WIND

AN INTEL COMPANY

WIND RIVER® LINUX

TOOLCHAIN AND BUILD SYSTEM USER'S GUIDE

7.0

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.
Toll free (U.S.A.): +1-800-545-WIND
Telephone: +1-510-748-4100
Facsimile: +1-510-749-2010

For additional contact information, see the Wind River website:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

Linux

Toolchain and Build System User's Guide, 7.0

26 April 2017

Contents

1
Setting Up the Build Environment

Build System Environment Options

Wind River Linux provides options for displaying information about and enhancing certain aspects of the build system.

The build system environment differs from that of the platform, application, or kernel developer in that all developers rely on the build system.

Where a platform developer will initialize and set up the environment to facilitate development as described in the *Wind River Linux Platform Developer's Guide: Initializing the Wind River Linux Environment*, and an application developer will source the SDK as described in the *Wind River Linux User Space Developer's Guide: Sourcing the SDK*, changes to the build system environment impact all users.

This includes the following:

- Displaying details about the BitBake build environment for use in managing your development workflow. For additional information, see The BitBake Build Environment Display Tool on page 2.
- Creating a reusable layer with open source package source not included with a Wind River Linux installation. For additional information, see Creating a Reusable Layer with Downloaded Package Source on page 13
- Adding new repository layers for all developers to use to customize their platform project builds. For additional information, see Subset Repositories of meta-openembedded Layers Overview on page 7.

If you wish to set up a specific environment as a developer, see:

- Platform developers see the *Wind River Linux Platform Developer's Guide: Initializing the Wind River Linux Environment*
- Application developers see *Wind River Linux User Space Developer's Guide: Application Developer Environment Overview*

The BitBake Build Environment Display Tool

The **wrbbutil show-env** tool is an external program that you can use to extract the build 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 **wrbbutil 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

You run **wrbbutil show-env** from inside the Wind River BitBake shell; which you start with the **make bbs** command.

Alternately, you can invoke **wrbbutil show-env** using a single command; as an option when running the **make bbc** command from the users shell. The benefit of this approach is that it does not require the interactive BitBake shell. For additional information, see Displaying the BitBake Environment Details on page 5.

The **wrbbutil 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 you run the commands 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 **wrbbutil show-env** command. See BitBake Environment Display Tool Reference on page 7.

The **wrbbutil 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 **wrbbutil show-env** to standard error, but all output related to the environment data will be emitted to standard output.

For example, if you run the following command, 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:

```
$ wrbbutil show-env > /tmp/environment
```

Tool Option Examples

For additional information, see BitBake Environment Display Tool Reference on page 7.

-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/wrbbutil 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="${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE'),d)[0] or
'defaultpkgname'}"
PN="zip"
PR="r2"
# PRAUTOINX="${PF}"
PRAUTOINX="zip-3.0-r2"
...
}
```

If the **-r** (or **--recipe**) option is not specified, then the environment data presented by **wrbbutil 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 it's 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 **wrbbutil show-env** command, the output will generate a shell unset command because of those internal flags:

```
$ ../layers/wrlcompat/scripts/wrbbutil 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/wrbbutil 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 **wrbbutil 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 **wrbbutil 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 use the following command to display the image name along with the default package name:

```
$ ../layers/wrlcompat/scripts/wrbbutil show-env DEFAULT_IMAGE PN
```

Which results in the following output:

```
...
DEFAULT_IMAGE="wrlinux-image-glibc-std"
# PN="${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE'),d)[0] or 'defaultpkgname'}"
PN="defaultpkgname"
```

In the context of the image, you can obtain the same output with the following command:

```
$ ../layers/wrlcompat/scripts/wrbbutil 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(d.getVar('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/wrbbutil 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(d.getVar('FILE'),d)[0] or 'defaultpkgname'}"
PN="zip"
```

Displaying the BitBake Environment Details

View BitBake environment information for a project using the **wrbbutil 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 **wrbbutil show-env** command.

Prerequisites

You must have built a platform project image before performing the following steps:

Procedure

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/wrbbutil show-env"` If using this option then you can skip Step 2.

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
   ```

3. Run the **wrbbutil show-env** command.

   Now that you are in the BitBake shell you can run the tool.

   ```
   $ ../layers/wrlcompat/scripts/wrbbutil 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} ${ERROR_QA}"
   ALL_QA="textrel files-invalid incompatible-license xorg-driver-abi
   libdir ldflags installed-vs-shipped rpaths dev-so debug-deps dev-deps debug-files
   arch pkgconfig la perms             useless-rpaths staticdev pkgvarcheck already-
   stripped             compile-host-path dep-cmp install-host-path
   packages-list perm-config perm-line perm-link pkgv-undefined             pn-
   overrides split-strip var-undefined version-going-backwards"
   ...
   # PN="${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE'),d)[0] or
   'defaultpkgname'}"
   PN="defaultpkgname"
   # PR="${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE'),d)[2] or 'r0'}"
   PR="r0"
   # PRAUTOINX="${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 **wrbbutil show-env** command follow this usage format: **show-env options** *VARNAME*.

wrbbutil show-env	**-f** | **--flags**	Append flag(s) to variable name.
wrbbutil show-env	**-q** | **--quote**	Map all special/unprintable characters to string escapes.
wrbbutil show-env	**- r** *recipeName* | **--recipe** *packageName*	Show environment from context of *packagName* recipe.
wrbbutil show-env	*VARNAME*	Show only the variable named *VARNAME*.

Subset Repositories of meta-openembedded Layers Overview

Wind River Linux provides the **git-repo-subset.sh** script 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 12.

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 9.

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.

Procedure

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:

   ```
   $ git-repo-subset.sh -s meta-xfce -w my_workdir git://
   git.openembedded.org/meta-openembedded meta-xfce
   ```

   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 *my_workdir*.

   To create a branch, instead of a new repository, use the **-b** option. For example:

   ```
   $ git-repo-subset.sh -b -s meta-xfce -w my_workdir git://
   git.openembedded.org/meta-openembedded meta-xfce
   ```

2. View the contents of the repository:

   ```
   $ 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
   ```

   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	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..

1. Create a filter-list file.

   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:

   ```
   meta-oe/classes/
   meta-oe/conf/
   meta-oe/COPYING.MIT
   meta-oe/licenses/
   meta-oe/README
   meta-oe/recipes-benchmark/iperf/
   ```

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

2. Run the following command to add the sub-repository:

   ```
   $ 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	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.
	1. Create a filter-list file.
	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:
	`meta-oe/classes/`
	`meta-oe/conf/`
	`meta-oe/COPYING.MIT`
	`meta-oe/licenses/`
	`meta-oe/README`
	`meta-oe/recipes-benchmark/iperf/`
	`meta-perl/conf/`
	`meta-perl/COPYING.MIT`
	`meta-perl/README`
	`meta-perl/recipes-perl/libtest/`
	`meta-ruby/`
	Save the file as **filter-list-multi-layers.txt**.
	2. Run the following command:
	`$ git-repo-subset.sh -w my_workdir -l path_to/filter-list-multi-layers.txt git://git.openembedded.org/meta-openembedded meta-openembedded-mysubset`
	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.
	3. View the contents of the repository:
	`$ tree -L 1 my_workdir/meta-openembedded-mysubset`
	`├── meta-oe`
	`├── meta-perl`
	`└── meta-ruby`
	Notice that it includes the layers and contents specified in the **filter-list-multi-layers.txt** file.

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 \
--enable-internet-download=yes \
--with-package-iperf \
--enable-reconfig=yes
```

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
```

The **--enable-internet-download=yes** option makes it possible to fetch the package source for the **iperf** package, used in this example to demonstrate how to add a package from the repository.

The **--enable-reconfig=yes** option is included to provide the ability to disable Internet download functionality after you have installed the **iperf** package. This setting is optional.

3. Obtain the **iperf** package source **.tar.gz** file.

   Run the following command from the platform project directory:

   ```
   $ make -C build iperf.fetch
   ```

   Once the command completes, the package file is available in ***projectDir*/bitbake_build/downloads/iperf-2.0.5.tar.gz**.

4. Optionally, remove Internet download functionality for the platform.

   Perform this step if you do not intend to add additional applications from the Internet to your platform project image.

   Repeat the **configure** command from *Step 2*, but remove the **--enable-internet-download=yes** option; for example:

   ```
   $ configDir/configure \
   --enable-board=qemux86-64 \
   --enable-kernel=standard  \
   --enable-rootfs=glibc_small \
   --with-layer=my_workdir/meta-oe \
   --with-package-iperf \
   --enable-reconfig=yes
   ```

5. Build the platform project.

   ```
   $ make
   ```

   Once the command completes, the contents of the layer will be part of your platform project image. The **iperf** binary is located in ***projectDir*/export/dist/usr/bin/iperf**.

Repository Subset Script Options Reference

You can run the **git-repo-subset.sh** script with a 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.

Option	Description
-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 subdirectory 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.

Creating a Reusable Layer with Downloaded Package Source

You can create a layer with package source not provided with your Wind River Linux installation.

Prerequisites

The following procedure requires that:

- You have a previously configured and built platform project—see the *Wind River Linux Platform Developer's Guide: Platform Project Image Configuration Overview* and the *Wind River Linux Getting Started Command Line Tutorials: Creating and Configuring a Platform Project*.
- You are familiar with package or component licensing requirements described in Licensing and Including External Package Sources on page 49. 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.

Procedure

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 Licensing and Including External Package Sources on page 49.

2. Optionally install a package from the downloaded source.

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

a) 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.

b) Build the **lame** package.

```
$ make lame
```

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

c) 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.

d) 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.

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 Licensing and Including External Package Sources on page 49.

2
Configuration and Build

Build Options

Using the make Command

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

Platform Projects

After you have configured a project as described in the *Wind River Linux Platform Developer's Guide: Platform Project Image Configuration Overview*, you can build it using the **make** command. The build produces the target software such as the kernel and file system for a particular board, depending on how you configured it.

When you run **make**, **make fs**, or **make all**, it builds (or rebuilds) the platform project using the specified options. If source changes are detected, the binary packages associated with those changes are automatically rebuilt. Like many Wind River Linux **make** targets, this is a wrapper to an equivalent of Yocto build command, in this case: **bitbake wrlinux-image-filesystem-type**; for a cross reference see Yocto Project Equivalent make Commands on page 18.

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 65 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 21.

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 21.

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

```
$ make -C build recipeName
```

Or simplify the command in the following manner:

```
$ 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 dependent 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 21.

Wind River Linux make command	Yocto Project BitBake equivalent
make bbs Sets up the BitBake environment, such as the variables required, before you can run BitBake commands. This command executes a new shell environment and configures the environment settings, including the working directory and *PATH*. To return to the previous environment, simply type **exit** to close the shell.	Source layers/oe-core/oe-init-buildenv bitbake_build
make	**bitbake** *imageName* For example, **bitbake wrlinux-image-glibc-std**. To determine the correct *imageName*, you can either: • 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: `DEFAULT_IMAGE = "wrlinux-image-glibc-std"` **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.
make *recipeName* Build the package's recipe *recipeName*.	**bitbake** *recipeName*
make *recipeName***.rebuild** Rebuild the package's recipe *recipeName*.	**bitbake -c rebuild** *recipeName*

Wind River Linux make command	Yocto Project BitBake equivalent
make linux-windriver Build the Wind River Linux kernel's recipe.	**bitbake linux-windriver**
make linux-windriver.rebuild Rebuild the Wind River Linux kernel's recipe.	**bitbake -c rebuild linux-windriver**

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* **NOTE:** Notice that the bitbake flag in this case is **-C** rather than **-c**.
make -C build *packageName***.quilt** and **make -C build** *packageName***.quiltprep**	**bitbake -c quiltprep** *recipeName* **NOTE: quiltprep** is not a community step. It is a Wind River addition.
make -C build *packageName***.addpkg** **NOTE:** You must specify the package name for this command option.	Edit **layers/local/*image*.bb** and add the following line: `IMAGE_INSTALL += "packageName"`
make -C build *packageName***.rmpkg** **NOTE:** You must specify the package name for this command option.	Edit **layers/local/*image*.bb** and remove or comment out the following line: `IMAGE_INSTALL += "packageName"`

Wind River Linux make command	Yocto equivalent
make -C build *packageName***.env**	**bitbake -e** *recipeName*

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 *projectDir*	Workbench Build Target	Description
make **make** **make all**	**fs**	Each of these commands builds a new file system from RPMs where available, use source otherwise. Note that there is no difference and the commands are interchangeable. For information on using **make**, see Using the make Command on page 17.
make build-all	**build-all**	Performs the same action as **make**. For information on forcing a source build, see Using the make Command on page 17.

make **Command in *projectDir***	**Workbench Build Target**	**Description**
make fetchall **make world.fetchall** **make universe.fetchall**		Fetches package sources so you can run a build without a connection to the Internet (offline). The following are descriptions of the commands: make 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 Command in *projectDir*	Workbench Build Target	Description
make fs-debug	**fs-debug**	This produces an additional file system image in the *projectDir*/**export** 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 help		View command-line help information for the **make** command
	delete	Remove the *project*_**prj** 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 ***projectDir*/config.log** file as a reference if you need to recreate the project at a later date.

make Command in *projectDir*	**Workbench Build Target**	**Description**
make send-error-report *server_location*		Used to send an error report to a specified error report server when a platform project is configure to enable error reporting. For additional information, see *Wind River Linux Platform Developer's Guide: Creating Platform Project Build Error Reports*.
make upgrade		Upgrades the platform project build to the latest product update. Each time you update Wind River Linux, run this command in the ***projectDir*** to ensure your platform project is built with the latest available features. Once this command is run, the ***projectDir*/config.log** file will indicate the RCPL (product update) in use, and automatically select the latest RCPL with the highest number.
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 ***projectDir*/export/dist** 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 ***projectDir*/export/host-tools.tar.bz2** archive.

make Command in *projectDir*	Workbench Build Target	Description
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 *PATH*. To return to the previous environment, simply type **exit** to close the shell. **NOTE:** This command is the equivalent of source layers/oe-core/oe-init-buildenv bitbake_build.

Image Deployment

make Command in *projectDir*	Workbench Build Target	Description
make start-qemu		Start a QEMU simulation. Use **make start-target TOPTS="--help"** for a list of options.
make start-target		Start a QEMU simulation. Use **make start-target TOPTS="--help"** for a list of options.
make usb-image		Use to create a bootable USB image from any existing platform project image. The image includes two partitions: 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

make Command in *projectDir*	Workbench Build Target	Description
make usb-image-burn		Use to create a bootable USB image from any existing platform project image, and burn the image directly to a USB flash drive.

Application Development

make Command in *projectDir*	Workbench Build Target	Description
make export-layer		Provides a simplified method to move project-specific application and kernel development to another platform project. Creates a directory and compressed file that includes the contents of the ***projectDir*/layers/local** directory, and any project-specific application and kernel configuration changes made to the platform project. This includes the sysroot and toolchain. Once the command completes, the directory and compressed **.tar** file are available in the ***projectDir*/export/export-layer** directory. The exported project directory and compressed file are named after the platform project directory, with a date and time stamp added; for example: **qemux86-64-glibc-small-20140424-1110PDT**

make **Command in *projectDir***	**Workbench Build Target**	**Description**
make export-sdk	**export-sdk**	Creates a SDK suitable for application development in the **export/** directory, which can be used for providing build specs in Workbench. This option first builds the root file system, then populates the SDK. This includes the sysroot and toolchain, and is the preferred method to use to set up the environment for application development.
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**	This performs the same action as **make export-sdk** above, but does not export the toolchain. This is designed to simplify application development overhead, for when updates occur for the sysroot, but not the (generally unchanging) toolchain.
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 **Command in** ***projectDir***	**Workbench Build Target**	**Description**
make populate-sdk		Same as **make sysroot**, above.
make populate-sysroot		Same as **make export-sysroot**, above. This is the equivalent of **bitbake -c populate_sdk** *imageName* within the BitBake environment.

Package and Recipe Management

make **Command in** ***projectDir***	**Workbench Build Target**	**Description**
make *recipeName*		Build the recipe *recipeName* This is the equivalent of **bitbake** *recipeName* within the BitBake environment.
make *recipeName* **.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 18 for a Yocto Project equivalent to this command.

make **Command in** ***projectDir***	**Workbench Build Target**	**Description**
make *recipeName* **.clean**		Clean the package identified by *recipeName* 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** *recipeName* within the BitBake environment.
make *recipeName* **.compile**		This will only do the compile. If you just specify *recipeName* (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** *recipeName* within the BitBake environment.
make *recipeName* **.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** *recipeName* within the BitBake environment.

make Command in *projectDir*	**Workbench Build Target**	**Description**
make *recipeName* **.install**		This is the equivalent of **bitbake -c install** *recipeName* within the BitBake environment.
make *recipeName* **.patch**		Copy package source into the build area and apply patches. This is the equivalent of **bitbake -c patch** *recipeName* within the BitBake environment.
make *recipeName* **.rebuild**		Clean, then build a package. This is the equivalent of **bitbake -c rebuild** *recipeName* within the BitBake environment.
make *recipeName* **.rebuild_nodep**		Rebuild *recipeName* 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** *recipeName* within the BitBake environment.

make Command in *projectDir*	Workbench Build Target	Description
make *packageName* **.rmpkg**		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** *packageName***.addpkg** 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 18 for a Yocto Project equivalent to this command.
make *recipeName* **.unpack**		Unpack the packages source but stop before patching phases. This is the equivalent of **bitbake -c unpack** *recipeName* 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 *projectDir*	Workbench Build Target	Description
	Kernel Configuration	Opens the Wind River Workbench tool for kernel configuration.
make linux-windriver	**kernel_build**	Builds the Wind River Linux kernel recipe.

make Command in *projectDir*	Workbench Build Target	Description
make linux-windriver.build	**kernel_build**	Builds the Linux kernel recipe. If you have made changes to the kernel in your project (for example with **make -C build linux-windriver.menuconfig**), run the **linux-windriver.rebuild** target, not this one, to get those changes to take effect. This is the equivalent of **bitbake linux-windriver** within the BitBake environment. Cleans the kernel build.
make linux-windriver.compile		Compiles the kernel source specified in the **EXTERNALSRC_pn-linux-windriver** option in the *projectDir*/**local.conf** file.
make linux-windriver.compile_kernelmodules		Compiles the any kernel modules from kernel source specified in the **EXTERNALSRC_pn-linux-windriver** option in the *projectDir*/**local.conf** file.
make linux-windriver.config	**kernel_config**	Extract and patch kernel source for kernel configuration. This is the equivalent of **bitbake -c config linux-windriver** within the BitBake environment.
make linux-windriver.extract_kernel_output		Creates a kernel image and modules **.tar** file in the location specified in the **KERNEL_EXTRACTDIR_pn-linux-windriver** option in the *projectDir*/**local.conf** file.

make Command in *projectDir*	Workbench Build Target	Description
make linux-windriver.menuconfig	**kernel_menuconfig**	Extracts and patches kernel source and launches a menu-based tool for kernel configuration. This is the equivalent of **bitbake -c menuconfig linux-windriver** within the BitBake environment.
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**	Extracts and patches kernel source and launches the X-Window tool for kernel configuration. This is the equivalent of **bitbake -c xconfig linux-windriver** within the BitBake environment.
make DTSbasename.dtb		This supersedes **make -C build linux-windriver.DTSbaseName.dtb** . It must be run from a **kds** shell.

Build Variables Reference

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 it is 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 1 **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.
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**.	
*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.
*VARIANT*_ **TARGET_COMMON_CFLAGS**	CFLAGS that are beneficial to pass to an application but not required to optimize for a multilib. Equivalent of **CFLAGS=**... in the environment or in a Makefile.
*VARIANT*_ **TARGET_CPU_VARIANT**	Name of a variant. Also used as the RPM architecture.
*VARIANT*_ **TARGET_ENDIAN**	**BIG** or **LITTLE**
*VARIANT*_ **TARGET_FUNDAMENTAL_ASFLAGS**	Flags to be passed to the assembler when using the toolchain wrapper to assemble with a given user space. These are hidden from applications.
*VARIANT*_ **TARGET_FUNDAMENTAL_CFLAGS**	lags to be passed to the compiler when using the toolchain wrapper to compile for a given user space. These are hidden from applications.
*VARIANT*_ **TARGET_FUNDAMENTAL_LDFLAGS**	Flags to be passed to the linker when using the toolchain wrapper. These are hidden.
*VARIANT*_ **TARGET_LIB_DIR**	The name of the library directory for the ABI - **lib**, **lib32**, **lib64**.

Variable	Description
*VARIANT_* **TARGET_OS**	**linux-gnu** or **linux-gnueabi**
*VARIANT_* **TARGET_RPM_PREFER_COLOR**	The preferred color when installing RPM packages to the architecture: • 0—No preference • 1—ELF32 • 2—ELF64 • 4—MIPS ELF32_n32 Color is RPM terminology for a bitmask used in resolving conflicts. If an RPM is going to install two files, and they have conflicting md5sum or sha1 values, 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).
*VARIANT_* **TARGET_RPM_TRANSACTION_COLOR**	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.
*VARIANT_* **TARGET_RPM_SYSROOT_DIR**	The internal gcc directory prefix to get to the sysroot information.
*VARIANT_* **TARGET_USERSPACE_BITS**	Bitsize of a word, **32** or **64**.
BSP-Specific Variables	
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.

Variable	Description
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 *projectDir* 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 **export** directory and will instead use the image you specified.
TARGET_QEMU_KERNEL_OPTS	These are any extra options you might want to pass to the kernel boot line to override the defaults.
TARGET_QEMU_OPTS	These are any additional options you need to pass to the QEMU binary to get it to run correctly. In the case of the ARM Versatile and MTI Malta boards, the **-M** argument is passed

Variable	Description
	so that the QEMU host binary will be configured with the correct simulation model since each host binary supports multiple simulation models within the same architecture.
Feature or Root File System Specific Items	
TARGET_LIBC	Value should be **glibc** or **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 **TARGET_COMMON_COPT** **TARGET_COMMON_CXXOPT**	These are all optional optimizations that override defaults in configure. Generally you use these if you want to change the optimizations for **-O**s and not **-O2**. See the **glibc_small rootfs** for an example.

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.

Build Logs Overview

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.

Specifying Parallel Builds with Workbench

Learn how to increase efficiency through parallel builds for multi-threaded applications.

In this example, you will use the Client/Server sample project to introduce multi-threaded application builds with Workbench. You will specify the number of threads that Workbench will use to build your application. Depending on the application, specifying the number of parallel jobs can improve application build time.

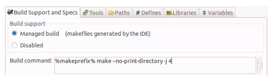
NOTE: This procedure only enhances the build time for multi-threaded applications. There is no performance gain for single-threaded applications.

To specify the number of parallel builds for an application, do the following:

Procedure

1. From the Workbench main menu, select **File** > **New** > **Example**.
2. Select **Wind River Linux Application Sample Project**, then click **Next**.
3. Select the **Client/Server Demonstration Program**, then click **Finish**. The sample project is created and displays in the Project Explorer.
4. Right-click on the **clientserver** project in the Project Explorer and select **Properties** to display the Properties dialog.
5. Select **Build Properties** in the left pane, then select the **Build Support and Specs** tab.
6. In the **Build command** field, enter the following:

   ```
   -j 4
   ```

Options	Description
 Build support: ● Managed build (makefiles generated by the IDE) ○ Disabled Build command: %makeprefix% make --no-print-directory -j 4	

   The value **4** represents the number of parallel jobs Workbench uses to build the application. You may want to experiment with this number to find the best performance gains for your application development needs.
7. Click **OK**.
8. Optionally, set the active build spec to match the platform project image that you wish to build the application for.

   This step is necessary if you want to test the application on a simulated or hardware target. If you do not set the build spec, it will default to the development host native environment (**native-standard-native**).

   To set the build spec to match a platform project image, right-click on the application project, then select **Build Options** > **Set Active Build Spec >** *platformProjectBSP-rootfsType-arch-projectDir*. For example, **qemux86-64-glibc_std-x86_64-qemux86_64_std_prj**.
9. Workbench creates a build spec for each platform project that you configure and build.
10. In the Project Explorer, right-click on the **clientserver** project and select **Build Project**.
11. When prompted to rebuild the index, select **Yes**. Select **Remember my decision** to avoid being prompted each time you make a properties change.

Workbench builds the executable, and the Build Console at the bottom of the Workbench perspective shows the build output.

Configuration and Processing

Layer Processing and Configuration

The build system configures layers in a hierarchical relationship.

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

Layers provide templates and packages, while templates provide configuration information. 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.

Project 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 the *Wind River Linux Platform Developer's Guide: Configuration Files and Platform Projects*

Recipes (**.bb**) files

See the *Wind River Linux Platform Developer's Guide: Recipes Overview*.

Classes (**.bbclass**) files.

Appends (**.bbappends**) files to existing layers and recipes. See the *Wind River Linux Platform Developer's Guide: Recipes Overview*.

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 the *Wind River Linux Platform Developer's Guide: Platform Project Image Configuration Overview*, and the *Wind River Linux Platform Developer's Guide: Platform Project make Command and Build Logs*.
- Custom layers and/or templates with their own configuration, recipes, classes, and append files. See the *Wind River Linux Platform Developer's Guide: Layers Overview*.
- Additional changes and additions that apply to the runtime file system only, and not the platform project image. See the *Wind River Linux Platform Developer's Guide: Options to Add Applications to a Project Image*.

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:

- *Wind River Linux Platform Developer's Guide: Configuration Files and Platform Projects*
- *Wind River Linux Platform Developer's Guide: Creating a New Layer*
- *Wind River Linux Platform Developer's Guide: Creating a Recipe File*

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 **ndefined** 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.

Procedure

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

Options	Description
Command line	Run the following command from the ***projectDir***: `$ echo VIRTUAL-RUNTIME_dev_manager = \" \" >> local.conf`
Edit local.conf file	**1.** Open the ***projectDir*/local.conf** file in an editor and add the following line to it: `VIRTUAL-RUNTIME_dev_manager=" "` **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.

2. Rebuild the file system.

Run the following command from the ***projectDir***.

```
$ make
```

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=""
```

The 'devshell' Development Shell

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

3

License Compliance

Open Source License Compliance Overview

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 the *Wind River Linux Platform Developer's Guide: Feature Templates in the Project Directory*.

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

Refer to the Archiver Options Reference on page 48.

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.

You can add the **archiver** feature to a new or existing platform project as described in the following procedure.

Procedure

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.
	`$ configDir/configure \`
	`--enable-board=qemux86-64 \`
	`--enable-kernel=standard \`
	`--enable-rootfs=glibc_small \`
	`--with-template=feature/archiver`

Enable the archiver feature on an existing platform project.	Run the following command from the platform project directory.
	`$ configDir/configure \`
	`--enable-board=qemux86-64 \`
	`--enable-kernel=standard \`
	`--enable-rootfs=glibc_small \`
	`--enable-reconfig \`
	`--with-template=feature/archiver`

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

2. Optionally, add a package.

   This step provides 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.

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.

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
```

5. Edit 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 48.

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 09 16:00 .
   drwxrwxr-x 3 dev1  dev1      4096 Oct 09 16:00 ..
   -rw-rw-r-- 2 dev1  dev1  32475140 Oct 09 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"	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. **NOTE:** The changes to the **archiver** are made by appending the options in the **local.conf** file. Usage example: `ARCHIVER_MODE[src] = "original"`
ARCHIVER_MODE[src]	"patched"	This value provides the patched source.
ARCHIVER_MODE[src]	"configured"	This value provides the configured source.
ARCHIVER_MODE[diff]	"1"	The patches between **do_unpack** and **do_patch** are created. You can set the one that you'd like to exclude from the diff as shown below: **ARCHIVER_MODE[diff-exclude] ?= ".pc autom4te.cache patches"**
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.

Licensing and Including External Package Sources

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.

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* value 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
```

Procedure

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	**1.** Run the **md5sum** command on the license file.
	`$ md5sum layers/local/recipes-local/my-app/LICENSE`
	The system returns the checksum value, for example:
	`2ebc7fac6e1e0a70c894cc14f4736a89 LICENSE`
	2. Enter the numerical value only in the **md5=** section. For example:
	`LIC_FILES_CHKSUM = "file://`
	`LICENSE;md5=f27defe1e96c2e1ecd4e0c9be8967949"`

Options	Description

Use the build system	**1.** Run the **make** command to build the package recipe.
	
	`$ make recipeName`
	
	The build will fail. This is expected.
	
	2. Scan the build output for the license checksum value.
	
	For example:
	
	`ERROR: my-app: md5 data is not matching for file://`
	`LICENSE;md5=8e7a4f4b63d12edc5f72e3020a3ffae8`
	`ERROR: my-app: The new md5 checksum is`
	`2ebc7fac6e1e0a70c894cc14f4736a89`
	
	- 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*.

Postrequisites

See also:

- *Wind River Linux Platform Developer's Guide: Recipes Overview*
- *The Yocto Project Poky Reference Manual: Track License Change* at http://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html#usingpoky-configuring-LIC_FILES_CHKSUM

4

Monitoring the Builds

Build Management 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 the *Wind River Linux Build System Command Line Tutorials: Running Toaster*

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

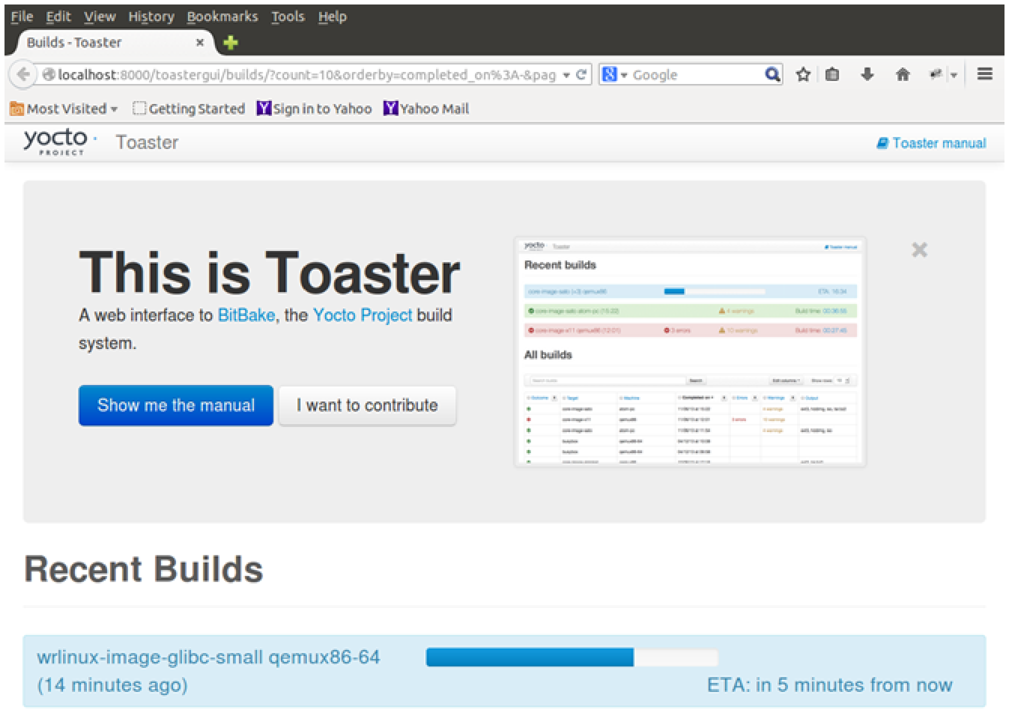
Sample Toaster Build Output

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.



This is Toaster

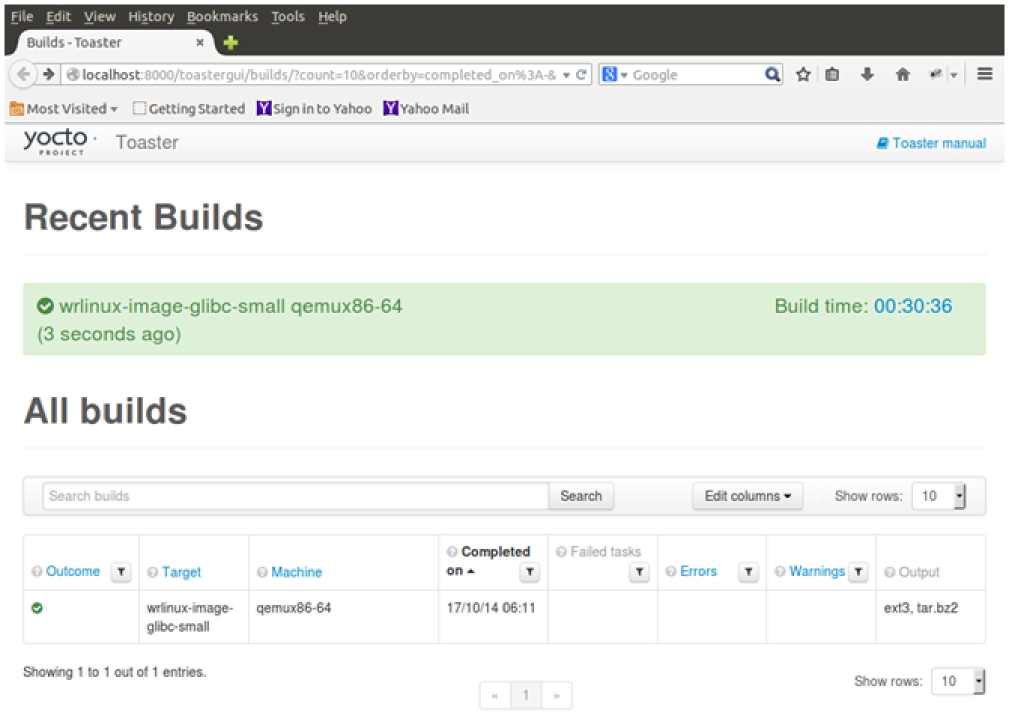
A web interface to BitBake, the Yocto Project build system.

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.


File Edit View History Bookmarks Tools Help
Builds - Toaster
localhost:8000/toastergui/builds/?count=10&orderby=completed_on%3A-&
Google
Most Visited
Getting Started
Sign in to Yahoo
Yahoo Mail
yocto PROJECT
Toaster
Toaster manual

Recent Builds

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

All builds

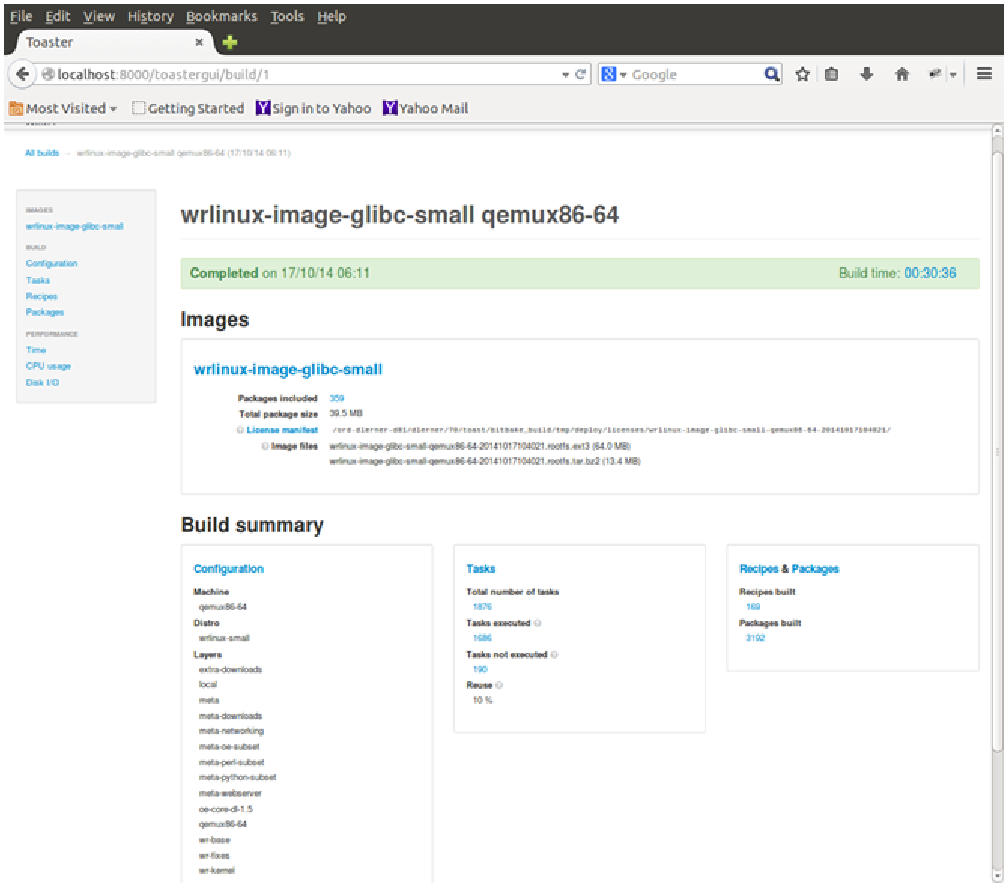
Search builds | Search | Edit columns | Show rows: 10

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

Showing 1 to 1 out of 1 entries.

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.


File Edit View History Bookmarks Tools Help
Toaster
localhost:8000/toastergui/build/1
Google
Most Visited
Getting Started
Sign in to Yahoo
Yahoo Mail

All builds → wrlinux-image-glibc-small qemux86-64 (17/10/14 06:11)

IMAGES
wrlinux-image-glibc-small
BUILD
Configuration
Tasks
Recipes
Packages
PERFORMANCE
Time
CPU usage
Disk I/O

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	/ord-dlerner-d81/dlerner/70/toast/bitbake_build/tmp/deploy/licenses/wrlinux-image-glibc-small-qemux86-64-20141017104021/
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)

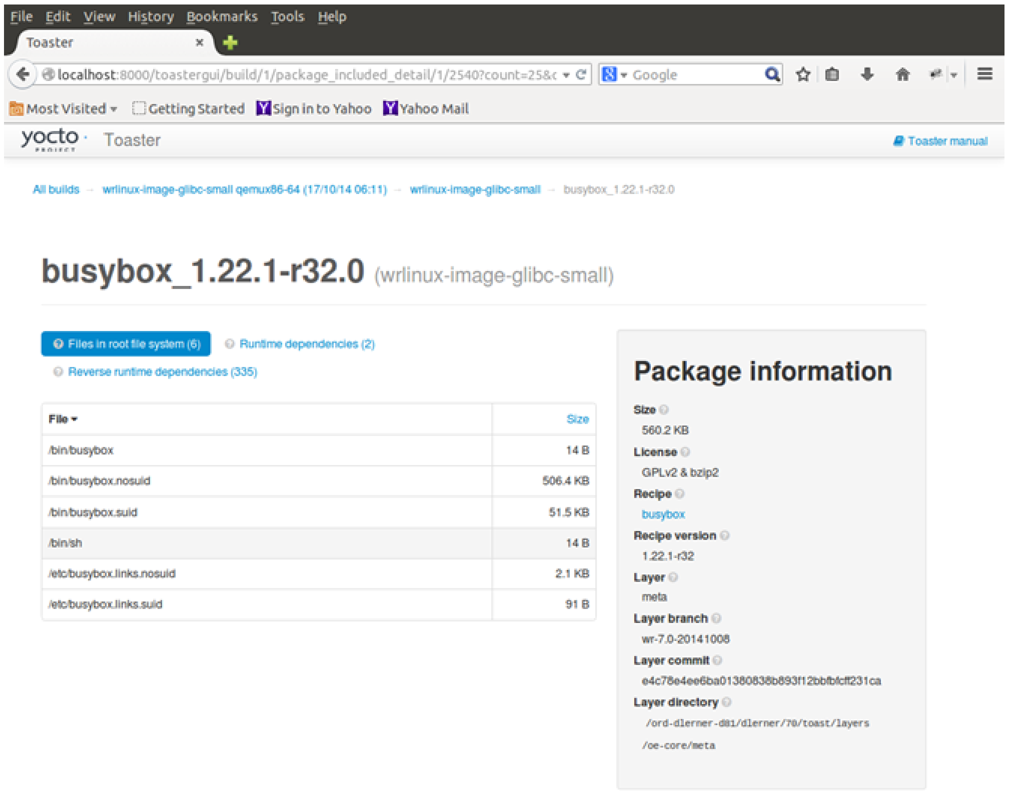
Build summary

Configuration
Machine
qemux86-64
Distro
wrlinux-small
Layers
extra-downloads
local
meta
meta-downloads
meta-networking
meta-oe-subset
meta-perl-subset
meta-python-subset
meta-webserver
oe-core-dl-1.5
qemux86-64
wr-base
wr-fixes
wr-kernel

Tasks
Total number of tasks
1876
Tasks executed
1686
Tasks not executed
190
Reuse
10 %

Recipes & Packages
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.


localhost:8000/toastergui/build/1/package_included_detail/1/2540?count=25&c

All builds → wrlinux-image-glibc-small qemux86-64 (17/10/14 06:11) → wrlinux-image-glibc-small → busybox_1.22.1-r32.0

busybox_1.22.1-r32.0 (wrlinux-image-glibc-small)

Files in root file system (6) | Runtime dependencies (2) | Reverse runtime dependencies (335)

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

Comparing and Verifying Platform Project Builds

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 with the Image Manifest and the Audit Tools on page 58. 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 with the Image Manifest and the Audit Tools

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

Prerequisites

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 Comparing and Verifying Platform Project Builds on page 57.

Procedure

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.

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**

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
```

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.

5 Managing the Package Revision Server

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.

This topic provides an overview for using the PR service. For a tutorial on managing many aspects of PR in your build environment, see the *Wind River Linux Build System Command Line Tutorials: Package Revision Server Tutorial*.

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 PR server sets the final version. The PR server starts at whatever value is set by the recipe. The default value is **r0**.

NOTE: The **feature/package-management** template adds the package-management features to the install package management tools, and preserves the package manager database on target. See the *Wind River Linux Platform Developer's Guide: Wind River Linux Build System Command Line Tutorials: Feature Templates in the Project Directory*.

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 the *Wind River Linux Build System Command Line Tutorials: Enabling Package Build History*.

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.

6 Optimizing the Build

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. .

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.

Build Acceleration 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 Command Line Tutorials: Creating and 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.

For a tutorial on using and securing a shared state cache server, see the *Wind River Linux Build System Command Line Tutorials: Shared State Cache Security Tutorial*.

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 *NATIVELSBSTRING*, 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 *NATIVELSBSTRING*. 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 *NATIVELSBSTRING* 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 *NATIVELSBSTRING* 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 *NATIVELSBSTRING* 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.

7 Migration

Build System Changes and Migrating your Project

Understanding the build system changes between Wind River Linux 4.x and 7.0 will help prepare you to complete your migration tasks.

This section describes changes to the Wind River Linux build system for version 7.0, from version 4.x. For information on new features available with 7.x, refer to the *Wind River Linux Release Notes*.

In general, configuring, building, and other workflow procedures remain the same in Wind River Linux 7.x as they were in Wind River Linux 4.x. However, there have been significant changes in the way the product is structured and in the build system itself with the introduction of the Yocto Project BitBake build system. For an overview of the changes between the BitBake and LDAT build systems, see Product Structure for Migration Purposes on page 70.

This includes the location of project-specific configuration files, layer configuration, and how the build system processes these files when you configure the platform project or build the file system.

Product Structure for Migration Purposes

Understanding the product structure changes between Wind River Linux 4.x and 7.x will help prepare you to better understand migration requirements.

Product Structure Changes Overview

With the introduction of Wind River Linux 7.x and the Yocto Project BitBake build system, the former Wind River Linux LDAT build system used by Wind River Linux 4.x significantly changes the structure of a configured and built platform project. These changes include:

Layer and template structure and content

> This structure is very different in that the Yocto Project build system relies on the concept of recipes and configuration files to define platform project configuration information. This change results in different layer and template directories, as well as how configuration information is specified for the build system. For a comparison, see BitBake and LDAT Comparison on page 71.

BSP structure and content

> Like layers and templates, the new build system uses recipes and configuration files to define BSP configuration information. This allows developers to append existing Wind River Linux 7.x BSPs with project-specific changes, simplifying the process of modifying a BSP to meet a given need. To learn how to modify an existing BSP, see the *Wind River Linux Kernel Developer's Guide:Kernel and BSP Changes*.

Replacing sysroots with a software development kit (SDK) for Linux and Windows development hosts

> The inclusion of a new SDK greatly simplifies application and package development, as once a platform project is built, the SDK is provided as an installable shell script. This script automatically installs the SDK, and sysroot information, and sources the development environment. As a result, the build paths and other elements of the SDK differ from their Wind River Linux 4.x counterparts. For additional information, see the *Wind River Linux User Space Developer's Guide: Application Development Changes*.

For information on the Wind River Linux 7.x installation and build environments, see the *Wind River Linux Getting Started: Installation Directory Structure* and the *Wind River Linux Platform Developer's Guide: Directory Structure for Platform Projects*.

BitBake and LDAT Comparison

Learn about the differences in the build systems and how it affects platform project migration.

BitBake and LDAT: Two Very Different Build Systems

While both build systems are based around the same core principle — creating a Wind River Linux target distribution using minimal configuration information — each uses a different approach to reach that goal. Some of the main differences are explained at a high level in Product Structure for Migration Purposes on page 70. With LDAT, much of the platform project configuration information was maintained in specific layer and template directories. This configuration information included Makefiles, patches, and packages for user space platform project elements, and kernel configuration fragments and patches for kernel or BSP-related elements.

With BitBake, all platform project configuration follows the same model in that configuration (***.conf**), recipes (***.bb**), and recipe append (***.bbappend**) files define the platform project's configuration. This is explained in the *Wind River Linux Platform Developer's Guide: Configuration Files and Platform Projects*.

With Wind River Linux 4.x, making changes, such as adding a new layer for testing purposes to a platform project, required that the project be configured and built separately. With Wind River Linux 7.x, it's possible to add and remove layers and rebuild the platform project by editing the project's ***projectDir*/bitbake_build/conf/bblayers.conf** configuration file. In addition, when changes to a platform project element require updating a recipe file, the ability to use recipe append (***.bbappend**) files to update related project elements helps simplify development.

These differences pose a challenge to platform project migration, though. The information in a typical application's or package's Makefile is now maintained in a recipe (***.bb**) file, in a layer specific to the BitBake development environment. To reuse platform project metadata, such as layers with applications and packages, you must first create a new platform project with Wind River Linux 7.x, and then import the layers, packages, and applications to it.

With Wind River Linux 7.x, the ***projectDir*/layers/wrlcompat** layer maintains the scripts necessary to use the Wind River Linux configure script used to create a new Wind River Linux platform project.

Build Environment Changes

Wind River Linux 7.x includes the following new build environment directories that were not included in Wind River Linux 4.x:

- **bitbake**
- **bitbake_build**
- **layers**
- **packages**
- **READMES**

For a description of the content of these directories, see the *Wind River Linux Platform Developer's Guide: Directory Structure for Platform Projects*.

In Wind River Linux 7.x, the **build** and **build-tools** directories contain links to directories in the **bitbake_build** directory and are created for convenience for those familiar with those directories in the LDAT build system. There is no longer a **build/INSTALL_STAGE/*packageName*** directory. The equivalent directory is now **build/*packageName*/package**.

In Wind River Linux 7.x, the **do_install** rule populates the **build/*packageName*/image** directory. The **do_package** rule moves the required files to the **build/*packageName*/package** directory, then the **build/*packageName*/package-split** directory based on a default set of filters. If necessary, you can override or append this in the recipe using the **${${PN}_FILES}** variable entry. Typically, an explicit packaging install rule, like the one in an LDAT spec file, is not required. For additional information, refer to the Open Embedded manual at http://www.embeddedlinux.org.cn/OEManual/recipes_packages.html.

In addition, the following directories from Wind River Linux 4.x no longer exist in Wind River Linux 7.x:

- **build-*lib***

  This directory only appeared in cases where the project supports and specifies multilibs. This information is now part of the build system by default.

- **filesystem/fs**

  This directory was previously used to add applications or packages to the target file system. The process for doing this has changed in Wind River Linux 7.x. For additional information, see the *Wind River Linux Platform Developer's Guide: Adding an Application to a Root File System with fs_final*.sh Scripts*.

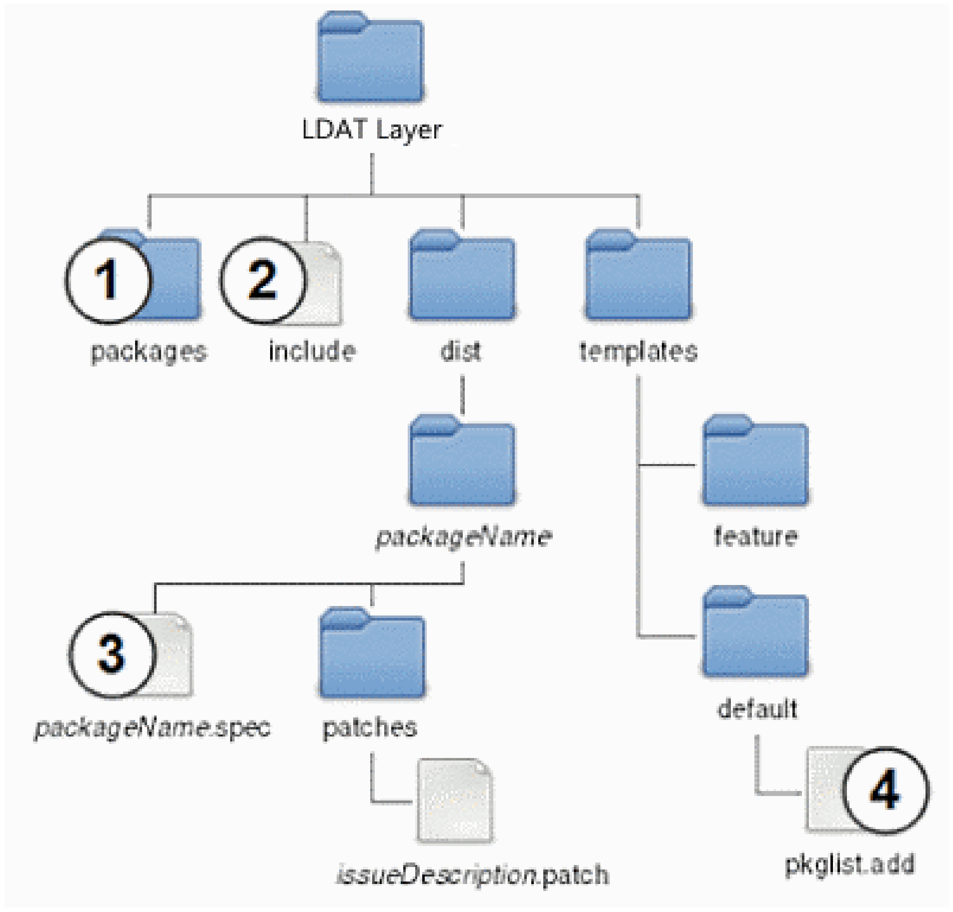
Layer Comparison

The layer requirements for BitBake are different from the requirements for LDAT. To import and/or modify a layer for use with Wind River Linux 7.x, it is important to understand these differences.

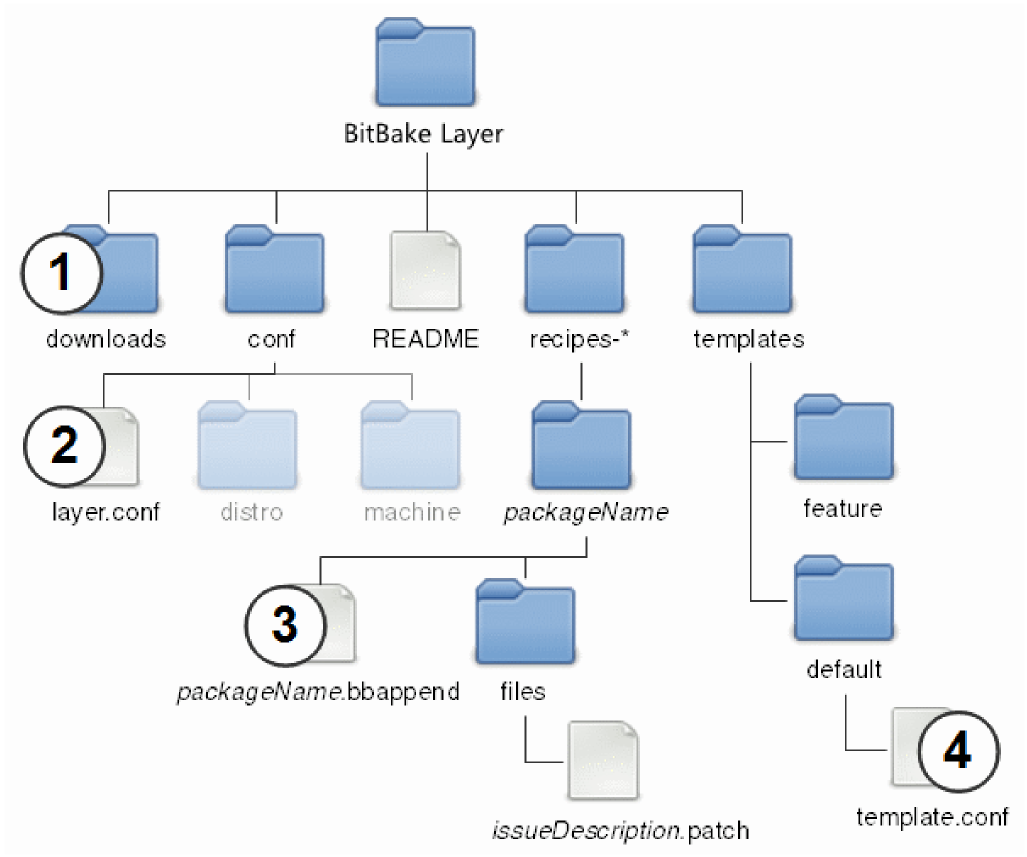
Layers in Wind River Linux 7.x are delivered as a git archive. You cannot see what they contain in the install directory without a git viewing tool. When you create a platform project, the layer is checked out into the platform project's **layers** directory. This is a new subdirectory of a Wind River Linux 7.x platform project. Once in the project, the content of the layer is visible as regular files. It is this project-local version of the layer that is used to build the platform project, and it does not update when you update your install directory with an updated version of Wind River Linux 7.x. To synchronize your project's layers with updates to the install directory, navigate to the platform project's top-level directory in a console and type the following:

```
$ make reconfig
```

The Wind River Linux 4.x layer is structured in the following manner:



In comparison, the Wind River Linux 7.x layer is structured as follows:



In these diagrams, the numerical references help explain similar information that exists in a Wind River Linux 4.x layer when compared to a Wind River Linux 7.x layer. For example:

1. The **packages** directory in Wind River Linux 4.x contained the packages required by the layer to function as intended. In Wind River Linux 7.x, this directory is now named **downloads**.

   The packages in Wind River Linux were primarily in source RPM (SRPM) packages (***.src.rpm** or ***.srpm**), a build driven by ***.spec** files that is not supported in Wind River Linux 7.x. The preferred migration is to extract the tar file from the SRPM into the Wind River Linux download directory and examine the spec file, and translate it and any patches and additional build steps it provides into a BitBake recipe. Once you have identified all the changes, you can retain the ***.src.rpm** file and move that into the download directory instead, and use it as a base for the BitBake recipe to apply the changes to. For an example of examining a ***.spec** file in this manner, see the *Wind River Linux Platform Developer's Guide: Updating a BitBake Recipe from an SRPM *.spec File*. This workflow is supported by the Import Package tool described in the *Wind River Linux Platform Developer's Guide: Importing an SRPM Package from the Web*. Additionally, some translation of the spec file into a BitBake recipe is provided by the **rpmparse.py** migration script (in the *Wind River Linux Platform Developer's Guide: Migration Script Reference*).

2. The **include** file provides a sequential list of items to process in the layer. The **include** file and other files with specific names placed in the root of a layer is replaced by the **layer.conf** file in Wind River Linux 7.x.

3. The *packageName*.**spec** file is used for SRPMS. It defines the build process for the package. With Wind River Linux 7.x, this information is defined in the *packageName*.**bbappend** file, and extended with files called *packageName*.**bbappend** and *packageName*.**inc**.
4. The **pkglist.add** file provides a list of packages to add to the build as part of including the layer in the build process. Typically, these packages are included in the layer's **packages** directory. With Wind River Linux 7.x, this information defined in the **pkglist.add**, **pkglist.delete** file and other specially named files placed in the template's root directory are replaced by the layer's **templates/default/template.conf** file.

Understanding this information will help you create and/or customize layers for use with your Wind River Linux 7.x platform project.