MASTER OF SCIENCE
IN ENGINEERING

Buildroot

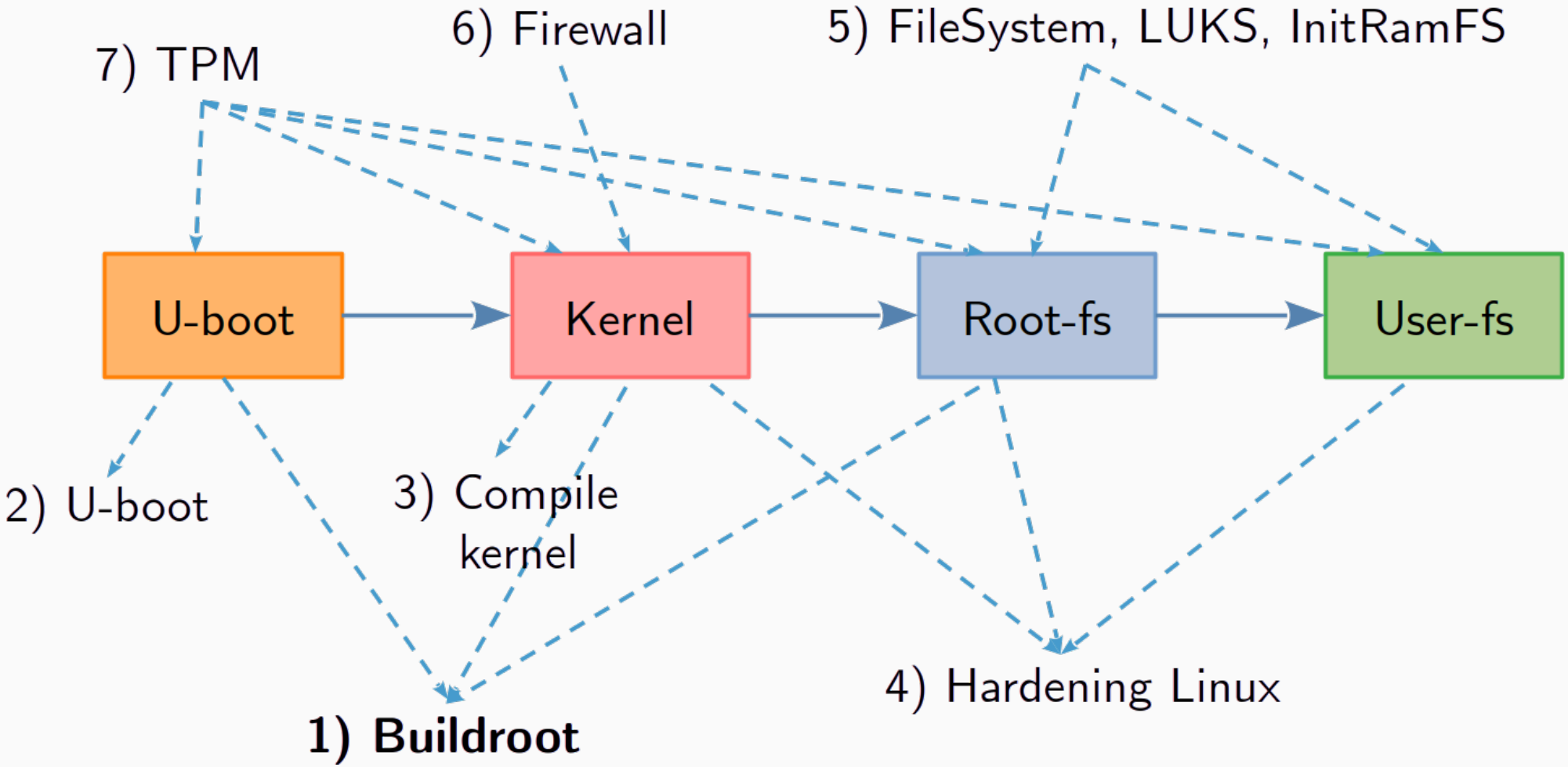
Jean-Roland Schuler, Florent Glück

October 20, 2025

Credits

A big thanks to Bootlin for creating great Buildroot slides, which many are the basis of this course's content.

Overview


7) TPM
6) Firewall
5) FileSystem, LUKS, InitRamFS
U-boot
Kernel
Root-fs
User-fs
2) U-boot
3) Compile kernel
4) Hardening Linux
1) Buildroot

The challenge of building an embedded Linux system

How to generate and integrate all software components (bootloader, kernel, libraries, tools, applications) into a full working system?

Solutions

- Manual system build:
  - difficult
  - time consuming
  - does not scale
  - hard to maintain
- Automated build systems

Building an embedded Linux system

- Different software framework exist to build Linux embedded systems: Yocto, Buildroot, PTXdist, OpenWRT, etc.
- The most widely used solutions:
  - **Yocto**: builds a complete Linux distribution with binary packages
    - Most powerful, but quite complex, and steep learning curve
  - **Buildroot**: builds a system image (or only certain parts) without binary packages
    - Much simpler to use, understand and modify

Buildroot (BR)

Buildroot is a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation.

Can handle everything

Cross-compilation toolchain, root filesystem generation, kernel image compilation and bootloader compilation.

Is very easy

Thanks to its kernel-like menuconfig, gconfig and xconfig configuration interfaces, building a basic system with Buildroot is easy and typically takes 15-30 minutes.

Supports several thousand packages

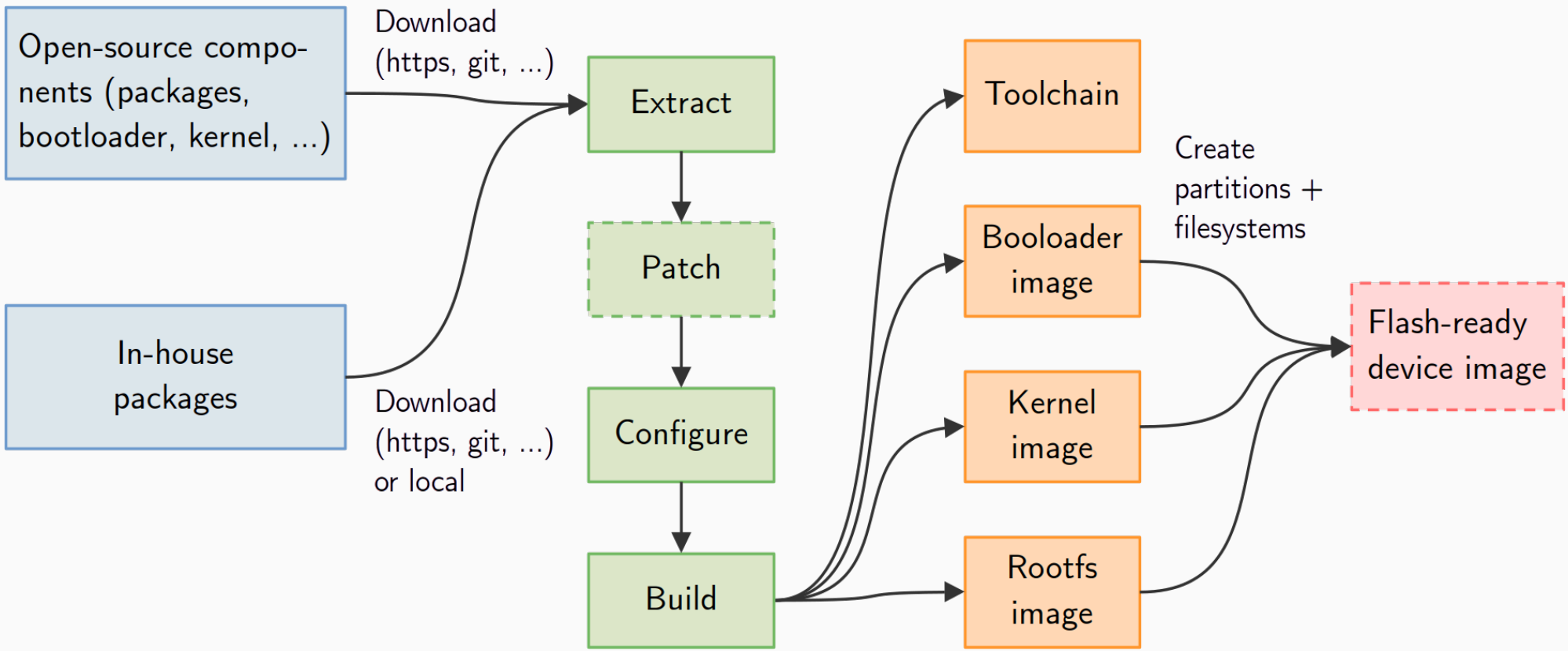
X.org stack, Gtk3, Qt 5, GStreamer, Webkit, Kodi, a large number of network-related and system-related utilities are supported.

BR at a glance

- Can build: toolchain, bootloader, kernel, rootfs[1], host tools, etc.
- Easy to configure: `make menuconfig`, `make xconfig`, etc.
- Easy to understand, well-known technologies: `make` and `Kconfig`
- Fast: builds a simple rootfs in a few minutes
- Small rootfs, starting at 2MB
- Extensive documentation
- 3200+ packages (libraries & apps)
- Many architectures supported
- Active community with regular releases
- Vendor neutral

[1] Abbreviation for "root filesystem"

BR general workflow



BR directories organization (1/2)

`arch`	Recipes for the various architectures
`board`	Board-specific patches and config files
`boot`	Bootloaders recipes, Makefile and config for bootloaders
`configs`	Config files for the various platforms
`dl`	Default directory where packages are downloaded
`docs`	Documentation files
`fs`	Recipes for generating rootfs images in various formats

BR directories organization (2/2)

`linux`	Recipes and patches for building the Linux kernel
`output`	Generate files: bootloaders, kernel, rootfs, host tools, etc.
`package`	Packages (host and target); contains recipes, patches and config to compile apps and libs
`support`	Miscellaneous scripts and tools
`system`	System recipes and rootfs skeleton
`toolchain`	Recipes, patches and config to generate a toolchain

BR configuration

- To display all possible `make` targets:

```
make help
```

- To display pre-defined configurations for a number of platforms:

```
make list-defconfigs
```

  - Platforms configurations are located in the configs directory

- To load a pre-defined configuration, e.g. `xyz_defconfig`:

```
make xyz_defconfig
```

  1. Sets the current config as being `xyz_defconfig`
  2. Copies the `xyz_defconfig` file to `.config`

BR configuration and build

Edit the current config; when exiting, saves it to `.config`:

```
make menuconfig
```

Save the current config (`.config`) into the last loaded configuration file (by default in `configs` directory):

```
make savedefconfig
```

Configure U-Boot:

```
make uboot-menuconfig
```

Configure the Linux kernel:

```
make linux-menuconfig
```

Configure Busybox:

```
make busybox-menuconfig
```

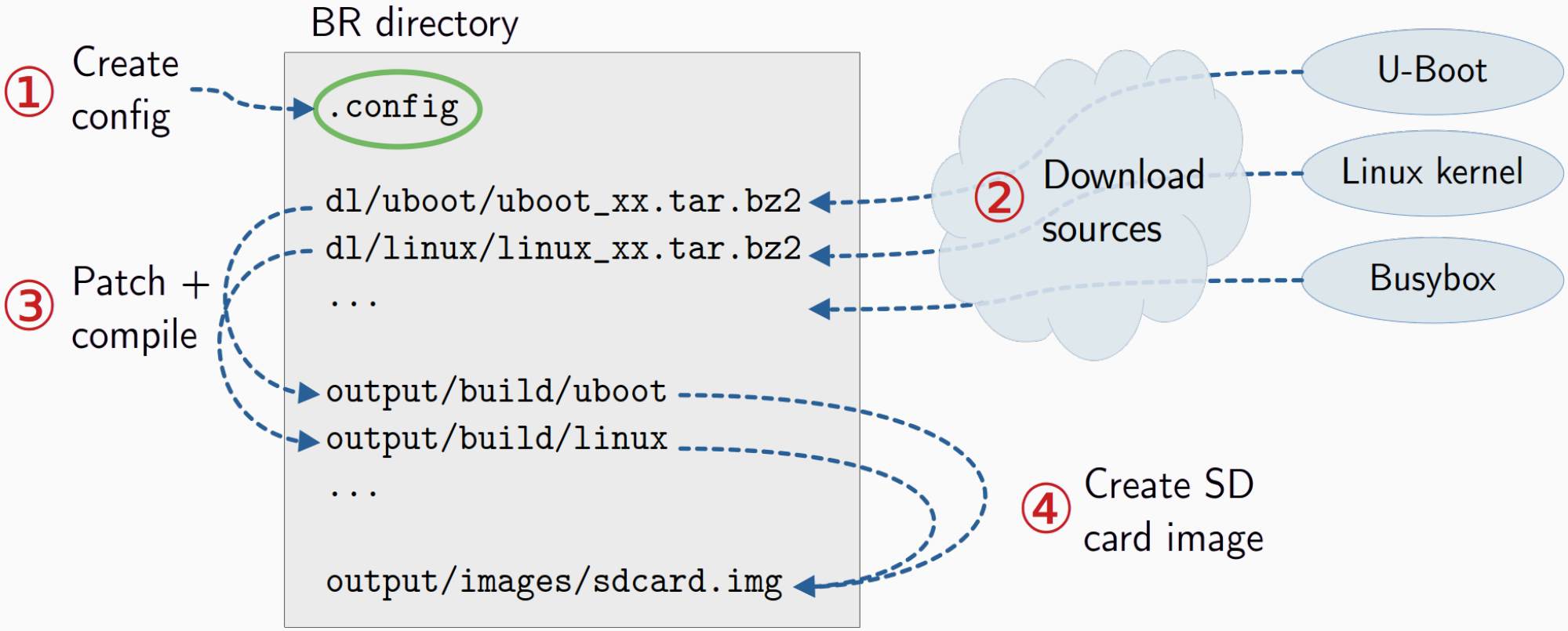
Compile everything:

```
make
```

Compile only U-Boot

```
make uboot-rebuild
```

BR workflow example (simplified)



BR workflow example, detailed

1. The `.config` file containing the platform config is read
2. Compressed archives of sources packages are downloaded to `dl`
3. Sources are extracted to `output/build`, each in their own directory
4. Packages sources can be patched (either from the global patch directory for the platform (e.g. `board/friendlyarm/nanopi-neo-plus2/patches`), or using specific patches located in each package directory, e.g. `package/busybox`)
5. Packages are configured (config depends on the type of package)
6. Packages are compiled (depends on the build system: make, cmake, autotools, etc.)
7. The rootfs is created after having applied the specified overlay in `board/friendlyarm/nanopi-neo-plus2/rootfs_overlay/`
8. The `post-build.sh` script is executed, which:
   - Creates `sdcard.img` using `genimage.sh` and `genimage.cfg`
     - these scripts and config file are located in `board/friendlyarm/nanopi-neo-plus2`

Root filesystem customization

- Default target's rootfs defined by a **skeleton** file hierarchy defined in `system/skeleton`
- To modify this rootfs hierarchy, use the **overlay** mechanism
- Principle:
  - Create your own file hierarchy into an **overlay** directory, e.g. `myrootfs`
  - Final rootfs obtained by **merging** the **skeleton** directory with the **overlay** (`myrootfs`) directory
  - If same files/directories are present in both the **overlay** and the **skeleton** → **overlay** files **override** **skeleton** files
- To specify an **overlay** directory to use: `make menuconfig` → `System configuration` → `Root filesystem overlay directories`

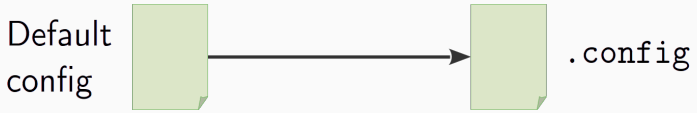
Package configuration

- Configuration is package-dependent
- Most common configuration types:
  - U-Boot, Linux, Busybox use `.config` files
  - Openssh, binutils, etc. use a `configure` script
    – mechanism part of the “autotools” build system
  - Some packages implement the configuration steps directly into a Makefile

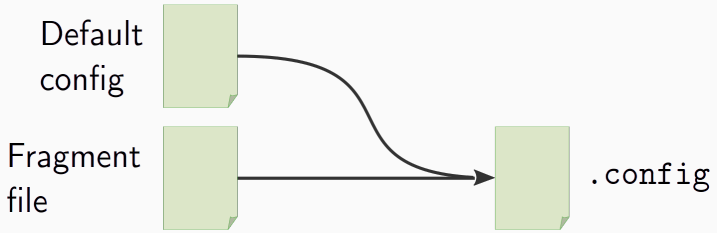
Configuring .config

Several possibility to configure the first `.config` file:

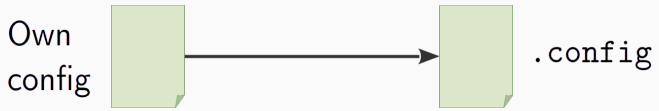
- Use the package default config



- If few modifications: use a **fragment** file
  - The fragment file is merged with the default config



- If many modifications: use your own config file
  - Possible to have a fragment file too



Configuration with fragment files

- Configuration with fragment files is supported by:
  - Linux kernel, U-Boot, Barebox, Busybox, uClibc
- Principle to create a fragment file:
  (1) Save the current config file (`.config`)
  (2) Create a new config file with the desired changes
  (3) Extract the differences between the two configuration files
  (4) Create a fragment file with the changes from the new configuration

Creating a fragment file: example with U-Boot (1/2)

```
cp output/build/uboot-xx/.config output/build/uboot-xx/.config.ori   # (1)
make uboot-menuconfig                # make the desired changes, then save (2)
diff output/build/uboot-xx/.config.ori output/build/uboot-xx/.config # (3)
    322,323c322,323
    < # CONFIG_FIT_ENABLE_SHA384_SUPPORT is not set
    < # CONFIG_FIT_ENABLE_SHA512_SUPPORT is not set
    ---
    > CONFIG_FIT_ENABLE_SHA384_SUPPORT=y
    > CONFIG_FIT_ENABLE_SHA512_SUPPORT=y
```

- Specify in Buildroot's U-Boot config, which fragment file to use, e.g.: `board/friendlyarm/nanopi-neo-plus2/uboot-extras.config`

Creating a fragment file: example with U-Boot (2/2)

- Add all lines starting with `>` (without including the `>`), to the fragment file `board/friendlyarm/nanopi-neo-plus2/uboot-extras.config` (4):

```
CONFIG_FIT_ENABLE_SHA384_SUPPORT=y
CONFIG_FIT_ENABLE_SHA512_SUPPORT=y
```

- To indicate an option that must not be selected, comment it out and append `" is not set"` after it (see U-Boot or Linux `.config` files as examples), e.g.:

```
# CONFIG_WLAN_VENDOR_BROADCOM is not set
```

Patching packages

- Packages might need to be patched to:
  - Fix cross-compilation issues
  - Backport bug or security fixes from upstream
  - Integrate new features or fixes not available upstream, or too specific to the product being made
- Patches are automatically applied: after extraction, but before configuration
- BR comes with patches for various packages, but we may need to add more for our own packages, or to existing packages
- Possible to define global patch directories: `make menuconfig` → `build options` → `global patch directories`

Patch conventions

- Patch files must start with a **sequence number** indicating the order in which they must be applied, e.g.:

```
$ ls packages/ffmpeg/*.patch
ffmpeg/0001-swscale-x86-yuv2rgb-Fix-build-without-SSSE3.patch
ffmpeg/0002-avcodec-vaapi_h264-skip-decode-if-pic-has-no-slices.patch
ffmpeg/0003-libavutil-Fix-mips-build.patch
ffmpeg/0004-configure-add-extralibs-to-extralibs_xxx.patch
```

- Each patch should contain a description of what it does
- Each patch should contain a Signed-off-by identifying the author
- Patches should be generated using `git format-patch` when possible

Creating patches with git (1/2)

- Create a git repo and commit something:

```
cd your_app
git init --initial-branch=main
git add .
git commit -m "my initial commit"
```

- Make changes (add/remove/edit files)
- To create a patch with the differences from the last commit:

```
git commit -a -m "bugfix and updated main"
git format-patch -1
```

This creates `0001-bugfix-and-updated-main.patch`

Creating patches with git (2/2)

- To create `n` (where `n` $>$ 0) patches for the last `n` commits, use:

```
git format-patch -n
```

- You can also create patches between two branches
- To create patches for all commits on the current branch that are not on the `main` branch:

```
git format-patch main -o my_dir
```

  The patches will be saved in the `my_dir` directory

How to patch a package?

- Configure BR to use a global patch directory (which must exist) with:
  - `make menuconfig` → `Build options` → `global patch directories`

1. Inside the patch directory, create a directory with the name of the package to patch, e.g. `busybox` or `libopenssl`
2. Put your patch in the directory above
3. Remove the package directory in `output/build`, for instance `output/build/busybox` for package `busybox`
4. Build again by running `make` or rebuild the package explicitly with `make xxx-rebuild`, for instance `make busybox-rebuild`

Creating our own Buildroot config: example

- BR configuration files for various platforms are located in the `configs` directory
- To create our own `yoyo` config based on `friendlyarm_nanopi_neo_plus2_defconfig`:

```
cp configs/friendlyarm_nanopi_neo_plus2_defconfig configs/yoyo_defconfig
```

- Configure BR for our specific platform, which creates a `.config` file:

```
make yoyo_defconfig
```

- Configure the platform and indicate where to save the `defconfig` file:

```
make menuconfig
```

Go to `Build options`:

```
    Commands  --->
(/workspace/buildroot/configs/yoyo_defconfig) Location to save buildroot config
($(TOPDIR)/dl) Download dir
```

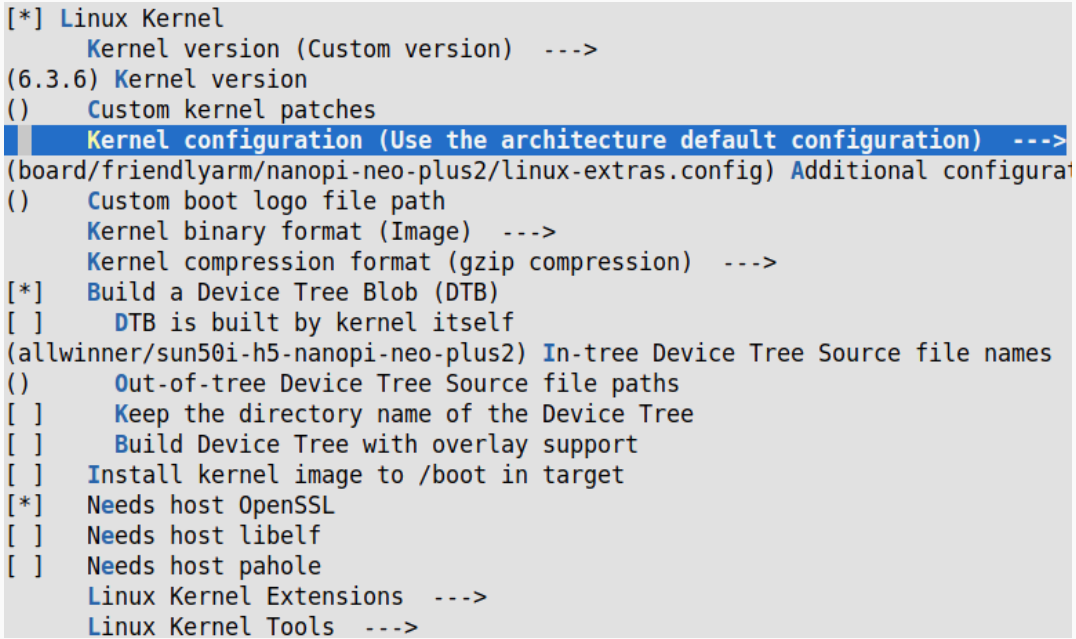
- Save the BR config file to `configs/yoyo_defconfig`:

```
make savedefconfig
```

Linux kernel configuration

- The Linux kernel can be configured to use:
  - a **default** or **own** config file
  - a **fragment** config file
  - a default or own Device Tree file
- Possible to use custom kernel patches
- To configure BR settings about the kernel (version, build a DTB, etc.): `make menuconfig` → `kernel`
- To configure the kernel itself:

```
make linux-menuconfig
```



Creating our own Linux kernel defconfig: example

- Configure Linux kernel for the specific platform:

```
make linux-menuconfig
```

  - Configuration saved to `output/build/linux-xx/.config`)

- Save the `.config` file into a `defconfig` file:

```
make linux-savedefconfig
```

  - Configuration saved to `output/build/linux-xx/defconfig`

- Copy it over, so we can point to it in the kernel's menu config:

```
cp output/build/linux-xx/defconfig  board/friendlyarm/nanopi-neo-plus2/
yoyo_linux_defconfig
```

  - `make menuconfig` → `Kernel` → `Kernel configuration` → `Using a custom (def)config file`

Linux kernel configuration: device tree

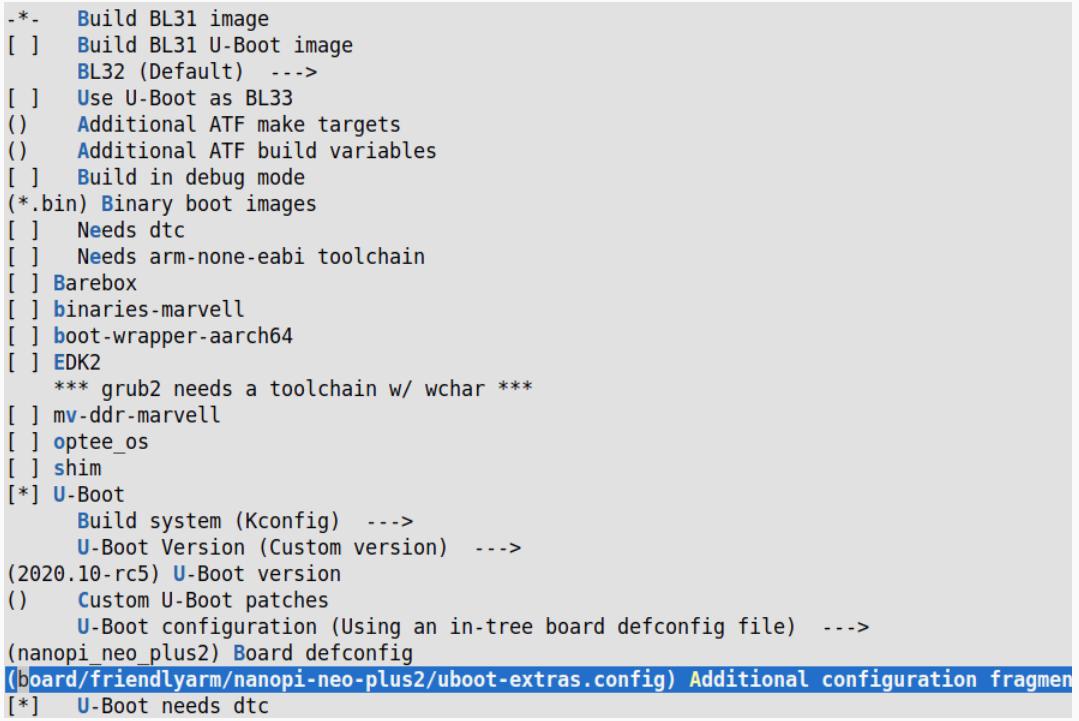
- The Flattened Device Tree (FTD) describes the hardware
- Device Tree sources are found in the kernel's `dts` directory, e.g.:
  `output/build/linux-xx/arch/arm64/boot/dts/allwinner/sun50i-h5-nanopi-neo-plus2.dts`

```
/dts-v1/;
#include "sun50i-h5.dtsi"

#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/input/input.h>
#include <dt-bindings/pinctrl/sun4i-a10.h>

/ {
        model = "FriendlyARM NanoPi NEO Plus2";
        compatible = "friendlyarm,nanopi-neo-plus2", "allwinner,sun50i-h5";

        aliases {
                ethernet0 = &emac;
                serial0 = &uart0;
        };

        chosen {
                stdout-path = "serial0:115200n8";
        };

        leds {
                compatible = "gpio-leds";

                led-0 {
                        label = "nanopi:green:pwr";
```

U-Boot configuration

- U-Boot can be configured to use:
  - the **standard** defconfig file, e.g.: `output/build/uboot-xx/configs/nanopi_neo_plus2_defconfig`
  - a **fragment** config file, e.g.: `board/friedlyarm/nanopi-neo-plus/uboot-extras.config`
- To configure U-Boot:

  `make menuconfig` ⟶ `Bootloaders`



Creating our own U-Boot defconfig: example

- Configure U-Boot for the specific platform:

```
make uboot-menuconfig
```

  - Configuration saved to `output/build/uboot-xx/.config`)

- Save the `.config` file to a `defconfig` file:

```
make uboot-savedefconfig
```

  - Configuration saved to `output/build/uboot-xx/defconfig`

- Copy it over, so we can point to it in U-Boot's menu config:

```
cp output/build/uboot-xx/defconfig board/friendlyarm/nanopi-neo-plus2/
yoyo_uboot_defconfig
```

  - `make menuconfig` ⟶ `Bootloaders` ⟶ `U-Boot configuration` ⟶ `Using a custom board (def)config file`

Busybox configuration

- Busybox packages are **not displayed** by BR
- To configure Busybox packages:

```
make busybox-menuconfig
```

- BR saves Busybox configuration to `output/build/busybox-xx/.config`

Adding a new package

- Packages (programs & libraries) are located in `package`
- How to add a **generic** `foo` package?
- Create the `package/foo` directory with the following files:
  - `Config.in`: describe, in the Kconfig language, the package's config options
  - `foo.mk`: Makefile describing where to fetch the source, how to build it and install it
  - `foo.hash` (optional): provide hashes to check the integrity of the downloaded archives and license files
  - `Sxx_foo`: the package's start scripts, typically if `foo` is a service

Post build script

- BR can execute scripts at various points during the build process:
  - Before starting the build process
  - At the end of the build, before creating filesystem images
  - After creating filesystem images
- For instance, to configure BR to execute a script at the end of the build, before creating filesystem images:
  - `make menuconfig` → `System configuration` → `Custom scripts to run before creating filesystem images`

Build output

When the build is completed, the `output` directory contains:

`build`	sources and compiled packages, kernel, bootloader, etc.
`images`	final images (rootfs, kernel, bootloaders)
`host`	tools built for the host, e.g. the toolchain
`staging`	legacy for backwards compatibility
`target`	almost the complete target rootfs, but with incorrect permissions

Per package actions

Let's consider the `abcd` package[1]. Use:

- `make abcd` builds and installs the package and all its dependencies
- `make abcd-source` downloads the package's sources to the `dl` directory
  - If the package is **already** in `dl`, it won't be downloaded again!
- `make abcd-extract` extracts the package's sources to the `build` directory
- `make abcd-patch` applies patches to the package
- `make abcd-reconfigure` restarts the build from the configure step
- `make abcd-rebuild` restarts the build from the build step
- `make abcd-reinstall` restarts the build from the install step
- `make abcd-dirclean` removes the package build directory
- See `<pkg>-xxx` from `make help` for more targets

[1] Also works with linux, uboot and busybox

Debugging build error

- Sometimes you may encounter errors during the build process
- Not always easy to determine their origin
- BR can be ran with the verbose variable `V=1` which tells BR to output all commands being executed
- Examples:

```
make V=1
```

```
make uboot-rebuild V=1
```

Cleaning

- To delete all generated files:

```
make clean
```

- **Deletes everything** in `output` directory
  - **Consquence**: all `.config` files saved in `output/build/...` (kernel, U-Boot, Busybox, etc.) will be deleted
- However, doesn't delete downloaded packages in `dl` directory

Build & configuration changes

- BR **does not** handle build dependencies when config changes
- Tracking the consequences of config changes is complicated, e.g.:
  - a toolchain setting is changed → everything must be rebuilt
  - a library is removed → all reverse dependencies must be rebuilt
- By design, **BR is simple**:
  - Executes the build procedure of the packages; doesn't track files installed by each package
  - Some features are not implemented to keep it simple (e.g. packages manager)
  - Relies on user's knowledge to know what to do
    - building is fast, so full rebuild is a non-issue!

Make sure to perform a **full rebuild** if "parent" dependencies might have changed!

Resources

- Buildroot website: https://buildroot.org
- Buildroot manual: https://buildroot.org/downloads/manual/manual.html
- Bootlin slides: https://bootlin.com/doc/training/buildroot/buildroot-slides.pdf
- https://wiki.friendlyelec.com/wiki/index.php/NanoPi_NEO_Plus2