Embedded Linux

LXE22109

Practical Labs

bootlin

https://bootlin.com

August 24, 2022

About this document

Updates to this document can be found on https://bootlin.com/doc/training/lxe22109-05.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

Copying this document

Corrections, suggestions, contributions and translations are welcome!

Advanced aspects

Objectives:

- *Use build time, dependency and filesystem size graphing capabilities*
- *Use licensing report generation, and add licensing information to your own packages*
- *Use* `BR2_EXTERNAL`

Build time graphing

When your embedded Linux system grows, its build time will also grow, so it is often interesting to understand where the build time is spent.

Since we just did a fresh clean rebuild at the end of the previous lab, we can analyze the build time. The raw data has been generated by Buildroot in `output/build/build-time.log`, which contains for each step of each package the start time and end time (in seconds since Epoch).

Now, let's get a better visualization of this raw data:

```
make graph-build
```

Note: you may need to install `python-matplotlib` on your machine.

The graphs are generated in `output/graphs`:

- `build.hist-build.pdf`, build time of each package, by build order
- `build.hist-duration.pdf`, build time of each package, by build duration
- `build.hist-name.pdf`, build time of each package, by package name
- `build.pie-packages.pdf`, build time of each package, in proportion of the total build time
- `build.pie-steps.pdf`, build time of each step

Explore those graphs, see which packages and steps are taking the biggest amount of time.

Note that when you don't do a clean rebuild, the `build-time.log` file gets appended and appended with all the successful builds, making the resulting graphs unexploitable. So remember to always do a clean full rebuild before looking at the build time graphs.

Dependency graphing

Another useful tool to analyze the build is graphing dependencies between packages. The dependency graph is generated for your current configuration: depending on the Buildroot configuration, a given package may have different dependencies.

To generate the full dependency graph, do:


```
make graph-depends
```

The graph is also generated in `output/graphs`, under the name `graph-depends.pdf`. On the graph, identify the `bar` and `ninvaders` packages you have created, and look at their dependencies to see if they match your expectations.

Now, let's draw a graph for a much bigger system. To do this, create a completely separate Buildroot output directory:

```
mkdir $HOME/lxe22109-05-labs/buildroot-output-test-graph/
cd $HOME/lxe22109-05-labs/buildroot-output-test-graph/
```

We're going to create a Buildroot configuration, so create a file named `.config` and put the following contents:

```
BR2_TOOLCHAIN_BUILDROOT_GLIBC=y
BR2_TOOLCHAIN_BUILDROOT_CXX=y
BR2_PACKAGE_MESA3D=y
BR2_PACKAGE_MESA3D_DRI_DRIVER_SWRAST=y
BR2_PACKAGE_MESA3D_OPENGL_EGL=y
BR2_PACKAGE_MESA3D_OPENGL_ES=y
BR2_PACKAGE_XORG7=y
BR2_PACKAGE_XSERVER_XORG_SERVER=y
BR2_PACKAGE_LIBGTK3=y
BR2_PACKAGE_WEBKITGTK=y
```

It represents a configuration that builds an internal toolchain, with a X.org graphic server, the Mesa3D OpenGL implementation, the Gtk3 library, and the Webkit Web rendering engine. We're not going to build this configuration, as it would take quite a bit of time, but we will generate the dependency graph for it.

First, let's run `make menuconfig` to expand this minimal configuration into a full configuration:

```
make -C $HOME/lxe22109-05-labs/buildroot/ O=$(pwd) menuconfig
```

Feel free to explore the configuration at this stage. Now, let's generate the dependency graph:

```
make graph-depends
```

Look at `graphs/graph-depends.pdf` and how complex it is. Now, let's look at the dependencies of one specific package, let's say `libgtk3`:

```
make libgtk3-graph-depends
```

Now, open the graph generated at `graphs/libgtk3-graph-depends.pdf`. As you can see, it is a lot more readable.

Such dependencies graphs are very useful to understand why a package is being built, and help identifying what you could do to reduce the number of packages that are part of the build.

Filesystem size graphing

Run `make graph-size` and watch the PDF generated at `output/graphs/graph-size.pdf`. You can also look at the CSV files generated in `output/graphs/`.

Licensing report

Go back to our original build directory, in `$HOME/lxe22109-05-labs/buildroot/`.

As explained during the lectures, Buildroot has a built-in mechanism to generate a licensing report, describing all the components part of the generated embedded Linux system, and their corresponding licenses.

Let's generate this report for our system:

```
make legal-info
```

In the output, you can see some interesting messages:

```
WARNING: bar: cannot save license (BAR_LICENSE_FILES not defined)
WARNING: libfoo: cannot save license (LIBFOO_LICENSE_FILES not defined)
WARNING: ninvaders: cannot save license (NINVADERS_LICENSE_FILES not defined)
```

So, now update your `ninvaders`, `libfoo` and `bar` packages to include license information. Run again `make legal-info`.

Now, explore `output/legal-info`, look at the `.csv` files, the `.txt` files, and the various directories. Buildroot has gathered for you most of what is needed to help with licensing compliance.

Use BR2_EXTERNAL

We should have used `BR2_EXTERNAL` since the beginning of the training, but we were busy learning about so many other things! So it's finally time to use `BR2_EXTERNAL`.

The whole point of `BR2_EXTERNAL` is to allow storing your project-specific packages, configuration files, root filesystem overlay or patches outside of the Buildroot tree itself. It makes it easier to separate the open-source packages from the proprietary ones, and it makes updating Buildroot itself a lot simpler.

So, as recommended in the slides, the goal now is to use `BR2_EXTERNAL` to move away from the main Buildroot tree the following elements:

- The `bar` and `libfoo` packages. We will keep the `ninvaders` package in the Buildroot tree, since it's a publicly available open-source package, so it should be submitted to the official Buildroot rather than kept in a `BR2_EXTERNAL` tree.
- The Linux kernel patch and Linux kernel configuration file.
- The *rootfs overlay*
- The *post-build script*
- The *defconfig*

Your `BR2_EXTERNAL` tree should look like this:

```
+-- board/
|   +-- bootlin/
|       +-- beagleboneblack/
|           +-- linux.config
|           +-- post-build.sh
|           +-- patches/
|               +-- linux/
|                   +-- 0001-Add-nunchuk-driver.patch
|                   +-- 0002-Add-i2c1-and-nunchuk-nodes-in-dts.patch
|           +-- rootfs-overlay/
|               +-- etc
|                   +-- network
```

```
|                        +-- interfaces
|                    +-- init.d
|                        +-- S30usbgadget
+-- package/
|   +-- bar
|       +-- 0001-Fix-missing-libconfig.h-include.patch
|       +-- bar.mk
|       +-- Config.in
|   +-- libfoo
|       +-- libfoo.mk
|       +-- Config.in
+-- configs
|   +-- bootlin_defconfig
+-- Config.in
+-- external.desc
+-- external.mk
```

Now, do a full rebuild using your `BR2_EXTERNAL` tree, and check that your system builds and runs fine!

Going further

If you have some time left, let's improve our setup to use *genimage*. This way, we will be able to generate a complete SD card image, which we can flash on a SD card, without having to manually create partitions. Follow those steps:

- Change the Buildroot configuration to generate an *ext4* filesystem image
- Take example on `board/stmicroelectronics/common/stm32mp157/genimage.cfg.template` to create your own `board/bootlin/stm32mp1/genimage.cfg`. Keep only the single Device Tree we need for our project.
- Adjust the Buildroot configuration to use the `support/scripts/genimage.sh` script as a *post-image* script, and pass `-c board/bootlin/stm32mp1/genimage.cfg` as *post-image* script arguments. Make sure to enable `BR2_PACKAGE_HOST_GENIMAGE`.

Advanced packaging

Objectives:

- *Package an application with a mandatory dependency and an optional dependency*
- *Package a library, hosted on GitHub*
- *Use* hooks *to tweak packages*
- *Add a patch to a package*

Start packaging application `bar`

For the purpose of this training, we have created a completely stupid and useless application called `bar`. Its home page is https://bootlin.com/~thomas/bar/, from where you can download an archive of the application's source code.

Create an initial package for `bar` in `package/bar`, with the necessary code in `package/bar/bar.mk` and `package/bar/Config.in`. Don't forget `package/bar/bar.hash`. At this point, your `bar.mk` should only define the `<pkg>_VERSION`, `<pkg>_SOURCE` and `<pkg>_SITE` variables, and a call to a package infrastructure.

Enable the `bar` package in your Buildroot configuration, and start the build. It should download `bar`, extract it, and start the configure script. And then it should fail with an error related to `libfoo`. And indeed, as the `README` file available in `bar`'s source code says, it has a mandatory dependency on `libfoo`. So let's move on to the next section, and we'll start packaging `libfoo`.

Packaging `libfoo`: initial packaging

According to `bar`'s `README` file, `libfoo` is only available on *GitHub* at https://github.com/tpetazzoni/libfoo.

Create an initial package for `libfoo` in `package/libfoo`, with the relevant minimal variables to get `libfoo` downloaded properly. Since it's hosted on *GitHub*, remember to use the `github` *make* function provided by Buildroot to define `<pkg>_SITE`. To learn more about this function, `grep` for it in the Buildroot tree, or read the Buildroot reference manual.

Also, notice that there is a version tagged `v0.1` in the GitHub repository, you should probably use it.

Enable the `libfoo` package and start the build. You should get an error due to the `configure` script being missing. What can you do about it? Hint: there is one Buildroot variable for *autotools* packages to solve this problem.

`libfoo` should now build fine. Look in `output/target/usr/lib`, the dynamic version of the library should be installed. However, if you look in `output/staging/`, you will see no sign of `libfoo`, neither the library in `output/staging/usr/lib` or the header file in `output/staging/usr/include`. This is an issue because the compiler will only look in `output/staging` for libraries and headers, so we must change our package so that it also installs to the *staging directory*.

Adjust your `libfoo.mk` file to achieve this, restart the build of `libfoo`, and make sure that you see `foo.h` in `output/staging/usr/include` and `libfoo.*` in `output/staging/usr/lib`.

Now everything looks good, but there are some more improvements we can do.

Improvements to `libfoo` packaging

If you look in `output/target/usr/bin`, you can see a program called `libfoo-example1`. This is just an example program for `libfoo`, it is typically not very useful in a real target system. So we would like this example program to not be installed. To achieve this, add a *post-install target hook* that removes `libfoo-example1`. Rebuild the `libfoo` package and verify that `libfoo-example1` has been properly removed.

Now, if you go in `output/build/libfoo-v0.1`, and run `./configure --help` to see the available options, you should see an option named `--enable-debug-output`, which enables a debugging feature of `libfoo`. Add a sub-option in `package/libfoo/Config.in` to enable the debugging feature, and the corresponding code in `libfoo.mk` to pass `--enable-debug-output` or `--disable-debug-output` when appropriate.

Enable this new option in `menuconfig`, and restart the build of the package. Verify in the build output that `--enable-debug-output` was properly passed as argument to the `configure` script.

Now, the packaging of `libfoo` seems to be alright, so let's get back to our `bar` application.

Finalize the packaging of `bar`

So, `bar` was failing to configure because `libfoo` was missing. Now that `libfoo` is available, modify `bar` to add `libfoo` as a dependency. Remember that this needs to be done in two places: `Config.in` file and `bar.mk` file.

Restart the build, and it should succeed! Now you can run the `bar` application on your target, and discover how absolutely useless it is, except for allowing you to learn about Buildroot packaging!

`bar` packaging: *libconfig* dependency

But there's some more things we can do to improve `bar`'s packaging. If you go to `output/build/bar-1.0` and run `./configure --help`, you will see that it supports a `--with-libconfig` option. And indeed, `bar`'s `README` file also mentions `libconfig` as an optional dependency.

So, change `bar.mk` to add *libconfig* as an optional dependency. No need to add a new `Config.in` option for that: just make sure that when *libconfig* is enabled in the Buildroot configuration, `--with-libconfig` is passed to `bar`'s *configure* script, and that *libconfig* is built before `bar`. Also, pass `--without-libconfig` when *libconfig* is not enabled.

Enable `libconfig` in your Buildroot configuration, and restart the build of `bar`. What happens?

It fails to build with messages like `error: unknown type name ‘config_t’`. Seems like the author of `bar` messed up and forgot to include the appropriate header file. Let's try to fix this: go to `bar`'s source code in `output/build/bar-1.0` and edit `src/main.c`. Right after the `#if defined(USE_LIBCONFIG)`, add a `#include <libconfig.h>`. Save, and restart the build of `bar`. Now it builds fine!

However, try to rebuild `bar` from scratch by doing `make bar-dirclean all`. The build problem happens again. This is because doing a change directly in `output/build/` might be good for doing a quick test, but not for a permanent solution: everything in `output/` is deleted when doing a `make clean`. So instead of manually changing the package source code, we need to generate a proper patch for it.

There are multiple ways to create patches, but we'll simply use Git to do so. As the `bar` project home page indicates, a Git repository is available on GitHub at `https://github.com/tpetazzoni/bar`.

Start by cloning the Git repository:

```
git clone https://github.com/tpetazzoni/bar.git
```

Once the cloning is done, go inside the `bar` directory, and create a new branch named `buildroot`, which starts the `v1.0` tag (which matches the `bar-1.0.tar.xz` tarball we're using):

```
git branch buildroot v1.0
```

Move to this newly created branch[1]:

```
git checkout buildroot
```

Do the `#include <libconfig.h>` change to `src/main.c`, and commit the result:

```
git commit -a -m "Fix missing <libconfig.h> include"
```

Generate the patch for the last commit (i.e the one you just created):

```
git format-patch HEAD^
```

and copy the generated `0001-*.patch` file to `package/bar/` in the Buildroot sources.

Now, restart the build with `make bar-dirclean all`, it should built fully successfully!

You can even check that `bar` is linked against `libconfig.so` by doing:

```
./output/host/usr/bin/arm-none-linux-gnueabihf-readelf -d output/target/usr/bin/\
    bar
```

On the target, test `bar`. Then, create a file called `bar.cfg` in the current directory, with the following contents:

```
verbose = "yes"
```

And run `bar` again, and see what difference it makes.

Congratulations, you've finished packaging the most useless application in the world!

Preparing for the next lab

In preparation for the next lab, we need to do a clean full rebuild, so simply issue:

```
make clean all 2>&1 | tee build.log
```

[1] Yes, we can use `git checkout -b` to create the branch and move to it in one command

