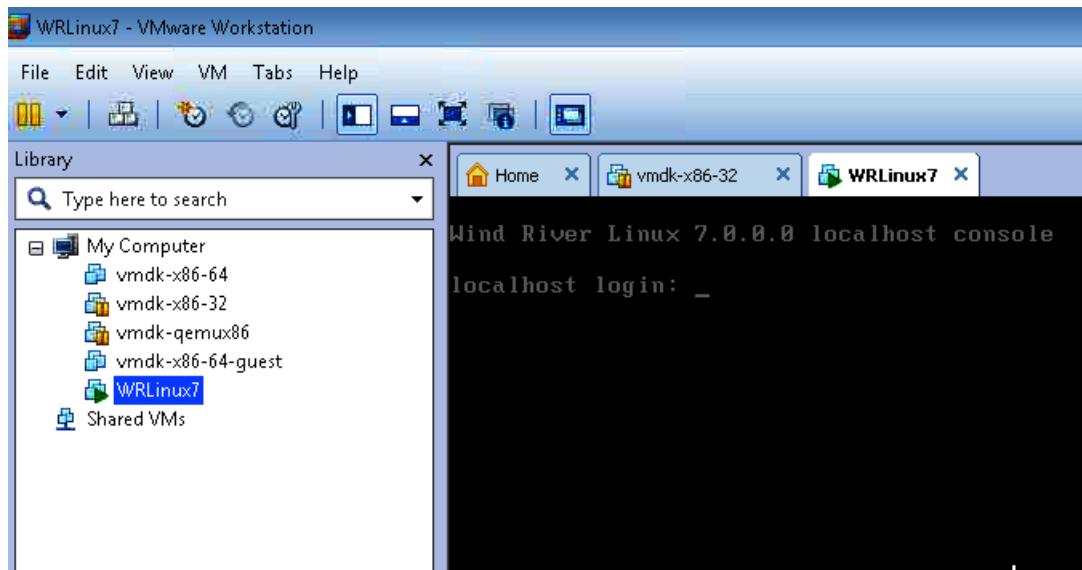
Step 4 Select a disk.

- a) Optional: Select **Use an existing virtual disk** and click **Next**.
- b) Input the vmdk filepath from the **Browse** window and select the vmdk image. Click **Next**.
- c) Select **Keep existing format** and click **Finish**.

Step 5 Boot the VM.

Click **Start**.

The **vmdk** image boots the target in VMware as shown.



Related Links

[About Configuring and Building Bootable Targets](#) on page 383

To create a single, bootable image from which to boot a target device, specify the **enable-bootimage** option.

[Configure Options Reference](#) on page 82

Many options to the **configure** script are available to customize your Wind River Linux projects.

[About the make Command](#) on page 98

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

Using VirtualBox with vmdk Boot Images

Learn how to deploy a **vmdk** image on a VirtualBox client.

The following procedure shows how to deploy a **vmdk** platform project image on a VirtualBox 4.3.12 client, on an Ubuntu 14.04 32-bit host.

Prior to installing and using VirtualBox with a **vmdk** image, the system BIOS must be set to enable **Virtualization and VT for Direct I/O** or its equivalent, otherwise the **vmdk** image will not be able to boot.

Step 1 Copy the bootable **vmdk** platform project image to the Ubuntu host, where the VirtualBox client is installed.

Step 2 Create the new VM.

- a) Click the **New** button on VirtualBox.
- b) Enter a name for the VM in the **Name** field.
- c) Enter **Linux** as the **Type** and click **Next**.
- d) Enter **Linux 3.x**, then select 32-bit or 64-bit, depending on your platform project image BSP selection as the **Version**.
- e) Click **Create**.

Step 3 Configure the memory size, hard disk, and configurations settings for the VM.

- a) **Memory size:** Set 256 MB or higher according to your host and target requirements, then click **Next**.
- b) **Hard drive:** Select **Use an existing virtual hard drive file**, then select the **vmdk** file. Click **Create**.

NOTE: The **vmdk** image must be used as a SATA drive. Verify it after the VM is created by clicking on **Settings > Storage > Controller: SATA**.

The **vmdk** image must located here.

Step 4 Optional: Set the configuration settings for the VM.

NOTE: Only the 32 bit image requires this step. It is optional for the 64-bit image.

- a) Optional: Click **Settings > System > Processor**.
- b) Select **Extended Features: Enable PAE/NX**.
- c) Click **OK**.

Step 5 Boot the VM.

The **vmdk** image boots the target in VirtualBox.

Related Links

[About Configuring and Building Bootable Targets](#) on page 383

To create a single, bootable image from which to boot a target device, specify the **enable-bootimage** option.

[Configure Options Reference](#) on page 82

Many options to the **configure** script are available to customize your Wind River Linux projects.

About the make Command on page 98

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

Deploying *initramfs* System Images

[About *initramfs* System Images](#) 421

[Creating *initramfs* Images](#) 422

[Adding Packages and Kernel Modules to *initramfs* Images](#) 423

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/layers/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 422.

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 421.

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 and Kernel Modules to initramfs Images

Once you create an initramfs image, you may need to add packages or kernel modules to it to meet your specific development requirements.

Use the procedures in this section to add packages or kernel modules to an initramfs image created using the procedures in [Creating initramfs Images](#) on page 422. For information on initramfs images in general, see: [About initramfs System Images](#) on page 421.

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

In this example, the platform project configured and built using the instructions in [Creating initramfs Images](#) on page 422, is used, although you can add this information to any `.conf` file in a valid project layer.

Step 2 Add the package inclusion information to the file.

For each package you want to add, add the following line to the file:

```
IMAGE_INSTALL_append_pn-wrlinux-image-initramfs += " packageName1"  
IMAGE_INSTALL_append_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.

To add a kernel module, add the following line:

```
IMAGE_INSTALL_INITRAMFS_append += "kernel-module-moduleName1"  
IMAGE_INSTALL_INITRAMFS_append += "kernel-module-moduleName2"
```

Step 3 Save the file once you are finished.

Step 4 Rebuild the platform project.

Run the following command from the project directory.

```
$ 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 422.

Deploying KVM System Images

[**About Creating and Deploying KVM Guest Images**](#) 425

[**Create the Host and Guest Systems**](#) 427

[**Deploying a KVM Host and Guest**](#) 428

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-x86-64 \
--with-template=feature/kvm
```

The configure command may take a couple moments to complete.

Step 2 Install the **netperf** package.

- Obtain the **netperf** package (**netperf-2.6.0.tar.bz2**).

This package is available online from <http://www.netperf.org>.

- Place the **netperf** package in the **projectDir/layers/local/download** directory.
- Update the **projectDir/local.conf** file, **LICENSE_FLAGS_WHITELIST** option to read:

```
LICENSE_FLAGS_WHITELIST += "non-commercial"
```

- Build and add the package.

```
$ make netperf.addpkg
```

Step 3 Add the **bridge-util** package.

```
$ make bridge-utils.addpkg
```

Step 4 Add the **wrs-kvm-helper** package.

```
$ make wrs-kvm-helper.addpkg
```

Step 5 Build the KVM host platform project image.

```
$ make
```

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 netperf.addpkg
```

Step 8 Build the KVM guest platform project image.

```
$ make
```

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 427.

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@qemu0 # ip link set macvtap0 address 1a:46:0b:ca:bc:7b up
root@qemu0 # 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 \
-drive file=usb.img \
-net nic,macaddr=1a:46:0b:ca:bc:7d,model=virtio \
-net tap,fd=3 3<>$TAPDEV
```

In this example, \$TAPDEV refers to /dev/tap.

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.

PART VIII

Testing

Running Linux Standard Base (LSB) Tests.....	433
Validating Platform Project Images with ptest.....	441
Running Open POSIX Tests.....	451

Running Linux Standard Base (LSB) Tests

About the LSB Tests	433
Testing LSB on Previously Configured and Built Target Platforms	434
Disabling Grsecurity Kernel Configurations on CGL Kernels	435
Running LSB Distribution Tests	435
Running LSB Application Tests	437

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 437.

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 binary 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 7.0 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 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 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 433.

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:

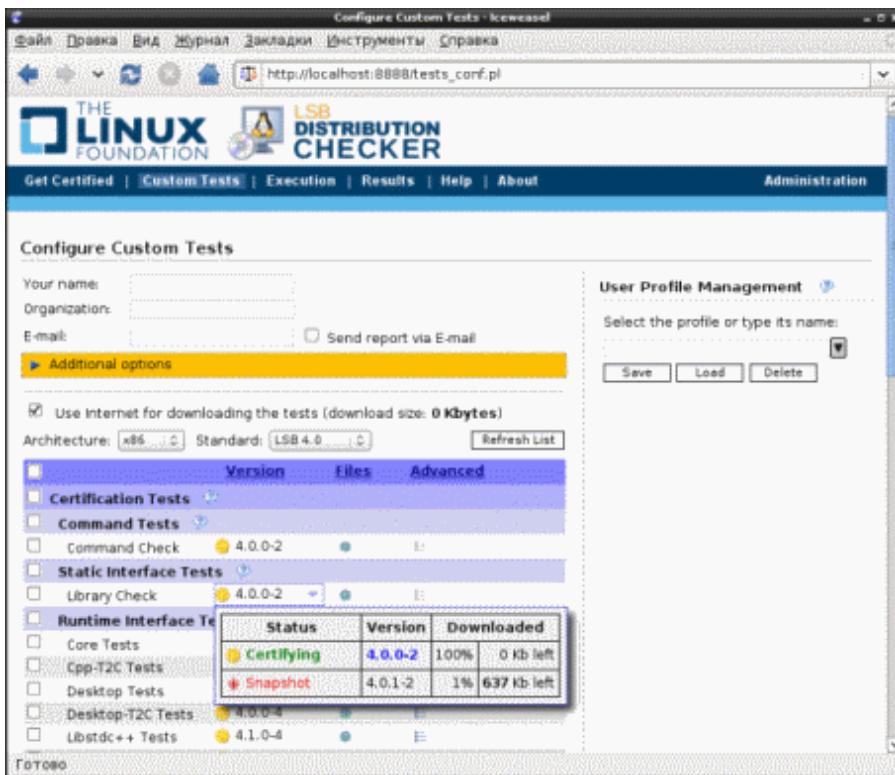
```
root@target~# sh /usr/bin/LSB_Test.sh
```

The script will begin downloading the required files, and will start the tests once complete.

Step 4 Visit the test interface.

Connect to the test interface at **http://localhost:8888** on the local machine, or at **http://targetIP:8888** from a remote machine with network access.

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/lsb>.

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 433. 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 in the main directory of the LSB Application Checker installation. 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**.

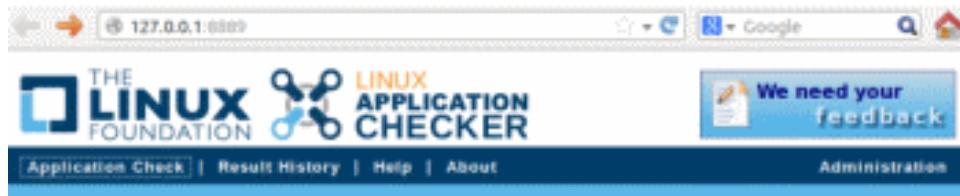
Step 4 Start the test interface.

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.



Welcome to the Linux Application Checker!

This tool enables you to analyze the compatibility of your application with various Linux distributions and the LSB standard by checking external dependencies (libraries and interfaces) required to run your application.

- » To specify your application for testing and run the analysis, please visit the [Application Check](#) page.
- » The results of previous test runs can be found on the [Results History](#) page.
- » If you need more information about the Linux Application Checker, please read the [Help](#) and [Getting Started](#) pages.

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.

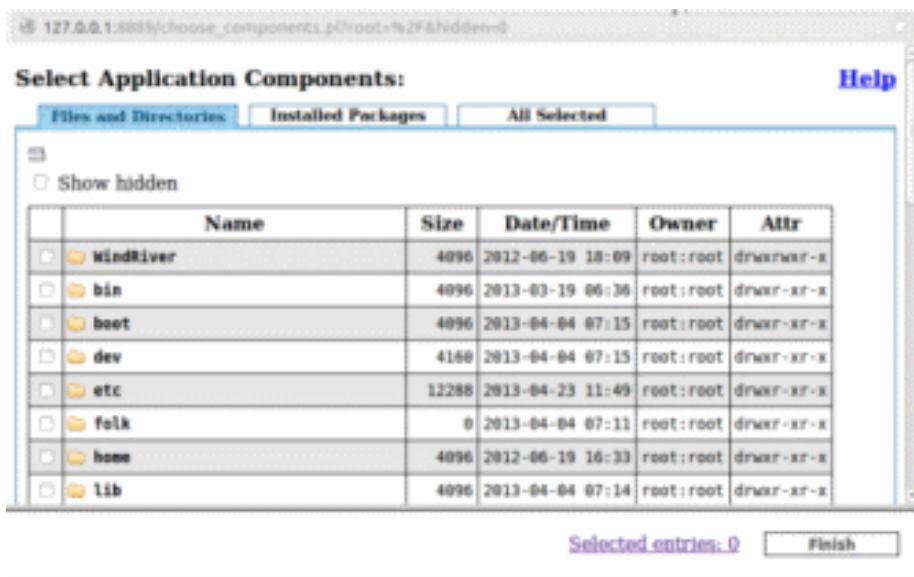
The Specify Your Application page loads.



Step 6 Enter application-specific information for the application you want to test.

- In the Name field, enter a name for your test report.
- In the Components field, enter the file path to the application, for example: **/usr/bin/hello**
- Click **Select Application Components** to specify the components associated with the application. This includes:
 - Individual files
 - Whole directories
 - Installed RPM packages (prepended 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. 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.

Validating Platform Project Images with ptest

Testing With ptest	441
Building a Platform Project and Running ptest to Collect Results	442
Running ptest-diff.sh	446
Running ptest-summary.sh	447
Processed ptest-run Log File Example Results	448
ptest-summary.sh and ptest-diff.sh Option Reference	450

Testing With ptest

Wind River Linux includes a validated version of **ptest** to use as a basis for validating the packages that comprise your platform project image.

Overview

Ptest refers to the package test functionality available in the Yocto build system. **Ptest** helps you build, install, and run test suites that are already present in many packages, and also enables the native tests supplied in many packages for use in a cross-build environment. Like any good top-level test harness, **ptest** consolidates each package's tests into a consistent format: PASS, FAIL or SKIP, the same format used by GNU **Automake**.

Including **ptest** in your platform project configuration enables all validated **ptest** recipes in the packages included with Wind River Linux. In addition, any third-party or Yocto packages with **ptest** build support will be built and run. Enabling **ptest** adds supporting packages to both your host and target image. For glibc_small file system images, this footprint increase may be significant and exceed on-target file-system resources, so an NFS mount must be used to accommodate the harness for testing purposes.

Extending ptest Support

Currently, only a small subset of the available packages provide **ptest** build support, although you can extend this support to additional packages in your platform project image. This is described at the Yocto Project website at: <https://wiki.yoctoproject.org/wiki/Ptest>.

The procedure for adding **ptest** support to package can be summarized as writing a BitBake recipe that breaks the existing tests into build and execute stages appropriate for a cross build environment.

ptest Collection and Comparison of Test Results Tools

The **ptest-diff.sh** tool compares the results of different **ptest** runs. The results from both development and runtime environment can be evaluated. The **ptest** results can be shown for the same runs for different BSPs, and for different runs for the same BSP. See [Running ptest-diff.sh](#) on page 446.

The **ptest-summary.sh** tool collects the results of different **ptest** runs. The results from both development and runtime environment can be evaluated. The **ptest** results can be shown for the same runs for different BSPs, and for different runs for the same BSP. See [Running ptest-summary.sh](#) on page 447.

Building a Platform Project and Running ptest to Collect Results

Configure a platform project with **ptest** support and run the test suite on a simulated or hardware target to collect test results.

The following procedure enables the Yocto **ptest** infrastructure within your platform project, including built-in **ptest** support for packages. There are several alternatives for running **ptest**. For example, if you run **ptest** from:

- The platform project directory:

```
$ make start-ptest
```

The **ptest** log file is written to the platform project directory.

- From a ssh shell:

```
$ ssh root@target_ip ptest-runner
```

The **ptest** log file is written to the host directory where you ran the command

- If you run the command on the target:

```
# ptest-runner
```

The **ptest** log file is written to the target directory where **ptest-runner** ran.

The following procedure shows how to use **ptest-diff.sh** to collect results.



NOTE: ptest scans can take long time to run on a slow target. For example, it takes more than 5 hours to finish all the ptests on **qemuarm9**. It is recommended that you allocate more memory to QEMU and enable the **kvm** feature for qemux86 and qemux86-64, which can speed up testing.

To boot qemu with 2GB of memory, run the following command:

```
$ make start-target TOPTS="-m 2048"
```

To boot qemux86 and qemux86-64 with 2GB of memory and **kvm** enabled, run the following command:

```
$ sudo make start-target TOPTS="-m 2048" TARGET_QEMU_OPTS='--enable-kvm'
```

Step 1 Create a platform project directory and navigate to it.

For example:

```
$ mkdir qemux86-64-glibc-std-ptest  
$ cd qemux86-64-glibc-std-ptest
```

Step 2 Configure a platform project using the **ptest** feature template.

You can use any one of the following configure script options to add **ptest** functionality:

- **--enable-test**
- **--with-template=feature/test**
- **--with-template=feature/ptest**

The following configure script example uses the **--enable-test** option:

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

Step 3 Build the file system.

```
$ make
```

This command can take some time to complete, depending on your host system configuration.

Step 4 Boot your target and run **ptest**.

Options	Description
QEMU simulator target	<p>Run the following command from the platform project directory:</p> <pre>\$ make start-ptest</pre> <p>This build target automatically starts the tests inside an ssh session once the target boots. The ssh session invokes the ptest-runner script on the target and logs the results to the root of the project directory on the development host.</p> <hr/> <p>NOTE: Run the following command if you want to have the log file saved on the QEMU target:</p> <pre>\$ make start-target</pre> <p>After logging in to the target, run the following command:</p> <pre># ptest-runner</pre> <p>The ptest log file is saved on the QEMU target. The ptest-runner only prints test results to the console, so it is necessary to save the output in a log file with meaningful name, such as qemux86-64-ptest-wrl7.log. To produce this log file, you could run:</p> <pre># ptest-runner tee qemux86-64-ptest-wrl7.log</pre> <p>See Running <i>ptest-diff.sh</i> on page 446 for collecting results.</p>

Options	Description
Hardware target	<ol style="list-style-type: none">1. Deploy the platform project file system and kernel and boot the target.2. On the development host, navigate to the platform project directory and run the ptest-runner script to start the test harness. The ptest-runner script can be started either from any login console or an ssh session, for example: <pre>\$ ssh root@target_ip ptest-runner tee ptest-run-`date +%Y-%m-%dT%H:%M`.log</pre>In this example, you would substitute <i>target_ip</i> for the IP address on the target. <p>NOTE: The command shown above can be used on both QEMU and hardware targets.</p> <p>For a QEMU target, a simple way to run ptest is to run the command from the build directory:</p> <pre>\$ make start-ptest</pre> This command boots the QEMU target and runs ptest automatically. The ptest results will be saved in a log file named using the same pattern as ptest-run-2014-07-25T09:01.log

Step 5 View the test results.

From the platform project directory, run the following command:

```
$ cat ptest-run-date-time.log
```

In this example, substitute *date-time* for the day and time the test was run, for example:

```
$ cat ptest-run-2014-08-13T10:05.log
```

Step 6 Optionally process the test results into a more readable format.

Wind River provides a script to process the **ptest** log file and provide specific details on the packages that were tested and their results. Run the following script from the platform project directory, including the **ptest** log file as the target.

```
$ layers/wr-base/recipes-support/ptest-runner/files/ptest-summary.sh -ds -l ./ptest-run-2014-08-13T10:05.log
```

 **NOTE:** If the ptest log is saved on the target (QEMU or hardware target), then the **ptest-summary.sh** tool should be run on the target directly. See [Running ptest-summary.sh](#) on page 447.

For an example of the formatted output, see [Processed ptest-run Log File Example Results](#) on page 448.

Once you have completed this procedure, you may want to extend **ptest** support to additional key packages in your platform project image. This is described at the Yocto Project website at: <https://wiki.yoctoproject.org/wiki/Ptest>.

NOTE: It is common for open source packages to be released with known test case failures. Wind River is not responsible for fixing errors in open source packages.

Running ptest-diff.sh

ptest-diff.sh adds comparison options as part of the standard ptest testing harness.

The **ptest-diff.sh** command has several options. See *ptest-summary.sh and ptest-diff.sh Option Reference* on page 450.

Usage: **/usr/bin/ptest-diff.sh option FILE1 FILE2** where **FILE1** is the basic ptest log for comparison and **FILE2** is the new ptest log file to be checked.

Step 1 Run the **ptest** command as part of target deployment.

See *Building a Platform Project and Running ptest to Collect Results* on page 442.

For a QEMU target, a simple way to run ptest is to run the command from the build directory:

```
$ make start-ptest
```

The command boots the QEMU target and runs ptest automatically. The ptest results are saved in a log file named something like: **ptest-run-2014-07-25T09:01.log**. It is recommended to rename the ptest log to something more meaningful.

Step 2 Compare ptest results from different runs.

The full output is saved in the log file **ptest-diff-output.log**. The following example runs a ptest log file against **wrl6**

```
# ptest-diff.sh gemux86-64-ptest-wrl6.log \
> gemux86-64-ptest-wrl7.log
```

The results output will be similar to the following:

```
---- The ptests previously are FAILED or SKIPPED, currently are PASSED ----
    glib-2.0/ptest
    kmod/ptest
---- The ptests previously are PASSED or FAILED, currently are SKIPPED ----
    perl/ptest
    strace/ptest
---- The ptests which are missing ----
    dbus-ptest/ptest
    rsyslog/ptest
---- The new ptests which are PASSED ----
    beecrypt/ptest
    diffutils/ptest
    flex/ptest
    gdbm/ptest
    libpcre/ptest
    parted/ptest
    zlib/ptest
---- The new ptests which are FAILED ----
    acl/ptest
    attr/ptest
    gawk/ptest
    openssh/ptest
    openssl/ptest
    sed/ptest
```

```
tcl/ptest  
udev/ptest
```

Step 3 Optional: Run the following command to review test case failures in a failed pteset:

```
# ptest-diff.sh -v qemux86-64-ptest-wrl6.log \  
> qemux86-64-ptest-wrl7.log -d
```

The results output will be similar to the following output example:

```
Base ptest log file: qemux86-64-ptest-wrl6.log  
Processing ptest log file: qemux86-64-ptest-wrl7.log  
---- The ptests previously are FAILED or SKIPPED, currently are PASSED ----  
    glib-2.0/ptest  
    kmod/ptest  
---- The ptests previously are PASSED or SKIPPED, currently are FAILED ----  
---- The ptests previously are PASSED or FAILED, currently are SKIPPED ----  
    perl/ptest  
    strace/ptest  
---- The ptests which are missing ----  
    dbus-ptest/ptest  
    rsyslog/ptest  
---- The new ptests which are PASSED ----  
    beerypt/ptest  
    diffutils/ptest  
    flex/ptest  
    gdbm/ptest  
    libpcre/ptest  
    parted/ptest  
    zlib/ptest  
---- The new ptests which are FAILED ----  
    acl/ptest  
    attr/ptest  
    gawk/ptest  
    openssh/ptest  
    openssl/ptest  
    sed/ptest  
    tcl/ptest  
    udev/ptest  
---- The new ptests which are SKIPPED ----
```

Running **ptest-summary.sh**

You can run the **ptest-summary.sh** command with several options. See [ptest-summary.sh and ptest-diff.sh Option Reference](#) on page 450.

Step 1 Run the **ptest** command as part of the deployment to the target.

For a QEMU target, you can simply run ptest from the build directory:

```
$ make start-ptest
```

This command boots the QEMU target and runs ptest automatically. The ptest results are saved in a log file named something like: **ptest-run-2014-07-25T09:01.log**. It is recommended that you rename the ptest log to something more meaningful.

Step 2 Run **ptext-summary.sh** from the target to see the results summary.

To get summary statistics run the following command:

```
# ptest-summary.sh -l qemux86-64-ptest-wrl7.log -s
```



NOTE: `ptest-summary.sh` is a shell script, so it can also be run on a build server

The summary results output will be similar to the following:

```
START: /usr/bin/ptest-runner
2014-07-22T09:23 -- 2014-07-22T13:29

Test suites executed:

package: /usr/lib64/acl/ptest
2014-07-22T09:23 -- 2014-07-22T09:24
.....
Passed: 9065
Failed: 336
Skipped: 59
```

Step 3 To get the per-ptest statistics run the command with the `-d` option.

```
# ptest-summary.sh -l qemux86-64-ptest-wrl7.log -d
```

The summary results output will be similar to the following output example:

```
/usr/lib64/acl/ptest
Passed: 256
Failed: 120
Skipped: 0
.....
```

Step 4 To get the failed test cases of each ptest, run the command with the `-f` and `-d` options.

Once ptest results for both files are complete you can review the results.

```
# ptest-summary.sh -l qemux86-64-ptest-wrl7.log -f -d
```

The results output will be similar to the following output example:

```
/usr/lib64/tcl/ptest
Passed: 145
Failed: 2
FAIL: http.test
FAIL: httpold.test
Skipped: 0
...
```

Processed ptest-run Log File Example Results

View an example of a processed `ptest-run-date-time.log` file.

```
START: /usr/bin/ptest-runner
2014-07-22T09:23 -- 2014-07-22T13:29

Test suites executed:

package: /usr/lib64/acl/ptest
2014-07-22T09:23 -- 2014-07-22T09:24

package: /usr/lib64/attr/ptest
2014-07-22T09:24 -- 2014-07-22T09:24

package: /usr/lib64/bash/ptest
2014-07-22T09:24 -- 2014-07-22T09:28
```

```
package: /usr/lib64/beecrypt/ptest
2014-07-22T09:28 -- 2014-07-22T09:29

package: /usr/lib64/busybox/ptest
2014-07-22T09:29 -- 2014-07-22T09:29

package: /usr/lib64/bzip2/ptest
2014-07-22T09:29 -- 2014-07-22T09:29

package: /usr/lib64/diffutils/ptest
2014-07-22T09:29 -- 2014-07-22T09:30

package: /usr/lib64/ethtool/ptest
2014-07-22T09:30 -- 2014-07-22T09:30

package: /usr/lib64/flex/ptest
2014-07-22T09:30 -- 2014-07-22T09:30

package: /usr/lib64/gawk/ptest
2014-07-22T09:30 -- 2014-07-22T09:31

package: /usr/lib64/gdbm/ptest
2014-07-22T09:31 -- 2014-07-22T09:31

package: /usr/lib64/glib-2.0/ptest
2014-07-22T09:31 -- 2014-07-22T10:00

package: /usr/lib64/kmod/ptest
2014-07-22T10:00 -- 2014-07-22T10:00

package: /usr/lib64/libpcre/ptest
2014-07-22T10:00 -- 2014-07-22T10:00

package: /usr/lib64/openssh/ptest
2014-07-22T10:00 -- 2014-07-22T10:37

package: /usr/lib64/openssl/ptest
2014-07-22T10:37 -- 2014-07-22T10:40

package: /usr/lib64/parted/ptest
2014-07-22T10:40 -- 2014-07-22T10:42

package: /usr/lib64/perl/ptest
2014-07-22T10:42 -- 2014-07-22T12:55

package: /usr/lib64/python/ptest
2014-07-22T12:55 -- 2014-07-22T13:05

package: /usr/lib64/sed/ptest
2014-07-22T13:05 -- 2014-07-22T13:07

package: /usr/lib64/strace/ptest
2014-07-22T13:07 -- 2014-07-22T13:07

package: /usr/lib64/tcl/ptest
2014-07-22T13:07 -- 2014-07-22T13:29

package: /usr/lib64/udev/ptest
2014-07-22T13:29 -- 2014-07-22T13:29

package: /usr/lib64/zlib/ptest
2014-07-22T13:29 -- 2014-07-22T13:29

Passed: 9065
Failed: 336
Skipped: 59
```

ptest-summary.sh and ptest-diff.sh Option Reference

Option descriptions for the **ptest-summary.sh** tool.

ptest-summary.sh Options

-d	Report detailed statistics (per-test pass/fail/skip)	
-f	Report failed test cases, must be used with -d option	
-h	Help and usage information (this message)	
-l	The name of the log file to process.	Typically something like ptest-run-2013-08-13T10:05.log
-s	Report summary statistics (aggregate pass/fail/skip)	

ptest-diff.sh Options

-d	Shows the detailed failures for the failed ptests.
-v	Verbose mode shows all section headers and test statistics.
-h	Help and usage information.

Running Open POSIX Tests

[About the Open POSIX Test Suite](#) 451

[Running the Open POSIX Tests](#) 452

About the Open POSIX Test Suite

Use the Open POSIX (Portable Operating System Interface) test suite to verify the POSIX-standards compliance of the application programming interface (API) for your Wind River Linux system image.

POSIX defines the API, utility interfaces, and command-line shells for software compatibility between Linux, or UNIX-based operating systems. Wind River includes the **Open POSIX Test Suite** as part of the **feature/test** template. When you configure and build a platform project using this template, the Open POSIX Test Suite is added to the target file system `/opt/open_posix_testsuite`.

To help simplify test execution, Wind River also provides the `/opt/open_posix_testsuite/wrLinux_posix/wr-runposix` wrapper script. Run this script without options to perform a complete POSIX test run on all scenario groups and test suites as defined in the subdirectories of the `/opt/open_posix_testsuite/conformance/interfaces` directory.

Once a test run completes, the test details are available in one of three separate files:

POSIX Test Results Test Log

This log file with PASS/FAIL test results is located in the `opt/open_posix_testsuite/wrLinux_posix/results` directory. This file includes a list of all the tests that were run, organized by scenario group. The file is named after the date and time of the test run, for example: `POSIX_RUN_ON_Mar_26_16h_57m_34s.log`.



NOTE: The file mentioned above indicates that the test run in this example completed on March 26, 2014 at 16:57. Your own test results will reflect the system date and time that the tests were run.

POSIX Failed Test Report

This report file is located on the target file system at **opt/open_posix_testsuite/wrLinux_posix/runtime/failtest.report**. This report lists the testcase(s) that failed, with the failure type as either: FAILED, HUNG, or UNRESOLVED.

POSIX Test Case Known Failures

This file, located on the target file system at **opt/open_posix_testsuite/wrLinux_posix/failtest/common**, provides the list of test cases and the Wind River Linux usage caveats and common workarounds when failures occur.

It is also possible to run tests on a specific scenario group or test suites, or run a specific test or tests from the command line. The scenario groups include:

- AIO
- MEM
- MSG
- SEM
- SIG
- THR
- TMR
- TPS

For detailed information on scenario groups and the specific tests included in them, refer to the Open POSIX website <http://posixtest.sourceforge.net/>

Running the Open POSIX Tests

Once you have configured and built a platform project with the **/feature/test** template, you are ready to run the Open POSIX Test Suite on the target platform.

Before you begin, you must configure and build a platform project using the **--with-template=feature/test** or **--enable-test** **configure** option. This is necessary to add the Open POSIX Test Suite to your platform project image.

NOTE: This example uses a QEMU emulation to launch a target window. If you are performing the tests on a hardware target board, launch the image on the target board as described in your board's BSP instructions.

Step 1 Launch the platform project image on a target.

```
$ make start-target
```

When prompted, enter **root** for the user name and password.

Step 2 Run the Open POSIX Test Suite.

There are several options available as shown below:

Options	Description
Full test, all groups	Run the following command: <pre># /opt/open_posix_testsuite/wrLinux_posix/wr-runposix [wr-runposix] Checking POSIX test plan.... [wr-runposix] POSIX test plan check pass [wr-runposix] Creating POSIX test file...</pre>
Test scenario group	Run the following command: <pre># /opt/open_posix_testsuite/wrLinux_posix/wr-runposix -f GROUP1, GROUP2</pre> In this example, <i>GROUPn</i> refers to the name of the test groups, as described in About the Open POSIX Test Suite on page 451.
Specific test suite	Run the following command: <pre># /opt/open_posix_testsuite/wrLinux_posix/wr-runposix -s SUITE1, SUITE2</pre> In this example, <i>SUITEn</i> refers to the name of the test suite in the <code>/opt/open_posix_testsuite/conformance/interfaces</code> directory. For example, the command: <pre># /opt/open_posix_testsuite/wrLinux_posix/wr-runposix -s timer_create</pre> runs all fifteen tests in the <code>/opt/open_posix_testsuite/conformance/interfaces/timer_create</code> directory.

Options	Description
Specific test suite and test in the suite	<p>Run the following command:</p> <pre># /opt/open_posix_testsuite/wrLinux_posix/wr-runposix -s timer_create -t timer_create_7-1.run-test SUMMARY ***** PASS 1 FAIL 0 ***** TOTAL 1 ***** <<Test end>> <<Test start>> Test Suite: clock_settime Test case: clock_settime_speculative_4-3.run-test ***** PASS 1 FAIL 0 ***** TOTAL 1 ***** <<Test end>> <<Test start>> Test Suite: clock_settime Test case: clock_settime_speculative_4-4.run-test</pre> <p>This example runs test 7-1, from the <code>/opt/open_posix_testsuite/conformance/interfaces/timer_create</code> directory.</p> <p>To run a different test, use the name of the directory and test located within it from the <code>/opt/open_posix_testsuite/conformance/interfaces</code> directory.</p>

Step 3 View the test results.

Open the log file in the `/opt/open_posix_testsuite/wrLinux_posix/results` directory. The file is named after the date and time of the test run, for example:

POSIX_RUN_ON_Month_Day_Hour_Minute_Second.log

The following output shows a completed test:

```
<<Test start>>
Test suite: timer_create
Test case: timer_create_speculative_5-1.run-test
*****
SUMMARY
*****
PASS      1
FAIL      0
*****
TOTAL     1
*****
<<Test end>>
<<wr-runposix Test Result>>
Total Tests: 15
Total Pass: 15
Total Fail: 0
Total Untested: 0
Total Unresolved: 0
Total Unsupported: 0
Total Hung: 0
Total Skip: 0
Kernel Version: 3.10.19.WR6.0.0.4_standard
Machine Architecture: x86_64
Hostname: qemu0
[wr-runposix] POSIX test passed
```

[Log Path] POSIX Test Log: /opt/open_posix_testsuite/wrlinux_posix/results/POSIX_RUN_ON-2014_Mar_27-14h_14m_46.log

Step 4 Optional: Troubleshoot the test results.

Refer to the `/opt/open_posix_testsuite/wrlinux_posix/failtest/common` file for a list of known Wind River Linux usage caveats and workarounds for completing the Open POSIX tests.

PART IX

Optimization

About Optimization.....	459
Analyzing and Optimizing Runtime Footprint.....	461
Reducing the Footprint.....	467
Analyzing and Optimizing Boot Time.....	471

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 467
- [*About devshell*](#) on page 468

Analyzing Boot-time

Analyzing boot-time (system startup) and using the data to reduce it. See:

- [*Analyzing and Optimizing Boot Time*](#) on page 471
- [*About Reducing Early Boot Time*](#) on page 475
- [*Reducing Network Initialization Time with Sleep Statements*](#) on page 475
- [*Reducing Device Initialization Time*](#) on page 476
- [*Removing Unnecessary Device Initialization Times*](#) on page 477

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 469

36

Analyzing and Optimizing Runtime Footprint

[Analyzing and Optimizing Runtime Footprint](#) 461

[Collecting Platform Project Footprint Data](#) 462

[Footprint \(fetch-footprint.sh\) Command Option Reference](#) 465

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.

- 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 then navigate to it.

```
$ mkdir -p qemux86-64_glibc-footprint && cd qemux86-64_glibc-footprint
```

- Configure the platform project.

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

- Build the file system.

```
$ make
```

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 462, 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 465, 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 (Optional step): Create an XML file you can use with Workbench to help you view and manage touched files.

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**.

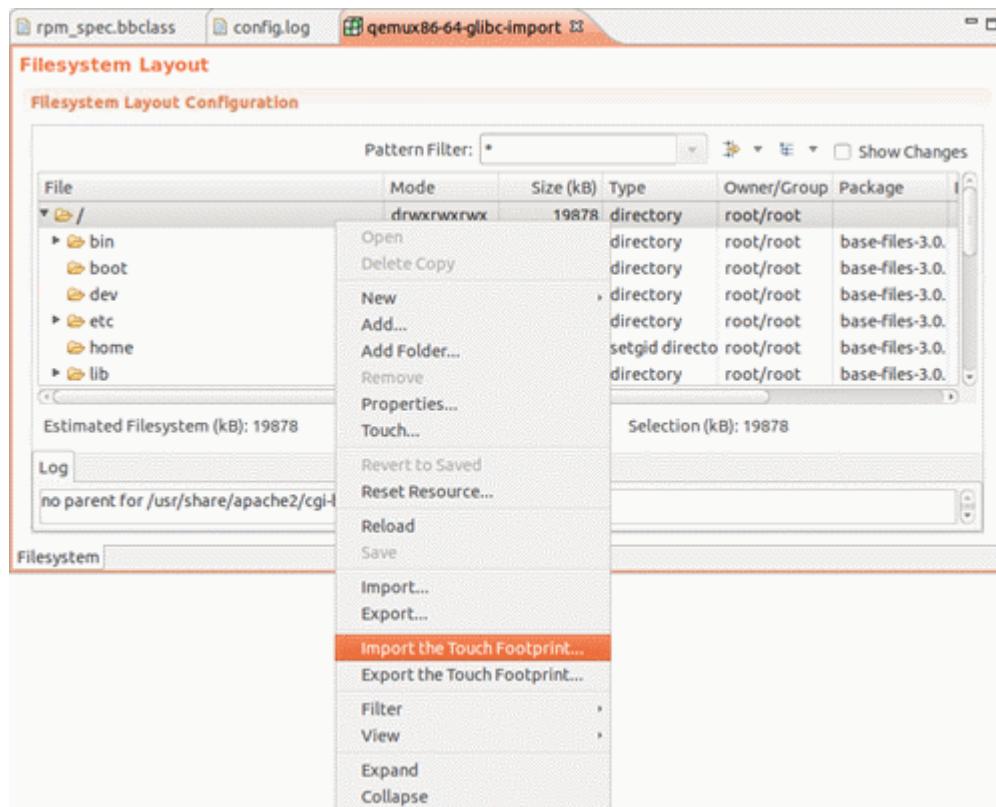
Select the directory for the platform project (where you created the footprint's XML file) and click **OK**.

- 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.

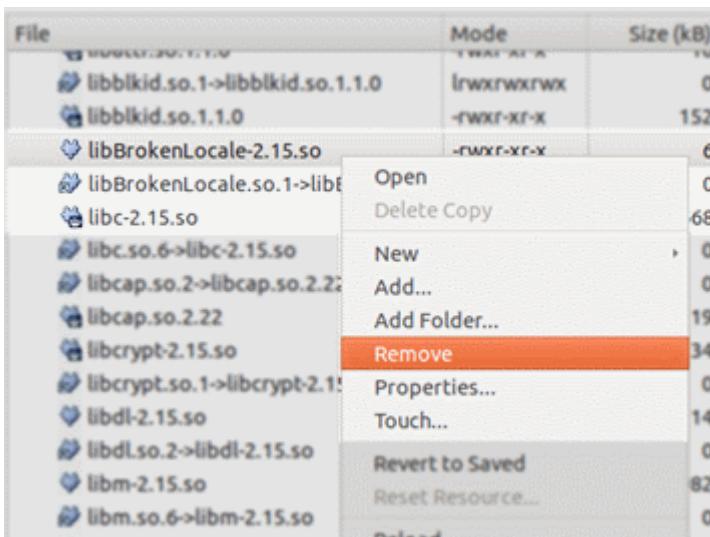
In the **Filesystem** tab, right-click and select **Import the Touch Footprint**. Browse to the XML file you copied to your development host in step [5.b](#) on page 463 and click **OK**.



- 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 touched files in the **Layout** view, 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 yyyy.mm.dd-HH:MM:SS	Specify the start date of the files to begin gathering footprint data on.
	-E yyyy.mm.dd-HH:MM:SS	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 <code>fetch_host_footprint</code> understands.
	-R	Display accessed, modified and changed timestamps.
	-X	Display the data in XML format.
	-T	Runs in BusyBox mode to display all timestamps in seconds from Epoch.

Reducing the Footprint

[About BusyBox 467](#)

[Configuring a Platform Project Image to Use BusyBox 467](#)

[About devshell 468](#)

[About Static Linking 469](#)

[About the Library Optimization Option 469](#)

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.

Related Links

[Configuring a Platform Project Image to Use BusyBox on page 467](#)

Add BusyBox to your platform project image to reduce its footprint.

Configuring a Platform Project Image to Use BusyBox

Add BusyBox to your platform project image to reduce its footprint.

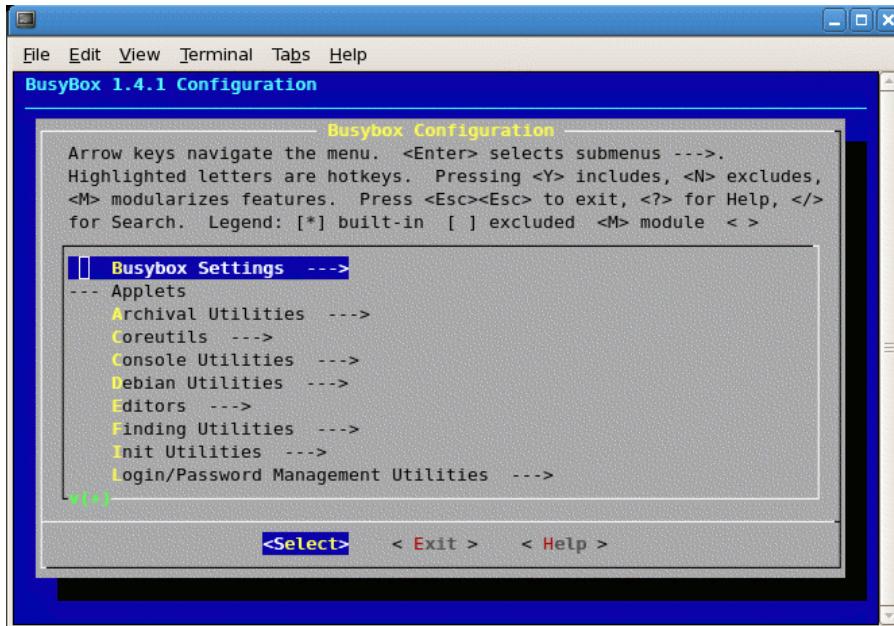
You may add or remove commands supported by the BusyBox executable in much the same way as you configure the Linux kernel. By removing commands you do not intend to use, you reduce the size of the executable even further.

Step 1 Open the BusyBox Configuration tool.

In the **projectDir/build** directory, enter the following command:

```
$ make 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 busybox
```

Related Links

[About BusyBox](#) on page 467

You can use BusyBox to reduce the footprint of your platform project image.

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*.

Related Links

<http://www.yoctoproject.org/docs/current/poky-ref-manual/poky-ref-manual.html#platdev-appdev-devshell>

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#include-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 82.

Like static linking (see *About Static Linking* on page 469), 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.

Analyzing and Optimizing Boot Time

[Analyzing and Optimizing Boot Time](#) 471

[Creating a Project to Collect Boot Time Data](#) 472

[Analyzing Early Boot Time](#) 473

[About Reducing Early Boot Time](#) 475

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.

Analyzing Early Boot Time

Use the **analyze-boottime** command to analyze early boot time data.

Before you begin, you must have 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 472.

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          patch_conexant_init
0.007950          serport_init
0.009109          pci_sysfs_init
0.009660          alsa_sound_init
0.009781          raid_init
...
0.171444          *          e100_init_module
0.223721          *          acpi_pci_root_init
0.266659          *          acpi_init
0.275421          *          dm_init
0.292454          serial8250_pnp_init
1.384742          *          pty_init
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:

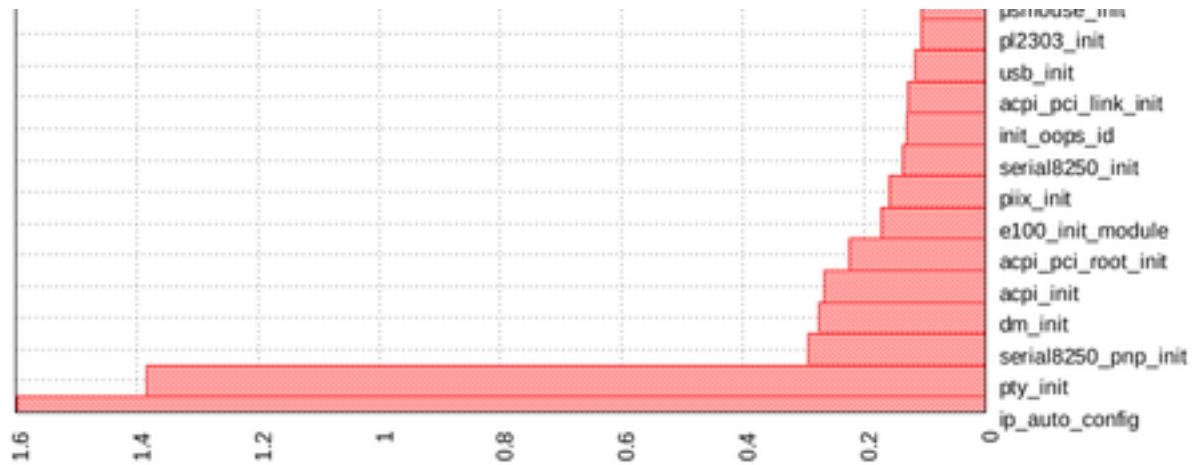
- `ip_auto_config`—see [Reducing Network Initialization Time with Sleep Statements](#) on page 475
- `pty_init`—see [Reducing Device Initialization Time](#) on page 476

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 .png file similar to the following sample:



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.

How you reclaim time from particular functions depends on the requirements of your particular end-system. For example, the instructions here refer to analyzing a single user system with known hardware and a fixed IP address. The challenges for a multi-user system with more complex networking requirements will require a different approach.

Reducing early boot time is accomplished by analyzing and optimizing to:

- Reduce network initialization time

The example output from [Analyzing Early Boot Time](#) on page 473 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 to avoid the overhead of DHCP negotiation.

You can assign a static IP address in QEMU with `make config-target` and setting the `TARGET_VIRT_IP` setting to a value such as 10.0.2.15.

With known hardware that does not require time to stabilize, it is possible to remove sleeps from the startup sequence. To do so, remove sleep statements in `build/linux-windriver/linux/net/ipv4/ipconfig.c`. See [Reducing Network Initialization Time with Sleep Statements](#) on page 475.

- Reducing device initialization time
- Removing unnecessary device initialization times

Reducing Network Initialization Time with Sleep Statements

You can improve network boot time by removing sleep statements.

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 472
- That 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 473.

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 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 linux-windriver.rebuild; make
```

Step 4 Reboot.

Step 5 Analyze the new boot logs as you did the first time in [Analyzing Early Boot Time](#) on page 473.

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  
0.292454          serial8250_pnp_init  
1.384742          *      pty_init  
1.597498          *      ip_auto_config  
  
Total : 325 Functions  
Boottime: 13.434060 sec  
  
$ host-cross/usr/bin/analyze-boottime -c -s \  
-i /tmp/w_fixedIP_no_sleeps/kernel-init.log |tail  
0.093698          slab_sysfs_init  
0.108006          *      ip_auto_config  
0.120562          *      e100_init_module  
0.142908          *      acpi_init  
0.151690          *      dm_init  
0.397953          pty_init  
  
Total : 325 Functions  
Boottime: 6.756338 sec
```

Notice the significant improvement in total boot time, as well as the fact that `ip_auto_config` is now not taking the most time.

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 472.
- You have run `analyze-boottime` command to obtain metrics on your device initialization time—see [Analyzing Early Boot Time](#) on page 473

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/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 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 473.

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 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/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 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**). 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, turn off only 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 473.

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 Development

Building a Self-Hosting Target System.....	481
Developing on the Target.....	483

Building a Self-Hosting Target System

[About Building Self-Hosting Target Systems](#) 481

[Building a Self-Hosting Platform Project Image](#) 482

About Building Self-Hosting Target Systems

Some newer hardware has the resources to allow builds to be done directly on the target.

Traditionally in the embedded world, target systems are built on a host system with lots of resources to produce a minimal sized target file system and kernel. With Wind River Linux 7.0, it is possible to create a self-hosting target system using the **feature/self-hosted** feature template.

This template adds most of the native packages to the target file system, providing a target system with the ability to build itself.



NOTE: The light-weight template **feature/target-toolchain** is also available, which adds only compiler toolchain and **binutils** to the target file system. For additional information, see [Feature Templates in the Project Directory](#) on page 61.

Although building a platform project image using the **feature/self-hosted** template creates a Wind River Linux typical of a (non-embedded) commercial distribution, it does not include an extensible package management system, and there is no support for compiling source RPMs (SRPMS). Despite this limitation, most portable packages will compile on the target.

Platform projects built on the target have the same limitations inherent in any Wind River Linux project, including::

- The build uses a pseudo environment.
- A user account is recommended to perform the build.
- You cannot build the project on an NFS mount. As a result, the target system must be a hardware target, and not a QEMU-based x86 platform project image.

In addition, the tools provided on the target file system can cross-compile the target itself or another Intel based target board, rebuild the target's file system on the target with added

features, or a rebuild a modified kernel. Alternately, you can build and deploy a QEMU simulated system on a physical target.

Building a Self-Hosting Platform Project Image

Learn how to build a platform project that supports a self-hosting target system.

All Intel targets support the **feature/self-hosted** template, but because this feature is intended for a hardware target, this procedure an Intel-based BSP.

Step 1 Configure the project.

Run the **configure** script to add the **feature/self-hosted** template.

```
$ configDir/configure \
--enable-board=intel-x86-64 \
--enable-rootfs=glibc_std \
--with-template=feature/self-hosted
```

Step 2 Build the target.

```
$ make
```

Developing on the Target

[Building a Wind River Linux Platform Project](#) 483

[Building Software Packages on the Target](#) 485

[Building a Modified Kernel on the Target](#) 487

Building a Wind River Linux Platform Project

Learn how to cross-compile an entire platform directly on the target if the hardware provides sufficient resources.

This procedure assumes that you have completed the steps in [Building a Self-Hosting Platform Project Image](#) on page 482.

This example illustrates cross-compiling on the target, and completely duplicates the process that normally takes place on a separate build host. In this case, the native tools provided in the **feature/self-hosted** template are used to bootstrap the build, and to build another copy of itself. The copy is then used to build a target file system and kernel.

Step 1 Build a platform project image with the **feature/self-hosted** template.

Perform this step on the development host. For additional information, see [Building a Self-Hosting Platform Project Image](#) on page 482.

Step 2 Export the build host Wind River Linux installation directory (*installDir*) to the NFS mount.

```
$ sudo echo "$installDir *(sync,rw,insecure,no_root_squash)" >> /etc/exports
```

This step saves time and disk space on the target system by making the contents of *installDir* available to the target system over an NFS mount on the host. Alternatively, you could also copy the contents of the *installDir* to the target system.

Step 3 Export the *installDir* on the host.

```
$ sudo service nfs-kernel-server restart
```



NOTE: The name of the NFS service varies by distribution, sometimes it is just **nfs**. If you do not have sudo rights, you can also start the user-mode NFS server provided with Wind River Linux.

Step 4 Connect the build host to the target system.

This step will vary with your target board or platform. Refer to your BSP documentation for the recommended method.

For additional information, see [About Target-based Networking](#) on page 491.

Step 5 Copy the kernel and rootfs to the target system's hard disk.

This includes copying the kernel and root file system files located on the host in **projectDir/export/images** to the partitions on your hardware target.

Step 6 Boot the target and log in.

This step will vary with your target board or platform. Refer to your BSP documentation for the recommended method.

Step 7 Add a normal user on the target:

- Create the user.

```
# useradd wruser
```

- Assign a password.

```
# passwd wruser
```

Step 8 Mount the build host's Wind River Linux install directory on the target:

- Create the mount point.

```
# mkdir /mnt/WindRiver
```

- Mount the install directory.

This step requires the IP address of the build host.

```
# mount -t nfs buildHostIP:installDir /mnt/WindRiver
```

- Change the permissions on **/opt**

```
# chmod a+wx /opt
```

- Change users.

```
# su - wruser
```

Step 9 Configure a simulator project on the target and build it.

- Create a target build directory and navigate to it.



NOTE: The platform project directory must not exist on an NFS mount. If it does, the build will fail.

```
# mkdir /opt/target-built-prj && cd /opt/target-built-prj
```

- b) Configure a new platform project.

```
# /mnt/WindRiver/wrlinux-7/wrlinux/configure \
--enable-board=qemu86-64 \
--enable-rootfs=glibc-std \
--enable-kernel=standard \
--enable-stand-alone-project=yes
```

- c) Build the project.

```
# make
```

 **NOTE:** The `--enable-stand-alone-project=yes` allows you to continue to build the project after build host's `installDir` mount is removed.

Step 10 Boot the QEMU target on the target.

```
# make start-target
```

Building Software Packages on the Target

Learn how to develop software packages on the self-hosted target system.

This example assumes that you have configured your project with the **feature/self-hosted** template as shown in [Building a Self-Hosting Platform Project Image](#) on page 482.

Most open source packages can now be installed on the target. In addition to a GNU toolchain, `make`, `autoconf`, `tar` and the basic headers and libraries needed for native compilation are available.

In this example; we have a large number of shell scripts that use `cURL` instead of `wget` from another project, and we wish to build this application with the most up to date encryption support.

The example also assumes QEMU support.

Step 1 Start the target.

Perform this step on the target system.

```
$ make start-target
```

Step 2 Ensure that DNS is configured.

```
# echo nameserver 8.8.8.8 >> /etc/resolv.conf
```

Replace 8.8.8.8 with a valid address for your lab environment.

Step 3 Download and build the **openSSL** library.

- a) Download the **openSSL** source.

```
# wget http://www.openssl.org/source/openssl-1.0.1e.tar.gz
```

- b) Unpack the downloaded file.

```
# tar xf openssl-1.0.1e.tar.gz
```

- c) Switch to the base of the source directory tree.

```
# cd openssl-1.0.1e
```

- d) Configure the build.

```
# ./config -fPIC
```

- e) Build the library.

```
# make
```

- f) Install the library.

```
# make install_sw
```

Once the command completes, the **openSSL** library is built and installed on the target.

Step 4 Download and build the **cURL** library.

- a) Change to your home directory.

```
# cd
```

- b) Download the **cURL** source.

```
# wget http://curl.haxx.se/download/curl-7.32.0.tar.gz
```

- c) Unpack the downloaded file.

```
# tar xf curl-7.30.0.tar.gz
```

- d) Switch to the base of the source directory tree.

```
# cd curl-7.30.0
```

- e) Prepare your environment for building the library.

```
# ./configure --with-ssl
```

- f) Build the library.

```
# make
```

- g) Install the library.

```
# make install
```

Once the command completes, the **cURL** library is built and installed on the target.

- h) Optionally, verify the **cURL** library installation was successful.

```
# curl -O https://www.openssl.org/source/openssl-1.0.1e.tar.gz
```

If the **cURL** library installed successfully, the system will return information related to the file's download progress.

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current	
			Dload	Upload	Total	Spent	Left	Speed
100	4355k	100	4355k	0	0	827k	0	0:00:05 0:00:05 --:--:-- 968k

Building a Modified Kernel on the Target

If you are profiling a leading-edge platform, it may be quicker to make minor kernel modification and rebuild on the target than to continually move new kernels from the build server to the test target.

This example assumes that you have configured your project with **feature/self-hosted** as shown in [Building a Self-Hosting Platform Project Image](#) on page 482.

All the tools to compile a Linux kernel are now resident on the target. Although it is possible to compile an arbitrary kernel from the internet, it is not recommended, and will produce an unsupported configuration. A similar workflow is to take the current kernel and modify it slightly; for example, as part of a benchmarking effort.

Step 1 Connect the build host to the target system.

This step will vary with your target board or platform. Refer to your BSP documentation for the recommended method.

For additional information, see [About Target-based Networking](#) on page 491.

Step 2 Copy the kernel and rootfs to the target system's hard disk.

This includes copying the kernel and root file system files located on the host in `projectDir/export/images` to the partitions on your hardware target.

Step 3 Boot the target and log in.

This step will vary with your target board or platform. Refer to your BSP documentation for the recommended method.

Step 4 Copy the kernel to your target file system from the target system terminal.

a) Change to your home directory.

```
# cd
```

b) Copy the kernel.

```
# scp -r build_host_ip:projectDir/build/linux-windriver/linux
```

c) Change to the `linux` directory.

```
# cd linux
```

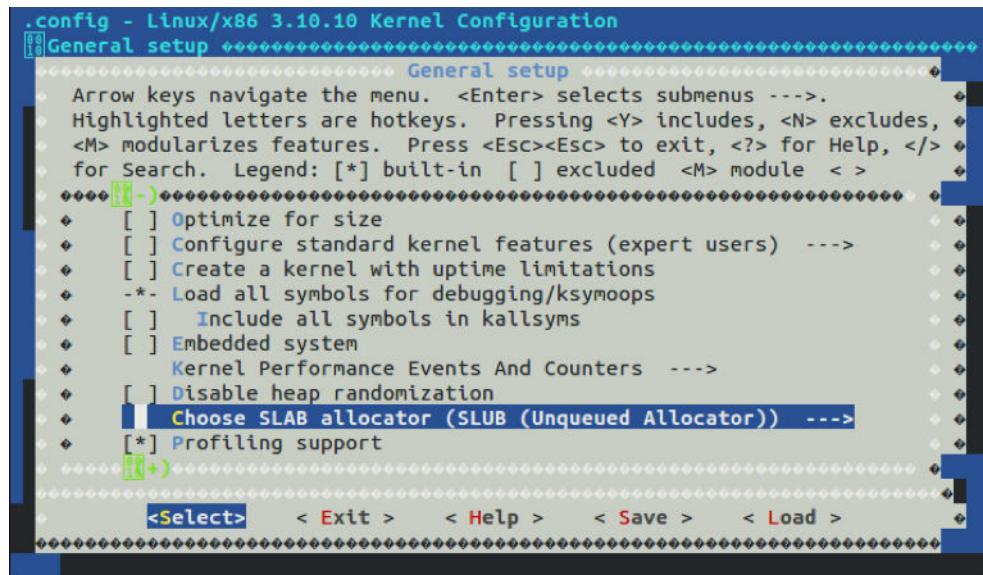
d) Configure the kernel.

```
# make menuconfig
```

The `menuconfig` utility loads.

a) Reconfigure your kernel as desired.

For example, to change the kernel heap algorithm from SLUB to SLAB, select **General setup > Choose SLAB allocator > SLAB**. then <Save>.



Step 5 Rebuild you kernel with the modified configuration.

```
# make bzImage
```

Step 6 Move the new kernel to the boot partition.

```
# cp arch/x86/boot/bzImage /boot/bzImage_test_SLAB
```

Step 7 Reboot the system.

```
# reboot
```

Step 8 Stop the bootloader.

The bootloader is currently set to boot from the original, pre-modified kernel. To use the new kernel, you must stop the bootloader, so you can access the boot menu to select the new kernel. This process will vary with your target board or platform.

For additional information, see [About Target-based Networking](#) on page 491.

Step 9 Select and boot the new kernel.

This process will vary with your target board or platform. For example, with some BSPs you do not mount the boot partition, and you cannot copy the kernel into it. Refer to your BSP documentation for the recommended method.

PART XI

Target-based Networking

About Target-based Networking.....	491
Setting Target and Server Host Names.....	493
Connecting a Board.....	495

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. For instructions on making these **README** files visible in your platform projec, see [README Files in the Development Environment](#) on page 47.



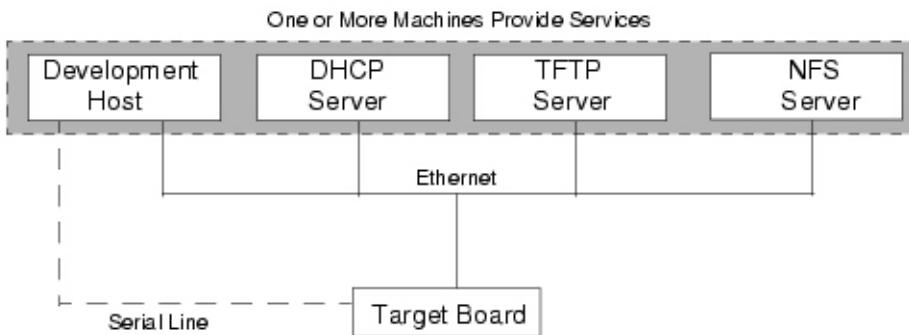
NOTE: Boot loader and network configuration are somewhat different for boards that use the PXE boot protocol. For details on network booting with PXE, refer to [About Configuring PXE](#) on page 496.

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, in which case a DHCP server supplies the IP address.
2. 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 distributed, 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.

42

Setting Target and Server Host Names

You may want to map your target and server IP addresses to host names for ease of reference.

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 212

The resulting file system will include the hosts file when downloaded from the server.

Step 3 Build the target file system.

```
$ make
```


Connecting a Board

[Configuring a Serial Connection to a Board](#) 495

[About Configuring PXE](#) 496

[Configuring PXE](#) 498

[Configuring DHCP](#) 499

[Configuring DHCP for PXE](#) 500

[Configuring NFS](#) 502

[Configuring TFTP](#) 503

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 and 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, serviced 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 and 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 uses TFTP to download the PXELinux boot loader. This 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 PXE bootloader searches for the correct configuration file. For example, if the PXE bootloader is looking for the configuration file for the **target.lab.org**, 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, the search would be conducted as follows:

1. 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.)

2. 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

3. Not finding a hexadecimal file name, the bootloader will search for files in the following order:

C0A80A0

C0A80A

C0A80

C0A8

C0A

C0

C

4. Finally, not finding any of these files, the PXE bootloader will look for a file named **default**.

Configuring PXE

This example illustrates the procedure to configure the Pre-boot Execution Environment (PXE) boot loader on most x86 boards with Wind River Linux board support packages (BSPs)

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 boot loader file into place..

Copy the PXELinux boot loader file

/usr/lib/syslinux/pixelinux.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/pixelinux.cfg
# touch /tftpboot/pixelinux.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 be 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 as the first boot option.

NOTE: You may be able to find PXE boot images on the Web, for example at <http://www.rom-o-matic.net/>.

Your target goes through the following sequence as it boots:

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 complete 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

/etc/dhcpd.conf

configuration file, and you must create a **dhcpd.leases** file if one does not already exist, as described in this section.

A sample file is presented below 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.

- Provide your board's MAC address as the value for **hardware Ethernet**.
- Adjust values in the sample **/etc/dhcpd.conf** to match your environment.

Values that may require adjustment include::

- **domain-name**
- **domain-name-servers**
- the board's **host name** and **fixed-address**

Step 2 Create a **dhcpd.leases** file if needed.

- Determine if the file exists.

```
ls /var/lib/dhcp/dhcpd.leases
```

- 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 specific instructions.

On Red Hat Linux, you would run:

```
$ service dhcpcd start
```

Configuring DHCP for PXE

DHCP configuration must be changed to support PXE.

Step 1 Edit the **/etc/dhcpd.conf** file.

- Add the PXE boot additions.

```
allow booting;
allow bootp;
```

- 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

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

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 specific instructions.

On Red Hat Linux, you would run:

```
$ service dhcpcd start
```

Configuring NFS

Booting a target from the network requires adjustments to your NFS settings to provide a root file system.

You must have NFS installed and enabled before proceeding. Refer to your host documentation for details.

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 following command sequence as root:

```
$ 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

To boot over a network, you must correctly set up TFTP to download the kernel image from a server.

Step 1 Make the kernel available for download.

The default TFTP download directory is typically **/tftpboot**. If a download directory for TFTP does not already exist, you must create it. For the name of your TFTP download directory and instructions to optionally change it, refer to your server documentation.

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**.

On many Linux systems, the TFTP server is automatically started by **inetd** or **xinetd** when a connection request is received. To enable the TFTP server with **xinetd**, 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.

Refer to your system documentation for details on how to enable TFTP.

Step 3 Restart **xinetd**.

The command to restart a service varies across Linux distribution. Consult your server's documentation for specific instructions.

On Red Hat Linux, you would run:

```
$ service xinitd restart
```

PART XII

Reference

Additional Documentation and Resources.....	507
Common make Command Target Reference.....	513
Build Variables.....	525
Platform Kernel Versions by Product Release.....	531
Lua Scripting in Spec Files.....	533

Additional Documentation and Resources

Document Conventions	507
Wind River Linux Documentation	508
Additional Resources	508
Open Source Documentation	509
External Documentation	511

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-7/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.

All Wind River Systems documentation is available in the Knowledge Library <https://knowledge.windriver.com>.

Additionally, much of the documentation is available through the start menu of the installation host, for example under **Applications > Wind River > Documentation** in the Gnome desktop for Linux.

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**).

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_7/wind_river_linux_users_guide_70/wind_river_linux_users_guide_70.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_7/wind_river_linux_users_guide_70/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>	http://www.rpm.org/max-rpm/
Additional, useful information on using RPMs	

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>	http://www.netfilter.org/documentation
This includes a good deal of documentation on packet filtering, NAT, and tutorials.	
<i>VPN HOWTO</i>	http://www.tldp.org/HOWTO/VPN-HOWTO/index.html

Security

Reference Title	Link
<i>Netfilter/Iptables HOWTO</i>	http://www.netfilter.org/documentation
This includes a good deal of documentation on packet filtering, NAT, and tutorials.	
<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
BitBake User Manual	Online: http://www.yoctoproject.org/docs/1.7/bitbake-user-manual/bitbake-user-manual.html
OpenEmbedded Core (OE-Core)	Online: http://www.openembedded.org/wiki/OpenEmbedded-Core
QEMU	Online: http://wiki.qemu.org
GNU Toolchain Documentation	<p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-arm-wrs-linux-gnueabi</i>—for ARM-based target systems</p> <p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-aarch64-wrs-linux-gnu</i>—for 64-bit ARM-based target systems</p> <p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-i686-wrs-linux-gnu</i>—for x86-based target systems</p> <p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-mips-wrs-linux-gnu</i>—for MIPS-based target systems</p>
GNU Toolchain Documentation	<p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-arm-wrs-linux-gnueabi</i>—for ARM-based target systems</p> <p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-aarch64-wrs-linux-gnu</i>—for 64-bit ARM-based target systems</p> <p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-i686-wrs-linux-gnu</i>—for x86-based target systems</p> <p><i>installDir/wrlinux-7/layers/binary-toolchain-4.9-3/share/doc/wrs-linux-mips-wrs-linux-gnu</i>—for MIPS-based target systems</p>

45

Common make Command Target Reference

Wind River Linux provides many **make** build arguments to simplify platform project image, application, kernel, and userspace development from the command-line and with Workbench.

Platform Project Development

make Command in <i>projectDir</i>	Workbench Build Target	Description
make	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.
make		For information on using make , see About the make Command on page 98.
make all		
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 98.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make fetchall		Fetches package sources so you can run a build offline, without a connection to the Internet.
make world.fetchall		The commands with options include: make fetchall
make universe.fetchall		Downloads the package sources required to build the filesystem based on the configuration of the platform project. This command is equivalent to make fs.fetchall .
	make world.fetchall	Downloads the package sources for all target system components.
	make universe.fetchall	Downloads all package sources. These commands can be useful when working with a minimal installation of Wind River Linux or with user-defined layers that perform package source downloads during build time.
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 gdb-server , or alternately deployed on with the file system for on-target debug with gdb . This build target is only supported with production builds; all other build types include debug information in the default file system image.

make Command in <i>projectDir</i>	Workbench Build Target	Description
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 config.sh (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 send-error-report <i>server_location</i>		Used to send an error report to a specified error report server when a platform project is configured to enable error reporting.
		For additional information, see Creating Platform Project Build Error Reports on page 30.
make upgrade		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.
		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.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make busybox.menuconfig		Menu-based tool to configure busybox This is the equivalent of bitbake -c menuconfig busybox within the BitBake environment.
make export-dist		Configures the <i>projectDir/export/dist</i> directory and builds the file system when necessary.
make host-tools		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.
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.
<hr/> <p>NOTE: This command is the equivalent of source layers/oe-core/oe-init-buildenv bitbake_build.</p> <hr/>		

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 Command in <i>projectDir</i>	Workbench Build Target	Description
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 usb-image-burn		Use to create a bootable USB image from any existing platform project image, and burn the image directly to a USB flash drive.

Application Development

make Command in <i>projectDir</i>	Workbench Build Target	Description
make export-layer		<p>Provides a simplified method to move project-specific application and kernel development to another platform project.</p> <p>Creates a directory and compressed file that includes the contents of the <i>projectDir/layers/local</i> directory, and any project-specific application and kernel configuration changes made to the platform project. This includes the sysroot and toolchain.</p> <p>Once the command completes, the directory and compressed .tar file are available in the <i>projectDir/export/export-layer</i> directory. The exported project directory and compressed file are named after the platform project directory, with a date and time stamp added; for example:</p> <p>qemux86-64-glibc-small-20140424-1110PDT</p>

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		Forces the sysroot population in the projectDir .
		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.
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 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 projectDir/export/host-tools.tar.bz2 archive.
make populate-sdk		Same as make sysroot , above.

make Command in <i>projectDir</i>	Workbench Build Target	Description
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 <i>recipeName</i>		Build the recipe <i>recipeName</i> This is the equivalent of bitbake <i>recipeName</i> within the BitBake environment.
make <i>recipeName</i>.addpkg		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 100 for a Yocto Project equivalent to this command.
make <i>recipeName</i>.clean		Clean the package identified by <i>recipeName</i> Using this command will undo any source file changes made in your package directory, consistent with Yocto Project and OpenEmbedded package build target rules. 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. This is the equivalent of bitbake -c clean <i>recipeName</i> within the BitBake environment.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make <i>recipeName</i>.compile		This will only do the compile. If you just specify <i>recipeName</i> (with no .compile suffix), the top level dependency of .sysroot will trigger and the build system will compile the package, generate an RPM, and install it to the sysroot.
		This is the equivalent of bitbake -c compile <i>recipeName</i> within the BitBake environment.
make <i>recipeName</i>.distclean		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.
		This is the equivalent of bitbake -c distclean <i>recipeName</i> within the BitBake environment.
make <i>recipeName</i>.install		This is the equivalent of bitbake -c install <i>recipeName</i> within the BitBake environment.
make <i>recipeName</i>.patch		Copy package source into the build area and apply patches.
		This is the equivalent of bitbake -c patch <i>recipeName</i> within the BitBake environment.
make <i>recipeName</i>.rebuild		Clean, then build a package.
		This is the equivalent of bitbake -c rebuild <i>recipeName</i> within the BitBake environment.
make <i>recipeName</i>.rebuild_nodep		Rebuild <i>recipeName</i> without rebuilding dependent packages. For example, enter make -C build linux-windriver.rebuild_nodep to rebuild the kernel without rebuilding dependent userspace packages.
		This is the equivalent of bitbake -c rebuild_nodep <i>recipeName</i> within the BitBake environment.

make Command in <i>projectDir</i>	Workbench Build Target	Description
make <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 100 for a Yocto Project equivalent to this command.
make <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 package-manager	package-manager	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
make linux-windriver	kernel_build	Build the Wind River Linux kernel recipe

make Command in <i>projectDir</i>	Workbench Build Target	Description
make linux-windriver.build	kernel_build	<p>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.</p> <p>This is the equivalent of bitbake linux-windriver within the BitBake environment.</p>
		Clean the kernel build
make linux-windriver.compile		<p>Compiles the kernel source specified in the EXTERNALSRC_pn-linux-windriver option in the <i>projectDir/local.conf</i> file.</p>
make linux-windriver.compile_kernelmodules		<p>Compiles the any kernel modules from kernel source specified in the EXTERNALSRC_pn-linux-windriver option in the <i>projectDir/local.conf</i> file.</p>
make linux-windriver.config	kernel_config	<p>Extract and patch kernel source for kernel configuration.</p> <p>This is the equivalent of bitbake -c config linux-windriver within the bitbake environment.</p>
make linux-windriver.extract_kernel_output		<p>Creates a kernel image and modules .tar file in the location specified in the KERNEL_EXTRACTDIR_pn-linux-windriver option in the <i>projectDir/local.conf</i> file.</p>
make linux-windriver.menuconfig	kernel_menuconfig	<p>Extract and patch kernel source and launch menu-based tool for kernel configuration.</p> <p>This is the equivalent of bitbake -c menuconfig linux-windriver within the bitbake environment.</p>

make Command in <i>projectDir</i>	Workbench Build Target	Description
make 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 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 DT\$basename.dtb		This supersedes make -C build linux-windriver.DTSbaseName.dtb . It must be run from a kds shell.

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 configure 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 templates/arch/... 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 ppc and ppc64 would be listed. A value from this variable is substituted for the VARIANT prefix in the following variables.
VARIANT_COMPATIBLE_CPU_VARIANT	Specifies all of the CPU variants that are compatible with the specific variant. For example ppc is compatible with ppc_750 .
VARIANT_TARGET_ARCH	The architecture used by GNU configure to specify that variant.

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**.

Variable	Description
<code>VARIANT_ TARGET_COMMON_CFLAGS</code>	CFLAGS that are beneficial to pass to an application but not required to optimize for a multilib. Equivalent of <code>CFLAGS=...</code> 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 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).
<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

Variable	Description
BOOTIMAGE_JFFS2_ARGS	For targets that support JFFS2 booting, these values will be passed when creating the JFFS2 image. Endianess (-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 BUILD_DIR . 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

Variable	Description
TARGET_QEMU_KERNEL_OPTS	<code>export</code> directory and will instead use the image you specified.
TARGET_QEMU_OPTS	These are any extra options you might want to pass to the kernel boot line to override the defaults.
TARGET_LIBC	Value should be glibc or eglibc . 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. This is hidden from the application space and is for internal Wind River use only.
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.

Platform Kernel Versions by Product Release

Linux Kernel versions shipped with Wind River Linux are updated regularly.

The following table provides platform kernel versions as they relate to Wind River Linux releases:

Product Platform Release	Kernel Version	Release Date
Wind River Linux 1.4	2.6.14	12 Oct 2006
Wind River Linux 1.5	2.6.14	01 May 2007
Wind River Linux 2.0	2.6.21	15 Dec 2007
Wind River Linux 3.0.1	2.6.27.21	14 Sep 2009
Wind River Linux 3.0.2	2.6.27.39	11 Dec 2009
Wind River Linux 3.0.3	2.6.27.47	04 Aug 2010
Wind River Linux 4	2.6.34.6	22 Oct 2010
Wind River Linux 4.1	2.6.34.8	18 Mar 2011
Wind River Linux 4.2	2.6.34.9	20 Jul 2011
Wind River Linux 4.3	2.6.34.10	15 Dec 2011
Wind River Linux 5	3.4.10	28 September 2012
Wind River Linux 5.0.1	3.4.34	20 March 2013
Wind River Linux 6.0	3.10.19	09 December 2013
Wind River Linux 7.0	3.14.23	02 February 2015

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>.

Index

.bb files 40
.bbclass files 40
.conf files 40, 41

A

about Yocto 17
add-on products 22
adding a device 178
adding a directory 179
adding a file 180
adding a pipe 181
adding application packages 209
adding functionality with kernel fragments 174
adding, application 212
analysis 61, 102
analysis tool
 support for MIPS targets 351
 support for non-MIPS targets 351
analysis tools 345
analysis, footprint 462
Apache 108
appending files or directories 315
application
 adding to root file system 211
 adding to rootfs with fs_final*.sh 212
application package 215
application packages
 adding 209, 213
 existing project 209
application tree 209
application, adding
 application tree 209
 changelist.xml 209
 configure command 209
applicaton package, verifying 214
archiver 128
archiver template 129
ARCHIVER_MODE 61, 131
audit reporting 293

B

baseline kernel 262
bash 461
bbappend file 284
BBFILES 41
bblayers.conf
bblayers.conf file contents 41
bc 217
benchmark 61
binutils 481
bitbake
 documentation 511

BitBake 227, 284
bitbake configuration 46
BitBake Environment Display Tool 222
bitbake invalid characters 35
BitBake name limitations 21
bitbake string values 46
BitBake task engine 468
BitBake **wrbutil show-env** 226
bitbake-layers
 flatten 164
board, connecting 496
board, connection 495
boot loader
 boot loader parameters 496
Bootable USB Images 394
bootloader 21
BSP 21
bsp branch 264
BSP cross-reference 21
BSP layer 152
build 102
build analysis 102
Build configuration option reference 53, 82
build environment 55
Build Environment Template 61
build history 253, 255
build variables
buildtools 103
busybox 235
BusyBox 467

C

Carrier Grade Linux 509
cfg 265
CGL Kernels 435
changelist.xml 178–181, 209, 211
clean up 285
config_baseline.cfg 284
config.sh
 build variables
configuration files 41
configuration fragments 67
configure command 209
configure script
 help 82
 options 82
 reference 82
cots 17
cross-compile 483
custom kernel headers
 exporting 316
custom kernel recipe 306, 308

D

debug
 kprobe 348–350
DEFAULT_IMAGE 226
Developer Web Site 508
development directory 35
development environment 17, 35, 55
development README 47
devshell 468
DHCP 500
DHCP server
 IP address 499
discretionary access control (DAC) 22
django 103
Django 108
DOC_COMPRESS 379
documentation
 HTML 508
 open source documentation 511
 PDF 508
documentation compression on target 379
dos2unix 219
downloads
 enable 192
 source 192
dummy kernel recipe 305

E

elfutils 61
embedded devices 17
empty string 46
empty values 46
emulated target 286
enable posix 452
enable test 452
environment display tool options 227
export 502
EXPORT_SYSROOT_HOSTS 202, 203
exported patch 240
exporting 199, 201
exporting a package 238
exportPatch command 237
exportPatches.tcl 237
extra downloads 192

F

fakestart.sh 380, 381
feature 61
feature branching 263
feature templates 61, 174
feature/self-hosted 483, 485, 487
feature/self-hosted feature template 481
feature/target-toolchain 481
fetch-footprint.sh 461, 465
Fibonacci 212
file system layout 177, 182

final kernel 262
footprint 461, 462, 467
footprint data 465
fragments 40
fs_final.sh script 374
fs_final*.sh 178, 212
fuzz factor 230

G

gdb 61, 209, 213, 214
git 265
git branch 262–264, 266
git clone README 47
git instaweb README 48
git kernel repository 261
git log 255
git show README 47
git-based installer 24, 25
git-based kernel sources
git-repo-subset script
 help 53
 options 53
 reference 53
git-repo-subset.sh
 about 48
 examples 50
 meta-openembedded 48, 50
 subset repository 48
 using 50

glibc
 custom distro 138
 customize 138
 distro features 138
 features map 140
 options map 140
glibc and libm 469
glibc-sato 18
glibc-small 18
glibc-std 18
GNU
 GNU Toolchain Documentation 511
GNU toolchain 188
group files 374
Grsecurity 435
gzip 379

H

history 253
host names
host, patch preparation 237

I

IMAGE_INSTALL 248
image-manifest
 about 126
 compare builds 126

importing packages 215
 importing, application package 215
 importing, from web 219
 importing, source package 217
 importing, SRPM package 219
 install directory 55
 installation 35
 installDir 507
 inter-dependencies 221
 Internet download 192

K

kernel
 initial configuration 282
 patching 299
 rebuilding 285
 kernel branch 263
 kernel build 264, 266
 kernel configuration
 identifying issues 293
 kernel configuration changes 297
 kernel configuration file 282
 kernel configuration fragments 282, 286
 kernel customization 264
 kernel development 265, 309
 kernel feature 261–263
 kernel fragment 284
 kernel git tree 263
 kernel history 261–263
 kernel meta 264
 kernel modification 487
 kernel modules 287, 289
 kernel output, extracting 309
 kernel patches 266
 kernel profile 18
 kernel recipe 305
 kernel recipes 266
 kernel sources, git-based
 kernel tools 265
 kernel tree 261
 kernel type 262, 263
 kernel type branch 263
 kernel types 18
 kernel uprev 262
 kernel version
 by release
 kernel with fragments 283, 284
 kernel workflow 264
KERNEL_INSTALL_HEADER 315
KERNEL_INSTALL_HEADER_append 315
 kernel, conditional 319
 kernel, patching 231
 kernel.org 261
 kgit 265
 kprobe 348–350
KVM
 creating host and guest 427
 deploying 428

L

LAMP 61
 LAN 376
 latest RCPL 23
 layer 71
 layer repositories 48
 layer, patch 241
 layer.conf file contents 41
 layers
 combine 164
 layers directory 35
 layers/local, directory structure 71
 ldconfig 375
 leading path name 230
 libm 469
 Library optimization 469
 license
 commercial 192
 flags 192
 non-commercial 192
 provider name 192
 type 192
 whitelist 192
 Linux Development 509
 Linux distributions 17
 Linux Standards Base 509
 linux-windriver kernel package 285
 linux-windriver.do_install() 315
 linux-windriver.do_populate_sysroot 315
 linux-windriver.install_kernel_headers 315
list-packageconfig-flags 221
 loading shared objects 375
 local area network, connecting target 376
 local layer
 patches 299
 populating 283
 local.conf file contents 41
LSB
 distribution tests 435
 pre-built target 434
 LSB application compliance
 Apply for Certification 437
 LSB Distribution Tests 433
 LSB Test Requirements 433
 lsbtesting 433
 ltrace 61
 lua scripting
 post-installation
 pre-installation

M

machine.conf file 41
 maintenance tool 23

make command
application development
kernel development
platform project image development
pr-server
reference
userspace development
man page compression [379](#)
managing files and directories with XML [178](#)
manual pages, support [378](#)
memstat [61](#)
menuconfig [289](#)
meta-series [266](#)
metadata [40](#)
multi-category security (MCS) [22](#)
multi-level security (MLS), [22](#)
multiple projects [108](#)
mysql [61](#)
mysql-odbc [61](#)

N

Network Boot Process
network server
Networking [509](#)
NFS [502](#)

O

Open POSIX Test Suite [451](#)
open source development [509](#)
open source documentation [509](#)

P

package [248, 255](#)
package build history [253](#)
package information [244](#)
Package Manager [244, 245, 249](#)
Package Revision
 PR [251](#)
 PR Server [251](#)
package-management [61](#)
package, removing [249](#)
packages
 importing [215](#)
packages, removing [245](#)
parse configure command [97](#)
password [374](#)
patch [229, 241](#)
patch application and resolution [230](#)
Patch Reversal [230](#)
patch, verifying [240](#)
patches [67, 237, 297, 299](#)
patching [230](#)
patching a package [238](#)
patching packages [235, 237](#)
patching, kernel [231](#)
performance [469](#)

platform project
 build errors [30](#)
 error report [30](#)
 error report server [30](#)
Platform Project Configure tool [97](#)
platform project image [209](#)
platform project information [244](#)
portability [469](#)
POSIX [451](#)
posix test [452](#)
PR server [253, 254](#)
preempt_rt [18](#)
preempt-rt [319, 321](#)
preemption levels [321](#)
PRMAIN_nohist [254](#)
processing, templates [175](#)
Processor type and features [319](#)
product version [23](#)
project directory [55](#)
project directory, creating
 command-line [56](#)
 Workbench [56](#)
projectDir [507](#)
pseudo [380, 381](#)
ptest [61, 441, 442](#)
ptest-diff.sh [446](#)
ptest-summary.sh [442, 447](#)
PXE [500](#)
python [103](#)

Q

QEMU
 applying options [360](#)
 boot options [360](#)
 command line options [357](#)
 configuration options [356](#)
 ending a session [361](#)
 port mapping [360](#)
 resolving errors [359](#)
 running multiple sessions [359](#)
 running simulation [357](#)
 starting a session [358](#)
 starting a session with graphics console [360](#)
 starting from disk [359](#)
 starting from ISO [359](#)
Quilt [237](#)

R

RCPL, latest [23](#)
README files [47](#)
README installDir [47, 48](#)
README, BSP [21](#)
real-time feature [321](#)
real-time kernel [321](#)
recipe [235](#)
recipe file, .bb [209](#)
recipes [40](#)

removing, package 249
 removing, packages 245
 root file system 211, 502
 root file system options 18
 rootfs types 18
 RPM 177, 377
 RPM, adding to a running target 377
 Running ptest 446, 447
 running target 377
 runtime
 footprint 461
 optimizing 461

S

sample Toaster build output 103
 saved state cache 103
 scc 231, 265
 SCC 297
 scm 261
 SDK
 exporting 199, 201–203
 importing 202, 203
 installing 199, 201
 Security 509
 Security Profile 22
 self-hosted target 485
 self-hosting target 481, 482
 serial connection 495, 496
 serial port 495
 server IP addresses
 Simics
 basic target console 366
 configuring a target 367
 graphics target console 366
 prerequisites 366
 x86 BSP acceleration 367
 source control 261
 source package, importing 217
 SQL queries 254
 SQLite 108
 sqlite3 254
 SRPM package 219
 sstate-cache
 security 114
 signing 114
 verify objects 114
 whitelist 114
 sstate-cache security
 passphrase 119
 private key 119
 public key 119
 standard branch 264
 standard kernel 264
 Static linking 469
 strace 61
 sublayer 152
 support 508
 symlink, options 182

System recovery 22

T

target IP addresses
 template 175
 template processing 175
 templates 61, 70, 173–175
 templates, adding 248
 terminal emulator 495, 496
 terminal shell 468
 test log 452
 testing 442
 TFTP 499, 503
 tiny kernel, about 18
 toaster 102
 Toaster
 home page 103
 running 103
 URL 103
 web server 103
 toochain 188
 tools 188
 touch.xml 182
 tree build 266
 troubleshoot test 452
 TUN/TAP
 configuring from command line 363
 configuring with existing target 362
 configuring with Workbench wizard 362
 turbostat 18

U

UEFI 18
 uimage 503
 unresolved rejects 230
 updating 23

V

verifying, application package 214
 virtio 425
 VirtualBox 418
 vmdk
 boot image 416
 VMware 416

W

wget 217
 white space 230
 Wind River Linux documentation 508
 Wind River make commands 100
 Wind River Online Support 508
 Windows application development
 SDK 202, 203
 Workbench 345, 355

Workbench analysis tools [345](#)
Workbench patch manager [229](#)
Workbench Tutorials [508](#)
workflow, patching [229](#)
wrbbutil show-env usage [227](#)
wrl-audit-image.py
 about [126](#)
 arguments [126](#)

Y

Yocto [102, 511](#)
Yocto compatible syntax [173](#)
Yocto infrastructure
Yocto Project [188, 468](#)