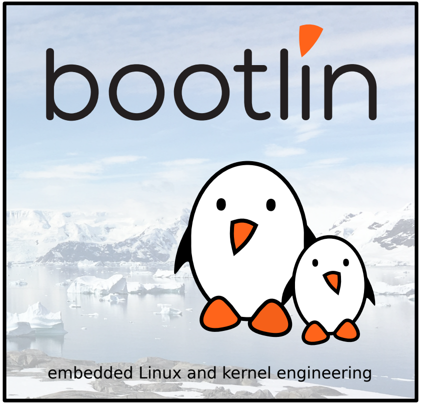
Advanced topics



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.

BR2_EXTERNAL: mechanism

- Specify, as a colon-separated list, the *external* directories in `BR2_EXTERNAL`
- 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))
```

Using 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 `.br-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

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

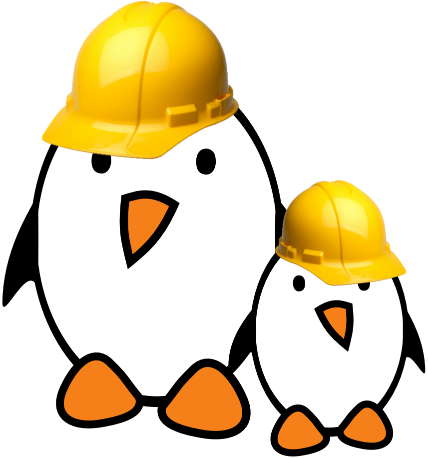
Support for top-level parallel build (2)

- 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

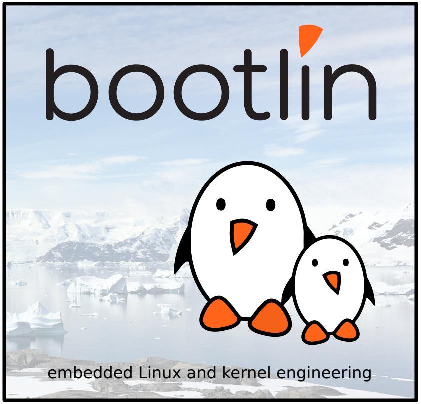
- Buildroot guarantees that for a given version/configuration, it will **always build the same components**, in the same version, with the same configuration.
- However, a number of aspects (time, user, build location) can affect the build and make two consecutive builds of the same configuration **not 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

Practical lab - Advanced aspects



- 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.)

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 $\rightarrow$ 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) • *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	**1.17.4**

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
```

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

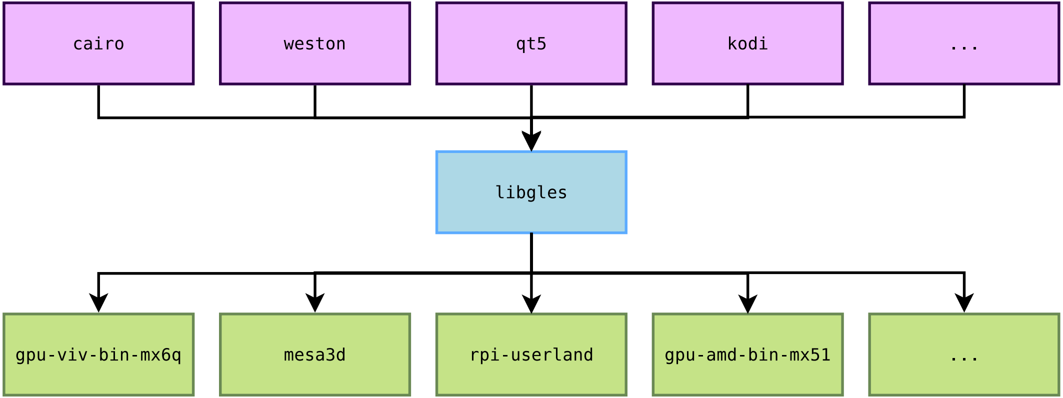
```
N:      Thomas Petazzoni <thomas.petazzoni@bootlin.com>
F:      arch/Config.in.arm
F:      boot/boot-wrapper-aarch64/
F:      boot/grub2/
F:      package/android-tools/
F:      package/cmake/
F:      package/cramfs/
[...]
F:      toolchain/

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

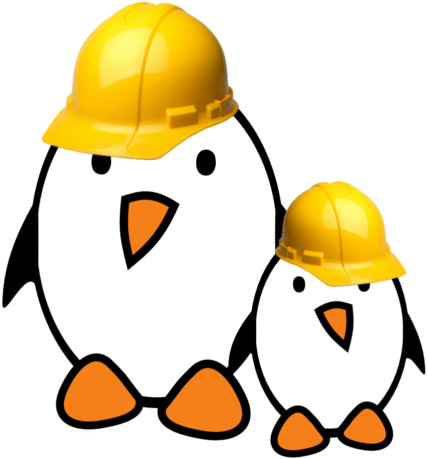
qt5/qt5base/Config.in

```
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

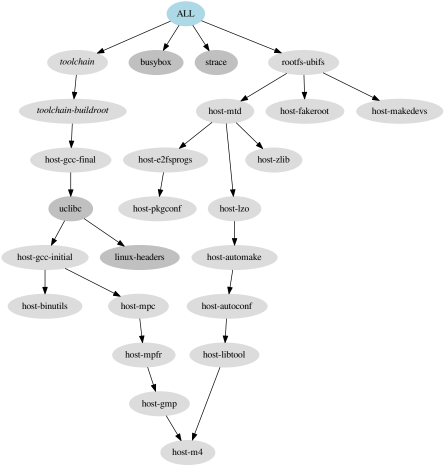
Analyzing the build: available tools

- 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/`

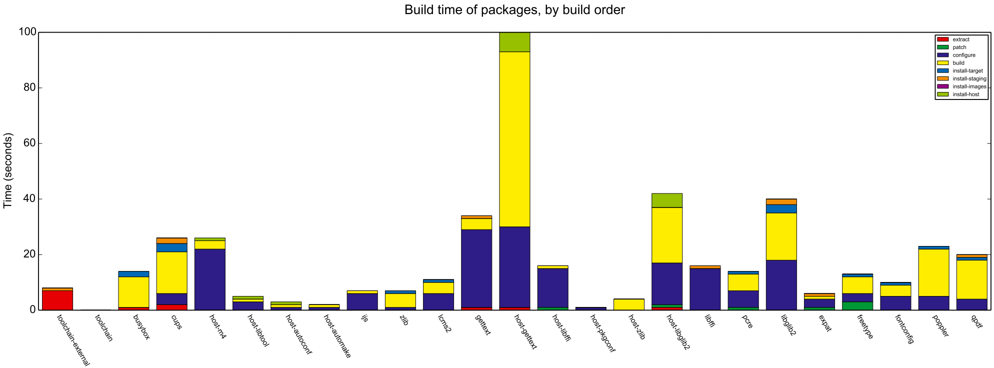
Dependency graph example



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