The Buildroot user manual

Table of Contents

I. Getting started
- 1. About Buildroot
- 2. System requirements
  - 2.1. Mandatory packages
  - 2.2. Optional packages
- 3. Getting Buildroot
- 4. Buildroot quick start
- 5. Community resources

II. User guide
- 6. Buildroot configuration
  - 6.1. Cross-compilation toolchain
  - 6.2. /dev management
  - 6.3. init system
- 7. Configuration of other components
- 8. General Buildroot usage
  - 8.1. *make* tips
  - 8.2. Understanding when a full rebuild is necessary
  - 8.3. Understanding how to rebuild packages
  - 8.4. Offline builds
  - 8.5. Building out-of-tree
  - 8.6. Environment variables
  - 8.7. Dealing efficiently with filesystem images
  - 8.8. Details about packages
  - 8.9. Graphing the dependencies between packages
  - 8.10. Graphing the build duration
  - 8.11. Graphing the filesystem size contribution of packages
  - 8.12. Top-level parallel build
  - 8.13. Advanced usage
- 9. Project-specific customization
  - 9.1. Recommended directory structure
  - 9.2. Keeping customizations outside of Buildroot
  - 9.3. Storing the Buildroot configuration
  - 9.4. Storing the configuration of other components
  - 9.5. Customizing the generated target filesystem
  - 9.6. Adding custom user accounts
  - 9.7. Customization *after* the images have been created
  - 9.8. Adding project-specific patches
  - 9.9. Adding project-specific packages
  - 9.10. Quick guide to storing your project-specific customizations
- 10. Integration topics
  - 10.1. Systemd
  - 10.2. Using SELinux in Buildroot
- 11. Frequently Asked Questions & Troubleshooting
  - 11.1. The boot hangs after *Starting network…*
  - 11.2. Why is there no compiler on the target?
  - 11.3. Why are there no development files on the target?
  - 11.4. Why is there no documentation on the target?
  - 11.5. Why are some packages not visible in the Buildroot config menu?
  - 11.6. Why not use the target directory as a chroot directory?
  - 11.7. Why doesn’t Buildroot generate binary packages (.deb, .ipkg…)?
  - 11.8. How to speed-up the build process?
- 12. Known issues
- 13. Legal notice and licensing
  - 13.1. Complying with open source licenses
  - 13.2. Complying with the Buildroot license
- 14. Beyond Buildroot
  - 14.1. Boot the generated images
  - 14.2. Chroot

III. Developer guide

15. How Buildroot works
16. Coding style
16.1. `Config.in` file
16.2. The `.mk` file
16.3. The `genimage.cfg` file
16.4. The documentation
16.5. Support scripts
17. Adding support for a particular board
18. Adding new packages to Buildroot
18.1. Package directory
18.2. Config files
18.3. The `.mk` file
18.4. The `.hash` file
18.5. The `SNNfoo` start script
18.6. Infrastructure for packages with specific build systems
18.7. Infrastructure for autotools-based packages
18.8. Infrastructure for CMake-based packages
18.9. Infrastructure for Python packages
18.10. Infrastructure for LuaRocks-based packages
18.11. Infrastructure for Perl/CPAN packages
18.12. Infrastructure for virtual packages
18.13. Infrastructure for packages using kconfig for configuration files
18.14. Infrastructure for rebar-based packages
18.15. Infrastructure for Waf-based packages
18.16. Infrastructure for Meson-based packages
18.17. Infrastructure for Cargo-based packages
18.18. Infrastructure for Go packages
18.19. Infrastructure for QMake-based packages
18.20. Infrastructure for packages building kernel modules
18.21. Infrastructure for asciidoc documents
18.22. Infrastructure specific to the Linux kernel package
18.23. Hooks available in the various build steps
18.24. Gettext integration and interaction with packages
18.25. Tips and tricks
18.26. Conclusion
19. Patching a package
19.1. Providing patches
19.2. How patches are applied
19.3. Format and licensing of the package patches
19.4. Integrating patches found on the Web
20. Download infrastructure
21. Debugging Buildroot
22. Contributing to Buildroot
22.1. Reproducing, analyzing and fixing bugs
22.2. Analyzing and fixing autobuild failures
22.3. Reviewing and testing patches
22.4. Work on items from the TODO list
22.5. Submitting patches
22.6. Reporting issues/bugs or getting help
22.7. Using the runtime tests framework
23. DEVELOPERS file and get-developers
24. Release Engineering
24.1. Releases
24.2. Development
IV. Appendix
25. Makedev syntax documentation
26. Makeusers syntax documentation
26.1. Caveat with automatic UIDs and GIDs
27. Migrating from older Buildroot versions
27.1. General approach
27.2. Migrating to 2016.11
27.3. Migrating to 2017.08

List of Examples

18.1. Config script: *divine* package
18.2. Config script: *imagemagick* package:

Buildroot 2023.02 manual generated on 2023-03-12 20:24:19 UTC from git revision 8cca1e6de1

The Buildroot manual is written by the Buildroot developers. It is licensed under the GNU General Public License, version 2. Refer to the COPYING file in the Buildroot sources for the full text of this license.

Copyright © The Buildroot developers <buildroot@buildroot.org>

Part I. Getting started

Chapter 1. About Buildroot

Buildroot is a tool that simplifies and automates the process of building a complete Linux system for an embedded system, using cross-compilation.

In order to achieve this, Buildroot is able to generate a cross-compilation toolchain, a root filesystem, a Linux kernel image and a bootloader for your target. Buildroot can be used for any combination of these options, independently (you can for example use an existing cross-compilation toolchain, and build only your root filesystem with Buildroot).

Buildroot is useful mainly for people working with embedded systems. Embedded systems often use processors that are not the regular x86 processors everyone is used to having in his PC. They can be PowerPC processors, MIPS processors, ARM processors, etc.

Buildroot supports numerous processors and their variants; it also comes with default configurations for several boards available off-the-shelf. Besides this, a number of third-party projects are based on, or develop their BSP [1] or SDK [2] on top of Buildroot.

---

[1] BSP: Board Support Package

[2] SDK: Software Development Kit

Chapter 2. System requirements

Buildroot is designed to run on Linux systems.

While Buildroot itself will build most host packages it needs for the compilation, certain standard Linux utilities are expected to be already installed on the host system. Below you will find an overview of the mandatory and optional packages (note that package names may vary between distributions).

2.1. Mandatory packages

- Build tools:
  - `which`
  - `sed`
  - `make` (version 3.81 or any later)
  - `binutils`
  - `build-essential` (only for Debian based systems)
  - `diffutils`
  - `gcc` (version 4.8 or any later)
  - `g++` (version 4.8 or any later)
  - `bash`

- `patch`
- `gzip`
- `bzip2`
- `perl` (version 5.8.7 or any later)
- `tar`
- `cpio`
- `unzip`
- `rsync`
- `file` (must be in `/usr/bin/file`)
- `bc`
- `findutils`

- Source fetching tools:
  - `wget`

2.2. Optional packages

- Recommended dependencies:

  Some features or utilities in Buildroot, like the legal-info, or the graph generation tools, have additional dependencies. Although they are not mandatory for a simple build, they are still highly recommended:

  - `python` (version 2.7 or any later)

- Configuration interface dependencies:

  For these libraries, you need to install both runtime and development data, which in many distributions are packaged separately. The development packages typically have a *-dev* or *-devel* suffix.

  - `ncurses5` to use the *menuconfig* interface
  - `qt5` to use the *xconfig* interface
  - `glib2`, `gtk2` and `glade2` to use the *gconfig* interface

- Source fetching tools:

  In the official tree, most of the package sources are retrieved using `wget` from *ftp*, *http* or *https* locations. A few packages are only available through a version control system. Moreover, Buildroot is capable of downloading sources via other tools, like `rsync` or `scp` (refer to Chapter 20, *Download infrastructure* for more details). If you enable packages using any of these methods, you will need to install the corresponding tool on the host system:

  - `bazaar`
  - `cvs`
  - `git`
  - `mercurial`
  - `rsync`
  - `scp`
  - `sftp`
  - `subversion`

- Java-related packages, if the Java Classpath needs to be built for the target system:
  - The `javac` compiler
  - The `jar` tool

- Documentation generation tools:
  - `asciidoc`, version 8.6.3 or higher
  - `w3m`
  - `python` with the `argparse` module (automatically present in 2.7+ and 3.2+)
  - `dblatex` (required for the pdf manual only)

- Graph generation tools:
  - `graphviz` to use *graph-depends* and *<pkg>-graph-depends*
  - `python-matplotlib` to use *graph-build*

Chapter 3. Getting Buildroot

Buildroot releases are made every 3 months, in February, May, August and November. Release numbers are in the format YYYY.MM, so for example 2013.02, 2014.08.

Release tarballs are available at http://buildroot.org/downloads/.

For your convenience, a Vagrantfile is available in `support/misc/Vagrantfile` in the Buildroot source tree to quickly set up a virtual machine with the needed dependencies to get started.

If you want to setup an isolated buildroot environment on Linux or Mac Os X, paste this line onto your terminal:

```
curl -O https://buildroot.org/downloads/Vagrantfile; vagrant up
```

If you are on Windows, paste this into your powershell:

```
(new-object System.Net.WebClient).DownloadFile(
"https://buildroot.org/downloads/Vagrantfile","Vagrantfile");
vagrant up
```

If you want to follow development, you can use the daily snapshots or make a clone of the Git repository. Refer to the Download page of the Buildroot website for more details.

Chapter 4. Buildroot quick start

Important: you can and should **build everything as a normal user**. There is no need to be root to configure and use Buildroot. By running all commands as a regular user, you protect your system against packages behaving badly during compilation and installation.

The first step when using Buildroot is to create a configuration. Buildroot has a nice configuration tool similar to the one you can find in the Linux kernel or in BusyBox.

From the buildroot directory, run

```
$ make menuconfig
```

for the original curses-based configurator, or

```
$ make nconfig
```

for the new curses-based configurator, or

```
$ make xconfig
```

for the Qt-based configurator, or

```
$ make gconfig
```

for the GTK-based configurator.

All of these "make" commands will need to build a configuration utility (including the interface), so you may need to install "development" packages for relevant libraries used by the configuration utilities. Refer to Chapter 2, *System requirements* for more details, specifically the optional requirements to get the dependencies of your favorite interface.

For each menu entry in the configuration tool, you can find associated help that describes the purpose of the entry. Refer to Chapter 6, *Buildroot configuration* for details on some specific configuration aspects.

Once everything is configured, the configuration tool generates a `.config` file that contains the entire configuration. This file will be read by the top-level Makefile.

To start the build process, simply run:

```
$ make
```

By default, Buildroot does not support top-level parallel build, so running `make -jN` is not necessary. There is however experimental support for top-level parallel build, see Section 8.12, "Top-level parallel build".

The `make` command will generally perform the following steps:

- download source files (as required);
- configure, build and install the cross-compilation toolchain, or simply import an external toolchain;
- configure, build and install selected target packages;
- build a kernel image, if selected;
- build a bootloader image, if selected;
- create a root filesystem in selected formats.

Buildroot output is stored in a single directory, `output/`. This directory contains several subdirectories:

- `images/` where all the images (kernel image, bootloader and root filesystem images) are stored. These are the files you need to put on your target system.
- `build/` where all the components are built (this includes tools needed by Buildroot on the host and packages compiled for the target). This directory contains one subdirectory for each of these components.
- `host/` contains both the tools built for the host, and the sysroot of the target toolchain. The former is an installation of tools compiled for the host that are needed for the proper execution of Buildroot, including the cross-compilation toolchain. The latter is a hierarchy similar to a root filesystem hierarchy. It contains the headers and libraries of all user-space packages that provide and install libraries used by other packages. However, this directory is *not* intended to be the root filesystem for the target: it contains a lot of development files, unstripped binaries and libraries that make it far too big for an embedded system. These development files are used to compile libraries and applications for the target that depend on other libraries.
- `staging/` is a symlink to the target toolchain sysroot inside `host/`, which exists for backwards compatibility.
- `target/` which contains *almost* the complete root filesystem for the target: everything needed is present except the device files in `/dev/` (Buildroot can't create them because Buildroot doesn't run as root and doesn't want to run as root). Also, it doesn't have the correct permissions (e.g. setuid for the busybox binary). Therefore, this directory **should not be used on your target**. Instead, you should use one of the images built in the `images/` directory. If you need an extracted image of the root filesystem for booting over NFS, then use the tarball image generated in `images/` and extract it as root. Compared to `staging/`, `target/` contains only the files and libraries needed to run the selected target applications: the development files (headers, etc.) are not present, the binaries are stripped.

These commands, `make menuconfig|nconfig|gconfig|xconfig` and `make`, are the basic ones that allow to easily and quickly generate images fitting your needs, with all the features and applications you enabled.

More details about the "make" command usage are given in Section 8.1, "*make* tips".

Chapter 5. Community resources

Like any open source project, Buildroot has different ways to share information in its community and outside.

Each of those ways may interest you if you are looking for some help, want to understand Buildroot or contribute to the project.

Mailing List
: Buildroot has a mailing list for discussion and development. It is the main method of interaction for Buildroot users and developers.

  Only subscribers to the Buildroot mailing list are allowed to post to this list. You can subscribe via the mailing list info page.

  Mails that are sent to the mailing list are also available in the mailing list archives, available through Mailman or at lore.kernel.org.

IRC
: The Buildroot IRC channel #buildroot is hosted on OFTC. It is a useful place to ask quick questions or discuss on certain topics.

  When asking for help on IRC, share relevant logs or pieces of code using a code sharing website, such as https://paste.ack.tf/.

  Note that for certain questions, posting to the mailing list may be better as it will reach more people, both developers and users.

Bug tracker
: Bugs in Buildroot can be reported via the mailing list or alternatively via the Buildroot bugtracker. Please refer to Section 22.6, "Reporting issues/bugs or getting help" before creating a bug report.

Wiki
: The Buildroot wiki page is hosted on the eLinux wiki. It contains some useful links, an overview of past and upcoming events, and a TODO list.

Patchwork
: Patchwork is a web-based patch tracking system designed to facilitate the contribution and management of contributions to an open-source project. Patches that have been sent to a mailing list are 'caught' by the system, and appear on a web page. Any comments posted that reference the patch are appended to the patch page too. For more information on Patchwork see http://jk.ozlabs.org/projects/patchwork/.

  Buildroot's Patchwork website is mainly for use by Buildroot's maintainer to ensure patches aren't missed. It is also used by Buildroot patch reviewers (see also Section 22.3.1, "Applying Patches from Patchwork"). However, since the website exposes patches and their corresponding review comments in a clean and concise web interface, it can be useful for all Buildroot developers.

  The Buildroot patch management interface is available at https://patchwork.ozlabs.org/project/buildroot/list/.

Part II. User guide

Chapter 6. Buildroot configuration

All the configuration options in `make *config` have a help text providing details about the option.

The `make *config` commands also offer a search tool. Read the help message in the different frontend menus to know how to use it:

- in *menuconfig*, the search tool is called by pressing `/`;
- in *xconfig*, the search tool is called by pressing `Ctrl` + `f`.

The result of the search shows the help message of the matching items. In *menuconfig*, numbers in the left column provide a shortcut to the corresponding entry. Just type this number to directly jump to the entry, or to the containing menu in case the entry is not selectable due to a missing dependency.

Although the menu structure and the help text of the entries should be sufficiently self-explanatory, a number of topics require additional explanation that cannot easily be covered in the help text and are therefore covered in the following sections.

6.1. Cross-compilation toolchain

A compilation toolchain is the set of tools that allows you to compile code for your system. It consists of a compiler (in our case, `gcc`), binary utils like assembler and linker (in our case, `binutils`) and a C standard library (for example GNU Libc, uClibc-ng).

The system installed on your development station certainly already has a compilation toolchain that you can use to compile an application that runs on your system. If you're using a PC, your compilation toolchain runs on an x86 processor and generates code for an x86 processor. Under most Linux systems, the compilation toolchain uses the GNU libc (glibc) as the C standard library. This compilation toolchain is called the "host compilation toolchain". The machine on which it is running, and on which you're working, is called the "host system" [3].

The compilation toolchain is provided by your distribution, and Buildroot has nothing to do with it (other than using it to build a cross-compilation toolchain and other tools that are run on the development host).

As said above, the compilation toolchain that comes with your system runs on and generates code for the processor in your host system. As your embedded system has a different processor, you need a cross-compilation toolchain - a compilation toolchain that runs on your *host system* but generates code for your *target system* (and target processor). For example, if your host system uses x86 and your target system uses ARM, the regular compilation toolchain on your host runs on x86 and generates code for x86, while the cross-compilation toolchain runs on x86 and generates code for ARM.

Buildroot provides two solutions for the cross-compilation toolchain:

- The **internal toolchain backend**, called `Buildroot toolchain` in the configuration interface.
- The **external toolchain backend**, called `External toolchain` in the configuration interface.

The choice between these two solutions is done using the `Toolchain Type` option in the `Toolchain` menu. Once one solution has been chosen, a number of configuration options appear, they are detailed in the following sections.

6.1.1. Internal toolchain backend

The *internal toolchain backend* is the backend where Buildroot builds by itself a cross-compilation toolchain, before building the userspace applications and libraries for your target embedded system.

This backend supports several C libraries: uClibc-ng, glibc and musl.

Once you have selected this backend, a number of options appear. The most important ones allow to:

- Change the version of the Linux kernel headers used to build the toolchain. This item deserves a few explanations. In the process of building a cross-compilation toolchain, the C library is being built. This library provides the interface between userspace applications and the Linux kernel. In order to know how to "talk" to the Linux kernel, the C library needs to have access to the *Linux kernel headers* (i.e. the `.h` files from the kernel), which define the interface between userspace and the kernel (system calls, data structures, etc.). Since this interface is backward compatible, the version of the Linux kernel headers used to build your toolchain do not need to match *exactly* the version of the Linux kernel you intend to run on your embedded system. They only need to have a version equal or older to the version of the Linux kernel you intend to run. If you use kernel headers that are more recent than the Linux kernel you run on your embedded system, then the C library might be using interfaces that are not provided by your Linux kernel.
- Change the version of the GCC compiler, binutils and the C library.
- Select a number of toolchain options (uClibc only): whether the toolchain should have RPC support (used mainly for NFS), wide-char support, locale support (for internationalization), C++ support or thread support. Depending on which options you choose, the number of userspace applications and libraries visible in Buildroot menus will change: many applications and libraries require certain toolchain options to be enabled. Most packages show a comment when a certain toolchain option is required to be able to enable those packages. If needed, you can further refine the uClibc configuration by running `make uclibc-menuconfig`. Note however that all packages in Buildroot are tested against the default uClibc configuration bundled in Buildroot: if you deviate from this configuration by removing features from uClibc, some packages may no longer build.

It is worth noting that whenever one of those options is modified, then the entire toolchain and system must be rebuilt. See Section 8.2, "Understanding when a full rebuild is necessary".

Advantages of this backend:

- Well integrated with Buildroot
- Fast, only builds what's necessary

Drawbacks of this backend:

- Rebuilding the toolchain is needed when doing `make clean`, which takes time. If you’re trying to reduce your build time, consider using the *External toolchain backend*.

6.1.2. External toolchain backend

The *external toolchain backend* allows to use existing pre-built cross-compilation toolchains. Buildroot knows about a number of well-known cross-compilation toolchains (from Linaro for ARM, Sourcery CodeBench for ARM, x86-64, PowerPC, and MIPS, and is capable of downloading them automatically, or it can be pointed to a custom toolchain, either available for download or installed locally.

Then, you have three solutions to use an external toolchain:

- Use a predefined external toolchain profile, and let Buildroot download, extract and install the toolchain. Buildroot already knows about a few CodeSourcery and Linaro toolchains. Just select the toolchain profile in `Toolchain` from the available ones. This is definitely the easiest solution.
- Use a predefined external toolchain profile, but instead of having Buildroot download and extract the toolchain, you can tell Buildroot where your toolchain is already installed on your system. Just select the toolchain profile in `Toolchain` through the available ones, unselect `Download toolchain automatically`, and fill the `Toolchain path` text entry with the path to your cross-compiling toolchain.
- Use a completely custom external toolchain. This is particularly useful for toolchains generated using crosstool-NG or with Buildroot itself. To do this, select the `Custom toolchain` solution in the `Toolchain` list. You need to fill the `Toolchain path`, `Toolchain prefix` and `External toolchain C library` options. Then, you have to tell Buildroot what your external toolchain supports. If your external toolchain uses the *glibc* library, you only have to tell whether your toolchain supports C++ or not and whether it has built-in RPC support. If your external toolchain uses the *uClibc* library, then you have to tell Buildroot if it supports RPC, wide-char, locale, program invocation, threads and C++. At the beginning of the execution, Buildroot will tell you if the selected options do not match the toolchain configuration.

Our external toolchain support has been tested with toolchains from CodeSourcery and Linaro, toolchains generated by crosstool-NG, and toolchains generated by Buildroot itself. In general, all toolchains that support the *sysroot* feature should work. If not, do not hesitate to contact the developers.

We do not support toolchains or SDK generated by OpenEmbedded or Yocto, because these toolchains are not pure toolchains (i.e. just the compiler, binutils, the C and C++ libraries). Instead these toolchains come with a very large set of pre-compiled libraries and programs. Therefore, Buildroot cannot import the *sysroot* of the toolchain, as it would contain hundreds of megabytes of pre-compiled libraries that are normally built by Buildroot.

We also do not support using the distribution toolchain (i.e. the gcc/binutils/C library installed by your distribution) as the toolchain to build software for the target. This is because your distribution toolchain is not a "pure" toolchain (i.e. only with the C/C++ library), so we cannot import it properly into the Buildroot build environment. So even if you are building a system for a x86 or x86_64 target, you have to generate a cross-compilation toolchain with Buildroot or crosstool-NG.

If you want to generate a custom toolchain for your project, that can be used as an external toolchain in Buildroot, our recommendation is to build it either with Buildroot itself (see Section 6.1.3, “Build an external toolchain with Buildroot”) or with crosstool-NG.

Advantages of this backend:

- Allows to use well-known and well-tested cross-compilation toolchains.
- Avoids the build time of the cross-compilation toolchain, which is often very significant in the overall build time of an embedded Linux system.

Drawbacks of this backend:

- If your pre-built external toolchain has a bug, may be hard to get a fix from the toolchain vendor, unless you build your external toolchain by yourself using Buildroot or Crosstool-NG.

6.1.3. Build an external toolchain with Buildroot

The Buildroot internal toolchain option can be used to create an external toolchain. Here are a series of steps to build an internal toolchain and package it up for reuse by Buildroot itself (or other projects).

Create a new Buildroot configuration, with the following details:

- Select the appropriate **Target options** for your target CPU architecture
- In the **Toolchain** menu, keep the default of **Buildroot toolchain** for **Toolchain type**, and configure your toolchain as desired
- In the **System configuration** menu, select **None** as the **Init system** and **none** as **/bin/sh**
- In the **Target packages** menu, disable **BusyBox**
- In the **Filesystem images** menu, disable **tar the root filesystem**

Then, we can trigger the build, and also ask Buildroot to generate a SDK. This will conveniently generate for us a tarball which contains our toolchain:

```
make sdk
```

This produces the SDK tarball in `$(O)/images`, with a name similar to `arm-buildroot-linux-uclibcgnueabi_sdk-`

`buildroot.tar.gz`. Save this tarball, as it is now the toolchain that you can re-use as an external toolchain in other Buildroot projects.

In those other Buildroot projects, in the **Toolchain** menu:

- Set **Toolchain type** to **External toolchain**
- Set **Toolchain** to **Custom toolchain**
- Set **Toolchain origin** to **Toolchain to be downloaded and installed**
- Set **Toolchain URL** to `file:///path/to/your/sdk/tarball.tar.gz`

External toolchain wrapper

When using an external toolchain, Buildroot generates a wrapper program, that transparently passes the appropriate options (according to the configuration) to the external toolchain programs. In case you need to debug this wrapper to check exactly what arguments are passed, you can set the environment variable `BR2_DEBUG_WRAPPER` to either one of:

- `0`, empty or not set: no debug
- `1`: trace all arguments on a single line
- `2`: trace one argument per line

6.2. /dev management

On a Linux system, the `/dev` directory contains special files, called *device files*, that allow userspace applications to access the hardware devices managed by the Linux kernel. Without these *device files*, your userspace applications would not be able to use the hardware devices, even if they are properly recognized by the Linux kernel.

Under `System configuration`, `/dev management`, Buildroot offers four different solutions to handle the `/dev` directory :

- The first solution is **Static using device table**. This is the old classical way of handling device files in Linux. With this method, the device files are persistently stored in the root filesystem (i.e. they persist across reboots), and there is nothing that will automatically create and remove those device files when hardware devices are added or removed from the system. Buildroot therefore creates a standard set of device files using a *device table*, the default one being stored in `system/device_table_dev.txt` in the Buildroot source code. This file is processed when Buildroot generates the final root filesystem image, and the *device files* are therefore not visible in the `output/target` directory. The `BR2_ROOTFS_STATIC_DEVICE_TABLE` option allows to change the default device table used by Buildroot, or to add an additional device table, so that additional *device files* are created by Buildroot during the build. So, if you use this method, and a *device file* is missing in your system, you can for example create a `board/<yourcompany>/<yourproject>/device_table_dev.txt` file that contains the description of your additional *device files*, and then you can set `BR2_ROOTFS_STATIC_DEVICE_TABLE` to `system/device_table_dev.txt board/<yourcompany>/<yourproject>/device_table_dev.txt`. For more details about the format of the device table file, see Chapter 25, *Makedev syntax documentation*.
- The second solution is **Dynamic using devtmpfs only**. *devtmpfs* is a virtual filesystem inside the Linux kernel that has been introduced in kernel 2.6.32 (if you use an older kernel, it is not possible to use this option). When mounted in `/dev`, this virtual filesystem will automatically make *device files* appear and disappear as hardware devices are added and removed from the system. This filesystem is not persistent across reboots: it is filled dynamically by the kernel. Using *devtmpfs* requires the following kernel configuration options to be enabled: `CONFIG_DEVTMPFS` and `CONFIG_DEVTMPFS_MOUNT`. When Buildroot is in charge of building the Linux kernel for your embedded device, it makes sure that those two options are enabled. However, if you build your Linux kernel outside of Buildroot, then it is your responsibility to enable those two options (if you fail to do so, your Buildroot system will not boot).
- The third solution is **Dynamic using devtmpfs + mdev**. This method also relies on the *devtmpfs* virtual filesystem detailed above (so the requirement to have `CONFIG_DEVTMPFS` and `CONFIG_DEVTMPFS_MOUNT` enabled in the kernel configuration still apply), but adds the `mdev` userspace utility on top of it. `mdev` is a program part of BusyBox that the kernel will call every time a device is added or removed. Thanks to the `/etc/mdev.conf` configuration file, `mdev` can be configured to for example, set specific permissions or ownership on a device file, call a script or application whenever a device appears or disappear, etc. Basically, it allows *userspace* to react on device addition and removal events. `mdev` can for example be used to automatically load kernel modules when devices appear on the system. `mdev` is also important if you have devices that require a firmware, as it will be responsible for pushing the firmware contents to the kernel. `mdev` is a lightweight implementation (with fewer features) of `udev`. For more details about `mdev` and the syntax of its configuration file, see http://git.busybox.net/busybox/tree/docs/mdev.txt.
- The fourth solution is **Dynamic using devtmpfs + eudev**. This method also relies on the *devtmpfs* virtual filesystem detailed above, but adds the `eudev` userspace daemon on top of it. `eudev` is a daemon that runs in the background, and gets called by the kernel when a device gets added or removed from the system. It is a more heavyweight solution than `mdev`, but provides higher flexibility. `eudev` is a standalone version of `udev`, the original userspace daemon used in most desktop Linux distributions, which is now part of Systemd. For more details, see http://en.wikipedia.org/wiki/Udev.

The Buildroot developers recommendation is to start with the **Dynamic using devtmpfs only** solution, until you have the need for userspace to be notified when devices are added/removed, or if firmwares are needed, in which case **Dynamic using devtmpfs + mdev** is usually a good solution.

Note that if `systemd` is chosen as init system, /dev management will be performed by the `udev` program provided by `systemd`.

6.3. init system

The *init* program is the first userspace program started by the kernel (it carries the PID number 1), and is responsible for starting the userspace services and programs (for example: web server, graphical applications, other network servers, etc.).

Buildroot allows to use three different types of init systems, which can be chosen from `System configuration`, `Init system`:

- The first solution is **BusyBox**. Amongst many programs, BusyBox has an implementation of a basic `init` program, which is sufficient for most embedded systems. Enabling the `BR2_INIT_BUSYBOX` will ensure BusyBox will build and install its `init` program. This is the default solution in Buildroot. The BusyBox `init` program will read the `/etc/inittab` file at boot to know what to do. The syntax of this file can be found in http://git.busybox.net/busybox/tree/examples/inittab (note that BusyBox `inittab` syntax is special: do not use a random `inittab` documentation from the Internet to learn about BusyBox `inittab`). The default `inittab` in Buildroot is stored in `system/skeleton/etc/inittab`. Apart from mounting a few important filesystems, the main job the default inittab does is to start the `/etc/init.d/rcS` shell script, and start a `getty` program (which provides a login prompt).
- The second solution is **systemV**. This solution uses the old traditional *sysvinit* program, packed in Buildroot in `package/sysvinit`. This was the solution used in most desktop Linux distributions, until they switched to more recent alternatives such as Upstart or Systemd. `sysvinit` also works with an `inittab` file (which has a slightly different syntax than the one from BusyBox). The default `inittab` installed with this init solution is located in `package/sysvinit/inittab`.
- The third solution is **systemd**. `systemd` is the new generation init system for Linux. It does far more than traditional *init* programs: aggressive parallelization capabilities, uses socket and D-Bus activation for starting services, offers on-demand starting of daemons, keeps track of processes using Linux control groups, supports snapshotting and restoring of the system state, etc. `systemd` will be useful on relatively complex embedded systems, for example the ones requiring D-Bus and services communicating between each other. It is worth noting that `systemd` brings a fairly big number of large dependencies: `dbus`, `udev` and more. For more details about `systemd`, see http://www.freedesktop.org/wiki/Software/systemd.

The solution recommended by Buildroot developers is to use the **BusyBox init** as it is sufficient for most embedded systems. **systemd** can be used for more complex situations.

---

[3] This terminology differs from what is used by GNU configure, where the host is the machine on which the application will run (which is usually the same as target)

Chapter 7. Configuration of other components

Before attempting to modify any of the components below, make sure you have already configured Buildroot itself, and have enabled the corresponding package.

BusyBox
: If you already have a BusyBox configuration file, you can directly specify this file in the Buildroot configuration, using `BR2_PACKAGE_BUSYBOX_CONFIG`. Otherwise, Buildroot will start from a default BusyBox configuration file.

  To make subsequent changes to the configuration, use `make busybox-menuconfig` to open the BusyBox configuration editor.

  It is also possible to specify a BusyBox configuration file through an environment variable, although this is not recommended. Refer to Section 8.6, "Environment variables" for more details.

uClibc
: Configuration of uClibc is done in the same way as for BusyBox. The configuration variable to specify an existing configuration file is `BR2_UCLIBC_CONFIG`. The command to make subsequent changes is `make uclibc-menuconfig`.

Linux kernel
: If you already have a kernel configuration file, you can directly specify this file in the Buildroot configuration, using `BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG`.

  If you do not yet have a kernel configuration file, you can either start by specifying a defconfig in the Buildroot configuration, using `BR2_LINUX_KERNEL_USE_DEFCONFIG`, or start by creating an empty file and specifying it as custom configuration file, using `BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG`.

  To make subsequent changes to the configuration, use `make linux-menuconfig` to open the Linux configuration editor.

Barebox
: Configuration of Barebox is done in the same way as for the Linux kernel. The corresponding configuration variables are `BR2_TARGET_BAREBOX_USE_CUSTOM_CONFIG` and `BR2_TARGET_BAREBOX_USE_DEFCONFIG`. To open the configuration editor, use `make barebox-menuconfig`.

U-Boot
: Configuration of U-Boot (version 2015.04 or newer) is done in the same way as for the Linux kernel. The corresponding configuration variables are `BR2_TARGET_UBOOT_USE_CUSTOM_CONFIG` and `BR2_TARGET_UBOOT_USE_DEFCONFIG`. To open the configuration editor, use `make uboot-menuconfig`.

Chapter 8. General Buildroot usage

8.1. *make* tips

This is a collection of tips that help you make the most of Buildroot.

Display all commands executed by make:

```
$ make V=1 <target>
```

Display the list of boards with a defconfig:

```
$ make list-defconfigs
```

Display all available targets:

```
$ make help
```

Not all targets are always available, some settings in the `.config` file may hide some targets:

- `busybox-menuconfig` only works when `busybox` is enabled;
- `linux-menuconfig` and `linux-savedefconfig` only work when `linux` is enabled;
- `uclibc-menuconfig` is only available when the uClibc C library is selected in the internal toolchain backend;
- `barebox-menuconfig` and `barebox-savedefconfig` only work when the `barebox` bootloader is enabled.
- `uboot-menuconfig` and `uboot-savedefconfig` only work when the `U-Boot` bootloader is enabled and the `uboot` build system is set to `Kconfig`.

Cleaning: Explicit cleaning is required when any of the architecture or toolchain configuration options are changed.

To delete all build products (including build directories, host, staging and target trees, the images and the toolchain):

```
$ make clean
```

Generating the manual: The present manual sources are located in the *docs/manual* directory. To generate the manual:

```
$ make manual-clean
$ make manual
```

The manual outputs will be generated in *output/docs/manual*.

Notes

- A few tools are required to build the documentation (see: Section 2.2, "Optional packages").

Resetting Buildroot for a new target: To delete all build products as well as the configuration:

```
$ make distclean
```

Notes. If `ccache` is enabled, running `make clean` or `distclean` does not empty the compiler cache used by Buildroot. To delete it, refer to Section 8.13.3, "Using ccache in Buildroot".

Dumping the internal make variables: One can dump the variables known to make, along with their values:

```
$ make -s printvars VARS='VARIABLE1 VARIABLE2'
VARIABLE1=value_of_variable
VARIABLE2=value_of_variable
```

It is possible to tweak the output using some variables:

- `VARS` will limit the listing to variables which names match the specified make-patterns - this must be set else nothing is printed
- `QUOTED_VARS`, if set to `YES`, will single-quote the value
- `RAW_VARS`, if set to `YES`, will print the unexpanded value

For example:

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES
BUSYBOX_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_ALL_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_PATCH_DEPENDENCIES=
BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES QUOTED_VARS=YES
BUSYBOX_DEPENDENCIES='skeleton toolchain'
BUSYBOX_FINAL_ALL_DEPENDENCIES='skeleton toolchain'
BUSYBOX_FINAL_DEPENDENCIES='skeleton toolchain'
BUSYBOX_FINAL_PATCH_DEPENDENCIES=''
```

```
BUSYBOX_RDEPENDENCIES='ncurses util-linux'
```

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES RAW_VARS=YES
BUSYBOX_DEPENDENCIES=skeleton toolchain
BUSYBOX_FINAL_ALL_DEPENDENCIES=$(sort $(BUSYBOX_FINAL_DEPENDENCIES) $(BUSYBOX_FINAL_PATCH_DEPENDENCIES))
BUSYBOX_FINAL_DEPENDENCIES=$(sort $(BUSYBOX_DEPENDENCIES))
BUSYBOX_FINAL_PATCH_DEPENDENCIES=$(sort $(BUSYBOX_PATCH_DEPENDENCIES))
BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

The output of quoted variables can be reused in shell scripts, for example:

```
$ eval $(make -s printvars VARS=BUSYBOX_DEPENDENCIES QUOTED_VARS=YES)
$ echo $BUSYBOX_DEPENDENCIES
skeleton toolchain
```

8.2. Understanding when a full rebuild is necessary

Buildroot does not attempt to detect what parts of the system should be rebuilt when the system configuration is changed through `make menuconfig`, `make xconfig` or one of the other configuration tools. In some cases, Buildroot should rebuild the entire system, in some cases, only a specific subset of packages. But detecting this in a completely reliable manner is very difficult, and therefore the Buildroot developers have decided to simply not attempt to do this.

Instead, it is the responsibility of the user to know when a full rebuild is necessary. As a hint, here are a few rules of thumb that can help you understand how to work with Buildroot:

- When the target architecture configuration is changed, a complete rebuild is needed. Changing the architecture variant, the binary format or the floating point strategy for example has an impact on the entire system.
- When the toolchain configuration is changed, a complete rebuild generally is needed. Changing the toolchain configuration often involves changing the compiler version, the type of C library or its configuration, or some other fundamental configuration item, and these changes have an impact on the entire system.
- When an additional package is added to the configuration, a full rebuild is not necessarily needed. Buildroot will detect that this package has never been built, and will build it. However, if this package is a library that can optionally be used by packages that have already been built, Buildroot will not automatically rebuild those. Either you know which packages should be rebuilt, and you can rebuild them manually, or you should do a full rebuild. For example, let’s suppose you have built a system with the `ctorrent` package, but without `openssl`. Your system works, but you realize you would like to have SSL support in `ctorrent`, so you enable the `openssl` package in Buildroot configuration and restart the build. Buildroot will detect that `openssl` should be built and will be build it, but it will not detect that `ctorrent` should be rebuilt to benefit from `openssl` to add OpenSSL support. You will either have to do a full rebuild, or rebuild `ctorrent` itself.
- When a package is removed from the configuration, Buildroot does not do anything special. It does not remove the files installed by this package from the target root filesystem or from the toolchain *sysroot*. A full rebuild is needed to get rid of this package. However, generally you don’t necessarily need this package to be removed right now: you can wait for the next lunch break to restart the build from scratch.
- When the sub-options of a package are changed, the package is not automatically rebuilt. After making such changes, rebuilding only this package is often sufficient, unless enabling the package sub-option adds some features to the package that are useful for another package which has already been built. Again, Buildroot does not track when a package should be rebuilt: once a package has been built, it is never rebuilt unless explicitly told to do so.
- When a change to the root filesystem skeleton is made, a full rebuild is needed. However, when changes to the root filesystem overlay, a post-build script or a post-image script are made, there is no need for a full rebuild: a simple `make` invocation will take the changes into account.
- When a package listed in `FOO_DEPENDENCIES` is rebuilt or removed, the package `foo` is not automatically rebuilt. For example, if a package `bar` is listed in `FOO_DEPENDENCIES` with `FOO_DEPENDENCIES = bar` and the configuration of the `bar` package is changed, the configuration change would not result in a rebuild of package `foo` automatically. In this scenario, you may need to either rebuild any packages in your build which reference `bar` in their `DEPENDENCIES`, or perform a full rebuild to ensure any `bar` dependent packages are up to date.

Generally speaking, when you’re facing a build error and you’re unsure of the potential consequences of the configuration changes you’ve made, do a full rebuild. If you get the same build error, then you are sure that the error is not related to partial rebuilds of packages, and if this error occurs with packages from the official Buildroot, do not hesitate to report the problem! As your experience with Buildroot progresses, you will progressively learn when a full rebuild is really necessary, and you will save more and more time.

For reference, a full rebuild is achieved by running:

```
$ make clean all
```

8.3. Understanding how to rebuild packages

One of the most common questions asked by Buildroot users is how to rebuild a given package or how to remove a package without rebuilding everything from scratch.

Removing a package is unsupported by Buildroot without rebuilding from scratch. This is because Buildroot doesn’t keep track of which package installs what files in the `output/staging` and `output/target` directories, or which package would be compiled differently depending on the availability of another package.

The easiest way to rebuild a single package from scratch is to remove its build directory in `output/build`. Buildroot will

then re-extract, re-configure, re-compile and re-install this package from scratch. You can ask buildroot to do this with the `make <package>-dirclean` command.

On the other hand, if you only want to restart the build process of a package from its compilation step, you can run `make <package>-rebuild`. It will restart the compilation and installation of the package, but not from scratch: it basically re-executes `make` and `make install` inside the package, so it will only rebuild files that changed.

If you want to restart the build process of a package from its configuration step, you can run `make <package>-reconfigure`. It will restart the configuration, compilation and installation of the package.

While `<package>-rebuild` implies `<package>-reinstall` and `<package>-reconfigure` implies `<package>-rebuild`, these targets as well as `<package>` only act on the said package, and do not trigger re-creating the root filesystem image. If re-creating the root filesystem in necessary, one should in addition run `make` or `make all`.

Internally, Buildroot creates so-called *stamp files* to keep track of which build steps have been completed for each package. They are stored in the package build directory, `output/build/<package>-<version>/` and are named `.stamp_<step-name>`. The commands detailed above simply manipulate these stamp files to force Buildroot to restart a specific set of steps of a package build process.

Further details about package special make targets are explained in Section 8.13.5, "Package-specific *make* targets".

8.4. Offline builds

If you intend to do an offline build and just want to download all sources that you previously selected in the configurator (*menuconfig*, *nconfig*, *xconfig* or *gconfig*), then issue:

```
$ make source
```

You can now disconnect or copy the content of your `dl` directory to the build-host.

8.5. Building out-of-tree

As default, everything built by Buildroot is stored in the directory `output` in the Buildroot tree.

Buildroot also supports building out of tree with a syntax similar to the Linux kernel. To use it, add `O=<directory>` to the make command line:

```
$ make O=/tmp/build menuconfig
```

Or:

```
$ cd /tmp/build; make O=$PWD -C path/to/buildroot menuconfig
```

All the output files will be located under `/tmp/build`. If the `O` path does not exist, Buildroot will create it.

Note: the `O` path can be either an absolute or a relative path, but if it's passed as a relative path, it is important to note that it is interpreted relative to the main Buildroot source directory, **not** the current working directory.

When using out-of-tree builds, the Buildroot `.config` and temporary files are also stored in the output directory. This means that you can safely run multiple builds in parallel using the same source tree as long as they use unique output directories.

For ease of use, Buildroot generates a Makefile wrapper in the output directory - so after the first run, you no longer need to pass `O=<...>` and `-C <...>`, simply run (in the output directory):

```
$ make <target>
```

8.6. Environment variables

Buildroot also honors some environment variables, when they are passed to `make` or set in the environment:

- `HOSTCXX`, the host C++ compiler to use
- `HOSTCC`, the host C compiler to use
- `UCLIBC_CONFIG_FILE=<path/to/.config>`, path to the uClibc configuration file, used to compile uClibc, if an internal toolchain is being built. Note that the uClibc configuration file can also be set from the configuration interface, so through the Buildroot `.config` file; this is the recommended way of setting it.
- `BUSYBOX_CONFIG_FILE=<path/to/.config>`, path to the BusyBox configuration file. Note that the BusyBox configuration file can also be set from the configuration interface, so through the Buildroot `.config` file; this is the recommended way of setting it.
- `BR2_CCACHE_DIR` to override the directory where Buildroot stores the cached files when using ccache.
- `BR2_DL_DIR` to override the directory in which Buildroot stores/retrieves downloaded files. Note that the Buildroot download directory can also be set from the configuration interface, so through the Buildroot `.config` file. See Section 8.13.4, "Location of downloaded packages" for more details on how you can set the download directory.

- `BR2_GRAPH_ALT`, if set and non-empty, to use an alternate color-scheme in build-time graphs
- `BR2_GRAPH_OUT` to set the filetype of generated graphs, either `pdf` (the default), or `png`.
- `BR2_GRAPH_DEPS_OPTS` to pass extra options to the dependency graph; see Section 8.9, "Graphing the dependencies between packages" for the accepted options
- `BR2_GRAPH_DOT_OPTS` is passed verbatim as options to the `dot` utility to draw the dependency graph.
- `BR2_GRAPH_SIZE_OPTS` to pass extra options to the size graph; see Section 8.11, "Graphing the filesystem size contribution of packages" for the acepted options

An example that uses config files located in the toplevel directory and in your $HOME:

```
$ make UCLIBC_CONFIG_FILE=uClibc.config BUSYBOX_CONFIG_FILE=$HOME/bb.config
```

If you want to use a compiler other than the default `gcc` or `g++` for building helper-binaries on your host, then do

```
$ make HOSTCXX=g++-4.3-HEAD HOSTCC=gcc-4.3-HEAD
```

8.7. Dealing efficiently with filesystem images

Filesystem images can get pretty big, depending on the filesystem you choose, the number of packages, whether you provisioned free space… Yet, some locations in the filesystems images may just be *empty* (e.g. a long run of *zeroes*); such a file is called a *sparse* file.

Most tools can handle sparse files efficiently, and will only store or write those parts of a sparse file that are not empty.

For example:

- `tar` accepts the `-S` option to tell it to only store non-zero blocks of sparse files:
  - `tar cf archive.tar -S [files…]` will efficiently store sparse files in a tarball
  - `tar xf archive.tar -S` will efficiently store sparse files extracted from a tarball
- `cp` accepts the `--sparse=WHEN` option (`WHEN` is one of `auto`, `never` or `always`):
  - `cp --sparse=always source.file dest.file` will make `dest.file` a sparse file if `source.file` has long runs of zeroes

Other tools may have similar options. Please consult their respective man pages.

You can use sparse files if you need to store the filesystem images (e.g. to transfer from one machine to another), or if you need to send them (e.g. to the Q&A team).

Note however that flashing a filesystem image to a device while using the sparse mode of `dd` may result in a broken filesystem (e.g. the block bitmap of an ext2 filesystem may be corrupted; or, if you have sparse files in your filesystem, those parts may not be all-zeroes when read back). You should only use sparse files when handling files on the build machine, not when transferring them to an actual device that will be used on the target.

8.8. Details about packages

Buildroot can produce a JSON blurb that describes the set of enabled packages in the current configuration, together with their dependencies, licenses and other metadata. This JSON blurb is produced by using the `show-info` make target:

```
make show-info
```

Buildroot can also produce details about packages as HTML and JSON output using the `pkg-stats` make target. Amongst other things, these details include whether known CVEs (security vulnerabilities) affect the packages in your current configuration. It also shows if there is a newer upstream version for those packages.

```
make pkg-stats
```

8.9. Graphing the dependencies between packages

One of Buildroot's jobs is to know the dependencies between packages, and make sure they are built in the right order. These dependencies can sometimes be quite complicated, and for a given system, it is often not easy to understand why such or such package was brought into the build by Buildroot.

In order to help understanding the dependencies, and therefore better understand what is the role of the different components in your embedded Linux system, Buildroot is capable of generating dependency graphs.

To generate a dependency graph of the full system you have compiled, simply run:

```
make graph-depends
```

You will find the generated graph in `output/graphs/graph-depends.pdf`.

If your system is quite large, the dependency graph may be too complex and difficult to read. It is therefore possible to

generate the dependency graph just for a given package:

```
make <pkg>-graph-depends
```

You will find the generated graph in `output/graph/<pkg>-graph-depends.pdf`.

Note that the dependency graphs are generated using the `dot` tool from the *Graphviz* project, which you must have installed on your system to use this feature. In most distributions, it is available as the `graphviz` package.

By default, the dependency graphs are generated in the PDF format. However, by passing the `BR2_GRAPH_OUT` environment variable, you can switch to other output formats, such as PNG, PostScript or SVG. All formats supported by the `-T` option of the `dot` tool are supported.

```
BR2_GRAPH_OUT=svg make graph-depends
```

The `graph-depends` behaviour can be controlled by setting options in the `BR2_GRAPH_DEPS_OPTS` environment variable. The accepted options are:

- `--depth N`, `-d N`, to limit the dependency depth to `N` levels. The default, `0`, means no limit.
- `--stop-on PKG`, `-s PKG`, to stop the graph on the package `PKG`. `PKG` can be an actual package name, a glob, the keyword *virtual* (to stop on virtual packages), or the keyword *host* (to stop on host packages). The package is still present on the graph, but its dependencies are not.
- `--exclude PKG`, `-x PKG`, like `--stop-on`, but also omits `PKG` from the graph.
- `--transitive`, `--no-transitive`, to draw (or not) the transitive dependencies. The default is to not draw transitive dependencies.
- `--colors R,T,H`, the comma-separated list of colors to draw the root package (`R`), the target packages (`T`) and the host packages (`H`). Defaults to: `lightblue,grey,gainsboro`

```
BR2_GRAPH_DEPS_OPTS='-d 3 --no-transitive --colors=red,green,blue' make graph-depends
```

8.10. Graphing the build duration

When the build of a system takes a long time, it is sometimes useful to be able to understand which packages are the longest to build, to see if anything can be done to speed up the build. In order to help such build time analysis, Buildroot collects the build time of each step of each package, and allows to generate graphs from this data.

To generate the build time graph after a build, run:

```
make graph-build
```

This will generate a set of files in `output/graphs` :

- `build.hist-build.pdf`, a histogram of the build time for each package, ordered in the build order.
- `build.hist-duration.pdf`, a histogram of the build time for each package, ordered by duration (longest first)
- `build.hist-name.pdf`, a histogram of the build time for each package, order by package name.
- `build.pie-packages.pdf`, a pie chart of the build time per package
- `build.pie-steps.pdf`, a pie chart of the global time spent in each step of the packages build process.

This `graph-build` target requires the Python Matplotlib and Numpy libraries to be installed (`python-matplotlib` and `python-numpy` on most distributions), and also the `argparse` module if you’re using a Python version older than 2.7 (`python-argparse` on most distributions).

By default, the output format for the graph is PDF, but a different format can be selected using the `BR2_GRAPH_OUT` environment variable. The only other format supported is PNG:

```
BR2_GRAPH_OUT=png make graph-build
```

8.11. Graphing the filesystem size contribution of packages

When your target system grows, it is sometimes useful to understand how much each Buildroot package is contributing to the overall root filesystem size. To help with such an analysis, Buildroot collects data about files installed by each package and using this data, generates a graph and CSV files detailing the size contribution of the different packages.

To generate these data after a build, run:

```
make graph-size
```

This will generate:

- `output/graphs/graph-size.pdf`, a pie chart of the contribution of each package to the overall root filesystem size
- `output/graphs/package-size-stats.csv`, a CSV file giving the size contribution of each package to the overall root filesystem size

- output/graphs/file-size-stats.csv, a CSV file giving the size contribution of each installed file to the package it belongs, and to the overall filesystem size.

This graph-size target requires the Python Matplotlib library to be installed (python-matplotlib on most distributions), and also the argparse module if you're using a Python version older than 2.7 (python-argparse on most distributions).

Just like for the duration graph, a BR2_GRAPH_OUT environment variable is supported to adjust the output file format. See Section 8.9, "Graphing the dependencies between packages" for details about this environment variable.

Additionally, one may set the environment variable BR2_GRAPH_SIZE_OPTS to further control the generated graph. Accepted options are:

- --size-limit X, -l X, will group all packages which individual contribution is below X percent, to a single entry labelled *Others* in the graph. By default, X=0.01, which means packages each contributing less than 1% are grouped under *Others*. Accepted values are in the range [0.0..1.0].
- --iec, --binary, --si, --decimal, to use IEC (binary, powers of 1024) or SI (decimal, powers of 1000; the default) prefixes.
- --biggest-first, to sort packages in decreasing size order, rather than in increasing size order.

Note. The collected filesystem size data is only meaningful after a complete clean rebuild. Be sure to run make clean all before using make graph-size.

To compare the root filesystem size of two different Buildroot compilations, for example after adjusting the configuration or when switching to another Buildroot release, use the size-stats-compare script. It takes two file-size-stats.csv files (produced by make graph-size) as input. Refer to the help text of this script for more details:

```
utils/size-stats-compare -h
```

8.12. Top-level parallel build

Note. This section deals with a very experimental feature, which is known to break even in some non-unusual situations. Use at your own risk.

Buildroot has always been capable of using parallel build on a per package basis: each package is built by Buildroot using make -jN (or the equivalent invocation for non-make-based build systems). The level of parallelism is by default number of CPUs + 1, but it can be adjusted using the BR2_JLEVEL configuration option.

Until 2020.02, Buildroot was however building packages in a serial fashion: each package was built one after the other, without parallelization of the build between packages. As of 2020.02, Buildroot has experimental support for **top-level parallel build**, which allows some signicant build time savings by building packages that have no dependency relationship in parallel. This feature is however marked as experimental and is known not to work in some cases.

In order to use top-level parallel build, one must:

1. Enable the option BR2_PER_PACKAGE_DIRECTORIES in the Buildroot configuration
2. Use make -jN when starting the Buildroot build

Internally, the BR2_PER_PACKAGE_DIRECTORIES will enable a mechanism called **per-package directories**, which will have the following effects:

- Instead of a global *target* directory and a global *host* directory common to all packages, per-package *target* and *host* directories will be used, in $(O)/per-package/<pkg>/target/ and $(O)/per-package/<pkg>/host/ respectively. Those folders will be populated from the corresponding folders of the package dependencies at the beginning of <pkg> build. The compiler and all other tools will therefore only be able to see and access files installed by dependencies explicitly listed by <pkg>.
- At the end of the build, the global *target* and *host* directories will be populated, located in $(O)/target and $(O)/host respectively. This means that during the build, those folders will be empty and it's only at the very end of the build that they will be populated.

8.13. Advanced usage

8.13.1. Using the generated toolchain outside Buildroot

You may want to compile, for your target, your own programs or other software that are not packaged in Buildroot. In order to do this you can use the toolchain that was generated by Buildroot.

The toolchain generated by Buildroot is located by default in output/host/. The simplest way to use it is to add output/host/bin/ to your PATH environment variable and then to use ARCH-linux-gcc, ARCH-linux-objdump, ARCH-linux-ld, etc.

Alternatively, Buildroot can also export the toolchain and the development files of all selected packages, as an SDK, by running the command make sdk. This generates a tarball of the content of the host directory output/host/, named <TARGET-TUPLE>_sdk-buildroot.tar.gz (which can be overriden by setting the environment variable BR2_SDK_PREFIX) and located in the output directory output/images/.

This tarball can then be distributed to application developers, when they want to develop their applications that are not

(yet) packaged as a Buildroot package.

Upon extracting the SDK tarball, the user must run the script `relocate-sdk.sh` (located at the top directory of the SDK), to make sure all paths are updated with the new location.

Alternatively, if you just want to prepare the SDK without generating the tarball (e.g. because you will just be moving the `host` directory, or will be generating the tarball on your own), Buildroot also allows you to just prepare the SDK with `make prepare-sdk` without actually generating a tarball.

For your convenience, by selecting the option `BR2_PACKAGE_HOST_ENVIRONMENT_SETUP`, you can get a `environment-setup` script installed in `output/host/` and therefore in your SDK. This script can be sourced with `. your/sdk/path/environment-setup` to export a number of environment variables that will help cross-compile your projects using the Buildroot SDK: the `PATH` will contain the SDK binaries, standard *autotools* variables will be defined with the appropriate values, and `CONFIGURE_FLAGS` will contain basic `./configure` options to cross-compile *autotools* projects. It also provides some useful commands. Note however that once this script is sourced, the environment is setup only for cross-compilation, and no longer for native compilation.

8.13.2. Using `gdb` in Buildroot

Buildroot allows to do cross-debugging, where the debugger runs on the build machine and communicates with `gdbserver` on the target to control the execution of the program.

To achieve this:

- If you are using an *internal toolchain* (built by Buildroot), you must enable `BR2_PACKAGE_HOST_GDB`, `BR2_PACKAGE_GDB` and `BR2_PACKAGE_GDB_SERVER`. This ensures that both the cross gdb and gdbserver get built, and that gdbserver gets installed to your target.
- If you are using an *external toolchain*, you should enable `BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY`, which will copy the gdbserver included with the external toolchain to the target. If your external toolchain does not have a cross gdb or gdbserver, it is also possible to let Buildroot build them, by enabling the same options as for the *internal toolchain backend*.

Now, to start debugging a program called `foo`, you should run on the target:

```
gdbserver :2345 foo
```

This will cause `gdbserver` to listen on TCP port 2345 for a connection from the cross gdb.

Then, on the host, you should start the cross gdb using the following command line:

```
<buildroot>/output/host/bin/<tuple>-gdb -ix <buildroot>/output/staging/usr/share/buildroot/gdbinit foo
```

Of course, `foo` must be available in the current directory, built with debugging symbols. Typically you start this command from the directory where `foo` is built (and not from `output/target/` as the binaries in that directory are stripped).

The `<buildroot>/output/staging/usr/share/buildroot/gdbinit` file will tell the cross gdb where to find the libraries of the target.

Finally, to connect to the target from the cross gdb:

```
(gdb) target remote <target ip address>:2345
```

8.13.3. Using `ccache` in Buildroot

ccache is a compiler cache. It stores the object files resulting from each compilation process, and is able to skip future compilation of the same source file (with same compiler and same arguments) by using the pre-existing object files. When doing almost identical builds from scratch a number of times, it can nicely speed up the build process.

`ccache` support is integrated in Buildroot. You just have to enable `Enable compiler cache` in `Build options`. This will automatically build `ccache` and use it for every host and target compilation.

The cache is located in the directory defined by the `BR2_CCACHE_DIR` configuration option, which defaults to `$HOME/.buildroot-ccache`. This default location is outside of Buildroot output directory so that it can be shared by separate Buildroot builds. If you want to get rid of the cache, simply remove this directory.

You can get statistics on the cache (its size, number of hits, misses, etc.) by running `make ccache-stats`.

The make target `ccache-options` and the `CCACHE_OPTIONS` variable provide more generic access to the ccache. For example

```
# set cache limit size
make CCACHE_OPTIONS="--max-size=5G" ccache-options

# zero statistics counters
make CCACHE_OPTIONS="--zero-stats" ccache-options
```

`ccache` makes a hash of the source files and of the compiler options. If a compiler option is different, the cached object file will not be used. Many compiler options, however, contain an absolute path to the staging directory. Because of this,

building in a different output directory would lead to many cache misses.

To avoid this issue, buildroot has the `Use relative paths` option (`BR2_CCACHE_USE_BASEDIR`). This will rewrite all absolute paths that point inside the output directory into relative paths. Thus, changing the output directory no longer leads to cache misses.

A disadvantage of the relative paths is that they also end up to be relative paths in the object file. Therefore, for example, the debugger will no longer find the file, unless you cd to the output directory first.

See the ccache manual’s section on "Compiling in different directories" for more details about this rewriting of absolute paths.

When `ccache` is enabled in Buildroot using the `BR2_CCACHE=y` option:

- `ccache` is used during the Buildroot build itself
- `ccache` is not used when building outside of Buildroot, for example when directly calling the cross-compiler or using the SDK

One can override this behavior using the `BR2_USE_CCACHE` environment variable: when set to `1`, usage of ccache is enabled (default during the Buildroot build), when unset or set to a value different from `1`, usage of ccache is disabled.

8.13.4. Location of downloaded packages

The various tarballs that are downloaded by Buildroot are all stored in `BR2_DL_DIR`, which by default is the `dl` directory. If you want to keep a complete version of Buildroot which is known to be working with the associated tarballs, you can make a copy of this directory. This will allow you to regenerate the toolchain and the target filesystem with exactly the same versions.

If you maintain several Buildroot trees, it might be better to have a shared download location. This can be achieved by pointing the `BR2_DL_DIR` environment variable to a directory. If this is set, then the value of `BR2_DL_DIR` in the Buildroot configuration is overridden. The following line should be added to `<~/.bashrc>`.

```
export BR2_DL_DIR=<shared download location>
```

The download location can also be set in the `.config` file, with the `BR2_DL_DIR` option. Unlike most options in the .config file, this value is overridden by the `BR2_DL_DIR` environment variable.

8.13.5. Package-specific *make* targets

Running `make <package>` builds and installs that particular package and its dependencies.

For packages relying on the Buildroot infrastructure, there are numerous special make targets that can be called independently like this:

```
make <package>-<target>
```

The package build targets are (in the order they are executed):

command/target	Description
`source`	Fetch the source (download the tarball, clone the source repository, etc)
`depends`	Build and install all dependencies required to build the package
`extract`	Put the source in the package build directory (extract the tarball, copy the source, etc)
`patch`	Apply the patches, if any
`configure`	Run the configure commands, if any
`build`	Run the compilation commands
`install-staging`	**target package:** Run the installation of the package in the staging directory, if necessary
`install-target`	**target package:** Run the installation of the package in the target directory, if necessary
`install`	**target package:** Run the 2 previous installation commands **host package:** Run the installation of the package in the host directory

Additionally, there are some other useful make targets:

command/target	Description
`show-depends`	Displays the first-order dependencies required to build the package
`show-recursive-depends`	Recursively displays the dependencies required to build the package
`show-rdepends`	Displays the first-order reverse dependencies of the package (i.e packages that directly depend on it)
`show-recursive-rdepends`	Recursively displays the reverse dependencies of the package (i.e the packages that depend on it, directly or indirectly)
`graph-depends`	Generate a dependency graph of the package, in the context of the current Buildroot configuration. See this section for more details about dependency graphs.
`graph-rdepends`	Generate a graph of this package reverse dependencies (i.e the packages that depend on it, directly or indirectly)
`dirclean`	Remove the whole package build directory
`reinstall`	Re-run the install commands
`rebuild`	Re-run the compilation commands - this only makes sense when using the `OVERRIDE_SRCDIR` feature or when you modified a file directly in the build directory
`reconfigure`	Re-run the configure commands, then rebuild - this only makes sense when using the `OVERRIDE_SRCDIR` feature or when you modified a file directly in the build directory

8.13.6. Using Buildroot during development

The normal operation of Buildroot is to download a tarball, extract it, configure, compile and install the software component found inside this tarball. The source code is extracted in `output/build/<package>-<version>`, which is a temporary directory: whenever `make clean` is used, this directory is entirely removed, and re-created at the next `make` invocation. Even when a Git or Subversion repository is used as the input for the package source code, Buildroot creates a tarball out of it, and then behaves as it normally does with tarballs.

This behavior is well-suited when Buildroot is used mainly as an integration tool, to build and integrate all the components of an embedded Linux system. However, if one uses Buildroot during the development of certain components of the system, this behavior is not very convenient: one would instead like to make a small change to the source code of one package, and be able to quickly rebuild the system with Buildroot.

Making changes directly in `output/build/<package>-<version>` is not an appropriate solution, because this directory is removed on `make clean`.

Therefore, Buildroot provides a specific mechanism for this use case: the `<pkg>_OVERRIDE_SRCDIR` mechanism. Buildroot reads an *override* file, which allows the user to tell Buildroot the location of the source for certain packages.

The default location of the override file is `$(CONFIG_DIR)/local.mk`, as defined by the `BR2_PACKAGE_OVERRIDE_FILE` configuration option. `$(CONFIG_DIR)` is the location of the Buildroot `.config` file, so `local.mk` by default lives side-by-side with the `.config` file, which means:

- In the top-level Buildroot source directory for in-tree builds (i.e., when `O=` is not used)
- In the out-of-tree directory for out-of-tree builds (i.e., when `O=` is used)

If a different location than these defaults is required, it can be specified through the `BR2_PACKAGE_OVERRIDE_FILE` configuration option.

In this *override* file, Buildroot expects to find lines of the form:

```
<pkg1>_OVERRIDE_SRCDIR = /path/to/pkg1/sources
```

```
<pkg2>_OVERRIDE_SRCDIR = /path/to/pkg2/sources
```

For example:

```
LINUX_OVERRIDE_SRCDIR = /home/bob/linux/
BUSYBOX_OVERRIDE_SRCDIR = /home/bob/busybox/
```

When Buildroot finds that for a given package, an `<pkg>_OVERRIDE_SRCDIR` has been defined, it will no longer attempt to download, extract and patch the package. Instead, it will directly use the source code available in the specified directory and `make clean` will not touch this directory. This allows to point Buildroot to your own directories, that can be managed by Git, Subversion, or any other version control system. To achieve this, Buildroot will use *rsync* to copy the source code of the component from the specified `<pkg>_OVERRIDE_SRCDIR` to `output/build/<package>-custom/`.

This mechanism is best used in conjunction with the `make <pkg>-rebuild` and `make <pkg>-reconfigure` targets. A `make <pkg>-rebuild all` sequence will *rsync* the source code from `<pkg>_OVERRIDE_SRCDIR` to `output/build/<package>-custom` (thanks to *rsync*, only the modified files are copied), and restart the build process of just this package.

In the example of the `linux` package above, the developer can then make a source code change in `/home/bob/linux` and then run:

```
make linux-rebuild all
```

and in a matter of seconds gets the updated Linux kernel image in `output/images`. Similarly, a change can be made to the BusyBox source code in `/home/bob/busybox`, and after:

```
make busybox-rebuild all
```

the root filesystem image in `output/images` contains the updated BusyBox.

Source trees for big projects often contain hundreds or thousands of files which are not needed for building, but will slow down the process of copying the sources with *rsync*. Optionally, it is possible define `<pkg>_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS` to skip syncing certain files from the source tree. For example, when working on the `webkitgtk` package, the following will exclude the tests and in-tree builds from a local WebKit source tree:

```
WEBKITGTK_OVERRIDE_SRCDIR = /home/bob/WebKit
WEBKITGTK_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS = \
        --exclude JSTests --exclude ManualTests --exclude PerformanceTests \
        --exclude WebDriverTests --exclude WebKitBuild --exclude WebKitLibraries \
        --exclude WebKit.xcworkspace --exclude Websites --exclude Examples
```

By default, Buildroot skips syncing of VCS artifacts (e.g., the **.git** and **.svn** directories). Some packages prefer to have these VCS directories available during build, for example for automatically determining a precise commit reference for version information. To undo this built-in filtering at a cost of a slower speed, add these directories back:

```
LINUX_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS = --include .git
```

Chapter 9. Project-specific customization

Typical actions you may need to perform for a given project are:

- configuring Buildroot (including build options and toolchain, bootloader, kernel, package and filesystem image type selection)
- configuring other components, like the Linux kernel and BusyBox
- customizing the generated target filesystem
  - adding or overwriting files on the target filesystem (using `BR2_ROOTFS_OVERLAY`)
  - modifying or deleting files on the target filesystem (using `BR2_ROOTFS_POST_BUILD_SCRIPT`)
  - running arbitrary commands prior to generating the filesystem image (using `BR2_ROOTFS_POST_BUILD_SCRIPT`)
  - setting file permissions and ownership (using `BR2_ROOTFS_DEVICE_TABLE`)
  - adding custom devices nodes (using `BR2_ROOTFS_STATIC_DEVICE_TABLE`)
- adding custom user accounts (using `BR2_ROOTFS_USERS_TABLES`)
- running arbitrary commands after generating the filesystem image (using `BR2_ROOTFS_POST_IMAGE_SCRIPT`)
- adding project-specific patches to some packages (using `BR2_GLOBAL_PATCH_DIR`)
- adding project-specific packages

An important note regarding such *project-specific* customizations: please carefully consider which changes are indeed project-specific and which changes are also useful to developers outside your project. The Buildroot community highly recommends and encourages the upstreaming of improvements, packages and board support to the official Buildroot project. Of course, it is sometimes not possible or desirable to upstream because the changes are highly specific or proprietary.

This chapter describes how to make such project-specific customizations in Buildroot and how to store them in a way that

you can build the same image in a reproducible way, even after running *make clean*. By following the recommended strategy, you can even use the same Buildroot tree to build multiple distinct projects!

9.1. Recommended directory structure

When customizing Buildroot for your project, you will be creating one or more project-specific files that need to be stored somewhere. While most of these files could be placed in *any* location as their path is to be specified in the Buildroot configuration, the Buildroot developers recommend a specific directory structure which is described in this section.

Orthogonal to this directory structure, you can choose *where* you place this structure itself: either inside the Buildroot tree, or outside of it using a br2-external tree. Both options are valid, the choice is up to you.

```
+-- board/
|   +-- <company>/
|       +-- <boardname>/
|           +-- linux.config
|           +-- busybox.config
|           +-- <other configuration files>
|           +-- post_build.sh
|           +-- post_image.sh
|           +-- rootfs_overlay/
|           |   +-- etc/
|           |   +-- <some files>
|           +-- patches/
|               +-- foo/
|               |   +-- <some patches>
|               +-- libbar/
|                   +-- <some other patches>
|
+-- configs/
|   +-- <boardname>_defconfig
|
+-- package/
|   +-- <company>/
|       +-- Config.in (if not using a br2-external tree)
|       +-- <company>.mk (if not using a br2-external tree)
|       +-- package1/
|       |    +-- Config.in
|       |    +-- package1.mk
|       +-- package2/
|           +-- Config.in
|           +-- package2.mk
|
+-- Config.in (if using a br2-external tree)
+-- external.mk (if using a br2-external tree)
+-- external.desc (if using a br2-external tree)
```

Details on the files shown above are given further in this chapter.

Note: if you choose to place this structure outside of the Buildroot tree but in a br2-external tree, the <company> and possibly <boardname> components may be superfluous and can be left out.

9.1.1. Implementing layered customizations

It is quite common for a user to have several related projects that partly need the same customizations. Instead of duplicating these customizations for each project, it is recommended to use a layered customization approach, as explained in this section.

Almost all of the customization methods available in Buildroot, like post-build scripts and root filesystem overlays, accept a space-separated list of items. The specified items are always treated in order, from left to right. By creating more than one such item, one for the common customizations and another one for the really project-specific customizations, you can avoid unnecessary duplication. Each layer is typically embodied by a separate directory inside `board/<company>/`. Depending on your projects, you could even introduce more than two layers.

An example directory structure for where a user has two customization layers *common* and *fooboard* is:

```
+-- board/
    +-- <company>/
        +-- common/
        |   +-- post_build.sh
        |   +-- rootfs_overlay/
        |   |   +-- ...
        |   +-- patches/
        |       +-- ...
        |
        +-- fooboard/
            +-- linux.config
            +-- busybox.config
            +-- <other configuration files>
            +-- post_build.sh
            +-- rootfs_overlay/
            |   +-- ...
            +-- patches/
                +-- ...
```

For example, if the user has the `BR2_GLOBAL_PATCH_DIR` configuration option set as:

```
BR2_GLOBAL_PATCH_DIR="board/<company>/common/patches board/<company>/fooboard/patches"
```

then first the patches from the *common* layer would be applied, followed by the patches from the *fooboard* layer.

9.2. Keeping customizations outside of Buildroot

As already briefly mentioned in Section 9.1, "Recommended directory structure", you can place project-specific customizations in two locations:

- directly within the Buildroot tree, typically maintaining them using branches in a version control system so that upgrading to a newer Buildroot release is easy.
- outside of the Buildroot tree, using the *br2-external* mechanism. This mechanism allows to keep package recipes, board support and configuration files outside of the Buildroot tree, while still having them nicely integrated in the build logic. We call this location a *br2-external tree*. This section explains how to use the br2-external mechanism and what to provide in a br2-external tree.

One can tell Buildroot to use one or more br2-external trees by setting the `BR2_EXTERNAL` make variable set to the path(s) of the br2-external tree(s) to use. It can be passed to any Buildroot `make` invocation. It is automatically saved in the hidden `.br2-external.mk` file in the output directory. Thanks to this, there is no need to pass `BR2_EXTERNAL` at every `make` invocation. It can however be changed at any time by passing a new value, and can be removed by passing an empty value.

Note. The path to a br2-external tree can be either absolute or relative. If it is passed as a relative path, it is important to note that it is interpreted relative to the main Buildroot source directory, **not** to the Buildroot output directory.

Note: If using an br2-external tree from before Buildroot 2016.11, you need to convert it before you can use it with Buildroot 2016.11 onward. See Section 27.2, "Migrating to 2016.11" for help on doing so.

Some examples:

```
buildroot/ $ make BR2_EXTERNAL=/path/to/foo menuconfig
```

From now on, definitions from the `/path/to/foo` br2-external tree will be used:

```
buildroot/ $ make
buildroot/ $ make legal-info
```

We can switch to another br2-external tree at any time:

```
buildroot/ $ make BR2_EXTERNAL=/where/we/have/bar xconfig
```

We can also use multiple br2-external trees:

```
buildroot/ $ make BR2_EXTERNAL=/path/to/foo:/where/we/have/bar menuconfig
```

Or disable the usage of any br2-external tree:

```
buildroot/ $ make BR2_EXTERNAL= xconfig
```

9.2.1. Layout of a br2-external tree

A br2-external tree must contain at least those three files, described in the following chapters:

- `external.desc`
- `external.mk`
- `Config.in`

Apart from those mandatory files, there may be additional and optional content that may be present in a br2-external tree, like the `configs/` or `provides/` directories. They are described in the following chapters as well.

A complete example br2-external tree layout is also described later.

The `external.desc` file

That file describes the br2-external tree: the *name* and *description* for that br2-external tree.

The format for this file is line based, with each line starting by a keyword, followed by a colon and one or more spaces, followed by the value assigned to that keyword. There are two keywords currently recognised:

- `name`, mandatory, defines the name for that br2-external tree. That name must only use ASCII characters in the set `[A-Za-z0-9_]`; any other character is forbidden. Buildroot sets the variable `BR2_EXTERNAL_$(NAME)_PATH` to the absolute path of the br2-external tree, so that you can use it to refer to your br2-external tree. This variable is available both in Kconfig, so you can use it to source your Kconfig files (see below) and in the Makefile, so that you can use it to include other Makefiles (see below) or refer to other files (like data files) from your br2-external tree.

  Note: Since it is possible to use multiple br2-external trees at once, this name is used by Buildroot to generate variables for each of those trees. That name is used to identify your br2-external tree, so try to come up with a name that really

describes your br2-external tree, in order for it to be relatively unique, so that it does not clash with another name from another br2-external tree, especially if you are planning on somehow sharing your br2-external tree with third parties or using br2-external trees from third parties.

- `desc`, optional, provides a short description for that br2-external tree. It shall fit on a single line, is mostly free-form (see below), and is used when displaying information about a br2-external tree (e.g. above the list of defconfig files, or as the prompt in the menuconfig); as such, it should relatively brief (40 chars is probably a good upper limit). The description is available in the `BR2_EXTERNAL_$(NAME)_DESC` variable.

Examples of names and the corresponding `BR2_EXTERNAL_$(NAME)_PATH` variables:

- `FOO` → `BR2_EXTERNAL_FOO_PATH`
- `BAR_42` → `BR2_EXTERNAL_BAR_42_PATH`

In the following examples, it is assumed the name to be set to `BAR_42`.

Note: Both `BR2_EXTERNAL_$(NAME)_PATH` and `BR2_EXTERNAL_$(NAME)_DESC` are available in the Kconfig files and the Makefiles. They are also exported in the environment so are available in post-build, post-image and in-fakeroot scripts.

The `Config.in` and `external.mk` files

Those files (which may each be empty) can be used to define package recipes (i.e. `foo/Config.in` and `foo/foo.mk` like for packages bundled in Buildroot itself) or other custom configuration options or make logic.

Buildroot automatically includes the `Config.in` from each br2-external tree to make it appear in the top-level configuration menu, and includes the `external.mk` from each br2-external tree with the rest of the makefile logic.

The main usage of this is to store package recipes. The recommended way to do this is to write a `Config.in` file that looks like:

```
source "$BR2_EXTERNAL_BAR_42_PATH/package/package1/Config.in"
source "$BR2_EXTERNAL_BAR_42_PATH/package/package2/Config.in"
```

Then, have an `external.mk` file that looks like:

```
include $(sort $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/package/*/*.mk))
```

And then in `$(BR2_EXTERNAL_BAR_42_PATH)/package/package1` and `$(BR2_EXTERNAL_BAR_42_PATH)/package/package2` create normal Buildroot package recipes, as explained in Chapter 18, *Adding new packages to Buildroot*. If you prefer, you can also group the packages in subdirectories called <boardname> and adapt the above paths accordingly.

You can also define custom configuration options in `Config.in` and custom make logic in `external.mk`.

The `configs/` directory

One can store Buildroot defconfigs in the `configs` subdirectory of the br2-external tree. Buildroot will automatically show them in the output of `make list-defconfigs` and allow them to be loaded with the normal `make <name>_defconfig` command. They will be visible in the *make list-defconfigs* output, below an `External configs` label that contains the name of the br2-external tree they are defined in.

Note: If a defconfig file is present in more than one br2-external tree, then the one from the last br2-external tree is used. It is thus possible to override a defconfig bundled in Buildroot or another br2-external tree.

The `provides/` directory

For some packages, Buildroot provides a choice between two (or more) implementations of API-compatible such packages. For example, there is a choice to choose either libjpeg ot jpeg-turbo; there is one between openssl or libressl; there is one to select one of the known, pre-configured toolchains…

It is possible for a br2-external to extend those choices, by providing a set of files that define those alternatives:

- `provides/toolchains.in` defines the pre-configured toolchains, which will then be listed in the toolchain selection;
- `provides/jpeg.in` defines the alternative libjpeg implementations;
- `provides/openssl.in` defines the alternative openssl implementations;
- `provides/skeleton.in` defines the alternative skeleton implementations;
- `provides/init.in` defines the alternative init system implementations, this can be used to select a default skeleton for your init.

Free-form content

One can store all the board-specific configuration files there, such as the kernel configuration, the root filesystem overlay, or any other configuration file for which Buildroot allows to set the location (by using the `BR2_EXTERNAL_$(NAME)_PATH` variable). For example, you could set the paths to a global patch directory, to a rootfs overlay and to the kernel configuration file as follows (e.g. by running `make menuconfig` and filling in these options):

```
BR2_GLOBAL_PATCH_DIR=$(BR2_EXTERNAL_BAR_42_PATH)/patches/
```

```
BR2_ROOTFS_OVERLAY=$(BR2_EXTERNAL_BAR_42_PATH)/board/<boardname>/overlay/
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE=$(BR2_EXTERNAL_BAR_42_PATH)/board/<boardname>/kernel.config
```

Additional Linux kernel extensions

Additional Linux kernel extensions (see Section 18.22.2, "linux-kernel-extensions") can be added by storing them in the `linux/` directory at the root of a br2-external tree.

Example layout

Here is an example layout using all features of br2-external (the sample content is shown for the file above it, when it is relevant to explain the br2-external tree; this is all entirely made up just for the sake of illustration, of course):

```
/path/to/br2-ext-tree/
  |- external.desc
  |      |name: BAR_42
  |      |desc: Example br2-external tree
  |      `----
  |
  |- Config.in
  |      |source "$BR2_EXTERNAL_BAR_42_PATH/toolchain/toolchain-external-mine/Config.in.options"
  |      |source "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-1/Config.in"
  |      |source "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-2/Config.in"
  |      |source "$BR2_EXTERNAL_BAR_42_PATH/package/my-jpeg/Config.in"
  |      |
  |      |config BAR_42_FLASH_ADDR
  |      |    hex "my-board flash address"
  |      |    default 0x10AD
  |      `----
  |
  |- external.mk
  |      |include $(sort $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/package/*/*.mk))
  |      |include $(sort $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/toolchain/*/*.mk))
  |      |
  |      |flash-my-board:
  |      |    $(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/flash-image \
  |      |        --image $(BINARIES_DIR)/image.bin \
  |      |        --address $(BAR_42_FLASH_ADDR)
  |      `----
  |
  |- package/pkg-1/Config.in
  |      |config BR2_PACKAGE_PKG_1
  |      |    bool "pkg-1"
  |      |    help
  |      |      Some help about pkg-1
  |      `----
  |- package/pkg-1/pkg-1.hash
  |- package/pkg-1/pkg-1.mk
  |      |PKG_1_VERSION = 1.2.3
  |      |PKG_1_SITE = /some/where/to/get/pkg-1
  |      |PKG_1_LICENSE = blabla
  |      |
  |      |define PKG_1_INSTALL_INIT_SYSV
  |      |    $(INSTALL) -D -m 0755 $(PKG_1_PKGDIR)/S99my-daemon \
  |      |                          $(TARGET_DIR)/etc/init.d/S99my-daemon
  |      |endef
  |      |
  |      |$(eval $(autotools-package))
  |      `----
  |- package/pkg-1/S99my-daemon
  |
  |- package/pkg-2/Config.in
  |- package/pkg-2/pkg-2.hash
  |- package/pkg-2/pkg-2.mk
  |
  |- provides/jpeg.in
  |      |config BR2_PACKAGE_MY_JPEG
  |      |    bool "my-jpeg"
  |      `----
  |- package/my-jpeg/Config.in
  |      |config BR2_PACKAGE_PROVIDES_JPEG
  |      |    default "my-jpeg" if BR2_PACKAGE_MY_JPEG
  |      `----
  |- package/my-jpeg/my-jpeg.mk
  |      |# This is a normal package .mk file
  |      |MY_JPEG_VERSION = 1.2.3
  |      |MY_JPEG_SITE = https://example.net/some/place
  |      |MY_JPEG_PROVIDES = jpeg
  |      |$(eval $(autotools-package))
  |      `----
  |
  |- provides/init.in
  |      |config BR2_INIT_MINE
  |      |    bool "my custom init"
  |      |    select BR2_PACKAGE_MY_INIT
  |      |    select BR2_PACKAGE_SKELETON_INIT_MINE if BR2_ROOTFS_SKELETON_DEFAULT
  |      `----
  |
  |- provides/skeleton.in
  |      |config BR2_ROOTFS_SKELETON_MINE
  |      |    bool "my custom skeleton"
  |      |    select BR2_PACKAGE_SKELETON_MINE
  |      `----
```

```
   |- package/skeleton-mine/Config.in
   |      |config BR2_PACKAGE_SKELETON_MINE
   |      |    bool
   |      |    select BR2_PACKAGE_HAS_SKELETON
   |      |
   |      |config BR2_PACKAGE_PROVIDES_SKELETON
   |      |    default "skeleton-mine" if BR2_PACKAGE_SKELETON_MINE
   |      `----
   |- package/skeleton-mine/skeleton-mine.mk
   |      |SKELETON_MINE_ADD_TOOLCHAIN_DEPENDENCY = NO
   |      |SKELETON_MINE_ADD_SKELETON_DEPENDENCY = NO
   |      |SKELETON_MINE_PROVIDES = skeleton
   |      |SKELETON_MINE_INSTALL_STAGING = YES
   |      |$(eval $(generic-package))
   |      `----
   |
   |- provides/toolchains.in
   |      |config BR2_TOOLCHAIN_EXTERNAL_MINE
   |      |    bool "my custom toolchain"
   |      |    depends on BR2_some_arch
   |      |    select BR2_INSTALL_LIBSTDCPP
   |      `----
   |- toolchain/toolchain-external-mine/Config.in.options
   |      |if BR2_TOOLCHAIN_EXTERNAL_MINE
   |      |config BR2_TOOLCHAIN_EXTERNAL_PREFIX
   |      |    default "arch-mine-linux-gnu"
   |      |config BR2_PACKAGE_PROVIDES_TOOLCHAIN_EXTERNAL
   |      |    default "toolchain-external-mine"
   |      |endif
   |      `----
   |- toolchain/toolchain-external-mine/toolchain-external-mine.mk
   |      |TOOLCHAIN_EXTERNAL_MINE_SITE = https://example.net/some/place
   |      |TOOLCHAIN_EXTERNAL_MINE_SOURCE = my-toolchain.tar.gz
   |      |$(eval $(toolchain-external-package))
   |      `----
   |
   |- linux/Config.ext.in
   |      |config BR2_LINUX_KERNEL_EXT_EXAMPLE_DRIVER
   |      |    bool "example-external-driver"
   |      |    help
   |      |      Example external driver
   |      |---
   |- linux/linux-ext-example-driver.mk
   |
   |- configs/my-board_defconfig
   |      |BR2_GLOBAL_PATCH_DIR="$(BR2_EXTERNAL_BAR_42_PATH)/patches/"
   |      |BR2_ROOTFS_OVERLAY="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/overlay/"
   |      |BR2_ROOTFS_POST_IMAGE_SCRIPT="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/post-image.sh"
   |      |BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/kernel.config"
   |      `----
   |
   |- patches/linux/0001-some-change.patch
   |- patches/linux/0002-some-other-change.patch
   |- patches/busybox/0001-fix-something.patch
   |
   |- board/my-board/kernel.config
   |- board/my-board/overlay/var/www/index.html
   |- board/my-board/overlay/var/www/my.css
   |- board/my-board/flash-image
   `- board/my-board/post-image.sh
         |#!/bin/sh
         |generate-my-binary-image \
         |    --root ${BINARIES_DIR}/rootfs.tar \
         |    --kernel ${BINARIES_DIR}/zImage \
         |    --dtb ${BINARIES_DIR}/my-board.dtb \
         |    --output ${BINARIES_DIR}/image.bin
         `----
```

The br2-external tree will then be visible in the menuconfig (with the layout expanded):

```
External options  --->
    *** Example br2-external tree (in /path/to/br2-ext-tree/)
    [ ] pkg-1
    [ ] pkg-2
    (0x10AD) my-board flash address
```

If you are using more than one br2-external tree, it would look like (with the layout expanded and the second one with name `FOO_27` but no `desc:` field in `external.desc`):

```
External options  --->
    Example br2-external tree  --->
        *** Example br2-external tree (in /path/to/br2-ext-tree)
        [ ] pkg-1
        [ ] pkg-2
        (0x10AD) my-board flash address
    FOO_27  --->
        *** FOO_27 (in /path/to/another-br2-ext)
        [ ] foo
        [ ] bar
```

Additionally, the jpeg provider will be visible in the jpeg choice:

```
Target packages  --->
    Libraries  --->
        Graphics  --->
            [*] jpeg support
                jpeg variant ()  --->
                    ( ) jpeg
                    ( ) jpeg-turbo
                        *** jpeg from: Example br2-external tree ***
                    (X) my-jpeg
                        *** jpeg from: FOO_27 ***
                    ( ) another-jpeg
```

And similarly for the toolchains:

```
Toolchain  --->
    Toolchain ()  --->
        ( ) Custom toolchain
            *** Toolchains from: Example br2-external tree ***
        (X) my custom toolchain
```

Note. The toolchain options in `toolchain/toolchain-external-mine/Config.in.options` will not appear in the `Toolchain` menu. They must be explicitly included from within the br2-external's top-level `Config.in` and will thus appear in the `External options` menu.

9.3. Storing the Buildroot configuration

The Buildroot configuration can be stored using the command `make savedefconfig`.

This strips the Buildroot configuration down by removing configuration options that are at their default value. The result is stored in a file called `defconfig`. If you want to save it in another place, change the `BR2_DEFCONFIG` option in the Buildroot configuration itself, or call make with `make savedefconfig BR2_DEFCONFIG=<path-to-defconfig>`.

The recommended place to store this defconfig is `configs/<boardname>_defconfig`. If you follow this recommendation, the configuration will be listed in `make list-defconfigs` and can be set again by running `make <boardname>_defconfig`.

Alternatively, you can copy the file to any other place and rebuild with `make defconfig BR2_DEFCONFIG=<path-to-defconfig-file>`.

9.4. Storing the configuration of other components

The configuration files for BusyBox, the Linux kernel, Barebox, U-Boot and uClibc should be stored as well if changed. For each of these components, a Buildroot configuration option exists to point to an input configuration file, e.g. `BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE`. To store their configuration, set these configuration options to a path where you want to save the configuration files, and then use the helper targets described below to actually store the configuration.

As explained in Section 9.1, "Recommended directory structure", the recommended path to store these configuration files is `board/<company>/<boardname>/foo.config`.

Make sure that you create a configuration file *before* changing the `BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE` etc. options. Otherwise, Buildroot will try to access this config file, which doesn't exist yet, and will fail. You can create the configuration file by running `make linux-menuconfig` etc.

Buildroot provides a few helper targets to make the saving of configuration files easier.

- `make linux-update-defconfig` saves the linux configuration to the path specified by `BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE`. It simplifies the config file by removing default values. However, this only works with kernels starting from 2.6.33. For earlier kernels, use `make linux-update-config`.
- `make busybox-update-config` saves the busybox configuration to the path specified by `BR2_PACKAGE_BUSYBOX_CONFIG`.
- `make uclibc-update-config` saves the uClibc configuration to the path specified by `BR2_UCLIBC_CONFIG`.
- `make barebox-update-defconfig` saves the barebox configuration to the path specified by `BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE`.
- `make uboot-update-defconfig` saves the U-Boot configuration to the path specified by `BR2_TARGET_UBOOT_CUSTOM_CONFIG_FILE`.
- For at91bootstrap3, no helper exists so you have to copy the config file manually to `BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE`.

9.5. Customizing the generated target filesystem

Besides changing the configuration through `make *config`, there are a few other ways to customize the resulting target filesystem.

The two recommended methods, which can co-exist, are root filesystem overlay(s) and post build script(s).

Root filesystem overlays (`BR2_ROOTFS_OVERLAY`)
: A filesystem overlay is a tree of files that is copied directly over the target filesystem after it has been built. To enable

this feature, set config option BR2_ROOTFS_OVERLAY (in the System configuration menu) to the root of the overlay. You can even specify multiple overlays, space-separated. If you specify a relative path, it will be relative to the root of the Buildroot tree. Hidden directories of version control systems, like .git, .svn, .hg, etc., files called .empty and files ending in ~ are excluded from the copy.

When BR2_ROOTFS_MERGED_USR is enabled, then the overlay must not contain the */bin*, */lib* or */sbin* directories, as Buildroot will create them as symbolic links to the relevant folders in */usr*. In such a situation, should the overlay have any programs or libraries, they should be placed in */usr/bin*, */usr/sbin* and */usr/lib*.

As shown in Section 9.1, "Recommended directory structure", the recommended path for this overlay is board/<company>/<boardname>/rootfs-overlay.

Post-build scripts (BR2_ROOTFS_POST_BUILD_SCRIPT)

Post-build scripts are shell scripts called *after* Buildroot builds all the selected software, but *before* the rootfs images are assembled. To enable this feature, specify a space-separated list of post-build scripts in config option BR2_ROOTFS_POST_BUILD_SCRIPT (in the System configuration menu). If you specify a relative path, it will be relative to the root of the Buildroot tree.

Using post-build scripts, you can remove or modify any file in your target filesystem. You should, however, use this feature with care. Whenever you find that a certain package generates wrong or unneeded files, you should fix that package rather than work around it with some post-build cleanup scripts.

As shown in Section 9.1, "Recommended directory structure", the recommended path for this script is board/<company>/<boardname>/post_build.sh.

The post-build scripts are run with the main Buildroot tree as current working directory. The path to the target filesystem is passed as the first argument to each script. If the config option BR2_ROOTFS_POST_SCRIPT_ARGS is not empty, these arguments will be passed to the script too. All the scripts will be passed the exact same set of arguments, it is not possible to pass different sets of arguments to each script.

In addition, you may also use these environment variables:

- BR2_CONFIG: the path to the Buildroot .config file
- CONFIG_DIR: the directory containing the .config file, and therefore the top-level Buildroot Makefile to use (which is correct for both in-tree and out-of-tree builds)
- HOST_DIR, STAGING_DIR, TARGET_DIR: see Section 18.6.2, "generic-package reference"
- BUILD_DIR: the directory where packages are extracted and built
- BINARIES_DIR: the place where all binary files (aka images) are stored
- BASE_DIR: the base output directory

Below three more methods of customizing the target filesystem are described, but they are not recommended.

Direct modification of the target filesystem

For temporary modifications, you can modify the target filesystem directly and rebuild the image. The target filesystem is available under output/target/. After making your changes, run make to rebuild the target filesystem image.

This method allows you to do anything to the target filesystem, but if you need to clean your Buildroot tree using make clean, these changes will be lost. Such cleaning is necessary in several cases, refer to Section 8.2, "Understanding when a full rebuild is necessary" for details. This solution is therefore only useful for quick tests: *changes do not survive the* make clean *command*. Once you have validated your changes, you should make sure that they will persist after a make clean, using a root filesystem overlay or a post-build script.

Custom target skeleton (BR2_ROOTFS_SKELETON_CUSTOM)

The root filesystem image is created from a target skeleton, on top of which all packages install their files. The skeleton is copied to the target directory output/target before any package is built and installed. The default target skeleton provides the standard Unix filesystem layout and some basic init scripts and configuration files.

If the default skeleton (available under system/skeleton) does not match your needs, you would typically use a root filesystem overlay or post-build script to adapt it. However, if the default skeleton is entirely different than what you need, using a custom skeleton may be more suitable.

To enable this feature, enable config option BR2_ROOTFS_SKELETON_CUSTOM and set BR2_ROOTFS_SKELETON_CUSTOM_PATH to the path of your custom skeleton. Both options are available in the System configuration menu. If you specify a relative path, it will be relative to the root of the Buildroot tree.

Custom skeletons don't need to contain the */bin*, */lib* or */sbin* directories, since they are created automatically during the build. When BR2_ROOTFS_MERGED_USR is enabled, then the custom skeleton must not contain the */bin*, */lib* or */sbin* directories, as Buildroot will create them as symbolic links to the relevant folders in */usr*. In such a situation, should the skeleton have any programs or libraries, they should be placed in */usr/bin*, */usr/sbin* and */usr/lib*.

This method is not recommended because it duplicates the entire skeleton, which prevents taking advantage of the fixes or improvements brought to the default skeleton in later Buildroot releases.

Post-fakeroot scripts (BR2_ROOTFS_POST_FAKEROOT_SCRIPT)

When aggregating the final images, some parts of the process requires root rights: creating device nodes in /dev, setting permissions or ownership to files and directories… To avoid requiring actual root rights, Buildroot uses fakeroot to simulate root rights. This is not a complete substitute for actually being root, but is enough for what

Buildroot needs.

Post-fakeroot scripts are shell scripts that are called at the *end* of the fakeroot phase, *right before* the filesystem image generator is called. As such, they are called in the fakeroot context.

Post-fakeroot scripts can be useful in case you need to tweak the filesystem to do modifications that are usually only available to the root user.

Note: It is recommended to use the existing mechanisms to set file permissions or create entries in `/dev` (see Section 9.5.1, "Setting file permissions and ownership and adding custom devices nodes") or to create users (see Section 9.6, "Adding custom user accounts")

Note: The difference between post-build scripts (above) and fakeroot scripts, is that post-build scripts are not called in the fakeroot context.

Note: Using `fakeroot` is not an absolute substitute for actually being root. `fakeroot` only ever fakes the file access rights and types (regular, block-or-char device…) and uid/gid; these are emulated in-memory.

9.5.1. Setting file permissions and ownership and adding custom devices nodes

Sometimes it is needed to set specific permissions or ownership on files or device nodes. For example, certain files may need to be owned by root. Since the post-build scripts are not run as root, you cannot do such changes from there unless you use an explicit fakeroot from the post-build script.

Instead, Buildroot provides support for so-called *permission tables*. To use this feature, set config option `BR2_ROOTFS_DEVICE_TABLE` to a space-separated list of permission tables, regular text files following the makedev syntax.

If you are using a static device table (i.e. not using `devtmpfs`, `mdev`, or `(e)udev`) then you can add device nodes using the same syntax, in so-called *device tables*. To use this feature, set config option `BR2_ROOTFS_STATIC_DEVICE_TABLE` to a space-separated list of device tables.

As shown in Section 9.1, "Recommended directory structure", the recommended location for such files is `board/<company>/<boardname>/`.

It should be noted that if the specific permissions or device nodes are related to a specific application, you should set variables `FOO_PERMISSIONS` and `FOO_DEVICES` in the package's `.mk` file instead (see Section 18.6.2, "`generic-package` reference").

9.6. Adding custom user accounts

Sometimes it is needed to add specific users in the target system. To cover this requirement, Buildroot provides support for so-called *users tables*. To use this feature, set config option `BR2_ROOTFS_USERS_TABLES` to a space-separated list of users tables, regular text files following the makeusers syntax.

As shown in Section 9.1, "Recommended directory structure", the recommended location for such files is `board/<company>/<boardname>/`.

It should be noted that if the custom users are related to a specific application, you should set variable `FOO_USERS` in the package's `.mk` file instead (see Section 18.6.2, "`generic-package` reference").

9.7. Customization *after* the images have been created

While post-build scripts (Section 9.5, "Customizing the generated target filesystem") are run *before* building the filesystem image, kernel and bootloader, **post-image scripts** can be used to perform some specific actions *after* all images have been created.

Post-image scripts can for example be used to automatically extract your root filesystem tarball in a location exported by your NFS server, or to create a special firmware image that bundles your root filesystem and kernel image, or any other custom action required for your project.

To enable this feature, specify a space-separated list of post-image scripts in config option `BR2_ROOTFS_POST_IMAGE_SCRIPT` (in the `System configuration` menu). If you specify a relative path, it will be relative to the root of the Buildroot tree.

Just like post-build scripts, post-image scripts are run with the main Buildroot tree as current working directory. The path to the `images` output directory is passed as the first argument to each script. If the config option `BR2_ROOTFS_POST_SCRIPT_ARGS` is not empty, these arguments will be passed to the script too. All the scripts will be passed the exact same set of arguments, it is not possible to pass different sets of arguments to each script.

Again just like for the post-build scripts, the scripts have access to the environment variables `BR2_CONFIG`, `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`, `BUILD_DIR`, `BINARIES_DIR`, `CONFIG_DIR` and `BASE_DIR`.

The post-image scripts will be executed as the user that executes Buildroot, which should normally *not* be the root user. Therefore, any action requiring root permissions in one of these scripts will require special handling (usage of fakeroot or sudo), which is left to the script developer.

9.8. Adding project-specific patches

It is sometimes useful to apply *extra* patches to packages - on top of those provided in Buildroot. This might be used to support custom features in a project, for example, or when working on a new architecture.

The BR2_GLOBAL_PATCH_DIR configuration option can be used to specify a space separated list of one or more directories containing package patches.

For a specific version <packageversion> of a specific package <packagename>, patches are applied from BR2_GLOBAL_PATCH_DIR as follows:

1. For every directory - <global-patch-dir> - that exists in BR2_GLOBAL_PATCH_DIR, a <package-patch-dir> will be determined as follows:
   - <global-patch-dir>/<packagename>/<packageversion>/ if the directory exists.
   - Otherwise, <global-patch-dir>/<packagename> if the directory exists.
2. Patches will then be applied from a <package-patch-dir> as follows:
   - If a series file exists in the package directory, then patches are applied according to the series file;
   - Otherwise, patch files matching *.patch are applied in alphabetical order. So, to ensure they are applied in the right order, it is highly recommended to name the patch files like this: <number>-<description>.patch, where <number> refers to the *apply order*.

For information about how patches are applied for a package, see Section 19.2, "How patches are applied"

The BR2_GLOBAL_PATCH_DIR option is the preferred method for specifying a custom patch directory for packages. It can be used to specify a patch directory for any package in buildroot. It should also be used in place of the custom patch directory options that are available for packages such as U-Boot and Barebox. By doing this, it will allow a user to manage their patches from one top-level directory.

The exception to BR2_GLOBAL_PATCH_DIR being the preferred method for specifying custom patches is BR2_LINUX_KERNEL_PATCH. BR2_LINUX_KERNEL_PATCH should be used to specify kernel patches that are available at a URL. **Note:** BR2_LINUX_KERNEL_PATCH specifies kernel patches that are applied after patches available in BR2_GLOBAL_PATCH_DIR, as it is done from a post-patch hook of the Linux package.

9.9. Adding project-specific packages

In general, any new package should be added directly in the package directory and submitted to the Buildroot upstream project. How to add packages to Buildroot in general is explained in full detail in Chapter 18, *Adding new packages to Buildroot* and will not be repeated here. However, your project may need some proprietary packages that cannot be upstreamed. This section will explain how you can keep such project-specific packages in a project-specific directory.

As shown in Section 9.1, "Recommended directory structure", the recommended location for project-specific packages is package/<company>/. If you are using the br2-external tree feature (see Section 9.2, "Keeping customizations outside of Buildroot") the recommended location is to put them in a sub-directory named package/ in your br2-external tree.

However, Buildroot will not be aware of the packages in this location, unless we perform some additional steps. As explained in Chapter 18, *Adding new packages to Buildroot*, a package in Buildroot basically consists of two files: a .mk file (describing how to build the package) and a Config.in file (describing the configuration options for this package).

Buildroot will automatically include the .mk files in first-level subdirectories of the package directory (using the pattern package/*/*.mk). If we want Buildroot to include .mk files from deeper subdirectories (like package/<company>/package1/) then we simply have to add a .mk file in a first-level subdirectory that includes these additional .mk files. Therefore, create a file package/<company>/<company>.mk with following contents (assuming you have only one extra directory level below package/<company>/):

```
include $(sort $(wildcard package/<company>/*/*.mk))
```

For the Config.in files, create a file package/<company>/Config.in that includes the Config.in files of all your packages. An exhaustive list has to be provided since wildcards are not supported in the source command of kconfig. For example:

```
source "package/<company>/package1/Config.in"
source "package/<company>/package2/Config.in"
```

Include this new file package/<company>/Config.in from package/Config.in, preferably in a company-specific menu to make merges with future Buildroot versions easier.

If using a br2-external tree, refer to Section 9.2, "Keeping customizations outside of Buildroot" for how to fill in those files.

9.10. Quick guide to storing your project-specific customizations

Earlier in this chapter, the different methods for making project-specific customizations have been described. This section will now summarize all this by providing step-by-step instructions to storing your project-specific customizations. Clearly, the steps that are not relevant to your project can be skipped.

1. make menuconfig to configure toolchain, packages and kernel.
2. make linux-menuconfig to update the kernel config, similar for other configuration like busybox, uclibc, …
3. mkdir -p board/<manufacturer>/<boardname>

4. Set the following options to board/<manufacturer>/<boardname>/<package>.config (as far as they are relevant):
   - BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE
   - BR2_PACKAGE_BUSYBOX_CONFIG
   - BR2_UCLIBC_CONFIG
   - BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE
   - BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE
   - BR2_TARGET_UBOOT_CUSTOM_CONFIG_FILE
5. Write the configuration files:
   - make linux-update-defconfig
   - make busybox-update-config
   - make uclibc-update-config
   - cp <output>/build/at91bootstrap3-*/.config board/<manufacturer>/<boardname>/at91bootstrap3.config
   - make barebox-update-defconfig
   - make uboot-update-defconfig
6. Create board/<manufacturer>/<boardname>/rootfs-overlay/ and fill it with additional files you need on your rootfs, e.g. board/<manufacturer>/<boardname>/rootfs-overlay/etc/inittab. Set BR2_ROOTFS_OVERLAY to board/<manufacturer>/<boardname>/rootfs-overlay.
7. Create a post-build script board/<manufacturer>/<boardname>/post_build.sh. Set BR2_ROOTFS_POST_BUILD_SCRIPT to board/<manufacturer>/<boardname>/post_build.sh
8. If additional setuid permissions have to be set or device nodes have to be created, create board/<manufacturer>/<boardname>/device_table.txt and add that path to BR2_ROOTFS_DEVICE_TABLE.
9. If additional user accounts have to be created, create board/<manufacturer>/<boardname>/users_table.txt and add that path to BR2_ROOTFS_USERS_TABLES.
10. To add custom patches to certain packages, set BR2_GLOBAL_PATCH_DIR to board/<manufacturer>/<boardname>/patches/ and add your patches for each package in a subdirectory named after the package. Each patch should be called <packagename>-<num>-<description>.patch.
11. Specifically for the Linux kernel, there also exists the option BR2_LINUX_KERNEL_PATCH with as main advantage that it can also download patches from a URL. If you do not need this, BR2_GLOBAL_PATCH_DIR is preferred. U-Boot, Barebox, at91bootstrap and at91bootstrap3 also have separate options, but these do not provide any advantage over BR2_GLOBAL_PATCH_DIR and will likely be removed in the future.
12. If you need to add project-specific packages, create package/<manufacturer>/ and place your packages in that directory. Create an overall <manufacturer>.mk file that includes the .mk files of all your packages. Create an overall Config.in file that sources the Config.in files of all your packages. Include this Config.in file from Buildroot's package/Config.in file.
13. make savedefconfig to save the buildroot configuration.
14. cp defconfig configs/<boardname>_defconfig

Chapter 10. Integration topics

This chapter discusses how various things are integrated at system level. Buildroot is highly configurable, almost everything discussed here can be changed or overridden by rootfs overlay or custom skeleton configuration.

10.1. Systemd

This chapter describes the decisions taken in Buildroot's integration of systemd, and how various use cases can be implemented.

10.1.1. DBus daemon

Systemd requires a DBus daemon. There are two options for it: traditional dbus (BR2_PACKAGE_DBUS) and bus1 dbus-broker (BR2_PACKAGE_DBUS_BROKER). At least one of them must be chosen. If both are included in the configuration, dbus-broker will be used as system bus, but the traditional dbus-daemon is still installed as well and can be used as session bus. Also its tools (e.g. dbus-send) can be used (systemd itself has busctl as an alternative). In addition, the traditional dbus package is the only one that provides libdbus, which is used by many packages as dbus integration library.

Both in the dbus and in the dbus-broker case, the daemon runs as user dbus. The DBus configuration files are also identical for both.

To make sure that only one of the two daemons is started as system bus, the systemd activation files of the dbus package (dbus.socket and the dbus.service symlink in multi-user.target.wants) are removed when dbus-broker is selected.

10.2. Using SELinux in Buildroot

SELinux is a Linux kernel security module enforcing access control policies. In addition to the traditional file permissions and access control lists, `SELinux` allows to write rules for users or processes to access specific functions of resources (files, sockets…).

SELinux has three modes of operation:

- *Disabled*: the policy is not applied
- *Permissive*: the policy is applied, and non-authorized actions are simply logged. This mode is often used for troubleshooting SELinux issues.
- *Enforcing*: the policy is applied, and non-authorized actions are denied

In Buildroot the mode of operation is controlled by the `BR2_PACKAGE_REFPOLICY_POLICY_STATE_*` configuration options. The Linux kernel also has various configuration options that affect how `SELinux` is enabled (see `security/selinux/Kconfig` in the Linux kernel sources).

By default in Buildroot the `SELinux` policy is provided by the upstream refpolicy project, enabled with `BR2_PACKAGE_REFPOLICY`.

10.2.1. Enabling SELinux support

To have proper support for `SELinux` in a Buildroot generated system, the following configuration options must be enabled:

- `BR2_PACKAGE_LIBSELINUX`
- `BR2_PACKAGE_REFPOLICY`

In addition, your filesystem image format must support extended attributes.

10.2.2. SELinux policy tweaking

The `SELinux refpolicy` contains modules that can be enabled or disabled when being built. Each module provide a number of `SELinux` rules. In Buildroot the non-base modules are disabled by default and several ways to enable such modules are provided:

- Packages can enable a list of `SELinux` modules within the `refpolicy` using the `<packagename>_SELINUX_MODULES` variable.
- Packages can provide additional `SELinux` modules by putting them (.fc, .if and .te files) in `package/<packagename>/selinux/`.
- Extra `SELinux` modules can be added in directories pointed by the `BR2_REFPOLICY_EXTRA_MODULES_DIRS` configuration option.
- Additional modules in the `refpolicy` can be enabled if listed in the `BR2_REFPOLICY_EXTRA_MODULES_DEPENDENCIES` configuration option.

Buildroot also allows to completely override the `refpolicy`. This allows to provide a full custom policy designed specifically for a given system. When going this way, all of the above mechanisms are disabled: no extra `SElinux` module is added to the policy, and all the available modules within the custom policy are enabled and built into the final binary policy. The custom policy must be a fork of the official refpolicy.

In order to fully override the `refpolicy` the following configuration variables have to be set:

- `BR2_PACKAGE_REFPOLICY_CUSTOM_GIT`
- `BR2_PACKAGE_REFPOLICY_CUSTOM_REPO_URL`
- `BR2_PACKAGE_REFPOLICY_CUSTOM_REPO_VERSION`

Chapter 11. Frequently Asked Questions & Troubleshooting

11.1. The boot hangs after *Starting network…*

If the boot process seems to hang after the following messages (messages not necessarily exactly similar, depending on the list of packages selected):

```
Freeing init memory: 3972K
Initializing random number generator... done.
Starting network...
Starting dropbear sshd: generating rsa key... generating dsa key... OK
```

then it means that your system is running, but didn't start a shell on the serial console. In order to have the system start a shell on your serial console, you have to go into the Buildroot configuration, in `System configuration`, modify `Run a getty (login prompt) after boot` and set the appropriate port and baud rate in the `getty options` submenu. This will automatically tune the `/etc/inittab` file of the generated system so that a shell starts on the correct serial port.

11.2. Why is there no compiler on the target?

It has been decided that support for the *native compiler on the target* would be stopped from the Buildroot-2012.11 release because:

- this feature was neither maintained nor tested, and often broken;
- this feature was only available for Buildroot toolchains;
- Buildroot mostly targets *small* or *very small* target hardware with limited resource onboard (CPU, ram, mass-storage), for which compiling on the target does not make much sense;
- Buildroot aims at easing the cross-compilation, making native compilation on the target unnecessary.

If you need a compiler on your target anyway, then Buildroot is not suitable for your purpose. In such case, you need a *real distribution* and you should opt for something like:

- openembedded
- yocto
- Debian
- Fedora
- openSUSE ARM
- Arch Linux ARM
- …

11.3. Why are there no development files on the target?

Since there is no compiler available on the target (see Section 11.2, "Why is there no compiler on the target?"), it does not make sense to waste space with headers or static libraries.

Therefore, those files are always removed from the target since the Buildroot-2012.11 release.

11.4. Why is there no documentation on the target?

Because Buildroot mostly targets *small* or *very small* target hardware with limited resource onboard (CPU, ram, mass-storage), it does not make sense to waste space with the documentation data.

If you need documentation data on your target anyway, then Buildroot is not suitable for your purpose, and you should look for a *real distribution* (see: Section 11.2, "Why is there no compiler on the target?").

11.5. Why are some packages not visible in the Buildroot config menu?

If a package exists in the Buildroot tree and does not appear in the config menu, this most likely means that some of the package's dependencies are not met.

To know more about the dependencies of a package, search for the package symbol in the config menu (see Section 8.1, "*make* tips").

Then, you may have to recursively enable several options (which correspond to the unmet dependencies) to finally be able to select the package.

If the package is not visible due to some unmet toolchain options, then you should certainly run a full rebuild (see Section 8.1, "*make* tips" for more explanations).

11.6. Why not use the target directory as a chroot directory?

There are plenty of reasons to **not** use the target directory a chroot one, among these:

- file ownerships, modes and permissions are not correctly set in the target directory;
- device nodes are not created in the target directory.

For these reasons, commands run through chroot, using the target directory as the new root, will most likely fail.

If you want to run the target filesystem inside a chroot, or as an NFS root, then use the tarball image generated in `images/` and extract it as root.

11.7. Why doesn't Buildroot generate binary packages (.deb, .ipkg…)?

One feature that is often discussed on the Buildroot list is the general topic of "package management". To summarize, the idea would be to add some tracking of which Buildroot package installs what files, with the goals of:

- being able to remove files installed by a package when this package gets unselected from the menuconfig;
- being able to generate binary packages (ipk or other format) that can be installed on the target without re-generating a new root filesystem image.

In general, most people think it is easy to do: just track which package installed what and remove it when the package is unselected. However, it is much more complicated than that:

- It is not only about the `target/` directory, but also the sysroot in `host/<tuple>/sysroot` and the `host/` directory itself. All files installed in those directories by various packages must be tracked.
- When a package is unselected from the configuration, it is not sufficient to remove just the files it installed. One must also remove all its reverse dependencies (i.e. packages relying on it) and rebuild all those packages. For example, package A depends optionally on the OpenSSL library. Both are selected, and Buildroot is built. Package A is built with

crypto support using OpenSSL. Later on, OpenSSL gets unselected from the configuration, but package A remains (since OpenSSL is an optional dependency, this is possible.) If only OpenSSL files are removed, then the files installed by package A are broken: they use a library that is no longer present on the target. Although this is technically doable, it adds a lot of complexity to Buildroot, which goes against the simplicity we try to stick to.

- In addition to the previous problem, there is the case where the optional dependency is not even known to Buildroot. For example, package A in version 1.0 never used OpenSSL, but in version 2.0 it automatically uses OpenSSL if available. If the Buildroot .mk file hasn't been updated to take this into account, then package A will not be part of the reverse dependencies of OpenSSL and will not be removed and rebuilt when OpenSSL is removed. For sure, the .mk file of package A should be fixed to mention this optional dependency, but in the mean time, you can have non-reproducible behaviors.
- The request is to also allow changes in the menuconfig to be applied on the output directory without having to rebuild everything from scratch. However, this is very difficult to achieve in a reliable way: what happens when the suboptions of a package are changed (we would have to detect this, and rebuild the package from scratch and potentially all its reverse dependencies), what happens if toolchain options are changed, etc. At the moment, what Buildroot does is clear and simple so its behaviour is very reliable and it is easy to support users. If configuration changes done in menuconfig are applied after the next make, then it has to work correctly and properly in all situations, and not have some bizarre corner cases. The risk is to get bug reports like "I have enabled package A, B and C, then ran make, then disabled package C and enabled package D and ran make, then re-enabled package C and enabled package E and then there is a build failure". Or worse "I did some configuration, then built, then did some changes, built, some more changes, built, some more changes, built, and now it fails, but I don't remember all the changes I did and in which order". This will be impossible to support.

For all these reasons, the conclusion is that adding tracking of installed files to remove them when the package is unselected, or to generate a repository of binary packages, is something that is very hard to achieve reliably and will add a lot of complexity.

On this matter, the Buildroot developers make this position statement:

- Buildroot strives to make it easy to generate a root filesystem (hence the name, by the way.) That is what we want to make Buildroot good at: building root filesystems.
- Buildroot is not meant to be a distribution (or rather, a distribution generator.) It is the opinion of most Buildroot developers that this is not a goal we should pursue. We believe that there are other tools better suited to generate a distro than Buildroot is. For example, Open Embedded, or openWRT, are such tools.
- We prefer to push Buildroot in a direction that makes it easy (or even easier) to generate complete root filesystems. This is what makes Buildroot stands out in the crowd (among other things, of course!)
- We believe that for most embedded Linux systems, binary packages are not necessary, and potentially harmful. When binary packages are used, it means that the system can be partially upgraded, which creates an enormous number of possible combinations of package versions that should be tested before doing the upgrade on the embedded device. On the other hand, by doing complete system upgrades by upgrading the entire root filesystem image at once, the image deployed to the embedded system is guaranteed to really be the one that has been tested and validated.

11.8. How to speed-up the build process?

Since Buildroot often involves doing full rebuilds of the entire system that can be quite long, we provide below a number of tips to help reduce the build time:

- Use a pre-built external toolchain instead of the default Buildroot internal toolchain. By using a pre-built Linaro toolchain (on ARM) or a Sourcery CodeBench toolchain (for ARM, x86, x86-64, MIPS, etc.), you will save the build time of the toolchain at each complete rebuild, approximately 15 to 20 minutes. Note that temporarily using an external toolchain does not prevent you to switch back to an internal toolchain (that may provide a higher level of customization) once the rest of your system is working;
- Use the `ccache` compiler cache (see: Section 8.13.3, "Using ccache in Buildroot");
- Learn about rebuilding only the few packages you actually care about (see Section 8.3, "Understanding how to rebuild packages"), but beware that sometimes full rebuilds are anyway necessary (see Section 8.2, "Understanding when a full rebuild is necessary");
- Make sure you are not using a virtual machine for the Linux system used to run Buildroot. Most of the virtual machine technologies are known to cause a significant performance impact on I/O, which is really important for building source code;
- Make sure that you're using only local files: do not attempt to do a build over NFS, which significantly slows down the build. Having the Buildroot download folder available locally also helps a bit.
- Buy new hardware. SSDs and lots of RAM are key to speeding up the builds.
- Experiment with top-level parallel build, see Section 8.12, "Top-level parallel build".

Chapter 12. Known issues

- It is not possible to pass extra linker options via `BR2_TARGET_LDFLAGS` if such options contain a `$` sign. For example, the following is known to break: `BR2_TARGET_LDFLAGS="-Wl,-rpath='$ORIGIN/../lib'"`
- The `libffi` package is not supported on the SuperH 2 and ARMv7-M architectures.
- The `prboom` package triggers a compiler failure with the SuperH 4 compiler from Sourcery CodeBench, version 2012.09.

Chapter 13. Legal notice and licensing

13.1. Complying with open source licenses

All of the end products of Buildroot (toolchain, root filesystem, kernel, bootloaders) contain open source software, released under various licenses.

Using open source software gives you the freedom to build rich embedded systems, choosing from a wide range of packages, but also imposes some obligations that you must know and honour. Some licenses require you to publish the license text in the documentation of your product. Others require you to redistribute the source code of the software to those that receive your product.

The exact requirements of each license are documented in each package, and it is your responsibility (or that of your legal office) to comply with those requirements. To make this easier for you, Buildroot can collect for you some material you will probably need. To produce this material, after you have configured Buildroot with `make menuconfig`, `make xconfig` or `make gconfig`, run:

```
make legal-info
```

Buildroot will collect legally-relevant material in your output directory, under the `legal-info/` subdirectory. There you will find:

- A `README` file, that summarizes the produced material and contains warnings about material that Buildroot could not produce.
- `buildroot.config`: this is the Buildroot configuration file that is usually produced with `make menuconfig`, and which is necessary to reproduce the build.
- The source code for all packages; this is saved in the `sources/` and `host-sources/` subdirectories for target and host packages respectively. The source code for packages that set `<PKG>_REDISTRIBUTE = NO` will not be saved. Patches that were applied are also saved, along with a file named `series` that lists the patches in the order they were applied. Patches are under the same license as the files that they modify. Note: Buildroot applies additional patches to Libtool scripts of autotools-based packages. These patches can be found under `support/libtool` in the Buildroot source and, due to technical limitations, are not saved with the package sources. You may need to collect them manually.
- A manifest file (one for host and one for target packages) listing the configured packages, their version, license and related information. Some of this information might not be defined in Buildroot; such items are marked as "unknown".
- The license texts of all packages, in the `licenses/` and `host-licenses/` subdirectories for target and host packages respectively. If the license file(s) are not defined in Buildroot, the file is not produced and a warning in the `README` indicates this.

Please note that the aim of the `legal-info` feature of Buildroot is to produce all the material that is somehow relevant for legal compliance with the package licenses. Buildroot does not try to produce the exact material that you must somehow make public. Certainly, more material is produced than is needed for a strict legal compliance. For example, it produces the source code for packages released under BSD-like licenses, that you are not required to redistribute in source form.

Moreover, due to technical limitations, Buildroot does not produce some material that you will or may need, such as the toolchain source code for some of the external toolchains and the Buildroot source code itself. When you run `make legal-info`, Buildroot produces warnings in the `README` file to inform you of relevant material that could not be saved.

Finally, keep in mind that the output of `make legal-info` is based on declarative statements in each of the packages recipes. The Buildroot developers try to do their best to keep those declarative statements as accurate as possible, to the best of their knowledge. However, it is very well possible that those declarative statements are not all fully accurate nor exhaustive. You (or your legal department) *have* to check the output of `make legal-info` before using it as your own compliance delivery. See the *NO WARRANTY* clauses (clauses 11 and 12) in the `COPYING` file at the root of the Buildroot distribution.

13.2. Complying with the Buildroot license

Buildroot itself is an open source software, released under the GNU General Public License, version 2 or (at your option) any later version, with the exception of the package patches detailed below. However, being a build system, it is not normally part of the end product: if you develop the root filesystem, kernel, bootloader or toolchain for a device, the code of Buildroot is only present on the development machine, not in the device storage.

Nevertheless, the general view of the Buildroot developers is that you should release the Buildroot source code along with the source code of other packages when releasing a product that contains GPL-licensed software. This is because the GNU GPL defines the "*complete source code*" for an executable work as "*all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable*". Buildroot is part of the *scripts used to control compilation and installation of the executable*, and as such it is considered part of the material that must be redistributed.

Keep in mind that this is only the Buildroot developers' opinion, and you should consult your legal department or lawyer in case of any doubt.

13.2.1. Patches to packages

Buildroot also bundles patch files, which are applied to the sources of the various packages. Those patches are not covered by the license of Buildroot. Instead, they are covered by the license of the software to which the patches are applied. When said software is available under multiple licenses, the Buildroot patches are only provided under the publicly accessible licenses.

See Chapter 19, *Patching a package* for the technical details.

Chapter 14. Beyond Buildroot

14.1. Boot the generated images

14.1.1. NFS boot

To achieve NFS-boot, enable *tar root filesystem* in the *Filesystem images* menu.

After a complete build, just run the following commands to setup the NFS-root directory:

```
sudo tar -xavf /path/to/output_dir/rootfs.tar -C /path/to/nfs_root_dir
```

Remember to add this path to `/etc/exports`.

Then, you can execute a NFS-boot from your target.

14.1.2. Live CD

To build a live CD image, enable the *iso image* option in the *Filesystem images* menu. Note that this option is only available on the x86 and x86-64 architectures, and if you are building your kernel with Buildroot.

You can build a live CD image with either IsoLinux, Grub or Grub 2 as a bootloader, but only Isolinux supports making this image usable both as a live CD and live USB (through the *Build hybrid image* option).

You can test your live CD image using QEMU:

```
qemu-system-i386 -cdrom output/images/rootfs.iso9660
```

Or use it as a hard-drive image if it is a hybrid ISO:

```
qemu-system-i386 -hda output/images/rootfs.iso9660
```

It can be easily flashed to a USB drive with `dd`:

```
dd if=output/images/rootfs.iso9660 of=/dev/sdb
```

14.2. Chroot

If you want to chroot in a generated image, then there are few thing you should be aware of:

- you should setup the new root from the *tar root filesystem* image;
- either the selected target architecture is compatible with your host machine, or you should use some `qemu-*` binary and correctly set it within the `binfmt` properties to be able to run the binaries built for the target on your host machine;
- Buildroot does not currently provide `host-qemu` and `binfmt` correctly built and set for that kind of use.

Part III. Developer guide

Chapter 15. How Buildroot works

As mentioned above, Buildroot is basically a set of Makefiles that download, configure, and compile software with the correct options. It also includes patches for various software packages - mainly the ones involved in the cross-compilation toolchain (`gcc`, `binutils` and `uClibc`).

There is basically one Makefile per software package, and they are named with the `.mk` extension. Makefiles are split into many different parts.

- The `toolchain/` directory contains the Makefiles and associated files for all software related to the cross-compilation toolchain: `binutils`, `gcc`, `gdb`, `kernel-headers` and `uClibc`.
- The `arch/` directory contains the definitions for all the processor architectures that are supported by Buildroot.
- The `package/` directory contains the Makefiles and associated files for all user-space tools and libraries that Buildroot can compile and add to the target root filesystem. There is one sub-directory per package.
- The `linux/` directory contains the Makefiles and associated files for the Linux kernel.
- The `boot/` directory contains the Makefiles and associated files for the bootloaders supported by Buildroot.
- The `system/` directory contains support for system integration, e.g. the target filesystem skeleton and the selection of an init system.
- The `fs/` directory contains the Makefiles and associated files for software related to the generation of the target root filesystem image.

Each directory contains at least 2 files:

- `something.mk` is the Makefile that downloads, configures, compiles and installs the package `something`.
- `Config.in` is a part of the configuration tool description file. It describes the options related to the package.

The main Makefile performs the following steps (once the configuration is done):

- Create all the output directories: `staging`, `target`, `build`, etc. in the output directory (`output/` by default, another value can be specified using `O=`)
- Generate the toolchain target. When an internal toolchain is used, this means generating the cross-compilation toolchain. When an external toolchain is used, this means checking the features of the external toolchain and importing it into the Buildroot environment.
- Generate all the targets listed in the `TARGETS` variable. This variable is filled by all the individual components' Makefiles. Generating these targets will trigger the compilation of the userspace packages (libraries, programs), the kernel, the bootloader and the generation of the root filesystem images, depending on the configuration.

Chapter 16. Coding style

Overall, these coding style rules are here to help you to add new files in Buildroot or refactor existing ones.

If you slightly modify some existing file, the important thing is to keep the consistency of the whole file, so you can:

- either follow the potentially deprecated coding style used in this file,
- or entirely rework it in order to make it comply with these rules.

16.1. `Config.in` file

`Config.in` files contain entries for almost anything configurable in Buildroot.

An entry has the following pattern:

```
config BR2_PACKAGE_LIBFOO
        bool "libfoo"
        depends on BR2_PACKAGE_LIBBAZ
        select BR2_PACKAGE_LIBBAR
        help
          This is a comment that explains what libfoo is. The help text
          should be wrapped.

          http://foosoftware.org/libfoo/
```

- The `bool`, `depends on`, `select` and `help` lines are indented with one tab.
- The help text itself should be indented with one tab and two spaces.
- The help text should be wrapped to fit 72 columns, where tab counts for 8, so 62 characters in the text itself.

The `Config.in` files are the input for the configuration tool used in Buildroot, which is the regular *Kconfig*. For further details about the *Kconfig* language, refer to http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

16.2. The `.mk` file

- Header: The file starts with a header. It contains the module name, preferably in lowercase, enclosed between separators made of 80 hashes. A blank line is mandatory after the header:

  ```
  ################################################################################
  #
  # libfoo
  #
  ################################################################################
  ```

- Assignment: use `=` preceded and followed by one space:

  ```
  LIBFOO_VERSION = 1.0
  LIBFOO_CONF_OPTS += --without-python-support
  ```

  Do not align the `=` signs.

- Indentation: use tab only:

  ```
  define LIBFOO_REMOVE_DOC
          $(RM) -fr $(TARGET_DIR)/usr/share/libfoo/doc \
                  $(TARGET_DIR)/usr/share/man/man3/libfoo*
  endef
  ```

  Note that commands inside a `define` block should always start with a tab, so *make* recognizes them as commands.

- Optional dependency:
  - Prefer multi-line syntax.

    YES:

    ```
    ifeq ($(BR2_PACKAGE_PYTHON3),y)
    LIBFOO_CONF_OPTS += --with-python-support
    LIBFOO_DEPENDENCIES += python3
    else
    ```

```
LIBFOO_CONF_OPTS += --without-python-support
endif
```

NO:

```
LIBFOO_CONF_OPTS += --with$(if $(BR2_PACKAGE_PYTHON3),,out)-python-support
LIBFOO_DEPENDENCIES += $(if $(BR2_PACKAGE_PYTHON3),python3,)
```

- Keep configure options and dependencies close together.

- Optional hooks: keep hook definition and assignment together in one if block.

  YES:

```
ifneq ($(BR2_LIBFOO_INSTALL_DATA),y)
define LIBFOO_REMOVE_DATA
        $(RM) -fr $(TARGET_DIR)/usr/share/libfoo/data
endef
LIBFOO_POST_INSTALL_TARGET_HOOKS += LIBFOO_REMOVE_DATA
endif
```

  NO:

```
define LIBFOO_REMOVE_DATA
        $(RM) -fr $(TARGET_DIR)/usr/share/libfoo/data
endef

ifneq ($(BR2_LIBFOO_INSTALL_DATA),y)
LIBFOO_POST_INSTALL_TARGET_HOOKS += LIBFOO_REMOVE_DATA
endif
```

16.3. The `genimage.cfg` file

`genimage.cfg` files contain the output image layout that genimage utility uses to create final .img file.

An example follows:

```
image efi-part.vfat {
        vfat {
                file EFI {
                        image = "efi-part/EFI"
                }

                file Image {
                        image = "Image"
                }
        }

        size = 32M
}

image sdimage.img {
        hdimage {
        }

        partition u-boot {
                image = "efi-part.vfat"
                offset = 8K
        }

        partition root {
                image = "rootfs.ext2"
                size = 512M
        }
}
```

- Every `section`(i.e. hdimage, vfat etc.), `partition` must be indented with one tab.
- Every `file` or other `subnode` must be indented with two tabs.
- Every node(`section`, `partition`, `file`, `subnode`) must have an open curly bracket on the same line of the node's name, while the closing one must be on a newline and after it a newline must be added except for the last one node. Same goes for its option, for example option `size =`.
- Every `option`(i.e. `image`, `offset`, `size`) must have the `=` assignment one space from it and one space from the value specified.
- Filename must at least begin with genimage prefix and have the .cfg extension to be easy to recognize.
- Allowed notations for `offset` and `size` options are: `G`, `M`, `K` (not `k`). If it's not possible to express a precise byte count with notations above then use hexadecimal `0x` prefix or, as last chance, the byte count. In comments instead use `GB`, `MB`, `KB` (not `kb`) in place of `G`, `M`, `K`.
- For GPT partitions, the `partition-type-uuid` value must be `U` for the EFI System Partition (expanded to `c12a7328-f81f-11d2-ba4b-00a0c93ec93b` by *genimage*), `F` for a FAT partition (expanded to `ebd0a0a2-b9e5-4433-87c0-68b6b72699c7` by *genimage*) or `L` for the root filesystem or other filesystems (expanded to `0fc63daf-8483-4772-8e79-3d69d8477de4` by *genimage*). Even though `L` is the default value of *genimage*, we prefer to have it explicitly specified in

our genimage.cfg files. Finally, these shortcuts should be used without double quotes, e.g partition-type-uuid = U. If an explicit GUID is specified, lower-case letters should be used.

The genimage.cfg files are the input for the genimage tool used in Buildroot to generate the final image file(i.e. sdcard.img). For further details about the *genimage* language, refer to https://github.com/pengutronix/genimage/blob/master/README.rst.

16.4. The documentation

The documentation uses the asciidoc format.

For further details about the asciidoc syntax, refer to https://asciidoc-py.github.io/userguide.html.

16.5. Support scripts

Some scripts in the support/ and utils/ directories are written in Python and should follow the PEP8 Style Guide for Python Code.

Chapter 17. Adding support for a particular board

Buildroot contains basic configurations for several publicly available hardware boards, so that users of such a board can easily build a system that is known to work. You are welcome to add support for other boards to Buildroot too.

To do so, you need to create a normal Buildroot configuration that builds a basic system for the hardware: (internal) toolchain, kernel, bootloader, filesystem and a simple BusyBox-only userspace. No specific package should be selected: the configuration should be as minimal as possible, and should only build a working basic BusyBox system for the target platform. You can of course use more complicated configurations for your internal projects, but the Buildroot project will only integrate basic board configurations. This is because package selections are highly application-specific.

Once you have a known working configuration, run make savedefconfig. This will generate a minimal defconfig file at the root of the Buildroot source tree. Move this file into the configs/ directory, and rename it <boardname>_defconfig. If the configuration is a bit more complicated, it is nice to manually reformat it and separate it into sections, with a comment before each section. Typical sections are *Architecture*, *Toolchain options* (typically just linux headers version), *Firmware*, *Bootloader*, *Kernel*, and *Filesystem*.

Always use fixed versions or commit hashes for the different components, not the "latest" version. For example, set BR2_LINUX_KERNEL_CUSTOM_VERSION=y and BR2_LINUX_KERNEL_CUSTOM_VERSION_VALUE to the kernel version you tested with.

It is recommended to use as much as possible upstream versions of the Linux kernel and bootloaders, and to use as much as possible default kernel and bootloader configurations. If they are incorrect for your board, or no default exists, we encourage you to send fixes to the corresponding upstream projects.

However, in the mean time, you may want to store kernel or bootloader configuration or patches specific to your target platform. To do so, create a directory board/<manufacturer> and a subdirectory board/<manufacturer>/<boardname>. You can then store your patches and configurations in these directories, and reference them from the main Buildroot configuration. Refer to Chapter 9, *Project-specific customization* for more details.

Before submitting patches for new boards it is recommended to test it by building it using latest gitlab-CI docker container. To do this use utils/docker-run script and inside it issue these commands:

```
$ make +<boardname>_defconfig+
$ make
```

Chapter 18. Adding new packages to Buildroot

This section covers how new packages (userspace libraries or applications) can be integrated into Buildroot. It also shows how existing packages are integrated, which is needed for fixing issues or tuning their configuration.

When you add a new package, be sure to test it in various conditions (see Section 18.25.3, “How to test your package”) and also check it for coding style (see Section 18.25.2, “How to check the coding style”).

18.1. Package directory

First of all, create a directory under the package directory for your software, for example libfoo.

Some packages have been grouped by topic in a sub-directory: x11r7, qt5 and gstreamer. If your package fits in one of these categories, then create your package directory in these. New subdirectories are discouraged, however.

18.2. Config files

For the package to be displayed in the configuration tool, you need to create a Config file in your package directory. There are two types: Config.in and Config.in.host.

18.2.1. Config.in file

For packages used on the target, create a file named Config.in. This file will contain the option descriptions related to our libfoo software that will be used and displayed in the configuration tool. It should basically contain:

```
config BR2_PACKAGE_LIBFOO
        bool "libfoo"
        help
          This is a comment that explains what libfoo is. The help text
          should be wrapped.

          http://foosoftware.org/libfoo/
```

The `bool` line, `help` line and other metadata information about the configuration option must be indented with one tab. The help text itself should be indented with one tab and two spaces, lines should be wrapped to fit 72 columns, where tab counts for 8, so 62 characters in the text itself. The help text must mention the upstream URL of the project after an empty line.

As a convention specific to Buildroot, the ordering of the attributes is as follows:

1. The type of option: `bool`, `string`··· with the prompt
2. If needed, the `default` value(s)
3. Any dependencies on the target in `depends on` form
4. Any dependencies on the toolchain in `depends on` form
5. Any dependencies on other packages in `depends on` form
6. Any dependency of the `select` form
7. The help keyword and help text.

You can add other sub-options into a `if BR2_PACKAGE_LIBFOO…endif` statement to configure particular things in your software. You can look at examples in other packages. The syntax of the `Config.in` file is the same as the one for the kernel Kconfig file. The documentation for this syntax is available at http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt

Finally you have to add your new `libfoo/Config.in` to `package/Config.in` (or in a category subdirectory if you decided to put your package in one of the existing categories). The files included there are *sorted alphabetically* per category and are *NOT* supposed to contain anything but the *bare* name of the package.

```
source "package/libfoo/Config.in"
```

18.2.2. `Config.in.host` file

Some packages also need to be built for the host system. There are two options here:

- The host package is only required to satisfy build-time dependencies of one or more target packages. In this case, add `host-foo` to the target package's `BAR_DEPENDENCIES` variable. No `Config.in.host` file should be created.
- The host package should be explicitly selectable by the user from the configuration menu. In this case, create a `Config.in.host` file for that host package:

  ```
  config BR2_PACKAGE_HOST_FOO
          bool "host foo"
          help
            This is a comment that explains what foo for the host is.

            http://foosoftware.org/foo/
  ```

  The same coding style and options as for the `Config.in` file are valid.

  Finally you have to add your new `libfoo/Config.in.host` to `package/Config.in.host`. The files included there are *sorted alphabetically* and are *NOT* supposed to contain anything but the *bare* name of the package.

  ```
  source "package/foo/Config.in.host"
  ```

  The host package will then be available from the `Host utilities` menu.

18.2.3. Choosing `depends on` or `select`

The `Config.in` file of your package must also ensure that dependencies are enabled. Typically, Buildroot uses the following rules:

- Use a `select` type of dependency for dependencies on libraries. These dependencies are generally not obvious and it therefore make sense to have the kconfig system ensure that the dependencies are selected. For example, the *libgtk2* package uses `select BR2_PACKAGE_LIBGLIB2` to make sure this library is also enabled. The `select` keyword expresses the dependency with a backward semantic.
- Use a `depends on` type of dependency when the user really needs to be aware of the dependency. Typically, Buildroot uses this type of dependency for dependencies on target architecture, MMU support and toolchain options (see Section 18.2.4, "Dependencies on target and toolchain options"), or for dependencies on "big" things, such as the X.org system. The `depends on` keyword expresses the dependency with a forward semantic.

Note. The current problem with the *kconfig* language is that these two dependency semantics are not internally linked. Therefore, it may be possible to select a package, whom one of its dependencies/requirement is not met.

An example illustrates both the usage of `select` and `depends on`.

```
config BR2_PACKAGE_RRDTOOL
        bool "rrdtool"
        depends on BR2_USE_WCHAR
        select BR2_PACKAGE_FREETYPE
        select BR2_PACKAGE_LIBART
        select BR2_PACKAGE_LIBPNG
        select BR2_PACKAGE_ZLIB
        help
          RRDtool is the OpenSource industry standard, high performance
          data logging and graphing system for time series data.

          http://oss.oetiker.ch/rrdtool/

comment "rrdtool needs a toolchain w/ wchar"
        depends on !BR2_USE_WCHAR
```

Note that these two dependency types are only transitive with the dependencies of the same kind.

This means, in the following example:

```
config BR2_PACKAGE_A
        bool "Package A"

config BR2_PACKAGE_B
        bool "Package B"
        depends on BR2_PACKAGE_A

config BR2_PACKAGE_C
        bool "Package C"
        depends on BR2_PACKAGE_B

config BR2_PACKAGE_D
        bool "Package D"
        select BR2_PACKAGE_B

config BR2_PACKAGE_E
        bool "Package E"
        select BR2_PACKAGE_D
```

- Selecting `Package C` will be visible if `Package B` has been selected, which in turn is only visible if `Package A` has been selected.
- Selecting `Package E` will select `Package D`, which will select `Package B`, it will not check for the dependencies of `Package B`, so it will not select `Package A`.
- Since `Package B` is selected but `Package A` is not, this violates the dependency of `Package B` on `Package A`. Therefore, in such a situation, the transitive dependency has to be added explicitly:

```
config BR2_PACKAGE_D
        bool "Package D"
        depends on BR2_PACKAGE_A
        select BR2_PACKAGE_B

config BR2_PACKAGE_E
        bool "Package E"
        depends on BR2_PACKAGE_A
        select BR2_PACKAGE_D
```

Overall, for package library dependencies, `select` should be preferred.

Note that such dependencies will ensure that the dependency option is also enabled, but not necessarily built before your package. To do so, the dependency also needs to be expressed in the `.mk` file of the package.

Further formatting details: see the coding style.

18.2.4. Dependencies on target and toolchain options

Many packages depend on certain options of the toolchain: the choice of C library, C++ support, thread support, RPC support, wchar support, or dynamic library support. Some packages can only be built on certain target architectures, or if an MMU is available in the processor.

These dependencies have to be expressed with the appropriate *depends on* statements in the Config.in file. Additionally, for dependencies on toolchain options, a `comment` should be displayed when the option is not enabled, so that the user knows why the package is not available. Dependencies on target architecture or MMU support should not be made visible in a comment: since it is unlikely that the user can freely choose another target, it makes little sense to show these dependencies explicitly.

The `comment` should only be visible if the `config` option itself would be visible when the toolchain option dependencies are met. This means that all other dependencies of the package (including dependencies on target architecture and MMU support) have to be repeated on the `comment` definition. To keep it clear, the `depends on` statement for these non-toolchain option should be kept separate from the `depends on` statement for the toolchain options. If there is a dependency on a config option in that same file (typically the main package) it is preferable to have a global `if … endif` construct rather than repeating the `depends on` statement on the comment and other config options.

The general format of a dependency `comment` for package foo is:

```
foo needs a toolchain w/ featA, featB, featC
```

for example:

```
mpd needs a toolchain w/ C++, threads, wchar
```

or

```
crda needs a toolchain w/ threads
```

Note that this text is kept brief on purpose, so that it will fit on a 80-character terminal.

The rest of this section enumerates the different target and toolchain options, the corresponding config symbols to depend on, and the text to use in the comment.

- Target architecture
  - Dependency symbol: `BR2_powerpc`, `BR2_mips`, ⋯ (see `arch/Config.in`)
  - Comment string: no comment to be added
- MMU support
  - Dependency symbol: `BR2_USE_MMU`
  - Comment string: no comment to be added
- Gcc *`_sync*`* built-ins used for atomic operations. They are available in variants operating on 1 byte, 2 bytes, 4 bytes and 8 bytes. Since different architectures support atomic operations on different sizes, one dependency symbol is available for each size:
  - Dependency symbol: `BR2_TOOLCHAIN_HAS_SYNC_1` for 1 byte, `BR2_TOOLCHAIN_HAS_SYNC_2` for 2 bytes, `BR2_TOOLCHAIN_HAS_SYNC_4` for 4 bytes, `BR2_TOOLCHAIN_HAS_SYNC_8` for 8 bytes.
  - Comment string: no comment to be added
- Gcc *`_atomic*`* built-ins used for atomic operations.
  - Dependency symbol: `BR2_TOOLCHAIN_HAS_ATOMIC`.
  - Comment string: no comment to be added
- Kernel headers
  - Dependency symbol: `BR2_TOOLCHAIN_HEADERS_AT_LEAST_X_Y`, (replace `X_Y` with the proper version, see `toolchain/Config.in`)
  - Comment string: `headers >= X.Y` and/or `headers <= X.Y` (replace `X.Y` with the proper version)
- GCC version
  - Dependency symbol: `BR2_TOOLCHAIN_GCC_AT_LEAST_X_Y`, (replace `X_Y` with the proper version, see `toolchain/Config.in`)
  - Comment string: `gcc >= X.Y` and/or `gcc <= X.Y` (replace `X.Y` with the proper version)
- Host GCC version
  - Dependency symbol: `BR2_HOST_GCC_AT_LEAST_X_Y`, (replace `X_Y` with the proper version, see `Config.in`)
  - Comment string: no comment to be added
  - Note that it is usually not the package itself that has a minimum host GCC version, but rather a host-package on which it depends.
- C library
  - Dependency symbol: `BR2_TOOLCHAIN_USES_GLIBC`, `BR2_TOOLCHAIN_USES_MUSL`, `BR2_TOOLCHAIN_USES_UCLIBC`
  - Comment string: for the C library, a slightly different comment text is used: `foo needs a glibc toolchain`, or `foo needs a glibc toolchain w/ C++`
- C++ support
  - Dependency symbol: `BR2_INSTALL_LIBSTDCPP`
  - Comment string: `C++`
- D support
  - Dependency symbol: `BR2_TOOLCHAIN_HAS_DLANG`
  - Comment string: `Dlang`

- Fortran support
  - Dependency symbol: `BR2_TOOLCHAIN_HAS_FORTRAN`
  - Comment string: `fortran`
- thread support
  - Dependency symbol: `BR2_TOOLCHAIN_HAS_THREADS`
  - Comment string: `threads` (unless `BR2_TOOLCHAIN_HAS_THREADS_NPTL` is also needed, in which case, specifying only `NPTL` is sufficient)
- NPTL thread support
  - Dependency symbol: `BR2_TOOLCHAIN_HAS_THREADS_NPTL`
  - Comment string: `NPTL`
- RPC support
  - Dependency symbol: `BR2_TOOLCHAIN_HAS_NATIVE_RPC`
  - Comment string: `RPC`
- wchar support
  - Dependency symbol: `BR2_USE_WCHAR`
  - Comment string: `wchar`
- dynamic library
  - Dependency symbol: `!BR2_STATIC_LIBS`
  - Comment string: `dynamic library`

18.2.5. Dependencies on a Linux kernel built by buildroot

Some packages need a Linux kernel to be built by buildroot. These are typically kernel modules or firmware. A comment should be added in the Config.in file to express this dependency, similar to dependencies on toolchain options. The general format is:

```
foo needs a Linux kernel to be built
```

If there is a dependency on both toolchain options and the Linux kernel, use this format:

```
foo needs a toolchain w/ featA, featB, featC and a Linux kernel to be built
```

18.2.6. Dependencies on udev /dev management

If a package needs udev /dev management, it should depend on symbol `BR2_PACKAGE_HAS_UDEV`, and the following comment should be added:

```
foo needs udev /dev management
```

If there is a dependency on both toolchain options and udev /dev management, use this format:

```
foo needs udev /dev management and a toolchain w/ featA, featB, featC
```

18.2.7. Dependencies on features provided by virtual packages

Some features can be provided by more than one package, such as the openGL libraries.

See Section 18.12, “Infrastructure for virtual packages” for more on the virtual packages.

18.3. The `.mk` file

Finally, here’s the hardest part. Create a file named `libfoo.mk`. It describes how the package should be downloaded, configured, built, installed, etc.

Depending on the package type, the `.mk` file must be written in a different way, using different infrastructures:

- **Makefiles for generic packages** (not using autotools or CMake): These are based on an infrastructure similar to the one used for autotools-based packages, but require a little more work from the developer. They specify what should be done for the configuration, compilation and installation of the package. This infrastructure must be used for all packages that do not use the autotools as their build system. In the future, other specialized infrastructures might be written for other build systems. We cover them through in a tutorial and a reference.
- **Makefiles for autotools-based software** (autoconf, automake, etc.): We provide a dedicated infrastructure for such packages, since autotools is a very common build system. This infrastructure *must* be used for new packages that rely on the autotools as their build system. We cover them through a tutorial and reference.

- **Makefiles for cmake-based software**: We provide a dedicated infrastructure for such packages, as CMake is a more and more commonly used build system and has a standardized behaviour. This infrastructure *must* be used for new packages that rely on CMake. We cover them through a tutorial and reference.
- **Makefiles for Python modules**: We have a dedicated infrastructure for Python modules that use the `distutils`, `flit`, `pep517` or `setuptools` mechanisms. We cover them through a tutorial and a reference.
- **Makefiles for Lua modules**: We have a dedicated infrastructure for Lua modules available through the LuaRocks web site. We cover them through a tutorial and a reference.

Further formatting details: see the writing rules.

18.4. The `.hash` file

When possible, you must add a third file, named `libfoo.hash`, that contains the hashes of the downloaded files for the `libfoo` package. The only reason for not adding a `.hash` file is when hash checking is not possible due to how the package is downloaded.

When a package has a version selection choice, then the hash file may be stored in a subdirectory named after the version, e.g. `package/libfoo/1.2.3/libfoo.hash`. This is especially important if the different versions have different licensing terms, but they are stored in the same file. Otherwise, the hash file should stay in the package's directory.

The hashes stored in that file are used to validate the integrity of the downloaded files and of the license files.

The format of this file is one line for each file for which to check the hash, each line with the following three fields separated by two spaces:

- the type of hash, one of:
  - `md5`, `sha1`, `sha224`, `sha256`, `sha384`, `sha512`
- the hash of the file:
  - for `md5`, 32 hexadecimal characters
  - for `sha1`, 40 hexadecimal characters
  - for `sha224`, 56 hexadecimal characters
  - for `sha256`, 64 hexadecimal characters
  - for `sha384`, 96 hexadecimal characters
  - for `sha512`, 128 hexadecimal characters
- the name of the file:
  - for a source archive: the basename of the file, without any directory component,
  - for a license file: the path as it appears in `FOO_LICENSE_FILES`.

Lines starting with a `#` sign are considered comments, and ignored. Empty lines are ignored.

There can be more than one hash for a single file, each on its own line. In this case, all hashes must match.

Note. Ideally, the hashes stored in this file should match the hashes published by upstream, e.g. on their website, in the e-mail announcement… If upstream provides more than one type of hash (e.g. `sha1` and `sha512`), then it is best to add all those hashes in the `.hash` file. If upstream does not provide any hash, or only provides an `md5` hash, then compute at least one strong hash yourself (preferably `sha256`, but not `md5`), and mention this in a comment line above the hashes.

Note. The hashes for license files are used to detect a license change when a package version is bumped. The hashes are checked during the make legal-info target run. For a package with multiple versions (like Qt5), create the hash file in a subdirectory `<packageversion>` of that package (see also Section 19.2, "How patches are applied").

The example below defines a `sha1` and a `sha256` published by upstream for the main `libfoo-1.2.3.tar.bz2` tarball, an `md5` from upstream and a locally-computed `sha256` hashes for a binary blob, a `sha256` for a downloaded patch, and an archive with no hash:

```
# Hashes from: http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.{sha1,sha256}:
sha1  486fb55c3efa71148fe07895fd713ea3a5ae343a  libfoo-1.2.3.tar.bz2
sha256  efc8103cc3bcb06bda6a781532d12701eb081ad83e8f90004b39ab81b65d4369  libfoo-1.2.3.tar.bz2

# md5 from: http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.md5, sha256 locally computed:
md5  2d608f3c318c6b7557d551a5a09314f03452f1a1  libfoo-data.bin
sha256  01ba4719c80b6fe911b091a7c05124b64eeece964e09c058ef8f9805daca546b  libfoo-data.bin

# Locally computed:
sha256  ff52101fb90bbfc3fe9475e425688c660f46216d7e751c4bbdb1dc85cdccacb9  libfoo-fix-blabla.patch

# Hash for license files:
sha256  a45a845012742796534f7e91fe623262ccfb99460a2bd04015bd28d66fba95b8  COPYING
sha256  01b1f9f2c8ee648a7a596a1abe8aa4ed7899b1c9e5551bda06da6e422b04aa55  doc/COPYING.LGPL
```

If the `.hash` file is present, and it contains one or more hashes for a downloaded file, the hash(es) computed by Buildroot (after download) must match the hash(es) stored in the `.hash` file. If one or more hashes do not match, Buildroot considers this an error, deletes the downloaded file, and aborts.

If the `.hash` file is present, but it does not contain a hash for a downloaded file, Buildroot considers this an error and aborts. However, the downloaded file is left in the download directory since this typically indicates that the `.hash` file is wrong but the downloaded file is probably OK.

Hashes are currently checked for files fetched from http/ftp servers, Git repositories, files copied using scp and local files. Hashes are not checked for other version control systems (such as Subversion, CVS, etc.) because Buildroot currently does not generate reproducible tarballs when source code is fetched from such version control systems.

Hashes should only be added in `.hash` files for files that are guaranteed to be stable. For example, patches auto-generated by Github are not guaranteed to be stable, and therefore their hashes can change over time. Such patches should not be downloaded, and instead be added locally to the package folder.

If the `.hash` file is missing, then no check is done at all.

18.5. The `SNNfoo` start script

Packages that provide a system daemon usually need to be started somehow at boot. Buildroot comes with support for several init systems, some are considered tier one (see Section 6.3, "init system"), while others are also available but do not have the same level of integration. Ideally, all packages providing a system daemon should provide a start script for BusyBox/SysV init and a systemd unit file.

For consistency, the start script must follow the style and composition as shown in the reference: `package/busybox/S01syslogd`. An annotated example of this style is shown below. There is no specific coding style for systemd unit files, but if a package comes with its own unit file, that is preferred over a buildroot specific one, if it is compatible with buildroot.

The name of the start script is composed of the `SNN` and the daemon name. The `NN` is the start order number which needs to be carefully chosen. For example, a program that requires networking to be up should not start before `S40network`. The scripts are started in alphabetical order, so `S01syslogd` starts before `S01watchdogd`, and `S02sysctl` start thereafter.

```
01: #!/bin/sh
02:
03: DAEMON="syslogd"
04: PIDFILE="/var/run/$DAEMON.pid"
05:
06: SYSLOGD_ARGS=""
07:
08: # shellcheck source=/dev/null
09: [ -r "/etc/default/$DAEMON" ] && . "/etc/default/$DAEMON"
10:
11: # BusyBox' syslogd does not create a pidfile, so pass "-n" in the command line
12: # and use "-m" to instruct start-stop-daemon to create one.
13: start() {
14:     printf 'Starting %s: ' "$DAEMON"
15:     # shellcheck disable=SC2086 # we need the word splitting
16:     start-stop-daemon -b -m -S -q -p "$PIDFILE" -x "/sbin/$DAEMON" \
17:             -- -n $SYSLOGD_ARGS
18:     status=$?
19:     if [ "$status" -eq 0 ]; then
20:             echo "OK"
21:     else
22:             echo "FAIL"
23:     fi
24:     return "$status"
25: }
26:
27: stop() {
28:     printf 'Stopping %s: ' "$DAEMON"
29:     start-stop-daemon -K -q -p "$PIDFILE"
30:     status=$?
31:     if [ "$status" -eq 0 ]; then
32:             rm -f "$PIDFILE"
33:             echo "OK"
34:     else
35:             echo "FAIL"
36:     fi
37:     return "$status"
38: }
39:
40: restart() {
41:     stop
42:     sleep 1
43:     start
44: }
45:
46: case "$1" in
47:     start|stop|restart)
48:             "$1";;
49:     reload)
50:             # Restart, since there is no true "reload" feature.
51:             restart;;
52:     *)
53:             echo "Usage: $0 {start|stop|restart|reload}"
54:             exit 1
55: esac
```

Note: programs that support reloading their configuration in some fashion (`SIGHUP`) should provide a `reload()` function

similar to `stop()`. The `start-stop-daemon` supports `-K -s HUP` for this. It is recommended to always append `-x "/sbin/$DAEMON"` to all the `start-stop-daemon` commands to ensure signals are set to a PID that matches `$DAEMON`.

Both start scripts and unit files can source command line arguments from `/etc/default/foo`, in general, if such a file does not exist it should not block the start of the daemon, unless there is some site specirfic command line argument the daemon requires to start. For start scripts a `FOO_ARGS="-s -o -m -e -args"` can be defined to a default value in and the user can override this from `/etc/default/foo`.

18.6. Infrastructure for packages with specific build systems

By *packages with specific build systems* we mean all the packages whose build system is not one of the standard ones, such as *autotools* or *CMake*. This typically includes packages whose build system is based on hand-written Makefiles or shell scripts.

18.6.1. `generic-package` tutorial

```
01: ################################################################################
02: #
03: # libfoo
04: #
05: ################################################################################
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_LICENSE = GPL-3.0+
11: LIBFOO_LICENSE_FILES = COPYING
12: LIBFOO_INSTALL_STAGING = YES
13: LIBFOO_CONFIG_SCRIPTS = libfoo-config
14: LIBFOO_DEPENDENCIES = host-libaaa libbbb
15:
16: define LIBFOO_BUILD_CMDS
17:     $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D) all
18: endef
19:
20: define LIBFOO_INSTALL_STAGING_CMDS
21:     $(INSTALL) -D -m 0755 $(@D)/libfoo.a $(STAGING_DIR)/usr/lib/libfoo.a
22:     $(INSTALL) -D -m 0644 $(@D)/foo.h $(STAGING_DIR)/usr/include/foo.h
23:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(STAGING_DIR)/usr/lib
24: endef
25:
26: define LIBFOO_INSTALL_TARGET_CMDS
27:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(TARGET_DIR)/usr/lib
28:     $(INSTALL) -d -m 0755 $(TARGET_DIR)/etc/foo.d
29: endef
30:
31: define LIBFOO_USERS
32:     foo -1 libfoo -1 * - - - LibFoo daemon
33: endef
34:
35: define LIBFOO_DEVICES
36:     /dev/foo c 666 0 0 42 0 - - -
37: endef
38:
39: define LIBFOO_PERMISSIONS
40:     /bin/foo f 4755 foo libfoo - - - - -
41: endef
42:
43: $(eval $(generic-package))
```

The Makefile begins on line 7 to 11 with metadata information: the version of the package (`LIBFOO_VERSION`), the name of the tarball containing the package (`LIBFOO_SOURCE`) (xz-ed tarball recommended) the Internet location at which the tarball can be downloaded from (`LIBFOO_SITE`), the license (`LIBFOO_LICENSE`) and file with the license text (`LIBFOO_LICENSE_FILES`). All variables must start with the same prefix, `LIBFOO_` in this case. This prefix is always the uppercased version of the package name (see below to understand where the package name is defined).

On line 12, we specify that this package wants to install something to the staging space. This is often needed for libraries, since they must install header files and other development files in the staging space. This will ensure that the commands listed in the `LIBFOO_INSTALL_STAGING_CMDS` variable will be executed.

On line 13, we specify that there is some fixing to be done to some of the *libfoo-config* files that were installed during `LIBFOO_INSTALL_STAGING_CMDS` phase. These *-config files are executable shell script files that are located in *$(STAGING_DIR)/usr/bin* directory and are executed by other 3rd party packages to find out the location and the linking flags of this particular package.

The problem is that all these *-config files by default give wrong, host system linking flags that are unsuitable for cross-compiling.

For example: *-I/usr/include* instead of *-I$(STAGING_DIR)/usr/include* or: *-L/usr/lib* instead of *-L$(STAGING_DIR)/usr/lib*

So some sed magic is done to these scripts to make them give correct flags. The argument to be given to `LIBFOO_CONFIG_SCRIPTS` is the file name(s) of the shell script(s) needing fixing. All these names are relative to *$(STAGING_DIR)/usr/bin* and if needed multiple names can be given.

In addition, the scripts listed in `LIBFOO_CONFIG_SCRIPTS` are removed from `$(TARGET_DIR)/usr/bin`, since they are not needed on the target.

Example 18.1. Config script: *divine* package

Package divine installs shell script *$(STAGING_DIR)/usr/bin/divine-config*.

So its fixup would be:

```
DIVINE_CONFIG_SCRIPTS = divine-config
```

Example 18.2. Config script: *imagemagick* package:

Package imagemagick installs the following scripts:
$(STAGING_DIR)/usr/bin/{Magick,Magick++,MagickCore,MagickWand,Wand}-config

So it's fixup would be:

```
IMAGEMAGICK_CONFIG_SCRIPTS = \
   Magick-config Magick++-config \
   MagickCore-config MagickWand-config Wand-config
```

On line 14, we specify the list of dependencies this package relies on. These dependencies are listed in terms of lower-case package names, which can be packages for the target (without the `host-` prefix) or packages for the host (with the `host-`) prefix). Buildroot will ensure that all these packages are built and installed *before* the current package starts its configuration.

The rest of the Makefile, lines 16..29, defines what should be done at the different steps of the package configuration, compilation and installation. `LIBFOO_BUILD_CMDS` tells what steps should be performed to build the package. `LIBFOO_INSTALL_STAGING_CMDS` tells what steps should be performed to install the package in the staging space. `LIBFOO_INSTALL_TARGET_CMDS` tells what steps should be performed to install the package in the target space.

All these steps rely on the `$(@D)` variable, which contains the directory where the source code of the package has been extracted.

On lines 31..33, we define a user that is used by this package (e.g. to run a daemon as non-root) (`LIBFOO_USERS`).

On line 35..37, we define a device-node file used by this package (`LIBFOO_DEVICES`).

On line 39..41, we define the permissions to set to specific files installed by this package (`LIBFOO_PERMISSIONS`).

Finally, on line 43, we call the `generic-package` function, which generates, according to the variables defined previously, all the Makefile code necessary to make your package working.

18.6.2. `generic-package` reference

There are two variants of the generic target. The `generic-package` macro is used for packages to be cross-compiled for the target. The `host-generic-package` macro is used for host packages, natively compiled for the host. It is possible to call both of them in a single `.mk` file: once to create the rules to generate a target package and once to create the rules to generate a host package:

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

This might be useful if the compilation of the target package requires some tools to be installed on the host. If the package name is `libfoo`, then the name of the package for the target is also `libfoo`, while the name of the package for the host is `host-libfoo`. These names should be used in the DEPENDENCIES variables of other packages, if they depend on `libfoo` or `host-libfoo`.

The call to the `generic-package` and/or `host-generic-package` macro **must** be at the end of the `.mk` file, after all variable definitions. The call to `host-generic-package` **must** be after the call to `generic-package`, if any.

For the target package, the `generic-package` uses the variables defined by the .mk file and prefixed by the uppercased package name: `LIBFOO_*`. `host-generic-package` uses the `HOST_LIBFOO_*` variables. For *some* variables, if the `HOST_LIBFOO_` prefixed variable doesn't exist, the package infrastructure uses the corresponding variable prefixed by `LIBFOO_`. This is done for variables that are likely to have the same value for both the target and host packages. See below for details.

The list of variables that can be set in a `.mk` file to give metadata information is (assuming the package name is `libfoo`) :

- `LIBFOO_VERSION`, mandatory, must contain the version of the package. Note that if `HOST_LIBFOO_VERSION` doesn't exist, it is assumed to be the same as `LIBFOO_VERSION`. It can also be a revision number or a tag for packages that are fetched directly from their version control system. Examples:
  - a version for a release tarball: `LIBFOO_VERSION = 0.1.2`
  - a sha1 for a git tree: `LIBFOO_VERSION = cb9d6aa9429e838f0e54faa3d455bcbab5eef057`
  - a tag for a git tree `LIBFOO_VERSION = v0.1.2`

Note: Using a branch name as `FOO_VERSION` is not supported, because it does not and can not work as people would expect it should:

1. due to local caching, Buildroot will not re-fetch the repository, so people who expect to be able to follow the remote repository would be quite surprised and disappointed;
2. because two builds can never be perfectly simultaneous, and because the remote repository may get new commits on the branch anytime, two users, using the same Buildroot tree and building the same configuration, may get different source, thus rendering the build non reproducible, and people would be quite surprised and disappointed.

- `LIBFOO_SOURCE` may contain the name of the tarball of the package, which Buildroot will use to download the tarball from `LIBFOO_SITE`. If `HOST_LIBFOO_SOURCE` is not specified, it defaults to `LIBFOO_SOURCE`. If none are specified, then the value is assumed to be `libfoo-$(LIBFOO_VERSION).tar.gz`. Example: `LIBFOO_SOURCE = foobar-$(LIBFOO_VERSION).tar.bz2`
- `LIBFOO_PATCH` may contain a space-separated list of patch file names, that Buildroot will download and apply to the package source code. If an entry contains `://`, then Buildroot will assume it is a full URL and download the patch from this location. Otherwise, Buildroot will assume that the patch should be downloaded from `LIBFOO_SITE`. If `HOST_LIBFOO_PATCH` is not specified, it defaults to `LIBFOO_PATCH`. Note that patches that are included in Buildroot itself use a different mechanism: all files of the form `*.patch` present in the package directory inside Buildroot will be applied to the package after extraction (see patching a package). Finally, patches listed in the `LIBFOO_PATCH` variable are applied *before* the patches stored in the Buildroot package directory.
- `LIBFOO_SITE` provides the location of the package, which can be a URL or a local filesystem path. HTTP, FTP and SCP are supported URL types for retrieving package tarballs. In these cases don't include a trailing slash: it will be added by Buildroot between the directory and the filename as appropriate. Git, Subversion, Mercurial, and Bazaar are supported URL types for retrieving packages directly from source code management systems. There is a helper function to make it easier to download source tarballs from GitHub (refer to Section 18.25.4, "How to add a package from GitHub" for details). A filesystem path may be used to specify either a tarball or a directory containing the package source code. See `LIBFOO_SITE_METHOD` below for more details on how retrieval works. Note that SCP URLs should be of the form `scp://[user@]host:filepath`, and that filepath is relative to the user's home directory, so you may want to prepend the path with a slash for absolute paths: `scp://[user@]host:/absolutepath`. The same goes for SFTP URLs. If `HOST_LIBFOO_SITE` is not specified, it defaults to `LIBFOO_SITE`. Examples: `LIBFOO_SITE=http://www.libfoosoftware.org/libfoo` `LIBFOO_SITE=http://svn.xiph.org/trunk/Tremor` `LIBFOO_SITE=/opt/software/libfoo.tar.gz` `LIBFOO_SITE=$(TOPDIR)/../src/libfoo`
- `LIBFOO_DL_OPTS` is a space-separated list of additional options to pass to the downloader. Useful for retrieving documents with server-side checking for user logins and passwords, or to use a proxy. All download methods valid for `LIBFOO_SITE_METHOD` are supported; valid options depend on the download method (consult the man page for the respective download utilities).
- `LIBFOO_EXTRA_DOWNLOADS` is a space-separated list of additional files that Buildroot should download. If an entry contains `://` then Buildroot will assume it is a complete URL and will download the file using this URL. Otherwise, Buildroot will assume the file to be downloaded is located at `LIBFOO_SITE`. Buildroot will not do anything with those additional files, except download them: it will be up to the package recipe to use them from `$(LIBFOO_DL_DIR)`.
- `LIBFOO_SITE_METHOD` determines the method used to fetch or copy the package source code. In many cases, Buildroot guesses the method from the contents of `LIBFOO_SITE` and setting `LIBFOO_SITE_METHOD` is unnecessary. When `HOST_LIBFOO_SITE_METHOD` is not specified, it defaults to the value of `LIBFOO_SITE_METHOD`. The possible values of `LIBFOO_SITE_METHOD` are:
  - `wget` for normal FTP/HTTP downloads of tarballs. Used by default when `LIBFOO_SITE` begins with `http://`, `https://` or `ftp://`.
  - `scp` for downloads of tarballs over SSH with scp. Used by default when `LIBFOO_SITE` begins with `scp://`.
  - `sftp` for downloads of tarballs over SSH with sftp. Used by default when `LIBFOO_SITE` begins with `sftp://`.
  - `svn` for retrieving source code from a Subversion repository. Used by default when `LIBFOO_SITE` begins with `svn://`. When a `http://` Subversion repository URL is specified in `LIBFOO_SITE`, one *must* specify `LIBFOO_SITE_METHOD=svn`. Buildroot performs a checkout which is preserved as a tarball in the download cache; subsequent builds use the tarball instead of performing another checkout.
  - `cvs` for retrieving source code from a CVS repository. Used by default when `LIBFOO_SITE` begins with `cvs://`. The downloaded source code is cached as with the `svn` method. Anonymous pserver mode is assumed otherwise explicitly defined on `LIBFOO_SITE`. Both `LIBFOO_SITE=cvs://libfoo.net:/cvsroot/libfoo` and `LIBFOO_SITE=cvs://:ext:libfoo.net:/cvsroot/libfoo` are accepted, on the former anonymous pserver access mode is assumed. `LIBFOO_SITE` *must* contain the source URL as well as the remote repository directory. The module is the package name. `LIBFOO_VERSION` is *mandatory* and *must* be a tag, a branch, or a date (e.g. "2014-10-20", "2014-10-20 13:45", "2014-10-20 13:45+01" see "man cvs" for further details).
  - `git` for retrieving source code from a Git repository. Used by default when `LIBFOO_SITE` begins with `git://`. The downloaded source code is cached as with the `svn` method.
  - `hg` for retrieving source code from a Mercurial repository. One *must* specify `LIBFOO_SITE_METHOD=hg` when `LIBFOO_SITE` contains a Mercurial repository URL. The downloaded source code is cached as with the `svn` method.
  - `bzr` for retrieving source code from a Bazaar repository. Used by default when `LIBFOO_SITE` begins with `bzr://`. The downloaded source code is cached as with the `svn` method.
  - `file` for a local tarball. One should use this when `LIBFOO_SITE` specifies a package tarball as a local filename. Useful for software that isn't available publicly or in version control.

- `local` for a local source code directory. One should use this when `LIBFOO_SITE` specifies a local directory path containing the package source code. Buildroot copies the contents of the source directory into the package's build directory. Note that for `local` packages, no patches are applied. If you need to still patch the source code, use `LIBFOO_POST_RSYNC_HOOKS`, see Section 18.23.1, "Using the `POST_RSYNC` hook".

- `LIBFOO_GIT_SUBMODULES` can be set to `YES` to create an archive with the git submodules in the repository. This is only available for packages downloaded with git (i.e. when `LIBFOO_SITE_METHOD=git`). Note that we try not to use such git submodules when they contain bundled libraries, in which case we prefer to use those libraries from their own package.
- `LIBFOO_GIT_LFS` should be set to `YES` if the Git repository uses Git LFS to store large files out of band. This is only available for packages downloaded with git (i.e. when `LIBFOO_SITE_METHOD=git`).
- `LIBFOO_STRIP_COMPONENTS` is the number of leading components (directories) that tar must strip from file names on extraction. The tarball for most packages has one leading component named "<pkg-name>-<pkg-version>", thus Buildroot passes --strip-components=1 to tar to remove it. For non-standard packages that don't have this component, or that have more than one leading component to strip, set this variable with the value to be passed to tar. Default: 1.
- `LIBFOO_EXCLUDES` is a space-separated list of patterns to exclude when extracting the archive. Each item from that list is passed as a tar's `--exclude` option. By default, empty.
- `LIBFOO_DEPENDENCIES` lists the dependencies (in terms of package name) that are required for the current target package to compile. These dependencies are guaranteed to be compiled and installed before the configuration of the current package starts. However, modifications to configuration of these dependencies will not force a rebuild of the current package. In a similar way, `HOST_LIBFOO_DEPENDENCIES` lists the dependencies for the current host package.
- `LIBFOO_EXTRACT_DEPENDENCIES` lists the dependencies (in terms of package name) that are required for the current target package to be extracted. These dependencies are guaranteed to be compiled and installed before the extract step of the current package starts. This is only used internally by the package infrastructure, and should typically not be used directly by packages.
- `LIBFOO_PATCH_DEPENDENCIES` lists the dependencies (in terms of package name) that are required for the current package to be patched. These dependencies are guaranteed to be extracted and patched (but not necessarily built) before the current package is patched. In a similar way, `HOST_LIBFOO_PATCH_DEPENDENCIES` lists the dependencies for the current host package. This is seldom used; usually, `LIBFOO_DEPENDENCIES` is what you really want to use.
- `LIBFOO_PROVIDES` lists all the virtual packages `libfoo` is an implementation of. See Section 18.12, "Infrastructure for virtual packages".
- `LIBFOO_INSTALL_STAGING` can be set to `YES` or `NO` (default). If set to `YES`, then the commands in the `LIBFOO_INSTALL_STAGING_CMDS` variables are executed to install the package into the staging directory.
- `LIBFOO_INSTALL_TARGET` can be set to `YES` (default) or `NO`. If set to `YES`, then the commands in the `LIBFOO_INSTALL_TARGET_CMDS` variables are executed to install the package into the target directory.
- `LIBFOO_INSTALL_IMAGES` can be set to `YES` or `NO` (default). If set to `YES`, then the commands in the `LIBFOO_INSTALL_IMAGES_CMDS` variable are executed to install the package into the images directory.
- `LIBFOO_CONFIG_SCRIPTS` lists the names of the files in *$(STAGING_DIR)/usr/bin* that need some special fixing to make them cross-compiling friendly. Multiple file names separated by space can be given and all are relative to *$(STAGING_DIR)/usr/bin*. The files listed in `LIBFOO_CONFIG_SCRIPTS` are also removed from `$(TARGET_DIR)/usr/bin` since they are not needed on the target.
- `LIBFOO_DEVICES` lists the device files to be created by Buildroot when using the static device table. The syntax to use is the makedevs one. You can find some documentation for this syntax in the Chapter 25, *Makedev syntax documentation*. This variable is optional.
- `LIBFOO_PERMISSIONS` lists the changes of permissions to be done at the end of the build process. The syntax is once again the makedevs one. You can find some documentation for this syntax in the Chapter 25, *Makedev syntax documentation*. This variable is optional.
- `LIBFOO_USERS` lists the users to create for this package, if it installs a program you want to run as a specific user (e.g. as a daemon, or as a cron-job). The syntax is similar in spirit to the makedevs one, and is described in the Chapter 26, *Makeusers syntax documentation*. This variable is optional.
- `LIBFOO_LICENSE` defines the license (or licenses) under which the package is released. This name will appear in the manifest file produced by `make legal-info`. If the license appears in the SPDX License List, use the SPDX short identifier to make the manifest file uniform. Otherwise, describe the license in a precise and concise way, avoiding ambiguous names such as `BSD` which actually name a family of licenses. This variable is optional. If it is not defined, `unknown` will appear in the `license` field of the manifest file for this package. The expected format for this variable must comply with the following rules:
  - If different parts of the package are released under different licenses, then `comma` separate licenses (e.g. `LIBFOO_LICENSE = GPL-2.0+, LGPL-2.1+`). If there is clear distinction between which component is licensed under what license, then annotate the license with that component, between parenthesis (e.g. `LIBFOO_LICENSE = GPL-2.0+ (programs), LGPL-2.1+ (libraries)`).
  - If some licenses are conditioned on a sub-option being enabled, append the conditional licenses with a comma (e.g.: `FOO_LICENSE += , GPL-2.0+ (programs)`); the infrastructure will internally remove the space before the comma.
  - If the package is dual licensed, then separate licenses with the `or` keyword (e.g. `LIBFOO_LICENSE = AFL-2.1 or GPL-2.0+`).
- `LIBFOO_LICENSE_FILES` is a space-separated list of files in the package tarball that contain the license(s) under which the package is released. `make legal-info` copies all of these files in the `legal-info` directory. See Chapter 13, *Legal notice and licensing* for more information. This variable is optional. If it is not defined, a warning will be produced to let you know, and `not saved` will appear in the `license files` field of the manifest file for this package.

- LIBFOO_ACTUAL_SOURCE_TARBALL only applies to packages whose LIBFOO_SITE / LIBFOO_SOURCE pair points to an archive that does not actually contain source code, but binary code. This a very uncommon case, only known to apply to external toolchains which come already compiled, although theoretically it might apply to other packages. In such cases a separate tarball is usually available with the actual source code. Set LIBFOO_ACTUAL_SOURCE_TARBALL to the name of the actual source code archive and Buildroot will download it and use it when you run make legal-info to collect legally-relevant material. Note this file will not be downloaded during regular builds nor by make source.
- LIBFOO_ACTUAL_SOURCE_SITE provides the location of the actual source tarball. The default value is LIBFOO_SITE, so you don't need to set this variable if the binary and source archives are hosted on the same directory. If LIBFOO_ACTUAL_SOURCE_TARBALL is not set, it doesn't make sense to define LIBFOO_ACTUAL_SOURCE_SITE.
- LIBFOO_REDISTRIBUTE can be set to YES (default) or NO to indicate if the package source code is allowed to be redistributed. Set it to NO for non-opensource packages: Buildroot will not save the source code for this package when collecting the legal-info.
- LIBFOO_FLAT_STACKSIZE defines the stack size of an application built into the FLAT binary format. The application stack size on the NOMMU architecture processors can't be enlarged at run time. The default stack size for the FLAT binary format is only 4k bytes. If the application consumes more stack, append the required number here.
- LIBFOO_BIN_ARCH_EXCLUDE is a space-separated list of paths (relative to the target directory) to ignore when checking that the package installs correctly cross-compiled binaries. You seldom need to set this variable, unless the package installs binary blobs outside the default locations, /lib/firmware, /usr/lib/firmware, /lib/modules, /usr/lib/modules, and /usr/share, which are automatically excluded.
- LIBFOO_IGNORE_CVES is a space-separated list of CVEs that tells Buildroot CVE tracking tools which CVEs should be ignored for this package. This is typically used when the CVE is fixed by a patch in the package, or when the CVE for some reason does not affect the Buildroot package. A Makefile comment must always precede the addition of a CVE to this variable. Example:

```
# 0001-fix-cve-2020-12345.patch
LIBFOO_IGNORE_CVES += CVE-2020-12345
# only when built with libbaz, which Buildroot doesn't support
LIBFOO_IGNORE_CVES += CVE-2020-54321
```

- LIBFOO_CPE_ID_* variables is a set of variables that allows the package to define its CPE identifier. The available variables are:
  - LIBFOO_CPE_ID_PREFIX, specifies the prefix of the CPE identifier, i.e the first three fields. When not defined, the default value is cpe:2.3:a.
  - LIBFOO_CPE_ID_VENDOR, specifies the vendor part of the CPE identifier. When not defined, the default value is <pkgname>_project.
  - LIBFOO_CPE_ID_PRODUCT, specifies the product part of the CPE identifier. When not defined, the default value is <pkgname>.
  - LIBFOO_CPE_ID_VERSION, specifies the version part of the CPE identifier. When not defined the default value is $(LIBFOO_VERSION).
  - LIBFOO_CPE_ID_UPDATE specifies the *update* part of the CPE identifier. When not defined the default value is *.

  If any of those variables is defined, then the generic package infrastructure assumes the package provides valid CPE information. In this case, the generic package infrastructure will define LIBFOO_CPE_ID.

  For a host package, if its LIBFOO_CPE_ID_* variables are not defined, it inherits the value of those variables from the corresponding target package.

The recommended way to define these variables is to use the following syntax:

```
LIBFOO_VERSION = 2.32
```

Now, the variables that define what should be performed at the different steps of the build process.

- LIBFOO_EXTRACT_CMDS lists the actions to be performed to extract the package. This is generally not needed as tarballs are automatically handled by Buildroot. However, if the package uses a non-standard archive format, such as a ZIP or RAR file, or has a tarball with a non-standard organization, this variable allows to override the package infrastructure default behavior.
- LIBFOO_CONFIGURE_CMDS lists the actions to be performed to configure the package before its compilation.
- LIBFOO_BUILD_CMDS lists the actions to be performed to compile the package.
- HOST_LIBFOO_INSTALL_CMDS lists the actions to be performed to install the package, when the package is a host package. The package must install its files to the directory given by $(HOST_DIR). All files, including development files such as headers should be installed, since other packages might be compiled on top of this package.
- LIBFOO_INSTALL_TARGET_CMDS lists the actions to be performed to install the package to the target directory, when the package is a target package. The package must install its files to the directory given by $(TARGET_DIR). Only the files required for *execution* of the package have to be installed. Header files, static libraries and documentation will be removed again when the target filesystem is finalized.
- LIBFOO_INSTALL_STAGING_CMDS lists the actions to be performed to install the package to the staging directory, when the package is a target package. The package must install its files to the directory given by $(STAGING_DIR). All development files should be installed, since they might be needed to compile other packages.

- `LIBFOO_INSTALL_IMAGES_CMDS` lists the actions to be performed to install the package to the images directory, when the package is a target package. The package must install its files to the directory given by `$(BINARIES_DIR)`. Only files that are binary images (aka images) that do not belong in the `TARGET_DIR` but are necessary for booting the board should be placed here. For example, a package should utilize this step if it has binaries which would be similar to the kernel image, bootloader or root filesystem images.
- `LIBFOO_INSTALL_INIT_SYSV`, `LIBFOO_INSTALL_INIT_OPENRC` and `LIBFOO_INSTALL_INIT_SYSTEMD` list the actions to install init scripts either for the systemV-like init systems (busybox, sysvinit, etc.), openrc or for the systemd units. These commands will be run only when the relevant init system is installed (i.e. if systemd is selected as the init system in the configuration, only `LIBFOO_INSTALL_INIT_SYSTEMD` will be run). The only exception is when openrc is chosen as init system and `LIBFOO_INSTALL_INIT_OPENRC` has not been set, in such situation `LIBFOO_INSTALL_INIT_SYSV` will be called, since openrc supports sysv init scripts. When systemd is used as the init system, buildroot will automatically enable all services using the `systemctl preset-all` command in the final phase of image building. You can add preset files to prevent a particular unit from being automatically enabled by buildroot.
- `LIBFOO_HELP_CMDS` lists the actions to print the package help, which is included to the main `make help` output. These commands can print anything in any format. This is seldom used, as packages rarely have custom rules. **Do not use this variable**, unless you really know that you need to print help.
- `LIBFOO_LINUX_CONFIG_FIXUPS` lists the Linux kernel configuration options that are needed to build and use this package, and without which the package is fundamentally broken. This shall be a set of calls to one of the kconfig tweaking option: `KCONFIG_ENABLE_OPT`, `KCONFIG_DISABLE_OPT`, or `KCONFIG_SET_OPT`. This is seldom used, as package usually have no strict requirements on the kernel options.

The preferred way to define these variables is:

```
define LIBFOO_CONFIGURE_CMDS
        action 1
        action 2
        action 3
endef
```

In the action definitions, you can use the following variables:

- `$(LIBFOO_PKGDIR)` contains the path to the directory containing the `libfoo.mk` and `Config.in` files. This variable is useful when it is necessary to install a file bundled in Buildroot, like a runtime configuration file, a splashscreen image…
- `$(@D)`, which contains the directory in which the package source code has been uncompressed.
- `$(LIBFOO_DL_DIR)` contains the path to the directory where all the downloads made by Buildroot for `libfoo` are stored in.
- `$(TARGET_CC)`, `$(TARGET_LD)`, etc. to get the target cross-compilation utilities
- `$(TARGET_CROSS)` to get the cross-compilation toolchain prefix
- Of course the `$(HOST_DIR)`, `$(STAGING_DIR)` and `$(TARGET_DIR)` variables to install the packages properly. Those variables point to the global *host*, *staging* and *target* directories, unless *per-package directory* support is used, in which case they point to the current package *host*, *staging* and *target* directories. In both cases, it doesn't make any difference from the package point of view: it should simply use `HOST_DIR`, `STAGING_DIR` and `TARGET_DIR`. See Section 8.12, "Top-level parallel build" for more details about *per-package directory* support.

Finally, you can also use hooks. See Section 18.23, "Hooks available in the various build steps" for more information.

18.7. Infrastructure for autotools-based packages

18.7.1. `autotools-package` tutorial

First, let's see how to write a `.mk` file for an autotools-based package, with an example :

```
01: ################################################################################
02: #
03: # libfoo
04: #
05: ################################################################################
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_INSTALL_STAGING = YES
11: LIBFOO_INSTALL_TARGET = NO
12: LIBFOO_CONF_OPTS = --disable-shared
13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf
14:
15: $(eval $(autotools-package))
```

On line 7, we declare the version of the package.

On line 8 and 9, we declare the name of the tarball (xz-ed tarball recommended) and the location of the tarball on the Web. Buildroot will automatically download the tarball from this location.

On line 10, we tell Buildroot to install the package to the staging directory. The staging directory, located in `output/staging/` is the directory where all the packages are installed, including their development files, etc. By default, packages are not installed to the staging directory, since usually, only libraries need to be installed in the staging directory:

their development files are needed to compile other libraries or applications depending on them. Also by default, when staging installation is enabled, packages are installed in this location using the `make install` command.

On line 11, we tell Buildroot to not install the package to the target directory. This directory contains what will become the root filesystem running on the target. For purely static libraries, it is not necessary to install them in the target directory because they will not be used at runtime. By default, target installation is enabled; setting this variable to NO is almost never needed. Also by default, packages are installed in this location using the `make install` command.

On line 12, we tell Buildroot to pass a custom configure option, that will be passed to the `./configure` script before configuring and building the package.

On line 13, we declare our dependencies, so that they are built before the build process of our package starts.

Finally, on line line 15, we invoke the `autotools-package` macro that generates all the Makefile rules that actually allows the package to be built.

18.7.2. `autotools-package` reference

The main macro of the autotools package infrastructure is `autotools-package`. It is similar to the `generic-package` macro. The ability to have target and host packages is also available, with the `host-autotools-package` macro.

Just like the generic infrastructure, the autotools infrastructure works by defining a number of variables before calling the `autotools-package` macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the autotools infrastructure: `LIBFOO_VERSION`, `LIBFOO_SOURCE`, `LIBFOO_PATCH`, `LIBFOO_SITE`, `LIBFOO_SUBDIR`, `LIBFOO_DEPENDENCIES`, `LIBFOO_INSTALL_STAGING`, `LIBFOO_INSTALL_TARGET`.

A few additional variables, specific to the autotools infrastructure, can also be defined. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them.

- `LIBFOO_SUBDIR` may contain the name of a subdirectory inside the package that contains the configure script. This is useful, if for example, the main configure script is not at the root of the tree extracted by the tarball. If `HOST_LIBFOO_SUBDIR` is not specified, it defaults to `LIBFOO_SUBDIR`.
- `LIBFOO_CONF_ENV`, to specify additional environment variables to pass to the configure script. By default, empty.
- `LIBFOO_CONF_OPTS`, to specify additional configure options to pass to the configure script. By default, empty.
- `LIBFOO_MAKE`, to specify an alternate `make` command. This is typically useful when parallel make is enabled in the configuration (using `BR2_JLEVEL`) but that this feature should be disabled for the given package, for one reason or another. By default, set to `$(MAKE)`. If parallel building is not supported by the package, then it should be set to `LIBFOO_MAKE=$(MAKE1)`.
- `LIBFOO_MAKE_ENV`, to specify additional environment variables to pass to make in the build step. These are passed before the `make` command. By default, empty.
- `LIBFOO_MAKE_OPTS`, to specify additional variables to pass to make in the build step. These are passed after the `make` command. By default, empty.
- `LIBFOO_AUTORECONF`, tells whether the package should be autoreconfigured or not (i.e. if the configure script and Makefile.in files should be re-generated by re-running autoconf, automake, libtool, etc.). Valid values are `YES` and `NO`. By default, the value is `NO`
- `LIBFOO_AUTORECONF_ENV`, to specify additional environment variables to pass to the *autoreconf* program if `LIBFOO_AUTORECONF=YES`. These are passed in the environment of the *autoreconf* command. By default, empty.
- `LIBFOO_AUTORECONF_OPTS` to specify additional options passed to the *autoreconf* program if `LIBFOO_AUTORECONF=YES`. By default, empty.
- `LIBFOO_GETTEXTIZE`, tells whether the package should be gettextized or not (i.e. if the package uses a different gettext version than Buildroot provides, and it is needed to run *gettextize*.) Only valid when `LIBFOO_AUTORECONF=YES`. Valid values are `YES` and `NO`. The default is `NO`.
- `LIBFOO_GETTEXTIZE_OPTS`, to specify additional options passed to the *gettextize* program, if `LIBFOO_GETTEXTIZE=YES`. You may use that if, for example, the `.po` files are not located in the standard place (i.e. in `po/` at the root of the package.) By default, *-f*.
- `LIBFOO_LIBTOOL_PATCH` tells whether the Buildroot patch to fix libtool cross-compilation issues should be applied or not. Valid values are `YES` and `NO`. By default, the value is `YES`
- `LIBFOO_INSTALL_STAGING_OPTS` contains the make options used to install the package to the staging directory. By default, the value is `DESTDIR=$(STAGING_DIR) install`, which is correct for most autotools packages. It is still possible to override it.
- `LIBFOO_INSTALL_TARGET_OPTS` contains the make options used to install the package to the target directory. By default, the value is `DESTDIR=$(TARGET_DIR) install`. The default value is correct for most autotools packages, but it is still possible to override it if needed.

With the autotools infrastructure, all the steps required to build and install the packages are already defined, and they generally work well for most autotools-based packages. However, when required, it is still possible to customize what is done in any particular step:

- By adding a post-operation hook (after extract, patch, configure, build or install). See Section 18.23, "Hooks available in the various build steps" for details.

- By overriding one of the steps. For example, even if the autotools infrastructure is used, if the package `.mk` file defines its own `LIBFOO_CONFIGURE_CMDS` variable, it will be used instead of the default autotools one. However, using this method should be restricted to very specific cases. Do not use it in the general case.

18.8. Infrastructure for CMake-based packages

18.8.1. `cmake-package` tutorial

First, let's see how to write a `.mk` file for a CMake-based package, with an example :

```
01: ################################################################################
02: #
03: # libfoo
04: #
05: ################################################################################
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_INSTALL_STAGING = YES
11: LIBFOO_INSTALL_TARGET = NO
12: LIBFOO_CONF_OPTS = -DBUILD_DEMOS=ON
13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf
14:
15: $(eval $(cmake-package))
```

On line 7, we declare the version of the package.

On line 8 and 9, we declare the name of the tarball (xz-ed tarball recommended) and the location of the tarball on the Web. Buildroot will automatically download the tarball from this location.

On line 10, we tell Buildroot to install the package to the staging directory. The staging directory, located in `output/staging/` is the directory where all the packages are installed, including their development files, etc. By default, packages are not installed to the staging directory, since usually, only libraries need to be installed in the staging directory: their development files are needed to compile other libraries or applications depending on them. Also by default, when staging installation is enabled, packages are installed in this location using the `make install` command.

On line 11, we tell Buildroot to not install the package to the target directory. This directory contains what will become the root filesystem running on the target. For purely static libraries, it is not necessary to install them in the target directory because they will not be used at runtime. By default, target installation is enabled; setting this variable to NO is almost never needed. Also by default, packages are installed in this location using the `make install` command.

On line 12, we tell Buildroot to pass custom options to CMake when it is configuring the package.

On line 13, we declare our dependencies, so that they are built before the build process of our package starts.

Finally, on line line 15, we invoke the `cmake-package` macro that generates all the Makefile rules that actually allows the package to be built.

18.8.2. `cmake-package` reference

The main macro of the CMake package infrastructure is `cmake-package`. It is similar to the `generic-package` macro. The ability to have target and host packages is also available, with the `host-cmake-package` macro.

Just like the generic infrastructure, the CMake infrastructure works by defining a number of variables before calling the `cmake-package` macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the CMake infrastructure: `LIBFOO_VERSION`, `LIBFOO_SOURCE`, `LIBFOO_PATCH`, `LIBFOO_SITE`, `LIBFOO_SUBDIR`, `LIBFOO_DEPENDENCIES`, `LIBFOO_INSTALL_STAGING`, `LIBFOO_INSTALL_TARGET`.

A few additional variables, specific to the CMake infrastructure, can also be defined. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them.

- `LIBFOO_SUBDIR` may contain the name of a subdirectory inside the package that contains the main CMakeLists.txt file. This is useful, if for example, the main CMakeLists.txt file is not at the root of the tree extracted by the tarball. If `HOST_LIBFOO_SUBDIR` is not specified, it defaults to `LIBFOO_SUBDIR`.
- `LIBFOO_CONF_ENV`, to specify additional environment variables to pass to CMake. By default, empty.
- `LIBFOO_CONF_OPTS`, to specify additional configure options to pass to CMake. By default, empty. A number of common CMake options are set by the `cmake-package` infrastructure; so it is normally not necessary to set them in the package's `*.mk` file unless you want to override them:
  - `CMAKE_BUILD_TYPE` is driven by `BR2_ENABLE_RUNTIME_DEBUG`;
  - `CMAKE_INSTALL_PREFIX`;
  - `BUILD_SHARED_LIBS` is driven by `BR2_STATIC_LIBS`;
  - `BUILD_DOC`, `BUILD_DOCS` are disabled;
  - `BUILD_EXAMPLE`, `BUILD_EXAMPLES` are disabled;

- BUILD_TEST, BUILD_TESTS, BUILD_TESTING are disabled.

- LIBFOO_SUPPORTS_IN_SOURCE_BUILD = NO should be set when the package cannot be built inside the source tree but needs a separate build directory.
- LIBFOO_MAKE, to specify an alternate make command. This is typically useful when parallel make is enabled in the configuration (using BR2_JLEVEL) but that this feature should be disabled for the given package, for one reason or another. By default, set to $(MAKE). If parallel building is not supported by the package, then it should be set to LIBFOO_MAKE=$(MAKE1).
- LIBFOO_MAKE_ENV, to specify additional environment variables to pass to make in the build step. These are passed before the make command. By default, empty.
- LIBFOO_MAKE_OPTS, to specify additional variables to pass to make in the build step. These are passed after the make command. By default, empty.
- LIBFOO_INSTALL_OPTS contains the make options used to install the package to the host directory. By default, the value is install, which is correct for most CMake packages. It is still possible to override it.
- LIBFOO_INSTALL_STAGING_OPTS contains the make options used to install the package to the staging directory. By default, the value is DESTDIR=$(STAGING_DIR) install/fast, which is correct for most CMake packages. It is still possible to override it.
- LIBFOO_INSTALL_TARGET_OPTS contains the make options used to install the package to the target directory. By default, the value is DESTDIR=$(TARGET_DIR) install/fast. The default value is correct for most CMake packages, but it is still possible to override it if needed.

With the CMake infrastructure, all the steps required to build and install the packages are already defined, and they generally work well for most CMake-based packages. However, when required, it is still possible to customize what is done in any particular step:

- By adding a post-operation hook (after extract, patch, configure, build or install). See Section 18.23, "Hooks available in the various build steps" for details.
- By overriding one of the steps. For example, even if the CMake infrastructure is used, if the package .mk file defines its own LIBFOO_CONFIGURE_CMDS variable, it will be used instead of the default CMake one. However, using this method should be restricted to very specific cases. Do not use it in the general case.

18.9. Infrastructure for Python packages

This infrastructure applies to Python packages that use the standard Python setuptools or pep517 mechanisms as their build system, generally recognizable by the usage of a setup.py script or pyproject.toml file.

18.9.1. python-package tutorial

First, let's see how to write a .mk file for a Python package, with an example :

```
01: ################################################################################
02: #
03: # python-foo
04: #
05: ################################################################################
06:
07: PYTHON_FOO_VERSION = 1.0
08: PYTHON_FOO_SOURCE = python-foo-$(PYTHON_FOO_VERSION).tar.xz
09: PYTHON_FOO_SITE = http://www.foosoftware.org/download
10: PYTHON_FOO_LICENSE = BSD-3-Clause
11: PYTHON_FOO_LICENSE_FILES = LICENSE
12: PYTHON_FOO_ENV = SOME_VAR=1
13: PYTHON_FOO_DEPENDENCIES = libmad
14: PYTHON_FOO_SETUP_TYPE = distutils
15:
16: $(eval $(python-package))
```

On line 7, we declare the version of the package.

On line 8 and 9, we declare the name of the tarball (xz-ed tarball recommended) and the location of the tarball on the Web. Buildroot will automatically download the tarball from this location.

On line 10 and 11, we give licensing details about the package (its license on line 10, and the file containing the license text on line 11).

On line 12, we tell Buildroot to pass custom options to the Python setup.py script when it is configuring the package.

On line 13, we declare our dependencies, so that they are built before the build process of our package starts.

On line 14, we declare the specific Python build system being used. In this case the distutils Python build system is used. The four supported ones are distutils, flit, pep517 and setuptools.

Finally, on line 16, we invoke the python-package macro that generates all the Makefile rules that actually allow the package to be built.

18.9.2. python-package reference

As a policy, packages that merely provide Python modules should all be named python-<something> in Buildroot. Other

packages that use the Python build system, but are not Python modules, can freely choose their name (existing examples in Buildroot are `scons` and `supervisor`).

The main macro of the Python package infrastructure is `python-package`. It is similar to the `generic-package` macro. It is also possible to create Python host packages with the `host-python-package` macro.

Just like the generic infrastructure, the Python infrastructure works by defining a number of variables before calling the `python-package` or `host-python-package` macros.

All the package metadata information variables that exist in the generic package infrastructure also exist in the Python infrastructure: `PYTHON_FOO_VERSION`, `PYTHON_FOO_SOURCE`, `PYTHON_FOO_PATCH`, `PYTHON_FOO_SITE`, `PYTHON_FOO_SUBDIR`, `PYTHON_FOO_DEPENDENCIES`, `PYTHON_FOO_LICENSE`, `PYTHON_FOO_LICENSE_FILES`, `PYTHON_FOO_INSTALL_STAGING`, etc.

Note that:

- It is not necessary to add `python` or `host-python` in the `PYTHON_FOO_DEPENDENCIES` variable of a package, since these basic dependencies are automatically added as needed by the Python package infrastructure.
- Similarly, it is not needed to add `host-setuptools` to `PYTHON_FOO_DEPENDENCIES` for setuptools-based packages, since it's automatically added by the Python infrastructure as needed.

One variable specific to the Python infrastructure is mandatory:

- `PYTHON_FOO_SETUP_TYPE`, to define which Python build system is used by the package. The four supported values are `distutils`, `flit`, `pep517` and `setuptools`. If you don't know which one is used in your package, look at the `setup.py` or `pyproject.toml` file in your package source code, and see whether it imports things from the `distutils`, `flit` module or the `setuptools` module. If the package is using a `pyproject.toml` file without any build-system requires and with a local in-tree backend-path one should use `pep517`.

A few additional variables, specific to the Python infrastructure, can optionally be defined, depending on the package's needs. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them, or none.

- `PYTHON_FOO_SUBDIR` may contain the name of a subdirectory inside the package that contains the main `setup.py` or `pyproject.toml` file. This is useful, if for example, the main `setup.py` or `pyproject.toml` file is not at the root of the tree extracted by the tarball. If `HOST_PYTHON_FOO_SUBDIR` is not specified, it defaults to `PYTHON_FOO_SUBDIR`.
- `PYTHON_FOO_ENV`, to specify additional environment variables to pass to the Python `setup.py` script (for distutils/setuptools packages) or the `support/scripts/pyinstaller.py` script (for flit/pep517 packages) for both the build and install steps. Note that the infrastructure is automatically passing several standard variables, defined in `PKG_PYTHON_DISTUTILS_ENV` (for distutils target packages), `HOST_PKG_PYTHON_DISTUTILS_ENV` (for distutils host packages), `PKG_PYTHON_SETUPTOOLS_ENV` (for setuptools target packages), `HOST_PKG_PYTHON_SETUPTOOLS_ENV` (for setuptools host packages), `PKG_PYTHON_PEP517_ENV` (for flit/pep517 target packages) and `HOST_PKG_PYTHON_PEP517_ENV` (for flit/pep517 host packages).
- `PYTHON_FOO_BUILD_OPTS`, to specify additional options to pass to the Python `setup.py` script during the build step, this generally only makes sense to use for distutils/setuptools based packages as flit/pep517 based packages do not pass these options to a `setup.py` script but instead pass them to `support/scripts/pyinstaller.py`. For target distutils packages, the `PKG_PYTHON_DISTUTILS_BUILD_OPTS` options are already passed automatically by the infrastructure.
- `PYTHON_FOO_INSTALL_TARGET_OPTS`, `PYTHON_FOO_INSTALL_STAGING_OPTS`, `HOST_PYTHON_FOO_INSTALL_OPTS` to specify additional options to pass to the Python `setup.py` script (for distutils/setuptools packages) or `support/scripts/pyinstaller.py` (for flit/pep517 packages) during the target installation step, the staging installation step or the host installation, respectively. Note that the infrastructure is automatically passing some options, defined in `PKG_PYTHON_DISTUTILS_INSTALL_TARGET_OPTS` or `PKG_PYTHON_DISTUTILS_INSTALL_STAGING_OPTS` (for target distutils packages), `HOST_PKG_PYTHON_DISTUTILS_INSTALL_OPTS` (for host distutils packages), `PKG_PYTHON_SETUPTOOLS_INSTALL_TARGET_OPTS` or `PKG_PYTHON_SETUPTOOLS_INSTALL_STAGING_OPTS` (for target setuptools packages), `HOST_PKG_PYTHON_SETUPTOOLS_INSTALL_OPTS` (for host setuptools packages) and `PKG_PYTHON_PEP517_INSTALL_TARGET_OPTS` or `PKG_PYTHON_PEP517_INSTALL_STAGING_OPTS` (for target flit/pep517 packages).

With the Python infrastructure, all the steps required to build and install the packages are already defined, and they generally work well for most Python-based packages. However, when required, it is still possible to customize what is done in any particular step:

- By adding a post-operation hook (after extract, patch, configure, build or install). See Section 18.23, "Hooks available in the various build steps" for details.
- By overriding one of the steps. For example, even if the Python infrastructure is used, if the package `.mk` file defines its own `PYTHON_FOO_BUILD_CMDS` variable, it will be used instead of the default Python one. However, using this method should be restricted to very specific cases. Do not use it in the general case.

18.9.3. Generating a `python-package` from a PyPI repository

If the Python package for which you would like to create a Buildroot package is available on PyPI, you may want to use the `scanpypi` tool located in `utils/` to automate the process.

You can find the list of existing PyPI packages here.

`scanpypi` requires Python's `setuptools` package to be installed on your host.

When at the root of your buildroot directory just do :

```
utils/scanpypi foo bar -o package
```

This will generate packages `python-foo` and `python-bar` in the package folder if they exist on https://pypi.python.org.

Find the `external python modules` menu and insert your package inside. Keep in mind that the items inside a menu should be in alphabetical order.

Please keep in mind that you'll most likely have to manually check the package for any mistakes as there are things that cannot be guessed by the generator (e.g. dependencies on any of the python core modules such as BR2_PACKAGE_PYTHON_ZLIB). Also, please take note that the license and license files are guessed and must be checked. You also need to manually add the package to the `package/Config.in` file.

If your Buildroot package is not in the official Buildroot tree but in a br2-external tree, use the -o flag as follows:

```
utils/scanpypi foo bar -o other_package_dir
```

This will generate packages `python-foo` and `python-bar` in the `other_package_directory` instead of `package`.

Option `-h` will list the available options:

```
utils/scanpypi -h
```

18.9.4. `python-package` CFFI backend

C Foreign Function Interface for Python (CFFI) provides a convenient and reliable way to call compiled C code from Python using interface declarations written in C. Python packages relying on this backend can be identified by the appearance of a `cffi` dependency in the `install_requires` field of their `setup.py` file.

Such a package should:

- add `python-cffi` as a runtime dependency in order to install the compiled C library wrapper on the target. This is achieved by adding `select BR2_PACKAGE_PYTHON_CFFI` to the package `Config.in`.

```
config BR2_PACKAGE_PYTHON_FOO
        bool "python-foo"
        select BR2_PACKAGE_PYTHON_CFFI # runtime
```

- add `host-python-cffi` as a build-time dependency in order to cross-compile the C wrapper. This is achieved by adding `host-python-cffi` to the `PYTHON_FOO_DEPENDENCIES` variable.

```
##############################################################################
#
# python-foo
#
##############################################################################

...

PYTHON_FOO_DEPENDENCIES = host-python-cffi

$(eval $(python-package))
```

18.10. Infrastructure for LuaRocks-based packages

18.10.1. `luarocks-package` tutorial

First, let's see how to write a `.mk` file for a LuaRocks-based package, with an example :

```
01: ##############################################################################
02: #
03: # lua-foo
04: #
05: ##############################################################################
06:
07: LUA_FOO_VERSION = 1.0.2-1
08: LUA_FOO_NAME_UPSTREAM = foo
09: LUA_FOO_DEPENDENCIES = bar
10:
11: LUA_FOO_BUILD_OPTS += BAR_INCDIR=$(STAGING_DIR)/usr/include
12: LUA_FOO_BUILD_OPTS += BAR_LIBDIR=$(STAGING_DIR)/usr/lib
13: LUA_FOO_LICENSE = luaFoo license
14: LUA_FOO_LICENSE_FILES = $(LUA_FOO_SUBDIR)/COPYING
15:
16: $(eval $(luarocks-package))
```

On line 7, we declare the version of the package (the same as in the rockspec, which is the concatenation of the upstream version and the rockspec revision, separated by a hyphen -).

On line 8, we declare that the package is called "foo" on LuaRocks. In Buildroot, we give Lua-related packages a name that

starts with "lua", so the Buildroot name is different from the upstream name. `LUA_FOO_NAME_UPSTREAM` makes the link between the two names.

On line 9, we declare our dependencies against native libraries, so that they are built before the build process of our package starts.

On lines 11-12, we tell Buildroot to pass custom options to LuaRocks when it is building the package.

On lines 13-14, we specify the licensing terms for the package.

Finally, on line 16, we invoke the `luarocks-package` macro that generates all the Makefile rules that actually allows the package to be built.

Most of these details can be retrieved from the `rock` and `rockspec`. So, this file and the Config.in file can be generated by running the command `luarocks buildroot foo lua-foo` in the Buildroot directory. This command runs a specific Buildroot addon of `luarocks` that will automatically generate a Buildroot package. The result must still be manually inspected and possibly modified.

- The `package/Config.in` file has to be updated manually to include the generated Config.in files.

18.10.2. `luarocks-package` reference

LuaRocks is a deployment and management system for Lua modules, and supports various `build.type`: `builtin`, `make` and `cmake`. In the context of Buildroot, the `luarocks-package` infrastructure only supports the `builtin` mode. LuaRocks packages that use the `make` or `cmake` build mechanisms should instead be packaged using the `generic-package` and `cmake-package` infrastructures in Buildroot, respectively.

The main macro of the LuaRocks package infrastructure is `luarocks-package`: like `generic-package` it works by defining a number of variables providing metadata information about the package, and then calling `luarocks-package`.

Just like the generic infrastructure, the LuaRocks infrastructure works by defining a number of variables before calling the `luarocks-package` macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the LuaRocks infrastructure: `LUA_FOO_VERSION`, `LUA_FOO_SOURCE`, `LUA_FOO_SITE`, `LUA_FOO_DEPENDENCIES`, `LUA_FOO_LICENSE`, `LUA_FOO_LICENSE_FILES`.

Two of them are populated by the LuaRocks infrastructure (for the `download` step). If your package is not hosted on the LuaRocks mirror `$(BR2_LUAROCKS_MIRROR)`, you can override them:

- `LUA_FOO_SITE`, which defaults to `$(BR2_LUAROCKS_MIRROR)`
- `LUA_FOO_SOURCE`, which defaults to `$(lowercase LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION).src.rock`

A few additional variables, specific to the LuaRocks infrastructure, are also defined. They can be overridden in specific cases.

- `LUA_FOO_NAME_UPSTREAM`, which defaults to `lua-foo`, i.e. the Buildroot package name
- `LUA_FOO_ROCKSPEC`, which defaults to `$(lowercase LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION).rockspec`
- `LUA_FOO_SUBDIR`, which defaults to `$(LUA_FOO_NAME_UPSTREAM)-$(LUA_FOO_VERSION_WITHOUT_ROCKSPEC_REVISION)`
- `LUA_FOO_BUILD_OPTS` contains additional build options for the `luarocks build` call.

18.11. Infrastructure for Perl/CPAN packages

18.11.1. `perl-package` tutorial

First, let’s see how to write a `.mk` file for a Perl/CPAN package, with an example :

```
01: ################################################################################
02: #
03: # perl-foo-bar
04: #
05: ################################################################################
06:
07: PERL_FOO_BAR_VERSION = 0.02
08: PERL_FOO_BAR_SOURCE = Foo-Bar-$(PERL_FOO_BAR_VERSION).tar.gz
09: PERL_FOO_BAR_SITE = $(BR2_CPAN_MIRROR)/authors/id/M/MO/MONGER
10: PERL_FOO_BAR_DEPENDENCIES = perl-strictures
11: PERL_FOO_BAR_LICENSE = Artistic or GPL-1.0+
12: PERL_FOO_BAR_LICENSE_FILES = LICENSE
13: PERL_FOO_BAR_DISTNAME = Foo-Bar
14:
15: $(eval $(perl-package))
```

On line 7, we declare the version of the package.

On line 8 and 9, we declare the name of the tarball and the location of the tarball on a CPAN server. Buildroot will automatically download the tarball from this location.

On line 10, we declare our dependencies, so that they are built before the build process of our package starts.

On line 11 and 12, we give licensing details about the package (its license on line 11, and the file containing the license text on line 12).

On line 13, the name of the distribution as needed by the script `utils/scancpan` (in order to regenerate/upgrade these package files).

Finally, on line 15, we invoke the `perl-package` macro that generates all the Makefile rules that actually allow the package to be built.

Most of these data can be retrieved from https://metacpan.org/. So, this file and the Config.in can be generated by running the script `utils/scancpan Foo-Bar` in the Buildroot directory (or in a br2-external tree). This script creates a Config.in file and foo-bar.mk file for the requested package, and also recursively for all dependencies specified by CPAN. You should still manually edit the result. In particular, the following things should be checked.

- If the perl module links with a shared library that is provided by another (non-perl) package, this dependency is not added automatically. It has to be added manually to `PERL_FOO_BAR_DEPENDENCIES`.
- The `package/Config.in` file has to be updated manually to include the generated Config.in files. As a hint, the `scancpan` script prints out the required `source "…"` statements, sorted alphabetically.

18.11.2. `perl-package` reference

As a policy, packages that provide Perl/CPAN modules should all be named `perl-<something>` in Buildroot.

This infrastructure handles various Perl build systems : `ExtUtils-MakeMaker` (EUMM), `Module-Build` (MB) and `Module-Build-Tiny`. `Build.PL` is preferred by default when a package provides a `Makefile.PL` and a `Build.PL`.

The main macro of the Perl/CPAN package infrastructure is `perl-package`. It is similar to the `generic-package` macro. The ability to have target and host packages is also available, with the `host-perl-package` macro.

Just like the generic infrastructure, the Perl/CPAN infrastructure works by defining a number of variables before calling the `perl-package` macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the Perl/CPAN infrastructure: `PERL_FOO_VERSION`, `PERL_FOO_SOURCE`, `PERL_FOO_PATCH`, `PERL_FOO_SITE`, `PERL_FOO_SUBDIR`, `PERL_FOO_DEPENDENCIES`, `PERL_FOO_INSTALL_TARGET`.

Note that setting `PERL_FOO_INSTALL_STAGING` to `YES` has no effect unless a `PERL_FOO_INSTALL_STAGING_CMDS` variable is defined. The perl infrastructure doesn’t define these commands since Perl modules generally don’t need to be installed to the `staging` directory.

A few additional variables, specific to the Perl/CPAN infrastructure, can also be defined. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them.

- `PERL_FOO_PREFER_INSTALLER`/`HOST_PERL_FOO_PREFER_INSTALLER`, specifies the preferred installation method. Possible values are `EUMM` (for `Makefile.PL` based installation using `ExtUtils-MakeMaker`) and `MB` (for `Build.PL` based installation using `Module-Build`). This variable is only used when the package provides both installation methods.
- `PERL_FOO_CONF_ENV`/`HOST_PERL_FOO_CONF_ENV`, to specify additional environment variables to pass to the `perl Makefile.PL` or `perl Build.PL`. By default, empty.
- `PERL_FOO_CONF_OPTS`/`HOST_PERL_FOO_CONF_OPTS`, to specify additional configure options to pass to the `perl Makefile.PL` or `perl Build.PL`. By default, empty.
- `PERL_FOO_BUILD_OPTS`/`HOST_PERL_FOO_BUILD_OPTS`, to specify additional options to pass to `make pure_all` or `perl Build build` in the build step. By default, empty.
- `PERL_FOO_INSTALL_TARGET_OPTS`, to specify additional options to pass to `make pure_install` or `perl Build install` in the install step. By default, empty.
- `HOST_PERL_FOO_INSTALL_OPTS`, to specify additional options to pass to `make pure_install` or `perl Build install` in the install step. By default, empty.

18.12. Infrastructure for virtual packages

In Buildroot, a virtual package is a package whose functionalities are provided by one or more packages, referred to as *providers*. The virtual package management is an extensible mechanism allowing the user to choose the provider used in the rootfs.

For example, *OpenGL ES* is an API for 2D and 3D graphics on embedded systems. The implementation of this API is different for the *Allwinner Tech Sunxi* and the *Texas Instruments OMAP35xx* platforms. So `libgles` will be a virtual package and `sunxi-mali-utgard` and `ti-gfx` will be the providers.

18.12.1. `virtual-package` tutorial

In the following example, we will explain how to add a new virtual package (*something-virtual*) and a provider for it (*some-provider*).

First, let’s create the virtual package.

18.12.2. Virtual package's `Config.in` file

The `Config.in` file of virtual package *something-virtual* should contain:

```
01: config BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
02:     bool
03:
04: config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
05:     depends on BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
06:     string
```

In this file, we declare two options, `BR2_PACKAGE_HAS_SOMETHING_VIRTUAL` and `BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL`, whose values will be used by the providers.

18.12.3. Virtual package's `.mk` file

The `.mk` for the virtual package should just evaluate the `virtual-package` macro:

```
01: ################################################################################
02: #
03: # something-virtual
04: #
05: ################################################################################
06:
07: $(eval $(virtual-package))
```

The ability to have target and host packages is also available, with the `host-virtual-package` macro.

18.12.4. Provider's `Config.in` file

When adding a package as a provider, only the `Config.in` file requires some modifications.

The `Config.in` file of the package *some-provider*, which provides the functionalities of *something-virtual*, should contain:

```
01: config BR2_PACKAGE_SOME_PROVIDER
02:     bool "some-provider"
03:     select BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
04:     help
05:       This is a comment that explains what some-provider is.
06:
07:       http://foosoftware.org/some-provider/
08:
09: if BR2_PACKAGE_SOME_PROVIDER
10: config BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL
11:     default "some-provider"
12: endif
```

On line 3, we select `BR2_PACKAGE_HAS_SOMETHING_VIRTUAL`, and on line 11, we set the value of `BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL` to the name of the provider, but only if it is selected.

18.12.5. Provider's `.mk` file

The `.mk` file should also declare an additional variable `SOME_PROVIDER_PROVIDES` to contain the names of all the virtual packages it is an implementation of:

```
01: SOME_PROVIDER_PROVIDES = something-virtual
```

Of course, do not forget to add the proper build and runtime dependencies for this package!

18.12.6. Notes on depending on a virtual package

When adding a package that requires a certain `FEATURE` provided by a virtual package, you have to use `depends on BR2_PACKAGE_HAS_FEATURE`, like so:

```
config BR2_PACKAGE_HAS_FEATURE
    bool

config BR2_PACKAGE_FOO
    bool "foo"
    depends on BR2_PACKAGE_HAS_FEATURE
```

18.12.7. Notes on depending on a specific provider

If your package really requires a specific provider, then you'll have to make your package `depends on` this provider; you can *not* `select` a provider.

Let's take an example with two providers for a `FEATURE`:

```
config BR2_PACKAGE_HAS_FEATURE
    bool
```

```
config BR2_PACKAGE_FOO
    bool "foo"
    select BR2_PACKAGE_HAS_FEATURE

config BR2_PACKAGE_BAR
    bool "bar"
    select BR2_PACKAGE_HAS_FEATURE
```

And you are adding a package that needs `FEATURE` as provided by `foo`, but not as provided by `bar`.

If you were to use `select BR2_PACKAGE_FOO`, then the user would still be able to select `BR2_PACKAGE_BAR` in the menuconfig. This would create a configuration inconsistency, whereby two providers of the same `FEATURE` would be enabled at once, one explicitly set by the user, the other implicitly by your `select`.

Instead, you have to use `depends on BR2_PACKAGE_FOO`, which avoids any implicit configuration inconsistency.

18.13. Infrastructure for packages using kconfig for configuration files

A popular way for a software package to handle user-specified configuration is `kconfig`. Among others, it is used by the Linux kernel, Busybox, and Buildroot itself. The presence of a .config file and a `menuconfig` target are two well-known symptoms of kconfig being used.

Buildroot features an infrastructure for packages that use kconfig for their configuration. This infrastructure provides the necessary logic to expose the package's `menuconfig` target as `foo-menuconfig` in Buildroot, and to handle the copying back and forth of the configuration file in a correct way.

The `kconfig-package` infrastructure is based on the `generic-package` infrastructure. All variables supported by `generic-package` are available in `kconfig-package` as well. See Section 18.6.2, "`generic-package` reference" for more details.

In order to use the `kconfig-package` infrastructure for a Buildroot package, the minimally required lines in the `.mk` file, in addition to the variables required by the `generic-package` infrastructure, are:

```
FOO_KCONFIG_FILE = reference-to-source-configuration-file

$(eval $(kconfig-package))
```

This snippet creates the following make targets:

- `foo-menuconfig`, which calls the package's `menuconfig` target
- `foo-update-config`, which copies the configuration back to the source configuration file. It is not possible to use this target when fragment files are set.
- `foo-update-defconfig`, which copies the configuration back to the source configuration file. The configuration file will only list the options that differ from the default values. It is not possible to use this target when fragment files are set.
- `foo-diff-config`, which outputs the differences between the current configuration and the one defined in the Buildroot configuration for this kconfig package. The output is useful to identify the configuration changes that may have to be propagated to configuration fragments for example.

and ensures that the source configuration file is copied to the build directory at the right moment.

There are two options to specify a configuration file to use, either `FOO_KCONFIG_FILE` (as in the example, above) or `FOO_KCONFIG_DEFCONFIG`. It is mandatory to provide either, but not both:

- `FOO_KCONFIG_FILE` specifies the path to a defconfig or full-config file to be used to configure the package.
- `FOO_KCONFIG_DEFCONFIG` specifies the defconfig *make* rule to call to configure the package.

In addition to these minimally required lines, several optional variables can be set to suit the needs of the package under consideration:

- `FOO_KCONFIG_EDITORS`: a space-separated list of kconfig editors to support, for example *menuconfig xconfig*. By default, *menuconfig*.
- `FOO_KCONFIG_FRAGMENT_FILES`: a space-separated list of configuration fragment files that are merged to the main configuration file. Fragment files are typically used when there is a desire to stay in sync with an upstream (def)config file, with some minor modifications.
- `FOO_KCONFIG_OPTS`: extra options to pass when calling the kconfig editors. This may need to include *$(FOO_MAKE_OPTS)*, for example. By default, empty.
- `FOO_KCONFIG_FIXUP_CMDS`: a list of shell commands needed to fixup the configuration file after copying it or running a kconfig editor. Such commands may be needed to ensure a configuration consistent with other configuration of Buildroot, for example. By default, empty.
- `FOO_KCONFIG_DOTCONFIG`: path (with filename) of the `.config` file, relative to the package source tree. The default, `.config`, should be well suited for all packages that use the standard kconfig infrastructure as inherited from the Linux kernel; some packages use a derivative of kconfig that use a different location.
- `FOO_KCONFIG_DEPENDENCIES`: the list of packages (most probably, host packages) that need to be built before this package's kconfig is interpreted. Seldom used. By default, empty.

- FOO_KCONFIG_SUPPORTS_DEFCONFIG: whether the package's kconfig system supports using defconfig files; few packages do not. By default, *YES*.

18.14. Infrastructure for rebar-based packages

18.14.1. rebar-package tutorial

First, let's see how to write a .mk file for a rebar-based package, with an example :

```
01: ################################################################################
02: #
03: # erlang-foobar
04: #
05: ################################################################################
06:
07: ERLANG_FOOBAR_VERSION = 1.0
08: ERLANG_FOOBAR_SOURCE = erlang-foobar-$(ERLANG_FOOBAR_VERSION).tar.xz
09: ERLANG_FOOBAR_SITE = http://www.foosoftware.org/download
10: ERLANG_FOOBAR_DEPENDENCIES = host-libaaa libbbb
11:
12: $(eval $(rebar-package))
```

On line 7, we declare the version of the package.

On line 8 and 9, we declare the name of the tarball (xz-ed tarball recommended) and the location of the tarball on the Web. Buildroot will automatically download the tarball from this location.

On line 10, we declare our dependencies, so that they are built before the build process of our package starts.

Finally, on line 12, we invoke the rebar-package macro that generates all the Makefile rules that actually allows the package to be built.

18.14.2. rebar-package reference

The main macro of the rebar package infrastructure is rebar-package. It is similar to the generic-package macro. The ability to have host packages is also available, with the host-rebar-package macro.

Just like the generic infrastructure, the rebar infrastructure works by defining a number of variables before calling the rebar-package macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the rebar infrastructure: ERLANG_FOOBAR_VERSION, ERLANG_FOOBAR_SOURCE, ERLANG_FOOBAR_PATCH, ERLANG_FOOBAR_SITE, ERLANG_FOOBAR_SUBDIR, ERLANG_FOOBAR_DEPENDENCIES, ERLANG_FOOBAR_INSTALL_STAGING, ERLANG_FOOBAR_INSTALL_TARGET, ERLANG_FOOBAR_LICENSE and ERLANG_FOOBAR_LICENSE_FILES.

A few additional variables, specific to the rebar infrastructure, can also be defined. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them.

- ERLANG_FOOBAR_USE_AUTOCONF, to specify that the package uses *autoconf* at the configuration step. When a package sets this variable to YES, the autotools infrastructure is used.

  Note. You can also use some of the variables from the autotools infrastructure: ERLANG_FOOBAR_CONF_ENV, ERLANG_FOOBAR_CONF_OPTS, ERLANG_FOOBAR_AUTORECONF, ERLANG_FOOBAR_AUTORECONF_ENV and ERLANG_FOOBAR_AUTORECONF_OPTS.

- ERLANG_FOOBAR_USE_BUNDLED_REBAR, to specify that the package has a bundled version of *rebar* **and** that it shall be used. Valid values are YES or NO (the default).

  Note. If the package bundles a *rebar* utility, but can use the generic one that Buildroot provides, just say NO (i.e., do not specify this variable). Only set if it is mandatory to use the *rebar* utility bundled in this package.

- ERLANG_FOOBAR_REBAR_ENV, to specify additional environment variables to pass to the *rebar* utility.
- ERLANG_FOOBAR_KEEP_DEPENDENCIES, to keep the dependencies described in the rebar.config file. Valid values are YES or NO (the default). Unless this variable is set to YES, the *rebar* infrastructure removes such dependencies in a post-patch hook to ensure rebar does not download nor compile them.

With the rebar infrastructure, all the steps required to build and install the packages are already defined, and they generally work well for most rebar-based packages. However, when required, it is still possible to customize what is done in any particular step:

- By adding a post-operation hook (after extract, patch, configure, build or install). See Section 18.23, "Hooks available in the various build steps" for details.
- By overriding one of the steps. For example, even if the rebar infrastructure is used, if the package .mk file defines its own ERLANG_FOOBAR_BUILD_CMDS variable, it will be used instead of the default rebar one. However, using this method should be restricted to very specific cases. Do not use it in the general case.

18.15. Infrastructure for Waf-based packages

18.15.1. waf-package tutorial

First, let’s see how to write a `.mk` file for a Waf-based package, with an example :

```
01: ################################################################################
02: #
03: # libfoo
04: #
05: ################################################################################
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_CONF_OPTS = --enable-bar --disable-baz
11: LIBFOO_DEPENDENCIES = bar
12:
13: $(eval $(waf-package))
```

On line 7, we declare the version of the package.

On line 8 and 9, we declare the name of the tarball (xz-ed tarball recommended) and the location of the tarball on the Web. Buildroot will automatically download the tarball from this location.

On line 10, we tell Buildroot what options to enable for libfoo.

On line 11, we tell Buildroot the dependencies of libfoo.

Finally, on line line 13, we invoke the `waf-package` macro that generates all the Makefile rules that actually allows the package to be built.

18.15.2. `waf-package` reference

The main macro of the Waf package infrastructure is `waf-package`. It is similar to the `generic-package` macro.

Just like the generic infrastructure, the Waf infrastructure works by defining a number of variables before calling the `waf-package` macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the Waf infrastructure: `LIBFOO_VERSION`, `LIBFOO_SOURCE`, `LIBFOO_PATCH`, `LIBFOO_SITE`, `LIBFOO_SUBDIR`, `LIBFOO_DEPENDENCIES`, `LIBFOO_INSTALL_STAGING`, `LIBFOO_INSTALL_TARGET`.

An additional variable, specific to the Waf infrastructure, can also be defined.

- `LIBFOO_SUBDIR` may contain the name of a subdirectory inside the package that contains the main wscript file. This is useful, if for example, the main wscript file is not at the root of the tree extracted by the tarball. If `HOST_LIBFOO_SUBDIR` is not specified, it defaults to `LIBFOO_SUBDIR`.
- `LIBFOO_NEEDS_EXTERNAL_WAF` can be set to `YES` or `NO` to tell Buildroot to use the bundled `waf` executable. If set to `NO`, the default, then Buildroot will use the waf executable provided in the package source tree; if set to `YES`, then Buildroot will download, install waf as a host tool and use it to build the package.
- `LIBFOO_WAF_OPTS`, to specify additional options to pass to the `waf` script at every step of the package build process: configure, build and installation. By default, empty.
- `LIBFOO_CONF_OPTS`, to specify additional options to pass to the `waf` script for the configuration step. By default, empty.
- `LIBFOO_BUILD_OPTS`, to specify additional options to pass to the `waf` script during the build step. By default, empty.
- `LIBFOO_INSTALL_STAGING_OPTS`, to specify additional options to pass to the `waf` script during the staging installation step. By default, empty.
- `LIBFOO_INSTALL_TARGET_OPTS`, to specify additional options to pass to the `waf` script during the target installation step. By default, empty.

18.16. Infrastructure for Meson-based packages

18.16.1. `meson-package` tutorial

Meson is an open source build system meant to be both extremely fast, and, even more importantly, as user friendly as possible. It uses Ninja as a companion tool to perform the actual build operations.

Let’s see how to write a `.mk` file for a Meson-based package, with an example:

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: FOO_VERSION = 1.0
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.gz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-3.0+
11: FOO_LICENSE_FILES = COPYING
12: FOO_INSTALL_STAGING = YES
13:
14: FOO_DEPENDENCIES = host-pkgconf bar
15:
```

```
16: ifeq ($(BR2_PACKAGE_BAZ),y)
17: FOO_CONF_OPTS += -Dbaz=true
18: FOO_DEPENDENCIES += baz
19: else
20: FOO_CONF_OPTS += -Dbaz=false
21: endif
22:
23: $(eval $(meson-package))
```

The Makefile starts with the definition of the standard variables for package declaration (lines 7 to 11).

On line line 23, we invoke the `meson-package` macro that generates all the Makefile rules that actually allows the package to be built.

In the example, `host-pkgconf` and `bar` are declared as dependencies in `FOO_DEPENDENCIES` at line 14 because the Meson build file of `foo` uses `pkg-config` to determine the compilation flags and libraries of package `bar`.

Note that it is not necessary to add `host-meson` in the `FOO_DEPENDENCIES` variable of a package, since this basic dependency is automatically added as needed by the Meson package infrastructure.

If the "baz" package is selected, then support for the "baz" feature in "foo" is activated by adding `-Dbaz=true` to `FOO_CONF_OPTS` at line 17, as specified in the `meson_options.txt` file in "foo" source tree. The "baz" package is also added to `FOO_DEPENDENCIES`. Note that the support for `baz` is explicitly disabled at line 20, if the package is not selected.

To sum it up, to add a new meson-based package, the Makefile example can be copied verbatim then edited to replace all occurences of `FOO` with the uppercase name of the new package and update the values of the standard variables.

18.16.2. `meson-package` reference

The main macro of the Meson package infrastructure is `meson-package`. It is similar to the `generic-package` macro. The ability to have target and host packages is also available, with the `host-meson-package` macro.

Just like the generic infrastructure, the Meson infrastructure works by defining a number of variables before calling the `meson-package` macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the Meson infrastructure: `FOO_VERSION`, `FOO_SOURCE`, `FOO_PATCH`, `FOO_SITE`, `FOO_SUBDIR`, `FOO_DEPENDENCIES`, `FOO_INSTALL_STAGING`, `FOO_INSTALL_TARGET`.

A few additional variables, specific to the Meson infrastructure, can also be defined. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them.

- `FOO_SUBDIR` may contain the name of a subdirectory inside the package that contains the main meson.build file. This is useful, if for example, the main meson.build file is not at the root of the tree extracted by the tarball. If `HOST_FOO_SUBDIR` is not specified, it defaults to `FOO_SUBDIR`.
- `FOO_CONF_ENV`, to specify additional environment variables to pass to `meson` for the configuration step. By default, empty.
- `FOO_CONF_OPTS`, to specify additional options to pass to `meson` for the configuration step. By default, empty.
- `FOO_CFLAGS`, to specify compiler arguments added to the package specific `cross-compile.conf` file `c_args` property. By default, the value of `TARGET_CFLAGS`.
- `FOO_CXXFLAGS`, to specify compiler arguments added to the package specific `cross-compile.conf` file `cpp_args` property. By default, the value of `TARGET_CXXFLAGS`.
- `FOO_LDFLAGS`, to specify compiler arguments added to the package specific `cross-compile.conf` file `c_link_args` and `cpp_link_args` properties. By default, the value of `TARGET_LDFLAGS`.
- `FOO_MESON_EXTRA_BINARIES`, to specify a space-separated list of programs to add to the `[binaries]` section of the meson `cross-compilation.conf` configuration file. The format is `program-name='/path/to/program'`, with no space around the `=` sign, and with the path of the program between single quotes. By default, empty. Note that Buildroot already sets the correct values for `c`, `cpp`, `ar`, `strip`, and `pkgconfig`.
- `FOO_MESON_EXTRA_PROPERTIES`, to specify a space-separated list of properties to add to the `[properties]` section of the meson `cross-compilation.conf` configuration file. The format is `property-name=<value>` with no space around the `=` sign, and with single quotes around string values. By default, empty. Note that Buildroot already sets values for `needs_exe_wrapper`, `c_args`, `c_link_args`, `cpp_args`, `cpp_link_args`, `sys_root`, and `pkg_config_libdir`.
- `FOO_NINJA_ENV`, to specify additional environment variables to pass to `ninja`, meson companion tool in charge of the build operations. By default, empty.
- `FOO_NINJA_OPTS`, to specify a space-separated list of targets to build. By default, empty, to build the default target(s).

18.17. Infrastructure for Cargo-based packages

Cargo is the package manager for the Rust programming language. It allows the user to build programs or libraries written in Rust, but it also downloads and manages their dependencies, to ensure repeatable builds. Cargo packages are called "crates".

18.17.1. `cargo-package` tutorial

The `Config.in` file of Cargo-based package *foo* should contain:

```
config BR2_PACKAGE_FOO
    bool "foo"
    depends on BR2_PACKAGE_HOST_RUSTC_TARGET_ARCH_SUPPORTS
    select BR2_PACKAGE_HOST_RUSTC
    help
      This is a comment that explains what foo is.

      http://foosoftware.org/foo/
```

And the `.mk` file for this package should contain:

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: FOO_VERSION = 1.0
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.gz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-3.0+
11: FOO_LICENSE_FILES = COPYING
12:
13: $(eval $(cargo-package))
```

The Makefile starts with the definition of the standard variables for package declaration (lines 7 to 11).

As seen in line 13, it is based on the `cargo-package` infrastructure. Cargo will be invoked automatically by this infrastructure to build and install the package.

It is still possible to define custom build commands or install commands (i.e. with FOO_BUILD_CMDS and FOO_INSTALL_TARGET_CMDS). Those will then replace the commands from the cargo infrastructure.

18.17.2. `cargo-package` reference

The main macros for the Cargo package infrastructure are `cargo-package` for target packages and `host-cargo-package` for host packages.

Just like the generic infrastructure, the Cargo infrastructure works by defining a number of variables before calling the `cargo-package` or `host-cargo-package` macros.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the Cargo infrastructure: `FOO_VERSION`, `FOO_SOURCE`, `FOO_PATCH`, `FOO_SITE`, `FOO_DEPENDENCIES`, `FOO_LICENSE`, `FOO_LICENSE_FILES`, etc.

A few additional variables, specific to the Cargo infrastructure, can also be defined. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them.

- `FOO_SUBDIR` may contain the name of a subdirectory inside the package that contains the Cargo.toml file. This is useful, if for example, it is not at the root of the tree extracted by the tarball. If `HOST_FOO_SUBDIR` is not specified, it defaults to `FOO_SUBDIR`.
- `FOO_CARGO_ENV` can be used to pass additional variables in the environment of `cargo` invocations. It used at both build and installation time
- `FOO_CARGO_BUILD_OPTS` can be used to pass additional options to `cargo` at build time.
- `FOO_CARGO_INSTALL_OPTS` can be used to pass additional options to `cargo` at install time.

A crate can depend on other libraries from crates.io or git repositories, listed in its `Cargo.toml` file. Buildroot automatically takes care of downloading such dependencies as part of the download step of packages that use the `cargo-package` infrastructure. Such dependencies are then kept together with the package source code in the tarball cached in Buildroot’s `DL_DIR`, and therefore the hash of the package’s tarball includes such dependencies.

This mechanism ensures that any change in the dependencies will be detected, and allows the build to be performed completely offline.

18.18. Infrastructure for Go packages

This infrastructure applies to Go packages that use the standard build system and use bundled dependencies.

18.18.1. `golang-package` tutorial

First, let’s see how to write a `.mk` file for a go package, with an example :

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: FOO_VERSION = 1.0
08: FOO_SITE = $(call github,bar,foo,$(FOO_VERSION))
09: FOO_LICENSE = BSD-3-Clause
```

```
10: FOO_LICENSE_FILES = LICENSE
11:
12: $(eval $(golang-package))
```

On line 7, we declare the version of the package.

On line 8, we declare the upstream location of the package, here fetched from Github, since a large number of Go packages are hosted on Github.

On line 9 and 10, we give licensing details about the package.

Finally, on line 12, we invoke the `golang-package` macro that generates all the Makefile rules that actually allow the package to be built.

18.18.2. `golang-package` reference

In their `Config.in` file, packages using the `golang-package` infrastructure should depend on `BR2_PACKAGE_HOST_GO_TARGET_ARCH_SUPPORTS` because Buildroot will automatically add a dependency on `host-go` to such packages. If you need CGO support in your package, you must add a dependency on `BR2_PACKAGE_HOST_GO_TARGET_CGO_LINKING_SUPPORTS`.

The main macro of the Go package infrastructure is `golang-package`. It is similar to the `generic-package` macro. The ability to build host packages is also available, with the `host-golang-package` macro. Host packages built by `host-golang-package` macro should depend on BR2_PACKAGE_HOST_GO_HOST_ARCH_SUPPORTS.

Just like the generic infrastructure, the Go infrastructure works by defining a number of variables before calling the `golang-package`.

All the package metadata information variables that exist in the generic package infrastructure also exist in the Go infrastructure: `FOO_VERSION`, `FOO_SOURCE`, `FOO_PATCH`, `FOO_SITE`, `FOO_SUBDIR`, `FOO_DEPENDENCIES`, `FOO_LICENSE`, `FOO_LICENSE_FILES`, `FOO_INSTALL_STAGING`, etc.

Note that it is not necessary to add `host-go` in the `FOO_DEPENDENCIES` variable of a package, since this basic dependency is automatically added as needed by the Go package infrastructure.

A few additional variables, specific to the Go infrastructure, can optionally be defined, depending on the package's needs. Many of them are only useful in very specific cases, typical packages will therefore only use a few of them, or none.

- The package must specify its Go module name in the `FOO_GOMOD` variable. If not specified, it defaults to `URL-domain/1st-part-of-URL/2nd-part-of-URL`, e.g `FOO_GOMOD` will take the value `github.com/bar/foo` for a package that specifies `FOO_SITE = $(call github,bar,foo,$(FOO_VERSION))`. The Go package infrastructure will automatically generate a minimal `go.mod` file in the package source tree if it doesn't exist.
- `FOO_LDFLAGS` and `FOO_TAGS` can be used to pass respectively the `LDFLAGS` or the `TAGS` to the `go` build command.
- `FOO_BUILD_TARGETS` can be used to pass the list of targets that should be built. If `FOO_BUILD_TARGETS` is not specified, it defaults to `.`. We then have two cases:
  - `FOO_BUILD_TARGETS` is `.`. In this case, we assume only one binary will be produced, and that by default we name it after the package name. If that is not appropriate, the name of the produced binary can be overridden using `FOO_BIN_NAME`.
  - `FOO_BUILD_TARGETS` is not `.`. In this case, we iterate over the values to build each target, and for each produced a binary that is the non-directory component of the target. For example if `FOO_BUILD_TARGETS = cmd/docker cmd/dockerd` the binaries produced are `docker` and `dockerd`.
- `FOO_INSTALL_BINS` can be used to pass the list of binaries that should be installed in `/usr/bin` on the target. If `FOO_INSTALL_BINS` is not specified, it defaults to the lower-case name of package.

With the Go infrastructure, all the steps required to build and install the packages are already defined, and they generally work well for most Go-based packages. However, when required, it is still possible to customize what is done in any particular step:

- By adding a post-operation hook (after extract, patch, configure, build or install). See Section 18.23, "Hooks available in the various build steps" for details.
- By overriding one of the steps. For example, even if the Go infrastructure is used, if the package `.mk` file defines its own `FOO_BUILD_CMDS` variable, it will be used instead of the default Go one. However, using this method should be restricted to very specific cases. Do not use it in the general case.

A Go package can depend on other Go modules, listed in its `go.mod` file. Buildroot automatically takes care of downloading such dependencies as part of the download step of packages that use the `golang-package` infrastructure. Such dependencies are then kept together with the package source code in the tarball cached in Buildroot's `DL_DIR`, and therefore the hash of the package's tarball includes such dependencies.

This mechanism ensures that any change in the dependencies will be detected, and allows the build to be performed completely offline.

18.19. Infrastructure for QMake-based packages

18.19.1. `qmake-package` tutorial

First, let's see how to write a `.mk` file for a QMake-based package, with an example :

```
01: ################################################################################
02: #
03: # libfoo
04: #
05: ################################################################################
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_CONF_OPTS = QT_CONFIG+=bar QT_CONFIG-=baz
11: LIBFOO_DEPENDENCIES = bar
12:
13: $(eval $(qmake-package))
```

On line 7, we declare the version of the package.

On line 8 and 9, we declare the name of the tarball (xz-ed tarball recommended) and the location of the tarball on the Web. Buildroot will automatically download the tarball from this location.

On line 10, we tell Buildroot what options to enable for libfoo.

On line 11, we tell Buildroot the dependencies of libfoo.

Finally, on line line 13, we invoke the `qmake-package` macro that generates all the Makefile rules that actually allows the package to be built.

18.19.2. `qmake-package` reference

The main macro of the QMake package infrastructure is `qmake-package`. It is similar to the `generic-package` macro.

Just like the generic infrastructure, the QMake infrastructure works by defining a number of variables before calling the `qmake-package` macro.

First, all the package metadata information variables that exist in the generic infrastructure also exist in the QMake infrastructure: `LIBFOO_VERSION`, `LIBFOO_SOURCE`, `LIBFOO_PATCH`, `LIBFOO_SITE`, `LIBFOO_SUBDIR`, `LIBFOO_DEPENDENCIES`, `LIBFOO_INSTALL_STAGING`, `LIBFOO_INSTALL_TARGET`.

An additional variable, specific to the QMake infrastructure, can also be defined.

- `LIBFOO_CONF_ENV`, to specify additional environment variables to pass to the `qmake` script for the configuration step. By default, empty.
- `LIBFOO_CONF_OPTS`, to specify additional options to pass to the `qmake` script for the configuration step. By default, empty.
- `LIBFOO_MAKE_ENV`, to specify additional environment variables to the `make` command during the build and install steps. By default, empty.
- `LIBFOO_MAKE_OPTS`, to specify additional targets to pass to the `make` command during the build step. By default, empty.
- `LIBFOO_INSTALL_STAGING_OPTS`, to specify additional targets to pass to the `make` command during the staging installation step. By default, `install`.
- `LIBFOO_INSTALL_TARGET_OPTS`, to specify additional targets to pass to the `make` command during the target installation step. By default, `install`.
- `LIBFOO_SYNC_QT_HEADERS`, to run syncqt.pl before qmake. Some packages need this to have a properly populated include directory before running the build.

18.20. Infrastructure for packages building kernel modules

Buildroot offers a helper infrastructure to make it easy to write packages that build and install Linux kernel modules. Some packages only contain a kernel module, other packages contain programs and libraries in addition to kernel modules. Buildroot's helper infrastructure supports either case.

18.20.1. `kernel-module` tutorial

Let's start with an example on how to prepare a simple package that only builds a kernel module, and no other component:

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: FOO_VERSION = 1.2.3
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-2.0
11: FOO_LICENSE_FILES = COPYING
12:
13: $(eval $(kernel-module))
14: $(eval $(generic-package))
```

Lines 7-11 define the usual meta-data to specify the version, archive name, remote URI where to find the package source, licensing information.

On line 13, we invoke the `kernel-module` helper infrastructure, that generates all the appropriate Makefile rules and variables to build that kernel module.

Finally, on line 14, we invoke the `generic-package` infrastructure.

The dependency on `linux` is automatically added, so it is not needed to specify it in `FOO_DEPENDENCIES`.

What you may have noticed is that, unlike other package infrastructures, we explicitly invoke a second infrastructure. This allows a package to build a kernel module, but also, if needed, use any one of other package infrastructures to build normal userland components (libraries, executables…). Using the `kernel-module` infrastructure on its own is not sufficient; another package infrastructure **must** be used.

Let's look at a more complex example:

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: FOO_VERSION = 1.2.3
08: FOO_SOURCE = foo-$(FOO_VERSION).tar.xz
09: FOO_SITE = http://www.foosoftware.org/download
10: FOO_LICENSE = GPL-2.0
11: FOO_LICENSE_FILES = COPYING
12:
13: FOO_MODULE_SUBDIRS = driver/base
14: FOO_MODULE_MAKE_OPTS = KVERSION=$(LINUX_VERSION_PROBED)
15:
16: ifeq ($(BR2_PACKAGE_LIBBAR),y)
17: FOO_DEPENDENCIES += libbar
18: FOO_CONF_OPTS += --enable-bar
19: FOO_MODULE_SUBDIRS += driver/bar
20: else
21: FOO_CONF_OPTS += --disable-bar
22: endif
23:
24: $(eval $(kernel-module))
26: $(eval $(autotools-package))
```

Here, we see that we have an autotools-based package, that also builds the kernel module located in sub-directory `driver/base` and, if libbar is enabled, the kernel module located in sub-directory `driver/bar`, and defines the variable `KVERSION` to be passed to the Linux buildsystem when building the module(s).

18.20.2. `kernel-module` reference

The main macro for the kernel module infrastructure is `kernel-module`. Unlike other package infrastructures, it is not stand-alone, and requires any of the other `*-package` macros be called after it.

The `kernel-module` macro defines post-build and post-target-install hooks to build the kernel modules. If the package's `.mk` needs access to the built kernel modules, it should do so in a post-build hook, **registered after** the call to `kernel-module`. Similarly, if the package's `.mk` needs access to the kernel module after it has been installed, it should do so in a post-install hook, **registered after** the call to `kernel-module`. Here's an example:

```
$(eval $(kernel-module))

define FOO_DO_STUFF_WITH_KERNEL_MODULE
    # Do something with it...
endef
FOO_POST_BUILD_HOOKS += FOO_DO_STUFF_WITH_KERNEL_MODULE

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

Finally, unlike the other package infrastructures, there is no `host-kernel-module` variant to build a host kernel module.

The following additional variables can optionally be defined to further configure the build of the kernel module:

- `FOO_MODULE_SUBDIRS` may be set to one or more sub-directories (relative to the package source top-directory) where the kernel module sources are. If empty or not set, the sources for the kernel module(s) are considered to be located at the top of the package source tree.
- `FOO_MODULE_MAKE_OPTS` may be set to contain extra variable definitions to pass to the Linux buildsystem.

You may also reference (but you may **not** set!) those variables:

- `LINUX_DIR` contains the path to where the Linux kernel has been extracted and built.
- `LINUX_VERSION` contains the version string as configured by the user.
- `LINUX_VERSION_PROBED` contains the real version string of the kernel, retrieved with running `make -C $(LINUX_DIR) kernelrelease`
- `KERNEL_ARCH` contains the name of the current architecture, like `arm`, `mips`…

18.21. Infrastructure for asciidoc documents

The Buildroot manual, which you are currently reading, is entirely written using the AsciiDoc mark-up syntax. The manual is then rendered to many formats:

- html
- split-html
- pdf
- epub
- text

Although Buildroot only contains one document written in AsciiDoc, there is, as for packages, an infrastructure for rendering documents using the AsciiDoc syntax.

Also as for packages, the AsciiDoc infrastructure is available from a br2-external tree. This allows documentation for a br2-external tree to match the Buildroot documentation, as it will be rendered to the same formats and use the same layout and theme.

18.21.1. `asciidoc-document` tutorial

Whereas package infrastructures are suffixed with `-package`, the document infrastructures are suffixed with `-document`. So, the AsciiDoc infrastructure is named `asciidoc-document`.

Here is an example to render a simple AsciiDoc document.

```
01: ################################################################################
02: #
03: # foo-document
04: #
05: ################################################################################
06:
07: FOO_SOURCES = $(sort $(wildcard $(FOO_DOCDIR)/*))
08: $(eval $(call asciidoc-document))
```

On line 7, the Makefile declares what the sources of the document are. Currently, it is expected that the document's sources are only local; Buildroot will not attempt to download anything to render a document. Thus, you must indicate where the sources are. Usually, the string above is sufficient for a document with no sub-directory structure.

On line 8, we call the `asciidoc-document` function, which generates all the Makefile code necessary to render the document.

18.21.2. `asciidoc-document` reference

The list of variables that can be set in a `.mk` file to give metadata information is (assuming the document name is `foo`) :

- `FOO_SOURCES`, mandatory, defines the source files for the document.
- `FOO_RESOURCES`, optional, may contain a space-separated list of paths to one or more directories containing so-called resources (like CSS or images). By default, empty.
- `FOO_DEPENDENCIES`, optional, the list of packages (most probably, host-packages) that must be built before building this document.

There are also additional hooks (see Section 18.23, "Hooks available in the various build steps" for general information on hooks), that a document may set to define extra actions to be done at various steps:

- `FOO_POST_RSYNC_HOOKS` to run additional commands after the sources have been copied by Buildroot. This can for example be used to generate part of the manual with information extracted from the tree. As an example, Buildroot uses this hook to generate the tables in the appendices.
- `FOO_CHECK_DEPENDENCIES_HOOKS` to run additional tests on required components to generate the document. In AsciiDoc, it is possible to call filters, that is, programs that will parse an AsciiDoc block and render it appropriately (e.g. ditaa or aafigure).
- `FOO_CHECK_DEPENDENCIES_<FMT>_HOOKS`, to run additional tests for the specified format `<FMT>` (see the list of rendered formats, above).

Buildroot sets the following variable that can be used in the definitions above:

- `$(FOO_DOCDIR)`, similar to `$(FOO_PKGDIR)`, contains the path to the directory containing `foo.mk`. It can be used to refer to the document sources, and can be used in the hooks, especially the post-rsync hook if parts of the documentation needs to be generated.
- `$(@D)`, as for traditional packages, contains the path to the directory where the document will be copied and built.

Here is a complete example that uses all variables and all hooks:

```
01: ################################################################################
02: #
03: # foo-document
04: #
05: ################################################################################
```

```
06:
07: FOO_SOURCES = $(sort $(wildcard $(FOO_DOCDIR)/*))
08: FOO_RESOURCES = $(sort $(wildcard $(FOO_DOCDIR)/ressources))
09:
10: define FOO_GEN_EXTRA_DOC
11:     /path/to/generate-script --outdir=$(@D)
12: endef
13: FOO_POST_RSYNC_HOOKS += FOO_GEN_EXTRA_DOC
14:
15: define FOO_CHECK_MY_PROG
16:     if ! which my-prog >/dev/null 2>&1; then \
17:         echo "You need my-prog to generate the foo document"; \
18:         exit 1; \
19:     fi
20: endef
21: FOO_CHECK_DEPENDENCIES_HOOKS += FOO_CHECK_MY_PROG
22:
23: define FOO_CHECK_MY_OTHER_PROG
24:     if ! which my-other-prog >/dev/null 2>&1; then \
25:         echo "You need my-other-prog to generate the foo document as PDF"; \
26:         exit 1; \
27:     fi
28: endef
29: FOO_CHECK_DEPENDENCIES_PDF_HOOKS += FOO_CHECK_MY_OTHER_PROG
30:
31: $(eval $(call asciidoc-document))
```

18.22. Infrastructure specific to the Linux kernel package

The Linux kernel package can use some specific infrastructures based on package hooks for building Linux kernel tools or/and building Linux kernel extensions.

18.22.1. linux-kernel-tools

Buildroot offers a helper infrastructure to build some userspace tools for the target available within the Linux kernel sources. Since their source code is part of the kernel source code, a special package, `linux-tools`, exists and re-uses the sources of the Linux kernel that runs on the target.

Let’s look at an example of a Linux tool. For a new Linux tool named `foo`, create a new menu entry in the existing `package/linux-tools/Config.in`. This file will contain the option descriptions related to each kernel tool that will be used and displayed in the configuration tool. It would basically look like:

```
01: config BR2_PACKAGE_LINUX_TOOLS_FOO
02:     bool "foo"
03:     select BR2_PACKAGE_LINUX_TOOLS
04:     help
05:       This is a comment that explains what foo kernel tool is.
06:
07:       http://foosoftware.org/foo/
```

The name of the option starts with the prefix `BR2_PACKAGE_LINUX_TOOLS_`, followed by the uppercase name of the tool (like is done for packages).

Note. Unlike other packages, the `linux-tools` package options appear in the `linux` kernel menu, under the `Linux Kernel Tools` sub-menu, not under the `Target packages` main menu.

Then for each linux tool, add a new `.mk.in` file named `package/linux-tools/linux-tool-foo.mk.in`. It would basically look like:

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: LINUX_TOOLS += foo
08:
09: FOO_DEPENDENCIES = libbbb
10:
11: define FOO_BUILD_CMDS
12:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools foo
13: endef
14:
15: define FOO_INSTALL_STAGING_CMDS
16:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
17:             DESTDIR=$(STAGING_DIR) \
18:             foo_install
19: endef
20:
21: define FOO_INSTALL_TARGET_CMDS
22:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
23:             DESTDIR=$(TARGET_DIR) \
24:             foo_install
25: endef
```

On line 7, we register the Linux tool `foo` to the list of available Linux tools.

On line 9, we specify the list of dependencies this tool relies on. These dependencies are added to the Linux package dependencies list only when the `foo` tool is selected.

The rest of the Makefile, lines 11-25 defines what should be done at the different steps of the Linux tool build process like for a `generic package`. They will actually be used only when the `foo` tool is selected. The only supported commands are `_BUILD_CMDS`, `_INSTALL_STAGING_CMDS` and `_INSTALL_TARGET_CMDS`.

Note. One **must not** call `$(eval $(generic-package))` or any other package infrastructure! Linux tools are not packages by themselves, they are part of the `linux-tools` package.

18.22.2. linux-kernel-extensions

Some packages provide new features that require the Linux kernel tree to be modified. This can be in the form of patches to be applied on the kernel tree, or in the form of new files to be added to the tree. The Buildroot's Linux kernel extensions infrastructure provides a simple solution to automatically do this, just after the kernel sources are extracted and before the kernel patches are applied. Examples of extensions packaged using this mechanism are the real-time extensions Xenomai and RTAI, as well as the set of out-of-tree LCD screens drivers `fbtft`.

Let's look at an example on how to add a new Linux extension `foo`.

First, create the package `foo` that provides the extension: this package is a standard package; see the previous chapters on how to create such a package. This package is in charge of downloading the sources archive, checking the hash, defining the licence informations and building user space tools if any.

Then create the *Linux extension* proper: create a new menu entry in the existing `linux/Config.ext.in`. This file contains the option descriptions related to each kernel extension that will be used and displayed in the configuration tool. It would basically look like:

```
01: config BR2_LINUX_KERNEL_EXT_FOO
02:     bool "foo"
03:     help
04:       This is a comment that explains what foo kernel extension is.
05:
06:       http://foosoftware.org/foo/
```

Then for each linux extension, add a new `.mk` file named `linux/linux-ext-foo.mk`. It should basically contain:

```
01: ################################################################################
02: #
03: # foo
04: #
05: ################################################################################
06:
07: LINUX_EXTENSIONS += foo
08:
09: define FOO_PREPARE_KERNEL
10:     $(FOO_DIR)/prepare-kernel-tree.sh --linux-dir=$(@D)
11: endef
```

On line 7, we add the Linux extension `foo` to the list of available Linux extensions.

On line 9-11, we define what should be done by the extension to modify the Linux kernel tree; this is specific to the linux extension and can use the variables defined by the `foo` package, like: `$(FOO_DIR)` or `$(FOO_VERSION)` … as well as all the Linux variables, like: `$(LINUX_VERSION)` or `$(LINUX_VERSION_PROBED)`, `$(KERNEL_ARCH)` … See the definition of those kernel variables.

18.23. Hooks available in the various build steps

The generic infrastructure (and as a result also the derived autotools and cmake infrastructures) allow packages to specify hooks. These define further actions to perform after existing steps. Most hooks aren't really useful for generic packages, since the `.mk` file already has full control over the actions performed in each step of the package construction.

The following hook points are available:

- `LIBFOO_PRE_DOWNLOAD_HOOKS`
- `LIBFOO_POST_DOWNLOAD_HOOKS`
- `LIBFOO_PRE_EXTRACT_HOOKS`
- `LIBFOO_POST_EXTRACT_HOOKS`
- `LIBFOO_PRE_RSYNC_HOOKS`
- `LIBFOO_POST_RSYNC_HOOKS`
- `LIBFOO_PRE_PATCH_HOOKS`
- `LIBFOO_POST_PATCH_HOOKS`
- `LIBFOO_PRE_CONFIGURE_HOOKS`
- `LIBFOO_POST_CONFIGURE_HOOKS`
- `LIBFOO_PRE_BUILD_HOOKS`

- LIBFOO_POST_BUILD_HOOKS
- LIBFOO_PRE_INSTALL_HOOKS (for host packages only)
- LIBFOO_POST_INSTALL_HOOKS (for host packages only)
- LIBFOO_PRE_INSTALL_STAGING_HOOKS (for target packages only)
- LIBFOO_POST_INSTALL_STAGING_HOOKS (for target packages only)
- LIBFOO_PRE_INSTALL_TARGET_HOOKS (for target packages only)
- LIBFOO_POST_INSTALL_TARGET_HOOKS (for target packages only)
- LIBFOO_PRE_INSTALL_IMAGES_HOOKS
- LIBFOO_POST_INSTALL_IMAGES_HOOKS
- LIBFOO_PRE_LEGAL_INFO_HOOKS
- LIBFOO_POST_LEGAL_INFO_HOOKS
- LIBFOO_TARGET_FINALIZE_HOOKS

These variables are *lists* of variable names containing actions to be performed at this hook point. This allows several hooks to be registered at a given hook point. Here is an example:

```
define LIBFOO_POST_PATCH_FIXUP
        action1
        action2
endef

LIBFOO_POST_PATCH_HOOKS += LIBFOO_POST_PATCH_FIXUP
```

18.23.1. Using the POST_RSYNC hook

The POST_RSYNC hook is run only for packages that use a local source, either through the local site method or the OVERRIDE_SRCDIR mechanism. In this case, package sources are copied using rsync from the local location into the buildroot build directory. The rsync command does not copy all files from the source directory, though. Files belonging to a version control system, like the directories .git, .hg, etc. are not copied. For most packages this is sufficient, but a given package can perform additional actions using the POST_RSYNC hook.

In principle, the hook can contain any command you want. One specific use case, though, is the intentional copying of the version control directory using rsync. The rsync command you use in the hook can, among others, use the following variables:

- $(SRCDIR): the path to the overridden source directory
- $(@D): the path to the build directory

18.23.2. Target-finalize hook

Packages may also register hooks in LIBFOO_TARGET_FINALIZE_HOOKS. These hooks are run after all packages are built, but before the filesystem images are generated. They are seldom used, and your package probably do not need them.

18.24. Gettext integration and interaction with packages

Many packages that support internationalization use the gettext library. Dependencies for this library are fairly complicated and therefore, deserve some explanation.

The *glibc* C library integrates a full-blown implementation of *gettext*, supporting translation. Native Language Support is therefore built-in in *glibc*.

On the other hand, the *uClibc* and *musl* C libraries only provide a stub implementation of the gettext functionality, which allows to compile libraries and programs using gettext functions, but without providing the translation capabilities of a full-blown gettext implementation. With such C libraries, if real Native Language Support is necessary, it can be provided by the libintl library of the gettext package.

Due to this, and in order to make sure that Native Language Support is properly handled, packages in Buildroot that can use NLS support should:

1. Ensure NLS support is enabled when BR2_SYSTEM_ENABLE_NLS=y. This is done automatically for *autotools* packages and therefore should only be done for packages using other package infrastructures.
2. Add $(TARGET_NLS_DEPENDENCIES) to the package <pkg>_DEPENDENCIES variable. This addition should be done unconditionally: the value of this variable is automatically adjusted by the core infrastructure to contain the relevant list of packages. If NLS support is disabled, this variable is empty. If NLS support is enabled, this variable contains host-gettext so that tools needed to compile translation files are available on the host. In addition, if *uClibc* or *musl* are used, this variable also contains gettext in order to get the full-blown *gettext* implementation.
3. If needed, add $(TARGET_NLS_LIBS) to the linker flags, so that the package gets linked with libintl. This is generally not needed with *autotools* packages as they usually detect automatically that they should link with libintl. However, packages using other build systems, or problematic autotools-based packages may need this. $(TARGET_NLS_LIBS) should be added unconditionally to the linker flags, as the core automatically makes it empty or defined to -lintl depending on the configuration.

No changes should be made to the `Config.in` file to support NLS.

Finally, certain packages need some gettext utilities on the target, such as the `gettext` program itself, which allows to retrieve translated strings, from the command line. In such a case, the package should:

- use `select BR2_PACKAGE_GETTEXT` in their `Config.in` file, indicating in a comment above that it's a runtime dependency only.
- not add any `gettext` dependency in the `DEPENDENCIES` variable of their `.mk` file.

18.25. Tips and tricks

18.25.1. Package name, config entry name and makefile variable relationship

In Buildroot, there is some relationship between:

- the *package name*, which is the package directory name (and the name of the `*.mk` file);
- the config entry name that is declared in the `Config.in` file;
- the makefile variable prefix.

It is mandatory to maintain consistency between these elements, using the following rules:

- the package directory and the `*.mk` name are the *package name* itself (e.g.: `package/foo-bar_boo/foo-bar_boo.mk`);
- the *make* target name is the *package name* itself (e.g.: `foo-bar_boo`);
- the config entry is the upper case *package name* with `.` and `-` characters substituted with _, prefixed with `BR2_PACKAGE_` (e.g.: `BR2_PACKAGE_FOO_BAR_BOO`);
- the `*.mk` file variable prefix is the upper case *package name* with `.` and `-` characters substituted with _ (e.g.: `FOO_BAR_BOO_VERSION`).

18.25.2. How to check the coding style

Buildroot provides a script in `utils/check-package` that checks new or changed files for coding style. It is not a complete language validator, but it catches many common mistakes. It is meant to run in the actual files you created or modified, before creating the patch for submission.

This script can be used for packages, filesystem makefiles, Config.in files, etc. It does not check the files defining the package infrastructures and some other files containing similar common code.

To use it, run the `check-package` script, by telling which files you created or changed:

```
$ ./utils/check-package package/new-package/*
```

If you have the `utils` directory in your path you can also run:

```
$ cd package/new-package/
$ check-package *
```

The tool can also be used for packages in a br2-external:

```
$ check-package -b /path/to/br2-ext-tree/package/my-package/*
```

18.25.3. How to test your package

Once you have added your new package, it is important that you test it under various conditions: does it build for all architectures? Does it build with the different C libraries? Does it need threads, NPTL? And so on…

Buildroot runs autobuilders which continuously test random configurations. However, these only build the `master` branch of the git tree, and your new fancy package is not yet there.

Buildroot provides a script in `utils/test-pkg` that uses the same base configurations as used by the autobuilders so you can test your package in the same conditions.

First, create a config snippet that contains all the necessary options needed to enable your package, but without any architecture or toolchain option. For example, let's create a config snippet that just enables `libcurl`, without any TLS backend:

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
```

If your package needs more configuration options, you can add them to the config snippet. For example, here's how you would test `libcurl` with `openssl` as a TLS backend and the `curl` program:

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
BR2_PACKAGE_LIBCURL_CURL=y
BR2_PACKAGE_OPENSSL=y
```

Then run the `test-pkg` script, by telling it what config snippet to use and what package to test:

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
```

By default, `test-pkg` will build your package against a subset of the toolchains used by the autobuilders, which has been selected by the Buildroot developers as being the most useful and representative subset. If you want to test all toolchains, pass the `-a` option. Note that in any case, internal toolchains are excluded as they take too long to build.

The output lists all toolchains that are tested and the corresponding result (excerpt, results are fake):

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
                armv5-ctng-linux-gnueabi [ 1/11]: OK
               armv7-ctng-linux-gnueabihf [ 2/11]: OK
                         br-aarch64-glibc [ 3/11]: SKIPPED
                            br-arcle-hs38 [ 4/11]: SKIPPED
                             br-arm-basic [ 5/11]: FAILED
                   br-arm-cortex-a9-glibc [ 6/11]: OK
                    br-arm-cortex-a9-musl [ 7/11]: FAILED
                    br-arm-cortex-m4-full [ 8/11]: OK
                               br-arm-full [ 9/11]: OK
                     br-arm-full-nothread [10/11]: FAILED
                       br-arm-full-static [11/11]: OK
11 builds, 2 skipped, 2 build failed, 1 legal-info failed
```

The results mean:

- `OK`: the build was successful.
- `SKIPPED`: one or more configuration options listed in the config snippet were not present in the final configuration. This is due to options having dependencies not satisfied by the toolchain, such as for example a package that `depends on BR2_USE_MMU` with a noMMU toolchain. The missing options are reported in `missing.config` in the output build directory (`~/br-test-pkg/TOOLCHAIN_NAME/` by default).
- `FAILED`: the build failed. Inspect the `logfile` file in the output build directory to see what went wrong:
  - the actual build failed,
  - the legal-info failed,
  - one of the preliminary steps (downloading the config file, applying the configuration, running `dirclean` for the package) failed.

When there are failures, you can just re-run the script with the same options (after you fixed your package); the script will attempt to re-build the package specified with `-p` for all toolchains, without the need to re-build all the dependencies of that package.

The `test-pkg` script accepts a few options, for which you can get some help by running:

```
$ ./utils/test-pkg -h
```

18.25.4. How to add a package from GitHub

Packages on GitHub often don't have a download area with release tarballs. However, it is possible to download tarballs directly from the repository on GitHub. As GitHub is known to have changed download mechanisms in the past, the *github* helper function should be used as shown below.

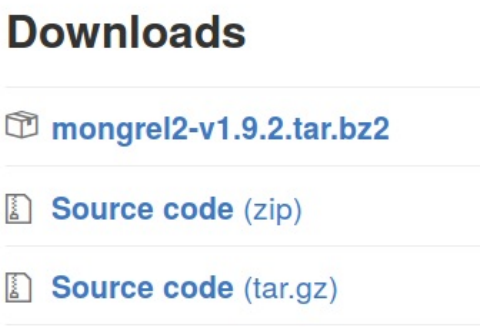
```
# Use a tag or a full commit ID
FOO_VERSION = 1.0
FOO_SITE = $(call github,<user>,<package>,v$(FOO_VERSION))
```

Notes

- The FOO_VERSION can either be a tag or a commit ID.
- The tarball name generated by github matches the default one from Buildroot (e.g.: `foo-f6fb6654af62045239caed5950bc6c7971965e60.tar.gz`), so it is not necessary to specify it in the `.mk` file.
- When using a commit ID as version, you should use the full 40 hex characters.
- When the tag contains a prefix such as `v` in `v1.0`, then the `VERSION` variable should contain just `1.0`, and the `v` should be added directly in the `SITE` variable, as illustrated above. This ensures that the `VERSION` variable value can be used to match against release-monitoring.org results.

If the package you wish to add does have a release section on GitHub, the maintainer may have uploaded a release tarball, or the release may just point to the automatically generated tarball from the git tag. If there is a release tarball uploaded by the maintainer, we prefer to use that since it may be slightly different (e.g. it contains a configure script so we don't need to do AUTORECONF).

You can see on the release page if it's an uploaded tarball or a git tag:



- If it looks like the image above then it was uploaded by the maintainer and you should use that link (in that example: *mongrel2-v1.9.2.tar.bz2*) to specify `FOO_SITE`, and not use the *github* helper.
- On the other hand, if there's is **only** the "Source code" link, then it's an automatically generated tarball and you should use the *github* helper function.

18.25.5. How to add a package from Gitlab

In a similar way to the `github` macro described in Section 18.25.4, "How to add a package from GitHub", Buildroot also provides the `gitlab` macro to download from Gitlab repositories. It can be used to download auto-generated tarballs produced by Gitlab, either for specific tags or commits:

```
# Use a tag or a full commit ID
FOO_VERSION = 1.0
FOO_SITE = $(call gitlab,<user>,<package>,v$(FOO_VERSION))
```

By default, it will use a `.tar.gz` tarball, but Gitlab also provides `.tar.bz2` tarballs, so by adding a `<pkg>_SOURCE` variable, this `.tar.bz2` tarball can be used:

```
# Use a tag or a full commit ID
FOO_VERSION = 1.0
FOO_SITE = $(call gitlab,<user>,<package>,v$(FOO_VERSION))
FOO_SOURCE = foo-$(FOO_VERSION).tar.bz2
```

If there is a specific tarball uploaded by the upstream developers in `https://gitlab.com/<project>/releases/`, do not use this macro, but rather use directly the link to the tarball.

18.26. Conclusion

As you can see, adding a software package to Buildroot is simply a matter of writing a Makefile using an existing example and modifying it according to the compilation process required by the package.

If you package software that might be useful for other people, don't forget to send a patch to the Buildroot mailing list (see Section 22.5, "Submitting patches")!

Chapter 19. Patching a package

While integrating a new package or updating an existing one, it may be necessary to patch the source of the software to get it cross-built within Buildroot.

Buildroot offers an infrastructure to automatically handle this during the builds. It supports three ways of applying patch sets: downloaded patches, patches supplied within buildroot and patches located in a user-defined global patch directory.

19.1. Providing patches

19.1.1. Downloaded

If it is necessary to apply a patch that is available for download, then add it to the `<packagename>_PATCH` variable. If an entry contains `://`, then Buildroot will assume it is a full URL and download the patch from this location. Otherwise, Buildroot will assume that the patch should be downloaded from `<packagename>_SITE`. It can be a single patch, or a tarball containing a patch series.

Like for all downloads, a hash should be added to the `<packagename>.hash` file.

This method is typically used for packages from Debian.

19.1.2. Within Buildroot

Most patches are provided within Buildroot, in the package directory; these typically aim to fix cross-compilation, libc support, or other such issues.

These patch files should be named `<number>-<description>.patch`.

Notes

- The patch files coming with Buildroot should not contain any package version reference in their filename.

- The field `<number>` in the patch file name refers to the *apply order*, and shall start at 1; It is preferred to pad the number with zeros up to 4 digits, like *git-format-patch* does. E.g.: `0001-foobar-the-buz.patch`
- The patch email subject prefix shall not be numbered. Patches shall be generated with the `git format-patch -N` command, since this numbering is automatically added for series. For example, the patch subject line should look like `Subject: [PATCH] foobar the buz` rather than `Subject: [PATCH n/m] foobar the buz`.
- Previously, it was mandatory for patches to be prefixed with the name of the package, like `<package>-<number>-<description>.patch`, but that is no longer the case. Existing packages will be fixed as time passes. *Do not prefix patches with the package name.*
- Previously, a `series` file, as used by `quilt`, could also be added in the package directory. In that case, the `series` file defines the patch application order. This is deprecated, and will be removed in the future. *Do not use a series file.*

19.1.3. Global patch directory

The `BR2_GLOBAL_PATCH_DIR` configuration file option can be used to specify a space separated list of one or more directories containing global package patches. See Section 9.8, "Adding project-specific patches" for details.

19.2. How patches are applied

1. Run the `<packagename>_PRE_PATCH_HOOKS` commands if defined;
2. Cleanup the build directory, removing any existing `*.rej` files;
3. If `<packagename>_PATCH` is defined, then patches from these tarballs are applied;
4. If there are some `*.patch` files in the package's Buildroot directory or in a package subdirectory named `<packageversion>`, then:
   - If a `series` file exists in the package directory, then patches are applied according to the `series` file;
   - Otherwise, patch files matching `*.patch` are applied in alphabetical order. So, to ensure they are applied in the right order, it is highly recommended to name the patch files like this: `<number>-<description>.patch`, where `<number>` refers to the *apply order*.
5. If `BR2_GLOBAL_PATCH_DIR` is defined, the directories will be enumerated in the order they are specified. The patches are applied as described in the previous step.
6. Run the `<packagename>_POST_PATCH_HOOKS` commands if defined.

If something goes wrong in the steps *3* or *4*, then the build fails.

19.3. Format and licensing of the package patches

Patches are released under the same license as the software they apply to (see Section 13.2, "Complying with the Buildroot license").

A message explaining what the patch does, and why it is needed, should be added in the header commentary of the patch.

You should add a `Signed-off-by` statement in the header of the each patch to help with keeping track of the changes and to certify that the patch is released under the same license as the software that is modified.

If the software is under version control, it is recommended to use the upstream SCM software to generate the patch set.

Otherwise, concatenate the header with the output of the `diff -purN package-version.orig/ package-version/` command.

If you update an existing patch (e.g. when bumping the package version), make sure the existing From header and Signed-off-by tags are not removed, but do update the rest of the patch comment when appropriate.

At the end, the patch should look like:

```
configure.ac: add C++ support test

Signed-off-by: John Doe <john.doe@noname.org>

--- configure.ac.orig
+++ configure.ac
@@ -40,2 +40,12 @@

AC_PROG_MAKE_SET
+
+AC_CACHE_CHECK([whether the C++ compiler works],
+                [rw_cv_prog_cxx_works],
+                [AC_LANG_PUSH([C++])
+                 AC_LINK_IFELSE([AC_LANG_PROGRAM([], [])],
+                                [rw_cv_prog_cxx_works=yes],
+                                [rw_cv_prog_cxx_works=no])
+                 AC_LANG_POP([C++])])
+
+AM_CONDITIONAL([CXX_WORKS], [test "x$rw_cv_prog_cxx_works" = "xyes"])
```

19.4. Integrating patches found on the Web

When integrating a patch of which you are not the author, you have to add a few things in the header of the patch itself.

Depending on whether the patch has been obtained from the project repository itself, or from somewhere on the web, add one of the following tags:

```
Backported from: <some commit id>
```

or

```
Fetch from: <some url>
```

It is also sensible to add a few words about any changes to the patch that may have been necessary.

Chapter 20. Download infrastructure

TODO

Chapter 21. Debugging Buildroot

It is possible to instrument the steps `Buildroot` does when building packages. Define the variable `BR2_INSTRUMENTATION_SCRIPTS` to contain the path of one or more scripts (or other executables), in a space-separated list, you want called before and after each step. The scripts are called in sequence, with three parameters:

- `start` or `end` to denote the start (resp. the end) of a step;
- the name of the step about to be started, or which just ended;
- the name of the package.

For example :

```
make BR2_INSTRUMENTATION_SCRIPTS="/path/to/my/script1 /path/to/my/script2"
```

The list of steps is:

- `extract`
- `patch`
- `configure`
- `build`
- `install-host`, when a host-package is installed in `$(HOST_DIR)`
- `install-target`, when a target-package is installed in `$(TARGET_DIR)`
- `install-staging`, when a target-package is installed in `$(STAGING_DIR)`
- `install-image`, when a target-package installs files in `$(BINARIES_DIR)`

The script has access to the following variables:

- `BR2_CONFIG`: the path to the Buildroot .config file
- `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`: see Section 18.6.2, “`generic-package` reference”
- `BUILD_DIR`: the directory where packages are extracted and built
- `BINARIES_DIR`: the place where all binary files (aka images) are stored
- `BASE_DIR`: the base output directory

Chapter 22. Contributing to Buildroot

There are many ways in which you can contribute to Buildroot: analyzing and fixing bugs, analyzing and fixing package build failures detected by the autobuilders, testing and reviewing patches sent by other developers, working on the items in our TODO list and sending your own improvements to Buildroot or its manual. The following sections give a little more detail on each of these items.

If you are interested in contributing to Buildroot, the first thing you should do is to subscribe to the Buildroot mailing list. This list is the main way of interacting with other Buildroot developers and to send contributions to. If you aren’t subscribed yet, then refer to Chapter 5, *Community resources* for the subscription link.

If you are going to touch the code, it is highly recommended to use a git repository of Buildroot, rather than starting from an extracted source code tarball. Git is the easiest way to develop from and directly send your patches to the mailing list. Refer to Chapter 3, *Getting Buildroot* for more information on obtaining a Buildroot git tree.

22.1. Reproducing, analyzing and fixing bugs

A first way of contributing is to have a look at the open bug reports in the Buildroot bug tracker. As we strive to keep the bug count as small as possible, all help in reproducing, analyzing and fixing reported bugs is more than welcome. Don’t hesitate to add a comment to bug reports reporting your findings, even if you don’t yet see the full picture.

22.2. Analyzing and fixing autobuild failures

The Buildroot autobuilders are a set of build machines that continuously run Buildroot builds based on random configurations. This is done for all architectures supported by Buildroot, with various toolchains, and with a random selection of packages. With the large commit activity on Buildroot, these autobuilders are a great help in detecting problems very early after commit.

All build results are available at http://autobuild.buildroot.org, statistics are at http://autobuild.buildroot.org/stats.php. Every day, an overview of all failed packages is sent to the mailing list.

Detecting problems is great, but obviously these problems have to be fixed as well. Your contribution is very welcome here! There are basically two things that can be done:

- Analyzing the problems. The daily summary mails do not contain details about the actual failures: in order to see what's going on you have to open the build log and check the last output. Having someone doing this for all packages in the mail is very useful for other developers, as they can make a quick initial analysis based on this output alone.
- Fixing a problem. When fixing autobuild failures, you should follow these steps:
  1. Check if you can reproduce the problem by building with the same configuration. You can do this manually, or use the br-reproduce-build script that will automatically clone a Buildroot git repository, checkout the correct revision, download and set the right configuration, and start the build.
  2. Analyze the problem and create a fix.
  3. Verify that the problem is really fixed by starting from a clean Buildroot tree and only applying your fix.
  4. Send the fix to the Buildroot mailing list (see Section 22.5, "Submitting patches"). In case you created a patch against the package sources, you should also send the patch upstream so that the problem will be fixed in a later release, and the patch in Buildroot can be removed. In the commit message of a patch fixing an autobuild failure, add a reference to the build result directory, as follows:

```
Fixes: http://autobuild.buildroot.org/results/51000a9d4656afe9e0ea6f07b9f8ed374c2e4069
```

22.3. Reviewing and testing patches

With the amount of patches sent to the mailing list each day, the maintainer has a very hard job to judge which patches are ready to apply and which ones aren't. Contributors can greatly help here by reviewing and testing these patches.

In the review process, do not hesitate to respond to patch submissions for remarks, suggestions or anything that will help everyone to understand the patches and make them better. Please use internet style replies in plain text emails when responding to patch submissions.

To indicate approval of a patch, there are three formal tags that keep track of this approval. To add your tag to a patch, reply to it with the approval tag below the original author's Signed-off-by line. These tags will be picked up automatically by patchwork (see Section 22.3.1, "Applying Patches from Patchwork") and will be part of the commit log when the patch is accepted.

Tested-by
: Indicates that the patch has been tested successfully. You are encouraged to specify what kind of testing you performed (compile-test on architecture X and Y, runtime test on target A, …). This additional information helps other testers and the maintainer.

Reviewed-by
: Indicates that you code-reviewed the patch and did your best in spotting problems, but you are not sufficiently familiar with the area touched to provide an Acked-by tag. This means that there may be remaining problems in the patch that would be spotted by someone with more experience in that area. Should such problems be detected, your Reviewed-by tag remains appropriate and you cannot be blamed.

Acked-by
: Indicates that you code-reviewed the patch and you are familiar enough with the area touched to feel that the patch can be committed as-is (no additional changes required). In case it later turns out that something is wrong with the patch, your Acked-by could be considered inappropriate. The difference between Acked-by and Reviewed-by is thus mainly that you are prepared to take the blame on Acked patches, but not on Reviewed ones.

If you reviewed a patch and have comments on it, you should simply reply to the patch stating these comments, without providing a Reviewed-by or Acked-by tag. These tags should only be provided if you judge the patch to be good as it is.

It is important to note that neither Reviewed-by nor Acked-by imply that testing has been performed. To indicate that you both reviewed and tested the patch, provide two separate tags (Reviewed/Acked-by and Tested-by).

Note also that *any developer* can provide Tested/Reviewed/Acked-by tags, without exception, and we encourage everyone to do this. Buildroot does not have a defined group of *core* developers, it just so happens that some developers are more active than others. The maintainer will value tags according to the track record of their submitter. Tags provided by a regular contributor will naturally be trusted more than tags provided by a newcomer. As you provide tags more regularly, your *trustworthiness* (in the eyes of the maintainer) will go up, but *any* tag provided is valuable.

Buildroot's Patchwork website can be used to pull in patches for testing purposes. Please see Section 22.3.1, "Applying Patches from Patchwork" for more information on using Buildroot's Patchwork website to apply patches.

22.3.1. Applying Patches from Patchwork

The main use of Buildroot's Patchwork website for a developer is for pulling in patches into their local git repository for testing purposes.

When browsing patches in the patchwork management interface, an `mbox` link is provided at the top of the page. Copy this link address and run the following commands:

```
$ git checkout -b <test-branch-name>
$ wget -O - <mbox-url> | git am
```

Another option for applying patches is to create a bundle. A bundle is a set of patches that you can group together using the patchwork interface. Once the bundle is created and the bundle is made public, you can copy the `mbox` link for the bundle and apply the bundle using the above commands.

22.4. Work on items from the TODO list

If you want to contribute to Buildroot but don't know where to start, and you don't like any of the above topics, you can always work on items from the Buildroot TODO list. Don't hesitate to discuss an item first on the mailing list or on IRC. Do edit the wiki to indicate when you start working on an item, so we avoid duplicate efforts.

22.5. Submitting patches

Note

Please, do not attach patches to bugs, send them to the mailing list instead.

If you made some changes to Buildroot and you would like to contribute them to the Buildroot project, proceed as follows.

22.5.1. The formatting of a patch

We expect patches to be formatted in a specific way. This is necessary to make it easy to review patches, to be able to apply them easily to the git repository, to make it easy to find back in the history how and why things have changed, and to make it possible to use `git bisect` to locate the origin of a problem.

First of all, it is essential that the patch has a good commit message. The commit message should start with a separate line with a brief summary of the change, prefixed by the area touched by the patch. A few examples of good commit titles:

- `package/linuxptp: bump version to 2.0`
- `configs/imx23evk: bump Linux version to 4.19`
- `package/pkg-generic: postpone evaluation of dependency conditions`
- `boot/uboot: needs host-{flex,bison}`
- `support/testing: add python-ubjson tests`

The description that follows the prefix should start with a lower case letter (i.e "bump", "needs", "postpone", "add" in the above examples).

Second, the body of the commit message should describe *why* this change is needed, and if necessary also give details about *how* it was done. When writing the commit message, think of how the reviewers will read it, but also think about how you will read it when you look at this change again a few years down the line.

Third, the patch itself should do only one change, but do it completely. Two unrelated or weakly related changes should usually be done in two separate patches. This usually means that a patch affects only a single package. If several changes are related, it is often still possible to split them up in small patches and apply them in a specific order. Small patches make it easier to review, and often make it easier to understand afterwards why a change was done. However, each patch must be complete. It is not allowed that the build is broken when only the first but not the second patch is applied. This is necessary to be able to use `git bisect` afterwards.

Of course, while you're doing your development, you're probably going back and forth between packages, and certainly not committing things immediately in a way that is clean enough for submission. So most developers rewrite the history of commits to produce a clean set of commits that is appropriate for submission. To do this, you need to use *interactive rebasing*. You can learn about it in the Pro Git book. Sometimes, it is even easier to discard you history with `git reset --soft origin/master` and select individual changes with `git add -i` or `git add -p`.

Finally, the patch should be signed off. This is done by adding `Signed-off-by: Your Real Name <your@email.address>` at the end of the commit message. `git commit -s` does that for you, if configured properly. The `Signed-off-by` tag means that you publish the patch under the Buildroot license (i.e. GPL-2.0+, except for package patches, which have the upstream license), and that you are allowed to do so. See the Developer Certificate of Origin for details.

When adding new packages, you should submit every package in a separate patch. This patch should have the update to `package/Config.in`, the package `Config.in` file, the `.mk` file, the `.hash` file, any init script, and all package patches. If the package has many sub-options, these are sometimes better added as separate follow-up patches. The summary line should be something like `<packagename>: new package`. The body of the commit message can be empty for simple packages, or it can contain the description of the package (like the Config.in help text). If anything special has to be done to build the package, this should also be explained explicitly in the commit message body.

When you bump a package to a new version, you should also submit a separate patch for each package. Don't forget to update the `.hash` file, or add it if it doesn't exist yet. Also don't forget to check if the `_LICENSE` and `_LICENSE_FILES` are still valid. The summary line should be something like `<packagename>: bump to version <new version>`. If the new version only contains security updates compared to the existing one, the summary should be `<packagename>: security bump to version <new version>` and the commit message body should show the CVE numbers that are fixed. If some

package patches can be removed in the new version, it should be explained explicitly why they can be removed, preferably with the upstream commit ID. Also any other required changes should be explained explicitly, like configure options that no longer exist or are no longer needed.

If you are interested in getting notified of build failures and of further changes in the packages you added or modified, please add yourself to the DEVELOPERS file. This should be done in the same patch creating or modifying the package. See the DEVELOPERS file for more information.

Buildroot provides a handy tool to check for common coding style mistakes on files you created or modified, called `check-package` (see Section 18.25.2, "How to check the coding style" for more information).

22.5.2. Preparing a patch series

Starting from the changes committed in your local git view, *rebase* your development branch on top of the upstream tree before generating a patch set. To do so, run:

```
$ git fetch --all --tags
$ git rebase origin/master
```

Now check the coding style for the changes you committed:

```
$ utils/docker-run make check-package
```

Now, you are ready to generate then submit your patch set.

To generate it, run:

```
$ git format-patch -M -n -s -o outgoing origin/master
```

This will generate patch files in the `outgoing` subdirectory, automatically adding the `Signed-off-by` line.

Once patch files are generated, you can review/edit the commit message before submitting them, using your favorite text editor.

Buildroot provides a handy tool to know to whom your patches should be sent, called `get-developers` (see Chapter 23, *DEVELOPERS file and get-developers* for more information). This tool reads your patches and outputs the appropriate `git send-email` command to use:

```
$ ./utils/get-developers outgoing/*
```

Use the output of `get-developers` to send your patches:

```
$ git send-email --to buildroot@buildroot.org --cc bob --cc alice outgoing/*
```

Alternatively, `get-developers -e` can be used directly with the `--cc-cmd` argument to `git send-email` to automatically CC the affected developers:

```
$ git send-email --to buildroot@buildroot.org \
      --cc-cmd './utils/get-developers -e' origin/master
```

`git` can be configured to automatically do this out of the box with:

```
$ git config sendemail.to buildroot@buildroot.org
$ git config sendemail.ccCmd "$(pwd)/utils/get-developers -e"
```

And then just do:

```
$ git send-email origin/master
```

Note that `git` should be configured to use your mail account. To configure `git`, see `man git-send-email` or google it.

If you do not use `git send-email`, make sure posted **patches are not line-wrapped**, otherwise they cannot easily be applied. In such a case, fix your e-mail client, or better yet, learn to use `git send-email`.

22.5.3. Cover letter

If you want to present the whole patch set in a separate mail, add `--cover-letter` to the `git format-patch` command (see `man git-format-patch` for further information). This will generate a template for an introduction e-mail to your patch series.

A *cover letter* may be useful to introduce the changes you propose in the following cases:

- large number of commits in the series;
- deep impact of the changes in the rest of the project;
- RFC [4];

- whenever you feel it will help presenting your work, your choices, the review process, etc.

22.5.4. Patches for maintenance branches

When fixing bugs on a maintenance branch, bugs should be fixed on the master branch first. The commit log for such a patch may then contain a post-commit note specifying what branches are affected:

```
package/foo: fix stuff

Signed-off-by: Your Real Name <your@email.address>
---
Backport to: 2020.02.x, 2020.05.x
(2020.08.x not affected as the version was bumped)
```

Those changes will then be backported by a maintainer to the affected branches.

However, some bugs may apply only to a specific release, for example because it is using an older version of a package. In that case, patches should be based off the maintenance branch, and the patch subject prefix must include the maintenance branch name (for example "[PATCH 2020.02.x]"). This can be done with the `git format-patch` flag `--subject-prefix`:

```
$ git format-patch --subject-prefix "PATCH 2020.02.x" \
    -M -s -o outgoing origin/2020.02.x
```

Then send the patches with `git send-email`, as described above.

22.5.5. Patch revision changelog

When improvements are requested, the new revision of each commit should include a changelog of the modifications between each submission. Note that when your patch series is introduced by a cover letter, an overall changelog may be added to the cover letter in addition to the changelog in the individual commits. The best thing to rework a patch series is by interactive rebasing: `git rebase -i origin/master`. Consult the git manual for more information.

When added to the individual commits, this changelog is added when editing the commit message. Below the `Signed-off-by` section, add `---` and your changelog.

Although the changelog will be visible for the reviewers in the mail thread, as well as in patchwork, `git` will automatically ignores lines below `---` when the patch will be merged. This is the intended behavior: the changelog is not meant to be preserved forever in the `git` history of the project.

Hereafter the recommended layout:

```
Patch title: short explanation, max 72 chars

A paragraph that explains the problem, and how it manifests itself. If
the problem is complex, it is OK to add more paragraphs. All paragraphs
should be wrapped at 72 characters.

A paragraph that explains the root cause of the problem. Again, more
than one paragraph is OK.

Finally, one or more paragraphs that explain how the problem is solved.
Don't hesitate to explain complex solutions in detail.

Signed-off-by: John DOE <john.doe@example.net>

---
Changes v2 -> v3:
  - foo bar  (suggested by Jane)
  - bar buz

Changes v1 -> v2:
  - alpha bravo  (suggested by John)
  - charly delta
```

Any patch revision should include the version number. The version number is simply composed of the letter `v` followed by an `integer` greater or equal to two (i.e. "PATCH v2", "PATCH v3" …).

This can be easily handled with `git format-patch` by using the option `--subject-prefix`:

```
$ git format-patch --subject-prefix "PATCH v4" \
    -M -s -o outgoing origin/master
```

Since git version 1.8.1, you can also use `-v <n>` (where <n> is the version number):

```
$ git format-patch -v4 -M -s -o outgoing origin/master
```

When you provide a new version of a patch, please mark the old one as superseded in patchwork. You need to create an account on patchwork to be able to modify the status of your patches. Note that you can only change the status of patches you submitted yourself, which means the email address you register in patchwork should match the one you use for sending patches to the mailing list.

You can also add the `--in-reply-to <message-id>` option when submitting a patch to the mailing list. The id of the mail

to reply to can be found under the "Message Id" tag on patchwork. The advantage of **in-reply-to** is that patchwork will automatically mark the previous version of the patch as superseded.

22.6. Reporting issues/bugs or getting help

Before reporting any issue, please check in the mailing list archive whether someone has already reported and/or fixed a similar problem.

However you choose to report bugs or get help, either by opening a bug in the bug tracker or by sending a mail to the mailing list, there are a number of details to provide in order to help people reproduce and find a solution to the issue.

Try to think as if you were trying to help someone else; in that case, what would you need?

Here is a short list of details to provide in such case:

- host machine (OS/release)
- version of Buildroot
- target for which the build fails
- package(s) for which the build fails
- the command that fails and its output
- any information you think that may be relevant

Additionally, you should add the `.config` file (or if you know how, a `defconfig`; see Section 9.3, "Storing the Buildroot configuration").

If some of these details are too large, do not hesitate to use a pastebin service. Note that not all available pastebin services will preserve Unix-style line terminators when downloading raw pastes. Following pastebin services are known to work correctly: - https://gist.github.com/ - http://code.bulix.org/

22.7. Using the runtime tests framework

Buildroot includes a run-time testing framework built upon Python scripting and QEMU runtime execution. The goals of the framework are the following:

- build a well defined Buildroot configuration
- optionally, verify some properties of the build output
- optionally, boot the build results under Qemu, and verify that a given feature is working as expected

The entry point to use the runtime tests framework is the `support/testing/run-tests` tool, which has a series of options documented in the tool's help *-h* description. Some common options include setting the download folder, the output folder, keeping build output, and for multiple test cases, you can set the JLEVEL for each.

Here is an example walk through of running a test case.

- For a first step, let us see what all the test case options are. The test cases can be listed by executing `support/testing/run-tests -l`. These tests can all be run individually during test development from the console. Both one at a time and selectively as a group of a subset of tests.

```
$ support/testing/run-tests -l
List of tests
test_run (tests.utils.test_check_package.TestCheckPackage)
test_run (tests.toolchain.test_external.TestExternalToolchainBuildrootMusl) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainBuildrootuClibc) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainCCache) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainCtngMusl) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainLinaroArm) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainSourceryArmv4) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainSourceryArmv5) ... ok
test_run (tests.toolchain.test_external.TestExternalToolchainSourceryArmv7) ... ok
[snip]
test_run (tests.init.test_systemd.TestInitSystemSystemdRoFull) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRoIfupdown) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRoNetworkd) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRwFull) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRwIfupdown) ... ok
test_run (tests.init.test_systemd.TestInitSystemSystemdRwNetworkd) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRo) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRoNet) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRw) ... ok
test_run (tests.init.test_busybox.TestInitSystemBusyboxRwNet) ... ok

Ran 157 tests in 0.021s

OK
```

- Then, to run one test case:

```
$ support/testing/run-tests -d dl -o output_folder -k tests.init.test_busybox.TestInitSystemBusyboxRw
15:03:26 TestInitSystemBusyboxRw                  Starting
15:03:28 TestInitSystemBusyboxRw                  Building
15:08:18 TestInitSystemBusyboxRw                  Building done
15:08:27 TestInitSystemBusyboxRw                  Cleaning up
```

```
.
Ran 1 test in 301.140s

OK
```

The standard output indicates if the test is successful or not. By default, the output folder for the test is deleted automatically unless the option `-k` is passed to **keep** the output directory.

22.7.1. Creating a test case

Within the Buildroot repository, the testing framework is organized at the top level in `support/testing/` by folders of `conf`, `infra` and `tests`. All the test cases live under the `tests` folder and are organized in various folders representing the category of test.

The best way to get familiar with how to create a test case is to look at a few of the basic file system `support/testing/tests/fs/` and init `support/testing/tests/init/` test scripts. Those tests give good examples of a basic tests that include both checking the build results, and doing runtime tests. There are other more advanced cases that use things like nested `br2-external` folders to provide skeletons and additional packages.

Creating a basic test case involves:

- Defining a test class that inherits from `infra.basetest.BRTest`
- Defining the `config` member of the test class, to the Buildroot configuration to build for this test case. It can optionally rely on configuration snippets provided by the runtime test infrastructure: `infra.basetest.BASIC_TOOLCHAIN_CONFIG` to get a basic architecture/toolchain configuration, and `infra.basetest.MINIMAL_CONFIG` to not build any filesystem. The advantage of using `infra.basetest.BASIC_TOOLCHAIN_CONFIG` is that a matching Linux kernel image is provided, which allows to boot the resulting image in Qemu without having to build a Linux kernel image as part of the test case, therefore significant decreasing the build time required for the test case.
- Implementing a `def test_run(self):` function to implement the actual tests to run after the build has completed. They may be tests that verify the build output, by running command on the host using the `run_cmd_on_host()` helper function. Or they may boot the generated system in Qemu using the `Emulator` object available as `self.emulator` in the test case. For example `self.emulator.boot()` allows to boot the system in Qemu, `self.emulator.login()` allows to login, `self.emulator.run()` allows to run shell commands inside Qemu.

After creating the test script, add yourself to the `DEVELOPERS` file to be the maintainer of that test case.

22.7.2. Debugging a test case

When a test case runs, the `output_folder` will contain the following:

```
$ ls output_folder/
TestInitSystemBusyboxRw/
TestInitSystemBusyboxRw-build.log
TestInitSystemBusyboxRw-run.log
```

`TestInitSystemBusyboxRw/` is the Buildroot output directory, and it is preserved only if the `-k` option is passed.

`TestInitSystemBusyboxRw-build.log` is the log of the Buildroot build.

`TestInitSystemBusyboxRw-run.log` is the log of the Qemu boot and test. This file will only exist if the build was successful and the test case involves booting under Qemu.

If you want to manually run Qemu to do manual tests of the build result, the first few lines of `TestInitSystemBusyboxRw-run.log` contain the Qemu command line to use.

You can also make modifications to the current sources inside the `output_folder` (e.g. for debug purposes) and rerun the standard Buildroot make targets (in order to regenerate the complete image with the new modifications) and then rerun the test.

22.7.3. Runtime tests and Gitlab CI

All runtime tests are regularly executed by Buildroot Gitlab CI infrastructure, see .gitlab.yml and https://gitlab.com/buildroot.org/buildroot/-/jobs.

You can also use Gitlab CI to test your new test cases, or verify that existing tests continue to work after making changes in Buildroot.

In order to achieve this, you need to create a fork of the Buildroot project on Gitlab, and be able to push branches to your Buildroot fork on Gitlab.

The name of the branch that you push will determine if a Gitlab CI pipeline will be triggered or not, and for which test cases.

In the examples below, the <name> component of the branch name is an arbitrary string you choose.

- To trigger all run-test test case jobs, push a branch that ends with `-runtime-tests`:

```
$ git push gitlab HEAD:<name>-runtime-tests
```

- To trigger one or several test case jobs, push a branch that ends with the complete test case name (`tests.init.test_busybox.TestInitSystemBusyboxRo`) or with the name of a category of tests (`tests.init.test_busybox`):

```
$ git push gitlab HEAD:<name>-<test case name>
```

Example to run one test:

```
$ git push gitlab HEAD:foo-tests.init.test_busybox.TestInitSystemBusyboxRo
```

Examples to run several tests part of the same group:

```
$ git push gitlab HEAD:foo-tests.init.test_busybox
$ git push gitlab HEAD:foo-tests.init
```

---

[4] RFC: (Request for comments) change proposal

Chapter 23. DEVELOPERS file and get-developers

The main Buildroot directory contains a file named `DEVELOPERS` that lists the developers involved with various areas of Buildroot. Thanks to this file, the `get-developers` tool allows to:

- Calculate the list of developers to whom patches should be sent, by parsing the patches and matching the modified files with the relevant developers. See Section 22.5, "Submitting patches" for details.
- Find which developers are taking care of a given architecture or package, so that they can be notified when a build failure occurs on this architecture or package. This is done in interaction with Buildroot's autobuild infrastructure.

We ask developers adding new packages, new boards, or generally new functionality in Buildroot, to register themselves in the `DEVELOPERS` file. As an example, we expect a developer contributing a new package to include in his patch the appropriate modification to the `DEVELOPERS` file.

The `DEVELOPERS` file format is documented in detail inside the file itself.

The `get-developers` tool, located in `utils/` allows to use the `DEVELOPERS` file for various tasks:

- When passing one or several patches as command line argument, `get-developers` will return the appropriate `git send-email` command. If the `-e` option is passed, only the email addresses are printed in a format suitable for `git send-email --cc-cmd`.
- When using the `-a <arch>` command line option, `get-developers` will return the list of developers in charge of the given architecture.
- When using the `-p <package>` command line option, `get-developers` will return the list of developers in charge of the given package.
- When using the `-c` command line option, `get-developers` will look at all files under version control in the Buildroot repository, and list the ones that are not handled by any developer. The purpose of this option is to help completing the `DEVELOPERS` file.
- When using the `-v` command line option, it validates the integrity of the DEVELOPERS file and will note WARNINGS for items that don't match.

Chapter 24. Release Engineering

24.1. Releases

The Buildroot project makes quarterly releases with monthly bugfix releases. The first release of each year is a long term support release, LTS.

- Quarterly releases: 2020.02, 2020.05, 2020.08, and 2020.11
- Bugfix releases: 2020.02.1, 2020.02.2, ⋯
- LTS releases: 2020.02, 2021.02, ⋯

Releases are supported until the first bugfix release of the next release, e.g., 2020.05.x is EOL when 2020.08.1 is released.

LTS releases are supported until the first bugfix release of the next LTS, e.g., 2020.02.x is supported until 2021.02.1 is released.

24.2. Development

Each release cycle consist of two months of development on the `master` branch and one month stabilization before the release is made. During this phase no new features are added to `master`, only bugfixes.

The stabilization phase starts with tagging `-rc1`, and every week until the release, another release candidate is tagged.

To handle new features and version bumps during the stabilization phase, a `next` branch may be created for these features.

Once the current release has been made, the `next` branch is merged into `master` and the development cycle for the next release continues there.

Part IV. Appendix

Chapter 25. Makedev syntax documentation

The makedev syntax is used in several places in Buildroot to define changes to be made for permissions, or which device files to create and how to create them, in order to avoid calls to mknod.

This syntax is derived from the makedev utility, and more complete documentation can be found in the `package/makedevs/README` file.

It takes the form of a space separated list of fields, one file per line; the fields are:

name	type	mode	uid	gid	major	minor	start	inc	count

There are a few non-trivial blocks:

- `name` is the path to the file you want to create/modify
- `type` is the type of the file, being one of:
  - f: a regular file
  - d: a directory
  - r: a directory recursively
  - c: a character device file
  - b: a block device file
  - p: a named pipe
- `mode` are the usual permissions settings (only numerical values are allowed)
- `uid` and `gid` are the UID and GID to set on this file; can be either numerical values or actual names
- `major` and `minor` are here for device files, set to `-` for other files
- `start`, `inc` and `count` are for when you want to create a batch of files, and can be reduced to a loop, beginning at `start`, incrementing its counter by `inc` until it reaches `count`

Let’s say you want to change the permissions of a given file; using this syntax, you will need to write:

```
/usr/bin/foo f 755 0 0 - - - - -
/usr/bin/bar f 755 root root - - - - -
/data/buz f 644 buz-user buz-group - - - - -
```

Alternatively, if you want to change owner/permission of a directory recursively, you can write (to set UID to foo, GID to bar and access rights to rwxr-x--- for the directory /usr/share/myapp and all files and directories below it):

```
/usr/share/myapp r 750 foo bar - - - - -
```

On the other hand, if you want to create the device file `/dev/hda` and the corresponding 15 files for the partitions, you will need for `/dev/hda`:

```
/dev/hda b 640 root root 3 0 0 0 -
```

and then for device files corresponding to the partitions of `/dev/hda`, `/dev/hdaX`, `X` ranging from 1 to 15:

```
/dev/hda b 640 root root 3 1 1 1 15
```

Extended attributes are supported if `BR2_ROOTFS_DEVICE_TABLE_SUPPORTS_EXTENDED_ATTRIBUTES` is enabled. This is done by adding a line starting with `|xattr` after the line describing the file. Right now, only capability is supported as extended attribute.

|xattr	capability

- `|xattr` is a "flag" that indicate an extended attribute
- `capability` is a capability to add to the previous file

If you want to add the capability cap_sys_admin to the binary foo, you will write :

```
/usr/bin/foo f 755 root root - - - - -
|xattr cap_sys_admin+eip
```

You can add several capabilities to a file by using several `|xattr` lines. If you want to add the capability cap_sys_admin and cap_net_admin to the binary foo, you will write :

```
/usr/bin/foo f 755 root root - - - - -
|xattr cap_sys_admin+eip
|xattr cap_net_admin+eip
```

Chapter 26. Makeusers syntax documentation

The syntax to create users is inspired by the makedev syntax, above, but is specific to Buildroot.

The syntax for adding a user is a space-separated list of fields, one user per line; the fields are:

username	uid	group	gid	password	home	shell	groups	comment

Where:

- `username` is the desired user name (aka login name) for the user. It can not be `root`, and must be unique. If set to `-`, then just a group will be created.
- `uid` is the desired UID for the user. It must be unique, and not `0`. If set to `-1` or `-2`, then a unique UID will be computed by Buildroot, with `-1` denoting a system UID from [100…999] and `-2` denoting a user UID from [1000…1999].
- `group` is the desired name for the user’s main group. It can not be `root`. If the group does not exist, it will be created.
- `gid` is the desired GID for the user’s main group. It must be unique, and not `0`. If set to `-1` or `-2`, and the group does not already exist, then a unique GID will be computed by Buildroot, with `-1` denoting a system GID from [100…999] and `-2` denoting a user GID from [1000…1999].
- `password` is the crypt(3)-encoded password. If prefixed with `!`, then login is disabled. If prefixed with `=`, then it is interpreted as clear-text, and will be crypt-encoded (using MD5). If prefixed with `!=`, then the password will be crypt-encoded (using MD5) and login will be disabled. If set to `*`, then login is not allowed. If set to `-`, then no password value will be set.
- `home` is the desired home directory for the user. If set to -, no home directory will be created, and the user’s home will be `/`. Explicitly setting `home` to `/` is not allowed.
- `shell` is the desired shell for the user. If set to `-`, then `/bin/false` is set as the user’s shell.
- `groups` is the comma-separated list of additional groups the user should be part of. If set to `-`, then the user will be a member of no additional group. Missing groups will be created with an arbitrary `gid`.
- `comment` (aka GECOS field) is an almost-free-form text.

There are a few restrictions on the content of each field:

- except for `comment`, all fields are mandatory.
- except for `comment`, fields may not contain spaces.
- no field may contain a colon (`:`).

If `home` is not `-`, then the home directory, and all files below, will belong to the user and its main group.

Examples:

```
foo -1 bar -1 !=blabla /home/foo /bin/sh alpha,bravo Foo user
```

This will create this user:

- `username` (aka login name) is: `foo`
- `uid` is computed by Buildroot
- main `group` is: `bar`
- main group `gid` is computed by Buildroot
- clear-text `password` is: `blabla`, will be crypt(3)-encoded, and login is disabled.
- `home` is: `/home/foo`
- `shell` is: `/bin/sh`
- `foo` is also a member of `groups`: `alpha` and `bravo`
- `comment` is: `Foo user`

```
test 8000 wheel -1 = - /bin/sh - Test user
```

This will create this user:

- `username` (aka login name) is: `test`
- `uid` is : `8000`

- main `group` is: `wheel`
- main group `gid` is computed by Buildroot, and will use the value defined in the rootfs skeleton
- `password` is empty (aka no password).
- `home` is `/` but will not belong to `test`
- `shell` is: `/bin/sh`
- `test` is not a member of any additional `groups`
- `comment` is: `Test user`

26.1. Caveat with automatic UIDs and GIDs

When updating buildroot or when packages are added or removed to/from the configuration, it is possible that the automatic UIDs and GIDs are changed. This can be a problem if persistent files were created with that user or group: after upgrade, they will suddenly have a different owner.

Therefore, it is advisable to perpetuate the automatic IDs. This can be done by adding a users table with the generated IDs. It is only needed to do this for UIDs that actually create persistent files, e.g. database.

Chapter 27. Migrating from older Buildroot versions

Some versions have introduced backward incompatibilities. This section explains those incompatibilities, and for each explains what to do to complete the migration.

27.1. General approach

To migrate from an older Buildroot version, take the following steps.

1. For all your configurations, do a build in the old Buildroot environment. Run `make graph-size`. Save `graphs/file-size-stats.csv` in a different location. Run `make clean` to remove the rest.
2. Review the specific migration notes below and make the required adaptations to external packages and custom build scripts.
3. Update Buildroot.
4. Run `make menuconfig` starting from the existing `.config`.
5. If anything is enabled in the Legacy menu, check its help text, unselect it, and save the configuration.
6. For more details, review the git commit messages for the packages that you need. Change into the `packages` directory and run `git log <old version>.. — <your packages>`.
7. Build in the new Buildroot environment.
8. Fix build issues in external packages (usually due to updated dependencies).
9. Run `make graph-size`.
10. Compare the new `file-size-stats.csv` with the original one, to check if no required files have disappeared and if no new big unneeded files have appeared.
11. For configuration (and other) files in a custom overlay that overwrite files created by Buildroot, check if there are changes in the Buildroot-generated file that need to be propagated to your custom file.

27.2. Migrating to 2016.11

Before Buildroot 2016.11, it was possible to use only one br2-external tree at once. With Buildroot 2016.11 came the possibility to use more than one simultaneously (for details, see Section 9.2, "Keeping customizations outside of Buildroot").

This however means that older br2-external trees are not usable as-is. A minor change has to be made: adding a name to your br2-external tree.

This can be done very easily in just a few steps:

- First, create a new file named `external.desc`, at the root of your br2-external tree, with a single line defining the name of your br2-external tree:

  ```
  $ echo 'name: NAME_OF_YOUR_TREE' >external.desc
  ```

  Note. Be careful when choosing a name: It has to be unique and be made with only ASCII characters from the set `[A-Za-z0-9_]`.

- Then, change every occurence of `BR2_EXTERNAL` in your br2-external tree with the new variable:

  ```
  $ find . -type f | xargs sed -i 's/BR2_EXTERNAL/BR2_EXTERNAL_NAME_OF_YOUR_TREE_PATH/g'
  ```

Now, your br2-external tree can be used with Buildroot 2016.11 onward.

Note: This change makes your br2-external tree incompatible with Buildroot before 2016.11.

27.3. Migrating to 2017.08

Before Buildroot 2017.08, host packages were installed in `$(HOST_DIR)/usr` (with e.g. the autotools' `--prefix=$(HOST_DIR)/usr`). With Buildroot 2017.08, they are now installed directly in `$(HOST_DIR)`.

Whenever a package installs an executable that is linked with a library in `$(HOST_DIR)/lib`, it must have an RPATH pointing to that directory.

An RPATH pointing to `$(HOST_DIR)/usr/lib` is no longer accepted.