WIND™

AN INTEL COMPANY

WIND RIVER® LINUX

WIND RIVER LINUX KERNEL COMMAND-LINE TUTORIALS

7.0

Corporate Headquarters

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

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

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

Linux

Wind River Linux Kernel Command-Line Tutorials

6 November 2015

Contents

1

Kernel Configuration and Patching with Fragments

Kernel Configuration and Patching with Fragments Tutorial

Learn about configuring and patching a Wind River Linux kernel with fragments through a series of self-paced procedures.

Overview

When you have completed this tutorial, you will:

- understand how to use fragments to specify changes to the kernel's configuration file
- use the ***projectDir*/layers/local** directory and **.bbappend** file to specify the location of the configuration fragment
- clean and configure the updated kernel package
- rebuild the kernel and file system
- test your kernel updates on a target platform

Tutorial Requirements

To complete this tutorial, you require a previously configured and built platform project. For additional information on creating a platform project, see *Wind River Linux Getting Started Command Line Tutorials: Creating and Configuring a Platform Project*.

Kernel Configuration and Patching with Fragments

Kernel configuration can be done conveniently using configuration fragments, which are small files that contain kernel configuration options in the syntax of the original kernel's **.config** configuration file.

Kernel fragments capture specific changes to the kernel's configuration file. Creating a basic infrastructure inside the local layer enables them.

Once the infrastructure is in place, the BitBake build system will incorporate the kernel fragments into the kernel configuration process to build the corresponding kernel image and associated kernel modules.

In this section, you will learn to reconfigure the Linux kernel to make some changes on the kernel modules that are installed by default.

The changes you will make include:

- Removing the **floppy** and **parport** (parallel port) modules, assuming that they are not necessary for the intended target.
- Turning the **minix** kernel module into a static kernel feature, so that its functionality is provided by the kernel image itself.
- Add the **pcspkr** (PC speaker) module.

Once complete, you will rebuild the kernel and file system, reboot the emulated target, and verify that your changes have been applied.

Select *Populate the Local Layer with the Required Subdirectories* on page 2 to begin the tutorial procedures.

Populate the Local Layer with the Required Subdirectories

Populate the local layer as part of configuring the Linux kernel with fragments.

Prerequisites

To perform this procedure, you must have a previously configured and built platform project. For additional information, see *Wind River Linux Platform Developer's Guide: Configuring and Building a Complete Run-Time Image*.

Populate the local layer with subdirectories.

From the platform project's main directory, enter the following command to create the required directories to maintain your kernel fragments:

```
$ mkdir -p layers/local/recipes-kernel/linux/linux-windriver
```

The basic directory structure necessary to support configuration fragments is dictated by the content of the *BBFILES* variable inside the ***projectDir*/layers/local/conf/layer.conf** file. See *Wind River Linux Platform Developer's Guide: Directory Structure for Platform Projects*.

More specifically, the element ***${LAYERDIR}*/recipes-*/*/*.bbappend** in this variable determines where the **.bbappend** files will be searched for. The part of the command line above that reads: **recipes-kernel/linux** complies with this pattern.

The **linux-windriver** subdirectory is used to further localize kernel configuration files for the kernels provided by Wind River and it is named after the Linux kernel package itself.

Postrequisites

Once the required directory structure is in place, you will need to create the kernel's **.bbappend** file to specify the location of the kernel configuration fragment as described in *Create the Kernel's BitBake Append (.bbappend) File* on page 3.

Create the Kernel's BitBake Append (.bbappend) File

The kernel's **.bbappend** file specifies the location of the kernel configuration fragment to the build system.

Prerequisites

This procedure requires that you have populated the ***projectDir*/layers/local** directory with the sub-directories required for patching the kernel as described in *Populate the Local Layer with the Required Subdirectories* on page 2.

Step 1 Create the **.bbappend** file for the kernel.

Run the following command to create the kernel's **.bbappend** file:

```
$ vi layers/local/recipes-kernel/linux/linux-windriver_3.14.bbappend
```

Step 2 Add the following default lines of code:

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://config_baseline.cfg"
```

The variable *FILESEXTRAPATHS_prepend* extends the search path of BitBake to include a directory named after the package being processed, **PN** for package name under the current directory, **THISDIR**. In this example, **PN** is **linux-windriver** and this explains why we originally created a sub-directory with this name.

The name of the kernel fragment is added to the BitBake variable *SRC-URI*, which holds the list of configuration files, of any kind, to be processed when building the project.

The syntax **file://config_baseline.cfg** is used to tell BitBake that the configuration fragment is to be found as a regular text file inside the layer, and not for example, through a source version control system somewhere else. This file is where you will add fragments that make changes to the kernel.

Postrequisites

The presence of the **.bbappend** file in the project directory will ensure that the build system is aware that a kernel configuration fragment requires processing as part of the platform project build. With this file in place, you must ensure that the configuration fragment contains the changes you want to make to the kernel as described in *Create the Kernel's Configuration Fragment* on page 4.

Create the Kernel's Configuration Fragment

Create a kernel fragment as part of configuring the Linux kernel with fragments.

Prerequisites

This procedure requires that you have created a **.bbappend** file for patching the kernel as described in *Create the Kernel's BitBake Append (.bbappend) File* on page 3.

Step 1 Create the configuration fragment for the kernel.

Create the kernel's **config_baseline.cfg** file.

```
$ vi layers/local/recipes-kernel/linux/linux-windriver/config_baseline.cfg
```

Step 2 Configure kernel fragments for this example.

The following lines add fragment configuration:

```
# CONFIG_BLK_DEV_FD is not set
# CONFIG_PARPORT is not set
CONFIG_MINIX_FS=y
CONFIG_INPUT_MISC=y
CONFIG_INPUT_PCSPKR=m
```

The configuration fragment(s) in this example have the same syntax as the **.config** file for the kernel. Note that we added the statement:

```
CONFIG_INPUT_MISC=y
```

which is a prerequisite for the option **CONFIG_INPUT_PCSPKR** to become available.

Also note that lines starting with the # character are not comments but indicate instead that a particular kernel feature is to be disabled.

At this moment the layer structure to support kernel configuration fragments should look like this:

```
layers/local/recipes-kernel/ |- linux |- linux-windriver |- config_baseline.cfg |-
linux-windriver_3.14.bbappend
```

Postrequisites

Now that you have created the configuration fragment, it must be added to the build to ensure the kernel features will work on the target platform. Prior to building the kernel, it is necessary to clean and configure the kernel package as described in *Clean up the Linux Kernel Package and Optionally Configure the Package* on page 4.

Clean up the Linux Kernel Package and Optionally Configure the Package

Clean up the Linux kernel package as part of configuring the Linux kernel with fragments.

Cleaning up the **linux-windriver** kernel package first is a necessary step to force the build system to subsequently reload all associated configuration files. You should do this every time you make changes to your kernel configuration fragments and prior to rebuilding the kernel package.

While optional, configuring the Linux kernel package ensures that configuration will happen automatically when you rebuild the package. Since configuration is done much faster than rebuilding. the advantage of doing the configuration step manually is that you can verify very quickly that the changes specified in the configuration fragment are correct by inspecting the generated configuration file of the kernel located at:

***projectDir*/build/linux-windriver-version/linux-qemux86-64-standard-build/.config**

Prerequisites

This procedure requires that you have previously created a kernel configuration fragment as described in *Create the Kernel's Configuration Fragment* on page 4.

Step 1 Clean up the Linux kernel package.

```
$ make linux-windriver.clean
```

Step 2 Optionally, configure the Linux kernel.

```
$ make linux-windriver.configure
```

Postrequisites

With the kernel package cleaned and configured to include the new configuration fragments, it is time to build the kernel and platform project file system as described in *Rebuild the Linux Kernel Package and File System* on page 5.

Rebuild the Linux Kernel Package and File System

Rebuild the kernel and file system as part of configuring the Linux kernel with fragments.

Once the kernel configuration fragment has been created, and the **linux-windriver** kernel package has been cleaned and configured as described in *Clean up the Linux Kernel Package and Optionally Configure the Package* on page 4, follow this procedure to rebuild the kernel.

Step 1 Rebuild the Linux kernel package and file system.

Run the following command from the platform project's directory to rebuild the kernel package:

```
$ make linux-windriver.rebuild
```

Once complete, the new **linux-windriver** package is available containing the modified kernel image to be used in the target.

Step 2 Rebuild the file system.

```
$ make
```

This command updates the root file system to include the new structure of kernel modules to be loaded on the target. Note that if your configuration fragments do not modify the current or default kernel modules then you do not need to rebuild the root file system.

For example, if the only line in the configuration fragment above had been **CONFIG_PRINTK_TIME=y**, then only the kernel image would have been modified when rebuilding the kernel package, and the root but the root file system would have remained the same.

Postrequisites

At this point, the changes you made to the kernel with your fragments are part of the platform project build. To verify that the changes are successful, you can run them on an emulated target as described in *Run the Emulated Target* on page 6.

Run the Emulated Target

Run the emulated target after you have configured the Linux kernel with fragments.

This procedure tests whether the kernel configuration fragments created in *Create the Kernel's Configuration Fragment* on page 4 function as expected on a simulated target platform.

Prerequisites

To complete this procedure, you must have previously configured the kernel package and rebuilt the file system as described in *Rebuild the Linux Kernel Package and File System* on page 5.

Step 1 Verify that the **pcspkr** module still exists on the target.

Run the following command from the platform project directory:

```
$ make start-target
```

Step 2 Log in to the system.

Use the user name **root** with password **root**.

Step 3 Verify the status of the **floppy**, **parport**, and **pcspkr** modules.

```
root@qemux86-64: ~# lsmod
```

The system should return the following:

```
Not tainted
pcspkr 2030 0 - Live 0xffffffffa0002000
```

The module list shows that the **floppy** and **parport** modules are no longer present and that the **pcspkr** module is active now.

Step 4 Review how the **pcspkr** module loads.

Run the following command on your qemux86-64 target:

```
$ cat /usr/lib64/udev/rules.d/60-persistent-input.rules | grep pcspkr
```

The system should return the following:

```
DRIVERS=="pcspkr",ENV{.INPUT_CLASS}="spkr"
```

Note that the automatic loading of modules is handled by the udev infrastructure.

Step 5 Verify that the minix file system is still supported.

```
root@qemux86-64: ~# cat /proc/filesystems |grep minix
```

The system should return the following:

```
minix
```

Support for the minix file system is still available but this time it is built into the Linux kernel image itself.

Step 6 Shut down the target.

Enter the following command in the emulator console:

```
root@qemux86-64: ~# halt
```

Kernel Configuration and Patching with Fragments Summary

In this tutorial, you learned how to use fragments to add, modify, and remove kernel options, and how to use **.bbappend** file to specify the file used to maintain the kernel configuration changes. In addition, you learned how to clean and rebuild the kernel package, and how to test your changes on a live target.

2

Kernel Configuration with menuconfig

Kernel Configuration with menuconfig Tutorial

Learn about kernel configuration with menuconfig.

Overview

When you have completed this tutorial, you will:

- learn how to view existing the kernel modules on the target platform
- remove kernel modules from the existing kernel configuration
- turn an existing kernel module into a static kernel feature
- add a new kernel module
- rebuild the kernel and file system
- test your kernel updates on a target platform

Tutorial Requirements

The following procedures require a configured and built platform project, with the kernel modules added in the *Kernel Configuration and Patching with Fragments Tutorial* on page 1 tutorial.

Kernel Configuration with menuconfig

menuconfig is a basic configuration mechanism provided by the Linux kernel build system that provides a menu-based access to the different kernel options.

In this section, you will learn to reconfigure the Linux kernel to make some changes on the kernel modules which are installed by default.

The changes you will make include:

- Removing the **floppy** and **parport** (parallel port) modules, assuming that they are not necessary for the intended target.
- Turning the **minix** kernel module into a static kernel feature, so that its functionality is provided by the kernel image itself.
- Adding the **pcspkr** (PC speaker) module.

Once complete, you will rebuild the kernel and file system, reboot the emulated target, and verify that your changes have been applied.

Select *Viewing Installed Kernel Modules* on page 10 to begin the tutorial procedures.

Viewing Installed Kernel Modules

Before you make changes to the kernel configuration, review the existing kernel modules to ensure that the configuration meets your expectations.

Viewing kernel modules is necessary to understand whether they need to be enabled or disabled as part of your overall project requirements. In this procedure, you will verify that the **floppy**, **minix**, and **parport** modules are running on the target system.

Prerequisites

The following procedure requires a configured and built platform project, with the kernel modules added in the *Kernel Configuration and Patching with Fragments Tutorial* on page 1 tutorial.

Step 1 Boot the emulated quemux86-64 target that you have built in the previous sections.

a) Launch the target.

From the platform project's directory, run the following command:

```
$ make start-target
```

b) Log in as user **root**, and password **root**.

You should have now access to the command line shell on the target.

Step 2 List the kernel modules installed on the target.

Run the following command from the target console:

```
root@qemux86-64:~# lsmod
```

The console should return the following output:

```
Not tainted
    parport 23894 1 parport_pc, Live 0xffffffffa0017000
    floppy 60578 0 - Live 0xffffffffa0022000
```

```
parport_pc 18367 0 - Live 0xffffffffa0038000
minix 29971 0 - Live 0xffffffffa0042000
```

This output represents the list of kernel modules loaded in the system. In this example, we will assume that **floppy** and **parport** (parallel port) modules are not required, so we will remove them. We will also integrate the **minix** module into the kernel image itself.

Step 3 Verify support for the minix file system.

Run the following command from the target console:

```
root@qemux86-64:~# cat /proc/filesystems |grep minix
```

The console should return the following output to indicate support for the minix file system:

```
minix
```

Note that since the **minix** module is already loaded, it is expected that the kernel supports it.

Step 4 Shutdown the emulated target.

Run the following command from the target console:

```
root@qemux86-64:~# shutdown now
```

This will cleanly shutdown the console window so you can make changes to the kernel's configuration.

Postrequisites

Now that you have verified the kernel modules, you must launch **menuconfig** to make changes to the configuration as described in *Launching menuconfig and GUI Configuration Tools* on page 11.

Launching menuconfig and GUI Configuration Tools

In order to use **menuconfig** or one of the other available tools for kernel configuration, you must launch them from the platform project directory.

The following procedure provides commands for launching **menuconfig**, along with other GUI Linux kernel configuration tools such as **xconfig** and **gconfig**.

NOTE: To use **xconfig** or **gconfig**, listed in the following commands, you must have the QT toolkit and QT development tools installed on your host.

For example, with a Debian-based workstation, you could use the command: **sudo apt-get install qt4-dev-tools qt4-qmake** to install QT.

Prerequisites

This procedure requires a previously built platform project.

Launch the **menuconfig** configuration tool for the kernel.

Enter one of the following commands from the platform project directory to launch the kernel configuration menu:

Options	Description
Run menuconfig in a separate terminal window:	`$ make linux-windriver.menuconfig`
Run the graphical xconfig interface to menuconfig:	`$ make linux-windriver.xconfig`
Run the graphical gconfig interface to menuconfig:	`$ make linux-windriver.gcconfig`

After a few seconds, a new terminal window appears with the kernel configuration menu.

Postrequisites

Now that menuconfig is running, you can make changes to the configuration as described in:

- *Removing Kernel Modules* on page 13
- *Turning an Existing Kernel Module into a Static Feature* on page 14
- *Adding a Kernel Module* on page 12

Adding a Kernel Module

Use **menuconfig** to add kernel modules to provide additional kernel features for your platform.

By default, miscellaneous device support is disabled. To add PC speaker support, you must enable it.

Prerequisites

The following procedure requires that **menuconfig** be running from the platform project directory as described in *Launching menuconfig and GUI Configuration Tools* on page 11.

Step 1 Navigate to the **pcspkr** module.

a) From the top kernel configuration menu, select **Device Drivers** > **Input device support**, then press **SPACE**.

b) Select **Miscellaneous devices**, then press **SPACE** to enable the sub-menu.

c) Press **ENTER** to view the submenu.

Step 2 Set **PC Speaker support** as a module.

Select **PC Speaker support** and press **SPACE** to add the PC Speaker support option as a module.

The marker is now a letter **M** indicating that the **PC Speaker support** option is a module.

Step 3 Return to the main configuration window.

Select **Exit** three times to return to the top level list of configuration options.

Postrequisites

Once you have added a kernel module, you can continue to make kernel configuration changes as described in *Turning an Existing Kernel Module into a Static Feature* on page 14 and *Removing Kernel Modules* on page 13.

If you are finished making changes, you will need to save your changes and rebuild the kernel as described in *Saving the Configuration and Rebuilding the Kernel* on page 14.

Removing Kernel Modules

Once you have verified that the kernel modules are enabled in the kernel, you can remove them with menuconfig.

Prerequisites

The following procedure requires that menuconfig be running from the platform project directory as described in *Launching menuconfig and GUI Configuration Tools* on page 11.

Step 1 Remove the **floppy** module from the kernel configuration.

a) From the top kernel configuration menu, select `**Device Drivers** > **Block devices** > **Normal floppy disk support** .

Normal floppy disk support should be listed with a letter **M** marker indicating that it is to be compiled as a module.

b) Press **SPACE** twice to remove this module from the build.

The marker is now blank indicating that the floppy module is not selected.

c) Select **Exit** at the bottom menu twice, using TAB or left/right arrow keys, to return to the top level list of configuration options.

Step 2 Remove the **parport** module from the kernel configuration.

a) From the top kernel configuration menu, select **Device Drivers** > **Parallel port support** .

Parallel port support should be listed with a letter `M` marker indicating that it is to be compiled as a module.

b) Press **SPACE** twice to remove this module from the build.

c) Select **Exit** at the bottom menu to return to the top level list of configuration options.

Postrequisites

Once you have removed a kernel module, you can continue to make kernel configuration changes as described in *Turning an Existing Kernel Module into a Static Feature* on page 14 and *Adding a Kernel Module* on page 12.

If you are finished making changes, you will need to save your changes and rebuild the kernel as described in *Saving the Configuration and Rebuilding the Kernel* on page 14.

Turning an Existing Kernel Module into a Static Feature

When you are sure that a kernel module should be a permanent part of your configuration, you can turn it into a static feature with **menuconfig**.

When developing and testing a platform and its kernel, kernel modules provide a means to add and remove features with little overhead. Towards the end of the development cycle, you may choose to make a module a static feature, which removes the module's startup and shutdown scripts from the kernel configuration.

Prerequisites

The following procedure requires that menuconfig be running from the platform project directory as described in *Launching menuconfig and GUI Configuration Tools* on page 11.

Step 1 Navigate to the **minix** kernel configuration option.

From the top kernel configuration menu, select **File systems** > **Miscellaneous filesystems** > **Minix file system support**.

Minix file system support should be listed with a letter **M** marker indicating that it is to be compiled as a module.

Step 2 Turn the **minix** module into a static feature.

Press **SPACE** once to turn it into a static kernel feature.

The marker is now an asterisk (*), indicating that the **minix** option is configured as a static kernel feature.

Step 3 Return to the main configuration window.

Select **Exit** at the bottom menu twice, pressing **TAB** to return to the top level list of configuration options.

Postrequisites

Once you have turned a kernel module into a static feature, you can continue to make kernel configuration changes as described in *Adding a Kernel Module* on page 12 and *Removing Kernel Modules* on page 13.

If you are finished making changes, you will need to save your changes and rebuild the kernel as described in *Saving the Configuration and Rebuilding the Kernel* on page 14.

Saving the Configuration and Rebuilding the Kernel

Once you are finished making kernel configuration changes, you must save them and rebuild the kernel to inlude the changes in your target platform image.

Prerequisites

The following procedure requires that menuconfig be running from the platform project directory as described in *Launching menuconfig and GUI Configuration Tools* on page 11.

In addition, it assumes that you have made changes to the kernel configuration as described in *Adding a Kernel Module* on page 12, *Removing Kernel Modules* on page 13, or *Turning an Existing Kernel Module into a Static Feature* on page 14.

Step 1 Save the new kernel configuration.

a) From the top kernel configuration menu, select **Exit**.

b) When prompted to save your new configuration, select **Yes** to finish the configuration session.

Step 2 Rebuild the kernel image and modules.

Run the following command from the platform project directory to rebuild the kernel image and modules:

```
$ make linux-windriver.rebuild
```

Step 3 Rebuild the root file system.

Run the following command from the platform project directory to rebuild the root file system and update the kernel modules as necessary:

```
$ make
```

Postrequisites

At this point, the changes you made to the kernel with **menuconfig** are part of the platform project build. To verify that the changes are successful, you can run them on an emulated target as described in *Testing the New Configuration on an Emulated Target* on page 15.

Testing the New Configuration on an Emulated Target

Once changes made with **menuconfig** are part of the platform project build, you can test them on an emulated target.

Prerequisites

The following procedure requires the kernel configuration changes made throughout this tutorial, specifically the rebuilt kernel and file system as described in *Saving the Configuration and Rebuilding the Kernel* on page 14.

Step 1 Boot the emulated target to test your new kernel configuration.

a) Launch the target.

```
$ make projectDir/start-target
```

b) After the target window boots, login as user **root** with password **root**.

Step 2 Verify the status of the **floppy**, **parport**, and **pcspkr** modules:

```
root@qemux86-64:~# lsmod
```

```
Not tainted
pcspkr 2030 0 - Live 0xffffffffa0002000
```

The module list shows that the **floppy** and **parport** modules are no longer present and that the **pcspkr** module is active now.

Step 3 See how the **pcspkr** module loads.

The udev infrastructure manages automatic module loading.

On your qemux86-64 target, type the following command:

```
$ cat /usr/lib64/udev/rules.d/60-persistent-input.rules | grep pcspkr
```

The system should return the following:

```
DRIVERS=="pcspkr",ENV{.INPUT_CLASS}="spkr"
```

Step 4 Verify that the minix file system is still supported:

```
root@qemux86-64:~# cat /proc/filesystems |grep minix
```

The system should return the following:

```
minix
```

Support for the minix file system is still available but this time it is built into the Linux kernel image itself.

Kernel Configuration with menuconfig Summary

In this tutorial, you learned how to use **menuconfig** to add, modify, and remove kernel options, and how to view installed kernel modules on a target platform. In addition, you learned how to rebuild the kernel package, and how to test your changes on a live target.

3

Kernel Module Configuration with make Rules

Kernel Module Configuration with make Rules Tutorial

Learn about kernel module configuration using **make** command rules.

Overview

When you have completed this tutorial, you will:

- learn how to view existing kernel modules in your platform project build
- add a new kernel module with the **make** command
- remove the kernel module from the platform project
- test your kernel module changes on a target platform

Tutorial Requirements

The following procedures require a configured and built platform project, with the kernel modules added in the *Kernel Configuration and Patching with Fragments Tutorial* on page 1 tutorial.

Specifically, this tutorial requires that the **pcspkr** module is part of the platform project build.

Kernel Module Configuration with make Rules

The **make** command provides a simplified means to add or remove selected kernel modules as needed.

After you build your project, all configured kernel modules become available for use with the **make** command, as detailed in this section.

After an initial build of the file system completes, all configured kernel modules become available as pre-compiled binaries inside your project's working space. The first thing to do is to determine which modules are available, and then use the **make** options to add or to remove selected modules.

In this example, once you determine which kernel modules are available, you are going to add the **pcspkr** module, verify that it is loaded in the target, and then remove it, using make commands.

You may have added the **pcspkr** module already using the **menuconfig** method as described in *Kernel Configuration with menuconfig Tutorial* on page 9. In this tutorial, that module will be added as a project package and not directly as a kernel option as was done before.

Select *Determining Available Kernel Modules in a Platform Project Build* on page 18 to begin the tutorial procedures.

Determining Available Kernel Modules in a Platform Project Build

Before you can use make rules to add or remove kernel modules, you must first determine whether the module is part of your platform project build.

Because the **make** command can only add kernel modules that are part of the platform project build, identifying which modules that are available will ensure that the **make** command will be successful when you use it to add a module.

Determine the kernel modules available in your platform project.

Run the following command from the platform project directory to list and sort these files:

```
$ find bitbake_build/tmp/deploy/rpm | grep kernel-module- | \

perl -p -i -e 's/.*(kernel-module-.*)-3.*/$1/' | sort
```

The result will be an alphabetically sorted list of all available modules already pre-compiled and ready to be used.

Kernel modules are packaged as individual files in the following directory:

***projectDir*/bitbake_build/tmp/deploy/rpm**

Their file names all start with the **kernel-module-** prefix.

Postrequisites

Now that you have verified the available kernel modules, you can add them as described in *Adding a Kernel Module with make Rules* on page 19.

Adding a Kernel Module with make Rules

Using a single **make** command with options, you can add a kernel module to add additional functionality to a platform project image.

Once you have verified that a kernel module in included in your platform project as described in *Determining Available Kernel Modules in a Platform Project Build* on page 18, use the following procedure to add it to the platform project's target file system.

Step 1 Add the **pcspkr** module to the build.

As stated in the tutorial requirements, the following command assumes the **pcspkr** module is available at this point because it was built previously in *Kernel Configuration and Patching with Fragments* on page 2

a) Add the module.

```
$ make kernel-module-pcspkr.addpkg
```

b) Verify that the kernel module package has been added.

```
$ cat layers/local/recipes-img/images/wrlinux-image-file-system.bbappend
```

In this example, *file-system* refers to the root file system used to configure your platform project. If you configured your platform project to use a glibc-small file system, the command would be:

```
$ cat layers/local/recipes-img/images/wrlinux-image-glibc-small.bbappend
```

The system will return the following output, after the line that declares #### **END Auto Generated by configure** ####:

```
#### END Auto Generated by configure ####
IMAGE_INSTALL += "kernel-module-pcspkr"
```

This indicates that the package will be included in the build.

c) Rebuild the root file system.

```
make
```

Step 2 Verify the **pcspkr** module exists on the target.

a) Launch the platform project image in an emulator.

```
make start-target
```

b) Once the emulator finishes booting, login as user **root** with password **root**.

c) Verify that the **pcspkr** module was added to the target.

```
root@qemux86-64: ~# lsmod
```

The system should return the following, indicating that the pcspkr module was added:

```
Not tainted
    pcspkr 2030 0 - Live 0xffffffffa0002000
```

d) See how the **pcspkr** module loads.

```
$ cat /usr/lib64/udev/rules.d/60-persistent-input.rules | grep pcspkr
```

The system should return the following:

```
DRIVERS=="pcspkr",ENV{.INPUT_CLASS}="spkr"
```

Wind River Linux **make** commands simplify the process for adding kernel modules as described in this procedure. You can remove kernel modules in a similar manner. For additional information, see *Removing a Kernel Module with make Rules* on page 20.

Removing a Kernel Module with make Rules

Using a single **make** command with options, you can remove a kernel module from a platform project image.

Once you have added a kernel module in included in your platform project as described in *Adding a Kernel Module with make Rules* on page 19, use the following procedure to remove it from the platform project's target file system.

The following procedure removes the **pcskpr** module.

Step 1 Remove the **pcspkr** module package.

Run the following command in the ***projectDir***:

```
$ make kernel-module-pcspkr.rmpkg
```

Step 2 Clean and rebuild the kernel image.

```
$ make kernel-module-pcspkr.clean
```

This updates the set of available kernel modules, removing the **pcspkr** module in the process.

Step 3 Verify the **pcspkr** module is removed from the build.

a) Launch the platform project in an emulator.

Run the following command in the ***projectDir***:

```
$ make start-target
```

b) After the emulator finishes booting, login as user **root** with password **root**.

c) Verify that the **pcspkr** module is removed from the target.

Run the following command on the target:

```
root@qemux86-64: ~# lsmod
```

The system should return the following, indicating that the **pcspkr** module was removed:

```
Not tainted
```

Notice that the **pcspkr** module no longer loads or is present.

d) Shut down the target.

```
root@qemux86-64: halt
```

Wind River Linux **make** commands simplify the process for removing kernel modules as described in this procedure. You can add kernel modules in a similar manner. For additional information, see *Adding a Kernel Module with make Rules* on page 19.

Kernel Module Configuration with make Rules Summary

In this tutorial, you learned how to use **make** rules to add and remove kernel options, and how to view the available kernel modules in your platform project. In addition, you learned how to rebuild the kernel package, and how to test your changes on a live target.

4

Creating Alternate Kernels from kernel.org Source

Creating Alternate Kernels from kernel.org Source Tutorial

Learn how to use kernel.org source to create an alternate kernel for your target platform.

Overview

When you have completed this tutorial, you will:

- update the platform project's **bblayers.conf** file to add development support
- create a **.bbappend** file to include kernel changes in your platform project build
- build the kernel

Tutorial Requirements

The following tutorial requires a configured and built platform project, as described in the *Wind River Linux Getting Started Command Line Tutorials: Creating and Configuring a Platform Project*.

Creating Alternate Kernels from kernel.org Source

Wind River provides the capability to build arbitrary git-based kernel sources using a development-only recipe. This recipe uses the Yocto infrastructure to clone and build directly from the desired kernel repository, starting from a user-specified tag and complete configuration.

NOTE: Only the kernel version supplied with Wind River Linux is validated and supported. Using any other kernel version is not covered by standard support.

This procedure is therefore suitable only for projects that are not under Wind River standard support, such as a Proof of Concept. It is expected that this procedure will build without errors with most BSPs, but it is unlikely the resulting kernel will boot without further configuration and patches.

Step 1 Update the platform project's **bblayers.conf** file to add kernel development support.

In a previously created Wind River Linux Platform project based on a standard kernel, add *projectDir*/**layers/wr-kernel/kernel-dev** to the file *projectDir*/**bitbake_build/conf/bblayers.conf**. This makes the **linux-windriver-custom** recipe available to the build.

For example:

```
$ echo 'BBLAYERS += "${WRL_TOP_BUILD_DIR}/layers/wr-kernel/kernel-dev" '  \
 >> projectDir/bitbake_build/conf/bblayers.conf
```

Step 2 Create a **.bbappend** file in the local layer of your build.

```
$ cd projectDir/layers/local/
$ mkdir -p recipes-kernel/linux
$ cd recipes-kernel/linux/
$ echo 'FILESEXTRAPATHS := "${THISDIR}/${PN}"' >> linux-windriver-custom.bbappend
```

Step 3 Update the **SRCREV** for the kernel version being built.

NOTE: Kernels revisions from 3.16 and greater have different build rules and cannot be built using Wind River Linux recipes.

This is the git hash of a tag in the kernel.org tree. The kernel will be cloned from the kernel.org git repository so it is necessary to have downloading enabled in your **local.conf** file.

For example, to build the Linux 3.15 kernel the tag is available in the first command line. Run each command to add all required SRCREV additions to your **linux-windriver-custom.bbappend** file.

```
$ echo 'SRCREV = "1860e379875dfe7271c649058aeddffe5afd9d0d"' >> linux-windriver-
custom.bbappend
$ echo 'SRCREV_machine = "${SRCREV}"' >> linux-windriver-custom.bbappend
$ echo 'SRCREV_meta = "${SRCREV}"' >> linux-windriver-custom.bbappend
```

To obtain a **SRCREV** for a local repository, you must first clone the repository, as described in *Wind River Linux Kernel and BSP Developer's Guide: Custom Kernel Branch Maintenance*. Once you have a local repository, navigate to the local location of the cloned repository and enter:

```
$ git whatchanged kernel_branch
```

The output will provide the **SRCREV** for the kernel revision in the branch.

Step 4 Make the recipe compatible with your machine.

```
$ echo 'COMPATIBLE_MACHINE = "${MACHINE}"' >> linux-windriver-custom.bbappend
```

Step 5 Add kernel-specific configuration and patches.

When building a particular kernel version, you will also need to add kernel configuration and patches that are specific to the new kernel version. These additional files can be placed in the local layer recipe directory you have just created.

For example, to add a **config/defconfig** fragment for the board to the **SRC_URI**:

```
$ mkdir -p linux-windriver-custom
$ cp /path_to_my_custom_defconfig linux-windriver-custom/defconfig
$ echo 'SRC_URI += " file://defconfig"' >> linux-windriver-custom.bbappend
```

Step 6 Optionally, add the location of the locally cloned kernel repository if you plan to build from a local git repository.

```
$ echo 'SRC_URI = "git:///path_to_local_git_repository/
linux.git;protocol=file;nocheckout=1"' >> linux-windriver-custom.bbappend
```

Once all additions are made to the **linux-windriver-custom.bbappend** file, it should contain the following content:

```
$ cat linux-windriver-custom.bbappend

FILESEXTRAPATHS := "${THISDIR}/${PN}"
COMPATIBLE_MACHINE = "${MACHINE}"
LINUX_VERSION = "3.15"
SRCREV = "1860e379875dfe7271c649058aeddffe5afd9d0d"
SRCREV_machine = "${SRCREV}"
SRCREV_meta = "${SRCREV}"
SRC_URI += " file://defconfig"
```

If you are using a local git repository to build your kernel, the file will also include the following line:

```
SRC_URI = "git:///path_to_local_git_repository/linux.git;protocol=file;nocheckout=1"
```

Step 7 Move to the root of your project and edit your **local.conf** file.

```
$ cd ../../../..
$ vi local.conf
```

Change your **PREFERRED_PROVIDER_virtual/kernel_*BSP_name*= "linux-windriver"** setting to refer to the **linux-windriver-custom** recipe, and save the file. For example:

```
PREFERRED_PROVIDER_virtual/kernel_qemuppc = "linux-windriver-custom"
```

NOTE: For CGL kernels you must add a flag in the **local.conf** file by adding the following line: **KERNEL_FEATURES_CLEAR="t"**

Step 8 Build the kernel.

Run the following command from the top-level folder in the ***projectDir***:

```
$ make linux-windriver-custom
```

If your kernel is compatible, you can build it into your file system image using:

```
$ make
```

NOTE: This procedure only replaces the kernel and not the file system components. The kernel-headers package that is exported to the SDK sysroot remains unchanged.

Creating Alternate Kernels from kernel.org Source Summary

In this tutorial, you learned how to update a platform project's bblayers.conf file to add development support. In addition, you learned how to create a custom kernel **.bbappend** file and update it to include all of the requirements necessary for making it possible for the new kernel to build successfully in your platform project.

5

Exporting Custom Kernel Headers

Exporting Custom Kernel Headers to the Sysroot Tutorial

Learn how to export custom kernel headers to the sysroot for use in cross-compiling applications.

Overview

When you have completed this tutorial, you will:

- learn how to unpack the Linux kernel and locate the source directory
- create a test file header
- commit the file to the kernel git repository
- update the platform project **layers/local/conf/layer.conf** file to include the new file header
- add files or directories to the exported source
- rebuild the kernel to include the custom header changes.

Tutorial Requirements

To complete this tutorial, you require a previously configured and built platform project. For additional information on creating a platform project, see *Wind River Linux Getting Started Command Line Tutorials: Creating and Configuring a Platform Project*.

Exporting Custom Kernel Headers to the Sysroot

It is possible to export custom kernel headers for application development cross-compilation using a built-in task for the Linux kernel.

The Wind River Linux kernel includes the **linux-windriver.install_kernel_headers** task, which enables developers to export their custom kernel headers to the sysroot for use in cross-compiling user space code. This task is provided by default and does not require any specific platform project configuration option. This task runs after **linux-windriver.do_install()** and before **linux-windriver.do_populate_sysroot**. Any header files and directories listed in the global variable *KERNEL_INSTALL_HEADER* are copied to the sysroot.

Each entry in *KERNEL_INSTALL_HEADER* is expected to exist in the Linux kernel source **include/** directory. If a file already exists in the destination, the build system will not overwrite it, but instead issue a warning. For example, to include a header file named **myfile.h**, the file must exist in the ***projectDir*/build/linux-windriver/linux/include** directory, or a subdirectory of it.

The *Adding a File or Directory to be Exported when Rebuilding a Kernel* on page 29 tutorial topic provides examples for using *add KERNEL_INSTALL_HEADER_append*.

To begin the tutorial, select *Exporting Custom Kernel Headers* on page 28.

Exporting Custom Kernel Headers

Use this procedure to export custom kernel headers for application development cross-compilation.

This procedure requires that any custom kernel header files that you want to export be located in the ***projectDir*/build/linux-windriver/linux/include** directory or a subdirectory. For additional information, see *Exporting Custom Kernel Headers to the Sysroot* on page 28.

Step 1 Unpack the Linux kernel.

Run the following command in the root of the ***projectDir***.

```
$ make linux-windriver.patch
```

Step 2 Navigate to the source directory of the kernel build.

```
$ cd build/linux-windriver/linux
```

Step 3 Optionally create a file to test this procedure.

If you do not have a file in the ***projectDir*/build/linux-windriver/linux/include** directory, you can use the following line to create one for testing purposes:

```
$ echo "#define my file" > include/myfile.h
```

Step 4 Add and commit your file to the git repository for the kernel.

```
$ git add include/myfile.h
$ git commit -m "new #define my file"
```

For the commit message, you can enter anything you like, specific to your custom header file.

Step 5 Open the ***projectDir*/layers/local/conf/layer.conf** file in a text editor.

Step 6 Add the header file to the ***projectDir*/layers/local/conf/layer.conf** file and save the file.

```
$ KERNEL_INSTALL_HEADER_append += "myfile.h"
```

This will include your custom header file in the build. For additional information on adding header files, see *Adding a File or Directory to be Exported when Rebuilding a Kernel* on page 29.

Step 7 Navigate back to the project directory.

```
$ cd ..
```

Step 8 Rebuild the kernel:

```
$ make linux-windriver
```

This can take some time to complete. When it finishes, your custom header file will be located in the ***projectDir*/bitbake_build/tmp/sysroots/*BSP_name*/usr/include** directory.

For a qemux86-64 BSP, the path would be ***projectDir*/bitbake_build/tmp/sysroots/qemux86-64/usr/include/myfile.h**. This places your custom header file in the appropriate directory for user space cross-compiling.

Adding a File or Directory to be Exported when Rebuilding a Kernel

Append files or directories to the *KERNEL_INSTALL_HEADER* variable each time the kernel is rebuilt as shown in these examples.

Prerequisites

This procedure is a supplement to *Exporting Custom Kernel Headers* on page 28.

Each entry in the *KERNEL_INSTALL_HEADER* variable is expected to exist in the Linux kernel source **include/** directory. To add a file or directory to be exported each time you rebuild the kernel, use *KERNEL_INSTALL_HEADER_append* to add to the variable as illustrated in the following example.

This variable is not configuration file-specific, and can be added to any of your layer configuration files, such as

***projectDir*/layers/local/conf/layer.conf**

Step 1 Open the ***projectDir*/layers/local/conf/layer.conf** file in a text editor.

Step 2 Update the *KERNEL_INSTALL_HEADER* variable.

- To add a single file, such as **myfile.h**:

```
KERNEL_INSTALL_HEADER_append += "myfiles/myfile.h"
```

- To add all files in a directory:

```
KERNEL_INSTALL_HEADER_append += "myfiles"
```

Step 3 Save the file.

Exporting Custom Kernel Headers Summary

In this tutorial, you learned how to export custom kernel headers and how to use the *add KERNEL_INSTALL_HEADER_append* to include additional files and directories as part of the exported content.

A

Wind River Education Resources

Wind River Education Services offers customized on-site and self-paced courses.

If you need a more detailed learning experience than is provided by these tutorials, you can purchase instructor-led courses on a wide range of Wind River Linux subjects from Wind River Education Services.

Courses are offered at your location or over the internet by professional instructors with years of experience working with Wind River Linux, and tailored to cover exactly the topics you choose.

See *http://windriver.com/education/* for more information.