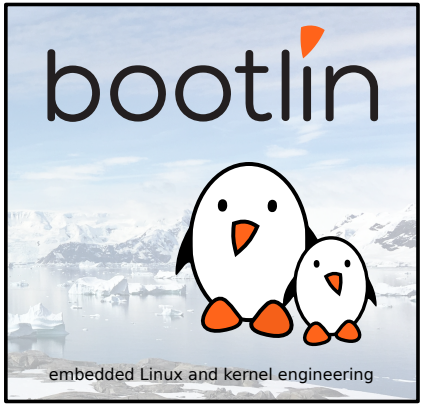
Download infrastructure in Buildroot



© Copyright 2004-2022, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Introduction

- One important aspect of Buildroot is to fetch source code or binary files from third party projects.
- Download supported from HTTP(S), FTP, Git, Subversion, CVS, Mercurial, etc.
- Being able to do reproducible builds over a long period of time requires understanding the download infrastructure.

Download location

- Each Buildroot package indicates in its `.mk` file which files it needs to be downloaded.
- Can be a tarball, one or several patches, binary files, etc.
- When downloading a file, Buildroot will successively try the following locations:
  1. The local `$(DL_DIR)` directory where downloaded files are kept
  2. The **primary site**, as indicated by `BR2_PRIMARY_SITE`
  3. The **original site**, as indicated by the package `.mk` file
  4. The **backup Buildroot mirror**, as indicated by `BR2_BACKUP_SITE`

DL_DIR

- Once a file has been downloaded by Buildroot, it is cached in the directory pointed by `$(DL_DIR)`, in a sub-directory named after the package.
- By default, `$(TOPDIR)/dl`
- Can be changed
  - using the `BR2_DL_DIR` configuration option
  - or by passing the `BR2_DL_DIR` environment variable, which overrides the config option of the same name
- The download mechanism is written in a way that allows independent parallel builds to share the same `DL_DIR` (using atomic renaming of files)
- No cleanup mechanism: files are only added, never removed, even when the package version is updated.

Primary site

- The BR2_PRIMARY_SITE option allows to define the location of a HTTP or FTP server.
- By default empty, so this feature is disabled.
- When defined, used in priority over the original location.
- Allows to do a local mirror, in your company, of all the files that Buildroot needs to download.
- When option BR2_PRIMARY_SITE_ONLY is enabled, only the *primary site* is used
  - It does not fall back on the original site and the backup Buildroot mirror
  - Guarantees that all downloads must be in the primary site

Backup Buildroot mirror

- Since sometimes the upstream locations disappear or are temporarily unavailable, having a backup server is useful
- Address configured through `BR2_BACKUP_SITE`
- Defaults to `http://sources.buildroot.net`
  - maintained by the Buildroot community
  - updated before every Buildroot release to contain the downloaded files for all packages
  - exception: cannot store all possible versions for packages that have their version as a configuration option. Generally only affects the kernel or bootloader, which typically don't disappear upstream.

Special case of VCS download

- When a package uses the source code from Git, Subversion or another VCS, Buildroot cannot directly download a tarball.
- It uses a VCS-specific method to fetch the specified version of the source from the VCS repository
- The source code is checked-out/cloned inside `DL_DIR` and kept to speed-up further downloads of the same project (caching only implemented for Git)
- Finally a tarball containing only the source code (and not the version control history or metadata) is created and stored in `DL_DIR`
  - Example: `avrdude-eabe067c4527bc2eedc5db9288ef5cf1818ec720.tar.gz`
- This tarball will be re-used for the next builds, and attempts are made to download it from the primary and backup sites.
- Due to this, always use a tag name or a full commit id, and never a branch name: the code will never be re-downloaded when the branch is updated.

File integrity checking

- Buildroot packages can provide a `.hash` file to provide *hashes* for the downloaded files.
- The download infrastructure uses this hash file when available to check the integrity of the downloaded files.
- Hashs are checked every time a downloaded file is used, even if it is already cached in `$(DL_DIR)`.
- If the hash is incorrect, the download infrastructure attempts to re-download the file once. If that still fails, the build aborts with an error.

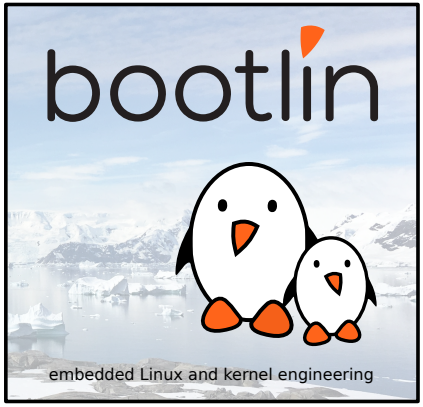
Hash checking message

```
strace-4.10.tar.xz: OK (md5: 107a5be455493861189e9b57a3a51912)
strace-4.10.tar.xz: OK (sha1: 5c3ec4c5a9eeb440d7ec70514923c2e7e7f9ab6c)
>>> strace 4.10 Extracting
```

Download-related make targets

- `make source`
  - Triggers the download of all the files needed to build the current configuration.
  - All files are stored in `$(DL_DIR)`
  - Allows to prepare a fully offline build
- `make external-deps`
  - Lists the files from `$(DL_DIR)` that are needed for the current configuration to build.
  - Does not guarantee that all files are in `$(DL_DIR)`, a `make source` is required

GNU Make 101



© Copyright 2004-2022, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Introduction

- Buildroot being implemented in **GNU Make**, it is quite important to know the basics of this language
  - Basics of *make* rules
  - Defining and referencing variables
  - Conditions
  - Defining and using functions
  - Useful *make* functions
- This does not aim at replacing a full course on *GNU Make*
- https://www.gnu.org/software/make/manual/make.html
- https://www.nostarch.com/gnumake

Basics of *make* rules

- At their core, *Makefiles* are simply defining **rules** to create **targets** from **prerequisites** using **recipe commands**

```
TARGET ...: PREREQUISITES ...
        RECIPE
        ...
```

- **target**: name of a file that is generated. Can also be an arbitrary action, like `clean`, in which case it's a **phony target**
- **prerequisites**: list of files or other targets that are needed as dependencies of building the current target.
- **recipe**: list of shell commands to create the target from the prerequisites

Rule example

Makefile

```
clean:
        rm -rf $(TARGET_DIR) $(BINARIES_DIR) $(HOST_DIR) \
                $(BUILD_DIR) $(BASE_DIR)/staging \
                $(LEGAL_INFO_DIR)

distclean: clean
        [...]
        rm -rf $(BR2_CONFIG) $(CONFIG_DIR)/.config.old \
               $(CONFIG_DIR)/.auto.deps
```

- `clean` and `distclean` are phony targets

Defining and referencing variables

- Defining variables is done in different ways:
  - `FOOBAR = value`, expanded at time of use
  - `FOOBAR := value`, expanded at time of assignment
  - `FOOBAR += value`, append to the variable, with a separating space, defaults to expanded at the time of use
  - `FOOBAR ?= value`, defined only if not already defined
  - Multi-line variables are described using `define NAME ... endef`:

```
define FOOBAR
line 1
line 2
endef
```

- Make variables are referenced using the `$(FOOBAR)` syntax.

Conditions

- With `ifeq` or `ifneq`

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

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

- With the `$(if ...)` make function:

```
HOSTAPD_LIBS += $(if $(BR2_STATIC_LIBS),-lcrypto -lz)
```

Defining and using functions

- Defining a function is exactly like defining a variable:

```
MESSAGE = echo "$(TERM_BOLD)>>> $($(PKG)_NAME) $($(PKG)_VERSION) $(call qstrip,$(1))$(TERM_RESET)"

define legal-license-header # pkg, license-file, {HOST|TARGET}
	printf "$(LEGAL_INFO_SEPARATOR)\n\t$(1):\
		$(2)\n$(LEGAL_INFO_SEPARATOR)\n\n\n" >>$(LEGAL_LICENSES_TXT_$(3))
endef
```

- Arguments accessible as `$(1)`, `$(2)`, etc.
- Called using the `$(call func,arg1,arg2)` construct

```
$(BUILD_DIR)/%/.stamp_extracted:
	[...]
	@$(call MESSAGE,"Extracting")

define legal-license-nofiles # pkg, {HOST|TARGET}
	$(call legal-license-header,$(1),unknown license file(s),$(2))
endef
```

Useful *make* functions

- `subst` and `patsubst` to replace text

```
ICU_SOURCE = icu4c-$(subst .,_,$(ICU_VERSION))-src.tgz
```

- `filter` and `filter-out` to filter entries
- `foreach` to implement loops

```
$(foreach incdir,$(TI_GFX_HDR_DIRS),
      $(INSTALL) -d $(STAGING_DIR)/usr/include/$(notdir $(incdir)); \
      $(INSTALL) -D -m 0644 $(@D)/include/$(incdir)/*.h \
            $(STAGING_DIR)/usr/include/$(notdir $(incdir))/
)
```

- `dir`, `notdir`, `addsuffix`, `addprefix` to manipulate file names

```
UBOOT_SOURCE = $(notdir $(UBOOT_TARBALL))

IMAGEMAGICK_CONFIG_SCRIPTS = \
       $(addsuffix -config,Magick MagickCore MagickWand Wand)
```

- And many more, see the *GNU Make* manual for details.

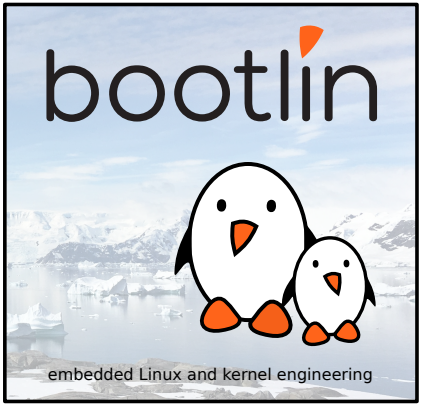
Writing recipes

- Recipes are just shell commands
- Each line must be indented with one `Tab`
- Each line of shell command in a given recipe is independent from the other: variables are not shared between lines in the recipe
- Need to use a single line, possibly split using \, to do complex shell constructs
- Shell variables must be referenced using `$$name`.

package/pppd/pppd.mk

```
define PPPD_INSTALL_RADIUS
	...
	for m in $(PPPD_RADIUS_CONF); do \
		$(INSTALL) -m 644 -D $(PPPD_DIR)/pppd/plugins/radius/etc/$$m \
			$(TARGET_DIR)/etc/ppp/radius/$$m; \
	done
	...
endef
```

Integrating new packages in Buildroot



© Copyright 2004-2022, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!

Why adding new packages in Buildroot?

- A *package* in Buildroot-speak is the **set of meta-information needed to automate the build process** of a certain component of a system.
- Can be used for open-source, third party proprietary components, or in-house components.
- Can be used for user space components (libraries and applications) but also for firmware, kernel drivers, bootloaders, etc.
- Do not confuse with the notion of *binary package* in a regular Linux distribution.

Basic elements of a Buildroot package

- A directory, `package/foo`
- A `Config.in` file, written in *kconfig* language, describing the configuration options for the package.
- A `<pkg>.mk` file, written in *make*, describing where to fetch the source, how to build and install it, etc.
- An optional `<pkg>.hash` file, providing hashes to check the integrity of the downloaded tarballs and license files.
- Optionally, `.patch` files, that are applied on the package source code before building.
- Optionally, any additional file that might be useful for the package: init script, example configuration file, etc.

Config.in file

package/<pkg>/Config.in: basics

- Describes the configuration options for the package.
- Written in the *kconfig* language.
- One option is mandatory to enable/disable the package, it **must** be named BR2_PACKAGE_<PACKAGE>.

```
config BR2_PACKAGE_STRACE
        bool "strace"
        help
          A useful diagnostic, instructional, and debugging tool.
          Allows you to track what system calls a program makes
          while it is running.

          http://sourceforge.net/projects/strace/
```

- The main package option is a `bool` with the package name as the prompt. Will be visible in `menuconfig`.
- The help text give a quick description, and the homepage of the project.

package/<pkg>/Config.in: inclusion

- The hierarchy of configuration options visible in `menuconfig` is built by reading the top-level `Config.in` file and the other `Config.in` file it includes.
- All `package/<pkg>/Config.in` files are included from `package/Config.in`.
- The location of a package in one of the package sub-menu is decided in this file.

package/Config.in

```
menu "Target packages"
menu "Audio and video applications"
        source "package/alsa-utils/Config.in"
        ...
endmenu
...
menu "Libraries"
menu "Audio/Sound"
        source "package/alsa-lib/Config.in"
        ...
endmenu
...
```

package/<pkg>/Config.in: dependencies

- *kconfig* allows to express dependencies using `select` or `depends on` statements
  - `select` is an automatic dependency: if option *A* `select` option *B*, as soon as *A* is enabled, *B* will be enabled, and cannot be unselected.
  - `depends on` is a user-assisted dependency: if option *A* `depends on` option *B*, *A* will only be visible when *B* is enabled.
- Buildroot uses them as follows:
  - `depends on` for architecture, toolchain feature, or *big* feature dependencies. E.g: package only available on x86, or only if wide char support is enabled, or depends on Python.
  - `select` for enabling the necessary other packages needed to build the current package (libraries, etc.)
- Such dependencies only ensure consistency at the configuration level. They **do not guarantee build ordering**!

package/<pkg>/Config.in: dependency example

btrfs-progs package

```
config BR2_PACKAGE_BTRFS_PROGS
        bool "btrfs-progs"
        depends on BR2_USE_MMU # util-linux
        depends on BR2_TOOLCHAIN_HAS_THREADS
        select BR2_PACKAGE_LZO
        select BR2_PACKAGE_UTIL_LINUX
        select BR2_PACKAGE_UTIL_LINUX_LIBBLKID
        select BR2_PACKAGE_UTIL_LINUX_LIBUUID
        select BR2_PACKAGE_ZLIB
        help
          Btrfs filesystem utilities

          https://btrfs.wiki.kernel.org/index.php/Main_Page

comment "btrfs-progs needs a toolchain w/ threads"
        depends on BR2_USE_MMU
        depends on !BR2_TOOLCHAIN_HAS_THREADS
```

- `depends on BR2_USE_MMU`, because the package uses `fork()`. Note that there is no comment displayed about this dependency, because it's a limitation of the architecture.
- `depends on BR2_TOOLCHAIN_HAS_THREADS`, because the package requires thread support from the toolchain. There is an associated comment, because such support can be added to the toolchain.
- Multiple `select BR2_PACKAGE_*`, because the package needs numerous libraries.

Dependency propagation

- A limitation of *kconfig* is that it doesn't propagate `depends on` dependencies accross `select` dependencies.
- Scenario: if package *A* has a `depends on FOO`, and package *B* has a `select A`, then package *B* must replicate the `depends on FOO`.

libglib2 package

```
config BR2_PACKAGE_LIBGLIB2
        bool "libglib2"
        select BR2_PACKAGE_GETTEXT if ...
        select BR2_PACKAGE_LIBICONV if ...
        select BR2_PACKAGE_LIBFFI
        select BR2_PACKAGE_ZLIB
        [...]
        depends on BR2_USE_WCHAR # gettext
        depends on BR2_TOOLCHAIN_HAS_THREADS
        depends on BR2_USE_MMU # fork()
[...]
```

neard package

```
config BR2_PACKAGE_NEARD
        bool "neard"
        depends on BR2_USE_WCHAR # libglib2
        # libnl, dbus, libglib2
        depends on BR2_TOOLCHAIN_HAS_THREADS
        depends on BR2_USE_MMU # dbus, libglib2
        select BR2_PACKAGE_DBUS
        select BR2_PACKAGE_LIBGLIB2
        select BR2_PACKAGE_LIBNL
[...]
```

Config.in.host for host packages?

- Most of the packages in Buildroot are *target* packages, i.e. they are cross-compiled for the target architecture, and meant to be run on the target platform.
- Some packages have a *host* variant, built to be executed on the build machine. Such packages are needed for the build process of other packages.
- The majority of *host* packages are not visible in `menuconfig`: they are just dependencies of other packages, the user doesn't really need to know about them.
- A few of them are potentially directly useful to the user (flashing tools, etc.), and can be shown in the *Host utilities* section of `menuconfig`.
- In this case, the configuration option is in a `Config.in.host` file, included from `package/Config.in.host`, and the option must be named `BR2_PACKAGE_HOST_<PACKAGE>`.

Config.in.host example

package/Config.in.host

```
menu "Host utilities"

        source "package/genimage/Config.in.host"
        source "package/lpc3250loader/Config.in.host"
        source "package/openocd/Config.in.host"
        source "package/qemu/Config.in.host"

endmenu
```

package/openocd/Config.in.host

```
config BR2_PACKAGE_HOST_OPENOCD
        bool "host openocd"
        help
          OpenOCD - Open On-Chip Debugger

          http://openocd.org
```

Config.in sub-options

- Additional sub-options can be defined to further configure the package, to enable or disable extra features.
- The value of such options can then be fetched from the package .mk file to adjust the build accordingly.
- Run-time configuration does not belong to Config.in.

package/pppd/Config.in

```
config BR2_PACKAGE_PPPD
        bool "pppd"
        depends on !BR2_STATIC_LIBS
        depends on BR2_USE_MMU
        ...

if BR2_PACKAGE_PPPD

config BR2_PACKAGE_PPPD_FILTER
        bool "filtering"
        select BR2_PACKAGE_LIBPCAP
        help
          Packet filtering abilities for pppd. If enabled,
          the pppd active-filter and pass-filter options
          are available.

config BR2_PACKAGE_PPPD_RADIUS
        bool "radius"
        help
          Install RADIUS support for pppd

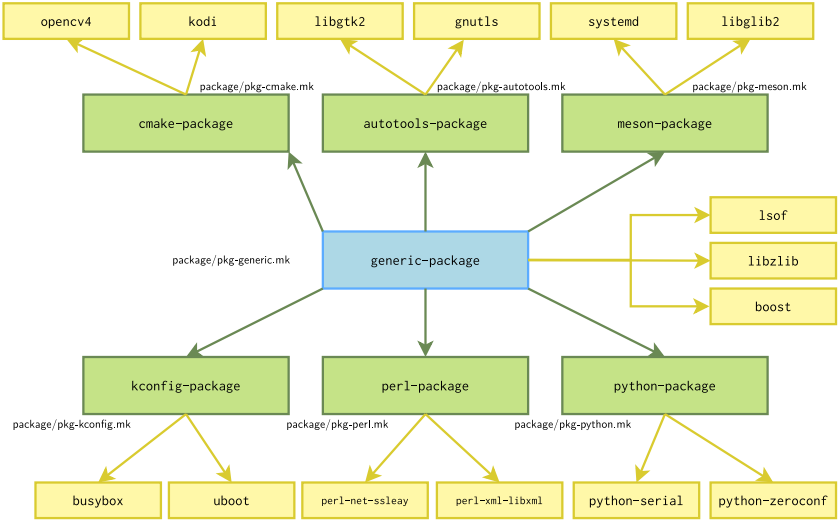
endif
```

Package infrastructures

Package infrastructures: what is it?

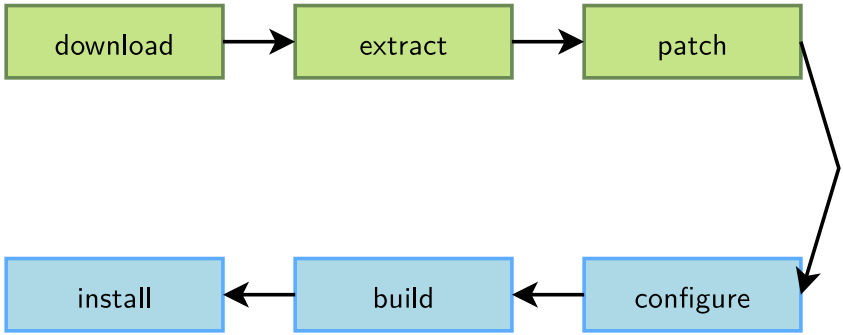
- Each software component to be built by Buildroot comes with its own *build system*.
- Buildroot does not re-invent the build system of each component, it simply uses it.
- Numerous build systems available: hand-written Makefiles or shell scripts, *autotools*, *Meson*, *CMake* and also some specific to languages: Python, Perl, Lua, Erlang, etc.
- In order to avoid duplicating code, Buildroot has *package infrastructures* for well-known build systems.
- And a generic package infrastructure for software components with non-standard build systems.

Package infrastructures



generic-package infrastructure

- To be used for software components having non-standard build systems.
- Implements a default behavior for the downloading, extracting and patching steps of the package build process.
- Implements init script installation, legal information collection, etc.
- Leaves to the package developer the responsibility of describing what should be done for the configuration, building and installation steps.


download
extract
patch
configure
build
install
Implemented by the generic-package infrastructure
Implemented by the package itself

Other package infrastructures

- The other package infrastructures are meant to be used when the software component uses a well-known build system.
- They *inherit* all the behavior of the `generic-package` infrastructure: downloading, extracting, patching, etc.
- And in addition to that, they typically implement a default behavior for the configuration, compilation and installation steps.
- For example, `autotools-package` will implement the configuration step as a call to the `./configure` script with the right arguments.
- `pkg-kconfig` is an exception, it only provides some helpers for packages using Kconfig, but does not implement the configure, build and installation steps.

.mk file for generic-package

The <pkg>.mk file

- The `.mk` file of a package does not look like a normal Makefile.
- It is a succession of variable definitions, which must be prefixed by the uppercase package name.
  - `FOOBAR_SITE = https://foobar.com/downloads/`
  - ```
    define FOOBAR_BUILD_CMDS
            $(MAKE) -C $(@D)
    endef
    ```
- And ends with a call to the desired package infrastructure macro.
  - `$(eval $(generic-package))`
  - `$(eval $(autotools-package))`
  - `$(eval $(host-autotools-package))`
- The variables tell the package infrastructure what to do for this specific package.
    ```

Naming conventions

- The Buildroot package infrastructures make a number of assumption on variables and files naming.
- The following **must** match to allow the package infrastructure to work for a given package:
  - The directory where the package description is located **must** be `package/<pkg>/`, where `<pkg>` is the lowercase name of the package.
  - The `Config.in` option enabling the package **must** be named `BR2_PACKAGE_<PKG>`, where `<PKG>` is the uppercase name of the package.
  - The variables in the `.mk` file **must** be prefixed with `<PKG>_`, where `<PKG>` is the uppercase name of the package.
- Note: a `-` in the lower-case package name is translated to `_` in the upper-case package name.

Naming conventions: global namespace

- The package infrastructure expects all variables it uses to be prefixed by the uppercase package name.
- If your package needs to define additional private variables not used by the package infrastructure, they **should** also be prefixed by the **uppercase package name**.
- The **namespace of variables is global** in Buildroot!
  - If two packages created a variable named `BUILD_TYPE`, it will silently conflict.

Behind the scenes

- Behind the scenes, `$(eval $(generic-package))`:
  - is a *make* macro that is expanded
  - infers the name of the current package by looking at the directory name: `package/<pkg>/<pkg>.mk`: `<pkg>` is the package name
  - will use all the variables prefixed by `<PKG>_`
  - and expand to a set of *make* rules and variable definitions that describe what should be done for each step of the package build process

.mk file: accessing the configuration

- The Buildroot `.config` file is a succession of lines `name = value`
  - This file is valid *make* syntax!
- The main Buildroot `Makefile` simply includes it, which turns every Buildroot configuration option into a *make* variable.
- From a package `.mk` file, one can directly use such variables:

```
ifeq ($(BR2_PACKAGE_LIBCURL),y)
...
endif

FOO_DEPENDENCIES += $(if $(BR2_PACKAGE_TIFF),tiff)
```

- Hint: use the *make* `qstrip` function to remove double quotes on string options:

```
NODEJS_MODULES_LIST = $(call qstrip,$(BR2_PACKAGE_NODEJS_MODULES_ADDITIONAL))
```

Download related variables

- <pkg>_SITE, **download location**
  - HTTP(S) or FTP URL where a tarball can be found, or the address of a version control repository.
  - `CAIRO_SITE = http://cairographics.org/releases`
  - `FMC_SITE = git://git.freescale.com/ppc/sdk/fmc.git`
- <pkg>_VERSION, **version of the package**
  - version of a tarball, or a commit, revision or tag for version control systems
  - `CAIRO_VERSION = 1.14.2`
  - `FMC_VERSION = fsl-sdk-v1.5-rc3`
- <pkg>_SOURCE, **file name** of the tarball
  - The full URL of the downloaded tarball is `$(<pkg>_SITE)/$(<pkg>_SOURCE)`
  - When not specified, defaults to `<pkg>-$(<pkg>_VERSION).tar.gz`
  - `CAIRO_SOURCE = cairo-$(CAIRO_VERSION).tar.xz`

Available download methods

- Buildroot can fetch the source code using different methods:
  - `wget`, for FTP/HTTP downloads
  - `scp`, to fetch the tarball using SSH/SCP
  - `svn`, for Subversion
  - `cvs`, for CVS
  - `git`, for Git
  - `hg`, for Mercurial
  - `bzr`, for Bazaar
  - `file`, for a local tarball
  - `local`, for a local directory
- In most cases, the fetching method is guessed by Buildroot using the `<pkg>_SITE` variable.
- Exceptions:
  - Git, Subversion or Mercurial repositories accessed over HTTP or SSH.
  - `file` and `local` methods
- In such cases, use `<pkg>_SITE_METHOD` explicitly.

Download methods examples

- Subversion repository accessed over HTTP:

```
CJSON_VERSION = 58
CJSON_SITE_METHOD = svn
CJSON_SITE = http://svn.code.sf.net/p/cjson/code
```

- Source code available in a local directory:

```
MYAPP_SITE = $(TOPDIR)/../apps/myapp
MYAPP_SITE_METHOD = local
```

  - The "*download*" will consist in copying the source code from the designated directory to the Buildroot per-package build directory.

Downloading more elements

- <pkg>_PATCH, a list of patches to download and apply before building the package. They are automatically applied by the package infrastructure.
- <pkg>_EXTRA_DOWNLOADS, a list of additional files to download together with the package source code. It is up to the package .mk file to do something with them.
- Two options:
  - Just a file name: assumed to be relative to <pkg>_SITE.
  - A full URL: downloaded over HTTP, FTP.
- Examples:

sysvinit.mk

```
SYSVINIT_PATCH = sysvinit_$(SYSVINIT_VERSION)dsf-13.1+squeeze1.diff.gz
```

perl.mk

```
PERL_CROSS_SITE = http://raw.github.com/arsv/perl-cross/releases
PERL_CROSS_SOURCE = perl-$(PERL_CROSS_BASE_VERSION)-cross-$(PERL_CROSS_VERSION).tar.gz
PERL_EXTRA_DOWNLOADS = $(PERL_CROSS_SITE)/$(PERL_CROSS_SOURCE)
```

Hash file

- In order to validate the integrity of downloaded files and license files, and make sure the user uses the version which was tested by the Buildroot developers, *cryptographic hashes* are used
- Each package may contain a file named `<package>.hash`, which gives the hashes of the files downloaded by the package.
- When present, the hashes for **all** files downloaded by the package must be documented.
- The *hash file* can also contain the hashes for the license files listed in `<pkg>_LICENSE_FILES`. This allows to detect changes in the license files.
- The syntax of the file is:

```
<hashtype>  <hash>  <file>
```

Note: the separator between fields is 2 spaces.

Hash file examples

package/perl/perl.hash

```
# Hashes from: https://www.cpan.org/src/5.0/perl-5.32.1.tar.xz.{md5,sha1,sha256}.txt
md5  7f104064b906ad8c7329ca5e409a32d7  perl-5.32.1.tar.xz
sha1  1fb4f710d139da1e1a3e1fa4eaba201fcaa8e18e  perl-5.32.1.tar.xz
sha256  57cc47c735c8300a8ce2fa0643507b44c4ae59012bfdad0121313db639e02309  perl-5.32.1.tar.xz

# Hashes from: https://github.com/arsv/perl-cross/releases/download/1.3.5/perl-cross-1.3.5.hash
sha256  91c66f6b2b99fccfd4fee14660b677380b0c98f9456359e91449798c2ad2ef25  perl-cross-1.3.5.tar.gz

# Locally calculated
sha256  dd90d4f42e4dcadf5a7c09eea0189d93c7b37ae560c91f0f6d5233ed3b9292a2  Artistic
sha256  d77d235e41d54594865151f4751e835c5a82322b0e87ace266567c3391a4b912  Copying
sha256  df6ad59aefea68676c38325f25f6707f026ddde6c71291b2ca231b6247859907  README
```

package/ipset/ipset.hash

```
# From http://ipset.netfilter.org/ipset-7.6.tar.bz2.md5sum.txt
md5 e107b679c3256af795261cece864d6d9  ipset-7.6.tar.bz2
# Calculated based on the hash above
sha256 0e7d44caa9c153d96a9b5f12644fbe35a632537a5a7f653792b72e53d9d5c2db  ipset-7.6.tar.bz2
# Locally calculated
sha256 231f7edcc7352d7734a96eef0b8030f77982678c516876fcb81e25b32d68564c  COPYING
```

Describing dependencies

- Dependencies expressed in `Config.in` do not enforce build order.
- The `<pkg>_DEPENDENCIES` variable is used to describe the dependencies of the current package.
- Packages listed in `<pkg>_DEPENDENCIES` are guaranteed to be built before the *configure* step of the current package starts.
- It can contain both target and host packages.
- It can be appended conditionally with additional dependencies.

python.mk

```
PYTHON_DEPENDENCIES = host-python libffi

ifeq ($(BR2_PACKAGE_PYTHON_READLINE),y)
PYTHON_DEPENDENCIES += readline
endif
```

Mandatory vs. optional dependencies

- Very often, software components have some **mandatory dependencies** and some **optional dependencies**, only needed for optional features.
- Handling mandatory dependencies in Buildroot consists in:
  - Using a `select` or `depends on` on the main package option in `Config.in`
  - Adding the dependency in `<pkg>_DEPENDENCIES`
- For optional dependencies, there are two possibilities:
  - Handle it automatically: in the `.mk` file, if the optional dependency is available, use it.
  - Handle it explicitly: add a package sub-option in the `Config.in` file.
- *Automatic* handling is usually preferred as it reduces the number of `Config.in` options, but it makes the possible dependency less visible to the user.

Dependencies: ntp example

- Mandatory dependency: libevent
- Optional dependency handled automatically: openssl

package/ntp/Config.in

```
config BR2_PACKAGE_NTP
        bool "ntp"
        select BR2_PACKAGE_LIBEVENT
[...]
```

package/ntp/ntp.mk

```
[...]
NTP_DEPENDENCIES = host-pkgconf libevent
[...]
ifeq ($(BR2_PACKAGE_OPENSSL),y)
NTP_CONF_OPTS += --with-crypto --enable-openssl-random
NTP_DEPENDENCIES += openssl
else
NTP_CONF_OPTS += --without-crypto --disable-openssl-random
endif
[...]
```

Dependencies: mpd example (1/2)

package/mpd/Config.in

```
menuconfig BR2_PACKAGE_MPD
        bool "mpd"
        depends on BR2_INSTALL_LIBSTDCPP
[...]
        select BR2_PACKAGE_BOOST
        select BR2_PACKAGE_LIBGLIB2
        select BR2_PACKAGE_LIBICONV if !BR2_ENABLE_LOCALE
[...]

config BR2_PACKAGE_MPD_FLAC
        bool "flac"
        select BR2_PACKAGE_FLAC
        help
          Enable flac input/streaming support.
          Select this if you want to play back FLAC files.
```

Dependencies: mpd example (2/2)

package/mpd/mpd.mk

```
MPD_DEPENDENCIES = host-pkgconf boost libglib2

[...]

ifeq ($(BR2_PACKAGE_MPD_FLAC),y)
MPD_DEPENDENCIES += flac
MPD_CONF_OPTS += --enable-flac
else
MPD_CONF_OPTS += --disable-flac
endif
```

Defining where to install (1)

- Target packages can install files to different locations:
  - To the *target* directory, `$(TARGET_DIR)`, which is what will be the target root filesystem.
  - To the *staging* directory, `$(STAGING_DIR)`, which is the compiler *sysroot*
  - To the *images* directory, `$(BINARIES_DIR)`, which is where final images are located.
- There are three corresponding variables, to define whether or not the package will install something to one of these locations:
  - `<pkg>_INSTALL_TARGET`, defaults to `YES`. If `YES`, then `<pkg>_INSTALL_TARGET_CMDS` will be called.
  - `<pkg>_INSTALL_STAGING`, defaults to `NO`. If `YES`, then `<pkg>_INSTALL_STAGING_CMDS` will be called.
  - `<pkg>_INSTALL_IMAGES`, defaults to `NO`. If `YES`, then `<pkg>_INSTALL_IMAGES_CMDS` will be called.

Defining where to install (2)

- A package for an application:
  - installs to `$(TARGET_DIR)` only
  - `<pkg>_INSTALL_TARGET` defaults to `YES`, so there is nothing to do
- A package for a shared library:
  - installs to both `$(TARGET_DIR)` and `$(STAGING_DIR)`
  - must set `<pkg>_INSTALL_STAGING = YES`
- A package for a pure header-based library, or a static-only library:
  - installs only to `$(STAGING_DIR)`
  - must set `<pkg>_INSTALL_TARGET = NO` and `<pkg>_INSTALL_STAGING = YES`
- A package installing a bootloader or kernel image:
  - installs to `$(BINARIES_DIR)`
  - must set `<pkg>_INSTALL_IMAGES = YES`

Defining where to install (3)

libyaml.mk

```
LIBYAML_INSTALL_STAGING = YES
```

eigen.mk

```
EIGEN_INSTALL_STAGING = YES
EIGEN_INSTALL_TARGET = NO
```

linux.mk

```
LINUX_INSTALL_IMAGES = YES
```

Describing actions for generic-package

- In a package using generic-package, only the download, extract and patch steps are implemented by the package infrastructure.
- The other steps should be described by the package .mk file:
  - <pkg>_CONFIGURE_CMDS, always called
  - <pkg>_BUILD_CMDS, always called
  - <pkg>_INSTALL_TARGET_CMDS, called when <pkg>_INSTALL_TARGET = YES, for target packages
  - <pkg>_INSTALL_STAGING_CMDS, called when <pkg>_INSTALL_STAGING = YES, for target packages
  - <pkg>_INSTALL_IMAGES_CMDS, called when <pkg>_INSTALL_IMAGES = YES, for target packages
  - <pkg>_INSTALL_CMDS, always called for host packages
- Packages are free to not implement any of these variables: they are all optional.

Describing actions: useful variables

Inside an action block, the following variables are often useful:

- `$(@D)` is the source directory of the package
- `$(MAKE)` to call `make`
- `$(MAKE1)` when the package doesn't build properly in parallel mode
- `$(TARGET_MAKE_ENV)` and `$(HOST_MAKE_ENV)`, to pass in the `$(MAKE)` environment to ensure the `PATH` is correct
- `$(TARGET_CONFIGURE_OPTS)` and `$(HOST_CONFIGURE_OPTS)` to pass `CC`, `LD`, `CFLAGS`, etc.
- `$(TARGET_DIR)`, `$(STAGING_DIR)`, `$(BINARIES_DIR)` and `$(HOST_DIR)`.

Describing actions: `iostat.mk` example

```
IOSTAT_VERSION = 2.2
IOSTAT_SITE = http://linuxinsight.com/sites/default/files
IOSTAT_LICENSE = GPL
IOSTAT_LICENSE_FILES = LICENSE

define IOSTAT_BUILD_CMDS
        $(MAKE) -C $(@D) $(TARGET_CONFIGURE_OPTS) \
                CFLAGS="$(TARGET_CFLAGS) -DHZ=100"
endef

define IOSTAT_INSTALL_TARGET_CMDS
        $(INSTALL) -D -m 0755 $(IOSTAT_DIR)/iostat \
                $(TARGET_DIR)/usr/bin/iostat
endef

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

Describing actions: libzlib.mk example

```
LIBZLIB_VERSION = 1.2.11
LIBZLIB_SOURCE = zlib-$(LIBZLIB_VERSION).tar.xz
LIBZLIB_SITE = http://www.zlib.net
LIBZLIB_INSTALL_STAGING = YES

define LIBZLIB_CONFIGURE_CMDS
        (cd $(@D); rm -rf config.cache; \
                $(TARGET_CONFIGURE_ARGS) \
                $(TARGET_CONFIGURE_OPTS) \
                CFLAGS="$(TARGET_CFLAGS) $(LIBZLIB_PIC)" \
                ./configure \
                $(LIBZLIB_SHARED) \
                --prefix=/usr \
        )
endef

define LIBZLIB_BUILD_CMDS
        $(TARGET_MAKE_ENV) $(MAKE1) -C $(@D)
endef

define LIBZLIB_INSTALL_STAGING_CMDS
        $(TARGET_MAKE_ENV) $(MAKE1) -C $(@D) DESTDIR=$(STAGING_DIR) LDCONFIG=true install
endef

define LIBZLIB_INSTALL_TARGET_CMDS
        $(TARGET_MAKE_ENV) $(MAKE1) -C $(@D) DESTDIR=$(TARGET_DIR) LDCONFIG=true install
endef

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

List of package infrastructures (1/2)

- `generic-package`, for packages not using a well-known build system. Already covered.
- `autotools-package`, for *autotools* based packages, covered later.
- `python-package`, for *distutils* and *setuptools* based Python packages
- `perl-package`, for *Perl* packages
- `luarocks-package`, for Lua packages hosted on `luarocks.org`
- `cmake-package`, for *CMake* based packages
- `waf-package`, for *Waf* based packages
- `qmake-package`, for *QMake* based packages

List of package infrastructures (2/2)

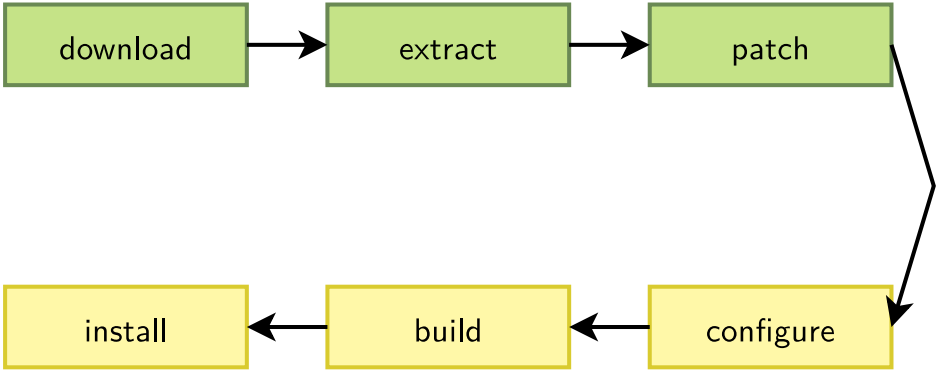
- `golang-package`, for packages written in Go
- `meson-package`, for packages using the Meson build system
- `cargo-package`, for packages written in Rust
- `kconfig-package`, to be used in conjunction with `generic-package`, for packages that use the *kconfig* configuration system
- `kernel-module-package`, to be used in conjunction with another package infrastructure, for packages that build kernel modules
- `rebar-package` for *Erlang* packages that use the *rebar* build system
- `virtual-package` for *virtual* packages, covered later.

autotools-package infrastructure

The autotools-package infrastructure: basics

- The autotools-package infrastructure inherits from generic-package and is specialized to handle *autotools* based packages.
- It provides a default implementation of:
  - <pkg>_CONFIGURE_CMDS. Calls the ./configure script with appropriate environment variables and arguments.
  - <pkg>_BUILD_CMDS. Calls make.
  - <pkg>_INSTALL_TARGET_CMDS, <pkg>_INSTALL_STAGING_CMDS and <pkg>_INSTALL_CMDS. Call make install with the appropriate DESTDIR.
- A normal *autotools* based package therefore does not need to describe any action: only metadata about the package.

The autotools-package: steps



The autotools-package infrastructure: variables

- It provides additional variables that can be defined by the package:
  - <pkg>_CONF_ENV to pass additional values in the environment of the ./configure script.
  - <pkg>_CONF_OPTS to pass additional options to the ./configure script.
  - <pkg>_INSTALL_OPTS, <pkg>_INSTALL_STAGING_OPTS and <pkg>_INSTALL_TARGET_OPTS to adjust the *make* target and options used for the installation.
  - <pkg>_AUTORECONF. Defaults to NO, can be set to YES if regenerating Makefile.in files and configure script is needed. The infrastructure will automatically make sure *autoconf*, *automake*, *libtool* are built.
  - <pkg>_GETTEXTIZE. Defaults to NO, can be set to YES to *gettextize* the package. Only makes sense if <pkg>_AUTORECONF = YES.

Canonical autotools-package example

libyaml.mk

```
################################################################################
#
# libyaml
#
################################################################################

LIBYAML_VERSION = 0.2.5
LIBYAML_SOURCE = yaml-$(LIBYAML_VERSION).tar.gz
LIBYAML_SITE = http://pyyaml.org/download/libyaml
LIBYAML_INSTALL_STAGING = YES
LIBYAML_LICENSE = MIT
LIBYAML_LICENSE_FILES = License
LIBYAML_CPE_ID_VENDOR = pyyaml

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

More complicated autotools-package example

```
GNUPG2_VERSION = 2.2.25
GNUPG2_SOURCE = gnupg-$(GNUPG2_VERSION).tar.bz2
GNUPG2_SITE = https://gnupg.org/ftp/gcrypt/gnupg
GNUPG2_LICENSE = GPL-3.0+
GNUPG2_LICENSE_FILES = COPYING
GNUPG2_CPE_ID_VENDOR = gnupg
GNUPG2_CPE_ID_PRODUCT = gnupg
GNUPG2_DEPENDENCIES = zlib libgpg-error libgcrypt libassuan \
        libksba libnpth host-pkgconf \
        $(if $(BR2_PACKAGE_LIBICONV),libiconv)

ifeq ($(BR2_PACKAGE_BZIP2),y)
GNUPG2_CONF_OPTS += --enable-bzip2 --with-bzip2=$(STAGING_DIR)
GNUPG2_DEPENDENCIES += bzip2
else
GNUPG2_CONF_OPTS += --disable-bzip2
endif

ifeq ($(BR2_PACKAGE_GNUTLS),y)
GNUPG2_CONF_OPTS += --enable-gnutls
GNUPG2_DEPENDENCIES += gnutls
else
GNUPG2_CONF_OPTS += --disable-gnutls
endif
```

```
[...]

ifeq ($(BR2_PACKAGE_LIBUSB),y)
GNUPG2_CONF_ENV += CPPFLAGS="$(TARGET_CPPFLAGS)
                             -I$(STAGING_DIR)/usr/include/libusb-1.0"
GNUPG2_CONF_OPTS += --enable-ccid-driver
GNUPG2_DEPENDENCIES += libusb
else
GNUPG2_CONF_OPTS += --disable-ccid-driver
endif

ifeq ($(BR2_PACKAGE_READLINE),y)
GNUPG2_CONF_OPTS += --with-readline=$(STAGING_DIR)
GNUPG2_DEPENDENCIES += readline
else
GNUPG2_CONF_OPTS += --without-readline
endif

$(eval $(autotools-package))
```

Target vs. host packages

Host packages

- As explained earlier, most packages in Buildroot are cross-compiled for the target. They are called **target packages**.
- Some packages however may need to be built natively for the build machine, they are called **host packages**. They can be needed for a variety of reasons:
  - Needed as a tool to build other things for the target. Buildroot wants to limit the number of host utilities required to be installed on the build machine, and wants to ensure the proper version is used. So it builds some host utilities by itself.
  - Needed as a tool to interact, debug, reflash, generate images, or other activities around the build itself.
  - Version dependencies: building a Python interpreter for the target needs a Python interpreter of the same version on the host.

Target vs. host: package name and variable prefixes

- Each package infrastructure provides a `<foo>-package` macro and a `host-<foo>-package` macro.
- For a given package in `package/baz/baz.mk`, `<foo>-package` will create a package named `baz` and `host-<foo>-package` will create a package named `host-baz`.
- `<foo>-package` will use the variables prefixed with `BAZ_`
- `host-<foo>-package` will use the variables prefixed with `HOST_BAZ_`

Target vs. host: variable inheritance

- For many variables, when HOST_BAZ_<var> is not defined, the package infrastructure *inherits* from BAZ_<var> instead.
  - True for <PKG>_SOURCE, <PKG>_SITE, <PKG>_VERSION, <PKG>_LICENSE, <PKG>_LICENSE_FILES, etc.
  - Defining <PKG>_SITE is sufficient, defining HOST_<PKG>_SITE is not needed.
  - It is still possible to override the value specifically for the host variant, but this is rarely needed.
- But not for all variables, especially commands
  - E.g. HOST_<PKG>_BUILD_CMDS is not inherited from <PKG>_BUILD_CMDS

Example 1: a pure build utility

- *bison*, a general-purpose parser generator.
- Purely used as build dependency in packages
  - `FBSET_DEPENDENCIES = host-bison host-flex`
- No `Config.in.host`, not visible in `menuconfig`.

package/bison/bison.mk

```
BISON_VERSION = 3.7.1
BISON_SOURCE = bison-$(BISON_VERSION).tar.xz
BISON_SITE = $(BR2_GNU_MIRROR)/bison
BISON_LICENSE = GPL-3.0+
BISON_LICENSE_FILES = COPYING
BISON_CPE_ID_VENDOR = gnu
# parallel build issue in examples/c/reccalc/
BISON_MAKE = $(MAKE1)
HOST_BISON_DEPENDENCIES = host-m4
HOST_BISON_CONF_OPTS = --enable-relocatable
HOST_BISON_CONF_ENV = ac_cv_libtextstyle=no

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

Example 2: filesystem manipulation tool

- `fatcat`, is designed to manipulate FAT filesystems, in order to explore, extract, repair, recover and forensic them.
- Not used as a build dependency of another package → visible in `menuconfig`.

package/fatcat/Config.in.host

```
config BR2_PACKAGE_HOST_FATCAT
        bool "host fatcat"
        help
          Fatcat is designed to manipulate FAT filesystems, in order
          to explore, extract, repair, recover and forensic them. It
          currently supports FAT12, FAT16 and FAT32.

          https://github.com/Gregwar/fatcat
```

package/fatcat/fatcat.mk

```
FATCAT_VERSION = 1.1.0
FATCAT_SITE = $(call github,Gregwar,fatcat,v$(FATCAT_VERSION))
FATCAT_LICENSE = MIT
FATCAT_LICENSE_FILES = LICENSE

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

Example 3: target and host of the same package

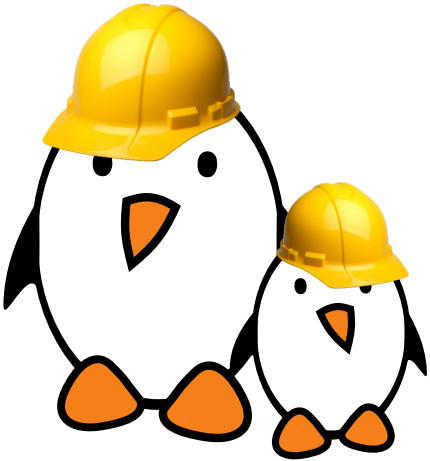
package/e2tools/e2tools.mk

```
E2TOOLS_VERSION = 0.0.16.4
E2TOOLS_SITE = $(call github,ndim,e2tools,v$(E2TOOLS_VERSION))

# Source coming from GitHub, no configure included.
E2TOOLS_AUTORECONF = YES
E2TOOLS_LICENSE = GPL-2.0
E2TOOLS_LICENSE_FILES = COPYING
E2TOOLS_DEPENDENCIES = e2fsprogs
E2TOOLS_CONF_ENV = LIBS="-lpthread"
HOST_E2TOOLS_DEPENDENCIES = host-e2fsprogs
HOST_E2TOOLS_CONF_ENV = LIBS="-lpthread"

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

Practical lab - New packages in Buildroot



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