Building and Customizing a Wind River Linux Platform Lab

WIND RIVER

Building and Customizing a Wind River Linux Platform Lab

Objective

In this lab, you will create, build, and test a new Wind River Linux platform using a simulated target. You will also learn how to:

- *make changes to the target file system*
- *configure and build individual packages*

NOTE: This lab should take approximately 45 minutes.

Setting Up the Build Environment

In this section, you will set up a new Wind River Linux platform build environment using command line tools provided with the Wind River Linux product. The build environment provides all the tools and configuration needed to cross compile and deploy a Wind River Linux kernel and user space for a particular target.

1. In a shell on the host, create a new directory structure to house your platform build. This can be located anywhere you like, but in this example, use the directory **$HOME/myplatform**:

```
mkdir -p $HOME/myplatform
cd $HOME/myplatform
```

2. To establish the build environment for building the platform, you must invoke the Wind River Linux **configure** script. This script is located within the Wind River Linux product installation. Using syntax similar to GNU autoconf, **configure** supports a number of command-line arguments for selecting a BSP, kernel type, and file system profile, as well as many other settings. Commonly used options are summarized in the table below:

Option Name	Purpose
--help	Display an extensive list of supported options.
--enable-board	Specify the target BSP.
--enable-kernel	Select the type of Wind River Linux kernel to use.
--enable-rootfs	Select the type of Wind River Linux user space to use.
--enable-build	Select how software should be built.
--with-layer	Include an additional layer.
--with-template	Include an additional template.
--enable-ccache	Enable the compiler cache (ccache) to speed up build time.
--with-ccache-dir	Specify an external compiler cache directory.
--with-sstate-dir	Bitbake shared state cache directory. Using a shared state cache significantly improves build times.

View the complete list of options available:

```
$WIND_BASE/wrlinux/configure --help
```

3. In this example, you will create a platform similar to the one shipped with the lab environment. Doing so will significantly reduce build time since the shared state cache found at **/Labs/sstate** can be used to accelerate the build.

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

The output of **configure** will stream by. Once **configure** finishes, your directory will contain the infrastructure needed to build and deploy your own customized Linux distribution. Take the time to familiarize yourself with the build environment. There are a few important directories to note:

- The **build** directory hosts soft links to the build directories for each package. This subdirectory also hosts a **Makefile** used to build special targets associated with the packages. For more details, execute the following command:

```
make -C build help
```

- The **bitbake_build/conf** directory contains the following important files:
    - **local.conf**, which contains crucial build configuration information. A soft link to this file is created in the top-level directory for your convenience.
    - **bblayers.conf** file lists the layers used in constructing the build environment.
- The **export** directory will contain the output of the build, including:
    - An archive containing the target file system, which can be extracted directly to a device file system or NFS share
    - **dist**, a directory containing the target file system. When using a simulator such as QEMU, it mounts this directory as a root file system during boot.
    - The Linux kernel image. The name of this image might vary, depending on your target.
    - An archive containing the kernel modules.
- The **host-cross** directory contains tools used to build the platform, including (but not limited to) the cross-compiling toolchain used to build programs for your target. Note that the directories inside host-cross are soft links to corresponding directories in **bitbake_build/tmp/sysroots**.
- The directory **layers/local** constitutes a “local layer” which functions as a container for your project-specific customizations.
- The **layers** directory also contains additional layers used in your project; the stock layers are all provided by the standard Wind River Linux installation, although you may include additional custom layers if needed.

Building the Platform

With a configured build environment in place, you may now build your platform. The output of this stage will be a Wind River Linux kernel image and root file system that are ready to use by the target; both will be found in the **export** directory as noted in the previous section.

The time required to build a platform varies on a number of factors, not the least of which being the overall speed of your host. Other factors that have an impact are:

- Whether or not the kernel needs to be built from source; a complete build of the kernel typically takes in the order of a half hour to an hour.
- The number of target user-space packages that need to be built from source. A basic **glibc_small** system can be built entirely from source in about one to two hours. A **glibc_std** system, on the other hand, can take many hours.
- Whether or not you are using a prepopulated **shared state cache**. If using a shared state cache containing all the needed objects, the above build time for a **glibc_small** system takes only minutes.

To provide for a better lab experience, the lab environment you are using includes a shared state cache that prepopulated with all the objects needed to build a **glibc_small** system for your target. Leveraging this using the **--with-sstate-dir** argument minimizes the build time required.

4. To build your platform image, simply invoke **make** as follows:

```
make
```

5. After the build finishes, explore the contents of the **export** directory. In particular, take note of **export/dist**, which contains a complete copy of your target file system. Note however, that some file attributes aren't quiet as expected. For example:

```
ls -l export/dist/dev
-rw-------. 1 wruser wruser 0 Jan 13 22:39 apm_bios
-rw-------. 1 wruser wruser 0 Jan 13 22:39 console
-rw-------. 1 wruser wruser 0 Jan 13 22:39 fb0
-rw-------. 1 wruser wruser 0 Jan 13 22:39 hda
-rw-------. 1 wruser wruser 0 Jan 13 22:39 hda1
-rw-------. 1 wruser wruser 0 Jan 13 22:39 hda10
-rw-------. 1 wruser wruser 0 Jan 13 22:39 hda11
…
```

Compare this to what you see in **/dev** on your host:

```
ls -l /dev
crw-rw----. 1 root video    10, 175 Jan 11 06:26 agpgart
crw-------. 1 root root     10, 235 Jan 11 06:26 autofs
drwxr-xr-x. 2 root root         300 Jan 11 06:26 block
drwxr-xr-x. 2 root root          80 Jan 11 06:26 bsg
c---------. 1 root root     10, 234 Jan 11 06:26 btrfs-control
drwxr-xr-x. 3 root root          60 Jan 11 06:26 bus
lrwxrwxrwx. 1 root root           3 Jan 11 06:26 cdrom -> sr0
drwxr-xr-x. 2 root root        3120 Jan 16 17:05 char
crw-------. 1 root root      5,   1 Jan 11 09:56 console
…
```

Focus not on the entries themselves, but rather the ownerships and attributes. Notice that the entries in **/dev** on your host:

- Have **root:root** ownership
- Are primarily pipes or block or character device nodes rather than regular files (as evidenced by the **b** , **c** and **p** bits in the leftmost column)
- The device nodes have major and minor numers associated with them (10,175, 5,1, etc)

Whereas the entries in **export/dist/dev** all appear as regular files owned by **wruser:wruser**.

The reason for this is because the entire build, including the generation of **export/dist**, was done as a regular user (**wruser**). Under these conditions, it would be impossible to generate device nodes or any files owned by **root**. Instead, the file system is built in a fake root environment called **pseudo**. To see the effect pseudo has on your view of the files, enter the pseudo environment and then reexamine the directory:

```
scripts/fakestart.sh
ls -l export/dist/dev
crw-rw---- 1 root   46  10, 134 Jan 13 22:39 apm_bios
crw-rw--w- 1 root tty    5,   1 Jan 13 22:39 console
crw------- 1 root root  29,   0 Jan 13 22:39 fb0
brw-rw---- 1 root disk   3,   0 Jan 13 22:39 hda
brw-rw---- 1 root root   3,   1 Jan 13 22:39 hda1
brw-rw---- 1 root root   3,  10 Jan 13 22:39 hda10
brw-rw---- 1 root root   3,  11 Jan 13 22:39 hda11
…
```

Like magic, all is as it should be. Always remember that if you want to see the file system exactly as it was meant to be, that you must first enter the pseudo environment in which it was generated.

6. To leave the pseudo environment, simply issue the command **exit**.

Deploying Your Platform

With a built platform project in place, it is time to deploy and test your image on your target. For a full discussion how to prepare and shut down your target, please refer to one of the following labs, depending on the type of your target:

- If you are simulated target (QEMU or Wind River Simics), use the command **make start-target** as outlined in the *Simulating Targets with QEMU* lab.
- If you are using a real board, please refer to the *Working with a Hardware Target* lab.

7. Test your platform in the simulator using the following command:

```
make start-target
```

8. Once your target boots, log in using the credentials *root* and password *root*.
9. Feel free to explore your target using the command-line shell. When done, the preferred method of shutting a Linux system is always using the **poweroff** command, although you can use other methods as well, as outlined in the *Simulating Targets with QEMU* lab.

Customizing the File System with changelist.xml

In this section, you will tweak the target file system with the help of a **changelist.xml** file. This file will be located in the local layer and will be executed by the build system just before the final file system image is generated.

Although these XML files might seem awkward to work with and edit, the main benefit is that they integrate nicely with the Workbench file system management tools.

10. Use a **changelist.xml** file to copy the **README.txt** file found in **/Labs/BuildSystemLab** directory into the **/root** directory of the target file system.

    To do this, create a file called **changelist.xml** in the **conf/image_final** subdirectory of the local layer (that is, **layers/local/conf/image_final** relative to the top-level directory of the project). Populate the file with the following content:

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

NOTE: This is important: the **cl action** line must be entered as one continuous line. Due to printing restrictions, it is broken across two lines in this document.

11. Once you have made your changes, rebuild the platform and start the target to test your changes. Verify that the file **/root/README.txt** can be found on the target, and that its contents match those on the host.

Customizing the File System with fs_final Scripts

In this section, you will tweak the target file system with the help of some **fs_final_*.sh** scripts. Like **changelist.xml** files, these scripts will be located in the local layer and will be executed by the build system just before the final file system image is generated.

Unlike **changelist.xml** files, **fs_final** scripts are plain shell scripts and are easier to write. But there is no integration with the Workbench file system management tools.

12. Use an **fs_final** script to add an **/etc/motd** file to your target file system. The contents of **/etc/motd** is displayed on the console right after you log in (**motd** originates from the phrase, “Message Of The Day”).

    To do this, create a file called **fs_final_motd.sh** in the same directory you used to create the **changelist.xml** file in the previous section. Populate the file with the following contents:

    ```
    echo “Welcome to your Wind River Linux 5.x image” > etc/motd
    ```

NOTE: The lack of a leading slash in **etc/motd** is intentional. The **fs_final** scripts are executed in the root directory of the target file system, but are *not* executed within a **chroot**ed environment. This allows **fs_final** scripts to have full access to the host file system as **changelist.xml** files do; but careful attention must be paid when referencing files. In this example, if you inadvertently wrote **/etc/motd**, you would actually be referring to **/etc/motd** *in the host file system*.

13. Rebuild your image and redeploy the updated image to your target. Log in to verify that your changes have taken effect; you should see the contents of the **motd** file display after you log in.

14. To see the benefits of the **fs_final** mechanism, suppose you want to embed information into your root file system that is not static. Suppose, for example, that you wanted to embed the build date and board name into the **motd**. Add the following code to **fs_final_motd.sh**:

```
echo "This image was built on $(date)" >> etc/motd
```

15. Now rebuild and redeploy the updated image to your target again. Log in to verify that your changes have taken effect. Notice that the date in the motd is hard-coded to the exact time that the **fs_final_motd.sh** script was executed in building the file system image.

16. Now, apply what you've learned to create a new **fs_final** script that creates a FIFO entry, **/mypipe** with mode 0777. If you're having difficulty, consult the online manual:

```
man mkfifo
```

If you're stuck, a solution can be found in **/Labs/BuildSystemLab/fs_final_fifo.sh**. Simply copy this solution into the same directory as **fs_final_motd.sh**, rebuild the file system, and deploy the image to the target.

Customizing and Building Individual Packages

Currently, your platform is built using the packages that ship with the Wind River Linux product. It's possible to rebuild individual packages from source if needed. Common reasons for wanting to do this include:

- You wish to apply a patch to the source of a particular package
- You want to make configuration changes to a package
- You want to be able to debug the software provided by a particular package.

In this section, you will focus on the **busybox** package, which provides the core run-time for **glibc_small** systems. In this exercise, you will modify the source of the **ls** program to print an additional message when run.

17. To modify the source to busybox, open a development shell for the package, as follows:

```
$ make -C build busybox.devshell
```

This will open a new terminal where the busybox source is located with all relevant patches already applied. Note the name of the directory where the terminal opens, **busybox-1.19.4**, which will be used in a later exercise.

18. Edit the code which implements the **ls** command provided by **coreutils/ls.c**. Add an additional message to beginning of the **ls_main()** function; for example:

    ```
    init_unicode();
    fprintf(stderr, "Hello from ls\n");
    if (ENABLE_FEATURE_LS_SORTFILES)
    …
    ```

19. Once you finish editing the file, exit the development shell by typing **exit** or pressing **CTRL+D**. You will be taken back to your original shell.

20. Now, rebuild **busybox** with your source changes:

```
make -C build busybox.rebuild
```

With the package rebuilt, once again rebuild the image and deploy it to the target to test your changes. Verify that every time you invoke the **ls** command, your message appears.

Note that the changes you have made are only temporary. If you do a **clean**, **distclean**, or **cleansstate**, your changes will be lost, and the original behavior of **ls** will be restored.

21. Now that you have seen how to modify software for your platform, clean the build area as follows:

```
make -C build busybox.clean
```

22. Rebuild busybox, then rebuild the image and deploy it to your target. Now you will notice that the behavior of **ls** is restored to its original state.

Building a Boot Image

As you have seen, building a kernel and target file system is relatively straightforward. The product of this build is:

- A target file system rooted at **export/dist**
- A compressed archive containing the file system contents in the **export** directory

When deploying to hardware, you will often need a **boot image**; that is, a self-contained file system image for deploying on to some form of medium; for example, flash storage.

Wind River Linux provides support for many common boot image formats:

- **JFFS2** is derived from the original Linux flash file system, JFFS. JFFS2 is a journaling read-write file system which performs on-the-fly compression. Operating on top of the MTD subsystem, JFFS2 addresses the unique requirements of flash, such as wear leveling.
- **UBIFS** is a read-write file system designed specifically for raw NAND flash. It takes care of bad sectors in flash by implementing a logical layer that maps used flash sector numbers to only good sectors in the physical media. UBIFS is supported natively by the Linux kernel. Note that to deploy a UBIFS file on the target's NAND you need to "ubinize" the UBIFS file first to generate a **.ubi** file.
- **CPIO** is a common packing format used to pack the root file system into a single file. This file, usually in a compressed form, can then be appended to the kernel image itself in such a way that the Linux boot logic can use it as a root file system contained entirely in RAM.

23. Reconfigure the project to generate JFFS2 and CPIO images by editing the file **local.conf** in your project directory and appending the following:

    ```
    IMAGE_FSTYPES += "jffs2 cpio.gz"
    ```

    Again, the syntax might seem a little peculiar, but this is again a fragment of Python code used by bitbake At a high level, **IMAGE_FSTYPES** lists the types of images to build. Refer to **local.conf** for a full list of supported types.

24. Once you have saved your changes to the file **local.conf**, rebuilt your project. When this finishes, you will see three files in the **export/images** directory with the suffixes **.jffs2** and **.cpio.gz**.

This concludes the lab. Do not proceed.