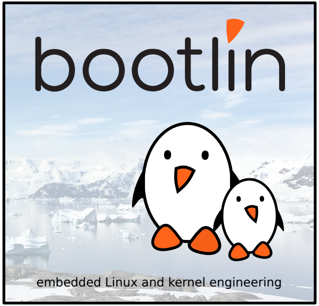
Buildroot system development training

Latest update: February 25, 2025.

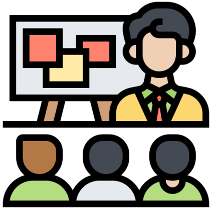
Document updates and training details:
https://bootlin.com/training/buildroot

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com



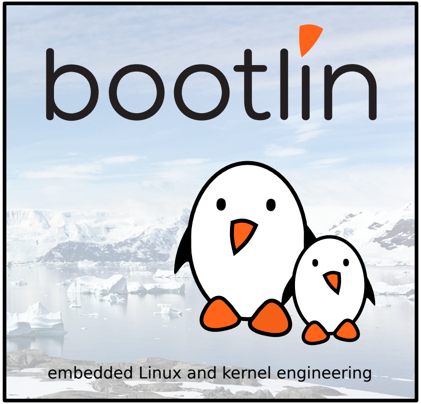
Buildroot system development training

- These slides are the training materials for Bootlin's *Buildroot system development* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/buildroot
- Contact: training@bootlin.com



Icon by Eucalyp, Flaticon

About Bootlin



Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Embedded Linux system development On-site: 4 or 5 days Online: 7 * 4 hours	Linux kernel driver development On-site: 5 days Online: 7 * 4 hours	Yocto Project system development On-site: 3 days Online: 4 * 4 hours	Buildroot system development On-site: 3 days Online: 5 * 4 hours
Understanding the Linux graphics stack On-site: 2 days Online: 4 * 4 hours	Embedded Linux audio On-site: 2 days Online: 4 * 4 hours	Real-Time Linux with PREEMPT_RT On-site: 2 days Online: 3 * 4 hours	Linux debugging, tracing, profiling and performance analysis On-site: 3 days Online: 4 * 4 hours

All our training materials are freely available
under a free documentation license (CC-BY-SA 3.0)
See https://bootlin.com/training/

Bootlin, an open-source contributor

- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 9000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 6000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- Twitter:
  https://twitter.com/bootlincom
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

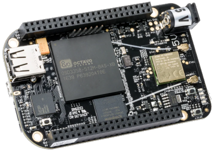
Icon by Freepik, Flaticon

Generic course information

Supported hardware

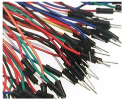
BeagleBone Black or BeagleBone Black Wireless, from BeagleBoard.org

- Texas Instruments AM335x (ARM Cortex-A8 CPU)
- SoC with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- 512 MB of RAM
- 4 GB of on-board eMMC storage
- USB host and USB device, microSD, micro HDMI
- WiFi and Bluetooth (wireless version), otherwise Ethernet
- 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- A huge number of expansion boards, called *capes*. See https://elinux.org/Beagleboard:BeagleBone_Capes.



Shopping list: hardware for this course

- BeagleBone Black or BeagleBone Black Wireless - Multiple distributors: See https://www.beagleboard.org/boards.
- MicroUSB cable
- USB Serial Cable - 3.3 V - Female ends (for serial console) [1]
- Nintendo Nunchuk with UEXT connector [2]
- Breadboard jumper wires - Male ends (to connect the Nunchuk) [3]
- MicroSD card



---

[1] https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-SERIAL-F/

[2] https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/

[3] https://www.olimex.com/Products/Breadboarding/JUMPER-WIRES/JW-110x10/

Training quiz and certificate

- You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.

Participate!

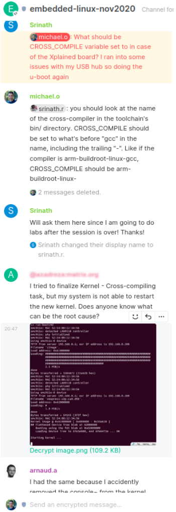
During the lectures...

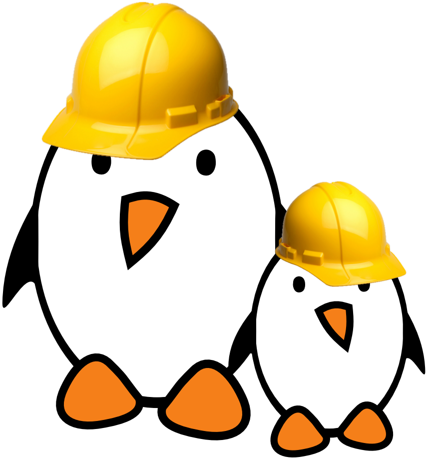
- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please always keep your camera on!
  - Also make sure your name is properly filled.
  - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.





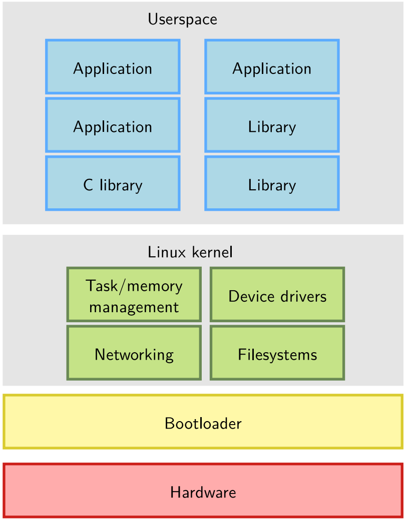
Prepare your lab environment

- Download and extract the lab archive

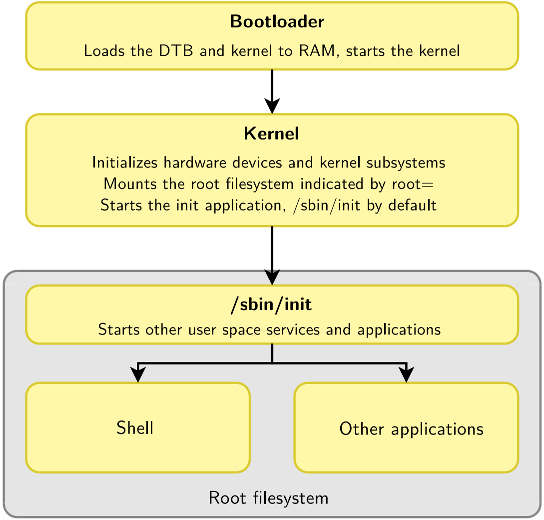
Introduction to Embedded Linux

Corrections, suggestions, contributions and translations are welcome!

Simplified Linux system architecture



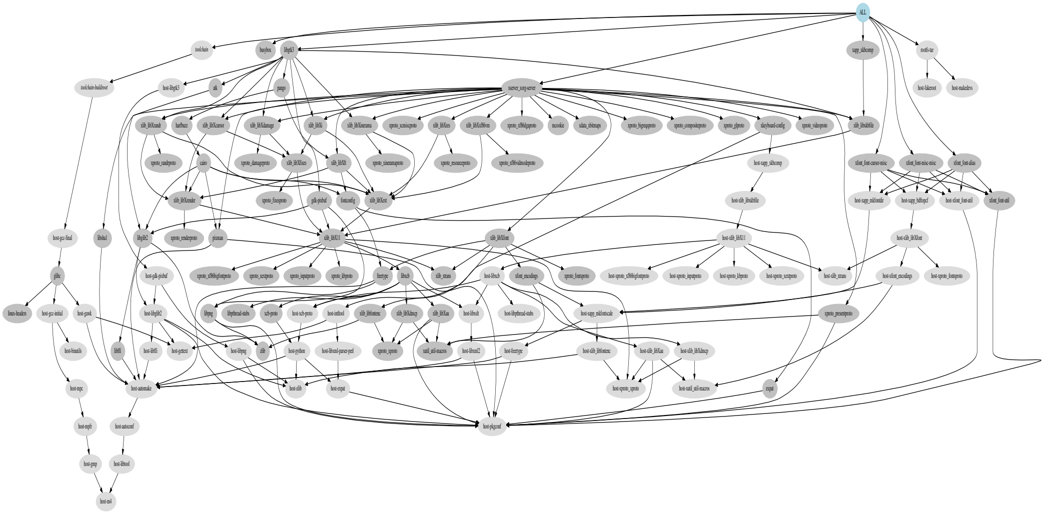
Overall Linux boot sequence



Embedded Linux work

- **BSP work**: porting the bootloader and Linux kernel, developing Linux device drivers.
- **system integration work**: assembling all the user space components needed for the system, configure them, develop the upgrade and recovery mechanisms, etc.
- **application development**: write the company-specific applications and libraries.

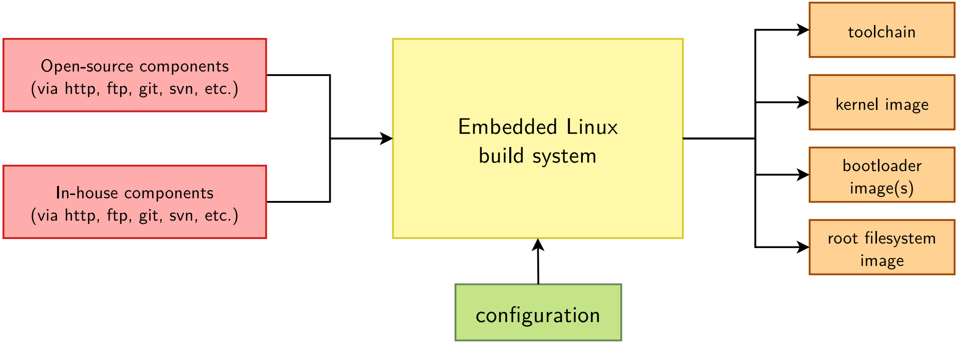
Complexity of user space integration



System integration: several possibilities

	Pros	Cons
Building everything manually	Full flexibility Learning experience	Dependency hell Need to understand a lot of details Version compatibility Lack of reproducibility
Binary distribution Debian, Ubuntu, Fedora, etc.	Easy to create and extend	Hard to customize Hard to optimize (boot time, size) Hard to rebuild the full system from source Large system Uses native compilation (slow) No well-defined mechanism to generate an image Lots of mandatory dependencies Not available for all architectures
Build systems Buildroot, Yocto, PTXdist, etc.	Nearly full flexibility Built from source: customization and optimization are easy Fully reproducible Uses cross-compilation Have embedded specific packages not necessarily in desktop distros Make more features optional	Not as easy as a binary distribution Build time

Embedded Linux build system: principle

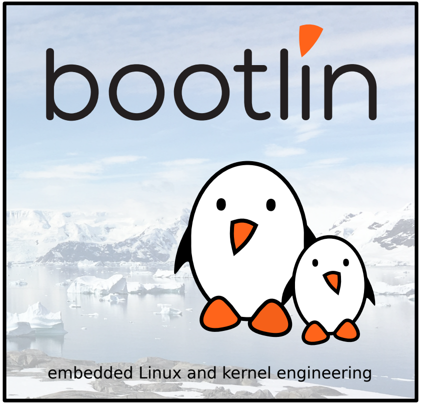


- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines
- Recipes for building components → easy

Embedded Linux build system: tools

- A wide range of solutions: Yocto/OpenEmbedded, PTXdist, Buildroot, OpenWRT, and more.
- Today, two solutions are emerging as the most popular ones
  - **Yocto/OpenEmbedded**
    Builds a complete Linux distribution with binary packages. Powerful, but somewhat complex, and quite steep learning curve.
  - **Buildroot**
    Builds a root filesystem image, no binary packages. Much simpler to use, understand and modify.

Introduction to Buildroot



Buildroot at a glance

- Can build a toolchain, a rootfs, a kernel, a bootloader
- **Easy to configure**: menuconfig, xconfig, etc.
- **Fast**: builds a simple root filesystem in a few minutes
- Easy to understand: written in make, extensive documentation
- **Small** root filesystem, starting at 2 MB
- **3200+ packages** for user space libraries/apps available
- **Many architectures** supported
- **Well-known technologies**: *make* and *kconfig*
- Vendor neutral
- Active community, regular releases
  - The present slides cover *Buildroot 2024.02*. There may be some differences if you use older or newer Buildroot versions.
- https://buildroot.org

Buildroot design goals

- Buildroot is designed with a few key goals:
  - Simple to use
  - Simple to customize
  - Reproducible builds
  - Small root filesystem
  - Relatively fast boot
  - Easy to understand
- Some of these goals require to not necessarily support all possible features
- There are some more complicated and featureful build systems available (Yocto Project, OpenEmbedded)

Getting Buildroot

- Stable Buildroot releases are published every three months
  - `YYYY.02`, `YYYY.05`, `YYYY.08`, `YYYY.11`
- Tarballs are available for each stable release
  - https://buildroot.org/downloads/
- However, it is generally more convenient to clone the Git repository
  - Allows to clearly identify the changes you make to the Buildroot source code
  - Simplifies the upstreaming of the Buildroot changes
  - `git clone https://git.buildroot.net/buildroot`
  - Git tags available for every stable release.
- One **long term support** release published every year
  - Maintained during one year
  - Security fixes, bug fixes, build fixes
  - Current LTS is release is `2024.02`, maintained until March-April 2025, next one will be `2025.02`.

Using Buildroot

- Implemented in `make`
  - With a few helper shell scripts
- All interaction happens by calling `make` in the main Buildroot sources directory.

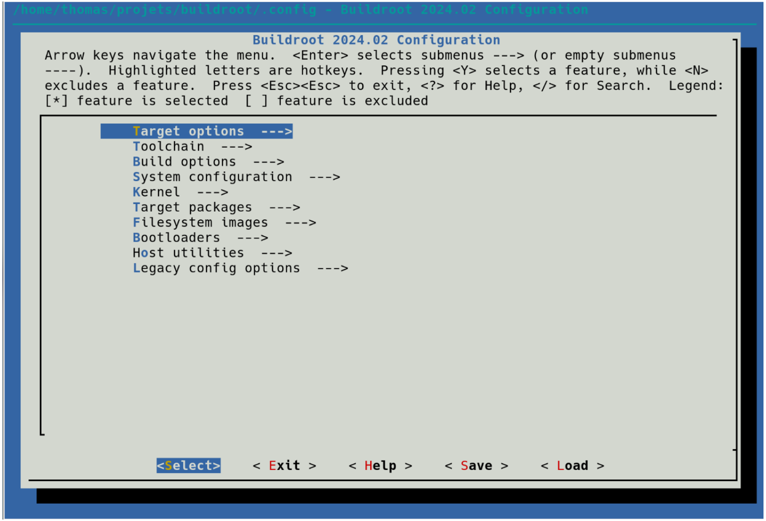
```
$ cd buildroot/
$ make help
```

- No need to run as `root`, Buildroot is designed to be executed with normal user privileges.
  - Running as root is even strongly discouraged!

Configuring Buildroot

- Like the Linux kernel, uses *Kconfig*
- A choice of configuration interfaces:
  - `make menuconfig`
  - `make nconfig`
  - `make xconfig`
  - `make gconfig`
- Make sure to install the relevant libraries in your system (*ncurses* for menuconfig/nconfig, *Qt* for xconfig, *Gtk* for gconfig)

Main menuconfig menu



Running the build

- As simple as:

```
$ make
```

- Often useful to keep a log of the build output, for analysis or investigation:

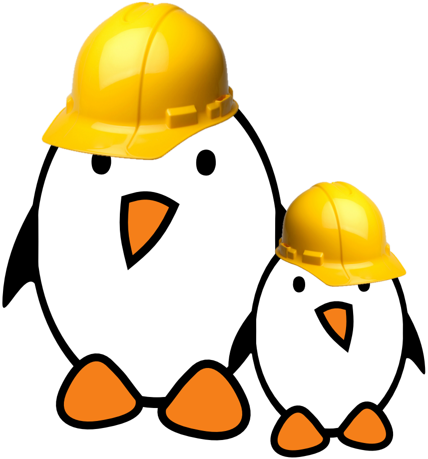
```
$ make 2>&1 | tee build.log
```

- Or the helper shell script provided by Buildroot:

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

Build results

- The build results are located in `output/images`
- Depending on the configuration, this directory will contain:
  - One or several root filesystem images, in various formats
  - One kernel image, possibly one or several Device Tree blobs
  - One or several bootloader images
- There is no standard way to install the images on any given device
  - Those steps are very device specific
  - Buildroot provides some tools to generate SD card / USB key images (*genimage*) or directly to flash or boot specific platforms: SAM-BA for Microchip, imx-usb-loader for i.MX6, OpenOCD, etc.



- Get Buildroot
- Configure a minimal system with Buildroot for the target hardware
- Do the build
- Prepare the target hardware for usage
- Flash and test the generated system

Managing the build and the configuration

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 4742 lines (as of Buildroot 2024.02)
  - 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 (3200+)
  - `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`
  - ≈142 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

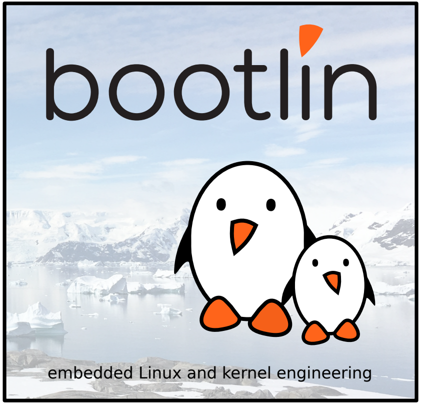
Build tree: $(O)/graphs

- 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

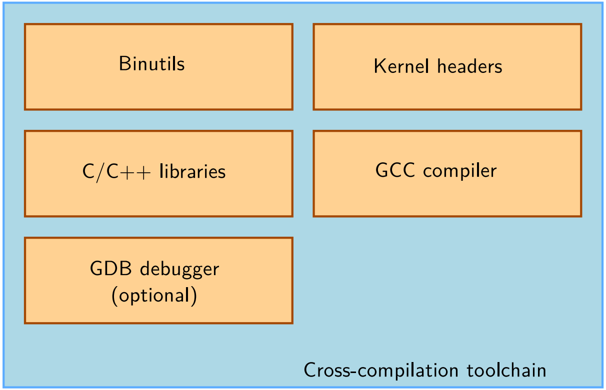
- output/
  - legal-info/
    - manifest.csv
    - host-manifest.csv
    - licenses.txt
    - licenses/
    - sources/
    - ...
  - Legal information: license of all packages, and their source code, plus a licensing manifest
  - Useful for license compliance
  - make legal-info
  - Variable: LEGAL_INFO_DIR

Toolchains in Buildroot



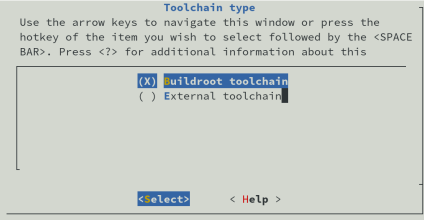
What is a cross-compilation toolchain?

- 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

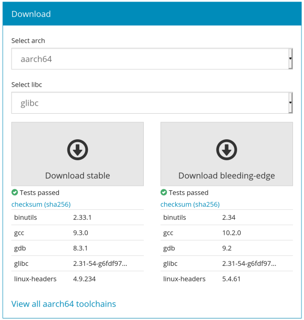
- Allows to re-use existing pre-built toolchains
- Great to:
  - save the build time of the toolchain
  - use vendor provided toolchain that are supposed to be reliable
- Several options:
  - Use an existing toolchain profile known by Buildroot
  - Download and install a custom external toolchain
  -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

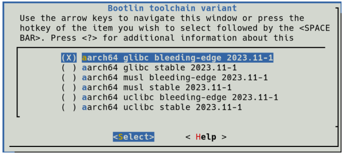
Existing external toolchain profile

- 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

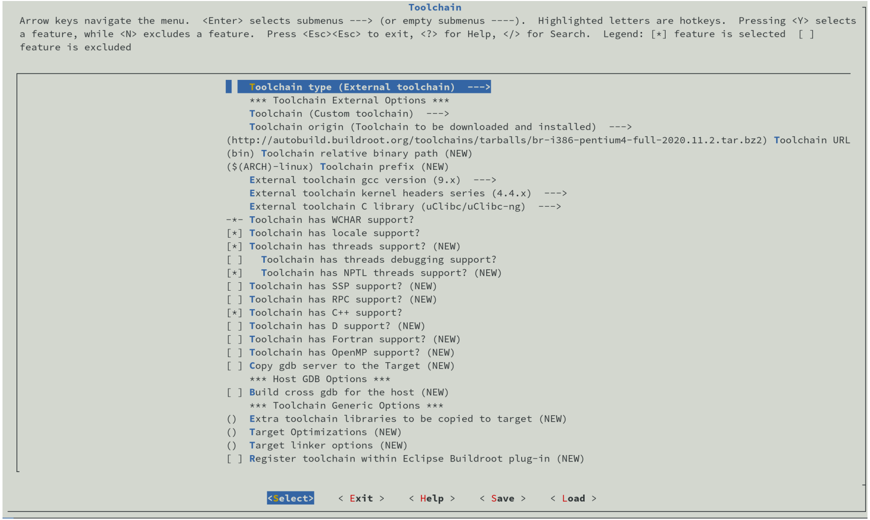




Custom external toolchains

- 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



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

Kernel version

- Most packages in Buildroot have a version fixed in their corresponding Buildroot package
- Some packages are more hardware-related, and there is a need to configure their version in a custom way
  - Linux kernel, bootloaders, firmware, etc.
- Linux kernel version options:
  - Latest version
  - Latest CIP SLTS
  - Latest CIP RT SLTS
  - Custom version
  - Custom Git repository
  - Custom Mercurial repository
  - Custom Subversion repository
- Using a custom version is recommended → ensures that upgrading Buildroot doesn't imply a change in the kernel version
- No relationship between Buildroot version and kernel version

Kernel version configuration

```
/home/thomas/projets/buildroot/.config - Buildroot 2024.02 Configuration

<X> Latest version (6.6)
    Latest CIP SLTS version (5.10.162-cip24)
    Latest CIP RT SLTS version (5.10.162-cip24-rt10)
    Custom version
    Custom tarball
    Custom Git repository
    Custom Mercurial repository
    Custom Subversion repository
```

Kernel configuration vs. Buildroot configuration

- 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

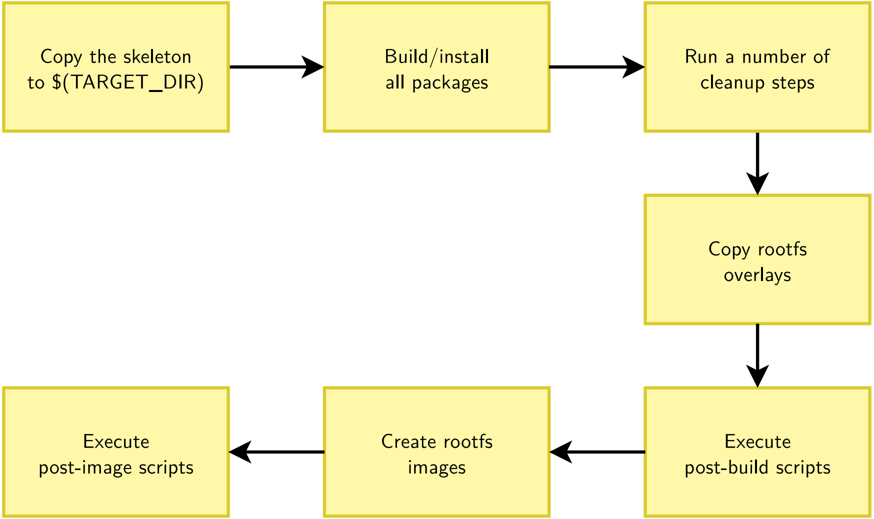
- Running one of the Linux kernel configuration interfaces:
  - `make linux-menuconfig`
  - `make linux-nconfig`
  - `make linux-xconfig`
  - `make linux-gconfig`
- Will load either the defined kernel *defconfig* or custom configuration file, and start the corresponding Linux kernel configuration interface.
- 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

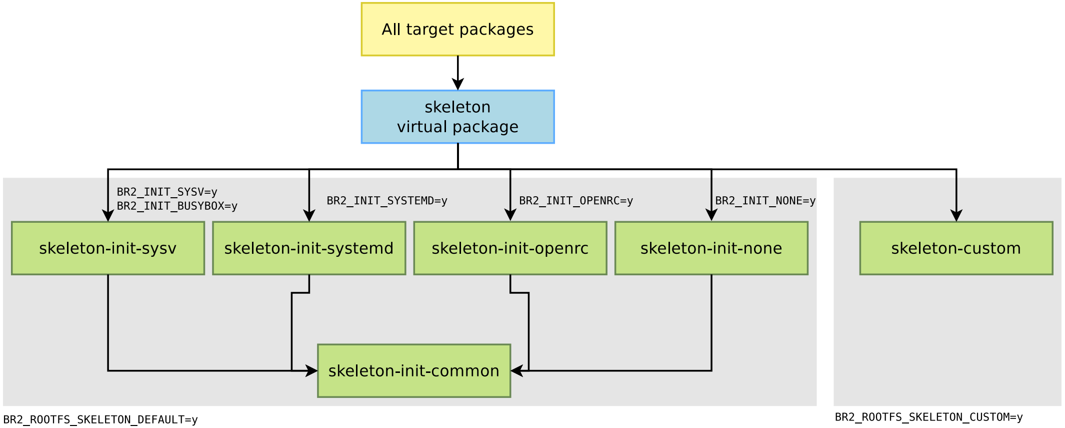
Overall rootfs construction steps



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

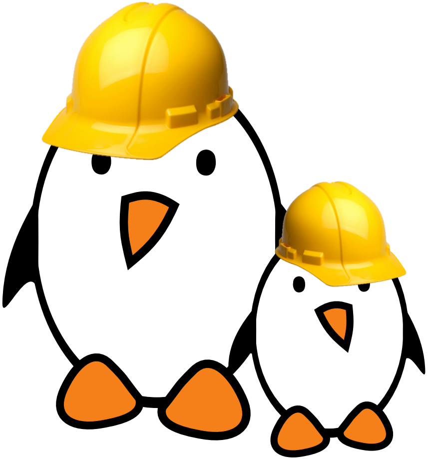
```
dd if=output/images/sdcard.img of=/dev/sdb
```

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

Download infrastructure in Buildroot

Introduction

- One important aspect of Buildroot is to fetch source code or binary files from third party projects.
- Download supported from HTTP(S), FTP, Git, Subversion, CVS, Mercurial, etc.
- Being able to do reproducible builds over a long period of time requires understanding the download infrastructure.

Download location

- Each Buildroot package indicates in its `.mk` file which files it needs to be downloaded.
- Can be a tarball, one or several patches, binary files, etc.
- When downloading a file, Buildroot will successively try the following locations:
  1. The local `$(DL_DIR)` directory where downloaded files are kept
  2. The **primary site**, as indicated by `BR2_PRIMARY_SITE`
  3. The **original site**, as indicated by the package `.mk` file
  4. The **backup Buildroot mirror**, as indicated by `BR2_BACKUP_SITE`

DL_DIR

- Once a file has been downloaded by Buildroot, it is cached in the directory pointed by `$(DL_DIR)`, in a sub-directory named after the package.
- By default, `$(TOPDIR)/dl`
- Can be changed
  - using the `BR2_DL_DIR` configuration option
  - or by passing the `BR2_DL_DIR` environment variable, which overrides the config option of the same name
- The download mechanism is written in a way that allows independent parallel builds to share the same `DL_DIR` (using atomic renaming of files)
- No cleanup mechanism: files are only added, never removed, even when the package version is updated.

Primary site

- The BR2_PRIMARY_SITE option allows to define the location of a HTTP or FTP server.
- By default empty, so this feature is disabled.
- When defined, used in priority over the original location.
- Allows to do a local mirror, in your company, of all the files that Buildroot needs to download.
- When option BR2_PRIMARY_SITE_ONLY is enabled, only the *primary site* is used
  - It does not fall back on the original site and the backup Buildroot mirror
  - Guarantees that all downloads must be in the primary site

Backup Buildroot mirror

- Since sometimes the upstream locations disappear or are temporarily unavailable, having a backup server is useful
- Address configured through `BR2_BACKUP_SITE`
- Defaults to http://sources.buildroot.net
  - maintained by the Buildroot community
  - updated before every Buildroot release to contain the downloaded files for all packages
  - exception: cannot store all possible versions for packages that have their version as a configuration option. Generally only affects the kernel or bootloader, which typically don't disappear upstream.

Special case of VCS download

- When a package uses the source code from Git, Subversion or another VCS, Buildroot cannot directly download a tarball.
- It uses a VCS-specific method to fetch the specified version of the source from the VCS repository
- The source code is checked-out/cloned inside `DL_DIR` and kept to speed-up further downloads of the same project (caching only implemented for Git)
- Finally a tarball containing only the source code (and not the version control history or metadata) is created and stored in `DL_DIR`
  - Example: `avrdude-eabe067c4527bc2eedc5db9288ef5cf1818ec720.tar.gz`
- This tarball will be re-used for the next builds, and attempts are made to download it from the primary and backup sites.
- Due to this, always use a tag name or a full commit id, and never a branch name: the code will never be re-downloaded when the branch is updated.

Vendoring

- Some language-specific package management systems like to download the dependencies by themselves: *vendoring*
- Examples: *Cargo* in the Rust ecosystem, or *Go*
- Problem for build systems: reproducibility of the builds, licensing, offline builds
- Buildroot supports vendoring dependencies for *cargo* and *go* packages
- Right after the download of the package source code, Buildroot invokes the language-specific vendoring tool, and bundles the dependencies inside the tarball

File integrity checking

- Buildroot packages can provide a `.hash` file to provide *hashes* for the downloaded files.
- The download infrastructure uses this hash file when available to check the integrity of the downloaded files.
- Hashes are checked every time a downloaded file is used, even if it is already cached in `$(DL_DIR)`.
- If the hash is incorrect, the download infrastructure attempts to re-download the file once. If that still fails, the build aborts with an error.

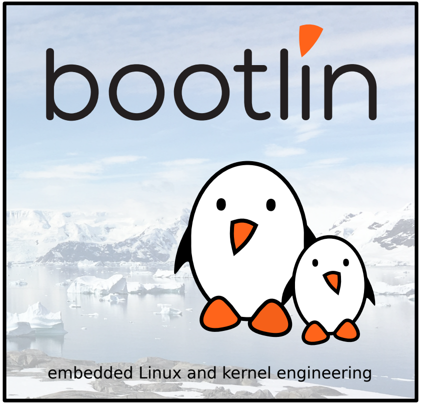
Hash checking message

```
strace-4.10.tar.xz: OK (md5: 107a5be455493861189e9b57a3a51912)
strace-4.10.tar.xz: OK (sha1: 5c3ec4c5a9eeb440d7ec70514923c2e7e7f9ab6c)
>>> strace 4.10 Extracting
```

Download-related make targets

- `make source`
  - Triggers the download of all the files needed to build the current configuration.
  - All files are stored in `$(DL_DIR)`
  - Allows to prepare a fully offline build
- `make external-deps`
  - Lists the files from `$(DL_DIR)` that are needed for the current configuration to build.
  - Does not guarantee that all files are in `$(DL_DIR)`, a `make source` is required

GNU Make 101



- Buildroot being implemented in **GNU Make**, it is quite important to know the basics of this language
  - Basics of *make* rules
  - Defining and referencing variables
  - Conditions
  - Defining and using functions
  - Useful *make* functions
- This does not aim at replacing a full course on *GNU Make*
- https://www.gnu.org/software/make/manual/make.html
- https://www.nostarch.com/gnumake

Basics of *make* rules

- At their core, *Makefiles* are simply defining **rules** to create **targets** from **prerequisites** using **recipe commands**

```
TARGET ...: PREREQUISITES ...
        RECIPE
        ...
```

- **target**: name of a file that is generated. Can also be an arbitrary action, like `clean`, in which case it's a **phony target**
- **prerequisites**: list of files or other targets that are needed as dependencies of building the current target.
- **recipe**: list of shell commands to create the target from the prerequisites

Rule example

Makefile

```
clean:
	rm -rf $(TARGET_DIR) $(BINARIES_DIR) $(HOST_DIR) \
	        $(BUILD_DIR) $(BASE_DIR)/staging \
	        $(LEGAL_INFO_DIR)

distclean: clean
	[...]
	rm -rf $(BR2_CONFIG) $(CONFIG_DIR)/.config.old \
	       $(CONFIG_DIR)/.auto.deps
```

- `clean` and `distclean` are phony targets

Defining and referencing variables

- Defining variables is done in different ways:
  - `FOOBAR = value`, expanded at time of use
  - `FOOBAR := value`, expanded at time of assignment
  - `FOOBAR += value`, append to the variable, with a separating space, defaults to expanded at the time of use
  - `FOOBAR ?= value`, defined only if not already defined
  - Multi-line variables are described using `define NAME ... endef`:

```
define FOOBAR
line 1
line 2
endef
```

- Make variables are referenced using the `$(FOOBAR)` syntax.

Conditions

- With `ifeq` or `ifneq`

```
ifeq ($(BR2_CCACHE),y)
CCACHE := $(HOST_DIR)/bin/ccache
endif

distclean: clean
ifeq ($(DL_DIR),$(TOPDIR)/dl)
	rm -rf $(DL_DIR)
endif
```

- With the `$(if ...)` make function:

```
HOSTAPD_LIBS += $(if $(BR2_STATIC_LIBS),-lcrypto -lz)
```

Defining and using functions

- Defining a function is exactly like defining a variable:

```
MESSAGE = echo "$(TERM_BOLD)>>> $($(PKG)_NAME) $($(PKG)_VERSION) $(call qstrip,$(1))$(TERM_RESET)"

define legal-license-header # pkg, license-file, {HOST|TARGET}
	printf "$(LEGAL_INFO_SEPARATOR)\n\t$(1):\
		$(2)\n$(LEGAL_INFO_SEPARATOR)\n\n\n" >>$(LEGAL_LICENSES_TXT_$(3))
endef
```

- Arguments accessible as `$(1)`, `$(2)`, etc.
- Called using the `$(call func,arg1,arg2)` construct

```
$(BUILD_DIR)/%/.stamp_extracted:
	[...]
	@$(call MESSAGE,"Extracting")

define legal-license-nofiles # pkg, {HOST|TARGET}
	$(call legal-license-header,$(1),unknown license file(s),$(2))
endef
```

Useful *make* functions

- subst and patsubst to replace text

```
ICU_SOURCE = icu4c-$(subst .,_,$(ICU_VERSION))-src.tgz
```

- filter and filter-out to filter entries
- foreach to implement loops

```
$(foreach incdir,$(TI_GFX_HDR_DIRS),
      $(INSTALL) -d $(STAGING_DIR)/usr/include/$(notdir $(incdir)); \
      $(INSTALL) -D -m 0644 $(@D)/include/$(incdir)/*.h \
              $(STAGING_DIR)/usr/include/$(notdir $(incdir))/
)
```

- dir, notdir, addsuffix, addprefix to manipulate file names

```
UBOOT_SOURCE = $(notdir $(UBOOT_TARBALL))

IMAGEMAGICK_CONFIG_SCRIPTS = \
        $(addsuffix -config,Magick MagickCore MagickWand Wand)
```

- And many more, see the *GNU Make* manual for details.

Writing recipes

- Recipes are just shell commands
- Each line must be indented with one `Tab`
- Each line of shell command in a given recipe is independent from the other: variables are not shared between lines in the recipe
- Need to use a single line, possibly split using \, to do complex shell constructs
- Shell variables must be referenced using `$$name`.

package/pppd/pppd.mk

```
define PPPD_INSTALL_RADIUS
	...
	for m in $(PPPD_RADIUS_CONF); do \
		$(INSTALL) -m 644 -D $(PPPD_DIR)/pppd/plugins/radius/etc/$$m \
			$(TARGET_DIR)/etc/ppp/radius/$$m; \
	done
	...
endef
```

Integrating new packages in Buildroot

Why adding new packages in Buildroot?

- A *package* in Buildroot-speak is the **set of meta-information needed to automate the build process** of a certain component of a system.
- Can be used for open-source, third party proprietary components, or in-house components.
- Can be used for user space components (libraries and applications) but also for firmware, kernel drivers, bootloaders, etc.
- Do not confuse with the notion of *binary package* in a regular Linux distribution.

Basic elements of a Buildroot package

- A directory, `package/foo`
- A `Config.in` file, written in *kconfig* language, describing the configuration options for the package.
- A `<pkg>.mk` file, written in *make*, describing where to fetch the source, how to build and install it, etc.
- An optional `<pkg>.hash` file, providing hashes to check the integrity of the downloaded tarballs and license files.
- Optionally, `.patch` files, that are applied on the package source code before building.
- Optionally, any additional file that might be useful for the package: init script, example configuration file, etc.

Config.in file

package/<pkg>/Config.in: basics

- Describes the configuration options for the package.
- Written in the *kconfig* language.
- One option is mandatory to enable/disable the package, it **must** be named BR2_PACKAGE_<PACKAGE>.

```
config BR2_PACKAGE_STRACE
	bool "strace"
	help
	  A useful diagnostic, instructional, and debugging tool.
	  Allows you to track what system calls a program makes
	  while it is running.

	  http://sourceforge.net/projects/strace/
```

- The main package option is a `bool` with the package name as the prompt. Will be visible in `menuconfig`.
- The help text give a quick description, and the homepage of the project.

package/<pkg>/Config.in: inclusion

- The hierarchy of configuration options visible in `menuconfig` is built by reading the top-level `Config.in` file and the other `Config.in` file it includes.
- All `package/<pkg>/Config.in` files are included from `package/Config.in`.
- The location of a package in one of the package sub-menu is decided in this file.

package/Config.in

```
menu "Target packages"
menu "Audio and video applications"
        source "package/alsa-utils/Config.in"
        ...
endmenu
...
menu "Libraries"
menu "Audio/Sound"
        source "package/alsa-lib/Config.in"
        ...
endmenu
...
```

package/<pkg>/Config.in: dependencies

- *kconfig* allows to express dependencies using `select` or `depends on` statements
  - `select` is an automatic dependency: if option *A* `select` option *B*, as soon as *A* is enabled, *B* will be enabled, and cannot be unselected.
  - `depends on` is a user-assisted dependency: if option *A* `depends on` option *B*, *A* will only be visible when *B* is enabled.
- Buildroot uses them as follows:
  - `depends on` for architecture, toolchain feature, or *big* feature dependencies. E.g: package only available on x86, or only if wide char support is enabled, or depends on Python.
  - `select` for enabling the necessary other packages needed to build the current package (libraries, etc.)
- Such dependencies only ensure consistency at the configuration level. They **do not guarantee build ordering**!

package/<pkg>/Config.in: dependency example

btrfs-progs package

```
config BR2_PACKAGE_BTRFS_PROGS
        bool "btrfs-progs"
        depends on BR2_USE_MMU # util-linux
        depends on BR2_TOOLCHAIN_HAS_THREADS
        select BR2_PACKAGE_LZO
        select BR2_PACKAGE_UTIL_LINUX
        select BR2_PACKAGE_UTIL_LINUX_LIBBLKID
        select BR2_PACKAGE_UTIL_LINUX_LIBUUID
        select BR2_PACKAGE_ZLIB
        help
          Btrfs filesystem utilities

          https://btrfs.wiki.kernel.org/index.php/Main_Page

comment "btrfs-progs needs a toolchain w/ threads"
        depends on BR2_USE_MMU
        depends on !BR2_TOOLCHAIN_HAS_THREADS
```

- `depends on BR2_USE_MMU`, because the package uses `fork()`. Note that there is no comment displayed about this dependency, because it's a limitation of the architecture.
- `depends on BR2_TOOLCHAIN_HAS_THREADS`, because the package requires thread support from the toolchain. There is an associated comment, because such support can be added to the toolchain.
- Multiple `select BR2_PACKAGE_*`, because the package needs numerous libraries.

Dependency propagation

- A limitation of *kconfig* is that it doesn't propagate `depends on` dependencies accross `select` dependencies.
- Scenario: if package *A* has a `depends on FOO`, and package *B* has a `select A`, then package *B* must replicate the `depends on FOO`.

libglib2 package

```
config BR2_PACKAGE_LIBGLIB2
	bool "libglib2"
	select BR2_PACKAGE_GETTEXT if ...
	select BR2_PACKAGE_LIBICONV if ...
	select BR2_PACKAGE_LIBFFI
	select BR2_PACKAGE_ZLIB
	[...]
	depends on BR2_USE_WCHAR # gettext
	depends on BR2_TOOLCHAIN_HAS_THREADS
	depends on BR2_USE_MMU # fork()
[...]
```

neard package

```
config BR2_PACKAGE_NEARD
	bool "neard"
	depends on BR2_USE_WCHAR # libglib2
	# libnl, dbus, libglib2
	depends on BR2_TOOLCHAIN_HAS_THREADS
	depends on BR2_USE_MMU # dbus, libglib2
	select BR2_PACKAGE_DBUS
	select BR2_PACKAGE_LIBGLIB2
	select BR2_PACKAGE_LIBNL
[...]
```

`Config.in.host` for host packages?

- Most of the packages in Buildroot are *target* packages, i.e. they are cross-compiled for the target architecture, and meant to be run on the target platform.
- Some packages have a *host* variant, built to be executed on the build machine. Such packages are needed for the build process of other packages.
- The majority of *host* packages are not visible in `menuconfig`: they are just dependencies of other packages, the user doesn't really need to know about them.
- A few of them are potentially directly useful to the user (flashing tools, etc.), and can be shown in the *Host utilities* section of `menuconfig`.
- In this case, the configuration option is in a `Config.in.host` file, included from `package/Config.in.host`, and the option must be named `BR2_PACKAGE_HOST_<PACKAGE>`.

Config.in.host example

package/Config.in.host

```
menu "Host utilities"

        source "package/genimage/Config.in.host"
        source "package/lpc3250loader/Config.in.host"
        source "package/openocd/Config.in.host"
        source "package/qemu/Config.in.host"

endmenu
```

package/openocd/Config.in.host

```
config BR2_PACKAGE_HOST_OPENOCD
        bool "host openocd"
        help
          OpenOCD - Open On-Chip Debugger

          http://openocd.org
```

`Config.in` sub-options

- Additional sub-options can be defined to further configure the package, to enable or disable extra features.
- The value of such options can then be fetched from the package `.mk` file to adjust the build accordingly.
- Run-time configuration does not belong to `Config.in`.

package/pppd/Config.in

```
config BR2_PACKAGE_PPPD
        bool "pppd"
        depends on !BR2_STATIC_LIBS
        depends on BR2_USE_MMU
        ...

if BR2_PACKAGE_PPPD

config BR2_PACKAGE_PPPD_FILTER
        bool "filtering"
        select BR2_PACKAGE_LIBPCAP
        help
          Packet filtering abilities for pppd. If enabled,
          the pppd active-filter and pass-filter options
          are available.

config BR2_PACKAGE_PPPD_RADIUS
        bool "radius"
        help
          Install RADIUS support for pppd

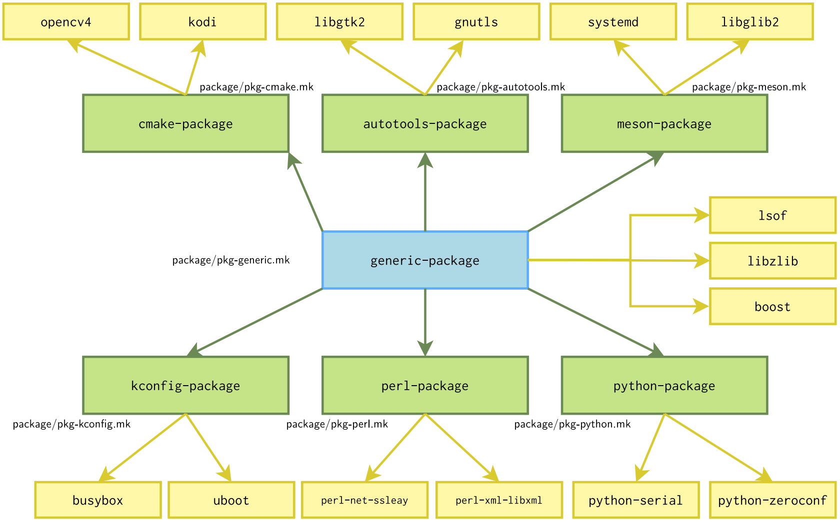
endif
```

Package infrastructures

Package infrastructures: what is it?

- Each software component to be built by Buildroot comes with its own *build system*.
- Buildroot does not re-invent the build system of each component, it simply uses it.
- Numerous build systems available: hand-written Makefiles or shell scripts, *autotools*, *Meson*, *CMake* and also some specific to languages: Python, Perl, Lua, Erlang, etc.
- In order to avoid duplicating code, Buildroot has *package infrastructures* for well-known build systems.
- And a generic package infrastructure for software components with non-standard build systems.

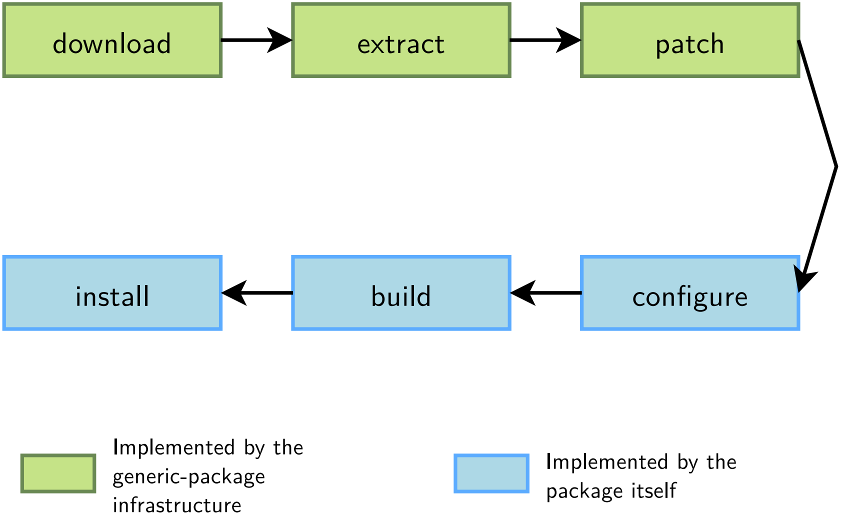
Package infrastructures



generic-package infrastructure

- To be used for software components having non-standard build systems.
- Implements a default behavior for the downloading, extracting and patching steps of the package build process.
- Implements init script installation, legal information collection, etc.
- Leaves to the package developer the responsibility of describing what should be done for the configuration, building and installation steps.

generic-package: steps



Other package infrastructures

- The other package infrastructures are meant to be used when the software component uses a well-known build system.
- They *inherit* all the behavior of the `generic-package` infrastructure: downloading, extracting, patching, etc.
- And in addition to that, they typically implement a default behavior for the configuration, compilation and installation steps.
- For example, `autotools-package` will implement the configuration step as a call to the `./configure` script with the right arguments.
- `pkg-kconfig` is an exception, it only provides some helpers for packages using Kconfig, but does not implement the configure, build and installation steps.

.mk file for generic-package

The <pkg>.mk file

- The .mk file of a package does not look like a normal Makefile.
- It is a succession of variable definitions, which must be prefixed by the uppercase package name.
  - `FOOBAR_SITE = https://foobar.com/downloads/`
  - ```
    define FOOBAR_BUILD_CMDS
            $(MAKE) -C $(@D)
    endef
    ```
- And ends with a call to the desired package infrastructure macro.
  - `$(eval $(generic-package))`
  - `$(eval $(autotools-package))`
  - `$(eval $(host-autotools-package))`
- The variables tell the package infrastructure what to do for this specific package.
    ```

Naming conventions

- The Buildroot package infrastructures make a number of assumption on variables and files naming.
- The following **must** match to allow the package infrastructure to work for a given package:
  - The directory where the package description is located **must** be `package/<pkg>/`, where `<pkg>` is the lowercase name of the package.
  - The `Config.in` option enabling the package **must** be named `BR2_PACKAGE_<PKG>`, where `<PKG>` is the uppercase name of the package.
  - The variables in the `.mk` file **must** be prefixed with `<PKG>_`, where `<PKG>` is the uppercase name of the package.
- Note: a `-` in the lower-case package name is translated to `_` in the upper-case package name.

Naming conventions: global namespace

- The package infrastructure expects all variables it uses to be prefixed by the uppercase package name.
- If your package needs to define additional private variables not used by the package infrastructure, they **should** also be prefixed by the **uppercase package name**.
- The **namespace of variables is global** in Buildroot!
  - If two packages created a variable named `BUILD_TYPE`, it will silently conflict.

Behind the scenes

- Behind the scenes, `$(eval $(generic-package))`:
  - is a *make* macro that is expanded
  - infers the name of the current package by looking at the directory name: `package/<pkg>/<pkg>.mk`: `<pkg>` is the package name
  - will use all the variables prefixed by `<PKG>_`
  - and expand to a set of *make* rules and variable definitions that describe what should be done for each step of the package build process

.mk file: accessing the configuration

- The Buildroot `.config` file is a succession of lines `name = value`
  - This file is valid *make* syntax!
- The main Buildroot `Makefile` simply includes it, which turns every Buildroot configuration option into a *make* variable.
- From a package `.mk` file, one can directly use such variables:

```
ifeq ($(BR2_PACKAGE_LIBCURL),y)
...
endif

FOO_DEPENDENCIES += $(if $(BR2_PACKAGE_TIFF),tiff)
```

- Hint: use the *make* `qstrip` function to remove double quotes on string options:

```
NODEJS_MODULES_LIST = $(call qstrip,$(BR2_PACKAGE_NODEJS_MODULES_ADDITIONAL))
```

Download related variables

- `<pkg>_SITE`, **download location**
  - HTTP(S) or FTP URL where a tarball can be found, or the address of a version control repository.
  - `CAIRO_SITE = http://cairographics.org/releases`
  - `FMC_SITE = git://git.freescale.com/ppc/sdk/fmc.git`
- `<pkg>_VERSION`, **version of the package**
  - version of a tarball, or a commit, revision or tag for version control systems
  - `CAIRO_VERSION = 1.14.2`
  - `FMC_VERSION = fsl-sdk-v1.5-rc3`
- `<pkg>_SOURCE`, **file name** of the tarball
  - The full URL of the downloaded tarball is `$(<pkg>_SITE)/$(<pkg>_SOURCE)`
  - When not specified, defaults to `<pkg>-$(<pkg>_VERSION).tar.gz`
  - `CAIRO_SOURCE = cairo-$(CAIRO_VERSION).tar.xz`

Available download methods

- Buildroot can fetch the source code using different methods:
  - `wget`, for FTP/HTTP downloads
  - `scp`, to fetch the tarball using SSH/SCP
  - `svn`, for Subversion
  - `cvs`, for CVS
  - `git`, for Git
  - `hg`, for Mercurial
  - `bzr`, for Bazaar
  - `file`, for a local tarball
  - `local`, for a local directory
- In most cases, the fetching method is guessed by Buildroot using the `<pkg>_SITE` variable.
- Exceptions:
  - Git, Subversion or Mercurial repositories accessed over HTTP or SSH.
  - `file` and `local` methods
- In such cases, use `<pkg>_SITE_METHOD` explicitly.

Download methods examples

- Subversion repository accessed over HTTP:

```
CJSON_VERSION = 58
CJSON_SITE_METHOD = svn
CJSON_SITE = http://svn.code.sf.net/p/cjson/code
```

- Source code available in a local directory:

```
MYAPP_SITE = $(TOPDIR)/../apps/myapp
MYAPP_SITE_METHOD = local
```

  - The "*download*" will consist in copying the source code from the designated directory to the Buildroot per-package build directory.

Downloading more elements

- <pkg>_PATCH, a list of patches to download and apply before building the package. They are automatically applied by the package infrastructure.
- <pkg>_EXTRA_DOWNLOADS, a list of additional files to download together with the package source code. It is up to the package .mk file to do something with them.
- Two options:
  - Just a file name: assumed to be relative to <pkg>_SITE.
  - A full URL: downloaded over HTTP, FTP.
- Examples:

sysvinit.mk

```
SYSVINIT_PATCH = sysvinit_$(SYSVINIT_VERSION)dsf-13.1+squeeze1.diff.gz
```

perl.mk

```
PERL_CROSS_SITE = http://raw.github.com/arsv/perl-cross/releases
PERL_CROSS_SOURCE = perl-$(PERL_CROSS_BASE_VERSION)-cross-$(PERL_CROSS_VERSION).tar.gz
PERL_EXTRA_DOWNLOADS = $(PERL_CROSS_SITE)/$(PERL_CROSS_SOURCE)
```

Hash file

- In order to validate the integrity of downloaded files and license files, and make sure the user uses the version which was tested by the Buildroot developers, *cryptographic hashes* are used
- Each package may contain a file named `<package>.hash`, which gives the hashes of the files downloaded by the package.
- When present, the hashes for **all** files downloaded by the package must be documented.
- The *hash file* can also contain the hashes for the license files listed in `<pkg>_LICENSE_FILES`. This allows to detect changes in the license files.
- The syntax of the file is:

```
<hashtype>  <hash>  <file>
```

Note: the separator between fields is 2 spaces.

Hash file examples

package/perl/perl.hash

```
# Hashes from: https://www.cpan.org/src/5.0/perl-5.32.1.tar.xz.{md5,sha1,sha256}.txt
md5  7f104064b906ad8c7329ca5e409a32d7  perl-5.32.1.tar.xz
sha1  1fb4f710d139da1e1a3e1fa4eaba201fcaa8e18e  perl-5.32.1.tar.xz
sha256  57cc47c735c8300a8ce2fa0643507b44c4ae59012bfdad0121313db639e02309  perl-5.32.1.tar.xz

# Hashes from: https://github.com/arsv/perl-cross/releases/download/1.3.5/perl-cross-1.3.5.hash
sha256  91c66f6b2b99fccfd4fee14660b677380b0c98f9456359e91449798c2ad2ef25  perl-cross-1.3.5.tar.gz

# Locally calculated
sha256  dd90d4f42e4dcadf5a7c09eea0189d93c7b37ae560c91f0f6d5233ed3b9292a2  Artistic
sha256  d77d235e41d54594865151f4751e835c5a82322b0e87ace266567c3391a4b912  Copying
sha256  df6ad59aefea68676c38325f25f6707f026ddde6c71291b2ca231b6247859907  README
```

package/ipset/ipset.hash

```
# From http://ipset.netfilter.org/ipset-7.6.tar.bz2.md5sum.txt
md5 e107b679c3256af795261cece864d6d9  ipset-7.6.tar.bz2
# Calculated based on the hash above
sha256 0e7d44caa9c153d96a9b5f12644fbe35a632537a5a7f653792b72e53d9d5c2db  ipset-7.6.tar.bz2
# Locally calculated
sha256 231f7edcc7352d7734a96eef0b8030f77982678c516876fcb81e25b32d68564c  COPYING
```

Describing dependencies

- Dependencies expressed in `Config.in` do not enforce build order.
- The `<pkg>_DEPENDENCIES` variable is used to describe the dependencies of the current package.
- Packages listed in `<pkg>_DEPENDENCIES` are guaranteed to be built before the *configure* step of the current package starts.
- It can contain both target and host packages.
- It can be appended conditionally with additional dependencies.

python.mk

```
PYTHON_DEPENDENCIES = host-python libffi

ifeq ($(BR2_PACKAGE_PYTHON_READLINE),y)
PYTHON_DEPENDENCIES += readline
endif
```

Mandatory vs. optional dependencies

- Very often, software components have some **mandatory dependencies** and some **optional dependencies**, only needed for optional features.
- Handling mandatory dependencies in Buildroot consists in:
  - Using a `select` or `depends on` on the main package option in `Config.in`
  - Adding the dependency in `<pkg>_DEPENDENCIES`
- For optional dependencies, there are two possibilities:
  - Handle it automatically: in the `.mk` file, if the optional dependency is available, use it.
  - Handle it explicitly: add a package sub-option in the `Config.in` file.
- *Automatic* handling is usually preferred as it reduces the number of `Config.in` options, but it makes the possible dependency less visible to the user.

Dependencies: ntp example

- Mandatory dependency: `libevent`
- Optional dependency handled automatically: `openssl`

package/ntp/Config.in

```
config BR2_PACKAGE_NTP
        bool "ntp"
        select BR2_PACKAGE_LIBEVENT
[...]
```

package/ntp/ntp.mk

```
[...]
NTP_DEPENDENCIES = host-pkgconf libevent
[...]
ifeq ($(BR2_PACKAGE_OPENSSL),y)
NTP_CONF_OPTS += --with-crypto --enable-openssl-random
NTP_DEPENDENCIES += openssl
else
NTP_CONF_OPTS += --without-crypto --disable-openssl-random
endif
[...]
```

Dependencies: mpd example (1/2)

package/mpd/Config.in

```
menuconfig BR2_PACKAGE_MPD
        bool "mpd"
        depends on BR2_INSTALL_LIBSTDCPP
[...]
        select BR2_PACKAGE_BOOST
        select BR2_PACKAGE_LIBGLIB2
        select BR2_PACKAGE_LIBICONV if !BR2_ENABLE_LOCALE
[...]

config BR2_PACKAGE_MPD_FLAC
        bool "flac"
        select BR2_PACKAGE_FLAC
        help
          Enable flac input/streaming support.
          Select this if you want to play back FLAC files.
```

Dependencies: mpd example (2/2)

package/mpd/mpd.mk

```
MPD_DEPENDENCIES = host-pkgconf boost libglib2

[...]

ifeq ($(BR2_PACKAGE_MPD_FLAC),y)
MPD_DEPENDENCIES += flac
MPD_CONF_OPTS += --enable-flac
else
MPD_CONF_OPTS += --disable-flac
endif
```

Defining where to install (1)

- Target packages can install files to different locations:
  - To the *target* directory, `$(TARGET_DIR)`, which is what will be the target root filesystem.
  - To the *staging* directory, `$(STAGING_DIR)`, which is the compiler *sysroot*
  - To the *images* directory, `$(BINARIES_DIR)`, which is where final images are located.
- There are three corresponding variables, to define whether or not the package will install something to one of these locations:
  - `<pkg>_INSTALL_TARGET`, defaults to `YES`. If `YES`, then `<pkg>_INSTALL_TARGET_CMDS` will be called.
  - `<pkg>_INSTALL_STAGING`, defaults to `NO`. If `YES`, then `<pkg>_INSTALL_STAGING_CMDS` will be called.
  - `<pkg>_INSTALL_IMAGES`, defaults to `NO`. If `YES`, then `<pkg>_INSTALL_IMAGES_CMDS` will be called.

Defining where to install (2)

- A package for an application:
  - installs to `$(TARGET_DIR)` only
  - `<pkg>_INSTALL_TARGET` defaults to `YES`, so there is nothing to do
- A package for a shared library:
  - installs to both `$(TARGET_DIR)` and `$(STAGING_DIR)`
  - must set `<pkg>_INSTALL_STAGING = YES`
- A package for a pure header-based library, or a static-only library:
  - installs only to `$(STAGING_DIR)`
  - must set `<pkg>_INSTALL_TARGET = NO` and `<pkg>_INSTALL_STAGING = YES`
- A package installing a bootloader or kernel image:
  - installs to `$(BINARIES_DIR)`
  - must set `<pkg>_INSTALL_IMAGES = YES`

Defining where to install (3)

libyaml.mk

```
LIBYAML_INSTALL_STAGING = YES
```

eigen.mk

```
EIGEN_INSTALL_STAGING = YES
EIGEN_INSTALL_TARGET = NO
```

linux.mk

```
LINUX_INSTALL_IMAGES = YES
```

Describing actions for generic-package

- In a package using generic-package, only the download, extract and patch steps are implemented by the package infrastructure.
- The other steps should be described by the package .mk file:
  - <pkg>_CONFIGURE_CMDS, always called
  - <pkg>_BUILD_CMDS, always called
  - <pkg>_INSTALL_TARGET_CMDS, called when <pkg>_INSTALL_TARGET = YES, for target packages
  - <pkg>_INSTALL_STAGING_CMDS, called when <pkg>_INSTALL_STAGING = YES, for target packages
  - <pkg>_INSTALL_IMAGES_CMDS, called when <pkg>_INSTALL_IMAGES = YES, for target packages
  - <pkg>_INSTALL_CMDS, always called for host packages
- Packages are free to not implement any of these variables: they are all optional.

Describing actions: useful variables

Inside an action block, the following variables are often useful:

- `$(@D)` is the source directory of the package
- `$(MAKE)` to call `make`
- `$(MAKE1)` when the package doesn't build properly in parallel mode
- `$(TARGET_MAKE_ENV)` and `$(HOST_MAKE_ENV)`, to pass in the `$(MAKE)` environment to ensure the `PATH` is correct
- `$(TARGET_CONFIGURE_OPTS)` and `$(HOST_CONFIGURE_OPTS)` to pass `CC`, `LD`, `CFLAGS`, etc.
- `$(TARGET_DIR)`, `$(STAGING_DIR)`, `$(BINARIES_DIR)` and `$(HOST_DIR)`.

Describing actions: iodine.mk example

```
IODINE_VERSION = 0.7.0
IODINE_SITE = http://code.kryo.se/iodine
IODINE_DEPENDENCIES = zlib
IODINE_LICENSE = MIT
IODINE_LICENSE_FILES = README

IODINE_CFLAGS = $(TARGET_CFLAGS)
[...]

define IODINE_BUILD_CMDS
	$(TARGET_CONFIGURE_OPTS) CFLAGS="$(IODINE_CFLAGS)" \
		$(MAKE) ARCH=$(BR2_ARCH) -C $(@D)
endef

define IODINE_INSTALL_TARGET_CMDS
	$(TARGET_CONFIGURE_OPTS) $(MAKE) -C $(@D) install DESTDIR="$(TARGET_DIR)" prefix=/usr
endef

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

Describing actions: `libzlib.mk` example

```
LIBZLIB_VERSION = 1.2.11
LIBZLIB_SOURCE = zlib-$(LIBZLIB_VERSION).tar.xz
LIBZLIB_SITE = http://www.zlib.net
LIBZLIB_INSTALL_STAGING = YES

define LIBZLIB_CONFIGURE_CMDS
	(cd $(@D); rm -rf config.cache; \
		$(TARGET_CONFIGURE_ARGS) \
		$(TARGET_CONFIGURE_OPTS) \
		CFLAGS="$(TARGET_CFLAGS) $(LIBZLIB_PIC)" \
		./configure \
		$(LIBZLIB_SHARED) \
		--prefix=/usr \
	)
endef

define LIBZLIB_BUILD_CMDS
	$(TARGET_MAKE_ENV) $(MAKE1) -C $(@D)
endef

define LIBZLIB_INSTALL_STAGING_CMDS
	$(TARGET_MAKE_ENV) $(MAKE1) -C $(@D) DESTDIR=$(STAGING_DIR) LDCONFIG=true install
endef

define LIBZLIB_INSTALL_TARGET_CMDS
	$(TARGET_MAKE_ENV) $(MAKE1) -C $(@D) DESTDIR=$(TARGET_DIR) LDCONFIG=true install
endef

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

List of package infrastructures (1/2)

- `generic-package`, for packages not using a well-known build system. Already covered.
- `autotools-package`, for *autotools* based packages, covered later.
- `python-package`, for *distutils* and *setuptools* based Python packages
- `perl-package`, for *Perl* packages
- `luarocks-package`, for Lua packages hosted on `luarocks.org`
- `cmake-package`, for *CMake* based packages
- `waf-package`, for *Waf* based packages
- `qmake-package`, for *QMake* based packages

List of package infrastructures (2/2)

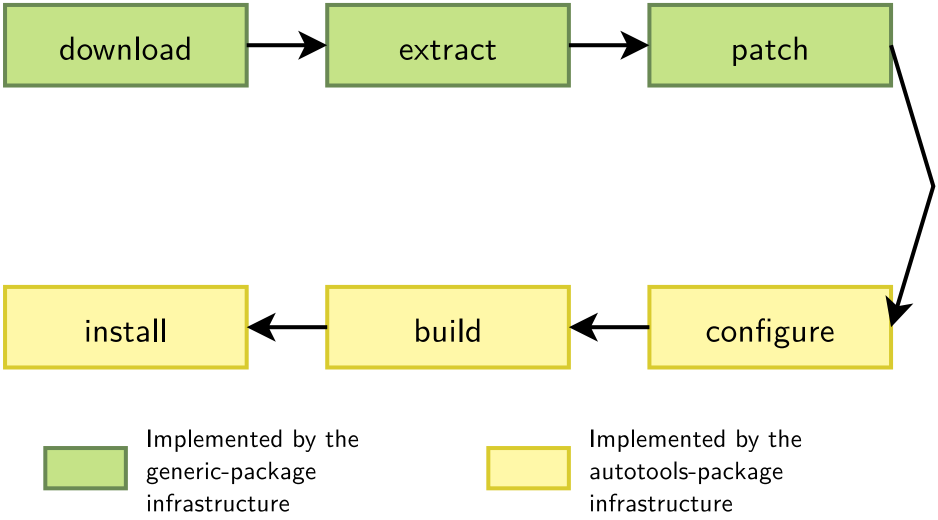
- `golang-package`, for packages written in Go
- `meson-package`, for packages using the Meson build system
- `cargo-package`, for packages written in Rust
- `kconfig-package`, to be used in conjunction with `generic-package`, for packages that use the *kconfig* configuration system
- `kernel-module-package`, to be used in conjunction with another package infrastructure, for packages that build kernel modules
- `rebar-package` for *Erlang* packages that use the *rebar* build system
- `virtual-package` for *virtual* packages, covered later.

autotools-package infrastructure

The autotools-package infrastructure: basics

- The autotools-package infrastructure inherits from generic-package and is specialized to handle *autotools* based packages.
- It provides a default implementation of:
  - <pkg>_CONFIGURE_CMDS. Calls the ./configure script with appropriate environment variables and arguments.
  - <pkg>_BUILD_CMDS. Calls make.
  - <pkg>_INSTALL_TARGET_CMDS, <pkg>_INSTALL_STAGING_CMDS and <pkg>_INSTALL_CMDS. Call make install with the appropriate DESTDIR.
- A normal *autotools* based package therefore does not need to describe any action: only metadata about the package.

The autotools-package: steps



The `autotools-package` infrastructure: variables

- It provides additional variables that can be defined by the package:
  - `<pkg>_CONF_ENV` to pass additional values in the environment of the `./configure` script.
  - `<pkg>_CONF_OPTS` to pass additional options to the `./configure` script.
  - `<pkg>_INSTALL_OPTS`, `<pkg>_INSTALL_STAGING_OPTS` and `<pkg>_INSTALL_TARGET_OPTS` to adjust the *make* target and options used for the installation.
  - `<pkg>_AUTORECONF`. Defaults to `NO`, can be set to `YES` if regenerating `Makefile.in` files and `configure` script is needed. The infrastructure will automatically make sure *autoconf*, *automake*, *libtool* are built.
  - `<pkg>_GETTEXTIZE`. Defaults to `NO`, can be set to `YES` to *gettextize* the package. Only makes sense if `<pkg>_AUTORECONF = YES`.

Canonical autotools-package example

libyaml.mk

```
################################################################################
#
# libyaml
#
################################################################################

LIBYAML_VERSION = 0.2.5
LIBYAML_SOURCE = yaml-$(LIBYAML_VERSION).tar.gz
LIBYAML_SITE = http://pyyaml.org/download/libyaml
LIBYAML_INSTALL_STAGING = YES
LIBYAML_LICENSE = MIT
LIBYAML_LICENSE_FILES = License
LIBYAML_CPE_ID_VENDOR = pyyaml

$(eval $(autotools-package))
$(eval $(host-autotools-package))
```

More complicated autotools-package example

```
GNUPG2_VERSION = 2.2.25
GNUPG2_SOURCE = gnupg-$(GNUPG2_VERSION).tar.bz2
GNUPG2_SITE = https://gnupg.org/ftp/gcrypt/gnupg
GNUPG2_LICENSE = GPL-3.0+
GNUPG2_LICENSE_FILES = COPYING
GNUPG2_CPE_ID_VENDOR = gnupg
GNUPG2_CPE_ID_PRODUCT = gnupg
GNUPG2_DEPENDENCIES = zlib libgpg-error libgcrypt libassuan \
        libksba libnpth host-pkgconf \
        $(if $(BR2_PACKAGE_LIBICONV),libiconv)

ifeq ($(BR2_PACKAGE_BZIP2),y)
GNUPG2_CONF_OPTS += --enable-bzip2 --with-bzip2=$(STAGING_DIR)
GNUPG2_DEPENDENCIES += bzip2
else
GNUPG2_CONF_OPTS += --disable-bzip2
endif

ifeq ($(BR2_PACKAGE_GNUTLS),y)
GNUPG2_CONF_OPTS += --enable-gnutls
GNUPG2_DEPENDENCIES += gnutls
else
GNUPG2_CONF_OPTS += --disable-gnutls
endif
```

```
[...]

ifeq ($(BR2_PACKAGE_LIBUSB),y)
GNUPG2_CONF_ENV += CPPFLAGS="$(TARGET_CPPFLAGS)
                             -I$(STAGING_DIR)/usr/include/libusb-1.0"
GNUPG2_CONF_OPTS += --enable-ccid-driver
GNUPG2_DEPENDENCIES += libusb
else
GNUPG2_CONF_OPTS += --disable-ccid-driver
endif

ifeq ($(BR2_PACKAGE_READLINE),y)
GNUPG2_CONF_OPTS += --with-readline=$(STAGING_DIR)
GNUPG2_DEPENDENCIES += readline
else
GNUPG2_CONF_OPTS += --without-readline
endif

$(eval $(autotools-package))
```

Target vs. host packages

Host packages

- As explained earlier, most packages in Buildroot are cross-compiled for the target. They are called **target packages**.
- Some packages however may need to be built natively for the build machine, they are called **host packages**. They can be needed for a variety of reasons:
  - Needed as a tool to build other things for the target. Buildroot wants to limit the number of host utilities required to be installed on the build machine, and wants to ensure the proper version is used. So it builds some host utilities by itself.
  - Needed as a tool to interact, debug, reflash, generate images, or other activities around the build itself.
  - Version dependencies: building a Python interpreter for the target needs a Python interpreter of the same version on the host.

Target vs. host: package name and variable prefixes

- Each package infrastructure provides a `<foo>-package` macro and a `host-<foo>-package` macro.
- For a given package in `package/baz/baz.mk`, `<foo>-package` will create a package named `baz` and `host-<foo>-package` will create a package named `host-baz`.
- `<foo>-package` will use the variables prefixed with `BAZ_`
- `host-<foo>-package` will use the variables prefixed with `HOST_BAZ_`

Target vs. host: variable inheritance

- For many variables, when HOST_BAZ_<var> is not defined, the package infrastructure *inherits* from BAZ_<var> instead.
  - True for <PKG>_SOURCE, <PKG>_SITE, <PKG>_VERSION, <PKG>_LICENSE, <PKG>_LICENSE_FILES, etc.
  - Defining <PKG>_SITE is sufficient, defining HOST_<PKG>_SITE is not needed.
  - It is still possible to override the value specifically for the host variant, but this is rarely needed.
- But not for all variables, especially commands
  - E.g. HOST_<PKG>_BUILD_CMDS is not inherited from <PKG>_BUILD_CMDS

Example 1: a pure build utility

- *bison*, a general-purpose parser generator.
- Purely used as build dependency in packages
  - `FBSET_DEPENDENCIES = host-bison host-flex`
- No `Config.in.host`, not visible in `menuconfig`.

package/bison/bison.mk

```
BISON_VERSION = 3.7.1
BISON_SOURCE = bison-$(BISON_VERSION).tar.xz
BISON_SITE = $(BR2_GNU_MIRROR)/bison
BISON_LICENSE = GPL-3.0+
BISON_LICENSE_FILES = COPYING
BISON_CPE_ID_VENDOR = gnu
# parallel build issue in examples/c/reccalc/
BISON_MAKE = $(MAKE1)
HOST_BISON_DEPENDENCIES = host-m4
HOST_BISON_CONF_OPTS = --enable-relocatable
HOST_BISON_CONF_ENV = ac_cv_libtextstyle=no

$(eval $(host-autotools-package))
```

Example 2: filesystem manipulation tool

- `fatcat`, is designed to manipulate FAT filesystems, in order to explore, extract, repair, recover and forensic them.
- Not used as a build dependency of another package → visible in `menuconfig`.

package/fatcat/Config.in.host

```
config BR2_PACKAGE_HOST_FATCAT
	bool "host fatcat"
	help
	  Fatcat is designed to manipulate FAT filesystems, in order
	  to explore, extract, repair, recover and forensic them. It
	  currently supports FAT12, FAT16 and FAT32.

	  https://github.com/Gregwar/fatcat
```

package/fatcat/fatcat.mk

```
FATCAT_VERSION = 1.1.0
FATCAT_SITE = $(call github,Gregwar,fatcat,v$(FATCAT_VERSION))
FATCAT_LICENSE = MIT
FATCAT_LICENSE_FILES = LICENSE

$(eval $(host-cmake-package))
```

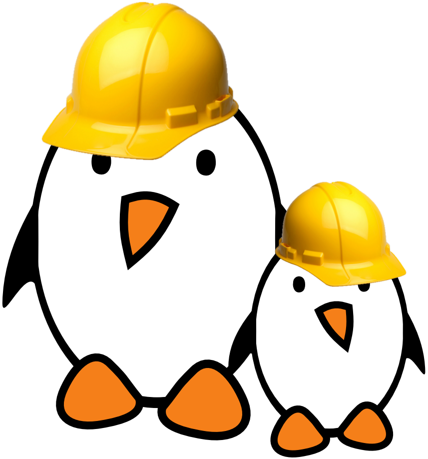
Example 3: target and host of the same package

package/e2tools/e2tools.mk

```
E2TOOLS_VERSION = 0.0.16.4
E2TOOLS_SITE = $(call github,ndim,e2tools,v$(E2TOOLS_VERSION))

# Source coming from GitHub, no configure included.
E2TOOLS_AUTORECONF = YES
E2TOOLS_LICENSE = GPL-2.0
E2TOOLS_LICENSE_FILES = COPYING
E2TOOLS_DEPENDENCIES = e2fsprogs
E2TOOLS_CONF_ENV = LIBS="-lpthread"
HOST_E2TOOLS_DEPENDENCIES = host-e2fsprogs
HOST_E2TOOLS_CONF_ENV = LIBS="-lpthread"

$(eval $(autotools-package))
$(eval $(host-autotools-package))
```



- Practical creation of several new packages in Buildroot, using the different package infrastructures.

Advanced package aspects

Licensing report

Licensing report: introduction

- A key aspect of embedded Linux systems is **license compliance**.
- Embedded Linux systems integrate together a number of open-source components, each distributed under its own license.
- The different open-source licenses may have **different requirements**, that must be met before the product using the embedded Linux system starts shipping.
- Buildroot helps in this license compliance process by offering the possibility of generating a number of **license-related information** from the list of selected packages.
- Generated using:

```
$ make legal-info
```

Licensing report: contents of `legal-info`

- `sources/` and `host-sources/`, all the source files that are redistributable (tarballs, patches, etc.)
- `manifest.csv` and `host-manifest.csv`, CSV files with the list of *target* and *host* packages, their version, license, etc.
- `licenses/` and `host-licenses/<pkg>/`, the full license text of all *target* and *host* packages, per package
- `buildroot.config`, the Buildroot `.config` file
- `legal-info.sha256` hashes of all *legal-info* files
- README

Including licensing information in packages

- `<pkg>_LICENSE`
  - Comma-separated **list of license(s)** under which the package is distributed.
  - Must use SPDX license codes, see https://spdx.org/licenses/
  - Can indicate which part is under which license (programs, tests, libraries, etc.)
- `<pkg>_LICENSE_FILES`
  - Space-separated **list of file paths** from the package source code containing the license text and copyright information
  - Paths relative to the package top-level source directory
- `<pkg>_REDISTRIBUTE`
  - Boolean indicating whether the package source code can be redistributed or not (part of the `legal-info` output)
  - Defaults to `YES`, can be overridden to `NO`
  - If `NO`, source code is not copied when generating the licensing report

Licensing information examples

linux.mk

```
LINUX_LICENSE = GPL-2.0
LINUX_LICENSE_FILES = COPYING
```

acl.mk

```
ACL_LICENSE = GPL-2.0+ (programs), LGPL-2.1+ (libraries)
ACL_LICENSE_FILES = doc/COPYING doc/COPYING.LGPL
```

owl-linux.mk

```
OWL_LINUX_LICENSE = PROPRIETARY
OWL_LINUX_LICENSE_FILES = LICENSE
OWL_LINUX_REDISTRIBUTE = NO
```

Security vulnerability tracking

Security vulnerability tracking

- Security has obviously become a key issue in embedded systems that are more and more commonly connected.
- Embedded Linux systems typically integrate 10-100+ open-source components → not easy to keep track of their potential security vulnerabilities
- Industry relies on *Common Vulnerability Exposure* (CVE) reports to document known security issues
- Buildroot is able to identify if packages are affected by known CVEs, by using the *National Vulnerability Database*
  - `make pkg-stats`
  - Produces `$(O)/pkg-stats.html`, `$(O)/pkg-stats.json`
- Note: this is limited to known CVEs. It does not guarantee the absence of security vulnerabilities.
- Only applies to open-source packages, not to your own custom code.

Example pkg-stats output

package/libao/libao.mk	0	**autotools** target	Yes	Yes	Yes	1.2.0	**1.2.0** found by distro	0	Link	CVE-2017-11548	cpe:2.3:a:xiph:libao:1.2.0:*:*:*:*:*:*:*
package/libcue/libcue.mk	0	**cmake** target	Yes	Yes	Yes	2.2.1	**2.2.1** found by distro	0	Link	N/A	no verified CPE identifier
package/libebur128/libebur128.mk	0	**cmake** target	Yes	Yes	Yes	1.2.4	**1.2.6** found by distro	0	Link	N/A	no verified CPE identifier
package/libffi/libffi.mk	7	**autotools** target + host	Yes	Yes	Yes	3.3	**3.3** found by distro	0	Link	N/A	cpe:2.3:a:libffi_project:libffi:3.3:rc0:*:*:*:*:*:*
package/libglib2/libglib2.mk	4	**meson** target + host	Yes	Yes	Yes	2.66.7	**2.68.1** found by distro	0	Link	CVE-2021-28153	cpe:2.3:a:gnome:glib:2.66.7:*:*:*:*:*:*:*
package/libid3tag/libid3tag.mk	0	**autotools** target	Yes	Yes	Yes	0.15.1b	**0.15.1b** found by distro	0	Link	N/A	no verified CPE identifier
package/liblo/liblo.mk	0	**autotools** target	Yes	Yes	Yes	0.31	**0.31** found by distro	0	Link	N/A	no verified CPE identifier
package/libmad/libmad.mk	2	**autotools** target	Yes	Yes	Yes	0.15.1b	**0.15.1b** found by distro	0	Link	CVE-2018-7263	no verified CPE identifier
package/libmodplug/libmodplug.mk	0	**autotools** target	Yes	Yes	Yes	0.8.9.0	**0.8.9.0** found by distro	0	Link	N/A	cpe:2.3:a:konstanty_bialkowski:libmodplug:0.8.9.0:*:*:*:*:*:*:* CPE identifier unknown in CPE database
package/libmpd/libmpd.mk	1	**autotools** target	Yes	Yes	Yes	11.8.17	**11.8.17** found by distro	0	invalid 502	N/A	no verified CPE identifier
package/libtool/libtool.mk	0	**autotools** target + host	Yes	Yes	Yes	2.4.6	**2.4.6** found by distro	0	Link	N/A	no verified CPE identifier

Packages affected by CVEs	5
Total number of CVEs affecting all packages	5
Packages with CPE ID	13
Packages without CPE ID	30

CPE: Common Platform Enumeration

- Concept of *Common Platform Enumeration*, which gives a unique identifier to a software release
  - E.g.: `cpe:2.3:a:xiph:libao:1.2.0:*:*:*:*:*:*:*`
- By default Buildroot uses:
  - `cpe:2.3:a:<pkg>_project:<pkg>:<pkg>_VERSION:*:*:*:*:*:*:*`
  - Not always correct!
- Can be modified using:
  - `<pkg>_CPE_ID_PREFIX`
  - `<pkg>_CPE_ID_VENDOR`
  - `<pkg>_CPE_ID_PRODUCT`
  - `<pkg>_CPE_ID_VERSION`
  - `<pkg>_CPE_ID_UPDATE`
- Concept of *CPE dictionary* provided by NVD, which contains all known CPEs.
  - `pkg-stats` checks if the CPE of each package is known in the *CPE dictionary*

NVD CVE-2020-35492 example

https://nvd.nist.gov/vuln/detail/CVE-2020-35492

Known Affected Software Configurations Switch to CPE 2.2

Configuration 1 (hide)

cpe:2.3:a:cairographics:cairo:*:*:*:*:*:*:*:*	**Up to (excluding)**
Hide Matching CPE(s) ▲	**1.17.4**

- *cpe:2.3:a:cairographics:cairo:-:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.0.0:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.0.2:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.0.4:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.2.0:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.2.2:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.2.4:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.2.6:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.4.0:*:*:*:*:*:*:**
- *cpe:2.3:a:cairographics:cairo:1.4.2:*:*:*:*:*:*:**

Showing 10 of 50 matching CPE(s) for the range. View All CPEs here

CPE information in packages

package/bash/bash.mk

```
BASH_CPE_ID_VENDOR = gnu
```

package/audit/audit.mk

```
AUDIT_CPE_ID_VENDOR = linux_audit_project
AUDIT_CPE_ID_PRODUCT = linux_audit
```

linux/linux.mk

```
LINUX_CPE_ID_VENDOR = linux
LINUX_CPE_ID_PRODUCT = linux_kernel
LINUX_CPE_ID_PREFIX = cpe:2.3:o
```

package/libffi/libffi.mk

```
LIBFFI_CPE_ID_VERSION = 3.3
LIBFFI_CPE_ID_UPDATE = rc0
```

<pkg>_IGNORE_CVES variable

- There are cases where a CVE reported by the *pkg-stats* tool in fact is not relevant:
  - The security fix has been backported into Buildroot
  - The vulnerability does not affect Buildroot due to how the package is configured or used
- The `<pkg>_IGNORE_CVES` variable allows a package to tell *pkg-stats* to ignore a particular CVE

package/bind/bind.mk

```
# Only applies to RHEL6.x with DNSSEC validation on
BIND_IGNORE_CVES = CVE-2017-3139
```

package/avahi/avahi.mk

```
# 0001-Fix-NULL-pointer-crashes-from-175.patch
AVAHI_IGNORE_CVES += CVE-2021-36217
```

Patching packages

Patching packages: why?

- In some situations, it might be needed to patch the source code of certain packages built by Buildroot.
- Useful to:
  - Fix cross-compilation issues
  - Backport bug or security fixes from upstream
  - Integrate new features or fixes not available upstream, or that are too specific to the product being made
- Patches are automatically applied by Buildroot, during the *patch* step, i.e. after extracting the package, but before configuring it.
- Buildroot already comes with a number of patches for various packages, but you may need to add more for your own packages, or to existing packages.

Patch application ordering

- Overall the patches are applied in this order:
  1. Patches mentioned in the `<pkg>_PATCH` variable of the package `.mk` file. They are automatically downloaded before being applied.
  2. Patches present in the package directory `package/<pkg>/*.patch`
  3. Patches present in the *global patch directories*
- In each case, they are applied:
  - In the order specified in a `series` file, if available
  - Otherwise, in alphabetic ordering

Patch conventions

- There are a few conventions and best practices that the Buildroot project encourages to use when managing patches
- Their name should start with a sequence number that indicates the ordering in which they should be applied.

ls package/nginx/*.patch

```
0001-auto-type-sizeof-rework-autotest-to-be-cross-compila.patch
0002-auto-feature-add-mechanism-allowing-to-force-feature.patch
0003-auto-set-ngx_feature_run_force_result-for-each-featu.patch
0004-auto-lib-libxslt-conf-allow-to-override-ngx_feature_.patch
0005-auto-unix-make-sys_nerr-guessing-cross-friendly.patch
```

- Each patch should contain a description of what the patch does, and if possible its upstream status.
- Each patch should contain a `Signed-off-by` that identifies the author of the patch.
- Patches should be generated using `git format-patch` when possible.

Patch example

```
From 81289d1d1adaf5a767a4b4d1309c286468cfd37f Mon Sep 17 00:00:00 2001
From: Samuel Martin <s.martin49@gmail.com>
Date: Thu, 24 Apr 2014 23:27:32 +0200
Subject: [PATCH] auto/type/sizeof: rework autotest to be cross-compilation
 friendly

Rework the sizeof test to do the checks at compile time instead of at
runtime. This way, it does not break when cross-compiling for a
different CPU architecture.

Signed-off-by: Samuel Martin <s.martin49@gmail.com>
---
 auto/types/sizeof | 42 ++++++++++++++++++++++++++++--------------
 1 file changed, 28 insertions(+), 14 deletions(-)

diff --git a/auto/types/sizeof b/auto/types/sizeof
index 9215a54..c2c3ede 100644
--- a/auto/types/sizeof
+++ b/auto/types/sizeof
@@ -14,7 +14,7 @@ END

 ngx_size=

-cat << END > $NGX_AUTOTEST.c
+cat << _EOF > $NGX_AUTOTEST.c
[...]
```

Global patch directories

- You can include patches for the different packages in their package directory, `package/<pkg>/`.
- However, doing this involves changing the Buildroot sources themselves, which may not be appropriate for some highly specific patches.
- The *global patch directories* mechanism allows to specify additional locations where Buildroot will look for patches to apply on packages.
- `BR2_GLOBAL_PATCH_DIR` specifies a space-separated list of directories containing patches.
- These directories must contain sub-directories named after the packages, themselves containing the patches to be applied.

Global patch directory example

Patching *strace*

```
$ ls package/strace/*.patch
0001-linux-aarch64-add-missing-header.patch

$ find ~/patches/
~/patches/
~/patches/strace/
~/patches/strace/0001-Demo-strace-change.patch

$ grep ^BR2_GLOBAL_PATCH_DIR .config
BR2_GLOBAL_PATCH_DIR="$(HOME)/patches"

$ make strace
[...]
>>> strace 4.10 Patching

Applying 0001-linux-aarch64-add-missing-header.patch using patch:
patching file linux/aarch64/arch_regs.h

Applying 0001-Demo-strace-change.patch using patch:
patching file README
[...]
```

Generating patches

- To generate the patches against a given package source code, there are typically two possibilities.
- Use the upstream version control system, often *Git*
- Use a tool called `quilt`
  - Useful when there is no version control system provided by the upstream project
  - https://savannah.nongnu.org/projects/quilt

Generating patches: with Git

Needs to be done outside of Buildroot: you cannot use the Buildroot package build directory.

1. Clone the upstream Git repository
   `git clone https://...`
2. Create a branch starting on the tag marking the stable release of the software as packaged in Buildroot
   `git checkout -b buildroot-changes v3.2`
3. Import existing Buildroot patches (if any)
   `git am /path/to/buildroot/package/<foo>/*.patch`
4. Make your changes and commit them
   `git commit -s -m ``this is a change''`
5. Generate the patches
   `git format-patch v3.2`

Generating patches: with Quilt

1. Extract the package source code:
   `tar xf /path/to/dl/<foo>-<version>.tar.gz`
2. Inside the package source code, create a directory for patches
   `mkdir patches`
3. Import existing Buildroot patches
   `quilt import /path/to/buildroot/package/<foo>/*.patch`
4. Apply existing Buildroot patches
   `quilt push -a`
5. Create a new patch
   `quilt new 0001-fix-header-inclusion.patch`
6. Edit a file
   `quilt edit main.c`
7. Refresh the patch
   `quilt refresh`

User, permission and device tables

Package-specific users

- The default skeleton in `system/skeleton/` has a number of default users/groups.
- Packages can define their own custom users/groups using the `<pkg>_USERS` variable:

```
define <pkg>_USERS
	username uid group gid password home shell groups comment
endef
```

- Examples:

```
define AVAHI_USERS
	avahi -1 avahi -1 * - - -
endef
```

```
define MYSQL_USERS
	mysql -1 nogroup -1 * /var/mysql - - MySQL daemon
endef
```

File permissions and ownership

- By default, before creating the root filesystem images, Buildroot changes the ownership of all files to `0:0`, i.e. `root:root`
- Permissions are preserved as is, but since the build is executed as non-root, it is not possible to install setuid applications.
- A default set of permissions for certain files or directories is defined in `system/device_table.txt`.
- The `<pkg>_PERMISSIONS` variable allows packages to define special ownership and permissions for files and directories:

```
define <pkg>_PERMISSIONS
name type mode uid gid major minor start inc count
endef
```

- The `major`, `minor`, `start`, `inc` and `count` fields are not used.

File permissions and ownership: examples

- `sudo` needs to be installed *setuid root*:

```
define SUDO_PERMISSIONS
        /usr/bin/sudo f 4755 0 0 - - - - -
endef
```

- `/var/lib/nginx` needs to be owned by `www-data`, which has UID/GID `33` defined in the skeleton:

```
define NGINX_PERMISSIONS
        /var/lib/nginx d 755 33 33 - - - - -
endef
```

Devices

- Defining devices only applies when the chosen `/dev` management strategy is *Static using a device table*. In other cases, *device files* are created dynamically.
- A default set of *device files* is described in `system/device_table_dev.txt` and created by Buildroot in the root filesystem images.
- When packages need some additional custom devices, they can use the `<pkg>_DEVICES` variable:

```
define <pkg>_DEVICES
name type mode uid gid major minor start inc count
endef
```

- Becoming less useful, since most people are using a dynamic `/dev` nowadays.

Devices: example

xenomai.mk

```
define XENOMAI_DEVICES
/dev/rtheap   c  666  0  0  10   254  0  0  -
/dev/rtscope  c  666  0  0  10   253  0  0  -
/dev/rtp      c  666  0  0  150  0    0  1  32
endef
```

Init scripts and systemd unit files

Init scripts, systemd unit files

- Buildroot supports several main init systems: *sysvinit*, *BusyBox*, *systemd*, *OpenRC*
- When packages want to install a program to be started at boot time, they need to install a startup script (*sysvinit*/*BusyBox*), a *systemd service* file, etc.
- They can do so using the following variables, which contain a list of shell commands.
  - `<pkg>_INSTALL_INIT_SYSV`
  - `<pkg>_INSTALL_INIT_SYSTEMD`
  - `<pkg>_INSTALL_INIT_OPENRC`
- Buildroot will execute the appropriate `<pkg>_INSTALL_INIT_xyz` commands of all enabled packages depending on the selected init system.

Init scripts, systemd unit files: example

bind.mk

```
define BIND_INSTALL_INIT_SYSV
        $(INSTALL) -m 0755 -D package/bind/S81named \
                $(TARGET_DIR)/etc/init.d/S81named
endef

define BIND_INSTALL_INIT_SYSTEMD
        $(INSTALL) -D -m 644 package/bind/named.service \
                $(TARGET_DIR)/usr/lib/systemd/system/named.service
endef
```

Config scripts

Config scripts: introduction

- Libraries not using `pkg-config` often install a **small shell script** that allows applications to query the compiler and linker flags to use the library.
- Examples: `curl-config`, `freetype-config`, etc.
- Such scripts will:
  - generally return results that are **not appropriate for cross-compilation**
  - be used by other cross-compiled Buildroot packages that use those libraries
- By listing such scripts in the `<pkg>_CONFIG_SCRIPTS` variable, Buildroot will **adapt the prefix, header and library paths** to make them suitable for cross-compilation.
- Paths in `<pkg>_CONFIG_SCRIPTS` are relative to `$(STAGING_DIR)/usr/bin`.

Config scripts: examples

libpng.mk

```
LIBPNG_CONFIG_SCRIPTS = \
        libpng$(LIBPNG_SERIES)-config libpng-config
```

imagemagick.mk

```
IMAGEMAGICK_CONFIG_SCRIPTS = \
        $(addsuffix -config,Magick MagickCore MagickWand Wand)

ifeq ($(BR2_INSTALL_LIBSTDCPP)$(BR2_USE_WCHAR),yy)
IMAGEMAGICK_CONFIG_SCRIPTS += Magick++-config
endif
```

Config scripts: effect

Without <pkg>_CONFIG_SCRIPTS

```
$ ./output/staging/usr/bin/libpng-config --cflags --ldflags
-I/usr/include/libpng16
-L/usr/lib -lpng16
```

With <pkg>_CONFIG_SCRIPTS

```
$ ./output/staging/usr/bin/libpng-config --cflags --ldflags
-I.../buildroot/output/host/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/include/libpng16
-L.../buildroot/output/host/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/lib -lpng16
```

Hooks

Hooks: principle (1)

- Buildroot *package infrastructure* often implement a default behavior for certain steps:
  - `generic-package` implements for all packages the download, extract and patch steps
  - Other infrastructures such as `autotools-package` or `cmake-package` also implement the configure, build and installations steps
- In some situations, the package may want to do **additional actions** before or after one of these steps.
- The **hook** mechanism allows packages to add such custom actions.

Hooks: principle (2)

- There are **pre** and **post** hooks available for all steps of the package compilation process:
  - download, extract, rsync, patch, configure, build, install, install staging, install target, install images, legal info
  - `<pkg>_(PRE|POST)_<step>_HOOKS`
  - Example: `CMAKE_POST_INSTALL_TARGET_HOOKS`, `CVS_POST_PATCH_HOOKS`, `BINUTILS_PRE_PATCH_HOOKS`
- Hook variables contain a list of make macros to call at the appropriate time.
  - Use `+=` to register an additional hook to a hook point
- Those make macros contain a list of commands to execute.

Hooks: examples

bind.mk: remove unneeded binaries

```
define BIND_TARGET_REMOVE_TOOLS
        rm -rf $(addprefix $(TARGET_DIR)/usr/bin/, $(BIND_TARGET_TOOLS_BIN))
endef

BIND_POST_INSTALL_TARGET_HOOKS += BIND_TARGET_REMOVE_TOOLS
```

vsftpd.mk: adjust configuration

```
define VSFTPD_ENABLE_SSL
        $(SED) 's/.*VSF_BUILD_SSL/#define VSF_BUILD_SSL/' \
                $(@D)/builddefs.h
endef

ifeq ($(BR2_PACKAGE_OPENSSL),y)
VSFTPD_DEPENDENCIES += openssl host-pkgconf
VSFTPD_LIBS += `$(PKG_CONFIG_HOST_BINARY) --libs libssl libcrypto`
VSFTPD_POST_CONFIGURE_HOOKS += VSFTPD_ENABLE_SSL
endif
```

Overriding commands

Overriding commands: principle

- In other situations, a package may want to completely **override** the default implementation of a step provided by a package infrastructure.
- A package infrastructure will in fact only implement a given step **if not already defined by a package**.
- So defining `<pkg>_EXTRACT_CMDS` or `<pkg>_BUILD_CMDS` in your package `.mk` file will override the package infrastructure implementation (if any).

Overriding commands: examples

jquery: source code is only one file

```
JQUERY_SITE = http://code.jquery.com
JQUERY_SOURCE = jquery-$(JQUERY_VERSION).min.js

define JQUERY_EXTRACT_CMDS
	cp $(DL_DIR)/$(JQUERY_SOURCE) $(@D)
endef
```

tftpd: install only what's needed

```
define TFTPD_INSTALL_TARGET_CMDS
	$(INSTALL) -D $(@D)/tftp/tftp $(TARGET_DIR)/usr/bin/tftp
	$(INSTALL) -D $(@D)/tftpd/tftpd $(TARGET_DIR)/usr/sbin/tftpd
endef

$(eval $(autotools-package))
```

Legacy handling

Legacy handling: `Config.in.legacy`

- When a `Config.in` option is removed, the corresponding value in the `.config` is silently removed.
- Due to this, when users upgrade Buildroot, they generally don't know that an option they were using has been removed.
- Buildroot therefore adds the removed config option to `Config.in.legacy` with a description of what has happened.
- If any of these legacy options is enabled then Buildroot refuses to build.

DEVELOPERS file

DEVELOPERS file: principle

- A top-level `DEVELOPERS` file lists Buildroot developers and contributors interested in specific packages, board *defconfigs* or architectures.
- Used by:
  - The `utils/get-developers` script to identify to whom a patch on an existing package should be sent
  - The Buildroot *autobuilder* infrastructure to notify build failures to the appropriate package or architecture developers
- Important to add yourself in `DEVELOPERS` if you contribute a new package/board to Buildroot.

DEVELOPERS file: extract

```
N:      Thomas Petazzoni <thomas.petazzoni@bootlin.com>
F:      arch/Config.in.arm
F:      boot/boot-wrapper-aarch64/
F:      boot/grub2/
F:      package/android-tools/
F:      package/cmake/
F:      package/cramfs/
[...]
F:      toolchain/

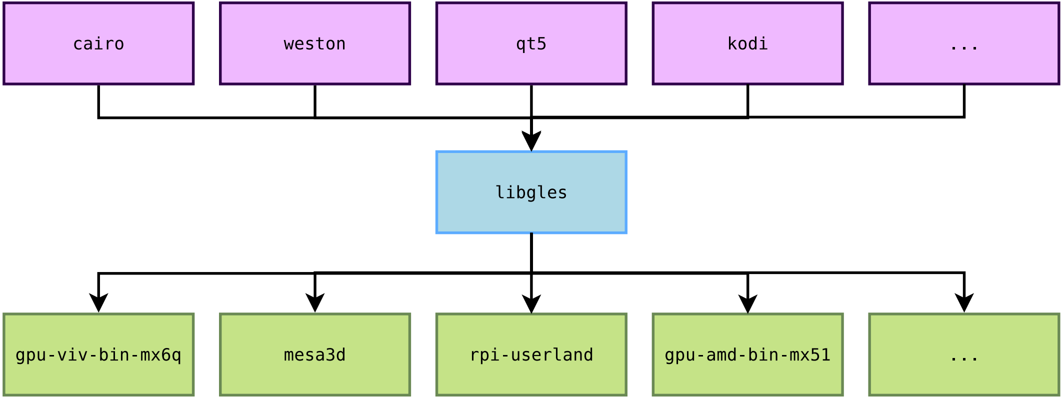
N:      Waldemar Brodkorb <wbx@openadk.org>
F:      arch/Config.in.bfin
F:      arch/Config.in.m68k
F:      arch/Config.in.or1k
F:      arch/Config.in.sparc
F:      package/glibc/
```

Virtual packages

Virtual packages

- There are situations where different packages provide an implementation of the same interface
- The most useful example is OpenGL
  - OpenGL is an API
  - Each HW vendor typically provides its own OpenGL implementation, each packaged as separate Buildroot packages
- Packages using the OpenGL interface do not want to know which implementation they are using: they are simply using the OpenGL API
- The mechanism of *virtual packages* in Buildroot allows to solve this situation.
  - `libgles` is a virtual package offering the OpenGL ES API
  - Ten packages are *providers* of the OpenGL ES API: `gpu-amd-bin-mx51`, `imx-gpu-viv`, `gcnano-binaries`, `mali-t76x`, `mesa3d`, `nvidia-driver`, `rpi-userland`, `sunxi-mali-mainline`, `ti-gfx`, `ti-sgx-um`

Virtual packages



Virtual package definition: Config.in

libgles/Config.in

```
config BR2_PACKAGE_HAS_LIBGLES
	bool

config BR2_PACKAGE_PROVIDES_LIBGLES
	depends on BR2_PACKAGE_HAS_LIBGLES
	string
```

- `BR2_PACKAGE_HAS_LIBGLES` is a hidden boolean
  - Packages needing OpenGL ES will `depends on` it.
  - Packages providing OpenGL ES will `select` it.
- `BR2_PACKAGE_PROVIDES_LIBGLES` is a hidden string
  - Packages providing OpenGL ES will define their name as the variable value
  - The `libgles` package will have a build dependency on this provider package.

Virtual package definition: .mk

libgles/libgles.mk

```
$(eval $(virtual-package))
```

- Nothing to do: the `virtual-package` infrastructure takes care of everything, using the `BR2_PACKAGE_HAS_<name>` and `BR2_PACKAGE_PROVIDES_<name>` options.

Virtual package provider

sunxi-mali-mainline/Config.in

```
config BR2_PACKAGE_SUNXI_MALI_MAINLINE
        bool "sunxi-mali-mainline"
        select BR2_PACKAGE_HAS_LIBEGL
        select BR2_PACKAGE_HAS_LIBGLES

config BR2_PACKAGE_PROVIDES_LIBGLES
        default "sunxi-mali-mainline"
```

sunxi-mali-mainline/sunxi-mali-mainline.mk

```
[...]
SUNXI_MALI_MAINLINE_PROVIDES = libegl libgles
[...]
```

- The variable `<pkg>_PROVIDES` is only used to detect if two providers for the same virtual package are enabled.

Virtual package user

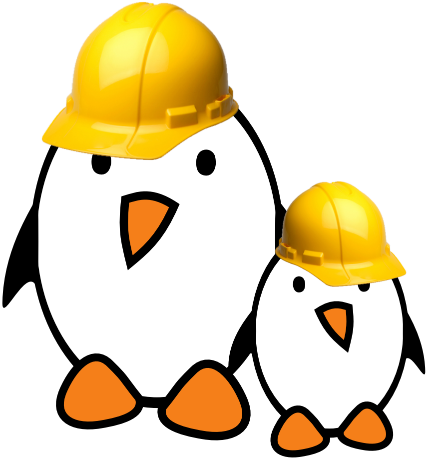
qt5/qt5base/Config.in

```
config BR2_PACKAGE_QT5BASE_OPENGL_ES2
        bool "OpenGL ES 2.0+"
        depends on BR2_PACKAGE_HAS_LIBGLES
        help
          Use OpenGL ES 2.0 and later versions.
```

qt5/qt5base/qt5base.mk

```
ifeq ($(BR2_PACKAGE_QT5BASE_OPENGL_DESKTOP),y)
QT5BASE_CONFIGURE_OPTS += -opengl desktop
QT5BASE_DEPENDENCIES   += libgl
else ifeq ($(BR2_PACKAGE_QT5BASE_OPENGL_ES2),y)
QT5BASE_CONFIGURE_OPTS += -opengl es2
QT5BASE_DEPENDENCIES   += libgles
else
QT5BASE_CONFIGURE_OPTS += -no-opengl
endif
```

Practical lab - Advanced packages



- 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

Analyzing the build

embedded Linux and kernel engineering

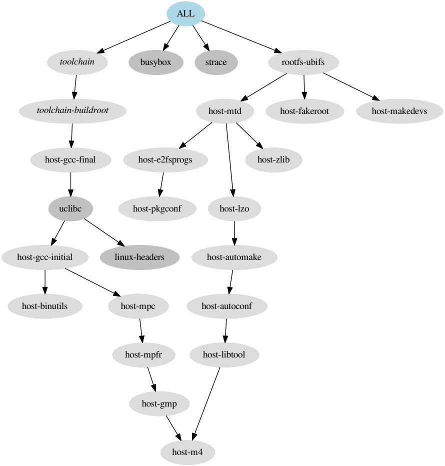
Analyzing the build: available tools

- Buildroot provides several useful tools to analyze the build:
  - The **licensing report**, covered in a previous section, which allows to analyze the list of packages and their licenses.
  - The **dependency graphing** tools
  - The **build time graphing** tools
  - The **filesystem size** tools

Dependency graphing

- Exploring the dependencies between packages is useful to understand
  - why a particular package is being brought into the build
  - if the build size and duration can be reduced
- `make graph-depends` to generate a full dependency graph, which can be huge!
- `make <pkg>-graph-depends` to generate the dependency graph of a given package
- The graph is done according to the current Buildroot configuration.
- Resulting graphs in `$(O)/graphs/`

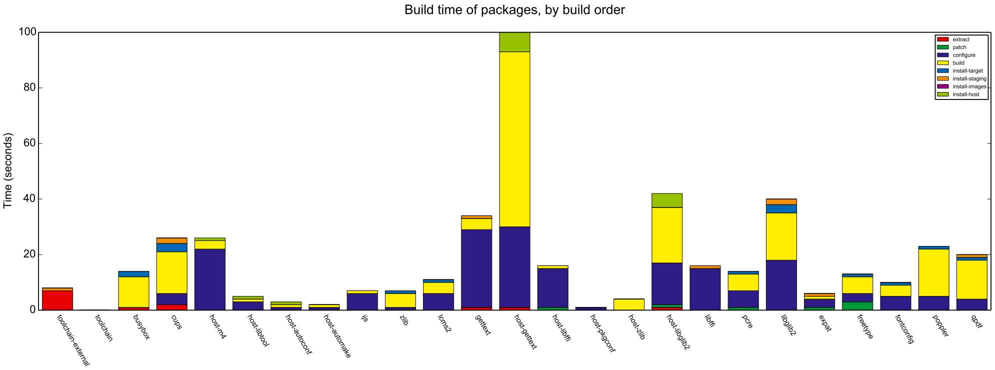
Dependency graph example



Build time graphing

- When the generated embedded Linux system grows bigger and bigger, the build time also increases.
- It is sometimes useful to analyze this build time, and see if certain packages are particularly problematic.
- Buildroot collects build duration data in the file `$(O)/build/build-time.log`
- `make graph-build` generates several graphs in `$(O)/graphs/`:
  - `build.hist-build.pdf`, build time in build order
  - `build.hist-duration.pdf`, build time by duration
  - `build.hist-name.pdf`, build time by package name
  - `build.pie-packages.pdf`, pie chart of the per-package build time
  - `build.pie-steps.pdf`, pie chart of the per-step build time
- Note: only works properly after a complete clean rebuild.

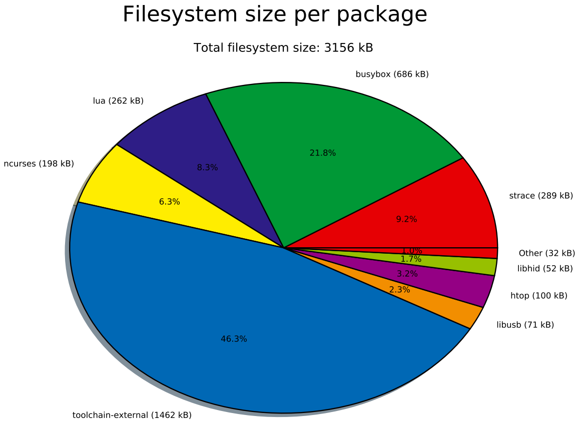
Build time graphing: example



Filesystem size graphing

- In many embedded systems, storage resources are limited.
- For this reason, it is useful to be able to analyze the size of your root filesystem, and see which packages are consuming the biggest amount of space.
- Allows to focus the size optimizations on the relevant packages.
- Buildroot collects data about the size installed by each package.
- `make graph-size` produces:
  - `file-size-stats.csv`, CSV with the raw data of the per-file size
  - `package-size-stats.csv`, CSV with the raw data of the per-package size
  - `graph-size.pdf`, pie chart of the per-package size consumption

Filesystem size graphing: example



Advanced topics

BR2_EXTERNAL: principle

- Storing your custom packages, custom configuration files and custom *defconfigs* inside the Buildroot tree may not be the most practical solution
  - Doesn't cleanly separate open-source parts from proprietary parts
  - Makes it harder to upgrade Buildroot
- The `BR2_EXTERNAL` mechanism allows to store your own package recipes, *defconfigs* and other artefacts **outside** of the Buildroot source tree.
- It is possible to use several `BR2_EXTERNAL` trees, to further separate various aspects of your project.
- Note: can only be used to add new packages, not to override existing Buildroot packages

BR2_EXTERNAL: example organization

- `project/`
  - `buildroot/`
    - The Buildroot source code, cloned from Git, or extracted from a release tarball.
  - `external1/`
  - `external2/`
    - Two external trees
  - `output-build1/`
  - `output-build2/`
    - Several *output* directories, to build various configurations
  - `custom-app/`
  - `custom-lib/`
    - The source code of your custom applications and libraries.

Using BR2_EXTERNAL

- Specify, as a colon-separated list, the *external* directories in `BR2_EXTERNAL`
- Not a configuration option, only an **environment variable** to be passed on the command line

```
make BR2_EXTERNAL=/path/to/external1:/path/to/external2
```

- **Automatically saved** in the hidden `.br2-external.mk` file in the output directory
  - no need to pass `BR2_EXTERNAL` at every make invocation
  - can be changed at any time by passing a new value, and removed by passing an empty value
- Can be either an **absolute** or a **relative** path, but if relative, important to remember that it's relative to the Buildroot source directory

BR2_EXTERNAL: important files

- Each *external* directory must contain:
  - `external.desc`, which provides a name and description
  - `Config.in`, configuration options that will be included in *menuconfig*
  - `external.mk`, will be included in the make logic
- If `configs` exists, it will be used when listing all *defconfigs*

BR2_EXTERNAL: recommended structure

```
+-- board/
|   +-- <company>/
|       +-- <boardname>/
|           +-- linux.config
|           +-- busybox.config
|           +-- <other configuration files>
|           +-- post_build.sh
|           +-- post_image.sh
|           +-- rootfs_overlay/
|           |   +-- etc/
|           |   +-- <some file>
|           +-- patches/
|               +-- libbar/
|                   +-- <some patches>
|
+-- configs/
|   +-- <boardname>_defconfig
|
```

```
+-- package/
|   +-- <company>/
|       +-- package1/
|       |    +-- Config.in
|       |    +-- package1.mk
|       +-- package2/
|           +-- Config.in
|           +-- package2.mk
|
+-- Config.in
+-- external.mk
+-- external.desc
```

BR2_EXTERNAL: external.desc

- File giving metadata about the *external tree*
- Mandatory name field, using characters in the set [A-Za-z0-9_]. Will be used to define BR2_EXTERNAL_<NAME>_PATH available in Config.in and .mk files, pointing to the external tree directory.
- Optional desc field, giving a free-form description of the external tree. Should be reasonably short.
- Example

```
name: FOOBAR
desc: Foobar Company
```

BR2_EXTERNAL: main Config.in

- Custom configuration options
- Configuration options for the external packages
- The $BR2_EXTERNAL_<NAME>_PATH variable is available, where NAME is defined in external.desc

Example Config.in

```
source "$BR2_EXTERNAL_<NAME>_PATH/package/package1/Config.in"
source "$BR2_EXTERNAL_<NAME>_PATH/package/package2/Config.in"
```

BR2_EXTERNAL: external.mk

- Can include custom *make* logic
- Generally only used to include the package `.mk` files

Example `external.mk`

```
include $(sort $(wildcard $(BR2_EXTERNAL_<NAME>_PATH)/package/*/*.mk))
```

Use BR2_EXTERNAL in your configuration

- In your Buildroot configuration, don't use absolute paths for the *rootfs overlay*, the *post-build scripts*, *global patch directories*, etc.
- If they are located in an external tree, you can use `$(BR2_EXTERNAL_<NAME>_PATH)` in your Buildroot configuration options.
- With the recommended structure shown before, a Buildroot configuration would look like:

```
BR2_GLOBAL_PATCH_DIR="$(BR2_EXTERNAL_<NAME>_PATH)/board/<company>/<boardname>/patches/"
...
BR2_ROOTFS_OVERLAY="$(BR2_EXTERNAL_<NAME>_PATH)/board/<company>/<boardname>/rootfs_overlay/"
...
BR2_ROOTFS_POST_BUILD_SCRIPT="$(BR2_EXTERNAL_<NAME>_PATH)/board/<company>/<boardname>/post_build.sh"
BR2_ROOTFS_POST_IMAGE_SCRIPT="$(BR2_EXTERNAL_<NAME>_PATH)/board/<company>/<boardname>/post_image.sh"
...
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="$(BR2_EXTERNAL_<NAME>_PATH)/board/<company>/<boardname>/linux.config"
```

Examples of BR2_EXTERNAL trees

- There are a number of publicly available BR2_EXTERNAL trees, especially from hardware vendors:
  - `buildroot-external-st`, maintained by Bootlin in partnership with ST, containing example configurations for the STM32MP1 platforms.
    https://github.com/bootlin/buildroot-external-st
  - `buildroot-external-microchip`, containing example configurations, additional packages and demo applications for Microchip ARM platforms.
    https://github.com/linux4sam/buildroot-external-microchip
  - `buildroot-external-boundary`, containing example configurations for Boundary Devices boards, mainly based on NXP i.MX processors.
    https://github.com/boundarydevices/buildroot-external-boundary

Package-specific targets: basics

- Internally, each package is implemented through a number of package-specific *make targets*
  - They can sometimes be useful to call directly, in certain situations.
- The targets used in the normal build flow of a package are:
  - `<pkg>`, fully build and install the package
  - `<pkg>-source`, just download the source code
  - `<pkg>-extract`, download and extract
  - `<pkg>-patch`, download, extract and patch
  - `<pkg>-configure`, download, extract, patch and configure
  - `<pkg>-build`, download, extract, patch, configure and build
  - `<pkg>-install-staging`, download, extract, patch, configure and do the staging installation (target packages only)
  - `<pkg>-install-target`, download, extract, patch, configure and do the target installation (target packages only)
  - `<pkg>-install`, download, extract, patch, configure and install

Package-specific targets: example (1)

```
$ make strace
>>> strace 4.10 Extracting
>>> strace 4.10 Patching
>>> strace 4.10 Updating config.sub and config.guess
>>> strace 4.10 Patching libtool
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ make strace-build
... nothing ...
$ make ltrace-patch
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Extracting
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Patching
$ make ltrace
>>> argp-standalone 1.3 Extracting
>>> argp-standalone 1.3 Patching
>>> argp-standalone 1.3 Updating config.sub and config.guess
>>> argp-standalone 1.3 Patching libtool
[...]
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Configuring
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Autoreconfiguring
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Patching libtool
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Building
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Installing to target
```

Package-specific targets: advanced

- Additional useful targets
  - `make <pkg>-show-depends`, show the package dependencies
  - `make <pkg>-graph-depends`, generates a dependency graph
  - `make <pkg>-dirclean`, completely remove the package source code directory. The next `make` invocation will fully rebuild this package.
  - `make <pkg>-reinstall`, force to re-execute the installation step of the package
  - `make <pkg>-rebuild`, force to re-execute the build and installation steps of the package
  - `make <pkg>-reconfigure`, force to re-execute the configure, build and installation steps of the package.

Package-specific targets: example (2)

```
$ make strace
>>> strace 4.10 Extracting
>>> strace 4.10 Patching
>>> strace 4.10 Updating config.sub and config.guess
>>> strace 4.10 Patching libtool
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ ls output/build/
strace-4.10 [...]
$ make strace-dirclean
rm -Rf /home/thomas/projets/buildroot/output/build/strace-4.10
$ ls output/build/
[... no strace-4.10 directory ...]
```

Package-specific targets: example (3)

```
$ make strace
>>> strace 4.10 Extracting
>>> strace 4.10 Patching
>>> strace 4.10 Updating config.sub and config.guess
>>> strace 4.10 Patching libtool
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ make strace-rebuild
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ make strace-reconfigure
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
```

make show-info

- `make show-info` outputs JSON text that describes the current configuration: enabled packages, in which version, their license, tarball, dependencies, etc.
- Can be useful for post-processing, build analysis, license compliance, etc.

```
$ make show-info | jq .
{
  "busybox": {
    "type": "target",
    "virtual": false,
    "version": "1.31.1",
    "licenses": "GPL-2.0",
    "dl_dir": "busybox",
    "install_target": true,
    "install_staging": false,
    "install_images": false,
    "downloads": [
      {
        "source": "busybox-1.31.1.tar.bz2",
        "uris": [
          "http+http://www.busybox.net/downloads",
          "http|urlencode+http://sources.buildroot.net/busybox",
        ]
      }
    ],
    "dependencies": [
      "host-skeleton",
      "host-tar",
      "skeleton",
      "toolchain"
    ],
    "reverse_dependencies": []
  },
```

Understanding rebuilds (1)

- Doing a **full rebuild** is achieved using:

```
$ make clean all
```

  - It will completely remove all build artefacts and restart the build from scratch
- Buildroot **does not try to be smart**
  - once the system has been built, if a configuration change is made, the next `make` will **not apply all the changes** made to the configuration.
  - being smart is very, very complicated if you want to do it in a reliable way.

Understanding rebuilds (2)

- When a package has been built by Buildroot, Buildroot keeps a **hidden file** telling that the package has been built.
  - Buildroot will therefore *never* rebuild that package, unless a **full rebuild is done**, or this specific package is **explicitly rebuilt**.
  - Buildroot does not *recurse* into each package at each `make` invocation, it would be too time-consuming. So if you change one source file in a package, Buildroot does not know it.
- When `make` is invoked, Buildroot **will always**:
  - Build the packages that have not been built in a previous build and install them to the target
  - Cleanup the target root filesystem from useless files
  - Run *post-build* scripts, copy *rootfs overlays*
  - Generate the root filesystem images
  - Run *post-image* scripts

Understanding rebuilds: scenarios (1)

- If you enable a new package in the configuration, and run `make`
  - Buildroot will build it and install it
  - However, other packages that may benefit from this package will not be rebuilt automatically
- If you remove a package from the configuration, and run `make`
  - Nothing happens. The files installed by this package are not removed from the target filesystem.
  - Buildroot does not track which files are installed by which package
  - Need to do a full rebuild to get the new result. Advice: do it only when really needed.
- If you change the sub-options of a package that has already been built, and run `make`
  - Nothing happens.
  - You can force Buildroot to rebuild this package using `make <pkg>-reconfigure` or `make <pkg>-rebuild`.

Understanding rebuilds: scenarios (2)

- If you make a change to a *post-build* script, a *rootfs overlay* or a *post-image* script, and run `make`
  - This is sufficient, since these parts are re-executed at every `make` invocation.
- If you change a fundamental system configuration option: architecture, type of toolchain or toolchain configuration, init system, etc.
  - You **must do a full rebuild**
- If you change some source code in `output/build/<foo>-<version>/` and issue `make`
  - The package will not be rebuilt automatically: Buildroot has a *hidden file* saying that the package was already built.
  - Use `make <pkg>-reconfigure` or `make <pkg>-rebuild`
  - And remember that doing changes in `output/build/<foo>-<version>/` can only be temporary: this directory is removed during a `make clean`.

Tips for building faster

- Build time is often an issue, so here are some tips to help
  - Use fast hardware: lots of RAM, and SSD
  - Do not use virtual machines
  - You can enable the ccache *compiler cache* using BR2_CCACHE
  - Use external toolchains instead of internal toolchains
  - Learn about rebuilding only the few packages you actually care about
  - Build everything locally, do not use NFS for building
  - Remember that you can do several independent builds in parallel in different output directories

Support for top-level parallel build (1)

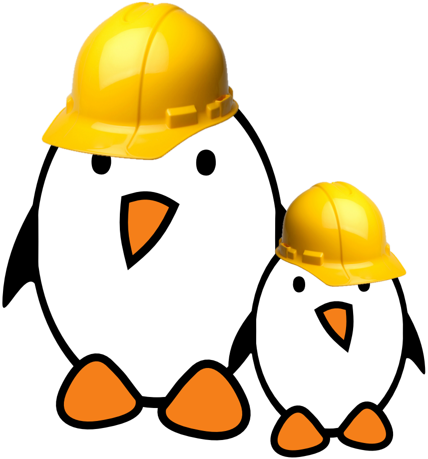
- Buildroot normally builds packages **sequentially**, one after the other.
- Calling Buildroot with `make -jX` has no effect
- Parallel build is used *within* the build of each package: Buildroot invokes each package build system with `make -jX`
  - This level of parallelization is controlled by `BR2_JLEVEL`
  - Defaults to 0, which means Buildroot auto-detects the number of CPUs cores
- Buildroot 2020.02 has introduced **experimental** support for top-level parallel build
  - Allows to build multiple different packages in parallel
  - Of course taking into account their dependencies
  - Allows to better use multi-core machines
  - Reduces build time significantly

Support for top-level parallel build (2)

- To use this experimental support:
  1. Enable `BR2_PER_PACKAGE_DIRECTORIES=y`
  2. Build with `make -jX`
- The *per-package* option ensures that each package uses its own `HOST_DIR`, `STAGING_DIR` and `TARGET_DIR` so that different packages can be built in parallel with no interference
- See `$(O)/per-package/<pkg>/`
- Limitations
  - Not yet supported by all packages, e.g *Qt5*
  - Absolutely requires that packages do not overwrite/change files installed by other packages
  - `<pkg>-reconfigure`, `<pkg>-rebuild`, `<pkg>-reinstall` not working

Reproducible builds

- Buildroot guarantees that for a given version/configuration, it will **always build the same components**, in the same version, with the same configuration.
- However, a number of aspects (time, user, build location) can affect the build and make two consecutive builds of the same configuration **not strictly identical**.
- `BR2_REPRODUCIBLE` enables experimental support for build reproducibility
- Goal: have **bit-identical results** when
  - Date/time is different (i.e. same build later)
  - Build location has the same path length



- Use `legal-info` for legal information extraction
- Use `graph-depends` for dependency graphing
- Use `graph-build` for build time graphing
- Use `BR2_EXTERNAL` to isolate the project-specific changes (packages, configs, etc.)

Application development

Building during development

- Buildroot is mainly a *final integration* tool: it is aimed at downloading and building **fixed** versions of software components, in a reproducible way.
- When doing active development of a software component, you need to be able to quickly change the code, build it, and deploy it on the target.
- The package build directory is temporary, and removed on `make clean`, so making changes here is not practical
- Buildroot does not automatically "update" your source code when the package is fetched from a version control system.
- Three solutions:
  - Build your software component outside of Buildroot during development. Doable for software components that are easy to build.
  - Use the `local` `SITE_METHOD` for your package
  - Use the `<pkg>_OVERRIDE_SRCDIR` mechanism

Building code for Buildroot

- The Buildroot cross-compiler is installed in `$(HOST_DIR)/bin`
- It is already set up to:
  - generate code for the configured architecture
  - look for libraries and headers in `$(STAGING_DIR)`
- Other useful tools that may be built by Buildroot are installed in `$(HOST_DIR)/bin`:
  - `pkg-config`, to find libraries. Beware that it is configured to return results for *target* libraries: it should only be used when cross-compiling.
  - `qmake`, when building Qt applications with this build system.
  - `autoconf`, `automake`, `libtool`, to use versions independent from the host system.
- Adding `$(HOST_DIR)/bin` to your `PATH` when cross-compiling is the easiest solution.

Building code for Buildroot: C program

Building a C program for the host

```
$ gcc -o foobar foobar.c
$ file foobar
foobar: ELF 64-bit LSB executable, x86-64, version 1...
```

Building a C program for the target

```
$ export PATH=$(pwd)/output/host/bin:$PATH
$ arm-linux-gcc -o foobar foobar.c
$ file foobar
foobar: ELF 32-bit LSB executable, ARM, EABI5 version 1...
```

Building code for Buildroot: pkg-config

Using the system pkg-config

```
$ pkg-config --cflags libpng
-I/usr/include/libpng12

$ pkg-config --libs libpng
-lpng12
```

Using the Buildroot pkg-config

```
$ export PATH=$(pwd)/output/host/bin:$PATH

$ pkg-config --cflags libpng
-I.../output/host/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/include/libpng16

$ pkg-config --libs libpng
-L.../output/host/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/lib -lpng16
```

Building code for Buildroot: autotools

- Building simple *autotools* components outside of Buildroot is easy:

```
$ export PATH=.../buildroot/output/host/bin/:$PATH
$ ./configure --host=arm-linux
```

- Passing `--host=arm-linux` tells the configure script to use the cross-compilation tools prefixed by `arm-linux-`.
- In more complex cases, some additional `CFLAGS` or `LDFLAGS` might be needed in the environment.

Building code for Buildroot: CMake

- Buildroot generates a *CMake toolchain file*, installed in `output/host/share/buildroot/toolchainfile.cmake`
- Tells *CMake* which cross-compilation tools to use
- Passed using the `CMAKE_TOOLCHAIN_FILE` *CMake* option
- https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html
- With this file, building *CMake* projects outside of Buildroot is easy:

```
$ cmake -DCMAKE_TOOLCHAIN_FILE=.../buildroot/output/host/share/buildroot/toolchainfile.cmake .
$ make
$ file app
app: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked...
```

Building code for Buildroot: Meson

- Buildroot generates a Meson *cross file*, installed in `output/host/etc/meson/cross-compilation.conf`
- Tells *Meson* which cross-compilation tools to use
- Passed using the `--cross-file` *Meson* option
- https://mesonbuild.com/Cross-compilation.html
- With this file, building *Meson* projects outside of Buildroot is easy:

```
$ mkdir build
$ meson --cross-file=../buildroot/output/host/etc/meson/cross-compilation.conf ..
$ ninja
$ file app
app: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked...
```

Building code for Buildroot: environment-setup

- Enable BR2_PACKAGE_HOST_ENVIRONMENT_SETUP
- Installs an helper shell script output/host/environment-setup that can be sourced in the shell to define a number of useful environment variables and aliases.
- Defines: CC, LD, AR, AS, CFLAGS, LDFLAGS, ARCH, etc.
- Defines configure as an alias to run a *configure* script with the right arguments, cmake as an alias to run *cmake* with the right arguments
- Drawback: once sourced, the shell environment is really only suitable for cross-compiling with Buildroot.

Building code for Buildroot: environment-setup

```
$ source output/host/environment-setup
 _           _ _     _                 _
| |__  _   _(_) | __| |_ __ ___   ___ | |_
| '_ \| | | | | |/ _` | '__/ _ \ / _ \| __|
| |_) | |_| | | | (_| | | | (_) | (_) | |_
|_.__/ \__,_|_|_|\__,_|_|  \___/ \___/ \__|

       Making embedded Linux easy!

Some tips:
* PATH now contains the SDK utilities
* Standard autotools variables (CC, LD, CFLAGS) are exported
* Kernel compilation variables (ARCH, CROSS_COMPILE, KERNELDIR) are exported
* To configure do "./configure $CONFIGURE_FLAGS" or use
  the "configure" alias
* To build CMake-based projects, use the "cmake" alias
```

```
$ echo $CC
/home/thomas/projets/buildroot/output/host/bin/arm-linux-gcc
$ echo $CFLAGS
-D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64 -Os -D_FORTIFY_SOURCE=1
$ echo $CROSS_COMPILE
/home/thomas/projets/buildroot/output/host/bin/arm-linux-
$ alias configure
alias configure='./configure --target=arm-buildroot-linux-gnueabihf --host=arm-buildroot-linux-gnueabihf \
    --build=x86_64-pc-linux-gnu --prefix=/usr --exec-prefix=/usr --sysconfdir=/etc --localstatedir=/var \
    --program-prefix='
```

local site method

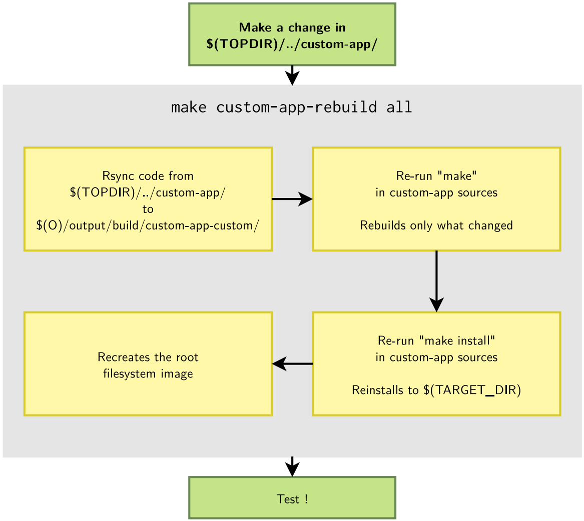
- Allows to tell Buildroot that the source code for a package is already available locally
- Allows to keep your source code under version control, separately, and have Buildroot always build your latest changes.
- Typical project organization:
  - `buildroot/`, the Buildroot source code
  - `external/`, your `BR2_EXTERNAL` tree
  - `custom-app/`, your custom application code
  - `custom-lib/`, your custom library
- In your package `.mk` file, use:

```
<pkg>_SITE = $(TOPDIR)/../custom-app
<pkg>_SITE_METHOD = local
```

Effect of `local` site method

- For the first build, the source code of your package is *rsync'ed* from `<pkg>_SITE` to the build directory, and built there.
- After making changes to the source code, you can run:
  - `make <pkg>-reconfigure`
  - `make <pkg>-rebuild`
  - `make <pkg>-reinstall`
- Buildroot will first *rsync* again the package source code (copying only the modified files) and restart the build from the requested step.

local site method workflow



<pkg>_OVERRIDE_SRCDIR

- The `local` site method solution is appropriate when the package uses this method for all developers
  - Requires that all developers fetch locally the source code for all custom applications and libraries
- An alternate solution is that packages for custom applications and libraries fetch their source code from version control systems
  - Using the `git`, `svn`, `cvs`, etc. fetching methods
- Then, locally, a user can **override** how the package is fetched using `<pkg>_OVERRIDE_SRCDIR`
  - It tells Buildroot to not *download* the package source code, but to copy it from a local directory.
- The package then behaves as if it was using the `local` site method.

Passing <pkg>_OVERRIDE_SRCDIR

- <pkg>_OVERRIDE_SRCDIR values are specified in a *package override file*, configured in BR2_PACKAGE_OVERRIDE_FILE, by default $(CONFIG_DIR)/local.mk.

Example local.mk

```
LIBPNG_OVERRIDE_SRCDIR = $(HOME)/projects/libpng
LINUX_OVERRIDE_SRCDIR = $(HOME)/projects/linux
```

Debugging: debugging symbols and stripping

- To use debuggers, you need the programs and libraries to be built with debugging symbols.
- The `BR2_ENABLE_DEBUG` option controls whether programs and libraries are built with debugging symbols
  - Disabled by default.
  - Sub-options allow to control the amount of debugging symbols (i.e. gcc options `-g1`, `-g2` and `-g3`).
- The `BR2_STRIP_strip` option allows to disable or enable stripping of binaries on the target.
  - Enabled by default.

Debugging: debugging symbols and stripping

- With `BR2_ENABLE_DEBUG=y` and `BR2_STRIP_strip=y`
  - get debugging symbols in `$(STAGING_DIR)` for libraries, and in the build directories for everything.
  - stripped binaries in `$(TARGET_DIR)`
  - Appropriate for **remote debugging**
- With `BR2_ENABLE_DEBUG=y` and `BR2_STRIP_strip` disabled
  - debugging symbols in both `$(STAGING_DIR)` and `$(TARGET_DIR)`
  - appropriate for **on-target debugging**

Debugging: remote debugging requirements

- To do remote debugging, you need:
  - A **cross-debugger**
    - With the *internal toolchain backend*, can be built using `BR2_PACKAGE_HOST_GDB=y`.
    - With the *external toolchain backend*, is either provided pre-built by the toolchain, or can be built using `BR2_PACKAGE_HOST_GDB=y`.
  - **gdbserver**
    - With the *internal toolchain backend*, can be built using `BR2_PACKAGE_GDB=y` + `BR2_PACKAGE_GDB_SERVER=y`
    - With the *external toolchain backend*, if `gdbserver` is provided by the toolchain it can be copied to the target using `BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY=y` or otherwise built from source like with the internal toolchain backend.

Debugging: remote debugging setup

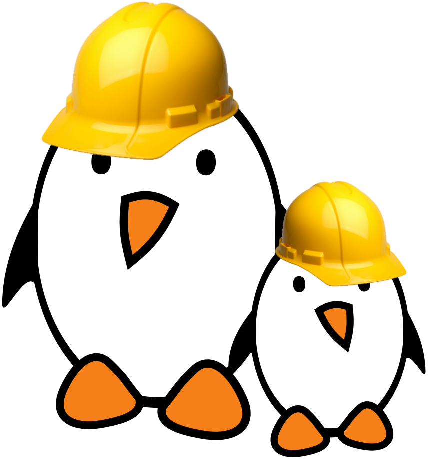
- On the target, start *gdbserver*
  - Use a TCP socket, network connectivity needed
  - The *multi* mode is quite convenient
  - `$ gdbserver --multi localhost:2345`
- On the host, start `<tuple>-gdb`
  - `$ ./output/host/bin/<tuple>-gdb <program>`
  - `<program>` is the path to the program to debug, with debugging symbols
- Inside *gdb*, you need to:
  - Connect to the target:
    `(gdb) target extended-remote <ip>:2345`
  - Tell the target which program to run:
    `(gdb) set remote exec-file myapp`
  - Set the path to the *sysroot* so that *gdb* can find debugging symbols for libraries:
    `(gdb) set sysroot ./output/staging/`
  - Start the program:
    `(gdb) run`

Debugging tools available in Buildroot

- Buildroot also includes a huge amount of other debugging or profiling related tools.
- To list just a few:
  - strace
  - ltrace
  - LTTng
  - perf
  - sysdig
  - sysprof
  - OProfile
  - valgrind
- Look in `Target packages` → `Debugging, profiling and benchmark` for more.

Generating a SDK for application developers

- If you would like application developers to build applications for a Buildroot generated system, without building Buildroot, you can generate a SDK.
- To achieve this:
  - Run `make sdk`, which prepares the SDK to be relocatable
  - Tarball the contents of the *host* directory, i.e. `output/host`
  - Share the tarball with your application developers
  - They must uncompress it, and run the `relocate-sdk.sh` script
- **Warning**: the SDK must remain in sync with the root filesystem running on the target, otherwise applications built with the SDK may not run properly.



- Build and run your own application
- Remote debug your application
- Use `<pkg>_OVERRIDE_SRCDIR`

Understanding Buildroot internals

Configuration system

- Uses, almost unchanged, the *kconfig* code from the kernel, in `support/kconfig` (variable `CONFIG`)
- *kconfig* tools are built in `$(BUILD_DIR)/buildroot-config/`
- The main `Config.in` file, passed to *config, is at the top-level of the Buildroot source tree

```
CONFIG_CONFIG_IN = Config.in
CONFIG = support/kconfig
BR2_CONFIG = $(CONFIG_DIR)/.config

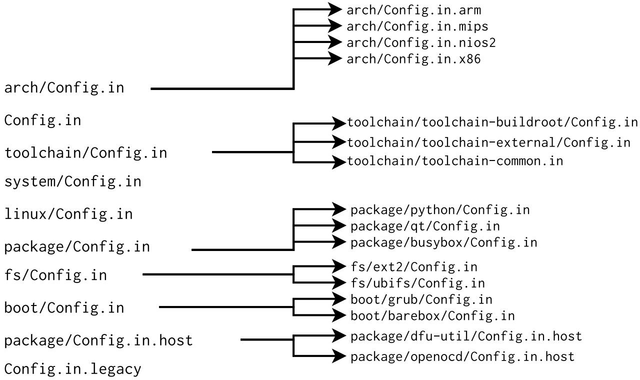
-include $(BR2_CONFIG)

$(BUILD_DIR)/buildroot-config/%onf:
        mkdir -p $(@D)/lxdialog
        ... $(MAKE) ... -C $(CONFIG) -f Makefile.br $(@F)

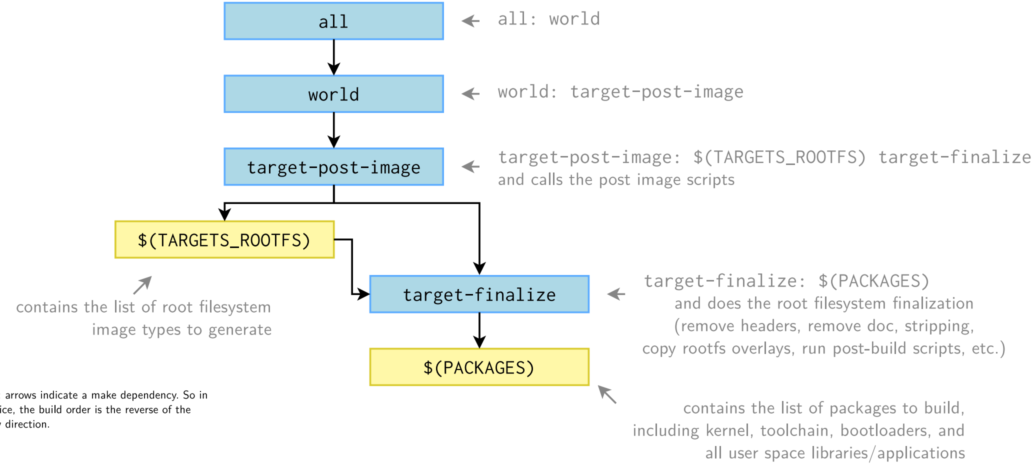
menuconfig: $(BUILD_DIR)/buildroot-config/mconf outputmakefile
        @$(COMMON_CONFIG_ENV) $< $(CONFIG_CONFIG_IN)
```

Configuration hierarchy

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options  --->
```



When you run make...



Note: arrows indicate a make dependency. So in practice, the build order is the reverse of the arrow direction.

Where is $(PACKAGES) filled?

Part of package/pkg-generic.mk

```
#  argument 1 is the lowercase package name
#  argument 2 is the uppercase package name, including a HOST_ prefix
#             for host packages

define inner-generic-package
 ...
$(2)_KCONFIG_VAR = BR2_PACKAGE_$(2)
 ...
ifeq ($$($$($(2)_KCONFIG_VAR)),y)
PACKAGES += $(1)
endif # $(2)_KCONFIG_VAR

endef # inner-generic-package
```

- Adds the lowercase name of an enabled package as a make target to the `$(PACKAGES)` variable
- `package/pkg-generic.mk` is really the core of the package infrastructure

Diving into pkg-generic.mk

- The `package/pkg-generic.mk` file is divided in two main parts:
  1. Definition of the actions done in each step of a package build process. Done through *stamp file targets*.
  2. Definition of the `inner-generic-package`, `generic-package` and `host-generic-package` macros, that define the sequence of actions, as well as all the variables needed to handle the build of a package.

Definition of the actions: code

```
$(BUILD_DIR)/%/.stamp_downloaded:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_extracted:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_patched:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_configured:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_built:
	# Do some stuff here
	$(Q)touch $@
```

```
$(BUILD_DIR)/%/.stamp_host_installed:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_staging_installed:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_images_installed:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_target_installed:
	# Do some stuff here
	$(Q)touch $@

$(BUILD_DIR)/%/.stamp_installed:
	# Do some stuff here
	$(Q)touch $@
```

- `$(BUILD_DIR)/%/` → build directory of any package
- a *make* target depending on one stamp file will trigger the corresponding action
- the *stamp file* prevents the action from being re-executed

Action example 1: download

```
# Retrieve the archive
$(BUILD_DIR)/%/.stamp_downloaded:
	$(foreach hook,$($(PKG)_PRE_DOWNLOAD_HOOKS),$(call $(hook))$(sep))
	[...]
	$(foreach p,$($(PKG)_ALL_DOWNLOADS),$(call DOWNLOAD,$(p))$(sep))
	$(foreach hook,$($(PKG)_POST_DOWNLOAD_HOOKS),$(call $(hook))$(sep))
	$(Q)mkdir -p $(@D)
	$(Q)touch $@
```

- Step handled by the package infrastructure
- In all *stamp file targets*, `PKG` is the upper case name of the package. So when used for BusyBox, `$($(PKG)_SOURCE)` is the value of `BUSYBOX_SOURCE`.
- *Hooks*: make macros called before and after each step.
- `<pkg>_ALL_DOWNLOADS` lists all the files to be downloaded, which includes the ones listed in `<pkg>_SOURCE`, `<pkg>_EXTRA_DOWNLOADS` and `<pkg>_PATCH`.

Action example 2: build

```
# Build
$(BUILD_DIR)/%/.stamp_built::
	@$(call step_start,build)
	@$(call MESSAGE,"Building")
	$(foreach hook,$($(PKG)_PRE_BUILD_HOOKS),$(call $(hook))$(sep))
	+$($(PKG)_BUILD_CMDS)
	$(foreach hook,$($(PKG)_POST_BUILD_HOOKS),$(call $(hook))$(sep))
	@$(call step_end,build)
	$(Q)touch $@
```

- Step handled by the package, by defining a value for `<pkg>_BUILD_CMDS`.
- Same principle of *hooks*
- `step_start` and `step_end` are part of instrumentation to measure the duration of each step (and other actions)

The generic-package macro

- Packages built for the target:

```
generic-package = $(call inner-generic-package,
                        $(pkgname),$(call UPPERCASE,$(pkgname)),
                        $(call UPPERCASE,$(pkgname)),target)
```

- Packages built for the host:

```
host-generic-package = $(call inner-generic-package,
                        host-$(pkgname),$(call UPPERCASE,host-$(pkgname)),
                        $(call UPPERCASE,$(pkgname)),host)
```

- In `package/libzlib/libzlib.mk`:

```
LIBZLIB_... = ...

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

- Leads to:

```
$(call inner-generic-package,libzlib,LIBZLIB,LIBZLIB,target)
$(call inner-generic-package,host-libzlib,HOST_LIBZLIB,LIBZLIB,host)
```

inner-generic-package: defining variables

Macro code

```
$(2)_TYPE     =  $(4)
$(2)_NAME     =  $(1)
$(2)_RAWNAME =  $$(patsubst host-%,%,$(1))

$(2)_BASE_NAME = $(1)-$$($(2)_VERSION)
$(2)_DIR       = $$(BUILD_DIR)/$$($(2)_BASE_NAME)



ifndef $(2)_SOURCE
 ifdef $(3)_SOURCE
  $(2)_SOURCE = $$($(3)_SOURCE)
 else
  $(2)_SOURCE ?=
    $$($(2)_RAWNAME)-$$($(2)_VERSION).tar.gz
 endif
endif

ifndef $(2)_SITE
 ifdef $(3)_SITE
  $(2)_SITE = $$($(3)_SITE)
 endif
endif

...
```

Expanded for host-libzlib

```
HOST_LIBZLIB_TYPE     = host
HOST_LIBZLIB_NAME     = host-libzlib
HOST_LIBZLIB_RAWNAME = libzlib

HOST_LIBZLIB_BASE_NAME =
  host-libzlib-$(HOST_LIBZLIB_VERSION)
HOST_LIBZLIB_DIR       =
  $(BUILD_DIR)/host-libzlib-$(HOST_LIBZLIB_VERSION)

ifndef HOST_LIBZLIB_SOURCE
 ifdef LIBZLIB_SOURCE
  HOST_LIBZLIB_SOURCE = $(LIBZLIB_SOURCE)
 else
  HOST_LIBZLIB_SOURCE ?=
   libzlib-$(HOST_LIBZLIB_VERSION).tar.gz
 endif
endif

ifndef HOST_LIBZLIB_SITE
 ifdef LIBZLIB_SITE
  HOST_LIBZLIB_SITE = $(LIBZLIB_SITE)
 endif
endif

...
```

inner-generic-package: dependencies

```
ifeq ($(4),target)
ifeq ($$($(2)_ADD_SKELETON_DEPENDENCY),YES)
$(2)_DEPENDENCIES += skeleton
endif
ifeq ($$($(2)_ADD_TOOLCHAIN_DEPENDENCY),YES)
$(2)_DEPENDENCIES += toolchain
endif
endif

...

ifeq ($$(BR2_CCACHE),y)
ifeq ($$(filter host-tar host-skeleton host-xz host-lzip host-fakedate host-ccache,$(1)),)
$(2)_DEPENDENCIES += host-ccache
endif
endif
```

- Adding the `skeleton` and `toolchain` dependencies to target packages. Except for some specific packages (e.g. C library).

inner-generic-package: stamp files

```
$(2)_TARGET_INSTALL =               $$($(2)_DIR)/.stamp_installed
$(2)_TARGET_INSTALL_TARGET =        $$($(2)_DIR)/.stamp_target_installed
$(2)_TARGET_INSTALL_STAGING =       $$($(2)_DIR)/.stamp_staging_installed
$(2)_TARGET_INSTALL_IMAGES =        $$($(2)_DIR)/.stamp_images_installed
$(2)_TARGET_INSTALL_HOST =          $$($(2)_DIR)/.stamp_host_installed
$(2)_TARGET_BUILD =                 $$($(2)_DIR)/.stamp_built
$(2)_TARGET_CONFIGURE =             $$($(2)_DIR)/.stamp_configured
$(2)_TARGET_RSYNC =                 $$($(2)_DIR)/.stamp_rsynced
$(2)_TARGET_RSYNC_SOURCE =          $$($(2)_DIR)/.stamp_rsync_sourced
$(2)_TARGET_PATCH =                 $$($(2)_DIR)/.stamp_patched
$(2)_TARGET_EXTRACT =               $$($(2)_DIR)/.stamp_extracted
$(2)_TARGET_SOURCE =                $$($(2)_DIR)/.stamp_downloaded
$(2)_TARGET_DIRCLEAN =              $$($(2)_DIR)/.stamp_dircleaned
```

- Defines shortcuts to reference the stamp files

```
$$($(2)_TARGET_INSTALL):                PKG=$(2)
$$($(2)_TARGET_INSTALL_TARGET):         PKG=$(2)
$$($(2)_TARGET_INSTALL_STAGING):        PKG=$(2)
$$($(2)_TARGET_INSTALL_IMAGES):         PKG=$(2)
$$($(2)_TARGET_INSTALL_HOST):           PKG=$(2)
[...]
```

- Pass variables to the stamp file targets, especially `PKG`

```
$(1):                   $(1)-install
$(1)-install:           $$($(2)_TARGET_INSTALL)

ifeq ($$($(2)_INSTALL_TARGET),YES)
$$($(2)_TARGET_INSTALL): $$($(2)_TARGET_INSTALL_TARGET)
endif
ifeq ($$($(2)_INSTALL_STAGING),YES)
$$($(2)_TARGET_INSTALL): $$($(2)_TARGET_INSTALL_STAGING)
endif
ifeq ($$($(2)_INSTALL_IMAGES),YES)
$$($(2)_TARGET_INSTALL): $$($(2)_TARGET_INSTALL_IMAGES)
endif

$(1)-install-target:           $$($(2)_TARGET_INSTALL_TARGET)
$$($(2)_TARGET_INSTALL_TARGET): $$($(2)_TARGET_BUILD)

$(1)-install-staging:                 $$($(2)_TARGET_INSTALL_STAGING)
$$($(2)_TARGET_INSTALL_STAGING):       $$($(2)_TARGET_BUILD)

$(1)-install-images:           $$($(2)_TARGET_INSTALL_IMAGES)
$$($(2)_TARGET_INSTALL_IMAGES): $$($(2)_TARGET_BUILD)
```

```
$(1)-build:             $$($(2)_TARGET_BUILD)
$$($(2)_TARGET_BUILD):  $$($(2)_TARGET_CONFIGURE)

$(1)-configure:                 $$($(2)_TARGET_CONFIGURE)
$$($(2)_TARGET_CONFIGURE):      | $$($(2)_FINAL_DEPENDENCIES)
$$($(2)_TARGET_CONFIGURE):      $$($(2)_TARGET_PATCH)

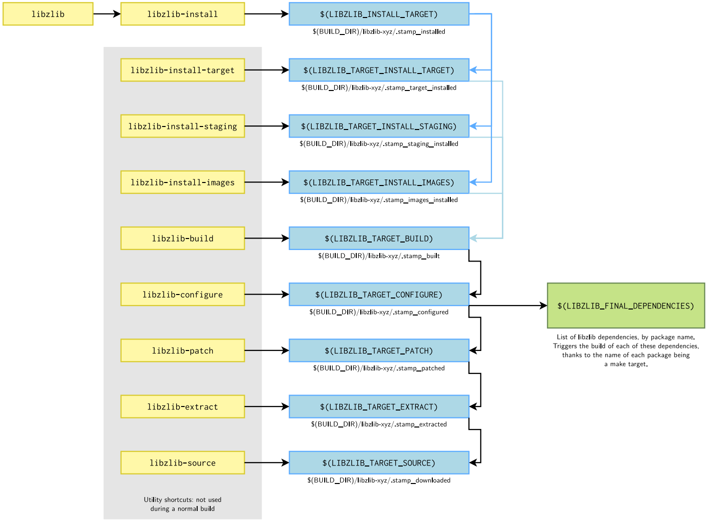
$(1)-patch:             $$($(2)_TARGET_PATCH)
$$($(2)_TARGET_PATCH):  $$($(2)_TARGET_EXTRACT)

$(1)-extract:                   $$($(2)_TARGET_EXTRACT)
$$($(2)_TARGET_EXTRACT):        $$($(2)_TARGET_SOURCE)
$$($(2)_TARGET_EXTRACT): | $$($(2)_FINAL_EXTRACT_DEPENDENCIES)

$(1)-source:            $$($(2)_TARGET_SOURCE)
$$($(2)_TARGET_SOURCE): | $$($(2)_FINAL_DOWNLOAD_DEPENDENCIES)

$$($(2)_TARGET_SOURCE): | prepare
$$($(2)_TARGET_SOURCE): | dependencies
```

inner-generic-package: sequencing diagram



Preparation work: prepare, dependencies

pkg-generic.mk

```
$$($(2)_TARGET_SOURCE): | prepare
$$($(2)_TARGET_SOURCE): | dependencies
```

- All packages have two targets in their dependencies:
  - `prepare`: generates a kconfig-related `auto.conf` file
  - `dependencies`: triggers the check of Buildroot system dependencies, i.e. things that must be installed on the machine to use Buildroot

Rebuilding packages?

- Once one step of a package build process has been done, it is never done again due to the *stamp file*
- Even if the package configuration is changed, or the package is disabled → Buildroot doesn't try to be smart
- One can force rebuilding a package from its configure, build or install step using `make <pkg>-reconfigure`, `make <pkg>-rebuild` or `make <pkg>-reinstall`

```
$(1)-clean-for-reinstall:
			rm -f $$($(2)_TARGET_INSTALL)
			rm -f $$($(2)_TARGET_INSTALL_STAGING)
			rm -f $$($(2)_TARGET_INSTALL_TARGET)
			rm -f $$($(2)_TARGET_INSTALL_IMAGES)
			rm -f $$($(2)_TARGET_INSTALL_HOST)

$(1)-reinstall:		$(1)-clean-for-reinstall $(1)

$(1)-clean-for-rebuild: $(1)-clean-for-reinstall
			rm -f $$($(2)_TARGET_BUILD)

$(1)-rebuild:		$(1)-clean-for-rebuild $(1)

$(1)-clean-for-reconfigure: $(1)-clean-for-rebuild
			rm -f $$($(2)_TARGET_CONFIGURE)

$(1)-reconfigure:	$(1)-clean-for-reconfigure $(1)
```

Specialized package infrastructures

- The `generic-package` infrastructure is fine for packages having a **custom** build system
- For packages using a **well-known build system**, we want to factorize more logic
- Specialized **package infrastructures** were created to handle these packages, and reduce the amount of duplication
- For *autotools*, *CMake*, *Python*, *Perl*, *Lua*, *Meson*, *Golang*, *QMake*, *kconfig*, *Rust*, *kernel-module*, *Erlang*, *Waf* packages

CMake package example: flann

package/flann/flann.mk

```
FLANN_VERSION = 1.9.1
FLANN_SITE = $(call github,mariusmuja,flann,$(FLANN_VERSION))
FLANN_INSTALL_STAGING = YES
FLANN_LICENSE = BSD-3-Clause
FLANN_LICENSE_FILES = COPYING
FLANN_CONF_OPTS = \
	-DBUILD_C_BINDINGS=ON \
	-DBUILD_PYTHON_BINDINGS=OFF \
	-DBUILD_MATLAB_BINDINGS=OFF \
	-DBUILD_EXAMPLES=$(if $(BR2_PACKAGE_FLANN_EXAMPLES),ON,OFF) \
	-DUSE_OPENMP=$(if $(BR2_GCC_ENABLE_OPENMP),ON,OFF) \
	-DPYTHON_EXECUTABLE=OFF \
	-DCMAKE_DISABLE_FIND_PACKAGE_HDF5=TRUE

$(eval $(cmake-package))
```

CMake package infrastructure (1/2)

```
define inner-cmake-package

$(2)_CONF_ENV                   ?=
$(2)_CONF_OPTS                  ?=
...

$(2)_SRCDIR                     = $$($(2)_DIR)/$$($(2)_SUBDIR)
$(2)_BUILDDIR                   = $$($(2)_SRCDIR)

ifndef $(2)_CONFIGURE_CMDS
ifeq ($(4),target)
define $(2)_CONFIGURE_CMDS
    (cd $$($$(PKG)_BUILDDIR) && \
     $$($$(PKG)_CONF_ENV) $$(HOST_DIR)/bin/cmake $$($$(PKG)_SRCDIR) \
         -DCMAKE_TOOLCHAIN_FILE="$$(HOST_DIR)/share/buildroot/toolchainfile.cmake" \
         ...
         $$($$(PKG)_CONF_OPTS) \
    )
endef
else
define $(2)_CONFIGURE_CMDS
... host case ...
endef
endif
endif
```

CMake package infrastructure (2/2)

```
$(2)_DEPENDENCIES += host-cmake

ifndef $(2)_BUILD_CMDS
ifeq ($(4),target)
define $(2)_BUILD_CMDS
        $$(TARGET_MAKE_ENV) $$($$(PKG)_MAKE_ENV) $$($$(PKG)_MAKE) $$($$(PKG)_MAKE_OPTS)
            -C $$($$(PKG)_BUILDDIR)
endef
else
... host case ...
endif
endif

... other commands ...

ifndef $(2)_INSTALL_TARGET_CMDS
define $(2)_INSTALL_TARGET_CMDS
        $$(TARGET_MAKE_ENV) $$($$(PKG)_MAKE_ENV) $$($$(PKG)_MAKE) $$($$(PKG)_MAKE_OPTS)
          $$($$(PKG)_INSTALL_TARGET_OPT) -C $$($$(PKG)_BUILDDIR)
endef
endif

$(call inner-generic-package,$(1),$(2),$(3),$(4))

endef

cmake-package = $(call inner-cmake-package,$(pkgname),...,target)
host-cmake-package = $(call inner-cmake-package,host-$(pkgname),...,host)
```

Autoreconf in pkg-autotools.mk

- Package infrastructures can also add additional capabilities controlled by variables in packages
- For example, with the autotools-package infra, one can do FOOBAR_AUTORECONF = YES in a package to trigger an *autoreconf* before the *configure* script is executed
- Implementation in pkg-autotools.mk

```
define AUTORECONF_HOOK
        @$$(call MESSAGE,"Autoreconfiguring")
        $$(Q)cd $$($$(PKG)_SRCDIR) && $$($$(PKG)_AUTORECONF_ENV) $$(AUTORECONF)
             $$($$(PKG)_AUTORECONF_OPTS)
        ...
endef

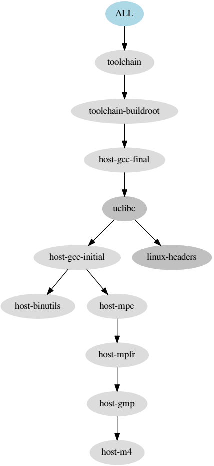
ifeq ($$($(2)_AUTORECONF),YES)
...
$(2)_PRE_CONFIGURE_HOOKS += AUTORECONF_HOOK
$(2)_DEPENDENCIES += host-automake host-autoconf host-libtool
endif
```

Toolchain support

- One *virtual package*, `toolchain`, with two implementations in the form of two packages: `toolchain-buildroot` and `toolchain-external`
- `toolchain-buildroot` implements the **internal toolchain back-end**, where Buildroot builds the cross-compilation toolchain from scratch. This package simply depends on `host-gcc-final` to trigger the entire build process
- `toolchain-external` implements the **external toolchain back-end**, where Buildroot uses an existing pre-built toolchain

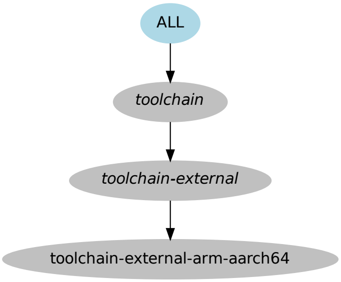
Internal toolchain back-end

- Build starts with utility host tools and libraries needed for gcc (`host-m4`, `host-mpc`, `host-mpfr`, `host-gmp`). Installed in `$(HOST_DIR)/{bin,include,lib}`
- Build goes on with the cross binutils, `host-binutils`, installed in `$(HOST_DIR)/bin`
- Then the first stage compiler, `host-gcc-initial`
- We need the `linux-headers`, installed in `$(STAGING_DIR)/usr/include`
- We build the C library, `uclibc` in this example. Installed in `$(STAGING_DIR)/lib`, `$(STAGING_DIR)/usr/include` and of course `$(TARGET_DIR)/lib`
- We build the final compiler `host-gcc-final`, installed in `$(HOST_DIR)/bin`



External toolchain back-end

- toolchain-external-package infrastructure, implementing the common logic for all external toolchains
  - Implemented in toolchain/toolchain-external/pkg-toolchain-external.mk
- Packages in toolchain/toolchain-external/ are using this infrastructure
  - E.g. toolchain-external-arm-aarch64, toolchain-external-bootlin
- toolchain-external is a virtual package itself depends on the selected external toolchain.



External toolchain example

toolchain/toolchain-external/toolchain-external-arm-aarch64/toolchain-external-arm-aarch64.mk

```
TOOLCHAIN_EXTERNAL_ARM_AARCH64_VERSION = 2020.11
TOOLCHAIN_EXTERNAL_ARM_AARCH64_SITE = \
    https://developer.arm.com/-/media/Files/downloads/
      gnu-a/10.2-$(TOOLCHAIN_EXTERNAL_ARM_AARCH64_VERSION)/binrel

TOOLCHAIN_EXTERNAL_ARM_AARCH64_SOURCE = \
    gcc-arm-10.2-$(TOOLCHAIN_EXTERNAL_ARM_AARCH64_VERSION)-x86_64-aarch64-none-linux-gnu.tar.xz

$(eval $(toolchain-external-package))
```

toolchain-external-package logic

1. Extract the toolchain to `$(HOST_DIR)/opt/ext-toolchain`
2. Run some checks on the toolchain to verify it matches the configuration specified in *menuconfig*
3. Copy the toolchain *sysroot* (C library and headers, kernel headers) to `$(STAGING_DIR)/usr/{include,lib}`
4. Copy the toolchain libraries to `$(TARGET_DIR)/usr/lib`
5. Create symbolic links or wrappers for the compiler, linker, debugger, etc from `$(HOST_DIR)/bin/<tuple>-<tool>` to `$(HOST_DIR)/opt/ext-toolchain/bin/<tuple>-<tool>`
6. A wrapper program is used for certain tools (gcc, ld, g++, etc.) in order to ensure a certain number of compiler flags are used, especially `--sysroot=$(STAGING_DIR)` and target-specific flags.

Root filesystem image generation

- Once all the targets in `$(PACKAGES)` have been built, it's time to create the root filesystem images
- First, the `target-finalize` target does some cleanup of `$(TARGET_DIR)` by removing documentation, headers, static libraries, etc.
- Then the root filesystem image targets listed in `$(ROOTFS_TARGETS)` are processed
- These targets are added by the common filesystem image generation infrastructure `rootfs`, in `fs/common.mk`
- The purpose of this infrastructure is to:
  - Collect the users, permissions and device tables
  - Make a copy of `TARGET_DIR` per filesystem image
  - Generate a shell script that assigns users, permissions and invokes the filesystem image creation utility
  - Invoke the shell script under `fakeroot`

fs/common.mk, dependencies and table generation

```
ROOTFS_COMMON_DEPENDENCIES = \
	host-fakeroot host-makedevs \
	$(BR2_TAR_HOST_DEPENDENCY) \
	$(if $(PACKAGES_USERS)$(ROOTFS_USERS_TABLES),host-mkpasswd)

rootfs-common: $(ROOTFS_COMMON_DEPENDENCIES) target-finalize
	@$(call MESSAGE,"Generating root filesystems common tables")
	rm -rf $(FS_DIR)
	mkdir -p $(FS_DIR)
	$(call PRINTF,$(PACKAGES_USERS)) >> $(ROOTFS_FULL_USERS_TABLE)
	cat $(ROOTFS_USERS_TABLES) >> $(ROOTFS_FULL_USERS_TABLE)
	$(call PRINTF,$(PACKAGES_PERMISSIONS_TABLE)) > $(ROOTFS_FULL_DEVICES_TABLE)
	cat $(ROOTFS_DEVICE_TABLES) >> $(ROOTFS_FULL_DEVICES_TABLE)
	$(call PRINTF,$(PACKAGES_DEVICES_TABLE)) >> $(ROOTFS_FULL_DEVICES_TABLE)
```

fs/common.mk, rootfs infrastructure 1

```
define inner-rootfs

ROOTFS_$(2)_IMAGE_NAME ?= rootfs.$(1)
ROOTFS_$(2)_FINAL_IMAGE_NAME = $$(strip $$(ROOTFS_$(2)_IMAGE_NAME))
ROOTFS_$(2)_DIR = $$(FS_DIR)/$(1)
ROOTFS_$(2)_TARGET_DIR = $$(ROOTFS_$(2)_DIR)/target

ROOTFS_$(2)_DEPENDENCIES += rootfs-common
```

```
$$(BINARIES_DIR)/$$(ROOTFS_$(2)_FINAL_IMAGE_NAME): $$(ROOTFS_$(2)_DEPENDENCIES)
	@$$(call MESSAGE,"Generating filesystem image $$(ROOTFS_$(2)_FINAL_IMAGE_NAME)")
	[...]
	mkdir -p $$(ROOTFS_$(2)_DIR)
	rsync -auH \
		--exclude=/$$(notdir $$(TARGET_DIR_WARNING_FILE)) \
		$$(BASE_TARGET_DIR)/ \
		$$(TARGET_DIR)
	echo '#!/bin/sh' > $$(FAKEROOT_SCRIPT)
	echo "set -e" >> $$(FAKEROOT_SCRIPT)
	echo "chown -h -R 0:0 $$(TARGET_DIR)" >> $$(FAKEROOT_SCRIPT)
	PATH=$$(BR_PATH) $$(TOPDIR)/support/scripts/mkusers $$(ROOTFS_FULL_USERS_TABLE) $$(TARGET_DIR) >> $$(FAKEROOT_SCRIPT)
	echo "$$(HOST_DIR)/bin/makedevs -d $$(ROOTFS_FULL_DEVICES_TABLE) $$(TARGET_DIR)" >> $$(FAKEROOT_SCRIPT)
	[...]
	$$(call PRINTF,$$(ROOTFS_$(2)_CMD)) >> $$(FAKEROOT_SCRIPT)
	chmod a+x $$(FAKEROOT_SCRIPT)
	PATH=$$(BR_PATH) $$(HOST_DIR)/bin/fakeroot -- $$(FAKEROOT_SCRIPT)
[...]
ifeq ($$(BR2_TARGET_ROOTFS_$(2)),y)
TARGETS_ROOTFS += rootfs-$(1)
endif
endef

rootfs = $(call inner-rootfs,$(pkgname),$(call UPPERCASE,$(pkgname)))
```

fs/ubifs/ubifs.mk

```
UBIFS_OPTS := -e $(BR2_TARGET_ROOTFS_UBIFS_LEBSIZE) \
              -c $(BR2_TARGET_ROOTFS_UBIFS_MAXLEBCNT) \
              -m $(BR2_TARGET_ROOTFS_UBIFS_MINIOSIZE)

ifeq ($(BR2_TARGET_ROOTFS_UBIFS_RT_ZLIB),y)
UBIFS_OPTS += -x zlib
endif
...

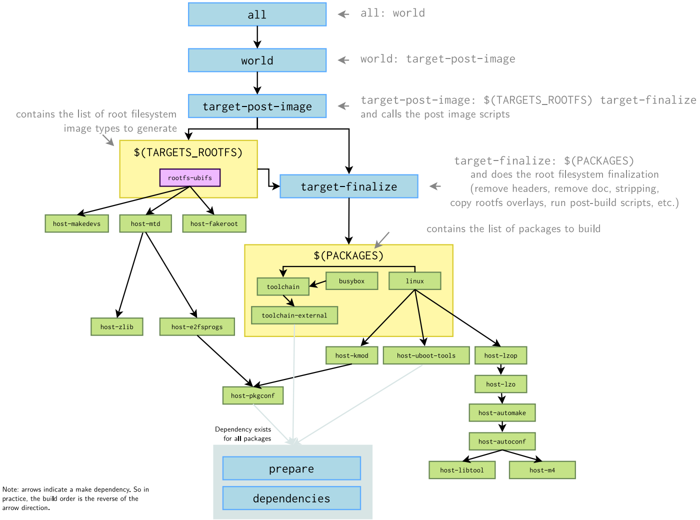
UBIFS_OPTS += $(call qstrip,$(BR2_TARGET_ROOTFS_UBIFS_OPTS))

ROOTFS_UBIFS_DEPENDENCIES = host-mtd

define ROOTFS_UBIFS_CMD
        $(HOST_DIR)/sbin/mkfs.ubifs -d $(TARGET_DIR) $(UBIFS_OPTS) -o $@
endef

$(eval $(rootfs))
```

Final example



Buildroot community: support and contribution

Corrections, suggestions, contributions and translations are welcome!

embedded Linux and kernel engineering

Documentation

- Buildroot comes with its own documentation
- Pre-built versions available at https://buildroot.org/docs.html (PDF, HTML, text)
- Source code of the manual located in `docs/manual` in the Buildroot sources
  - Written in *Asciidoc* format
- The manual can be built with:
  - `make manual`
  - or just `make manual-html`, `make manual-pdf`, `make manual-epub`, `make manual-text`, `make manual-split-html`
  - A number of tools need to be installed on your machine, see the manual itself.

Getting support

- Free support
  - The *mailing list* for e-mail discussion
    http://lists.busybox.net/mailman/listinfo/buildroot
    1400+ subscribers, quite heavy traffic.
  - The IRC channel, #buildroot on the OFTC network, for interactive discussion
    60+ people, most available during European daylight hours
  - Bug tracker
    https://bugs.busybox.net/buglist.cgi?product=buildroot
- Commercial support
  - A number of embedded Linux services companies, including Bootlin, can provide commercial services around Buildroot.

Tips to get free support

- If you have a build issue to report:
  - Make sure to reproduce after a `make clean all` cycle
  - Include the Buildroot version, Buildroot `.config` that reproduces the issue, and last 100-200 lines of the build output in your report.
  - Use *pastebin* sites like `https://paste.ack.tf/` when reporting issues over IRC.
- The community will be much more likely to help you if you use a recent Buildroot version.

Release schedule

- The Buildroot community publishes stable releases every three months.
- YYYY.02, YYYY.05, YYYY.08 and YYYY.11 every year.
- The three months cycle is split in two periods
  - Two first months of active development
  - One month of stabilization before the release
- At the beginning of the stabilization phase, `-rc1` is released.
- Several `-rc` versions are published during this stabilization phase, until the final release.
- Development not completely stopped during the stabilization, a `next` branch is opened.
- The YYYY.02 is a long term support release, maintained during one year with security, bug and build fixes.

Contribution process

- Contributions are made in the form of patches
- Created with `git` and sent by e-mail to the mailing list
  - Use `git send-email` to avoid issues
  - Use `get-developers` to know to who patches should be sent
- The patches are reviewed, tested and discussed by the community
  - You may be requested to modify your patches, and submit updated versions
- Once ready, they are applied by one of the project maintainers
- Some contributions may be rejected if they do not fall within the Buildroot principles/ideas, as discussed by the community.

Patchwork

- Tool that records all patches sent on the mailing list
- Allows the community to see which patches need review/testing, and the maintainers which patches can be applied.
- Everyone can create an account to manage his own patches
- https://patchwork.ozlabs.org/project/buildroot/list/

ID	Patch	Series	A/F/R/T	S/W/F	Date	Submitter	Delegate	State
1059249	[1/1] package/bird: add optional protocols	[1/1] package/bird: add optional protocols	- - - -	0 0 0	2019-03-20	Adrien Gallouët		New
1059199	[v2,3/3] gummiboot: fix compatibility with newer glibc versions	Gummiboot fixup	- - - -	0 0 0	2019-03-20	Esben Haabendal		New
1059198	[v2,2/3] gummiboot: upgrade to last commit before being removed	Gummiboot fixup	- - - -	0 0 0	2019-03-20	Esben Haabendal		New
1059200	[v2,1/3] gummiboot: use new official upstream git repository	Gummiboot fixup	- - - -	0 0 0	2019-03-20	Esben Haabendal		New
1059145	[v2] support/testing: add xserver + Mesa OpenGL (GLX) + glxinfo	[v2] support/testing: add xserver + Mesa OpenGL (GLX) + glxinfo	- - - -	0 0 0	2019-03-20	Romain Naour		New
1058782	[v5,3/4] configs: qemu_riscv*_virt: Use OpenSBI by default	Untitled series #98112	- - - -	0 0 0	2019-03-20	Alistair Francis		New
1058784	[v5,4/4] boot/riscv-pk: Deprecate riscv-pk and BBL	[v5,1/4] board/qemu/riscv32-virt: Update linux config	- - - -	0 0 0	2019-03-20	Alistair Francis		New
1058785	[v5,2/4] boot/opensbi: new package	[v5,1/4] board/qemu/riscv32-virt: Update linux config	- - - -	0 0 0	2019-03-20	Alistair Francis		New
1058783	[v5,1/4] board/qemu/riscv32-virt: Update linux config	[v5,1/4] board/qemu/riscv32-virt: Update linux config	- - - -	0 0 0	2019-03-20	Alistair Francis		New
1058701	[v3,2/2] package/uftp: openssl is optional, not mandatory	[v3,1/2] package/uftp: fix openssl static linking	- - - -	0 0 0	2019-03-19	Fabrice Fontaine		New
1058700	[v3,1/2] package/uftp: fix openssl static linking	[v3,1/2] package/uftp: fix openssl static linking	- 1 - -	0 0 0	2019-03-19	Fabrice Fontaine		New
1058645	[1/1] mender-grubenv: new package	[1/1] mender-grubenv: new package	- - - -	0 0 0	2019-03-19	Adam Duskett		New
1058365	[RFC] openssh: add option to allow login as root	[RFC] openssh: add option to allow login as root	- - - -	0 0 0	2019-03-19	Esben Haabendal		New
1058357	[3/3] gummiboot: fix compatibility with newer glibc versions	Gummiboot fixup	- - - -	0 0 0	2019-03-19	Esben Haabendal		New
1058356	[2/3] gummiboot: upgrade to last commit before being removed	Gummiboot fixup	- - - -	0 0 0	2019-03-19	Esben Haabendal		New
1058355	[1/3] gummiboot: use new official upstream git repository	Gummiboot fixup	- - - -	0 0 0	2019-03-19	Esben Haabendal		New
1058140	[v2,8/8] support/testing: test_optee.py: test optee boot and testsuite	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New
1058139	[v2,7/8] testing: test can use the locally generated qemu host tool	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New
1058141	[v2,6/8] configs/qemu_armv7a_tz_virt: Armv7-A emulation with TrustZone services	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New
1058136	[v2,5/8] package/optee-test: fix dependency in TAs build	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New
1058137	[v2,4/8] boot/arm-trusted-firmware: support alternate image files	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New
1058138	[v2,3/8] boot/arm-trusted-firmware: support debug mode	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New
1058135	[v2,2/8] boot/arm-trusted-firmware: in-tree and OP-TEE BL32	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New
1058134	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	[v2,1/8] boot/arm-trusted-firmware: support 32bit Arm targets	- - - -	0 0 0	2019-03-18	Etienne Carriere		New

Automated build testing

- The enormous number of configuration options in Buildroot make it very difficult to test all combinations.
- Random configurations are therefore built 24/7 by multiple machines.
  - Random choice of architecture/toolchain combination from a pre-defined list
  - Random selection of packages using `make randpackageconfig`
  - Random enabling of features like static library only, or `BR2_ENABLE_DEBUG=y`
- Scripts and tools publicly available at https://git.buildroot.net/buildroot-test/
- Results visible at http://autobuild.buildroot.org/
- Daily e-mails with the build results of the past day

autobuild.buildroot.org

Buildroot tests

Date	Duration	Status	Commit ID	Submitter	Arch/Subarch	Failure reason	Libc	Static?	Data
2019-03-20 16:32:05	01:12:43	OK	master 83f7a923	Thomas Petazzoni *(gcc159)*	mips64el / mips64r6	none	glibc	N	dir, end log, config, defconfig
2019-03-20 16:30:42	04:02:55	OK	master 3033e83d	Thomas Petazzoni *(gcc159)*	arm / cortex-a8	none	glibc	N	dir, end log, config, defconfig
2019-03-20 16:30:27	25:32	OK	master f389df23	Yann E. MORIN	powerpc64le / power8	none	glibc	N	dir, end log, config, defconfig
2019-03-20 16:22:37	02:12:40	OK	master 3033e83d	Thomas Petazzoni *(gcc160)*	arc / archs	none	glibc	N	dir, end log, config, defconfig
2019-03-20 16:20:16	30:31	OK	master 83f7a923	Mark Corbin *(Embecosm Godzilla U18.10)*	riscv64	none	glibc	N	dir, end log, config, defconfig
2019-03-20 16:15:11	03:08:12	NOK	master 3033e83d	Yann E. MORIN	m68k / 68040	host-uboot-tools-2019.01	uclibc	N	dir, end log, config, defconfig
2019-03-20 16:05:02	02:47:01	NOK	master 3033e83d	Peter Korsgaard *(gcc112/ppc64le/Centos 7/gcc 4.8.5)*	arc / arc700	host-uboot-tools-2019.01	uclibc	N	dir, end log, config, defconfig
2019-03-20 16:04:03	31:27	OK	master 83f7a923	Yann E. MORIN	nios2	none	glibc	N	dir, end log, config, defconfig
2019-03-20 16:03:52	06:29:57	OK	master 3033e83d	Giulio Benetti *(Micronova srl Server)*	mipsel / mips32	none	uclibc	N	dir, end log, config, defconfig
2019-03-20 16:01:04	01:24:17	OK	master 3033e83d	Thomas Petazzoni *(gcc159)*	arc / archs	none	glibc	N	dir, end log, config, defconfig
2019-03-20 15:57:47	03:07:12	OK	master 3033e83d	Matt Weber *(U14.04 Sandboxed)*	xtensa	none	uclibc	N	dir, end log, config, defconfig
2019-03-20 15:50:20	22:35	OK	master 83f7a923	Andre Hentschel	arm / arm1176jzf-s	none	uclibc	N	dir, end log, config, defconfig
2019-03-20 15:49:56	01:13:49	NOK	master 3033e83d	Matt Weber *(U14.04 Sandboxed)*	arm / arm926ej-s	host-go-1.12.1	uclibc	Y	dir, end log, config, defconfig
2019-03-20 15:48:49	04:06:01	OK	master 3033e83d	Mark Corbin *(Embecosm Godzilla U18.10)*	riscv32	none	glibc	N	dir, end log, config, defconfig
2019-03-20 15:43:42	16:59	OK	2018.02.x 3a2c33cf	Thomas Petazzoni *(gcc160)*	powerpc / 603e	none	uclibc	N	dir, end log, config, defconfig
2019-03-20 15:37:11	56:16	OK	master 3033e83d	Yann E. MORIN	nios2	none	glibc	N	dir, end log, config, defconfig
2019-03-20 15:31:34	01:52:13	OK	2018.11.x 3e37abb3	Yann E. MORIN	xtensa	none	uclibc	N	dir, end log, config, defconfig
2019-03-20 15:26:00	01:02:47	OK	2018.02.x 3a2c33cf	Thomas Petazzoni *(gcc160)*	arm / arm926ej-s	none	uclibc	Y	dir, end log, config, defconfig
2019-03-20 15:25:55	01:46:38	OK	master 3033e83d	Andre Hentschel	mips64el / mips64r6	none	glibc	N	dir, end log, config, defconfig

Autobuild daily reports

```
Subject: [Buildroot] [autobuild.buildroot.net] Build results for 2019-03-19

Build statistics for 2019-03-19
===============================

      branch |  OK | NOK | TIM | TOT |
   2018.02.x |  18 |   3 |   0 |  21 |
   2018.11.x |  36 |   1 |   0 |  37 |
   2019.02.x |  25 |   4 |   0 |  29 |
      master | 166 | 105 |   3 | 274 |

Results for branch 'master'
===========================

Classification of failures by reason
------------------------------------

                 unknown | 22
      angularjs-legal-info | 15
   host-uboot-tools-2019.01 | 11
[...]

Detail of failures
------------------

     sparc | android-tools-4.2.2+git2013... | NOK | http://autobuild.buildroot.net/results/f1648f245d77f85661bc0d2f1e8097c3695206d8 |
   mips64el |           angularjs-legal-info | NOK | http://autobuild.buildroot.net/results/fdf6b64648dfa58ec74de31104a1a71248242d80 |
[...]
       arm |         glib-networking-2.58.0 | NOK | http://autobuild.buildroot.net/results/fc2e68921bd84d13d2e9bc900a91e46b08d698fe |
```

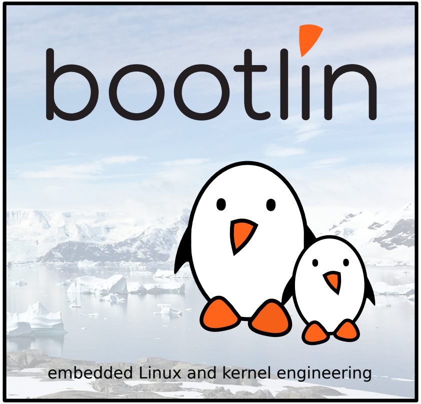
Additional testing effort

- Run-time test infrastructure in `support/testing`
  - Contains a number of test cases that verify that specific Buildroot configurations build correctly, and boot correctly under Qemu.
  - Validates filesystem format support, specific packages, core Buildroot functionality.
  - `./support/testing/run-tests -l`
  - `./support/testing/run-tests tests.fs.test_ext.TestExt2`
  - Run regularly on *Gitlab CI*
- All *defconfigs* in `configs/` are built every week on *Gitlab CI*

Acknowledgements

- Bootlin would like to thank the following members of the Buildroot community for their useful comments and reviews during the development of these training materials:
  - Thomas De Schampheleire
  - Peter Korsgaard
  - Yann E. Morin
  - Arnout Vandecappelle
  - Gustavo Zacarias

Last slides



Thank you!

And may the Source be with you

Rights to copy

Document sources: https://github.com/bootlin/training-materials/