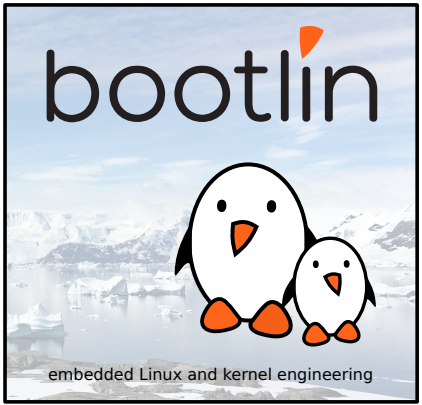
Getting started with Buildroot

Thomas Petazzoni
thomas.petazzoni@bootlin.com



Thomas Petazzoni

- CTO/Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
  - Freely available training materials
- Co-maintainer of **Buildroot**
- Living in **Toulouse**, France

Building an embedded Linux system

Pre-built binary Linux distributions

+ Readily available
- Large, usually 100+ MB
- Not available for all architectures
- Not easy to customize
- Generally require native compilation

Building an embedded Linux system

Manual system building

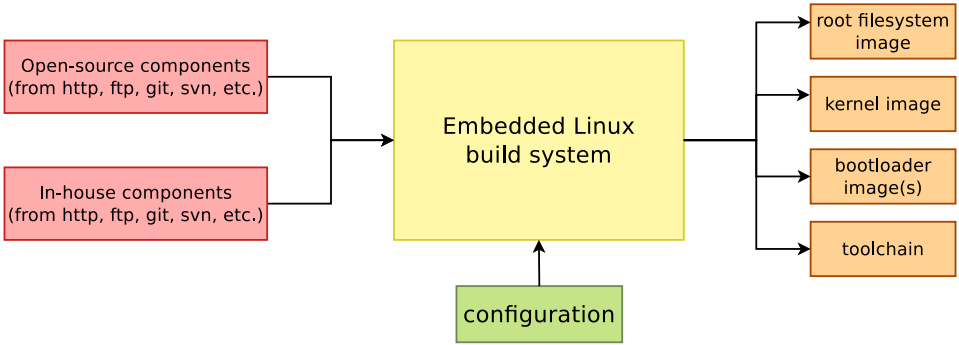
+ Smaller and flexible
- Very hard to handle cross-compilation and dependencies
- Not reproducible
- No benefit from other people's work

Building an embedded Linux system

Embedded Linux build systems

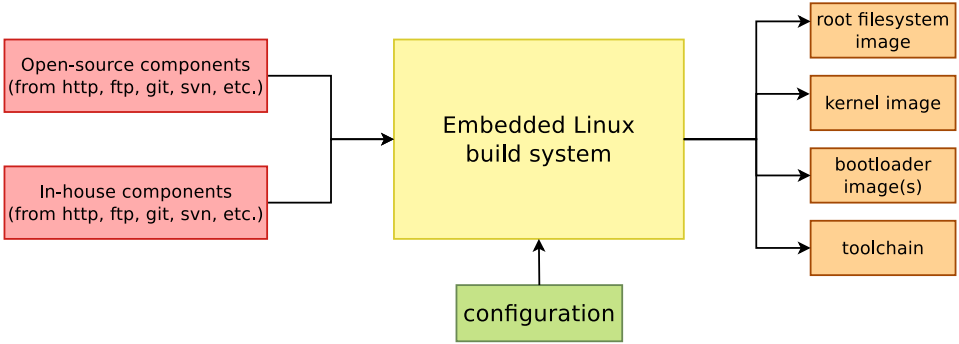
+ Small and flexible
+ Reproducible, handles cross-compilation and dependencies
+ Available for virtually all architectures
- One tool to learn
- Build time

Embedded Linux build system: principle



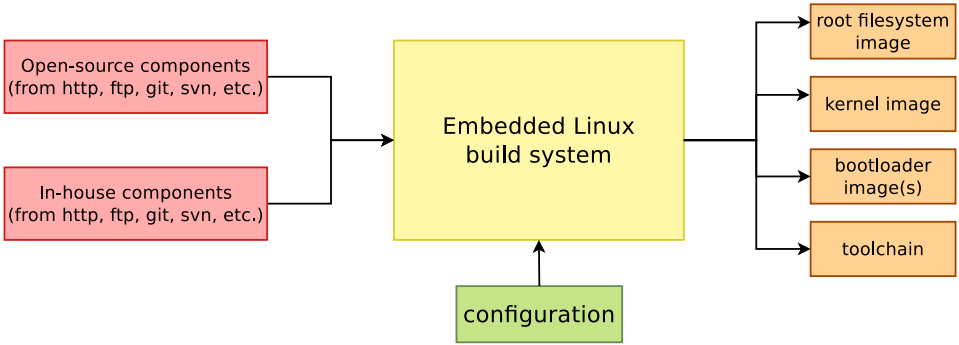
- Building from source → lot of flexibility

Embedded Linux build system: principle



- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines

Embedded Linux build system: principle



- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines
- Recipes for building components → easy

Buildroot at a glance

- Is an **embedded Linux build system**, builds from source:
  - cross-compilation toolchain
  - root filesystem with many libraries/applications, cross-built
  - kernel and bootloader images
- **Fast**, simple root filesystem in minutes
- **Easy** to use and understand: kconfig and make
- **Small** root filesystem, default 2 MB
- More than **2500 packages** available
- Generates filesystem images, not a distribution
- Vendor neutral
- Active community, stable releases every 3 months
- Started in 2001, oldest still maintained build system
- `http://buildroot.org`

Today's lab

- **Step 1**: do a minimal build for the PocketBeagle, with just a bootloader, Linux kernel and minimal root filesystem. Generate a ready-to-use SD card image.
- **Step 2**: enable network over USB and SSH connectivity using Dropbear. Shows how to use a rootfs overlay and how to add packages.
- **Step 3**: customize the Linux kernel configuration, compile a small application that uses the GPIO, first manually, and then using a new Buildroot package
- **Don't hesitate to request help and ask questions!**

Step 1
Minimal build for the PocketBeagle

Getting Buildroot

Cloning Buildroot

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

Note: if cloning is too slow, you can use `buildroot.tar.xz` from the USB stick.

Create a branch based on the latest LTS

```
$ git checkout -b ede-lab 2019.02.6
```

Get one U-Boot integration improvement

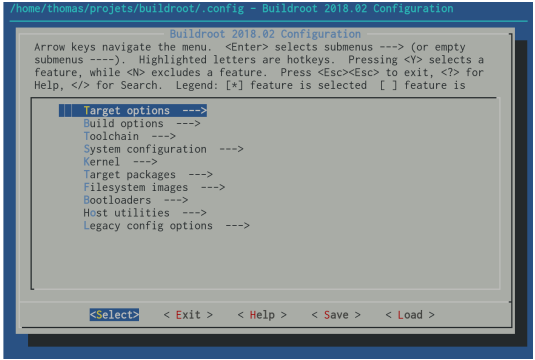
We need to cherry-pick one more recent commit, to be able to pass arbitrary variables to the U-Boot build:

```
$ git cherry-pick cc151c3993090a52d1fef8532f52d74ee6d924c9
```

Starting the configuration

- Buildroot has a large number of pre-defined configuration files for popular HW platforms:
  `make list-defconfigs`
- For learning purposes, we are going to create our own configuration from scratch for the Pocket Beagle

```
$ make menuconfig
```



menuconfig: Target options

- **Target architecture**: *ARM (little endian)*
- **Target architecture variant**: *Cortex-A8*

Build options

- **Global patch directories**: `board/e-ale/pocketbeagle/patches/`
- We will need to apply one patch to Linux to improve the PocketBeagle Device Tree description. Buildroot automatically applies patches from *global patch directories* subfolders named after Buildroot packages.
- We will add those patches once we are done with the configuration.

menuconfig: Toolchain

- **Toolchain type**: *External toolchain*
- Buildroot supports:
  - *Internal toolchain*: Buildroot builds a cross-compilation toolchain from scratch. Flexible, but additional build time needed.
  - *External toolchain*: Buildroot downloads and uses a pre-built cross-compilation toolchain.
- On ARM, the ARM-provided toolchain (*Arm ARM 2018.11*) is automatically chosen by default as an external toolchain.

menuconfig: System configuration

- **System host name**: `ede`
- **System banner**: `Hello EDE`
- **Init system**: keep the default of `Busybox`, Buildroot also supports systemd, sysvinit.
- **/dev management**: keep the default of `devtmpfs`, Buildroot also supports udev, systemd, etc.

menuconfig: Kernel

- Enable **Linux Kernel**
- **Kernel version**: *Custom version*
- **Kernel version**: *5.3*
- No need to specify a path to patches, we are already using the *global patch directory mechanism*
- **Defconfig name**: `omap2plus`
- Enable **Build a Device Tree Blob**
- **In-tree Device Tree Source file names**: `am335x-pocketbeagle`

menuconfig: Target packages

- Since **Busybox** is chosen as init system, it is already forcefully selected.
- **Busybox** provides all we need for a minimal Linux system, so no need to enable other packages.
- You can have a look at the choice of 2500+ packages, we'll use a few more in the next steps.

menuconfig: Filesystem images

- Enable **ext2/3/4 root filesystem**, and select the `ext4` variant
- Disable **tar the root filesystem**

menuconfig: Bootloaders

- Enable **U-Boot**
- **Build system**: *Kconfig*. We use a modern U-Boot!
- **Custom version**: *2019.10*
  - Using a fixed version instead of the default version allows to ensure our build will always build that specific version we have tested.
- **Board defconfig**: `am335x_evm`
- **U-Boot binary format**: `u-boot.img`. Indeed, U-Boot itself will be the second stage bootloader.
- Enable **U-Boot needs dtc**
- Enable **Install U-Boot SPL binary image**. This enables building the first stage bootloader.
- Set **U-Boot SPL/TPL binary image name** to *MLO*, the name required on AM335x.
- Set **Custom make options** to `DEVICE_TREE=am335x-pocketbeagle`

Adding the patches

- Before starting the build, we need to add the Linux kernel and U-Boot patches needed to support the Pocket Beagle.
- Create the folder `board/ede/pocketbeagle/patches/` that we referenced as a *global patch directory*
- Copy to this folder the contents of the `patches/` folder of the USB stick.

```
$ tree board/e-ale/pocketbeagle/
board/e-ale/pocketbeagle/
- patches
  - linux
    - 0001-ARM-dts-describe-the-MCP23S18-connected-on-Pocket-Be.patch
```

Pre-populating the download folder

- By default, Buildroot caches all the downloaded tarballs in `dl/`
- In order to speed up the build process and avoid long download times, we are going to pre-populate this download folder.
- Copy the `dl/` folder from the USB stick to the Buildroot source directory.

Doing the build

To run the build, do:

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

This allows to complete log of the build output.
Alternatively, there is a wrapper script provided by Buildroot:

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

The build will take a while (14-15 minutes on your instructor machine), because the `omap2plus_defconfig` kernel configuration has a LOT of features enabled.

During the build: exploring the build output

- All the output produced by Buildroot is stored in `output/`
- Can be customized using `O=` for out-of-tree build
- `output/` contains
  - `output/build`, with one sub-directory for the source code of each component
  - `output/host`, which contains all native utilities needed for the build, including the cross-compiler
  - `output/host/<tuple>/sysroot`, which contains all the headers and libraries built for the target
  - `output/target`, which contains *almost* the target root filesystem
  - `output/images`, the final images
- `dl/` contains downloaded artefacts, can be customized by the `BR2_DL_DIR` env. variable

During the build: summarized build process

1. Check core dependencies
2. For each selected package, after taking care of its dependencies: download, extract, patch, configure, build, install
   - To `target/` for target apps and libs
   - To `host/<tuple>/sysroot` for target libs
   - To `host/` for native apps and libs
   - Filesystem skeleton and toolchain are handled as regular packages
3. Copy *rootfs overlay*
4. Call *post build* scripts
5. Generate the root filesystem image
6. Call *post image* scripts

Build finished: what do we have ?

- `ls output/images`
  - `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
- → this doesn't give us a bootable SD card image
- Let's create one using *genimage*

SD card image with genimage: intro

- Tool from *Pengutronix*
- Given a configuration file, creates a block device image with a partition table and filesystems
- Need to be called at the very end of the build: we will run it in a *post image* script
- Our SD card image will have:
  - One FAT partition with bootloader, kernel image and Device Tree
  - One ext4 partition with the root filesystem

SD card image with genimage: configuration

- In `menuconfig`
  - **System configuration → Custom scripts to run after creation filesystem images**, set to `board/ede/pocketbeagle/post-image.sh`
  - **Host utilities**, enable `host genimage`, `host mtools`, `host dosfstools`. This will make sure those tools are built for the build machine
- Copy `genimage.cfg`, `post-image.sh` and `uEnv.txt` from the USB stick to `board/ede/pocketbeagle`
  - `genimage.cfg`, *genimage* configuration file, have a look at it
  - `post-image.sh`, shell script that calls *genimage* with the appropriate arguments
  - `uEnv.txt`, U-Boot script to boot the system

SD card image: building and booting

- Start the build again with `make`
- It will only build a few additional tools (genimage, etc.) and produce the SD card image
- The SD card image will be `output/images/sdcard.img`
- Transfer to your SD card

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

- Insert the SD card in the PocketBeagle, and boot it (serial port at 115200 bps)
- Login as `root`, no password.
- System weights 18.4 MB, of which 12.1 MB are kernel modules

Storing our Buildroot configuration persistently

- Our current configuration is stored in `.config`
  - Will be lost upon `make distclean`
  - Or if we create a different configuration for a different project
- Save it as a *defconfig* file:

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

- And have a look at `configs/ede_pocketbeagle_defconfig`

Step 2
Network connectivity over USB: SSH

Network configuration: using a rootfs overlay

- To enable network over USB, we will need:
  - An init script that loads the relevant kernel modules and uses *configfs* to set up the USB gadget device. It will be installed as `/etc/init.d/S30usbgadget`.
  - A customized `/etc/network/interfaces` file
- We will use a **root filesystem overlay** to add those customizations to the root filesystem.
- In `menuconfig`
  - **System configuration**, set **Root filesystem overlay directories** to `board/ede/pocketbeagle/overlay/`
  - **System configuration**, set **Root password** to a non-empty one
  - **Target packages**, **Networking applications**, enable `dropbear`
- Copy the contents of `overlay/` from the USB stick to `board/ede/pocketbeagle/overlay/`, and of course, have a look at it!

Using the network

- Restart the build

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

- Reflash the SD card
- Boot
- On your PC, configure the new network interface with the IP address 192.168.42.1
- If you're using network manager:

```
$ nmcli con add con-name buildroot-target type ethernet \
    ifname enp57s0u1u6u4 ip4 192.168.42.1/24
```

- SSH into the board, from your PC:

```
$ ssh root@192.168.42.2
```

Step 3
Developing an application

Adding libgpiod and GPIO support

- We are going to write a demo application based on the `libgpiod` library, to manipulate GPIOs.
- Let's first add this library in your Buildroot configuration, in `menuconfig`, enable `BR2_PACKAGE_LIBGPIOD` and its tools.
- We also need to enable the GPIO kernel driver, using the option `CONFIG_PINCTRL_MCP23S08`. To do this, run `make linux-menuconfig`, which will open up the Linux kernel *menuconfig*, and enable the driver.
  - Note: this change would be lost during a `make clean`. To make it persistent, using a configuration file fragment would be appropriate.
- Run `make` to rebuild the system, reflash on the SD card, and reboot

Testing GPIO support

- We can list the list the GPIO chips using `gpiodetect`:

```
# gpiodetect
```

- We can use the GPIOs using `gpioset`:

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

Cross-compiling a simple application

- Let's now build a simple application to see how to use the Buildroot cross-compiler
- Copy the `ede-gpio-app` folder from the USB stick side-by-side with Buildroot:

```
- buildroot/
- ede-gpio-app/
  - ede-gpio-app.c
  - Makefile
```

- To build the application:

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

Running the application on the target

- Copy it to the target:

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

- And run it:

```
# ./ede-gpio-app
```

Buildroot package for our application

- Building manually is fine for quick experiments, but we definitely want the build process to be fully automated by Buildroot.
- In order to do this, one can create new **packages** for the different applications/libraries that should be compiled and installed in the target root filesystem.
- A package in Buildroot consists of:
  - `package/<pkg>/Config.in`, describing the configuration options, in the *kconfig* syntax
  - `package/<pkg>/<pkg>.mk`, describing how to download, build and install the package, written in *make*
  - `package/<pkg>/<pkg>.hash`, containing hashes to validate that the downloaded have the expected contents
  - `package/<pkg>/*.patch`, patches to apply to the package source code, if needed

Creating the package: `Config.in`

package/ede-gpio-app/Config.in

```
config BR2_PACKAGE_EDE_GPIO_APP
        bool "ede-gpio-app"
        depends on BR2_TOOLCHAIN_HEADERS_AT_LEAST_4_8 # libgpiod
        select BR2_PACKAGE_LIBGPIOD
        help
          This is the EDE GPIO demo application.

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

Creating the package: including `Config.in`

package/Config.in

```
...
        source "package/ede-gpio-app/Config.in"
...
```

You can run `make menuconfig`, and see that your new option is there!

Creating the package: ede-gpio-app.mk

package/ede-gpio-app/ede-gpio-app.mk

```
################################################################################
#
# ede-gpio-app
#
################################################################################

EDE_GPIO_APP_SITE = $(TOPDIR)/../ede-gpio-app
EDE_GPIO_APP_SITE_METHOD = local
EDE_GPIO_APP_DEPENDENCIES = libgpiod

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

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

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

Creating the package: using it

- Enable your new package in `make menuconfig`
- Run the build with `make`
- Reflash your SD card and reboot
- Your new application is in `/usr/bin` on the target

Documentation and support

- Extensive manual: `https://buildroot.org/downloads/manual/manual.html`
- 3-day training course, with freely available materials: `https://bootlin.com/training/buildroot/`
- Mailing list: `http://lists.busybox.net/pipermail/buildroot/`
- IRC channel: `buildroot` on Freenode