Buildroot system development training

embedded Linux and kernel engineering
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
About Bootlin

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
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](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
  - 8000 patches contributed
  - [https://bootlin.com/community/contributions/kernel-contributions/](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
  - 5000 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
Generic course information

Embedded Linux and kernel engineering

Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
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.
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.

kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
Prepare your lab environment

- Download and extract the lab archive
Introduction to Embedded Linux
Simplified Linux system architecture

Userspace
- Application
- Application
- C library

Linux kernel
- Task/memory management
- Networking
- Device drivers
- Filesystems

Bootloader

Hardware
Overall Linux boot sequence

1. **Bootloader**
   - Loads the DTB and kernel to RAM, starts the kernel

2. **Kernel**
   - Initializes hardware devices and kernel subsystems
   - Mounts the root filesystem indicated by root=
   - Starts the init application, /sbin/init by default

3. **/sbin/init**
   - Starts other user space services and applications

   - **Shell**
   - **Other applications**

   - Root filesystem
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

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

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](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)
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/](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 \textit{Kconfig}

- A choice of configuration interfaces:
  - \texttt{make menuconfig}
  - \texttt{make nconfig}
  - \texttt{make xconfig}
  - \texttt{make gconfig}

- Make sure to install the relevant libraries in your system (\textit{ncurses} for \texttt{menuconfig/nconfig}, \textit{Qt} for \texttt{xconfig}, \textit{Gtk} for \texttt{gconfig})
Main menuconfig menu

Buildroot 2024.02 Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is excluded

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

<select>  < Exit >  < Help >  < Save >  < Load >
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
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.
Practical lab - Basic Buildroot usage

- 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

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
Default build organization

- All the build output goes into a directory called `output/` within the top-level Buildroot source directory.
  - 0 = 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`
To start an out of tree build, two solutions:

- From the Buildroot source tree, simplify specify a \texttt{O=} variable:

\begin{verbatim}
make \texttt{O=}../foo-output/ menuconfig
\end{verbatim}

- From an empty output directory, specify \texttt{O=} and the path to the Buildroot source tree:

\begin{verbatim}
make -C ../buildroot/ \texttt{O=}$(pwd) menuconfig
\end{verbatim}

Once one out of tree operation has been done (\texttt{menuconfig}, loading a defconfig, etc.), Buildroot creates a small wrapper \texttt{Makefile} in the output directory.

This wrapper \texttt{Makefile} then avoids the need to pass \texttt{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
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
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:

```plaintext
BR2_arm=y
```

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

```plaintext
BR2_arm=y
BR2_PACKAGE_STRESS=y
```
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
make <foo>_defconfig
```

- It overwrites the current `.config`, if any

To create a *defconfig*, run:

```make
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.

```bash
$ make qemu_arm_vexpress_defconfig
$ make
```

- Additional instructions often available in `board/<boardname>`, e.g.: `board/qemu/arm-vexpress/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.

```plaintext
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

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
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
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.
Build tree
Build tree: $(O)

- output/

- Global output directory

- Can be customized for out-of-tree build by passing \(O=\langle\text{dir}\rangle\)

- Variable: 0 (as passed on the command line)

- Variable: BASE_DIR (as an absolute path)
Build tree: $(0)/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: $(0)/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: $(0)/staging

- output/
  - staging/
  - Just a symbolic link to the sysroot, i.e. to host/<tuple>/sysroot/.
  - Available for convenience
Build tree: $(0)/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: $(0)/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: $(0)/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: $(0)/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

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 an existing pre-built toolchain
- Selected from **Toolchain → Toolchain type**.
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 \((\geq 10 \text{ MB})\)
  - uClibc-ng and musl, smaller C libraries. uClibc-ng supports non-MMU architectures. Good for very small systems \((< 10 \text{ MB})\).
- 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

<table>
<thead>
<tr>
<th>Toolchain type (External toolchain) ---&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>*** Toolchain External Options ***</td>
</tr>
<tr>
<td>Toolchain (Custom toolchain) ---&gt;</td>
</tr>
<tr>
<td>Toolchain origin (Toolchain to be downloaded and installed) ---&gt;</td>
</tr>
<tr>
<td>(<a href="http://autobuild.buildroot.org/toolchains/tarballs/br-1386-pentium4-full-2020.11.2.tar.bz2">http://autobuild.buildroot.org/toolchains/tarballs/br-1386-pentium4-full-2020.11.2.tar.bz2</a>) Toolchain URL</td>
</tr>
<tr>
<td>(bin) Toolchain relative binary path (NEW)</td>
</tr>
<tr>
<td>($ARCH)-linux Toolchain prefix (NEW)</td>
</tr>
<tr>
<td>External toolchain gcc version (9.x) ---&gt;</td>
</tr>
<tr>
<td>External toolchain kernel headers series (4.4.x) ---&gt;</td>
</tr>
<tr>
<td>External toolchain C library (uclibc/uclibc-ng) ---&gt;</td>
</tr>
<tr>
<td>--- Toolchain has WCHAR support?</td>
</tr>
<tr>
<td>[*] Toolchain has locale support?</td>
</tr>
<tr>
<td>[ ] Toolchain has threads support? (NEW)</td>
</tr>
<tr>
<td>[*] Toolchain has NPTL threads support? (NEW)</td>
</tr>
<tr>
<td>[ ] Toolchain has SSP support? (NEW)</td>
</tr>
<tr>
<td>[ ] Toolchain has RPC support? (NEW)</td>
</tr>
<tr>
<td>[*] Toolchain has C++ support?</td>
</tr>
<tr>
<td>[ ] Toolchain has D support? (NEW)</td>
</tr>
<tr>
<td>[ ] Toolchain has Fortran support? (NEW)</td>
</tr>
<tr>
<td>[ ] Toolchain has OpenMP support? (NEW)</td>
</tr>
<tr>
<td>[ ] Copy gdb server to the Target (NEW)</td>
</tr>
<tr>
<td>*** Host GDB Options ***</td>
</tr>
<tr>
<td>[ ] Build cross gdb for the host (NEW)</td>
</tr>
<tr>
<td>*** Toolchain Generic Options ***</td>
</tr>
<tr>
<td>() Extra toolchain libraries to be copied to target (NEW)</td>
</tr>
<tr>
<td>() Target Optimizations (NEW)</td>
</tr>
<tr>
<td>() Target linker options (NEW)</td>
</tr>
<tr>
<td>[ ] Register toolchain within Eclipse Buildroot plug-in (NEW)</td>
</tr>
</tbody>
</table>

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

1. Copy the skeleton to $(TARGET_DIR)$
2. Build/install all packages
3. Run a number of cleanup steps
4. Copy rootfs overlays
5. Execute post-image scripts
6. Create rootfs images
7. Execute post-build scripts
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

All target packages

skeleton virtual package

skeleton-init-sysv
skeleton-init-systemd
skeleton-init-openrc
skeleton-init-none

skeleton-init-common

BR2_ROOTFS_SKELETON_DEFAULT=y

BR2_INIT_SYSV=y
BR2_INIT_BUSYBOX=y

BR2_INIT_SYSTEMD=y

BR2_INIT_OPENRC=y

BR2_INIT_NONE=y

BR2_ROOTFS_SKELETON_CUSTOM=y
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.
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.

```bash
$ 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**
```bash
#!/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		/applog	jffs2	defaults		0	0" > /
   $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.
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_ROOTFSDEVICE_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

<table>
<thead>
<tr>
<th></th>
<th>type</th>
<th>mode</th>
<th>uid</th>
<th>gid</th>
<th>major</th>
<th>minor</th>
<th>start</th>
<th>inc</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>dev</td>
<td>d</td>
<td>755</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>tmp</td>
<td>d</td>
<td>1777</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>var/www</td>
<td>d</td>
<td>755</td>
<td>33</td>
<td>33</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</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**

<table>
<thead>
<tr>
<th>#</th>
<th>&lt;name&gt;</th>
<th>&lt;type&gt;</th>
<th>&lt;mode&gt;</th>
<th>&lt;uid&gt;</th>
<th>&lt;gid&gt;</th>
<th>&lt;major&gt;</th>
<th>&lt;minor&gt;</th>
<th>&lt;start&gt;</th>
<th>&lt;inc&gt;</th>
<th>&lt;count&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>/dev/mem</td>
<td>c</td>
<td>640</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>/dev/kmem</td>
<td>c</td>
<td>640</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>/dev/i2c-</td>
<td>c</td>
<td>666</td>
<td>0</td>
<td>0</td>
<td>89</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</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

<table>
<thead>
<tr>
<th>username</th>
<th>uid</th>
<th>group</th>
<th>gid</th>
<th>password</th>
<th>home</th>
<th>shell</th>
<th>groups</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>-1</td>
<td>bar</td>
<td>-1</td>
<td>!=blabla</td>
<td>/home/foo</td>
<td>/bin/sh</td>
<td>alpha,bravo</td>
<td>Foo user</td>
</tr>
<tr>
<td>test</td>
<td>8000</td>
<td>wheel</td>
<td>-1</td>
<td>=</td>
<td>/bin/sh</td>
<td>-</td>
<td></td>
<td>Test user</td>
</tr>
</tbody>
</table>
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.
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

```bash
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
```
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.
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.
Practical lab - Root filesystem construction

- 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
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.
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`
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.
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.
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 vending 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
Introduction

- 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.nostarch.com/gnumake](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**

```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 ... endif`:

```plaintext
define FOOBAR
line 1
line 2
endif
```

Make variables are referenced using the `$(FOOBAR)` syntax.
With \texttt{ifeq} or \texttt{ifneq}

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

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

With the \texttt{\$(if ...)} make function:

```bash
HOSTAPD_LIBS += $(if $(BR2_STATIC_LIBS),-lcrypto -lz)
```
Defining and using functions

Defining a function is exactly like defining a variable:

```bash
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\n$(1):\n$(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

```bash
$(BUILD_DIR)/%.stamp_extracted:
    [...] @$(call MESSAGE,"Extracting")
declare 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 \n        $(STAGING_DIR)/usr/include/$\$(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 $$\text{name}$$.

package/pppd/pppd.mk

```matlab
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

Integrating new packages in Buildroot

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
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.
Integrating new packages in Buildroot

Config.in file
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>`.

```c
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.
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.

```python
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* selects 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**!
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 across *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
    # libn1, 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
[[...]]
```
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>`.
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
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.
Integrating new packages in Buildroot

Package infrastructures
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

openvc4  kodi  libgtk2  gnutls  systemd  libglib2

cmake-package  autotools-package  meson-package

generic-package

kconfig-package  perl-package  python-package

busybox  uboot  perl-net-ssl  perl-xml-libxml  python-serial  python-zeroconf

https://bootlin.com
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

1. Download
2. Extract
3. Patch
4. Configure
5. Build
6. Install

Implemented by the generic-package infrastructure

Implemented by the package itself
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.
Integrating new packages in Buildroot

.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 assumptions 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, $(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
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:

```make
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:

```make
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
  - `bzx`, 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+squeezel.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\}
  ```
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.
package/perl/perl.hash

# Hashes from: https://www.cpan.org/src/5.0/perl-5.32.1.tar.xz.{md5,sha1,sha256}.txt
md5 7f104064b906aad8c7329ca5e409a32d7 perl-5.32.1.tar.xz
sha1 1fb4f710d139daelea3e1fa4eaba21fcaa8e18e perl-5.32.1.tar.xz
sha256 57cc47c735c83008ce2fa0643507b44c4ae59012bfdad012131d6b39e02309 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 dd90d4f42e44cad5a7c09eea0189d93c7b37ae560c91f0f6d5233ed3b9292a2 Artistic
sha256 d77d235e41d54594865151f4751e835c5a82322b0e87ace266567c3391a4b912 Copying
sha256 df6ad59aefea68676c38325f25f670f026ddde6c71291b2ca231b6247859907 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 0e7d44ca9c153d96a9b5f12644fb35a632537a5a7f653792b72e53d9d5c2db ipset-7.6.tar.bz2
# Locally calculated
sha256 231f7edcc7352d7734a96ee0b8030f77982678c516876fcb81e25b32d68564c 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

```plaintext
config BR2_PACKAGE_NTP
    bool "ntp"
    select BR2_PACKAGE_LIBEVENT

[...]
```

package/ntp/ntp.mk

```plaintext
[...]
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
[...]```
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.
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.
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; \n    $(TARGET_CONFIGURE_ARGS) \n    $(TARGET_CONFIGURE_OPTS) \n    CFLAGS="$(TARGET_CFLAGS) $(LIBZLIB_PIC)" \n    ./configure \n    $(LIBZLIB_SHARED) \n    --prefix=/usr \n)
endef

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

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

define LIBZLIB_INSTALL_TARGET_CMDS
    $(TARGET_MAKE_ENV) $(MAKE) -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
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

1. Download
2. Extract
3. Patch
4. Configure
5. Build
6. Install
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`. 
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=$(STAGING_DIR)
else
  GNUPG2_CONF_OPTS += --disable-bzip2
endif
ifeq ($(BR2_PACKAGE_LIBUSB),y)
  GNUPG2_CONF_ENV += CPPFLAGS="$(TARGET_CPPFLAGS) -I$(STAGING_DIR)/usr/include/libusb-1.0"
  GNUPG2_CONF_OPTS += --enable-ccid-driver
else
  GNUPG2_CONF_OPTS += --disable-ccid-driver
endif
ifeq ($(BR2_PACKAGE_READLINE),y)
  GNUPG2_CONF_OPTS += --with-readline=$(STAGING_DIR)
else
  GNUPG2_CONF_OPTS += --without-readline
endif
$(eval $(autotools-package))
Integrating new packages in Buildroot

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_`
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

```make
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

```c
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

```make
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 lab - New packages in Buildroot

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

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
Advanced package aspects

Licensing report
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:

```bash
$ 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/](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_REDISIBUTE = NO
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.
<table>
<thead>
<tr>
<th>Package</th>
<th>Target</th>
<th>CPE Identifier</th>
<th>CVE ID</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>package/libao/libao.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>1.2.0</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libcue/libcue.mk</td>
<td>cmake</td>
<td>Yes</td>
<td>2.2.1</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libebur128/libebur128.mk</td>
<td>cmake</td>
<td>Yes</td>
<td>1.2.4</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libffi/libffi.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>3.3</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target + host</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libglib2/libglib2.mk</td>
<td>meson</td>
<td>Yes</td>
<td>2.66.7</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target + host</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libid3tag/libid3tag.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>0.15.1b</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/liblilo/liblilo.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>0.31</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libmad/libmad.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>0.15.1b</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libmodplug/libmodplug.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>0.8.9.0</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libmpd/libmpd.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>11.8.17</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>package/libtool/libtool.mk</td>
<td>autotools</td>
<td>Yes</td>
<td>2.4.6</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>target + host</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Packages affected by CVEs: 5
Total number of CVEs affecting all packages: 5
Packages with CPE ID: 13
Packages without CPE ID: 30
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*
## Known Affected Software Configurations

**Configuration 1**

<table>
<thead>
<tr>
<th>CPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpe:2.3:a:cairographics:cairo:<em>:</em>:<em>:</em>:<em>:</em>:*</td>
<td>Up to (excluding) 1.17.4</td>
</tr>
</tbody>
</table>

Showing 10 of 50 matching CPE(s) for the range. [View All CPEs here](https://nvd.nist.gov/vuln/detail/CVE-2020-35492)
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
There are cases where a CVE reported by the \textit{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 \texttt{<pkg>_IGNORE_CVES} variable allows a package to tell \textit{pkg-stats} to ignore a particular CVE

\begin{verbatim}
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
\end{verbatim}
Patching packages
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
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.
From 81289d1d1ada5a767a4b4d1309c286468cf37f 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
     
     ngx_size=
     -cat << END > $NGX_AUTOTEST.c
     +cat << _EOF > $NGX_AUTOTEST.c
     [...]
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
[...]
```
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](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
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
```
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:

```plaintext
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
```
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:

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

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

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.
bind.mk

define BIND_INSTALL_INIT_SYSV
  \n  $(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
Advanced package aspects

Config scripts
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`. 
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
**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
Buildroot \textit{package infrastructure} often implement a default behavior for certain steps:

- \texttt{generic-package} implements for all packages the download, extract and patch steps
- Other infrastructures such as \texttt{autotools-package} or \texttt{cmake-package} also implement the configure, build and installations steps

In some situations, the package may want to do \textbf{additional actions} before or after one of these steps.

The \textbf{hook} mechanism allows packages to add such custom actions.
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.
bind.mk: remove unneeded binaries

```makefile
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

```makefile
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
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).
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 TFTP_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.
Advanced package aspects

DEVELOPERS file
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/
F: package/glibc/

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

cairo
weston
qt5
kodi
...

libgles

gpu-viv-bin-mx6q
mesa3d
rpi-userland
gpu-amd-bin-mx51
...

https://bootlin.com
libgles/Config.in

```c
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 depend 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.
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

```plaintext
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

```plaintext
[...
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

```plaintext
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

```plaintext
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
```
- 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
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

Build time of packages, by build order

- Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
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 per package

Total filesystem size: 3156 kB

- toolchain-external (1462 kB) - 46.3%
- busybox (686 kB) - 21.8%
- lua (262 kB) - 8.3%
- ncurses (198 kB) - 6.3%
- strace (289 kB) - 9.2%
- libhid (52 kB) - 1.7%
- htop (100 kB) - 3.2%
- libusb (71 kB) - 2.3%
- Other (32 kB) - 1.1%
- libusb (52 kB) - 1.7%
- Other (32 kB) - 1.1%
- Total filesystem size: 3156 kB
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
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`
BR2EXTERNAL: 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
```
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
```
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**

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

```bash
$ 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://www.busybox.net/downloads",
          "http://sources.buildroot.net/busybox",
          "http://sources.buildroot.net/busybox"
        ]
      }
    ],
    "dependencies": [
      "host-skeleton",
      "host-tar",
      "skeleton",
      "toolchain"
    ],
    "reverse_dependencies": []
  }
}
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.
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
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

© Copyright 2004-2024, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!
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.
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...
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.
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

With this file, building *CMake* projects outside of Buildroot is easy:

```bash
$ 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](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...
```
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

```bash
$ 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='./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

- Make a change in 
  $(TOPDIR)/../custom-app/

- make custom-app-rebuild all

- Rsync code from
  $(TOPDIR)/../custom-app/
  to
  $(O)/output/build/custom-app-custom/

- Re-run "make"
  in custom-app sources
  Rebuilds only what changed

- Recreates the root
  filesystem image

- Re-run "make install"
  in custom-app sources
  Reinstalls to $(TARGET_DIR)

- Test !
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` values are specified in a package override file, configured in `BR2_PACKAGE_OVERRIDE_FILE`, by default `$\text{CONFIG\_DIR}/local.mk$`.

**Example** local.mk

```
LIBPNG_OVERRIDE_SRCDIR = $(HOME)/projects/libpng
LINUX_OVERRIDE_SRCDIR = $(HOME)/projects/linux
```
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.
With `BR2_ENABLE_DEBUG=y` and `BR2_STRIP_strip=y`
- get debugging symbols in `$\text{STAGING\_DIR}$` for libraries, and in the build directories for everything.
- stripped binaries in `$\text{TARGET\_DIR}$`
- Appropriate for **remote debugging**

With `BR2_ENABLE_DEBUG=y` and `BR2_STRIP_strip` disabled
- debugging symbols in both `$\text{STAGING\_DIR}$` and `$\text{TARGET\_DIR}$`
- appropriate for **on-target debugging**
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.
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.
Practical lab - Application development

- 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)
```
When you run `make`...

- `all`: world
- `world`: target-post-image
- `target-post-image`: `$(TARGETS_ROOTFS)` target-finalize and calls the post image scripts
- `$(TARGETS_ROOTFS)` contains the list of root filesystem image types to generate
- `target-finalize`: `$(PACKAGES)` and does the root filesystem finalization (remove headers, remove doc, stripping, copy rootfs overlays, run post-build scripts, etc.)
- `$(PACKAGES)` contains the list of packages to build, including kernel, toolchain, bootloaders, and all user space libraries/applications

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

```makefile
# 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

enddef # 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
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

```bash
# 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)
    $(Q)mkdir -p $(@)
    $(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

```bash
# 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:

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

▶ Packages built for the host:

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

▶ In package/libzlib/libzlib.mk:

```bash
LIBZLIB... = ...
$(eval $(generic-package))
$(eval $(host-generic-package))
```

▶ Leads to:

```bash
$(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

```bash
$(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

```bash
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

...  
```
Adding the **skeleton** and **toolchain** dependencies to target packages. Except for some specific packages (e.g. C library).
inner-generic-package: stamp files

- Defines shortcuts to reference the stamp files

- Pass variables to the stamp file targets, especially PKG
inner-generic-package: sequencing

$(1)-install: $(1)-install

ifeq ($(2)_INSTALL_TARGET),YES)
$$($(2)_TARGET_INSTALL_TARGET): $$($(2)_TARGET_INSTALL_TARGET)
endif

ifeq ($(2)_INSTALL_STAGING),YES)
$$($(2)_TARGET_INSTALL_STAGING): $$($(2)_TARGET_INSTALL_STAGING)
endif

ifeq ($(2)_INSTALL_IMAGES),YES)
$$($(2)_TARGET_INSTALL_IMAGES): $$($(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)

ifeq ($(2)_TARGET_BUILD),YES)
$$($(2)_TARGET_CONFIGURE): $$($(2)_TARGET_BUILD)
endif

ifeq ($(2)_TARGET_PATCH),YES)
$$($(2)_TARGET_PATCH): $$($(2)_TARGET_BUILD)
endif

ifeq ($(2)_TARGET_EXTRACT),YES)
$$($(2)_TARGET_EXTRACT): $$($(2)_TARGET_BUILD)
endif

$(1)-source: $$($(2)_TARGET_SOURCE)
$$($(2)_TARGET_SOURCE): $$($(2)_TARGET_BUILD)

ifeq ($(2)_FINAL_DOWNLOAD_DEPENDENCIES),YES)
$$(2)_TARGET_SOURCE): prepare
endif

ifeq ($(2)_FINAL_DOWNLOAD_DEPENDENCIES),YES)
$$(2)_TARGET_SOURCE): dependencies
endif
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

```bash
$(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))
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) 
   -DMAKE_TOOLCHAIN_FILE="$$($(HOST_DIR)/share/buildroot/toolchainfile.cmake" 
   ...
   $$($(PKG)_CONF_OPTS) 
)
endif
else
define $(2)_CONFIGURE_CMDS
... host case ...
endif
endif
$(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)
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
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.
toolchain/toolchain-external/toolchain-external-arm-aarch64/toolchain-external-arm-aarch64.mk

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

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

$(eval $(toolchain-external-package))
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.
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
ROOTFS_COMMON_DEPENDENCIES = \ 
    host-fakeroott host-makedevs \ 
    $(BR2_TAR_HOST_DEPENDENCY) \ 
    $(if $(PACKAGES_USERS)$\(\text{ROOTFS\_USERS\_TABLES}\),host-mkpasswd)

rootfs-common: $(ROOTFS_COMMON_DEPENDENCIES) target-finalize
    @$(call MESSAGE,"\Generating root filesystems common tables")
    rm -rf $(FS\_DIR)
    mkdir -p $(FS\_DIR)
    $(call PRINTF,$\(\text{PACKAGES\_USERS}\)) \(\text{ROOTFS\_FULL\_USERS\_TABLE}\)
    cat $(ROOTFS\_USERS\_TABLES) >> $\(\text{ROOTFS\_FULL\_USERS\_TABLE}\)
    $(call PRINTF,$\(\text{PACKAGES\_PERMISIONS\_TABLE}\)) \(\text{ROOTFS\_FULL\_DEVICES\_TABLE}\)
    cat $(ROOTFS\_DEVICE\_TABLES) >> $\(\text{ROOTFS\_FULL\_DEVICES\_TABLE}\)
    $(call PRINTF,$\(\text{PACKAGES\_DEVICES\_TABLE}\)) >> $\(\text{ROOTFS\_FULL\_DEVICES\_TABLE}\)
```makefile
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
```
$$\text{(BINARIES_DIR)}/$$\text{(ROOTFS_$(2)_FINAL_IMAGE_NAME)}: \text{$$\text{(ROOTFS_$(2)_DEPENDENCIES)}\}}$$

```makefile
$@ \$(call MESSAGE,"Generating filesystem image $$\text{(ROOTFS_$(2)_FINAL_IMAGE_NAME)}")
```
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))
Buildroot community: support and contribution
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.
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/
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/](https://git.buildroot.net/buildroot-test/)
- Results visible at [http://autobuild.buildroot.org/](http://autobuild.buildroot.org/)
- Daily e-mails with the build results of the past day
<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Duration</th>
<th>Status</th>
<th>Commit ID</th>
<th>Submitter</th>
<th>Arch/Subarch</th>
<th>Failure reason</th>
<th>Lic</th>
<th>Static?</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-03-20</td>
<td>16:32:05</td>
<td>01:12:43</td>
<td>OK</td>
<td>master</td>
<td>Thomas Petzanzoni</td>
<td>mips64el / mips64e</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:30:42</td>
<td>04:02:55</td>
<td>OK</td>
<td>master</td>
<td>Thomas Petzanzoni</td>
<td>arm / cortex-a7</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:30:27</td>
<td>25:32</td>
<td>OK</td>
<td>master</td>
<td>Yann E. MORIN</td>
<td>powerpc64le / poweri6</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:22:37</td>
<td>02:12:40</td>
<td>OK</td>
<td>master</td>
<td>Thomas Petzanzoni</td>
<td>arm / arches</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:20:16</td>
<td>30:31</td>
<td>OK</td>
<td>master</td>
<td>Mark Corbin</td>
<td>riscv64</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:15:11</td>
<td>03:08:12</td>
<td>NOK</td>
<td>master</td>
<td>Yann E. MORIN</td>
<td>m68k / 8040</td>
<td>host-shoot-tools-2019.01</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:05:02</td>
<td>02:47:01</td>
<td>NOK</td>
<td>master</td>
<td>Peter Korparaard</td>
<td>arm / arches</td>
<td>host-shoot-tools-2019.01</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:04:03</td>
<td>31:27</td>
<td>OK</td>
<td>master</td>
<td>Yann E. MORIN</td>
<td>nios2</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:03:52</td>
<td>06:29:57</td>
<td>OK</td>
<td>master</td>
<td>Giusto Benetti</td>
<td>mips64el / mips32</td>
<td>none</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>16:01:04</td>
<td>01:24:17</td>
<td>OK</td>
<td>master</td>
<td>Thomas Petzanzoni</td>
<td>arm / arches</td>
<td>none</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:57:47</td>
<td>03:07:12</td>
<td>OK</td>
<td>master</td>
<td>Matt Weber</td>
<td>xtensa</td>
<td>none</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:50:20</td>
<td>22:35</td>
<td>OK</td>
<td>master</td>
<td>Andre Hentschel</td>
<td>arm / arm1176p2s</td>
<td>none</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:49:56</td>
<td>01:13:49</td>
<td>NOK</td>
<td>master</td>
<td>Matt Weber</td>
<td>arm / arm926ej-s</td>
<td>host-go.1.12.1</td>
<td>uclibc</td>
<td>Y</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:48:49</td>
<td>04:06:01</td>
<td>OK</td>
<td>master</td>
<td>Mark Corbin</td>
<td>riscv32</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:43:42</td>
<td>16:59</td>
<td>OK</td>
<td>master</td>
<td>Thomas Petzanzoni</td>
<td>powerpc / 603e</td>
<td>none</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:37:11</td>
<td>56:16</td>
<td>OK</td>
<td>master</td>
<td>Yann E. MORIN</td>
<td>nios2</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:31:34</td>
<td>01:52:13</td>
<td>OK</td>
<td>master</td>
<td>Yann E. MORIN</td>
<td>xtensa</td>
<td>none</td>
<td>uclibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:26:00</td>
<td>01:02:47</td>
<td>OK</td>
<td>master</td>
<td>Thomas Petzanzoni</td>
<td>arm / arm926ej-s</td>
<td>none</td>
<td>uclibc</td>
<td>Y</td>
<td>dir, end log, config, defconfig</td>
</tr>
<tr>
<td>2019-03-20</td>
<td>15:25:55</td>
<td>01:46:38</td>
<td>OK</td>
<td>master</td>
<td>Andre Hentschel</td>
<td>mips64el / mips64e</td>
<td>none</td>
<td>glibc</td>
<td>N</td>
<td>dir, end log, config, defconfig</td>
</tr>
</tbody>
</table>
Subject: [Buildroot] [autobuild.buildroot.net] Build results for 2019-03-19

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

<table>
<thead>
<tr>
<th>branch</th>
<th>OK</th>
<th>NOK</th>
<th>TIM</th>
<th>TOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2018.02.x</td>
<td>18</td>
<td>3</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>2018.11.x</td>
<td>36</td>
<td>1</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>2019.02.x</td>
<td>25</td>
<td>4</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>master</td>
<td>166</td>
<td>105</td>
<td>3</td>
<td>274</td>
</tr>
</tbody>
</table>

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 *GitHub CI*

- All `defconfigs` in `configs/` are built every week on *GitHub CI*
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
Thank you!
And may the Source be with you
Rights to copy

© Copyright 2004-2024, Bootlin
License: Creative Commons Attribution - Share Alike 3.0
https://creativecommons.org/licenses/by-sa/3.0/legalcode
You are free:
  ▶ to copy, distribute, display, and perform the work
  ▶ to make derivative works
  ▶ to make commercial use of the work

Under the following conditions:
  ▶ **Attribution.** You must give the original author credit.
  ▶ **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
  ▶ For any reuse or distribution, you must make clear to others the license terms of this work.
  ▶ Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

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