# Buildroot

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



## About Buildroot

Buildroot is a set of Makefiles and patches that allows you to easily generate a cross-compilation toolchain, a root filesystem and a Linux kernel image for your target. Buildroot can be used for one, two or all of these options, independently.

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.

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** or **dietlibc**). 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". 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.

Even if your embedded system uses an x86 processor, you might be interested in Buildroot for two reasons:

- The compilation toolchain on your host certainly uses the GNU Libc which is a complete but huge C standard library. Instead of using GNU Libc on your target system, you can use uClibc which is a tiny C standard library. If you want to use this C library, then you need a compilation toolchain to generate binaries linked with it. Buildroot can do that for you.
- Buildroot automates the building of a root filesystem with all needed tools like busybox. That makes it much easier than doing it by hand.

You might wonder why such a tool is needed when you can compile `gcc`, `binutils`, `uClibc` and all the other tools by hand. Of course doing so is possible but, dealing with all of the configure options and problems of every `gcc` or `binutils` version is very time-consuming and uninteresting. Buildroot automates this process through the use of Makefiles and has a collection of patches for each `gcc` and `binutils` version to make them work on most architectures.

Moreover, Buildroot provides an infrastructure for reproducing the build process of your kernel, cross-toolchain, and embedded root filesystem. Being able to reproduce the build process will be useful when a component needs to be patched or updated or when another person is supposed to take over the project.

## Obtaining 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
```

## Using Buildroot

Buildroot has a nice configuration tool similar to the one you can find in the Linux kernel (**http://www.kernel.org/**) or in Busybox (**http://www.busybox.org/**). 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, so you may need to install "development" packages for relevant libraries used by the configuration utilities. On Debian-like systems, the `libncurses5-dev` package is required to use the *menuconfig* interface, `libqt4-dev` is required to use the *xconfig* interface, and `libglib2.0-dev, libgtk2.0-dev and libglade2-dev` are needed to use the *gconfig* 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`.

This command will generally perform the following steps:

- Download source files (as required)
- Configure, build and install the cross-compiling toolchain if an internal toolchain is used, or import a toolchain if an external toolchain is used
- 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). 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, unless the `development files in target filesystem` option is selected.
- `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.

## *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.

## *Building out-of-tree*

Buildroot 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>
```

## *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
- `BUSYBOX_CONFIG_FILE=<path/to/.config>`, path to the Busybox configuration file
- `BUILDROOT_DL_DIR` to override the directory in which Buildroot stores/retrieves downloaded files

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

```
$ make UCLIBC_CONFIG_FILE=uClibc.config BUSYBOX_CONFIG_FILE=$HOI
```

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
```

## Customizing the generated target filesystem

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.
- Create your own *target skeleton*. You can start with the default skeleton available under `fs/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. At build time, the contents of the skeleton are copied to output/target before any package installation.
- In the Buildroot configuration, you can specify the path to a post-build script, that gets called *after* Buildroot builds all the selected software, but *before* the rootfs packages are assembled. The destination root filesystem folder is given as the first argument to this script, and this script can then be used to copy programs, static data or any other needed file to 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 a post-build cleanup script.
- A special package, *customize*, stored in `package/customize` can be used. You can put all the files that you want to see in the final target root filesystem in `package/customize/source`, and then enable this special package in the configuration system.

## 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:

1. Do an initial compilation of Buildroot, with busybox, without trying to customize it.

2. Invoke `make busybox-menuconfig`. The nice configuration tool appears, and you can customize everything.
3. 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 **environment variables**.

## Customizing the uClibc configuration

Just like **BusyBox**, **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:

1. Do an initial compilation of Buildroot without trying to customize uClibc.
2. 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.
3. Copy the `$(O)/toolchain/uclibc-VERSION/.config` file to a different place (like `toolchain/uClibc/uClibc-myconfig.config`, or `board/mymanufacturer/myboard/uClibc.config`) and adjust the uClibc configuration (configuration option `BR2_UCLIBC_CONFIG`) to use this configuration instead of the default one.
4. Run the compilation of Buildroot again.

Otherwise, you can simply change `toolchain/uClibc/uClibc.config`, without running the configuration assistant.

If you want to use an existing config file for uclibc, then see section **environment variables**.

## Customizing the Linux kernel configuration

The Linux kernel configuration can be customized just like **BusyBox** and **uClibc** 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 **environment variables**.

## 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 currently 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. However, implementing clean package removal is on the TODO-list of Buildroot developers.

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.

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). The problem is that these stamp files are not uniformly named and handled by the different packages, so some understanding of the particular package is needed.

For packages relying on Buildroot packages infrastructures (see **this section** for details), the following stamp files are relevant:

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

For other packages, an analysis of the specific *package.mk* file is needed. For example, the zlib Makefile used to look like this (before it was converted to the generic package infrastructure):

```
$(ZLIB_DIR)/.configured: $(ZLIB_DIR)/.patched
        (cd $(ZLIB_DIR); rm -rf config.cache; \
                [...]
        )
        touch $@

$(ZLIB_DIR)/libz.a: $(ZLIB_DIR)/.configured
        $(MAKE) -C $(ZLIB_DIR) all libz.a
        touch -c $@
```

If you want to trigger the reconfiguration, you need to remove `output/build/zlib-version/.configured`. If you want to trigger only the recompilation, you need to remove `output/build/zlib-version/libz.a`.

Note that most packages, if not all, will progressively be ported over to the generic or autotools infrastructure, making it much easier to rebuild individual packages.

# 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 tool chain (`gcc`, `binutils` and `uClibc`).

There is basically one Makefile per software package, and they are named with the `.mk` extension. Makefiles are split into three main sections:

- **toolchain** (in 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`.
- **package** (in the `package/` directory) contains the Makefiles and associated files for all user-space tools that Buildroot can compile and add to the target root filesystem. There is one sub-directory per tool.
- **target** (in the `target` directory) contains the Makefiles and associated files for software related to the generation of the target root filesystem image. Four types of

filesystems are supported: ext2, jffs2, cramfs and squashfs. For each of them there is a sub-directory with the required files. There is also a `default/` directory that contains the target filesystem skeleton.

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):

1. Create all the output directories: `staging`, `target`, `build`, `stamps`, etc. in the output directory (`output/` by default, another value can be specified using `O=`)
2. 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.
3. 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.

# 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 `MYBOARD_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 platform, 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` (after replacing, of course, MANUFACTURER and BOARDNAME with the appropriate values, in lower case letters). You can then store your patches and configurations in these directories, and reference them from the main Buildroot configuration.

## 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.

## 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`.

## Location of downloaded packages

It might be useful to know that the various tarballs that are downloaded by the Makefiles are all stored in the `DL_DIR` which by default is the `dl` directory. It's useful, for example, if you want to keep a complete version of Buildroot which is known to be working with the associated tarballs. 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 accessed by creating a symbolic link from the `dl` directory to the shared download location:

```
$ ln -s <shared download location> dl
```

Another way of accessing a shared download location is to create the `BUILDROOT_DL_DIR` environment variable. If this is set, then the value of DL_DIR in the project is overridden. The following line should be added to `"~/.bashrc"`.

```
$ export BUILDROOT_DL_DIR <shared download location>
```

## Using an external toolchain

Using an already existing toolchain is useful for different reasons:

- you already have a toolchain that is known to work for your specific CPU

- you want to speed up the Buildroot build process by skipping the long toolchain build part
- the toolchain generation feature of Buildroot is not sufficiently flexible for you (for example if you need to generate a system with *glibc* instead of *uClibc*)

Buildroot supports using existing toolchains through a mechanism called *external toolchain*. The external toolchain mechanism is enabled in the `Toolchain` menu, by selecting `External toolchain` in `Toolchain type`.

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 toolchains for ARM, PowerPC, MIPS and SuperH. Just select the toolchain profile in `Toolchain` through 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. 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, 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.

## 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.

### *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`, `java`, `x11r7`, and `games`. If your package fits in one of these categories, then create your package directory in these.

### *`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/
```

Of course, you can add other options 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://lxr.free-electrons.com/source/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"
```

### *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): These are based on an infrastructure similar to the one used for autotools-based packages, but requires 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 a **tutorial** and a **reference**.
- **Makefiles for autotools-based software** (autoconf, automake, etc.): We provide a dedicated infrastructure for such packages, since autotools is a very common build system. This infrastructure *must* be used for new packages that rely on the autotools as their build system.
  We cover them through a **tutorial** and a **reference**.
- **Manual Makefiles:** These are currently obsolete, and no new manual Makefiles should be added. However, since there are still many of them in the tree, we keep them documented in a **tutorial**.

### Makefile for generic packages : tutorial

```
############################################################
#
# libfoo
#
############################################################
LIBFOO_VERSION = 1.0
LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
LIBFOO_SITE = http://www.foosoftware.org/download
LIBFOO_INSTALL_STAGING = YES
LIBFOO_DEPENDENCIES = host-libaaa libbbb

define LIBFOO_BUILD_CMDS
    $(MAKE) CC=$(TARGET_CC) LD=$(TARGET_LD) -C $(@D) all
endef

define LIBFOO_INSTALL_STAGING_CMDS
```

```
17:     $(INSTALL) -D -m 0755 $(@D)/libfoo.a $(STAGING_DIR)/usr/
18:     $(INSTALL) -D -m 0644 $(@D)/foo.h $(STAGING_DIR)/usr/inc
19:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(STAGING_DIR)/us
20: endef
21:
22: define LIBFOO_INSTALL_TARGET_CMDS
23:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(TARGET_DIR)/usr
24:     $(INSTALL) -d -m 0755 $(TARGET_DIR)/etc/foo.d
25: endef
26:
27: $(eval $(call GENTARGETS,package,libfoo))
```

The Makefile begins on line 6 to 8 with metadata information: the version of the package (`LIBFOO_VERSION`), the name of the tarball containing the package (`LIBFOO_SOURCE`) and the Internet location at which the tarball can be downloaded (`LIBFOO_SITE`). 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 9, 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 10, 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 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.

Finally, on line 27, we call the `GENTARGETS` which generates, according to the variables defined previously, all the Makefile code necessary to make your package working.

### Makefile for generic packages : reference

The `GENTARGETS` macro takes three arguments:

- The first argument is the package directory prefix. If your package is in `package/libfoo`, then the directory prefix is `package`. If your package is in `package/editors/foo`, then the directory prefix must be `package/editors`.
- The second argument is the lower-cased package name. It must match the prefix of the variables in the `.mk` file and must match the configuration option name in the `Config.in` file. For example, if the package name is `libfoo`, then the variables in the `.mk` file must start with `LIBFOO_` and the configuration option in the `Config.in` file must be `BR2_PACKAGE_LIBFOO`.
- The third argument is optional. It can be used to tell if the package is a target package (cross-compiled for the target) or a host package (natively compiled for the host). If unspecified, it is assumed that it is a target package. See below for details.

For a given package, in a single `.mk` file, it is possible to call GENTARGETS twice, once to create the rules to generate a target package and once to create the rules to generate a host package:

```
$(eval $(call GENTARGETS,package,libfoo))
$(eval $(call GENTARGETS,package,libfoo,host))
```

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 `GENTARGETS` macro **must** be at the end of the `.mk` file, after all variable definitions.

For the target package, the `GENTARGETS` uses the variables defined by the .mk file and prefixed by the uppercased package name: `LIBFOO_*`. For the host package, it uses the `HOST_LIBFOO_*`. 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 Subversion or Git branch or tag, for packages that are fetched directly from their revision control system.
  Example: `LIBFOO_VERSION = 0.1.2`
- `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 the name of a patch, that will be downloaded from the same location as the tarball indicated in `LIBFOO_SOURCE`. If `HOST_LIBFOO_PATCH` is not specified, it defaults to `LIBFOO_PATCH`. Also note that another mechanism is available to patch a package: all files of the form `packagename-packageversion-description.patch` present in the package directory inside Buildroot will be applied to the package after extraction.
- `LIBFOO_SITE` may contain the Internet location of the package. It can either be the HTTP or FTP location of a tarball, or the URL of a Git or Subversion repository (see `LIBFOO_SITE_METHOD` below). If `HOST_LIBFOO_SITE` is not specified, it defaults to `LIBFOO_SITE`. If none are specified, then the location is assumed to be `http://$$(BR2_SOURCEFORGE_MIRROR).dl.sourceforge.net/sourceforge/packag`
  Examples:
  `LIBFOO_SITE=http://www.libfoosoftware.org/libfoo`
  `LIBFOO_SITE=http://svn.xiph.org/trunk/Tremor/`
- `LIBFOO_SITE_METHOD` may contain the method to fetch the package source code. It can either be `wget` (for normal FTP/HTTP downloads of tarballs), `svn`, `git` or `bzr`. When not specified, it is guessed from the URL given in `LIBFOO_SITE`: `svn://`, `git://` and `bzr://` URLs will use the `svn`, `git` and `bzr` methods respectively. All other URL-types will use the `wget` method. So for example, in the

case of a package whose source code is available through Subversion repository on HTTP, one *must* specifiy `LIBFOO_SITE_METHOD=svn`. For `svn` and `git` methods, what Buildroot does is a checkout/clone of the repository which is then tarballed and stored into the download cache. Next builds will not checkout/clone again, but will use the tarball directly. When `HOST_LIBFOO_SITE_METHOD` is not specified, it defaults to the value of `LIBFOO_SITE_METHOD`. See `package/multimedia/tremor/` for an example.
- `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 dependency 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.

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_CONFIGURE_CMDS`, used to list the actions to be performed to configure the package before its compilation
- `LIBFOO_BUILD_CMDS`, used to list the actions to be performed to compile the package
- `HOST_LIBFOO_INSTALL_CMDS`, used to list 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 `$(HOST_DIR)`. All files, including development files such as headers should be installed, since other packages might be compiled on top of this package.
- `LIBFOO_INSTALL_TARGET_CMDS`, used to list the actions to be performed to install the package to the target directory, when the package is a target package. The package must install its files to the directory given by `$(TARGET_DIR)`. Only the

files required for *documentation* and *execution* of the package should be installed. Header files should not be installed, they will be copied to the target, if the `development files in target filesystem` option is selected.
- `LIBFOO_INSTALL_STAGING_CMDS`, used to list 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 `$(STAGING_DIR)`. All development files should be installed, since they might be needed to compile other packages.
- `LIBFOO_CLEAN_CMDS`, used to list the actions to perform to clean up the build directory of the package.
- `LIBFOO_UNINSTALL_TARGET_CMDS`, used to list the actions to uninstall the package from the target directory `$(TARGET_DIR)`
- `LIBFOO_UNINSTALL_STAGING_CMDS`, used to list the actions to uninstall the package from the staging directory `$(STAGING_DIR)`.

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 is not user definable, so `LIBFOO_POST_PATCH_HOOKS` will be userful for generic packages.

The following hook points are available:

- `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)

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
```

### Makefile for autotools-based packages : tutorial

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

```
#############################################################
#
# libfoo
#
#############################################################
LIBFOO_VERSION = 1.0
LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
LIBFOO_SITE = http://www.foosoftware.org/download
LIBFOO_INSTALL_STAGING = YES
LIBFOO_INSTALL_TARGET = YES
```

```
11: LIBFOO_CONF_OPT = --enable-shared
12: LIBFOO_DEPENDENCIES = libglib2 host-pkg-config
13:
14: $(eval $(call AUTOTARGETS,package,libfoo))
```

On line 6, we declare the version of the package.

On line 7 and 8, 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 9, 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 10, we tell Buildroot to also install the package to the target directory. This directory contains what will become the root filesystem running on the target. Usually, we try not to install header files and to install stripped versions of the binary. By default, target installation is enabled, so in fact, this line is not strictly necessary. Also by default, packages are installed in this location using the `make install` command.

On line 11, 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 12, we declare our dependencies, so that they are built before the build process of our package starts.

Finally, on line line 14, we invoke the `AUTOTARGETS` macro that generates all the Makefile rules that actually allows the package to be built.

### Makefile for autotools packages : reference

The main macro of the autotools package infrastructure is `AUTOTARGETS`. It has the same number of arguments and the same semantic as the `GENTARGETS` macro, which is the main macro of the generic package infrastructure. For autotools packages, the ability to have target and host packages is also available (and is actually widely used).

Just like the generic infrastructure, the autotools infrastructure works by defining a number of variables before calling the AUTOTARGETS 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.

### Makefile for CMake-based packages : 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: LIBFOO_VERSION = 1.0
07: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
08: LIBFOO_SITE = http://www.foosoftware.org/download
09: LIBFOO_INSTALL_STAGING = YES
10: LIBFOO_INSTALL_TARGET = YES
11: LIBFOO_CONF_OPT = -DBUILD_DEMOS=ON
12: LIBFOO_DEPENDENCIES = libglib2 host-pkg-config
13:
14: $(eval $(call CMAKETARGETS,package,libfoo))
```

On line 6, we declare the version of the package.

On line 7 and 8, 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 9, 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 10, we tell Buildroot to also install the package to the target directory. This directory contains what will become the root filesystem running on the target. Usually, we try not to install header files and to install stripped versions of the binary. By default, target installation is enabled, so in fact, this line is not strictly necessary. Also by default, packages are installed in this location using the `make install` command.

On line 11, we tell Buildroot to pass custom options to CMake when it is configuring the package.

On line 12, we declare our dependencies, so that they are built before the build process of our package starts.

Finally, on line line 14, we invoke the `CMAKETARGETS` macro that generates all the Makefile rules that actually allows the package to be built.

### Makefile for CMake packages : reference

The main macro of the CMake package infrastructure is `CMAKETARGETS`. It has the same number of arguments and the same semantic as the `GENTARGETS` macro, which is the main macro of the generic package infrastructure. For CMake packages, the ability to have target and host packages is also available.

Just like the generic infrastructure, the CMake infrastructure works by defining a number of variables before calling the `CMAKETARGETS` 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.

### Manual Makefile : tutorial

**NOTE: new manual makefiles should not be created, and existing manual makefiles should be converted either to the generic, autotools or cmake infrastructure. This section is only kept to document the existing manual makefiles and to help understand how they work.**

```
#############################################################
#
# libfoo
#
#############################################################
LIBFOO_VERSION:=1.0
LIBFOO_SOURCE:=libfoo-$(LIBFOO_VERSION).tar.gz
LIBFOO_SITE:=http://www.foosoftware.org/downloads
```

```
LIBFOO_DIR:=$(BUILD_DIR)/foo-$(FOO_VERSION)
LIBFOO_BINARY:=foo
LIBFOO_TARGET_BINARY:=usr/bin/foo

$(DL_DIR)/$(LIBFOO_SOURCE):
    $(call DOWNLOAD,$(LIBFOO_SITE),$(LIBFOO_SOURCE))

$(LIBFOO_DIR)/.source: $(DL_DIR)/$(LIBFOO_SOURCE)
    $(ZCAT) $(DL_DIR)/$(LIBFOO_SOURCE) | tar -C $(BUILD_DIR)
    touch $@

$(LIBFOO_DIR)/.configured: $(LIBFOO_DIR)/.source
    (cd $(LIBFOO_DIR); rm -rf config.cache; \
            $(TARGET_CONFIGURE_OPTS) \
            $(TARGET_CONFIGURE_ARGS) \
            ./configure \
            --target=$(GNU_TARGET_NAME) \
            --host=$(GNU_TARGET_NAME) \
            --build=$(GNU_HOST_NAME) \
            --prefix=/usr \
            --sysconfdir=/etc \
    )
    touch $@

$(LIBFOO_DIR)/$(LIBFOO_BINARY): $(LIBFOO_DIR)/.configured
    $(MAKE) CC=$(TARGET_CC) -C $(LIBFOO_DIR)

$(TARGET_DIR)/$(LIBFOO_TARGET_BINARY): $(LIBFOO_DIR)/$(LIBFO
    $(MAKE) DESTDIR=$(TARGET_DIR) -C $(LIBFOO_DIR) install-s
    rm -Rf $(TARGET_DIR)/usr/man

libfoo: uclibc ncurses $(TARGET_DIR)/$(LIBFOO_TARGET_BINARY)

libfoo-source: $(DL_DIR)/$(LIBFOO_SOURCE)

libfoo-clean:
    $(MAKE) prefix=$(TARGET_DIR)/usr -C $(LIBFOO_DIR) uninst
    -$(MAKE) -C $(LIBFOO_DIR) clean
```

```
47:
48: libfoo-dirclean:
49:     rm -rf $(LIBFOO_DIR)
50:
51: ############################################################
52: #
53: # Toplevel Makefile options
54: #
55: ############################################################
56: ifeq ($(BR2_PACKAGE_LIBFOO),y)
57: TARGETS+=libfoo
58: endif
```

First of all, this Makefile example works for a package which comprises a single binary executable. For other software, such as libraries or more complex stuff with multiple binaries, it must be adapted. For examples look at the other `*.mk` files in the `package` directory.

At lines **6-11**, a couple of useful variables are defined:

- `LIBFOO_VERSION`: The version of *libfoo* that should be downloaded.
- `LIBFOO_SOURCE`: The name of the tarball of *libfoo* on the download website or FTP site. As you can see `LIBFOO_VERSION` is used.
- `LIBFOO_SITE`: The HTTP or FTP site from which *libfoo* archive is downloaded. It must include the complete path to the directory where `LIBFOO_SOURCE` can be found.
- `LIBFOO_DIR`: The directory into which the software will be configured and compiled. Basically, it's a subdirectory of `BUILD_DIR` which is created upon decompression of the tarball.
- `LIBFOO_BINARY`: Software binary name. As said previously, this is an example for a package with a single binary.
- `LIBFOO_TARGET_BINARY`: The full path of the binary inside the target filesystem.

Lines **13-14** define a target that downloads the tarball from the remote site to the download directory (`DL_DIR`).

Lines **16-18** define a target and associated rules that uncompress the downloaded tarball. As you can see, this target depends on the tarball file so that the previous target (lines **13-14**) is called before executing the rules of the current target.

Uncompressing is followed by *touching* a hidden file to mark the software as having been uncompressed. This trick is used everywhere in a Buildroot Makefile to split steps (download, uncompress, configure, compile, install) while still having correct dependencies.

Lines **20-31** define a target and associated rules that configure the software. It depends on the previous target (the hidden `.source` file) so that we are sure the software has been uncompressed. In order to configure the package, it basically runs the well-known `./configure` script. As we may be doing cross-compilation, `target`, `host` and `build` arguments are given. The prefix is also set to `/usr`, not because the software will be installed in `/usr` on your host system, but because the software will be installed in `/usr` on the target filesystem. Finally it creates a `.configured` file to mark the software as configured.

Lines **33-34** define a target and a rule that compile the software. This target will create the binary file in the compilation directory and depends on the software being already configured (hence the reference to the `.configured` file). It basically runs `make` inside the source directory.

Lines **36-38** define a target and associated rules that install the software inside the target filesystem. They depend on the binary file in the source directory to make sure the software has been compiled. They use the `install-strip` target of the software `Makefile` by passing a `DESTDIR` argument so that the `Makefile` doesn't try to install the software in the host `/usr` but rather in the target `/usr`. After the installation, the `/usr/man` directory inside the target filesystem is removed to save space.

Line **40** defines the main target of the software — the one that will eventually be used by the top level `Makefile` to download, compile, and then install this package. This target should first of all depend on all needed dependencies of the software (in our example, *uclibc* and *ncurses*) and also depend on the final binary. This last dependency will call all previous dependencies in the correct order.

Line **42** defines a simple target that only downloads the code source. This is not used during normal operation of Buildroot, but is needed if you intend to download all required sources at once for later offline build. Note that if you add a new package, providing a `libfoo-source` target is *mandatory* to support users that wish to do offline-builds. Furthermore, it eases checking if all package-sources are downloadable.

Lines **44-46** define a simple target to clean the software build by calling the Makefile with the appropriate options. The `-clean` target should run `make clean` on $(BUILD_DIR)/package-version and MUST uninstall all files of the package from $(STAGING_DIR) and from $(TARGET_DIR).

Lines **48-49** define a simple target to completely remove the directory in which the software was uncompressed, configured and compiled. The `-dirclean` target MUST completely rm $(BUILD_DIR)/ package-version.

Lines **51-58** add the target `libfoo` to the list of targets to be compiled by Buildroot, by first checking if the configuration option for this package has been enabled using the configuration tool. If so, it then "subscribes" this package to be compiled by adding the package to the TARGETS global variable. The name added to the TARGETS global variable is the name of this package's target, as defined on line **40**, which is used by Buildroot to download, compile, and then install this package.

## *Gettext integration and interaction with packages*

Many packages that support internationalization use the gettext library. Dependencies for this library are fairly complicated and therefore, deserves 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

Therefore, packages that unconditionally need gettext should:

1. Use `select BR2_PACKAGE_GETTEXT if BR2_NEEDS_GETTEXT` and possibly `select BR2_PACKAGE_LIBINTL if BR2_NEEDS_GETTEXT`, if libintl is also needed
2. Use `$(if $(BR2_NEEDS_GETTEXT),gettext)` in the package `DEPENDENCIES` variable

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

1. Use `select BR2_PACKAGE_GETTEXT if BR2_NEEDS_GETTEXT_IF_LOCALE` and possibly `select BR2_PACKAGE_LIBINTL if BR2_NEEDS_GETTEXT_IF_LOCALE`, if libintl is also needed
2. Use `$(if $(BR2_NEEDS_GETTEXT_IF_LOCALE),gettext)` in the package `DEPENDENCIES` variable

### *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 Buildroot developers!

# Frequently asked questions

- **The boot hangs after `Starting network...`**
- **module-init-tools fails to build with `cannot find -lc`**

### *The boot hangs after `Starting network...`*

If the boot process seems to hang after the following messages (messages not necessarly 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
```

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 in 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.

### *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.

## Resources

To learn more about Buildroot you can visit these websites:

- **http://www.uclibc.org/**
- **http://www.busybox.net/**