Getting started with Buildroot - Lab

Thomas Petazzoni, Bootlin

August 14, 2018

These lab instructions are written for the *Getting started with Buildroot* tutorial of the *Embedded Apprentice Linux Engineer* track. They are designed to work for the *PocketBeagle* hardware platform.

Initial configuration and build

Our first step to get started with Buildroot is obviously to build a minimal Linux system, with just a bootloader, Linux kernel and simple user-space.

The Buildroot at `https://github.com/e-ale/buildroot-e-ale` provides a ready-to-use *defconfig* to the PocketBeagle platform. However, for educational purposes, we are going to start straight from the official vanilla Buildroot, and build our configuration from scratch.

Getting Buildroot

Start by cloning Buildroot's official Git repository:

```
$ git clone git://git.busybox.net/buildroot
$ cd buildroot/
```

(Note: if download speed is too slow, you can use the `buildroot.tar.xz` tarball provided by the instructor)

We'll base our work on the 2018.02 release, which is the latest LTS release, so we create a branch based on the 2018.02 tag:

```
$ git checkout -b e-ale 2018.02
```

Creating a minimal configuration

Now it's time to start creating our configuration:

```
$ make menuconfig
```

In the configuration, we'll have to customize a number of options, as detailed below. Of course, take this opportunity to navigate in all the options, and discover what Buildroot can do.

- In *Target options*
  - Change *Target architecture* to *ARM (little endian)*
  - Change *Target architecture variant* to *Cortex-A8*
- In *Build options*, set *global patch directories* to `board/e-ale/pocketbeagle/patches/`. This will allow us to put patches for Linux, U-Boot other packages in subdirectories of `board/e-ale/pocketbeagle/patches/`.
- In *Toolchain*
  - Change *Toolchain type* to *External toolchain*. By default, Buildroot builds its own toolchain, but it can also use pre-built external toolchains. We'll use the latter, in order to save build time. Since we've selected an ARM platform, a Linaro toolchain is automatically selected, which will work for us.
- In *System configuration*, you can customize the *System host name* and *System banner* if you wish. Keep the default values for the rest.
- In *Kernel*
  - Enable the *Linux kernel*, obviously!
  - Choose *Custom version* as the *Kernel version*
  - Choose *4.14.24* as *Kernel version*
  - Patches will already be applied to the kernel, thanks to us having defined a *global patch directory* above.
  - Choose `omap2plus` as the *Defconfig name*
  - We'll need the Device Tree of the PocketBeagle, so enable *Build a Device Tree Blob (DTB)*
  - And use `am335x-pocketbeagle` as the *Device Tree Source file names*
- In *Target packages*, we'll keep just Busybox enabled for now. In the next sections, we'll enable more packages.
- In *Filesystem images*, enable *ext2/3/4 root filesystem*, select the *ext4* variant. You can also disable the *tar* filesystem image, which we won't need.
- In *Bootloaders*, enable *U-Boot*, and in *U-Boot*:
  - Switch the *Build system* option to *Kconfig*: we are going to use a modern U-Boot, so let's take advantage of its modern build system!
  - Use a *Custom version* of value *2018.01*. You'll notice that the current default is already *2018.01*. However, Buildroot upstream is regularly updating this to the latest U-Boot version. However, to have a reproducible setup, we really want to use a fixed version.
  - Use `am335x_pocketbeagle` as the *Board defconfig*

- The *U-Boot binary format* should be changed from `u-boot.bin` to `u-boot.img`. Indeed, this second stage bootloader will be loaded by a first stage bootloader, and needs to have the proper header to be loaded by the first stage.
- Enable *Install U-Boot SPL binary image* to also install the first stage bootloader. Its name in *U-Boot SPL/TPL binary image name(s)* should be changed to `MLO` since that's how U-Boot names it, and how the AM335x expects it to be named.

As you have noticed, in the configuration, we have referenced `board/e-ale/pocketbeagle/patches` as a directory containing patches for various packages. We now need to add the U-Boot and Linux patches that add support for the *PocketBeagle*, which are not upstream yet. They are available from the `patches/` folder in the USB stick provided by your instructor, just copy it to `board/e-ale/pocketbeagle` so that you get the following directory hierarchy:

```
$ tree board/e-ale/pocketbeagle/
board/e-ale/pocketbeagle/
- patches
  - linux
    - 0001-Stripped-back-pocketbeagle-devicetree.patch
  - uboot
    - 0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
    - 0002-U-Boot-BeagleBone-Cape-Manager.patch
    - 0003-pocketbeagle-tweaks.patch
```

Running the build

To start the build, you can run just `make`. But it's often convenient to keep the build output in a log file, so you can do:

```
$ make 2>&1 | tee build.log
```

or alternatively use a wrapper script provided by Buildroot:

```
$ ./utils/brmake
```

The build will take a while (about 14-15 minutes on your instructor's machine). If the Internet connectivity is slow, the USB stick that your instructor has provided contains the files to be downloaded. Just copy them into the `dl/` folder, and restart the build.

The overall build takes quite some time, because the Linux kernel configuration `omap2plus_defconfig`, which supports all OMAP2, OMAP3, OMAP4 and AM335x platforms has a *lot* of drivers and options enabled. It would definitely be possible to make a smaller kernel configuration for the *Pocket Beagle*, reducing the kernel size and boot time.

At the end of the build, the output is located in `output/images`. We have:

- `MLO`, the first stage bootloader

- `u-boot.img`, the second stage bootloader
- `zImage`, the Linux kernel image
- `am335x-pocketbeagle.dtb`, the Linux kernel Device Tree Blob
- `rootfs.ext4`, the root filesystem image

However, that doesn't immediately give us a bootable SD card image. We could create it manually, but that wouldn't be really nice. So move on to the next section to see how Buildroot can create the SD card image for you.

Creating an SD card image

To create an SD card image, we'll use a tool called `genimage`, which provided a configuration file, will output the image of a block device, with multiple partitions, each containing a filesystem. See `https://git.pengutronix.de/cgit/genimage/tree/README.rst` for some documentation about *genimage* and its configuration file format.

`genimage` needs to be called at the very end of the build. To achieve this, Buildroot provides a mechanism called *post-image scripts*, which are arbitrary scripts called at the end of the build. We will use it to create an SD card image with:

- A FAT partition containing the bootloader images, the kernel image and Device Tree
- An ext4 partition containing the root filesystem

In addition, the U-Boot bootloader for the *PocketBeagle* is configured by default to load a file called `uEnv.txt` to indicate what should be done at boot time. This file should also be stored in the first partition of the SD card.

So, go back to `make menuconfig`, and adjust the following options:

- In *System configuration*
  - Set *Custom scripts to run after creating filesystem images* to `board/e-ale/pocketbeagle/post-image.sh`
- In *Host utilities*, enable `host genimage`, `host mtools` and `host dosfstools`. *mtools* and *dosfstools* are needed because our *genimage* configuration includes the creation of a FAT partition.

Copy `genimage.cfg`, `post-image.sh` and `uEnv.txt` from the USB stick provided by your instructor to `board/e-ale/pocketbeagle`.

Restart the build again. Once the build is finished, you should now have an `sdcard.img` file in `output/images/`, ready for flashing:

```
$ sudo dd if=output/images/sdcard.img of=/dev/mmcblk0 bs=1M
```

Insert the SD card in the PocketBeagle, and boot it. Provided you have a serial port connection at 115200 bps, you should see the system booting. You can login as `root` with no password, and

explore the (very minimal) system. The system weights 18.4 MB, of which 12.1 MB are kernel modules.

Storing our Buildroot configuration

Our Buildroot configuration is currently stored as `.config`, which is not under version control and would be removed by a `make distclean`. So, let's store it as a *defconfig* file:

```
$ make BR2_DEFCONFIG=configs/eale_pocketbeagle_defconfig savedefconfig
```

And then look at `configs/eale_pocketbeagle_defconfig` to see what your configuration looks like.

Network over USB and SSH connection

In this section, we will setup network over USB and add an SSH server to our build.

To enable network over USB, we need to change two things to our root filesystem:

- Add an init script that will load the necessary kernel modules, and do the appropriate `configfs` tweaks to set up the USB gadget device. This script will be installed in `/etc/init.d/S30usbgadget` in the root filesystem.
- Customize the `/etc/network/interfaces` file to set up the `usb0` network interface.

In order to achieve this, we are going to use the concept of *root filesystem overlay* of Buildroot. Filesystem overlays are simply directories that get copied over the root filesystem at the end of the build.

Go to the Buildroot configuration, and in *System configuration*, change *Root filesystem overlay directories* to `board/e-ale/pocketbeagle/overlay/`. In the same menu set the *Root password* to a non-empty string of your choice. Finally in the *Target packages* menu, *Networking applications* submenu, enable `dropbear`.

Then, copy the contents of the `overlay/` directory available on the USB stick provided by your instructor to `board/e-ale/pocketbeagle/overlay/`. Take this opportunity to have a look at the contents of the overlay.

Restart the build, reflash your system, and boot it. You should see the USB Gadget being configured at boot time. Its IP address is `192.168.42.2`. On your PC, assign `192.168.42.1` as the IP address of the USB network interface that just appeared, and then you can connect over SSH to your PocketBeagle:

```
$ ssh root@192.168.42.2
```

Developing an application

Adding and testing *libgpiod*

Now, we will improve our application to make it use GPIOs through the *gpiod* library. To achieve this:

- Enable `libgpiod` and its tools from Buildroot `menuconfig`
- Enable the `CONFIG_PINCTRL_MCP23S08` option in the Linux kernel configuration, by running `make linux-menuconfig`. This option enables the driver of the GPIO expander used on the *BaconBits* add-on board. Note that this change to the Linux kernel configuration will be lost if you do a `make clean`. To make things persistent, the nicest solution is to add `CONFIG_PINCTRL_MCP23S08=y` to a new config fragment file in `board/eale/pocketbeagle/` and reference this fragment from `BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES` Buildroot option. This way your Linux kernel configuration is the standard `omap2plus_defconfig` plus your tweaks.

Re-build your system and reflash it. You should now be able to play with GPIOs with the *libgpiod* tools. The syntax of the `gpioset` program is `gpioset <chip name/number> <offset1> =<value1>`:

```
# gpioset gpiochip4 0=0
# gpioset gpiochip4 1=0
# gpioset gpiochip4 1=1
# gpioset gpiochip4 0=1
```

Cross-compiling a simple application

The USB stick provided by your instructor contains a small application that uses *libgpiod*, in the `eale-gpio-app` folder. Copy this folder side by side to Buildroot:

```
- buildroot
  - arch
  - output
  - ...
- eale-gpio-app
  - eale-gpio-app.c
  - Makefile
```

For now, we'll build it manually:

```
$ ./output/host/bin/arm-linux-gnueabihf-gcc -o eale-gpio-app \
    ../eale-gpio-app/eale-gpio-app.c -lgpiod
```

Transfer it to the target:

```
$ scp eale-gpio-app root@192.168.42.2:
```

And then run it on the target:

```
# ./eale-gpio-app
```

A package for our application

Compiling your application manually is fine during development and if your application is simple, but it would be better to have Buildroot build it for us. To achieve this, we need to create a new Buildroot package.

Start by creating the `package/eale-gpio-app` directory. Inside this directory, place a `Config.in` file containing:

```
config BR2_PACKAGE_EALE_GPIO_APP
        bool "eale-gpio-app"
        depends on BR2_TOOLCHAIN_HEADERS_AT_LEAST_4_8 # libgpiod
        select BR2_PACKAGE_LIBGPIOD
        help
          This is the E-ALE GPIO demo application.

comment "eale-gpio-app needs kernel headers >= 4.8"
        depends on !BR2_TOOLCHAIN_HEADERS_AT_LEAST_4_8
```

What this file does is:

- Describe an option that will be visible in *menuconfig* to enable/disable the compilation of our demo application
- *Selects* the libgpiod package, because it is a dependency of our demo application.
- The `BR2_TOOLCHAIN_HEADERS_AT_LEAST_4_8` dependency is inherited from *libgpiod*. *libgpiod* requires at least kernel headers from Linux 4.8, and therefore our application also has this requirement.

Then, edit `package/Config.in` and add a new line in the *Hardware handling* category to include the newly created `package/eale-gpio-app/Config.in`.

Now, we need to describe how to build this package, by adding `package/eale-gpio-app/eale-gpio-app.mk` with the following contents:

```
################################################################################
#
# eale-gpio-app
#
################################################################################

EALE_GPIO_APP_SITE = $(TOPDIR)/../eale-gpio-app
EALE_GPIO_APP_SITE_METHOD = local
EALE_GPIO_APP_DEPENDENCIES = libgpiod

define EALE_GPIO_APP_BUILD_CMDS
    $(TARGET_MAKE_ENV) $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D)
endef

define EALE_GPIO_APP_INSTALL_TARGET_CMDS
    $(TARGET_MAKE_ENV) $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D) \
        DESTDIR=$(TARGET_DIR) install
endef

$(eval $(generic-package))
```

Make sure that the lines inside `EALE_GPIO_APP_BUILD_CMDS` and `EALE_GPIO_APP_INSTALL_TARGET_CMDS` are prefixed with one tab.

What this file does is:

- Describe where the source code for our application is located. Here, we use the *local* site method, to indicate that the source code is locally available (as opposed to a tarball available on a HTTP server)
- Describe the application dependency. This ensures that *libgpiod* gets built before our application.
- Describe how to build and install our application, by using the *Makefile* provided by the application. `TARGET_MAKE_ENV` and `TARGET_CONFIGURE_OPTS` are variables provided by Buildroot, which define standard variables such as `CC`, `CFLAGS`, etc. to make sure the application gets cross-compiled.

With this in place, go to *menuconfig* and enable your application. Restart the build with `make`. You should now see your application being built and installed to the root filesystem. Reflash the SD card image, and verify that your application is in `/usr/bin/` on the target, and that it works as expected. If it does, congratulations!

Going further

Using Linux kernel configuration fragment

When we enabled support for the MCP23S08 GPIO expander, we kept the Linux kernel configuration change only in `output/build/linux-4.14.24`, which means it will be lost when we will clean the build. To fix this:

- Add `CONFIG_PINCTRL_MCP23S08=y` to a new file called `board/e-ale/pocketbeagle/linux.frag`
- In *menuconfig*, set `BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES` to `board/e-ale/pocketbeagle/linux.frag`

Then, do a complete rebuild of your kernel: `make linux-dirclean` followed by `make`.

Generate legal information

To help with license compliance, Buildroot can generate a legal report:

```
$ make legal-info
```

Then look in `output/legal-info`, where you will find the license text for all packages, tarballs, as well as a manifest in the form of a CSV file (`manifest.csv`).

Analyzing the build

Sometimes, it is useful to analyze the size of the filesystem or the build duration. Buildroot provides convenient tooling to achieve this. For the data to be correct, we need to have a completely clean build:

```
$ make clean all
```

Once the build is done, run `make graph-size` and look at `output/graphs/graph-size.pdf`. Then run `make graph-build` and look at `output/graphs/build.hist-build.pdf`.