Managing the build and the configuration

embedded Linux and kernel engineering

Default build organization

- All the build output goes into a directory called `output/` within the top-level Buildroot source directory.
  - `O = output`
- The configuration file is stored as `.config` in the top-level Buildroot source directory.
  - `CONFIG_DIR = $(TOPDIR)`
  - `TOPDIR = $(shell pwd)`
- `buildroot/`
  - `.config`
  - `arch/`
  - `package/`
  - `output/`
  - `fs/`
  - ...

Out of tree build: introduction

- Out of tree build allows to use an output directory different than `output/`
- Useful to build different Buildroot configurations from the same source tree.
- Customization of the output directory done by passing `O=/path/to/directory` on the command line.
- Configuration file stored inside the `$(O)` directory, as opposed to inside the Buildroot sources for the in-tree build case.
- `project/`
  - `buildroot/`, Buildroot sources
  - `foo-output/`, output of a first project
    - `.config`
  - `bar-output/`, output of a second project
    - `.config`

Out of tree build: using

- To start an out of tree build, two solutions:
  - From the Buildroot source tree, simplify specify a `O=` variable:

```
make O=../foo-output/ menuconfig
```

  - From an empty output directory, specify `O=` and the path to the Buildroot source tree:

```
make -C ../buildroot/ O=$(pwd) menuconfig
```

- Once one out of tree operation has been done (`menuconfig`, loading a defconfig, etc.), Buildroot creates a small wrapper `Makefile` in the output directory.
- This wrapper `Makefile` then avoids the need to pass `O=` and the path to the Buildroot source tree.

Out of tree build: example

1. You are in your Buildroot source tree:

```
$ ls
arch board boot ... Makefile ... package ...
```

2. Create a new output directory, and move to it:

```
$ mkdir ../foobar-output
$ cd ../foobar-output
```

3. Start a new Buildroot configuration:

```
$ make -C ../buildroot O=$(pwd) menuconfig
```

4. Start the build (passing `O=` and `-C` no longer needed thanks to the wrapper):

```
$ make
```

5. Adjust the configuration again, restart the build, clean the build:

```
$ make menuconfig
$ make
$ make clean
```

Full config file vs. *defconfig*

- The `.config` file is a *full* config file: it contains the value for all options (except those having unmet dependencies)
- The default `.config`, without any customization, has 4467 lines (as of Buildroot 2021.02)
  - Not very practical for reading and modifying by humans.
- A *defconfig* stores only the values for options for which the non-default value is chosen.
  - Much easier to read
  - Can be modified by humans
  - Can be used for automated construction of configurations

defconfig: example

- For the default Buildroot configuration, the *defconfig* is empty: everything is the default.
- If you change the architecture to be ARM, the *defconfig* is just one line:

```
BR2_arm=y
```

- If then you also enable the `stress` package, the *defconfig* will be just two lines:

```
BR2_arm=y
BR2_PACKAGE_STRESS=y
```

Using and creating a *defconfig*

- To use a *defconfig*, copying it to `.config` is not sufficient as all the missing (default) options need to be expanded.
- Buildroot allows to load *defconfig* stored in the `configs/` directory, by doing:
  `make <foo>_defconfig`
  - It overwrites the current `.config`, if any
- To create a *defconfig*, run:
  `make savedefconfig`
  - Saved in the file pointed by the `BR2_DEFCONFIG` configuration option
  - By default, points to `defconfig` in the current directory if the configuration was started from scratch, or points to the original *defconfig* if the configuration was loaded from a defconfig.
  - Move it to `configs/` to make it easily loadable with `make <foo>_defconfig`.

Existing *defconfigs*

- Buildroot comes with a number of existing *defconfigs* for various publicly available hardware platforms:
  - RaspberryPi, BeagleBone Black, CubieBoard, Microchip evaluation boards, Minnowboard, various i.MX6 boards
  - QEMU emulated platforms
- List them using `make list-defconfigs`
- Most built-in *defconfigs* are minimal: only build a toolchain, bootloader, kernel and minimal root filesystem.

```
$ make qemu_arm_vexpress_defconfig
$ make
```

- Additional instructions often available in `board/<boardname>`, e.g.: `board/qemu/arm-vexpess/readme.txt`.
- Your own *defconfigs* can obviously be more featureful

Assembling a *defconfig* (1/2)

- *defconfigs* are trivial text files, one can use simple concatenation to assemble them from fragments.

platform1.frag

```
BR2_arm=y
BR2_TOOLCHAIN_BUILDROOT_WCHAR=y
BR2_GCC_VERSION_7_X=y
```

platform2.frag

```
BR2_mipsel=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_CODESOURCERY_MIPS=y
```

packages.frag

```
BR2_PACKAGE_STRESS=y
BR2_PACKAGE_MTD=y
BR2_PACKAGE_LIBCONFIG=y
```

Assembling a *defconfig* (2/2)

debug.frag

```
BR2_ENABLE_DEBUG=y
BR2_PACKAGE_STRACE=y
```

Build a release system for *platform1*

```
$ ./support/kconfig/merge_config.sh platform1.frag packages.frag
$ make
```

Build a debug system for *platform2*

```
$ ./support/kconfig/merge_config.sh platform2.frag packages.frag \
        debug.frag
$ make
```

- Saving fragments is not possible; it must be done manually from an existing *defconfig*

Other building tips

- Cleaning targets
  - Cleaning all the build output, but keeping the configuration file:

```
$ make clean
```

  - Cleaning everything, including the configuration file, and downloaded file if at the default location:

```
$ make distclean
```

- Verbose build
  - By default, Buildroot hides a number of commands it runs during the build, only showing the most important ones.
  - To get a fully verbose build, pass `V=1`:

```
$ make V=1
```

  - Passing `V=1` also applies to packages, like the Linux kernel, busybox...

Buildroot source and build trees

Source tree

Source tree (1/5)

- `Makefile`
  - top-level `Makefile`, handles the configuration and general orchestration of the build
- `Config.in`
  - top-level `Config.in`, main/general options. Includes many other `Config.in` files
- `arch/`
  - `Config.in.*` files defining the architecture variants (processor type, ABI, floating point, etc.)
  - `Config.in`, `Config.in.arm`, `Config.in.x86`, `Config.in.microblaze`, etc.

Source tree (2/5)

- `toolchain/`
  - packages for generating or using toolchains
  - `toolchain/` virtual package that depends on either `toolchain-buildroot` or `toolchain-external`
  - `toolchain-buildroot/` virtual package to build the internal toolchain
  - `toolchain-external/` virtual package to download/import the external toolchain
- `system/`
  - `skeleton/` the rootfs skeleton
  - `Config.in`, options for system-wide features like init system, `/dev` handling, etc.
- `linux/`
  - `linux.mk`, the Linux kernel package

Source tree (3/5)

- package/
  - all the user space packages (2800+)
  - busybox/, gcc/, qt5/, etc.
  - pkg-generic.mk, core package infrastructure
  - pkg-cmake.mk, pkg-autotools.mk, pkg-perl.mk, etc. Specialized package infrastructures
- fs/
  - logic to generate filesystem images in various formats
  - common.mk, common logic
  - cpio/, ext2/, squashfs/, tar/, ubifs/, etc.
- boot/
  - bootloader packages
  - at91bootstrap3/, barebox/, grub2/, syslinux/, uboot/, etc.

Source tree (4/5)

- `configs/`
  - default configuration files for various platforms
  - similar to kernel defconfigs
  - `atmel_xplained_defconfig`, `beaglebone_defconfig`, `raspberrypi_defconfig`, etc.
- `board/`
  - board-specific files (kernel configuration files, kernel patches, image flashing scripts, etc.)
  - typically go together with a *defconfig* in `configs/`
- `support/`
  - misc utilities (kconfig code, libtool patches, download helpers, and more.)

Source tree (5/5)

- `utils/`
  - Various utilities useful to Buildroot developers
  - `brmake`, make wrapper, with logging
  - `get-developers`, to know to whom patches should be sent
  - `test-pkg`, to validate that a package builds properly
  - `scanpipy`, `scancpan` to generate Python/Perl package `.mk` files
  - ...
- `docs/`
  - Buildroot documentation
  - Written in AsciiDoc, can generate HTML, PDF, TXT versions: `make manual`
  - ≈135 pages PDF document
  - Also available pre-generated online.
  - https://buildroot.org/downloads/manual/manual.html

Build tree

Build tree: $(O)

- `output/`
- Global output directory
- Can be customized for out-of-tree build by passing `O=<dir>`
- Variable: `O` (as passed on the command line)
- Variable: `BASE_DIR` (as an absolute path)

Build tree: $(O)/build

- output/
  - build/
    - buildroot-config/
    - busybox-1.22.1/
    - host-pkgconf-0.8.9/
    - kmod-1.18/
    - build-time.log
  - Where all source tarballs are extracted
  - Where the build of each package takes place
  - In addition to the package sources and object files, *stamp* files are created by Buildroot
  - Variable: `BUILD_DIR`

Build tree: `$(O)/host`

- `output/`
  - `host/`
    - `lib`
    - `bin`
    - `sbin`
    - `<tuple>/sysroot/bin`
    - `<tuple>/sysroot/lib`
    - `<tuple>/sysroot/usr/lib`
    - `<tuple>/sysroot/usr/bin`
  - Contains both the tools built for the host (cross-compiler, etc.) and the *sysroot* of the toolchain
  - Variable: `HOST_DIR`
  - Host tools are directly in `host/`
  - The *sysroot* is in `host/<tuple>/sysroot/usr`
  - `<tuple>` is an identifier of the architecture, vendor, operating system, C library and ABI. E.g: `arm-unknown-linux-gnueabihf`.
  - Variable for the *sysroot*: `STAGING_DIR`

Build tree: $(O)/staging

- output/
  - staging/
  - Just a symbolic link to the *sysroot*, i.e. to host/<tuple>/sysroot/.
  - Available for convenience

Build tree: $(O)/target

- `output/`
  - `target/`
    - `bin/`
    - `etc/`
    - `lib/`
    - `usr/bin/`
    - `usr/lib/`
    - `usr/share/`
    - `usr/sbin/`
    - `THIS_IS_NOT_YOUR_ROOT_FILESYSTEM`
    - ...
  - The target root filesystem
  - Usual Linux hierarchy
  - Not completely ready for the target: permissions, device files, etc.
  - Buildroot does not run as root: all files are owned by the user running Buildroot, not *setuid*, etc.
  - Used to generate the final root filesystem images in `images/`
  - Variable: `TARGET_DIR`

Build tree: $(O)/images

- output/
  - images/
    - zImage
    - armada-370-mirabox.dtb
    - rootfs.tar
    - rootfs.ubi
  - Contains the final images: kernel image, bootloader image, root filesystem image(s)
  - Variable: BINARIES_DIR

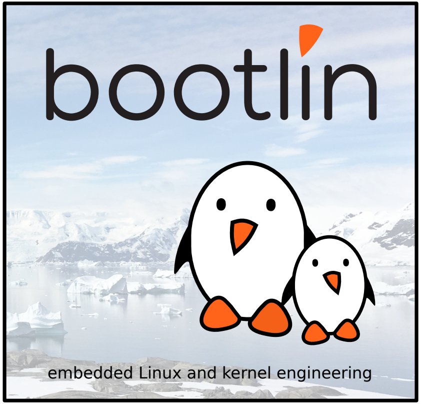
Build tree: $(O)/graphs

- `output/`
  - `graphs/`
  - Visualization of Buildroot operation: dependencies between packages, time to build the different packages
  - `make graph-depends`
  - `make graph-build`
  - `make graph-size`
  - Variable: `GRAPHS_DIR`
  - See the section *Analyzing the build* later in this training.

Build tree: $(O)/legal-info

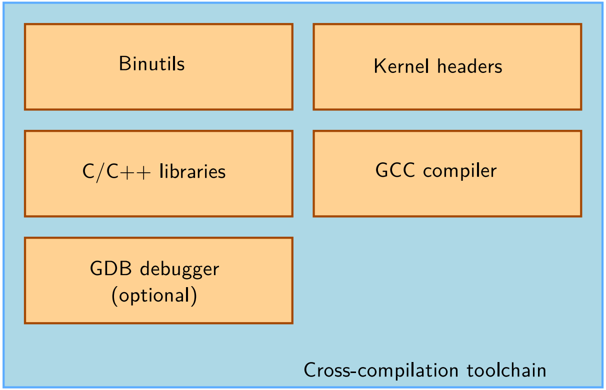
- `output/`
  - `legal-info/`
    - `manifest.csv`
    - `host-manifest.csv`
    - `licenses.txt`
    - `licenses/`
    - `sources/`
    - ...
  - Legal information: license of all packages, and their source code, plus a licensing manifest
  - Useful for license compliance
  - `make legal-info`
  - Variable: `LEGAL_INFO_DIR`

Toolchains in Buildroot



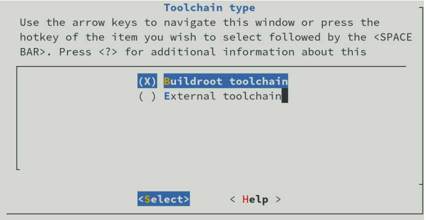
What is a cross-compilation toolchain?

- A set of tools to build and debug code for a target architecture, from a machine running a different architecture.
- Example: building code for ARM from a x86-64 PC.



Two possibilities for the toolchain

- Buildroot offers two choices for the toolchain, called **toolchain backends**:
  - The **internal toolchain** backend, where Buildroot builds the toolchain entirely from source
  - The **external toolchain** backend, where Buildroot uses a existing pre-built toolchain
- Selected from `Toolchain` → `Toolchain type`.



Internal toolchain backend

- Makes Buildroot build the entire cross-compilation toolchain from source.
- Provides a lot of flexibility in the configuration of the toolchain.
  - Kernel headers version
  - C library: Buildroot supports uClibc, (e)glibc and musl
    - glibc, the standard C library. Good choice if you don't have tight space constraints ($>=$ 10 MB)
    - uClibc-ng and musl, smaller C libraries. uClibc-ng supports non-MMU architectures. Good for very small systems ($<$ 10 MB).
  - Different versions of binutils and gcc. Keep the default versions unless you have specific needs.
  - Numerous toolchain options: C++, LTO, OpenMP, libmudflap, graphite, and more depending on the selected C library.
- Building a toolchain takes quite some time: 15-20 minutes on moderately recent machines.

Internal toolchain backend: result

- `host/bin/<tuple>-<tool>`, the cross-compilation tools: compiler, linker, assembler, and more. The compiler is hidden behind a wrapper program.
- `host/<tuple>/`
  - `sysroot/usr/include/`, the kernel headers and C library headers
  - `sysroot/lib/` and `sysroot/usr/lib/`, C library and gcc runtime
  - `include/c++/`, C++ library headers
  - `lib/`, host libraries needed by gcc/binutils
- `target/`
  - `lib/` and `usr/lib/`, C and C++ libraries
- The compiler is configured to:
  - generate code for the architecture, variant, FPU and ABI selected in the `Target options`
  - look for libraries and headers in the *sysroot*
  - no need to pass weird gcc flags!

External toolchain backend possibilities

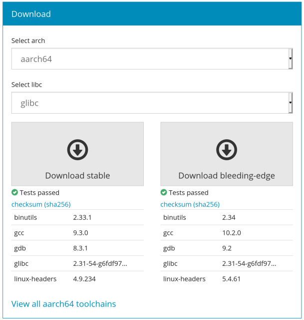
- Allows to re-use existing pre-built toolchains
- Great to:
  - save the build time of the toolchain
  - use vendor provided toolchain that are supposed to be reliable
- Several options:
  - Use an existing toolchain profile known by Buildroot
  - Download and install a custom external toolchain
  - Directly use a pre-installed custom external toolchain

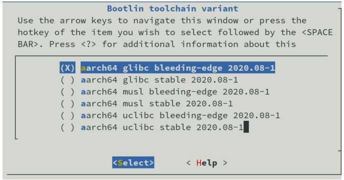
Existing external toolchain profile

- Buildroot already knows about a wide selection of publicly available toolchains.
- Toolchains from
  - ARM (ARM and AArch64)
  - Mentor Graphics (AArch64, ARM, MIPS, NIOS-II)
  - Imagination Technologies (MIPS)
  - Synopsys (ARC)
  - Bootlin
- In such cases, Buildroot is able to download and automatically use the toolchain.
- It already knows the toolchain configuration: C library being used, kernel headers version, etc.
- Additional profiles can easily be added.

Existing external toolchains: Bootlin toolchains

- https://toolchains.bootlin.com
- A set of 169 pre-built toolchains, freely available
  - 41 different CPU architecture variants
  - All possible C libraries supported: glibc, uClibc-ng, musl
  - Toolchains built with Buildroot!
- Two versions for each toolchain
  - *stable*, which uses the default version of gcc, binutils and gdb in Buildroot
  - *bleeding-edge*, which uses the latest version of gcc, binutils and gdb in Buildroot
- Directly integrated in Buildroot





Custom external toolchains

- If you have a custom external toolchain, for example from your vendor, select `Custom toolchain` in `Toolchain`.
- Buildroot can download and extract it for you
  - Convenient to share toolchains between several developers
  - Option `Toolchain to be downloaded and installed` in `Toolchain origin`
  - The URL of the toolchain tarball is needed
- Or Buildroot can use an already installed toolchain
  - Option `Pre-installed toolchain` in `Toolchain origin`
  - The local path to the toolchain is needed
- In both cases, you will have to tell Buildroot the configuration of the toolchain: C library, kernel headers version, etc.
  - Buildroot needs this information to know which packages can be built with this toolchain
  - Buildroot will check those values at the beginning of the build

Custom external toolchain example configuration

```
                                   Toolchain
Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> selects
a feature, while <N> excludes a feature.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] feature is selected  [ ]
feature is excluded

                                   Toolchain type (External toolchain)  --->
                                   *** Toolchain External Options ***
                                   Toolchain (Custom toolchain)  --->
                                   Toolchain origin (Toolchain to be downloaded and installed)  --->
                               (http://autobuild.buildroot.org/toolchains/tarballs/br-i386-pentium4-full-2020.11.2.tar.bz2) Toolchain URL
                               (bin) Toolchain relative binary path (NEW)
                               ($(ARCH)-linux) Toolchain prefix (NEW)
                                   External toolchain gcc version (9.x)  --->
                                   External toolchain kernel headers series (4.4.x)  --->
                                   External toolchain C library (uClibc/uClibc-ng)  --->
                               -*- Toolchain has WCHAR support?
                               [*] Toolchain has locale support?
                               [*] Toolchain has threads support? (NEW)
                               [ ]   Toolchain has threads debugging support?
                               [*]   Toolchain has NPTL threads support? (NEW)
                               [ ] Toolchain has SSP support? (NEW)
                               [ ] Toolchain has RPC support? (NEW)
                               [*] Toolchain has C++ support?
                               [ ] Toolchain has D support? (NEW)
                               [ ] Toolchain has Fortran support? (NEW)
                               [ ] Toolchain has OpenMP support? (NEW)
                               [ ] Copy gdb server to the Target (NEW)
                                   *** Host GDB Options ***
                               [ ] Build cross gdb for the host (NEW)
                                   *** Toolchain Generic Options ***
                               ()  Extra toolchain libraries to be copied to target (NEW)
                               ()  Target Optimizations (NEW)
                               ()  Target linker options (NEW)
                               [ ] Register toolchain within Eclipse Buildroot plug-in (NEW)

                          <Select>    < Exit >    < Help >    < Save >    < Load >
```

External toolchain: result

- `host/opt/ext-toolchain`, where the original toolchain tarball is extracted. Except when a local pre-installed toolchain is used.
- `host/bin/<tuple>-<tool>`, symbolic links to the cross-compilation tools in their original location. Except the compiler, which points to a wrapper program.
- `host/<tuple>/`
  - `sysroot/usr/include/`, the kernel headers and C library headers
  - `sysroot/lib/` and `sysroot/usr/lib/`, C library and gcc runtime
  - `include/c++/`, C++ library headers
- `target/`
  - `lib/` and `usr/lib/`, C and C++ libraries
- The wrapper takes care of passing the appropriate flags to the compiler.
  - Mimics the internal toolchain behavior

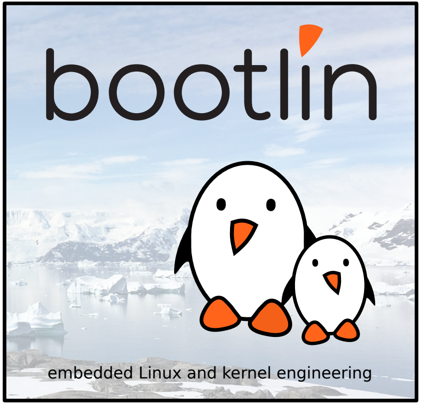
Kernel headers version

- One option in the toolchain menu is particularly important: the kernel headers version.
- When building user space programs, libraries or the C library, kernel headers are used to know how to interface with the kernel.
- This kernel/user space interface is **backward compatible**, but can introduce new features.
- It is therefore important to use kernel headers that have a version **equal or older** than the kernel version running on the target.
- With the internal toolchain backend, choose an appropriate kernel headers version.
- With the external toolchain backend, beware when choosing your toolchain.

Other toolchain menu options

- The toolchain menu offers a few other options:
  - *Target optimizations*
    - Allows to pass additional compiler flags when building target packages
    - Do not pass flags to select a CPU or FPU, these are already passed by Buildroot
    - Be careful with the flags you pass, they affect the entire build
  - *Target linker options*
    - Allows to pass additional linker flags when building target packages
  - gdb/debugging related options
    - Covered in our *Application development* section later.

Managing the Linux kernel configuration



Introduction

- The Linux kernel itself uses *kconfig* to define its configuration
- Buildroot cannot replicate all Linux kernel configuration options in its `menuconfig`
- Defining the Linux kernel configuration therefore needs to be done in a special way.
- Note: while described with the example of the Linux kernel, this discussion is also valid for other packages using *kconfig*: `barebox`, `uclibc`, `busybox` and `uboot`.

Defining the configuration

- In the `Kernel` menu in `menuconfig`, 3 possibilities to configure the kernel:
  1. `Use a defconfig`
     - Will use a *defconfig* provided within the kernel sources
     - Available in `arch/<ARCH>/configs` in the kernel sources
     - Used unmodified by Buildroot
     - Good starting point
  2. `Use a custom config file`
     - Allows to give the path to either a full `.config`, or a minimal *defconfig*
     - Usually what you will use, so that you can have a custom configuration
  3. `Use the architecture default configuration`
     - Use the *defconfig* provided by the architecture in the kernel source tree. Some architectures (e.g ARM64) have a single *defconfig*.
- Configuration can be further tweaked with `Additional fragments`
  - Allows to pass a list of configuration file fragments.
  - They can complement or override configuration options specified in a *defconfig* or a full configuration file.

Examples of kernel configuration

`stm32mp157a_dk1_defconfig`: custom configuration file

```
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="board/stmicroelectronics/stm32mp157a-dk1/linux.config"
```

`ts4900_defconfig`: standard kernel defconfig

```
BR2_LINUX_KERNEL_DEFCONFIG="imx_v6_v7"
```

`warpboard_defconfig`: standard kernel defconfig + fragment

```
BR2_LINUX_KERNEL_DEFCONFIG="imx_v6_v7"
BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES="board/freescale/warpboard/linux.fragment"
```

`linux.fragment`: contains extra kernel options

```
CONFIG_CFG80211_WEXT=y
```

Changing the configuration

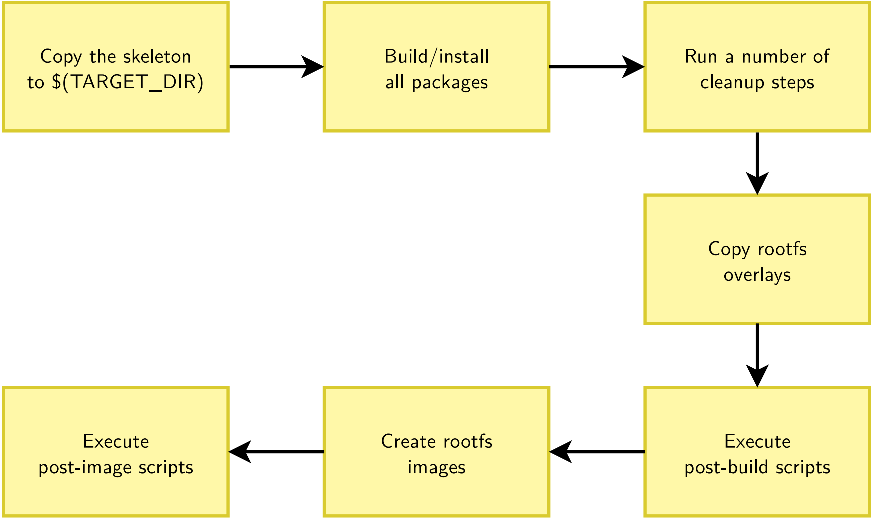
- Running one of the Linux kernel configuration interfaces:
  - `make linux-menuconfig`
  - `make linux-nconfig`
  - `make linux-xconfig`
  - `make linux-gconfig`
- Will load either the defined kernel *defconfig* or custom configuration file, and start the corresponding Linux kernel configuration interface.
- Changes made are only made in `$(O)/build/linux-<version>/`, i.e. they are not preserved across a clean rebuild.
- To save them:
  - `make linux-update-config`, to save a full config file
  - `make linux-update-defconfig`, to save a minimal defconfig
  - Only works if a *custom configuration file* is used

Typical flow

1. `make menuconfig`
   - Start with a *defconfig* from the kernel, say `mvebu_v7_defconfig`
2. Run `make linux-menuconfig` to customize the configuration
3. Do the build, test, tweak the configuration as needed.
4. You cannot do `make linux-update-{config,defconfig}`, since the Buildroot configuration points to a kernel *defconfig*
5. `make menuconfig`
   - Change to a custom configuration file. There's no need for the file to exist, it will be created by Buildroot.
6. `make linux-update-defconfig`
   - Will create your custom configuration file, as a minimal *defconfig*

Root filesystem in Buildroot

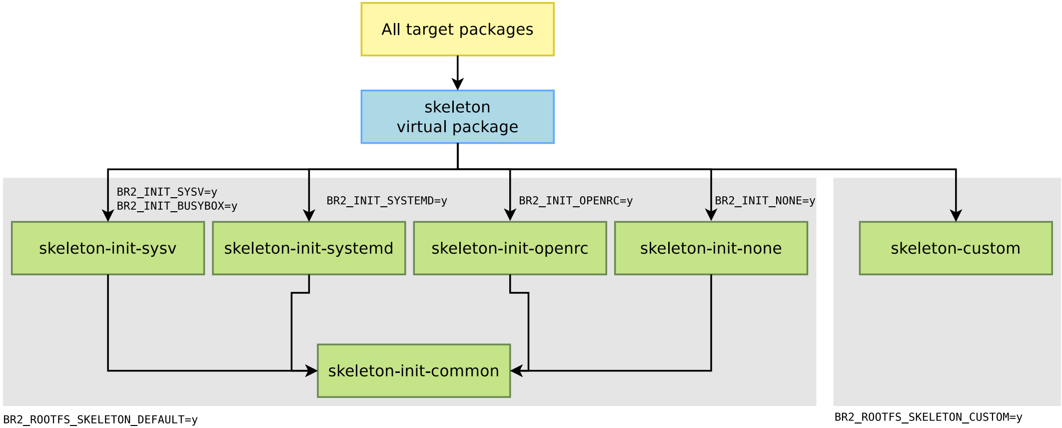
Overall rootfs construction steps



Root filesystem skeleton

- The base of a Linux root filesystem: UNIX directory hierarchy, a few configuration files and scripts in `/etc`. No programs or libraries.
- All target packages depend on the `skeleton` package, so it is essentially the first thing copied to `$(TARGET_DIR)` at the beginning of the build.
- `skeleton` is a virtual package that will depend on:
  - `skeleton-init-{sysv,systemd,openrc,none}` depending on the init system being selected
  - `skeleton-custom` when a custom skeleton is selected
- All of `skeleton-init-{sysv,systemd,openrc,none}` depend on `skeleton-init-common`
  - Copies `system/skeleton/*` to `$(TARGET_DIR)`
- `skeleton-init-{sysv,systemd,openrc}` install additional files specific to those *init systems*

Skeleton packages dependencies



Custom root filesystem skeleton

- A custom *skeleton* can be used, through the `BR2_ROOTFS_SKELETON_CUSTOM` and `BR2_ROOTFS_SKELETON_CUSTOM_PATH` options.
- In this case: `skeleton` depends on `skeleton-custom`
- Completely replaces `skeleton-init-*`, so the custom skeleton must provide everything.
- Not recommended though:
  - the base is usually good for most projects.
  - skeleton only copied at the beginning of the build, so a skeleton change needs a full rebuild
- Use *rootfs overlays* or *post-build scripts* for root filesystem customization (covered later)

Installation of packages

- All the selected target packages will be built (can be BusyBox, Qt, OpenSSH, lighttpd, and many more)
- Most of them will install files in `$(TARGET_DIR)`: programs, libraries, fonts, data files, configuration files, etc.
- This is really the step that will bring the vast majority of the files in the root filesystem.
- Covered in more details in the section about creating your own Buildroot packages.

Cleanup step

- Once all packages have been installed, a cleanup step is executed to reduce the size of the root filesystem.
- It mainly involves:
  - Removing header files, pkg-config files, CMake files, static libraries, man pages, documentation.
  - Stripping all the programs and libraries using `strip`, to remove unneeded information. Depends on `BR2_ENABLE_DEBUG` and `BR2_STRIP_*` options.
  - Additional specific clean up steps: clean up unneeded Python files when Python is used, etc. See `TARGET_FINALIZE_HOOKS` in the Buildroot code.

Root filesystem overlay

- To customize the contents of your root filesystem, to add configuration files, scripts, symbolic links, directories or any other file, one possible solution is to use a **root filesystem overlay**.
- A *root filesystem overlay* is simply a directory whose contents will be **copied over the root filesystem**, after all packages have been installed. Overwriting files is allowed.
- The option `BR2_ROOTFS_OVERLAY` contains a space-separated list of overlay paths.

```
$ grep ^BR2_ROOTFS_OVERLAY .config
BR2_ROOTFS_OVERLAY="board/myproject/rootfs-overlay"
$ find -type f board/myproject/rootfs-overlay
board/myproject/rootfs-overlay/etc/ssh/sshd_config
board/myproject/rootfs-overlay/etc/init.d/S99myapp
```

Post-build scripts

- Sometimes a *root filesystem overlay* is not sufficient: you can use **post-build scripts**.
- Can be used to **customize existing files**, **remove unneeded files** to save space, add **new files that are generated dynamically** (build date, etc.)
- Executed before the root filesystem image is created. Can be written in any language, shell scripts are often used.
- `BR2_ROOTFS_POST_BUILD_SCRIPT` contains a space-separated list of post-build script paths.
- `$(TARGET_DIR)` path passed as first argument, additional arguments can be passed in the `BR2_ROOTFS_POST_SCRIPT_ARGS` option.
- Various environment variables are available:
  - `BR2_CONFIG`, path to the Buildroot .config file
  - `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`, `BUILD_DIR`, `BINARIES_DIR`, `BASE_DIR`

Post-build script: example

board/myproject/post-build.sh

```
#!/bin/sh

# Generate a file identifying the build (git commit and build date)
echo $(git describe) $(date +%Y-%m-%d-%H:%M:%S) > \
    $TARGET_DIR/etc/build-id

# Create /applog mountpoint, and adjust /etc/fstab
mkdir -p $TARGET_DIR/applog
grep -q "^/dev/mtdblock7" $TARGET_DIR/etc/fstab || \
    echo "/dev/mtdblock7\t\t/applog\tjffs2\tdefaults\t\t0\t0" >> \
    $TARGET_DIR/etc/fstab

# Remove unneeded files
rm -rf $TARGET_DIR/usr/share/icons/bar
```

Buildroot configuration

```
BR2_ROOTFS_POST_BUILD_SCRIPT="board/myproject/post-build.sh"
```

Generating the filesystem images

- In the `Filesystem images` menu, you can select which filesystem image formats to generate.
- To generate those images, Buildroot will generate a shell script that:
  - **Changes the owner** of all files to `0:0` (root user)
  - Takes into account the global **permission and device tables**, as well as the per-package ones.
  - Takes into account the **global and per-package users tables**.
  - Runs the **filesystem image generation utility**, which depends on each filesystem type (`genext2fs`, `mkfs.ubifs`, `tar`, etc.)
- This script is executed using a tool called *fakeroot*
  - Allows to fake being root so that permissions and ownership can be modified, device files can be created, etc.
  - Advanced: possibility of running a custom script inside *fakeroot*, see `BR2_ROOTFS_POST_FAKEROOT_SCRIPT`.

Permission table

- By default, all files are owned by the `root` user, and the permissions with which they are installed in `$(TARGET_DIR)` are preserved.
- To customize the ownership or the permission of installed files, one can create one or several **permission tables**
- `BR2_ROOTFS_DEVICE_TABLE` contains a space-separated list of permission table files. The option name contains *device* for backward compatibility reasons only.
- The `system/device_table.txt` file is used by default.
- Packages can also specify their own permissions. See the *Advanced package aspects* section for details.

Permission table example

```
#<name>     <type>  <mode>  <uid>   <gid>   <major> <minor> <start> <inc>   <count>
/dev        d       755     0       0       -       -       -       -       -
/tmp        d       1777    0       0       -       -       -       -       -
/var/www    d       755     33      33      -       -       -       -       -
```

Device table

- When the system is using a static `/dev`, one may need to create additional *device nodes*
- Done using one or several **device tables**
- `BR2_ROOTFS_STATIC_DEVICE_TABLE` contains a space-separated list of device table files.
- The `system/device_table_dev.txt` file is used by default.
- Packages can also specify their own device files. See the *Advanced package aspects* section for details.

Device table example

```
# <name>        <type> <mode> <uid>  <gid>  <major> <minor> <start> <inc>  <count>
/dev/mem        c      640    0      0      1       1       0       0      -
/dev/kmem       c      640    0      0      1       2       0       0      -
/dev/i2c-       c      666    0      0      89      0       0       1      4
```

Users table

- One may need to add specific UNIX users and groups in addition to the ones available in the default skeleton.
- BR2_ROOTFS_USERS_TABLES is a space-separated list of user tables.
- Packages can also specify their own users. See the *Advanced package aspects* section for details.

Users table example

```
# <username> <uid> <group> <gid> <password> <home>    <shell> <groups>    <comment>
foo          -1    bar     -1    !=blabla   /home/foo /bin/sh alpha,bravo Foo user
test         8000  wheel   -1    =          -         /bin/sh -           Test user
```

Post-image scripts

- Once all the filesystem images have been created, at the very end of the build, **post-image** scripts are called.
- They allow to do any custom action at the end of the build. For example:
  - Extract the root filesystem to do NFS booting
  - Generate a final firmware image
  - Start the flashing process
- `BR2_ROOTFS_POST_IMAGE_SCRIPT` is a space-separated list of *post-image* scripts to call.
- Post-image scripts are called:
  - from the Buildroot source directory
  - with the `$(BINARIES_DIR)` path as first argument
  - with the contents of the `BR2_ROOTFS_POST_SCRIPT_ARGS` as other arguments
  - with a number of available environment variables: `BR2_CONFIG`, `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`, `BUILD_DIR`, `BINARIES_DIR` and `BASE_DIR`.

Init mechanism

- Buildroot supports multiple *init* implementations:
  - **BusyBox init**, the default. Simplest solution.
  - **sysvinit**, the old style featureful *init* implementation
  - **systemd**, the modern init system
  - **OpenRC**, the init system used by Gentoo
- Selecting the *init* implementation in the `System configuration` menu will:
  - Ensure the necessary packages are selected
  - Make sure the appropriate init scripts or configuration files are installed by packages. See *Advanced package aspects* for details.

/dev management method

- Buildroot supports four methods to handle the `/dev` directory:
  - Using **devtmpfs**. `/dev` is managed by the kernel *devtmpfs*, which creates device files automatically. Default option.
  - Using **static /dev**. This is the old way of doing `/dev`, not very practical.
  - Using **mdev**. `mdev` is part of BusyBox and can run custom actions when devices are added/removed. Requires *devtmpfs* kernel support.
  - Using **eudev**. Forked from `systemd`, allows to run custom actions. Requires *devtmpfs* kernel support.
- When *systemd* is used, the only option is *udev* from *systemd* itself.

Other customization options

- There are various other options to customize the root filesystem:
  - **getty** options, to run a login prompt on a serial port or screen
  - **hostname** and **banner** options
  - **DHCP network** on one interface (for more complex setups, use an *overlay*)
  - **root password**
  - **timezone** installation and selection
  - **NLS**, Native Language Support, to support message translation
  - **locale** files installation and filtering (to install translations only for a subset of languages, or none at all)

Deploying the images

- By default, Buildroot simply stores the different images in `$(O)/images`
- It is up to the user to deploy those images to the target device.
- Possible solutions:
  - For removable storage (SD card, USB keys):
    - manually create the partitions and extract the root filesystem as a tarball to the appropriate partition.
    - use a tool like `genimage` to create a complete image of the media, including all partitions
  - For NAND flash:
    - Transfer the image to the target, and flash it.
  - NFS booting
  - initramfs

Deploying the images: genimage

- `genimage` allows to create the complete image of a block device (SD card, USB key, hard drive), including multiple partitions and filesystems.
- For example, allows to create an image with two partitions: one FAT partition for bootloader and kernel, one ext4 partition for the root filesystem.
- Also allows to place the bootloader at a fixed offset in the image if required.
- The helper script `support/scripts/genimage.sh` can be used as a *post-image* script to call *genimage*
- More and more widely used in Buildroot default configurations

Deploying the images: genimage example

genimage-raspberrypi.cfg

```
image boot.vfat {
	vfat {
		files = {
			"bcm2708-rpi-b.dtb",
			"rpi-firmware/bootcode.bin",
			"rpi-firmware/cmdline.txt",
			"kernel-marked/zImage"
			[...]
		}
	}

	size = 32M
}
```

```
image sdcard.img {
	hdimage {
	}

	partition boot {
		partition-type = 0xC
		bootable = "true"
		image = "boot.vfat"
	}

	partition rootfs {
		partition-type = 0x83
		image = "rootfs.ext4"
	}
}
```

defconfig

```
BR2_ROOTFS_POST_IMAGE_SCRIPT="support/scripts/genimage.sh"
BR2_ROOTFS_POST_SCRIPT_ARGS="-c board/raspberrypi/genimage-raspberrypi.cfg"
```

flash

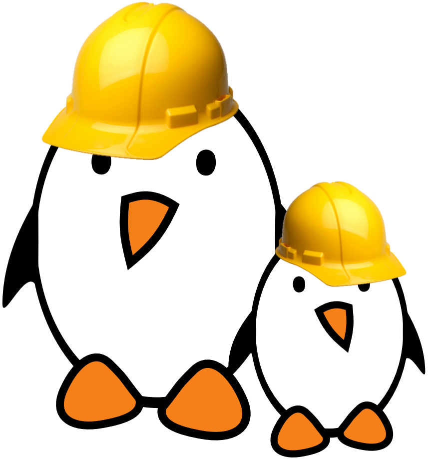
```
dd if=output/images/sdcard.img of=/dev/sdb
```

Deploying the image: NFS booting

- Many people try to use `$(O)/target` directly for NFS booting
  - This cannot work, due to permissions/ownership being incorrect
  - Clearly explained in the `THIS_IS_NOT_YOUR_ROOT_FILESYSTEM` file.
- Generate a tarball of the root filesystem
- Use `sudo tar -C /nfs -xf output/images/rootfs.tar` to prepare your NFS share.

Deploying the image: initramfs

- Another common use case is to use an *initramfs*, i.e. a root filesystem fully in RAM.
  - Convenient for small filesystems, fast booting or kernel development
- Two solutions:
  - `BR2_TARGET_ROOTFS_CPIO=y` to generate a *cpio* archive, that you can load from your bootloader next to the kernel image.
  - `BR2_TARGET_ROOTFS_INITRAMFS=y` to directly include the *initramfs* inside the kernel image. Only available when the kernel is built by Buildroot.



- Explore the build output
- Customize the root filesystem using a rootfs overlay
- Use a post-build script
- Customize the kernel with patches and additional configuration options
- Add more packages
- Use defconfig files and out of tree build