Creating Layers Lab

WIND RIVER

Creating Layers Lab

Objective

In this lab you will learn how to capture common project settings into a custom bitbake layer which can be reused in multiple projects.

NOTE: This lab should take approximately 60 minutes.

Lab Overview

*The content that makes up your Wind River Linux platform project is delivered through a collection of **layers.** Most of the layers included in standard projects are provided by the Wind River Linux product and common to every project. Every project also comes equipped with a built-in **local layer**, which can be used to develop custom content unique to your project. Platforms can be further enhanced by including additional layers.*

Throughout this training, you have customized your platform project in a number of different ways. In most cases, these changes have been recorded in the local layer, where these customizations can survive for the life of your project, but are only available to that particular project.

Customizations can be shared with other platform projects by placing them into an external custom layer. This lab will walk you through the creation of a new layer, as well as populating it with some common customizations.

Creating the Layer Directory Structure

A layer is nothing more than a directory structure populated with a collection of configuration files and other resources needed to build software. Creating a new layer is simply a matter of creating directories and files in the places the build system expects to find them.

In this section, you will create the skeleton of a new layer, which will be populated with content in subsequent sections.

Note that some layers contain additional elements beyond those which are discussed in this lab; these exercises only touch on the elements commonly used in custom application layers.

1. Create the top level directory for your layer.

```
mkdir /home/wruser/MyLayer
```

NOTE:
- Layers can reside anywhere in your file system.
- The top-level directory name (in this case, **MyLayer**) should match the layer name.
- Layer names can be anything you like. In the Yocto Project community, it is a common practice to use the prefix **meta-**.

2. Execute the following command to navigate to the layer.

```
cd /home/wruser/MyLayer
```

3. Execute the following command to create the directory **conf**, which will contain the **layer.conf** file. Other configuration files can also be maintained here, as well.

```
mkdir conf
```

4. Execute the following command to create the directory **downloads**. If your layer provides packages, then local copies of the archives associated with the packages (if any) will be maintained here.

```
mkdir downloads
```

5. Execute the following command to create the directory **recipes-custom**, which will contain the following:

   - **Recipes:** provide configuration information, commonly used to provide instructions for building a particular package.
   - **Append files:** provide the ability to extend existing recipes. This capability enables your layer to add to, or modify aspects of, an existing recipe, without having to copy and modify the original recipe.

   Note that the naming of this directory is somewhat arbitrary; the only requirement is that the name begins with the prefix **recipes-**. It's fairly common for layers to have multiple recipe directories to main logical groupings of content.

```
mkdir recipes-custom
```

6. Lastly, execute the following command to create a **templates/feature** directory structure. The feature directory will contain Wind River Linux **templates** provided by your layer, if any.

```
mkdir -p templates/feature
```

7. Now create the **layers.conf** file in the **conf** directory. This file is a requirement for all layers; without it the bitbake system will not recognize your layer.

   Most **layer.conf** files contain many common elements. As such, when creating a new layer, it is common practice to copy an existing **layer.conf** file, making minor customizations needed to match the layer name. This file can be modified thereafter, as needed.

   For this step, copy the **layer.conf** from the local layer of an existing project; this can be the standard project included with the lab environment (refer to the *Getting Started* lab to identify where this project is located), or it can be one of the projects you've created in a previous lab.

```
cp $HOME/myplatform/layers/local/conf/layer.conf conf
```

8. Replace any instances of the string **local** within your new **layer.conf** with the name of your layer, **MyLayer**.

9. Take a moment to acquaint yourself with the contents of your **layer.conf** file, which should now resemble the following:

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

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

BBFILE_COLLECTIONS += "MyLayer"
BBFILE_PATTERN_MyLayer := "^${LAYERDIR}/"
BBFILE_PRIORITY_MyLayer = "10"

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

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

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

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

The elements of this file are interpreted as follows:

- The **BBPATH** variable is used by bitbake to maintain a list of directories to search for crucial information (like **layer.conf** files). Layers augment this variable in the manner shown to make themselves known to bitbake.
- The **BBFILES** variable is used by bitbake to maintain a list of recipes (***.bb**) and append files (***.bbappend**). Layers manipulate this variable as shown to publish any recipes or append files they provide. From studying this expression, you will see that this is where the requirement for the **recipes-** prefix, mentioned previously, comes from. Although you are free to modify this expression, you shouldn't, because this naming scheme is a standard Yocto Project convention.
- **BBFILE_COLLECTIONS** maintains a list of layers. Therefore, this is where you declare the name for your layer.
- **BBFILE_PATTERN_MyLayer** is a layer-specific variable specifying a regular expression which bitbake can use to identify files (in **BBFILES**) belonging to your layer. In this instance, the standard expression provided matches any file beginning with the layer directory, **${LAYERDIR}**.

- **BBFILE_PRIORITY_MyLayer** is another layer-specific variable specifying a priority which bitbake assigns to your layer. Bitbake uses layer priorities to determine the order in which to find recipes and other resources provided by your layer. Priorities are used to resolve ambiguities when multiple layers contain overlapping resources.
- The augmentation of **PATH** is optional, and meaningful only if your layer has a **scripts** directory containing utilities to be run on the host.
- **WRL_CHANGELIST_PATH** and **WRL_FS_FINAL_PATH** enable **fs_final.sh** and **changelist.xml** file system finalizers to be offered by your layer. This functionality is provided by the Wind River Linux image classes, which are enabled by default when using the Wind River Linux **configure** script to set up your build environment.
- **PREMIRRORS** maintains a list of directories containing source archives, which bitbake can use in lieu of downloading over the network. Layers augment this variable in the manner shown to enable such archives to be found in the **downloads** directory, in a variety of formats.

NOTE: Although providing a **README** file in your layer's top-level directory is not a technical necessity, Yocto Project compliance requires the presence of such a file to document your layer's purpose and usage instructions.
For examples, refer to any of the standard product layers which can be found in any configured build environment; for example, **/home/wruser/myplatform/layers/wr-base/README**.

Your layer, although devoid of any actual content, is now technically complete, and is ready to be included into projects. In the exercises that follow, you will populate your layer with content related to some of the common layer use cases.

Using Your Layer

In this section, you will add the layer you created in the previous section to a new platform project. Throughout the rest of the lab, you can use this project to incrementally verify that your layer settings are taking effect. Take the time to open a new terminal window so that you can maintain the shell from the previous section in your layer directory, and one in your new build environment.

10. In the new shell, create a directory for your new project and switch into this directory.

```
mkdir /home/wruser/layer_test_prj
cd /home/wruser/layer_test_prj
```

11. Configure a new build environment. The settings for **--enable-board**, **--enable-kernel**, and **--enable-rootfs** should match the target included in your lab environment to improve build times; you can use the **$BSP**, **$KERNEL**, and **$ROOTFS** variables to specify these. In addition to these three arguments, you should include:
    - **--with-layer=/home/wruser/MyLayer** to include your custom layer
    - **--with-sstate-dir=/Labs/sstate** to speed up the build

```
$WIND_BASE/wrlinux/configure --enable-board=$BSP \
                             --enable-kernel=$KERNEL \
                             --enable-rootfs=$ROOTFS \
                             --with-layer=/home/wruser/MyLayer \
                             --with-sstate-dir=/Labs/sstate
```

12. Examine the contents of **bitbake_build/conf/bblayers.conf** and notice that a reference to your layer is contained within. Since your layer is essentially empty at this point, it will not exert any real influence over the build environment yet. In the sections that follow, you will build up your layer and observe the impact it has on projects that include it.

NOTE: This section illustrates how to add your layer to a new platform project. You can also add layers to existing projects. This is covered in the *Reconfiguring Projects* lab.

Adding Packages

You will undoubtedly need to add packages or applications to your projects at one point or another. In the *Integrating Packages* lab, you learned how to integrate upstream and in-house applications into your build environment, using the project's local layer. In this section, you will embed a package which is known to build properly into your new custom layer.

The package you will add to your layer is called **figlet**. It contains the following parts, which you will find common to most packages. These can all be found in the **/Labs/LayersAndTemplates** directory.

- An archive, **figlet-2.2.5.tar.gz**, containing the upstream source for this package. Upstream packages are commonly delivered in the form of an archive. Package source can also be delivered as a source tree, or via an SCM like **git** or **subversion**.
- A recipe, **figlet_2.2.5.bb**, which provides the bitbake instructions for building the package. A recipe is always required in order to build a package.
- A patch, **makefile-integration.patch**, which must be applied to the upstream source prior to building, in order for the package to build properly. Patches aren't

always required; some upstream source is able to build cleanly without the need for patching. In this particular case, however, a patch is required.

For this package to be represented properly, all components must be included in the proper locations within your layer.

NOTE: Be sure to perform all steps in your layer directory, unless otherwise noted.

13. Because the source is delivered as an archive, the archive must be copied into your layer's **downloads** directory.

```
cp /Labs/LayersAndTemplates/figlet-2.2.5.tar.gz downloads
```

14. Next, create a package-specific directory within your layer's **recipes-custom** directory. Here, you will place the package recipe. Other components needed to build **figlet** will be rooted here, as well.

```
mkdir recipes-custom/figlet
```

NOTE: There is no technical requirement to name this directory after your package. In fact, you could have a single directory containing all of the configuration information for all of the packages provided by your layer. From a manageability standpoint, however, it makes good sense keep your recipes organized in directories named after their associated packages.

15. Execute the following command to copy the recipe into this newly created directory.

```
cp /Labs/LayersAndTemplates/figlet_2.2.5.bb recipes-custom/figlet
```

16. Execute the following command to create the subdirectory **files** which will be used to additional resources needed to build the package, such as:

    - Patches to be applied to the source prior to building, if required
    - Package source tree, if no archive is provided

```
mkdir recipes-custom/figlet/files
```

NOTE: The name **files** is a requirement in this case. Bitbake will search this subdirectory for patches and other resources related to the package.

17. Execute the following command to copy the needed patch file into the **files** subdirectory.

```
cp /Labs/LayersAndTemplates/makefile-integration.patch \
   recipes-custom/figlet/files
```

18. Now your layer contains sufficient information to build and include **figlet** into your image. Verify that figlet is buildable within the project you created in the previous section. Switch to your project shell, and issue the following:

```
make -C build figlet
```

    This will build the **figlet** package. Assuming this succeeds, this is sufficient to prove that the **figlet** package has been properly integrated into your layer.

19. Now, wire your layer to automatically include the **figlet** package into the image. Back in your layer shell, add the following line to the file **conf/layer.conf**:

    ```
    IMAGE_INSTALL_append += "figlet"
    ```

20. In your project shell, execute the following command to build the image.

```
make
```

21. When the build completes, you the **figlet** application is built into your target file system. Execute the following command to verify that the target file system includes the **figlet** binary.

```
find export/dist –name figlet
```

Modifying File System Content

As you saw in the *Building and Customizing a Wind River Linux Platform* lab, there are a couple of mechanisms provided by Wind River Linux which allow direct manipulation of the target file system content: **changelist.xml**, and **fs_final.sh**.

In this section, you will integrate both into your custom layer. This will give you some insight into the naming requirements for these files, as well as how to deal with multiple **changelist.xml** files.

Perform this section in your layer directory.

22. Execute the following command to create a new subdirectory called **image_final** within your layer's **conf** directory. This is the standard place searched in layers for **changelist.xml** and **fs_final.sh** files.

```
mkdir conf/image_final
```

23. The directory **/Labs/LayersAndTemplates** contains two scripts (**motd.sh** and **fifo.sh**). Both scripts are designed to be run as **fs_final** scripts at file system finalization time. Execute the following command to copy these files into the **image_final** subdirectory, observing the following naming requirements:

    - Files must begin with the prefix **fs_final**.
    - Files must end with the suffix **.sh**.

    This naming scheme is required in order for the system to find your scripts.

```
cp /Labs/LayersAndTemplates/motd.sh
     conf/image_final/fs_final_motd.sh
cp /Labs/LayersAndTemplates/fifo.sh
     conf/image_final/fs_final_fifo.sh
```

24. Test the new additions to your layer by switching to your project shell and building the project image which you are using to test your layer.

```
make
```

25. Execute the following command to verify that the **/etc/motd** in the target file system contains the modified text.

```
cat export/dist/etc/motd
Welcome to your Wind River Linux 5 image
```

26. Also verify that a FIFO node called **mypipe** exists in the root directory (**export/dist**), with mode 0777. To verify this effectively, you will need to enter the fake root environment, as outlined in the *Building and Customizing a Wind River Linux Platform* lab. The procedure is quickly recapped below:

```
scripts/fakestart.sh
ls -l export/dist/mypipe
```

    When done, leave the fake root environment using the **exit** command.

27. Switch back to your layer shell and turn your attention to the **changelist.xml** files. In **/Labs/LayersAndTemplates**, you will find two files, **changelist_login_msg.xml** and **changelist_readme.xml**. Because there can be only one **changelist.xml** file in a layer, you must learn how to merge these files, should you ever need to. Begin by copying the first file into the **conf/image_final** directory.

```
cp /Labs/LayersAndTemplates/changelist_login_msg.xml \
   conf/image_final/changelist.xml
```

28. Now, take a look at the second file, **changelist_readme.xml**.

```
<?xml version="1.0" encoding="UTF-8"?>
<layout_change_list version="1">
<change_list>
<cl action="addfile" name="/root/README.txt" \
    source="/Labs/LayersAndTemplates/README.txt">
</change_list>
</layout_change_list>
```

To merge this file, you will need to copy only the **cl** directive from this file into the main **changelist.xml**. When done, **conf/image_final/changelist.xml** should resemble the following.

NOTE: The command in a **changelist.xml** file should not be split across lines. It is only done in this document to fit the page.

```
<?xml version="1.0" encoding="UTF-8"?>
<layout_change_list version="1">
<change_list>
<cl action="addfile" name="/root/.profile" \
    source="/Labs/LayersAndTemplates/dotprofile">
<cl action="addfile" name="/root/README.txt" \
    source="/Labs/LayersAndTemplates/README.txt">
</change_list>
</layout_change_list>
```

29. Save the file and test the new additions to your layer by rebuilding the project image which you are using to test your layer. Verify that the files **/root/README.txt** and **/root/.profile** exist in the target file system.

Kernel Configuration and Patching

The *Configuring and Patching the Kernel* lab walked you through the integration of kernel patches and configuration fragments into your build environment's local layer. In this section, you will learn how to replicate these features in your new custom layer.

The following components are required to patch and/or configure the kernel:

- a feature description (**.scc**) file which pulls in configuration fragments and/or patches
- if you are configuring the kernel, a configuration fragment which specifies kernel configurations settings
- if you are patching the kernel, a patch or set of patches

Perform this section in your layer directory.

30. Kernel-related configuration requires a bit of additional directory infrastructure in your layer, just like packages do. Create a kernel-specific directory within your layer, which will house the append file you're going to create, as well as all the kernel-related configuration you want to include.

```
mkdir -p recipes-kernel/linux/linux-windriver-3.4
```

    The components of this new path represent the following:

    - the new directory **recipes-kernel**, parallel to the **recipes-custom** used in the previous section. It's not required to keep kernel-related configuration separated like this. In truth, you could put your kernel configuration into **recipes-custom** if you really wanted to. But because the kernel is so special, it is common practice to stash kernel-related configuration into a separate subdirectory called **recipes-kernel**.
    - the directory **linux**, which provides a container for the append file you're going to create. As with packages, the name of this directory component is not significant as far as the build system is concerned. But from a manageability standpoint, **linux** is the name commonly used for kernel configuration.
    - the directory **linux-windriver-3.4**, which will contain the kernel-related material you want to include in your layer. The name of this component must be matched to the append file you're going to create which, in turn, is matched with the recipe being used for the target kernel.

31. Now execute the following command to copy the kernel material into the **linux-windriver-3.4** directory: a patch which implements a large startup banner, as well as a configuration fragment which sets a default message for the banner.

```
cp /Labs/LayersAndTemplates/0001-Implemented-big-boot-
     banner.patch \
   /Labs/LayersAndTemplates/banner.cfg  \
   recipes-kernel/linux/linux-windriver-3.4
```

32. In that same directory, create a feature description file, **banner.scc**, which pulls in the configuration fragment and patch.

    ```
    kconf non-hardware banner.cfg
    patch 0001-Implemented-big-boot-banner.patch
    ```

33. Finally, in the directory **linux**, create the append file. This file will extend the kernel recipe, adding a reference to the feature description created in the previous step. This is a name-sensitive component, and must be named **linux-windriver_3.4.bbappend**. Notice the following:

    - the naming for recipes (and append files) separates the name and version with an underscore (**_**) character rather than a hyphen (**-**)
    - **linux-windriver** is the formal package name for the Wind River Linux kernel
    - the version, **3.4**, must match the version of the kernel being used in your product
    - the file must have a **.bbappend** suffix. If you instead use a suffix of **.bb**, then the contents of your file will clobber the original kernel recipe!

    The contents of this file should resemble the following:

    ```
    EXTRA_KERNEL_SRC_URI += "file://banner.scc"
    EXTRA_KERNEL_FILES_append := "${THISDIR}/${PN}-${PV}:"
    ```

    This performs the following:

    - adds **banner.scc** to the list of components which the kernel build system will look for (as maintained by the build variable **EXTRA_KERNEL_SRC_URI**)
    - extends the search path of the kernel build system by adding to the **EXTRA_KERNEL_FILES** variable. **${THISDIR}** expands to the directory containing the append file, while **${PN}** and **${PV}** expand to the package name and version, respectively (thus evaluating to **linux-windriver-3.4**).

34. Now, test the new additions to your layer by invoking the kernel **configure** target in your project shell.

```
make -C build linux-windriver.configure
```

Examine the kernel configuration file, found at **build/linux-windriver-3.4-r0/linux-$BSP-$KERNEL-build/.config** (the exact path will depend on the board and kernel selection). Within, you should find the two settings from **banner.cfg**. These configuration settings are not part of the standard kernel settings and will only appear if your configuration fragment was applied successfully.

```
CONFIG_BANNER=y
CONFIG_BANNER_TEXT="Wind River Linux"
```

Also verify that the patch applied by checking for the presence of a file called **banner.c** in the **build/linux-windriver-3.4-r0/linux/init** directory. This file is not part of the standard kernel source; rather it is provided by the patch you added earlier in this section.

To fully test the feature, you should:

- Rebuild the kernel
- Rebuild your image
- Deploy the image to the target
- Boot the target

Do this if time permits; rebuilding the kernel can take some time.

This concludes the lab. Do not proceed.