The Buildroot user manual

Contents

Buildroot usage and documentation by Thomas Petazzoni. Contributions from Karsten Kruse, Ned Ludd, Martin Herren and others.

Buildroot 2013.08 manual generated on 2013-08-31 00:35:03 CEST from git revision 53d8d9c

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

Starting up

2.1 System requirements

Buildroot is designed to run on Linux systems.

Buildroot needs some software to be already installed on the host system; here are the lists of the mandatory and optional packages (package names may vary between distributions).

Take care to *install both runtime and development data*, especially for the libraries that may be packaged in 2 distinct packages.

2.1.1 Mandatory packages

- Build tools:
  - `which`
  - `sed`
  - `make` (version 3.81 or any later)
  - `binutils`
  - `build-essential` (only for Debian based systems)
  - `gcc` (version 2.95 or any later)
  - `g++` (version 2.95 or any later)
  - `bash`
  - `patch`
  - `gzip`
  - `bzip2`
  - `perl`
  - `tar`
  - `cpio`
  - `python` (version 2.6 or 2.7)
  - `unzip`
  - `rsync`
- Source fetching tools:
  - `wget`

2.1.2 Optional packages

- Source fetching tools:

  In the official tree, most of the package sources are retrieved using `wget`; a few are only available through their `git`, `mercurial`, or `svn` repository.

  All other source fetching methods are implemented and may be used in a development context (further details: refer to Section 6.4).

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

- Configuration interface dependencies (requires development libraries):

  - `ncurses5` to use the *menuconfig* interface
  - `qt4` to use the *xconfig* interface
  - `glib2`, `gtk2` and `glade2` to use the *gconfig* interface

- 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`

2.2 Getting Buildroot

Buildroot releases are made approximately every 3 months. Direct Git access and daily snapshots are also available, if you want more bleeding edge.

Releases are available at http://buildroot.net/downloads/.

The latest snapshot is always available at http://buildroot.net/downloads/snapshots/buildroot-snapshot.tar.bz2, and previous snapshots are also available at http://buildroot.net/downloads/snapshots/.

To download Buildroot using Git, you can simply follow the rules described on the "Accessing Git" page (http://buildroot.net/-git.html) of the Buildroot website (http://buildroot.net). For the impatient, here's a quick recipe:

```
$ git clone git://git.buildroot.net/buildroot
```

2.3 Using Buildroot

Buildroot has a nice configuration tool similar to the one you can find in the Linux kernel or in Busybox. Note that you can **and should build everything as a normal user**. There is no need to be root to configure and use Buildroot. The first step is to run the configuration assistant:

```
$ make menuconfig
```

to run the curses-based configurator, or

```
$ make xconfig
```

or

```
$ make gconfig
```

to run the Qt or GTK-based configurators.

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. Check Section 2.1 to know what Buildroot needs, and specifically the optional requirements Section 2.1.2 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.

Once everything is configured, the configuration tool generates a `.config` file that contains the description of your configuration. It will be used by the Makefiles to do what's needed.

Let's go:

```
$ make
```

You **should never** use `make -jN` with Buildroot: it does not support *top-level parallel make*. Instead, use the `BR2_JLEVEL` option to tell Buildroot to run each package compilation with `make -jN`.

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

- download source files (as required);
- configure, build and install the cross-compiling toolchain using the appropriate toolchain backend, or simply import an external toolchain;
- build/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.
- `build/` where all the components except for the cross-compilation toolchain are built (this includes tools needed to run Buildroot on the host and packages compiled for the target). The `build/` directory contains one subdirectory for each of these components.
- `staging/` which contains a hierarchy similar to a root filesystem hierarchy. This directory contains the installation of the cross-compilation toolchain and all the userspace packages selected for the target. 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.
- `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.

- `host/` contains the installation of tools compiled for the host that are needed for the proper execution of Buildroot, including the cross-compilation toolchain.
- `toolchain/` contains the build directories for the various components of the cross-compilation toolchain.

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

More details about the "make" command usage are given in Section 3.2.

Chapter 3

Working with Buildroot

This section explains how you can customize Buildroot to fit your needs.

3.1 Details on Buildroot configuration

All the configuration options in `make *config` have a help text providing details about the option. However, a number of topics require additional details that cannot easily be covered in the help text and are there covered in the following sections.

3.1.1 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).

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" [1].

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 different solutions to build, or use existing cross-compilation toolchains:

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

The choice between these three 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.

[1] 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)

3.1.1.1 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 is the historical backend of Buildroot, and has been limited for a long time to the usage of the uClibc C library. Support for the *eglibc* C library has been added in 2013 and is at this point considered experimental. See the *External toolchain backend* and *Crosstool-NG toolchain backend* for other solutions to use *glibc* or *eglibc*.

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 and the configuration of the uClibc C library (if uClibc is selected). The default options are usually fine. However, if you really need to specifically customize the configuration of your uClibc C library, you can pass a specific configuration file here. Or alternatively, you can run the `make uclibc-menuconfig` command to get access to uClibc's configuration interface. Note 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.
- Change the version of the GCC compiler and binutils.
- Select a number of toolchain options (uClibc only): whether the toolchain should have largefile support (i.e support for files larger than 2 GB on 32 bits systems), IPv6 support, RPC support (used mainly for NFS), wide-char support, locale support (for internationalization), C++ support, 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.

It is worth noting that whenever one of those options is modified, then the entire toolchain and system must be rebuilt. See Section 3.5.1.

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*.

3.1.1.2 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, x86-64, PowerPC, MIPS and SuperH, Blackfin toolchains from ADI, Xilinx toolchains for Microblaze, etc.) 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, Linaro, Blackfin and Xilinx 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. 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 largefile, IPv6, 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 from the ELDK of Denx, for two reasons:

- The ELDK does not contain a pure toolchain (i.e just the compiler, binutils, the C and C++ libraries), but a toolchain that comes 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.
- The ELDK toolchains have a completely non-standard custom mechanism to handle multiple library variants. Instead of using the standard GCC *multilib* mechanism, the ARM ELDK uses different symbolic links to the compiler to differentiate between library variants (for ARM soft-float and ARM VFP), and the PowerPC ELDK compiler uses a `CROSS_COMPILE` environment variable. This non-standard behaviour makes it difficult to support ELDK in 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 definitely to build it with crosstool-NG. We recommend to build the toolchain separately from Buildroot, and then *import* it in Buildroot using the external toolchain backend.

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.
- Not limited to uClibc: glibc and eglibc toolchains are supported.

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 Crosstool-NG.

3.1.1.3 Crosstool-NG toolchain backend

The *Crosstool-NG toolchain backend* integrates the Crosstool-NG project with Buildroot. Crosstool-NG is a highly-configurable, versatile and well-maintained tool to build cross-compilation toolchains.

If you select the `Crosstool-NG toolchain` option in `Toolchain Type`, then you will be offered to:

- Choose which C library you want to use. Crosstool-NG supports the three most important C libraries used in Linux systems: glibc, eglibc and uClibc
- Choose a custom Crosstool-NG configuration file. Buildroot has its own default configuration file (one per C library choice), but you can provide your own. Another option is to run `make ctng-menuconfig` to get access to the Crosstool-NG configuration interface. However, note that all Buildroot packages have only been tested with the default Crosstool-NG configurations.
- Choose a number of toolchain options (rather limited if glibc or eglibc are used, or numerous if uClibc is used)

When you will start the Buildroot build process, Buildroot will download and install the Crosstool-NG tool, build and install its required dependencies, and then run Crosstool-NG with the provided configuration.

Advantages of this backend:

- Not limited to uClibc: glibc and eglibc are supported.
- Vast possibilities of toolchain configuration.

Drawbacks of this backend:

- Crosstool-NG is not perfectly integrated with Buildroot. For example, Crosstool-NG has its own download infrastructure, not integrated with the one in Buildroot (for example a Buildroot `make source` will not download all the source code tarballs needed by Crosstool-NG).
- The toolchain is completely rebuilt from scratch if you do a `make clean`.

3.1.2 /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 accross 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 Section 11.1.

- 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 accross 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 responsability to enable those two options (if you fail to do so, your Buildroot system will not boot).
- The third solution is **Dynamic using 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 udev**. This method also relies on the *devtmpfs* virtual filesystem detailed above, but adds the `udev` userspace daemon on top of it. `udev` 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 and is sometimes mandatory for some system components (systemd for example). `udev` is the mechanism used in most desktop Linux distributions. For more details about `udev`, see http://en.wikipedia.org/wiki/Udev.

The Buildroot developers recommandation 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 mdev** is usually a good solution.

3.1.3 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`, `glib` 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.

3.2 *make* tips

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

Configuration searches: The `make *config` commands 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.

Display all commands executed by make:

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

Display all available targets:

```
$ make help
```

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

- `linux-menuconfig` and `linux-savedefconfig` only work when `linux` is enabled;
- `uclibc-menuconfig` is only available when the Buildroot internal toolchain backend is used;
- `ctng-menuconfig` is only available when the crosstool-NG backend is used;
- `barebox-menuconfig` and `barebox-savedefconfig` only work when the `barebox` bootloader is enabled.

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

- `asciidoc` is required to build the documentation (see: Section 2.1.2).
- There is a known issue that you can't build it under Debian Squeeze.

Reseting 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 5.2.2.

3.3 Customization

3.3.1 Customizing the generated target filesystem

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

- Customize the target filesystem directly and rebuild the image. The target filesystem is available under `output/target/`. You can simply make your changes here and run make afterwards - this will rebuild the target filesystem image. This method allows you to do anything to the target filesystem, but if you decide to completely rebuild your toolchain and tools, these changes will be lost. This solution is therefore only useful for quick tests only: *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` by using one of the following methods.
- Create a filesystem overlay: a tree of files that are copied directly over the target filesystem after it has been built. Set `BR2_ROOTFS_OVERLAY` to the top of the tree. `.git`, `.svn`, `.hg` directories, `.empty` files and files ending with ~ are excluded. *Among these first 3 methods, this one should be preferred.*
- In the Buildroot configuration, you can specify the paths to one or more **post-build scripts**. These scripts are called in the given order, *after* Buildroot builds all the selected software, but *before* the rootfs images are assembled. The `BR2_ROOTFS_POST_BUILD_SCRIPT` allows you to specify the location of your post-build scripts. This option can be found in the `System configuration` menu. The destination root filesystem folder is given as the first argument to these scripts, and these scripts can then be used to 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. You may also use these variables in your post-build script:
  - `BUILDROOT_CONFIG`: the path to the Buildroot .config file
  - `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`: see Section 6.2.4.2
  - `BINARIES_DIR`: the place where all binary files (aka images) are stored
  - `BASE_DIR`: the base output directory
- Create your own *target skeleton*. You can start with the default skeleton available under `system/skeleton` and then customize it to suit your needs. The `BR2_ROOTFS_SKELETON_CUSTOM` and `BR2_ROOTFS_SKELETON_CUSTOM_PATH` will allow you to specify the location of your custom skeleton. These options can be found in the `System configuration` menu. At build time, the contents of the skeleton are copied to output/target before any package installation. Note that this method is **not recommended**, as it duplicates the entire skeleton, which prevents from taking advantage of the fixes or improvements brought to the default Buildroot skeleton. The recommended method is to use the *post-build scripts* mechanism described in the previous item.

Note also that you can use the **post-image scripts** if you want to perform some specific actions *after* all filesystem images have been created (for example 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), you can specify a space-separated list of scripts in the `BR2_ROOTFS_POST_IMAGE_SCRIPT` configuration option. This option can be found in the `System configuration` menu as well.

Each of those scripts will be called with the path to the `images` output directory as first argument, and will be executed with the main Buildroot source directory as the current directory. Those 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 *post-image scripts* will require special handling (usage of fakeroot or sudo), which is left to the script developer.

Just like for the *post-build scripts* mentioned above, you also have access to the following environment variables from your *post-image scripts*: `BUILDROOT_CONFIG`, `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`, `BINARIES_DIR` and `BASE_DIR`.

Additionally, each of the `BR2_ROOTFS_POST_BUILD_SCRIPT` and `BR2_ROOTFS_POST_IMAGE_SCRIPT` scripts will be passed the arguments specified in `BR2_ROOTFS_POST_SCRIPT_ARGS` (if that is not empty). 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.

3.3.2 Customizing the Busybox configuration

Busybox is very configurable, and you may want to customize it. You can follow these simple steps to do so. This method isn't optimal, but it's simple, and it works:

- Do an initial compilation of Buildroot, with busybox, without trying to customize it.
- Invoke `make busybox-menuconfig`. The nice configuration tool appears, and you can customize everything.
- Run the compilation of Buildroot again.

Otherwise, you can simply change the `package/busybox/busybox-<version>.config` file, if you know the options you want to change, without using the configuration tool.

If you want to use an existing config file for busybox, then see Section 3.5.5.

3.3.3 Customizing the uClibc configuration

Just like BusyBox Section 3.3.2, uClibc offers a lot of configuration options. They allow you to select various functionalities depending on your needs and limitations.

The easiest way to modify the configuration of uClibc is to follow these steps:

- Do an initial compilation of Buildroot without trying to customize uClibc.
- Invoke `make uclibc-menuconfig`. The nice configuration assistant, similar to the one used in the Linux kernel or Buildroot, appears. Make your configuration changes as appropriate.
- Copy the `$(O)/build/uClibc-VERSION/.config` file to a different place (e.g. `board/MANUFACTURER/BOARDNAME/uClibc.config`) and adjust the uClibc configuration file option `BR2_UCLIBC_CONFIG` to refer to this configuration instead of the default one.
- Run the compilation of Buildroot again.

Otherwise, you can simply change `package/uclibc/uClibc-VERSION.config`, without running the configuration assistant.

If you want to use an existing config file for uClibc, then see Section 3.5.5.

3.3.4 Customizing the Linux kernel configuration

The Linux kernel configuration can be customized just like BusyBox Section 3.3.2 and uClibc Section 3.3.3 using `make linux-menuconfig`. Make sure you have enabled the kernel build in `make menuconfig` first. Once done, run `make` to (re)build everything.

If you want to use an existing config file for Linux, then see Section 3.5.5.

3.3.5 Customizing the toolchain

There are three distinct types of toolchain backend supported in Buildroot, available under the menu `Toolchain`, invoking `make menuconfig`.

3.3.5.1 Using the external toolchain backend

There is no way of tuning an external toolchain since Buildroot does not generate it.

It also requires to set the Buildroot settings according to the toolchain ones (see Section 3.1.1.2).

3.3.5.2 Using the internal Buildroot toolchain backend

The internal Buildroot toolchain backend **only** allows to generate **uClibc-based toolchains**.

However, it allows to tune major settings, such as:

- Linux headers version;
- uClibc configuration (see uClibc Section 3.3.3);
- Binutils, GCC, Gdb and toolchain options.

These settings are available after selecting the `Buildroot toolchain` type in the menu `Toolchain`.

3.3.5.3 Using the Crosstool-NG backend

The crosstool-NG toolchain backend enables a rather limited set of settings under the Buildroot `Toolchain` menu:

- The crosstool-NG configuration file
- Gdb and some toolchain options

Then, the toolchain can be fine-tuned by invoking `make ctng-menuconfig`.

3.4 Storing the configuration

When you have a buildroot configuration that you are satisfied with and you want to share it with others, put it under revision control or move on to a different buildroot project, you need to store the configuration so it can be rebuilt later. The configuration that needs to be stored consists of the buildroot configuration, the configuration files for packages that you use (kernel, busybox, uClibc, . . .), and your rootfs modifications.

3.4.1 Basics for storing the configuration

3.4.1.1 Buildroot configuration

For storing the buildroot configuration itself, buildroot offers the following 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, or call make with `make savedefconfig BR2_DEFCONFIG=<path-to-defconfig>`. The usual place is `configs/<boardname>_defconfig`. The configuration can then be rebuilt 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>`.

3.4.1.2 Other package configuration

The configuration files for busybox, the linux kernel, barebox, uClibc and crosstool-NG should be stored as well if changed. For each of these, a buildroot configuration option exists to point to an input configuration file, e.g. `BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE`. To save their configuration, set those configuration options to a path outside your output directory, e.g. `board/<manufacturer>/<boardname>/linux.config`. Then, copy the configuration files to that path.

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.
- For crosstool-NG and at91bootstrap3, no helper exists so you have to copy the config file manually to BR2_TOOLCHAIN_CTNG_CONFIG, resp. BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE.

3.4.2 Creating your own board support

Creating your own board support in Buildroot allows users of a particular hardware platform to easily build a system that is known to work.

To do so, you need to create a normal Buildroot configuration that builds a basic system for the hardware: 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.

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.

3.4.3 Step-by-step instructions for storing configuration

To store the configuration for a specific product, device or application, it is advisable to use the same conventions as for the board support: put the buildroot defconfig in the configs directory, and any other files in a subdirectory of the boards directory. This section gives step-by-step instructions about how to do that. Of course, you can skip the steps that are not relevant for your use case.

1. make menuconfig to configure toolchain, packages and kernel.
2. make linux-menuconfig to update the kernel config, similar for other configuration.
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_TOOLCHAIN_CTNG_CONFIG
   - BR2_UCLIBC_CONFIG
   - BR2_TARGET_AT91BOOTSTRAP3_CUSTOM_CONFIG_FILE

   - BR2_TARGET_BAREBOX_CUSTOM_CONFIG_FILE

5. Write the configuration files:
   - make linux-update-defconfig
   - make busybox-update-config
   - cp <output>/build/build-toolchain/.config board/<manufacturer>/<boardname>/ctng.config
   - make uclibc-update-config
   - cp <output>/build/at91bootstrap3-*/.config board/<manufacturer>/<boardname>/at91bootstrap3.config
   - make barebox-update-defconfig
6. Create board/<manufacturer>/<boardname>/fs-overlay/ and fill it with additional files you need on your rootfs, e.g. board/<manufacturer>/<boardname>/fs-overlay/etc/inittab. Set BR2_ROOTFS_OVERLAY to board/<manufacturer>/<boardname>/fs-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. make savedefconfig to save the buildroot configuration.
10. cp defconfig configs/<boardname>_defconfig
11. To add patches to the linux build, set BR2_LINUX_KERNEL_PATCH to board/<manufacturer>/<boardname>/patches/linux/ and add your patches in that directory. Each patch should be called linux-<num>-<description>.patch. Similar for U-Boot, barebox, at91bootstrap and at91bootstrap3.
12. If you need modifications to other packages, or if you need to add packages, do that directly in the packages/ directory, following the instructions in Section 6.2.

3.4.4 Customizing packages

It is sometimes useful to apply *extra* patches to packages - over and above 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 file option can be used to specify a directory containing global package patches.

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

First, the default Buildroot patch set for the package is applied.

If the directory $(BR2_GLOBAL_PATCH_DIR)/<packagename>/<packageversion> exists, then all *.patch files in the directory will be applied.

Otherwise, if the directory $(BR2_GLOBAL_PATCH_DIR)/<packagename> exists, then all *.patch files in the directory will be applied.

3.5 Daily use

3.5.1 Understanding when a full rebuild is necessary

A full rebuild is achieved by running:

```
$ make clean all
```

In some cases, a full rebuild is mandatory:

- each time the toolchain properties are changed, this includes:
  - after changing any toolchain option under the *Toolchain* menu (if the internal Buildroot backend is used);
  - after running `make ctng-menuconfig` (if the crosstool-NG backend is used);
  - after running `make uclibc-menuconfig`.
- after removing some libraries from the package selection.

In some cases, a full rebuild is recommended:

- after adding some libraries to the package selection (otherwise, packages that can be optionally linked against those libraries won't be rebuilt, so they won't support those new available features).

In other cases, it is up to you to decide if you should run a full rebuild, but you should know what is impacted and understand what you are doing anyway.

3.5.2 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.

For convenience, the special make targets <package>-reconfigure and <package>-rebuild repeat the configure resp. build steps.

However, if you don't want to rebuild the package completely from scratch, a better understanding of the Buildroot internals is needed. Internally, to keep track of which steps have been done and which steps remain to be done, Buildroot maintains stamp files (empty files that just tell whether this or that action has been done):

- `output/build/<package>-<version>/.stamp_configured`. If removed, Buildroot will trigger the recompilation of the package from the configuration step (execution of `./configure`).
- `output/build/<package>-<version>/.stamp_built`. If removed, Buildroot will trigger the recompilation of the package from the compilation step (execution of `make`).

Note: toolchain packages use custom makefiles. Their stamp files are named differently.

Further details about package special make targets are explained in Section 5.2.4.

3.5.3 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*, *xconfig* or *gconfig*), then issue:

```
$ make source
```

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

3.5.4 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
```

Or:

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

All the output files will be located under /tmp/build.

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>
```

3.5.5 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.
- BUILDROOT_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; this is the recommended way of setting it.

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
```

3.6 Integration with Eclipse

While a part of the embedded Linux developers like classical text editors like Vim or Emacs, and command-line based interfaces, a number of other embedded Linux developers like richer graphical interfaces to do their development work. Eclipse being one of the most popular Integrated Development Environment, Buildroot integrates with Eclipse in order to ease the development work of Eclipse users.

Our integration with Eclipse simplifies the compilation, remote execution and remote debugging of applications and libraries that are built on top of a Buildroot system. It does not integrate the Buildroot configuration and build processes themselves with Eclipse. Therefore, the typical usage model of our Eclipse integration would be:

- Configure your Buildroot system with `make menuconfig`, `make xconfig` or any other configuration interface provided with Buildroot.
- Build your Buildroot system by running `make`.
- Start Eclipse to develop, execute and debug your own custom applications and libraries, that will rely on the libraries built and installed by Buildroot.

The Buildroot Eclipse integration installation process and usage is described in detail at https://github.com/mbats/eclipse-buildroot-bundle/wiki.

3.7 Hacking Buildroot

If Buildroot does not yet fit all your requirements, you may be interested in hacking it to add:

- new packages: refer to the Developer guide Section 6.2
- new board support: refer to the Developer guide Section 3.4.2.

Chapter 4

Frequently Asked Questions & Troubleshooting

4.1 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, `System configuration`, and modify `Port to run a getty (login prompt) on` and `Baudrate to use` as appropriate. This will automatically tune the `/etc/inittab` file of the generated system so that a shell starts on the correct serial port.

4.2 module-init-tools fails to build with *cannot find -lc*

If the build of `module-init-tools` for the host fails with:

```
/usr/bin/ld: cannot find -lc
```

then probably you are running a Fedora (or similar) distribution, and you should install the `glibc-static` package. This is because the `module-init-tools` build process wants to link statically against the C library.

4.3 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 does not make much sense.

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
- emdebian
- Fedora
- openSUSE ARM
- Arch Linux ARM
- ...

4.4 Why are there no development files on the target?

Since there is no compiler available on the target (see Section 4.3), 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.

4.5 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 4.3).

4.6 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 3.2).

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 3.2 for more explanations).

4.7 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.

Chapter 5

Going further in Buildroot's innards

5.1 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`, `stamps`, etc. in the output directory (`output/` by default, another value can be specified using `O=`)
- Generate all the targets listed in the `BASE_TARGETS` variable. 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.

5.2 Advanced usage

5.2.1 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/usr/bin/` to your PATH environment variable and then to use `ARCH-linux-gcc`, `ARCH-linux-objdump`, `ARCH-linux-ld`, etc.

It is possible to relocate the toolchain - but then `--sysroot` must be passed every time the compiler is called to tell where the libraries and header files are.

It is also possible to generate the Buildroot toolchain in a directory other than `output/host` by using the `Build options` → `Host dir` option. This could be useful if the toolchain must be shared with other users.

5.2.2 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 `$HOME/.buildroot-ccache`. It is stored 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`.

5.2.3 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 `BUILDROOT_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 BUILDROOT_DL_DIR <shared download location>
```

The download location can also be set in the `.config` file, with the `BR2_DL_DIR` option. This value is overridden by the `BUILDROOT_DL_DIR` environment variable.

5.2.4 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 dependencies required to build the package
`clean`	Run the clean command of the package, also uninstall the package from both the target and the staging directory; *note that this is not implemented for all packages*
`dirclean`	Remove the whole package build directory
`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

Chapter 6

Developer Guidelines

6.1 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.

6.1.1 `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.

	  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 `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.

6.1.2 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_OPT += --without-python-support
```

  It is also possible to align the = signs:

```
LIBFOO_VERSION   = 1.0
LIBFOO_SOURCE    = foo-$(LIBFOO_VERSION).tar.gz
LIBFOO_CONF_OPT += --without-python-support
```

- 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_PYTHON),y)
LIBFOO_CONF_OPT += --with-python-support
LIBFOO_DEPENDENCIES += python
else
LIBFOO_CONF_OPT += --without-python-support
endif
```

    NO:

```
LIBFOO_CONF_OPT += --with$(if $(BR2_PACKAGE_PYTHON),,out)-python-support
LIBFOO_DEPENDENCIES += $(if $(BR2_PACKAGE_PYTHON),python,)
```

  – 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
```

6.1.3 The documentation

The documentation uses the asciidoc format.

For further details about the asciidoc syntax, refer to http://www.methods.co.nz/asciidoc/userguide.html.

6.2 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.

6.2.1 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: `multimedia`, `x11r7`, `efl` and `matchbox`. If your package fits in one of these categories, then create your package directory in these. New subdirectories are discouraged, however.

6.2.2 `Config.in` file

Then, 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.

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

The `bool` line, `help` line and other meta-informations about the configuration option must be indented with one tab. The help text itself should be indented with one tab and two spaces, and it must mention the upstream URL of the project.

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"
```

6.2.2.1 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 express 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 toolchain options (target architecture, MMU support, C library, C++ support, large file support, thread support, RPC support, IPV6 support, WCHAR support), or for dependencies on "big" things, such as the X.org system. For dependencies on toolchain options, there should be a `comment` that is displayed when the option is not enabled, so that the user knows why the package is not available. The `depends on` keyword express 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_ACL
        bool "acl"
        select BR2_PACKAGE_ATTR
        depends on BR2_LARGEFILE
        help
          POSIX Access Control Lists, which are used to define more
          fine-grained discretionary access rights for files and
          directories.
          This package also provides libacl.

          http://savannah.nongnu.org/projects/acl

comment "acl requires a toolchain with LARGEFILE support"
        depends on !BR2_LARGEFILE
```

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"
        select BR2_PACKAGE_B
        depends on BR2_PACKAGE_A

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

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 Section 6.1.1.

6.2.3 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, installation and cleanup 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 Section 6.2.4.1 and a reference Section 6.2.4.2.
- **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 Section 6.2.5.1 and reference Section 6.2.5.2.
- **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 Section 6.2.6.1 and reference Section 6.2.6.2.

Further formating details: see the writing rules Section 6.1.2.

6.2.4 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.

6.2.4.1 `generic-package` Tutorial

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

LIBFOO_VERSION = 1.0
LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
LIBFOO_SITE = http://www.foosoftware.org/download
LIBFOO_LICENSE = GPLv3+
```

```
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:     $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -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_DEVICES
32:     /dev/foo  c  666  0  0  42  0  -  -  -
33: endef
34:
35: define LIBFOO_PERMISSIONS
36:     /bin/foo  f  4755  0  0  -  -  -  -  -
37: endef
38:
39: define LIBFOO_USERS
40:     foo -1 libfoo -1 * - - - LibFoo daemon
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`) 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 6.1 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 6.2 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 line 31..33, we define a device-node file used by this package (`LIBFOO_DEVICES`).

On line 35..37, we define the permissions to set to specific files installed by this package (`LIBFOO_PERMISSIONS`).

On lines 39..41, we define a user that is used by this package (eg. to run a daemon as non-root) (`LIBFOO_USERS`).

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.

6.2.4.2 `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.

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, branch or tag for packages that are fetched directly from their revision control system.

Examples:
`LIBFOO_VERSION =0.1.2`
`LIBFOO_VERSION =cb9d6aa9429e838f0e54faa3d455bcbab5eef057`
`LIBFOO_VERSION =stable`

- `LIBFOO_SOURCE` may contain the name of the tarball of the package. If `HOST_LIBFOO_SOURCE` is not specified, it defaults to `LIBFOO_SOURCE`. If none are specified, then the value is assumed to be `packagename-$(LIBFOO_VERSION).tar.gz`.
  Example: `LIBFOO_SOURCE =foobar-$(LIBFOO_VERSION).tar.bz2`
- `LIBFOO_PATCH` may contain a space-separated list of patch file names, that will be downloaded from the same location as the tarball indicated in `LIBFOO_SOURCE`, and then applied to the package source code. 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 `<packagename>-*.patch` present in the package directory inside Buildroot will be applied to the package after extraction (see patching a package Section 6.3). 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. Git, Subversion, Mercurial, and Bazaar are supported URL types for retrieving packages directly from source code management systems. 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`.
  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=git://github.com/kergoth/tslib.git`
  `LIBFOO_SITE=/opt/software/libfoo.tar.gz`
  `LIBFOO_SITE=$(TOPDIR)/../src/libfoo/`
- `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://`.
  - `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.
  - `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.
- `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. In a similar way, `HOST_LIBFOO_DEPENDENCIES` lists the dependencies for the current host package.

- `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_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 Section 11.1. 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 Section 11.1. 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 (eg. as a daemon, or as a cron-job). The syntax is similar in spirit to the makedevs one, and is described in the Section 11.2. 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 following list Section 7.2, use the same string 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.
- `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 7 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_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.

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 `$(STAGING_DIR)`. All development files should be installed, since they might be needed to compile other packages.
- `LIBFOO_CLEAN_CMDS`, lists the actions to perform to clean up the build directory of the package.
- `LIBFOO_UNINSTALL_TARGET_CMDS` lists the actions to uninstall the package from the target directory `$(TARGET_DIR)`
- `LIBFOO_UNINSTALL_STAGING_CMDS` lists the actions to uninstall the package from the staging directory `$(STAGING_DIR)`.
- `LIBFOO_INSTALL_INIT_SYSV` and `LIBFOO_INSTALL_INIT_SYSTEMD` list the actions to install init scripts either for the systemV-like init systems (busybox, sysvinit, etc.) 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 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:

- `$(@D)`, which contains the directory in which the package source code has been uncompressed.
- `$(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.

The last feature of the generic infrastructure is the ability to add 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 hooks are more useful for packages using the autotools infrastructure described below. However, since they are provided by the generic infrastructure, they are documented here. The exception is `LIBFOO_POST_PATCH_HOOKS`. Patching the package and producing legal info are not user definable, so `LIBFOO_POST_PATCH_HOOKS` and `LIBFOO_POST_LEGAL_INFO_HOOKS` are useful for generic packages.

The following hook points are available:

- `LIBFOO_POST_DOWNLOAD_HOOKS`
- `LIBFOO_POST_EXTRACT_HOOKS`
- `LIBFOO_PRE_PATCH_HOOKS`
- `LIBFOO_POST_PATCH_HOOKS`
- `LIBFOO_PRE_CONFIGURE_HOOKS`
- `LIBFOO_POST_CONFIGURE_HOOKS`
- `LIBFOO_POST_BUILD_HOOKS`
- `LIBFOO_POST_INSTALL_HOOKS` (for host packages only)
- `LIBFOO_POST_INSTALL_STAGING_HOOKS` (for target packages only)
- `LIBFOO_POST_INSTALL_TARGET_HOOKS` (for target packages only)
- `LIBFOO_POST_LEGAL_INFO_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
```

6.2.5 Infrastructure for autotools-based packages

6.2.5.1 `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_OPT = --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 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.

6.2.5.2 `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_OPT`, 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_OPT`, 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_OPT` to specify additional options passed to the *autoreconf* program if `LIBFOO_AUTORECONF=YES`. By default, empty.
- `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_OPT` contains the make options used to install the package to the staging directory. By default, the value is `DESTDIR=$(STAGING_DIR) install`, which is correct for most autotools packages. It is still possible to override it.
- `LIBFOO_INSTALL_TARGET_OPT` contains the make options used to install the package to the target directory. By default, the value is `DESTDIR=$(TARGET_DIR) install`. The default value is correct for most autotools packages, but it is still possible to override it if needed.
- `LIBFOO_CLEAN_OPT` contains the make options used to clean the package. By default, the value is `clean`.
- `LIBFOO_UNINSTALL_STAGING_OPT`, contains the make options used to uninstall the package from the staging directory. By default, the value is `DESTDIR=$$(STAGING_DIR) uninstall`.
- `LIBFOO_UNINSTALL_TARGET_OPT`, contains the make options used to uninstall the package from the target directory. By default, the value is `DESTDIR=$$(TARGET_DIR) uninstall`.

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 the reference documentation of the generic infrastructure 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.

6.2.6 Infrastructure for CMake-based packages

6.2.6.1 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_OPT = -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 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.

6.2.6.2 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_OPT`, to specify additional configure options to pass to CMake. 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_OPT`, to specify additional variables to pass to make in the build step. These are passed after the `make` command. By default, empty.
- `LIBFOO_INSTALL_STAGING_OPT` contains the make options used to install the package to the staging directory. By default, the value is `DESTDIR=$(STAGING_DIR) install`, which is correct for most CMake packages. It is still possible to override it.
- `LIBFOO_INSTALL_TARGET_OPT` contains the make options used to install the package to the target directory. By default, the value is `DESTDIR=$(TARGET_DIR) install`. The default value is correct for most CMake packages, but it is still possible to override it if needed.
- `LIBFOO_CLEAN_OPT` contains the make options used to clean the package. By default, the value is `clean`.

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 the reference documentation of the generic infrastructure 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.

6.2.7 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 *uClibc* C library doesn't implement gettext functionality; therefore with this C library, a separate gettext must be compiled. On the other hand, the *glibc* C library does integrate its own gettext, and in this case the separate gettext library should not be compiled, because it creates various kinds of build failures.

Additionally, some packages (such as `libglib2`) do require gettext unconditionally, while other packages (those who support `--disable-nls` in general) only require gettext when locale support is enabled.

Therefore, Buildroot defines two configuration options:

- `BR2_NEEDS_GETTEXT`, which is true as soon as the toolchain doesn't provide its own gettext implementation
- `BR2_NEEDS_GETTEXT_IF_LOCALE`, which is true if the toolchain doesn't provide its own gettext implementation and if locale support is enabled

Packages that need gettext only when locale support is enabled should:

- use `select BR2_PACKAGE_GETTEXT if BR2_NEEDS_GETTEXT_IF_LOCALE` in the `Config.in` file;
- use `$(if $(BR2_NEEDS_GETTEXT_IF_LOCALE),gettext)` in the package `DEPENDENCIES` variable in the `.mk` file.

Packages that unconditionally need gettext (which should be very rare) should:

- use `select BR2_PACKAGE_GETTEXT if BR2_NEEDS_GETTEXT` in the `Config.in` file;
- use `$(if $(BR2_NEEDS_GETTEXT),gettext)` in the package `DEPENDENCIES` variable in the `.mk` file.

6.2.8 Tips and tricks

6.2.8.1 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* `.` and `-` characters substituted with `_` (e.g.: `FOO_BAR_BOO_VERSION`).

6.2.8.2 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.

```
FOO_VERSION = v1.0 # tag or (abbreviated) commit ID
FOO_SITE = http://github.com/<user>/<package>/tarball/$(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-1234567.tar.gz`), so it is not necessary to specify it in the `.mk` file.
- When using a commit ID as version, usually the first 7 characters of the SHA1 are enough.

6.2.9 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 10.1)!

6.3 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.

6.3.1 Providing patches

6.3.1.1 Downloaded

If it is necessary to apply a patch that is available for download, then it to the <packagename>_PATCH variable. It is downloaded from the same site as the package itself. It can be a single patch, or a tarball containing a patch series.

This method is typically used for packages from Debian.

6.3.1.2 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 <packagename>-<number>-<description>.patch.

A series file, as used by quilt, may also be added in the package directory. In that case, the series file defines the patch application order.

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*.

6.3.1.3 Global patch directory

The BR2_GLOBAL_PATCH_DIR configuration file option can be used to specify a directory containing global package patches. See Section 3.4.4 for details.

6.3.2 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 directory or in the 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 <packagename>-*.patch are applied in alphabetical order. So, to ensure they are applied in the right order, it is hightly recommended to named the patch files like this: <packagename>-<number>-<description>.patch, where <number> refers to the *apply order*.
5. Run the <packagename>_POST_PATCH_HOOKS commands if defined.

If something goes wrong in the steps *3* or *4*, then the build fails.

6.3.3 Format and licensing of the package patches

Patches are released under the same license as the software that is modified.

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.

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"])
```

6.3.4 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.

6.4 Download infrastructure

TODO

Chapter 7

Legal notice and licensing

7.1 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/` subdirectory (except for proprietary packages, whose source code is not saved); patches applied to some packages by Buildroot are distributed with the Buildroot sources and are not duplicated in the `sources/` subdirectory.
- A manifest file 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".
- A `licenses/` subdirectory, which contains the license text of packages. 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 and the Buildroot source code itself (including patches to packages for which source distribution is required). When you run `make legal-info`, Buildroot produces warnings in the `README` file to inform you of relevant material that could not be saved.

7.2 License abbreviations

Here is a list of the licenses that are most widely used by packages in Buildroot, with the name used in the manifest file:

- `GPLv2`: GNU General Public License, version 2;
- `GPLv2+`: GNU General Public License, version 2 or (at your option) any later version;
- `GPLv3`: GNU General Public License, version 3;
- `GPLv3+`: GNU General Public License, version 3 or (at your option) any later version;
- `GPL`: GNU General Public License (any version);
- `LGPLv2`: GNU Library General Public License, version 2;
- `LGPLv2+`: GNU Library General Public License, version 2.1 or (at your option) any later version;
- `LGPLv2.1`: GNU Lesser General Public License, version 2.1;
- `LGPLv2.1+`: GNU Lesser General Public License, version 2.1 or (at your option) any later version;
- `LGPLv3`: GNU Lesser General Public License, version 3;
- `LGPLv3+`: GNU Lesser General Public License, version 3 or (at your option) any later version;
- `LGPL`: GNU Lesser General Public License (any version);
- `BSD-4c`: Original BSD 4-clause license;
- `BSD-3c`: BSD 3-clause license;
- `BSD-2c`: BSD 2-clause license;
- `MIT`: MIT-style license.

7.3 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. 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.

Chapter 8

Beyond Buildroot

8.1 Boot the generated images

8.1.1 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.

8.2 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.

Chapter 9

Getting involved

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

One piece of it is the document you are currently reading ;-).

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

9.1 Mailing List

Buildroot has a mailing list http://lists.busybox.net/pipermail/buildroot for discussion and development.

9.1.1 Subscribing to the mailing list

You can subscribe by visiting http://lists.busybox.net/mailman/listinfo/buildroot. Only subscribers to the Buildroot mailing list are allowed to post to this list.

The list is also available through *Gmane* http://gmane.org, at `gmane.comp.lib.uclibc.buildroot` http://dir.gmane.org/-gmane.comp.lib.uclibc.buildroot.

9.1.2 Searching the List Archives

Please search the mailing list archives before asking questions on the mailing list, since there is a good chance someone else has asked the same question before. Checking the archives is a great way to avoid annoying everyone on the list with frequently asked questions...

9.2 IRC

The Buildroot IRC is irc://freenode.net/#buildroot. The channel `#buildroot` is hosted on Freenode http://webchat.freenode.net.

When asking for help on IRC, share relevant logs or pieces of code using a code sharing website.

9.3 Patchwork

The Buildroot patch management interface is at http://patchwork.buildroot.org.

All patches and comments sent through the mailing list are automatically indexed in patchwork.

9.4 Bugtracker

The Buildroot bugtracker is at https://bugs.busybox.net.

To open a bug, see Section 10.4.

9.5 Buildroot wikipage

After the Buildroot developer day on February 3, 2012, a page dedicated to Buildroot has been created on elinux.org.

This page is reachable at http://elinux.org/Buildroot.

Currently, this page is mainly used as a *todo-list*.

9.6 Events

9.6.1 Buildroot Developer Days aside ELC-E 2012 (November 3-4, 2012 - Barcelona)

- Event page: http://elinux.org/Buildroot:DeveloperDaysELCE2012

9.6.2 Buildroot presentation at LSM 2012 (July 12-14, 2012 - Geneva)

- Announcement: http://lists.busybox.net/pipermail/buildroot/2012-May/053845.html

9.6.3 Buildroot Developer Days aside FOSDEM 2012 (February 3, 2012 - Brussels)

- Announcement & agenda thread: http://lists.busybox.net/pipermail/buildroot/2012-January/049340.html
- Report: http://lists.busybox.net/pipermail/buildroot/2012-February/050371.html

Chapter 10

Contributing to Buildroot

If you want to contribute to Buildroot, you will need a git view of the project. Refer to Section 2.2 to get it.

Currently, the mailing list is the central place for contribution. If you have not already subscribed to it, then refer to Section 9.1.1.

Recently, a web interface is also used to manage patches sent to the mailing list, see Section 9.3.

Note
Please, do not attach patches to bugs, send them to the mailing list instead (see Section 10.1).

10.1 Submitting patches

When your changes are done, and committed in your local git view, *rebase* your development branch on top of the upstream tree before generating the patch set. To do so, run:

```
$ git fetch --all --tags
$ git rebase origin/master
```

Here, 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.

Lastly, send/submit your patch set to the Buildroot mailing list:

```
$ git send-email --to buildroot@busybox.net outgoing/*
```

Note that `git` should be configured to use your mail account. To configure `git`, see `man git-send-email` or google it.

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, use `git send-email` to send your patches.

10.1.1 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 [1];
- whenever you feel it will help presenting your work, your choices, the review process, etc.

10.1.2 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, the changelog may be added in the cover letter rather than in the individual commits.

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 less than 80-character length

Some more paragraph giving some more details.

And yet another paragraph giving more details.

Signed-off-by John Doe <john.doe@noname.org>
---
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 -o outgoing origin/master
```

10.2 Reviewing/Testing 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.

Some tags are used to help following the state of any patch posted on the mailing-list:

Acked-by
: Indicates that the patch can be committed.

Tested-by
: Indicates that the patch has been tested. It is useful but not necessary to add a comment about what has been tested.

[1] RFC: (Request for comments) change proposal

10.3 Autobuild

The Buildroot community is currently setting up automatic builds in order to test more and more configurations. All build results are available at http://autobuild.buildroot.org

A good way to contribute is by fixing broken builds.

In the commit message of a patch fixing an *autobuild*, add a reference to the *build result directory* (the `dir` link in the *data column*):

```
Fixes http://autobuild.buildroot.org/results/51000a9d4656afe9e0ea6f07b9f8ed374c2e4069
```

10.4 Reporting issues/bugs, get help

Before reporting any issue, please check the mailing list archive Section 9.1.1 in case someone has already reported and fixed a similar problem.

However you choose to report bugs or get help, opening a bug Section 9.4 or send a mail to the mailing list Section 9.1.1, 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) which the build fails
- the command that fails and its output
- any information you think that may be relevant

Additionnally, your can add the `.config` file.

If some of these details are too large, do not hesitate to use a pastebin service (see http://www.similarsitesearch.com/alternatives-to/pastebin.com).

Chapter 11

Appendix

11.1 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 line for each file, with the following layout:

name	type	mode	uid	gid	major	minor	start	inc	count

There are a few non-trivial blocks here:

- `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
  - c: a character device file
  - b: a block device file
  - p: a named pipe
- `mode`, `uid` and `gid` are the usual permissions settings
- `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 put:

```
/usr/bin/foobar f       644     0       0       -       -       -       -       -
```

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     0       0       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     0       0       3       1       1       1       15
```

11.2 Makeuser 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.
- `uid` is the desired UID for the user. It must be unique, and not `0`. If set to `-1`, then a unique UID will be computed by Buildroot in the range [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`, and the group does not already exist, then a unique GID will be computed by Buildroot in the range [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.
- `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`

11.3 Package Selection for the target

11.4 Host utilities

The following packages are all available in the menu `Host utilities`.

11.5 Deprecated features

The following features are marked as *deprecated* in Buildroot due to their status either too old or unmaintained.