Buildroot: a deep dive into the core

Thomas Petazzoni
Free Electrons
thomas.petazzoni@free-electrons.com

Thomas Petazzoni

- CTO and Embedded Linux engineer at Free Electrons
  - Embedded Linux and Android **development**: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux, Linux driver development, Android system and Yocto/OpenEmbedded **training courses**, with materials freely available under a Creative Commons license.
  - `http://free-electrons.com`
- Contributions
  - **Kernel support for the Marvell Armada** ARM SoCs from Marvell
  - Major contributor to **Buildroot**, an open-source, simple and fast embedded Linux build system
- Living in **Toulouse**, south west of France

Agenda

1. Quick introduction about Buildroot
2. Source tree and output tree
3. Configuration system
4. From *make* to the generic package infrastructure
5. Specialized package infrastructures
6. Toolchain support
7. Root filesystem image generation
8. Overall build logic

Buildroot at a glance

- An **embedded Linux build system**, builds from source:
  - cross-compilation toolchain
  - root filesystem with many libraries/applications, cross-built
  - kernel and bootloader images
- **Fast**, simple root filesystem in minutes
- **Easy** to use and understand: kconfig and make
- **Small** root filesystem, default 2 MB
- More than **1200 packages** available
- Generates filesystem images, not a distribution
- Vendor neutral
- Active community, regular releases
- Started in 2001, oldest still maintained build system
- `http://buildroot.org`

Demonstration

A demonstration is worth many slides!

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- top-level `Makefile`, handles the configuration and general orchestration of the build

Source tree

- Makefile
- Config.in
- arch/
- toolchain/
- system/
- linux/
- package/
- fs/
- boot/
- configs/
- board/
- support/
- docs/

- top-level Config.in, main/general options. Includes many other Config.in files

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- `Config.in.*` files defining the architecture variants (processor type, ABI, floating point, etc.)
- `Config.in`, `Config.in.arm`, `Config.in.x86`, `Config.in.microblaze`, etc.

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- packages for generating or using toolchains
- `toolchain/` virtual package that depends on either `toolchain-buildroot` or `toolchain-external`
- `toolchain-buildroot/` virtual package to build the internal toolchain
- `toolchain-external/` package to handle external toolchains

Source tree

- Makefile
- Config.in
- arch/
- toolchain/
- system/
- linux/
- package/
- fs/
- boot/
- configs/
- board/
- support/
- docs/

- skeleton/ the rootfs skeleton
- Config.in, options for system-wide features like init system, /dev handling, etc.

Source tree

- Makefile
- Config.in
- arch/
- toolchain/
- system/
- linux/
- package/
- fs/
- boot/
- configs/
- board/
- support/
- docs/

- linux.mk, the Linux kernel package

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- all the userspace packages (1200+)
- `busybox/`, `gcc/`, `qt5/`, etc.
- `pkg-generic.mk`, core package infrastructure
- `pkg-cmake.mk`, `pkg-autotools.mk`, `pkg-perl.mk`, etc. Specialized package infrastructures

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- logic to generate filesystem images in various formats
- `common.mk`, common logic
- `cpio/`, `ext2/`, `squashfs/`, `tar/`, `ubifs/`, etc.

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- bootloader packages
- `at91bootstrap3/`, `barebox/`, `grub/`, `syslinux/`, `uboot/`, etc.

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- default configuration files for various platforms
- similar to kernel defconfigs
- `atmel_xplained_defconfig`, `beaglebone_defconfig`, `raspberrypi_defconfig`, etc.

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`

- board-specific files (kernel configuration files, kernel patches, image flashing scripts, etc.)
- typically go together with a *defconfig* in `configs/`

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
  - misc utilities (kconfig code, libtool patches, download helpers, and more.)
- `docs/`

Source tree

- `Makefile`
- `Config.in`
- `arch/`
- `toolchain/`
- `system/`
- `linux/`
- `package/`
- `fs/`
- `boot/`
- `configs/`
- `board/`
- `support/`
- `docs/`
  - Buildroot documentation
  - 90 pages PDF document

Output tree

- output/
  - build/
  - host/
  - staging/
  - target/
  - images/
  - graphs/
  - legal-info/

Output tree

- `output/`
  - `build/`
  - `host/`
  - `staging/`
  - `target/`
  - `images/`
  - `graphs/`
  - `legal-info/`

- Global output directory
- Can be customized for out-of-tree build by passing `O=<dir>`
- Variable: `O` (as passed on the command line)
- Variable: `BASE_DIR` (as an absolute path)

Output tree

- output/
  - build/
    - buildroot-config/
    - busybox-1.22.1/
    - host-pkgconf-0.8.9/
    - kmod-1.18/
    - build-time.log
  - host/
  - staging/
  - target/
  - images/
  - graphs/
  - legal-info/

- Where all source tarballs are extracted
- Where the build of each package takes place
- In addition to the package sources and object files, *stamp* files are created by Buildroot
- Variable: `BUILD_DIR`

Output tree

- output/
  - build/
  - host/
    - usr/lib
    - usr/bin
    - usr/sbin
    - usr/<tuple>/sysroot/bin
    - usr/<tuple>/sysroot/lib
    - usr/<tuple>/sysroot/usr/lib
    - usr/<tuple>/sysroot/usr/bin
  - staging/
  - target/
  - images/
  - graphs/
  - legal-info/

- Contains both the tools built for the host (cross-compiler, etc.) and the *sysroot* of the toolchain
- Variable: HOST_DIR
- Host tools are directly in host/usr
- The *sysroot* is in host/<tuple>/sysroot/usr
- Variable for the *sysroot*: STAGING_DIR

Output tree

- `output/`
  - `build/`
  - `host/`
  - `staging/`
  - `target/`
  - `images/`
  - `graphs/`
  - `legal-info/`

- Just a symbolic link to the *sysroot*, i.e to `host/<tuple>/sysroot/`.
- Available for convenience

Output tree

- output/
  - build/
  - host/
  - staging/
  - target/
    - bin/
    - etc/
    - lib/
    - usr/bin/
    - usr/lib/
    - usr/share/
    - usr/sbin/
    - THIS_IS_NOT_YOUR_ROOT_FILESYSTEM
    - ...
  - images/
  - graphs/
  - legal-info/

- The target root filesystem
- Usual Linux hierarchy
- Not completely ready for the target: permissions, device files, etc.
- Buildroot does not run as root: all files are owned by the user running Buildroot, not *setuid*, etc.
- Used to generate the final root filesystem images in `images/`
- Variable: `TARGET_DIR`

Output tree

- output/
  - build/
  - host/
  - staging/
  - target/
  - images/
    - zImage
    - armada-370-mirabox.dtb
    - rootfs.tar
    - rootfs.ubi
  - graphs/
  - legal-info/

- Contains the final images: kernel image, bootloader image, root filesystem image(s)
- Variable: BINARIES_DIR

Output tree

- output/
  - build/
  - host/
  - staging/
  - target/
  - images/
  - graphs/
  - legal-info/

- Visualization of Buildroot operation: dependencies between packages, time to build the different packages
- make graph-depends
- make graph-build

Output tree

- output/
  - build/
  - host/
  - staging/
  - target/
  - images/
  - graphs/
  - legal-info/
    - manifest.csv
    - host-manifest.csv
    - licenses.txt
    - licenses/
    - sources/
    - ...

- Legal information: license of all packages, and their source code, plus a licensing manifest
- Useful for license compliance
- `make legal-info`
- Variable: `LEGAL_INFO_DIR`

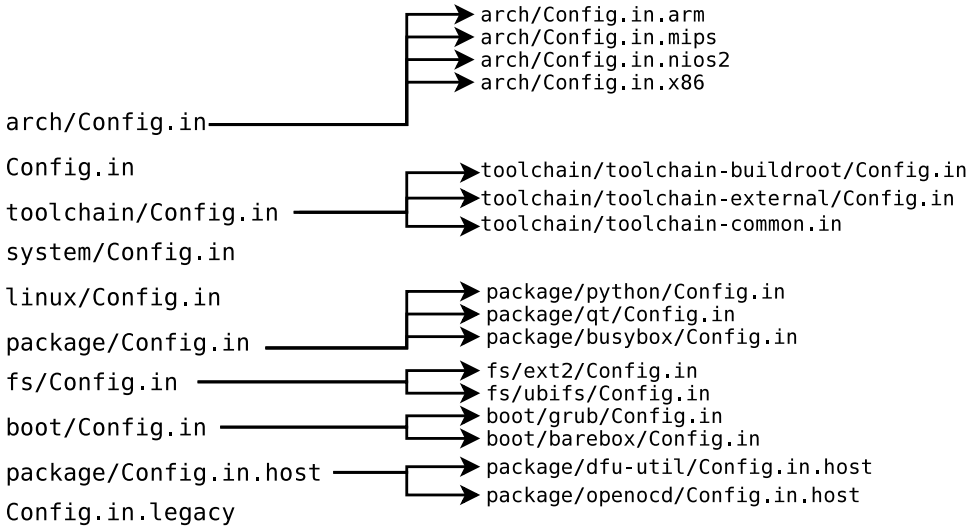
Configuration system

- Uses, almost unchanged, the *kconfig* code from the kernel, in `support/kconfig` (variable `CONFIG`)
- *kconfig* tools are built in `$(BUILD_DIR)/buildroot-config/`
- The main `Config.in` file, passed to menuconfig/xconfig, is at the top-level of the Buildroot source tree
- Config file saved as `.config` in the output directory (except for in-tree builds)
- `.config` included in `Makefile` → config values readily available as make variables.

```
CONFIG_CONFIG_IN = Config.in
CONFIG = support/kconfig
BR2_CONFIG = $(CONFIG_DIR)/.config

-include $(BR2_CONFIG)

$(BUILD_DIR)/buildroot-config/%onf:
        mkdir -p $(@D)/lxdialog
        $(MAKE) ... -C $(CONFIG) -f Makefile.br $(@F)

menuconfig: $(BUILD_DIR)/buildroot-config/mconf outputmakefile
        @mkdir -p $(BUILD_DIR)/buildroot-config
        @$(COMMON_CONFIG_ENV) $< $(CONFIG_CONFIG_IN)
```

Configuration hierarchy

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



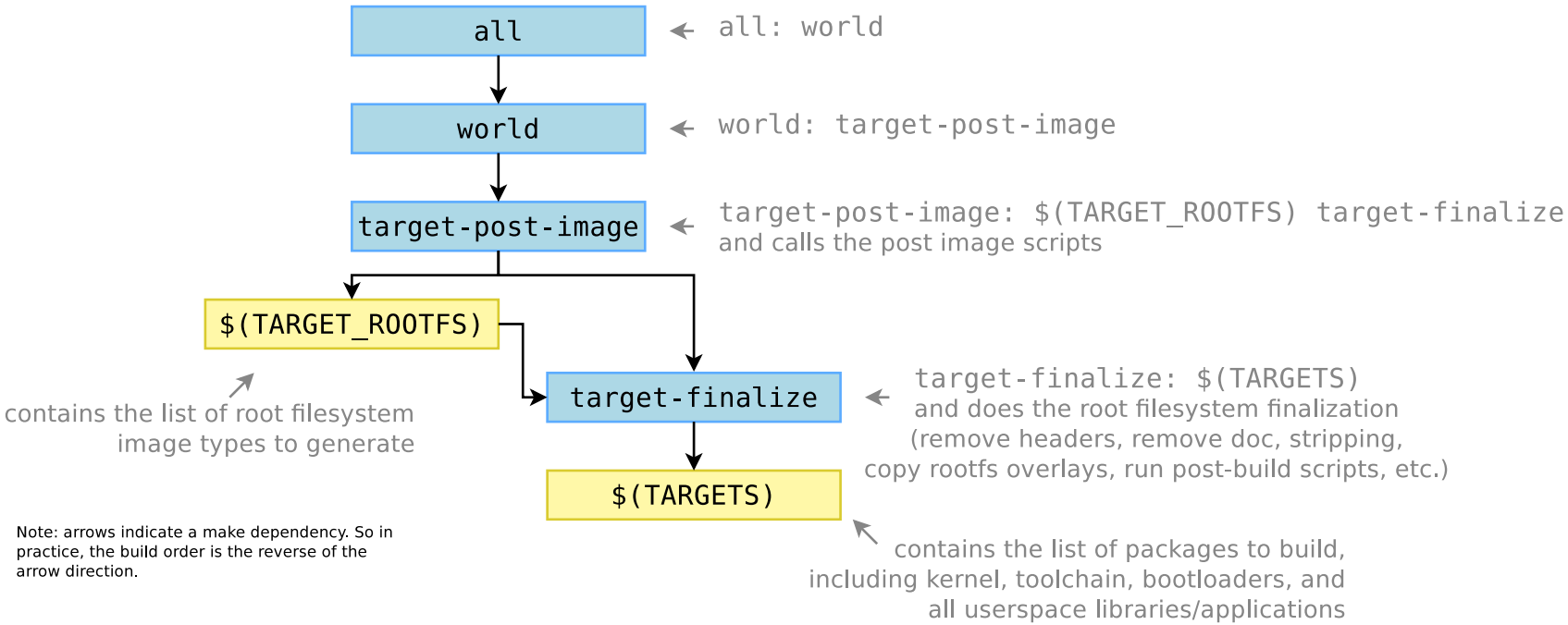
Example of package `Config.in`

package/httping/Config.in

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

config BR2_PACKAGE_HTTPING
        bool "httping"
        depends on BR2_USE_WCHAR
        select BR2_PACKAGE_GETTEXT if BR2_NEEDS_GETTEXT
        help
          Httping is like 'ping' but for http-requests.
          ...
          http://www.vanheusden.com/httping/

if BR2_PACKAGE_HTTPING

config BR2_PACKAGE_HTTPING_TFO
        bool "TCP Fast Open (TFO) support"

endif
```

- All packages have a main option named `BR2_PACKAGE_<pkg>`
- Sub-options are used for additional tuning of the package
- `depends on` to depend on toolchain features or *big* features (e.g. X.org)
- `select` used for most library dependencies, to make them transparent to the user

When you run `make`...



Where is $(TARGETS) filled?

Part of package/pkg-generic.mk

```
#  argument 1 is the lowercase package name
#  argument 2 is the uppercase package name, including a HOST_ prefix
#             for host packages

define inner-generic-package
 ...
$(2)_KCONFIG_VAR = BR2_PACKAGE_$(2)
 ...
ifeq ($$($$($(2)_KCONFIG_VAR)),y)
TARGETS += $(1)
endif # $(2)_KCONFIG_VAR

endef # inner-generic-package
```

- Adds the lowercase name of an enabled package as a make target to the $(TARGETS) variable
- package/pkg-generic.mk is really the core of the package infrastructure

Example of `yavta.mk`

```
YAVTA_VERSION = 82ff2efdb9787737b9f21b6f4759f077c827b238
YAVTA_SITE = git://git.ideasonboard.org/yavta.git
YAVTA_LICENSE = GPLv2+
YAVTA_LICENSE_FILES = COPYING.GPL

define YAVTA_BUILD_CMDS
        $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D)
endef

define YAVTA_INSTALL_TARGET_CMDS
        $(INSTALL) -m 0755 -D $(@D)/yavta $(TARGET_DIR)/usr/bin/yavta
endef

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

- A package is just a definition of variables starting with the package name
  - Some variables are simple values: version, site, license, license files
  - Some variables contain commands: build and installation commands

Example of `zlib.mk` (1/2)

```
ZLIB_VERSION = 1.2.8
ZLIB_SOURCE = zlib-$(ZLIB_VERSION).tar.xz
ZLIB_SITE = http://downloads.sourceforge.net/project/libpng/zlib/$(ZLIB_VERSION)
ZLIB_LICENSE = zlib license
ZLIB_LICENSE_FILES = README
ZLIB_INSTALL_STAGING = YES
...
define ZLIB_CONFIGURE_CMDS
        (cd $(@D); rm -rf config.cache; \
                $(TARGET_CONFIGURE_ARGS) \
                $(TARGET_CONFIGURE_OPTS) \
                CFLAGS="$(TARGET_CFLAGS) $(ZLIB_PIC)" \
                ./configure \
                $(ZLIB_SHARED) \
                --prefix=/usr \
        )
endef

define ZLIB_BUILD_CMDS
        $(MAKE1) -C $(@D)
endef

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

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

Example of `zlib.mk` (2/2)

```
define HOST_ZLIB_CONFIGURE_CMDS
        (cd $(@D); rm -rf config.cache; \
                $(HOST_CONFIGURE_ARGS) \
                $(HOST_CONFIGURE_OPTS) \
                ./configure \
                --prefix="$(HOST_DIR)/usr" \
                --sysconfdir="$(HOST_DIR)/etc" \
        )
endef

define HOST_ZLIB_BUILD_CMDS
        $(MAKE1) -C $(@D)
endef

define HOST_ZLIB_INSTALL_CMDS
        $(MAKE1) -C $(@D) LDCONFIG=true install
endef

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

Diving into pkg-generic.mk

- The package/pkg-generic.mk file is divided in two main parts:
  - Definition of the actions done in each step of a package build process. Done through *stamp file targets*.
  - Definition of the inner-generic-package, generic-package and host-generic-package macros, that define the sequence of actions, as well as all the variables needed to handle the build of a package.

Definition of the actions: code

```
$(BUILD_DIR)/%/.stamp_downloaded:
        # Do some stuff here
        $(Q)touch $@

$(BUILD_DIR)/%/.stamp_extracted:
        # Do some stuff here
        $(Q)touch $@

$(BUILD_DIR)/%/.stamp_patched:
        # Do some stuff here
        $(Q)touch $@

$(BUILD_DIR)/%/.stamp_configured:
        # Do some stuff here
        $(Q)touch $@

$(BUILD_DIR)/%/.stamp_built:
        # Do some stuff here
        $(Q)touch $@
```

```
$(BUILD_DIR)/%/.stamp_host_installed:
        # Do some stuff here
        $(Q)touch $@

$(BUILD_DIR)/%/.stamp_staging_installed:
        # Do some stuff here
        $(Q)touch $@

$(BUILD_DIR)/%/.stamp_images_installed:
        # Do some stuff here
        $(Q)touch $@

$(BUILD_DIR)/%/.stamp_target_installed:
        # Do some stuff here
        $(Q)touch $@
```

- `$(BUILD_DIR)/%/` → build directory of any package
- a *make* target depending on one stamp file will trigger the corresponding action
- the *stamp file* prevents the action from being re-executed

Action example 1: download

```
# Retrieve the archive
$(BUILD_DIR)/%/.stamp_downloaded:
        $(foreach hook,$($(PKG)_PRE_DOWNLOAD_HOOKS),$(call $(hook))$(sep))
        [...]
        $(if $($(PKG)_SOURCE),$(call DOWNLOAD,$($(PKG)_SITE:/=)/$($(PKG)_SOURCE)))
        $(foreach p,$($(PKG)_EXTRA_DOWNLOADS),$(call DOWNLOAD,$($(PKG)_SITE:/=)/$(p))$(sep))
        $(foreach p,$($(PKG)_PATCH),\
                $(if $(findstring ://,$(p)),\
                        $(call DOWNLOAD,$(p)),\
                        $(call DOWNLOAD,$($(PKG)_SITE:/=)/$(p))\
                )\
        $(sep))
        $(foreach hook,$($(PKG)_POST_DOWNLOAD_HOOKS),$(call $(hook))$(sep))
        $(Q)mkdir -p $(@D)
        $(Q)touch $@
```

- Step handled by the package infrastructure
- In all *stamp file targets*, `PKG` is the upper case name of the package. So when used for Busybox, `$($(PKG)_SOURCE)` is the value of `BUSYBOX_SOURCE`.
- *Hooks*: make macros called before and after each step.
- Downloads the files mentioned in `<pkg>_SOURCE`, `<pkg>_EXTRA_DOWNLOADS` and `<pkg>_PATCH`.

Action example 2: build

```
# Build
$(BUILD_DIR)/%/.stamp_built::
        @$(call step_start,build)
        @$(call MESSAGE,"Building")
        $(foreach hook,$($(PKG)_PRE_BUILD_HOOKS),$(call $(hook))$(sep))
        +$($(PKG)_BUILD_CMDS)
        $(foreach hook,$($(PKG)_POST_BUILD_HOOKS),$(call $(hook))$(sep))
        $(Q)touch $@
        @$(call step_end,build)
```

- Step handled by the package, by defining a value for `<pkg>_BUILD_CMDS`.
- Same principle of *hooks*
- `step_start` and `step_end` are part of instrumentation to measure the duration of each step (and other actions)

The generic-package macro

- Packages built for the target:

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

- Packages built for the host:

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

- In `package/zlib/zlib.mk`:

```
ZLIB_... = ...

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

- Leads to:

```
$(call inner-generic-package,zlib,ZLIB,ZLIB,target)
$(call inner-generic-package,host-zlib,HOST_ZLIB,ZLIB,host)
```

inner-generic-package: defining variables

Macro code

```
$(2)_TYPE     =  $(4)
$(2)_NAME     =  $(1)
$(2)_RAWNAME =  $$(patsubst host-%,%,$(1))

$(2)_BASE_NAME = $(1)-$$($(2)_VERSION)
$(2)_DIR       = $$(BUILD_DIR)/$$($(2)_BASE_NAME)



ifndef $(2)_SOURCE
 ifdef $(3)_SOURCE
  $(2)_SOURCE = $$($(3)_SOURCE)
 else
  $(2)_SOURCE ?=
    $$($(2)_RAWNAME)-$$($(2)_VERSION).tar.gz
 endif
endif

ifndef $(2)_SITE
 ifdef $(3)_SITE
  $(2)_SITE = $$($(3)_SITE)
 endif
endif

...
```

Expanded for host-zlib

```
HOST_ZLIB_TYPE     = host
HOST_ZLIB_NAME     = host-zlib
HOST_ZLIB_RAWNAME = zlib

HOST_ZLIB_BASE_NAME =
  host-zlib-$(HOST_ZLIB_VERSION)
HOST_ZLIB_DIR       =
  $(BUILD_DIR)/host-zlib-$(HOST_ZLIB_VERSION)

ifndef HOST_ZLIB_SOURCE
 ifdef ZLIB_SOURCE
  HOST_ZLIB_SOURCE = $(ZLIB_SOURCE)
 else
  HOST_ZLIB_SOURCE ?=
   zlib-$(HOST_ZLIB_VERSION).tar.gz
 endif
endif

ifndef HOST_ZLIB_SITE
 ifdef ZLIB_SITE
  HOST_ZLIB_SITE = $(ZLIB_SITE)
 endif
endif

...
```

inner-generic-package: dependencies

```
ifeq ($(4),host)
$(2)_DEPENDENCIES ?= $$(filter-out  host-toolchain $(1),\
        $$(patsubst host-host-%,host-%,$$(addprefix host-,$$($(3)_DEPENDENCIES))))
endif
```

- Dependencies of host packages, if not explicitly specified, are derived from the dependencies of the target package, by adding a `host-` prefix to each dependency.
  - If a package `foo` defines `FOO_DEPENDENCIES = bar baz host-buzz`, then the `host-foo` package will have `host-bar`, `host-baz` and `host-buzz` in its dependencies.

```
ifeq ($(4),target)
ifeq ($$($(2)_ADD_TOOLCHAIN_DEPENDENCY),YES)
$(2)_DEPENDENCIES += toolchain
endif
endif
```

- Adding the `toolchain` dependency to target packages. Except for some specific packages (e.g. C library).

inner-generic-package: stamp files

```
$(2)_TARGET_INSTALL_TARGET =    $$($(2)_DIR)/.stamp_target_installed
$(2)_TARGET_INSTALL_STAGING =   $$($(2)_DIR)/.stamp_staging_installed
$(2)_TARGET_INSTALL_IMAGES =    $$($(2)_DIR)/.stamp_images_installed
$(2)_TARGET_INSTALL_HOST =      $$($(2)_DIR)/.stamp_host_installed
$(2)_TARGET_BUILD =             $$($(2)_DIR)/.stamp_built
$(2)_TARGET_CONFIGURE =         $$($(2)_DIR)/.stamp_configured
$(2)_TARGET_RSYNC =             $$($(2)_DIR)/.stamp_rsynced
$(2)_TARGET_RSYNC_SOURCE =      $$($(2)_DIR)/.stamp_rsync_sourced
$(2)_TARGET_PATCH =             $$($(2)_DIR)/.stamp_patched
$(2)_TARGET_EXTRACT =           $$($(2)_DIR)/.stamp_extracted
$(2)_TARGET_SOURCE =            $$($(2)_DIR)/.stamp_downloaded
$(2)_TARGET_DIRCLEAN =          $$($(2)_DIR)/.stamp_dircleaned
```

- Defines shortcuts to reference the stamp files

```
$$($(2)_TARGET_INSTALL_TARGET):         PKG=$(2)
$$($(2)_TARGET_INSTALL_STAGING):        PKG=$(2)
$$($(2)_TARGET_INSTALL_IMAGES):         PKG=$(2)
$$($(2)_TARGET_INSTALL_HOST):           PKG=$(2)
[...]
```

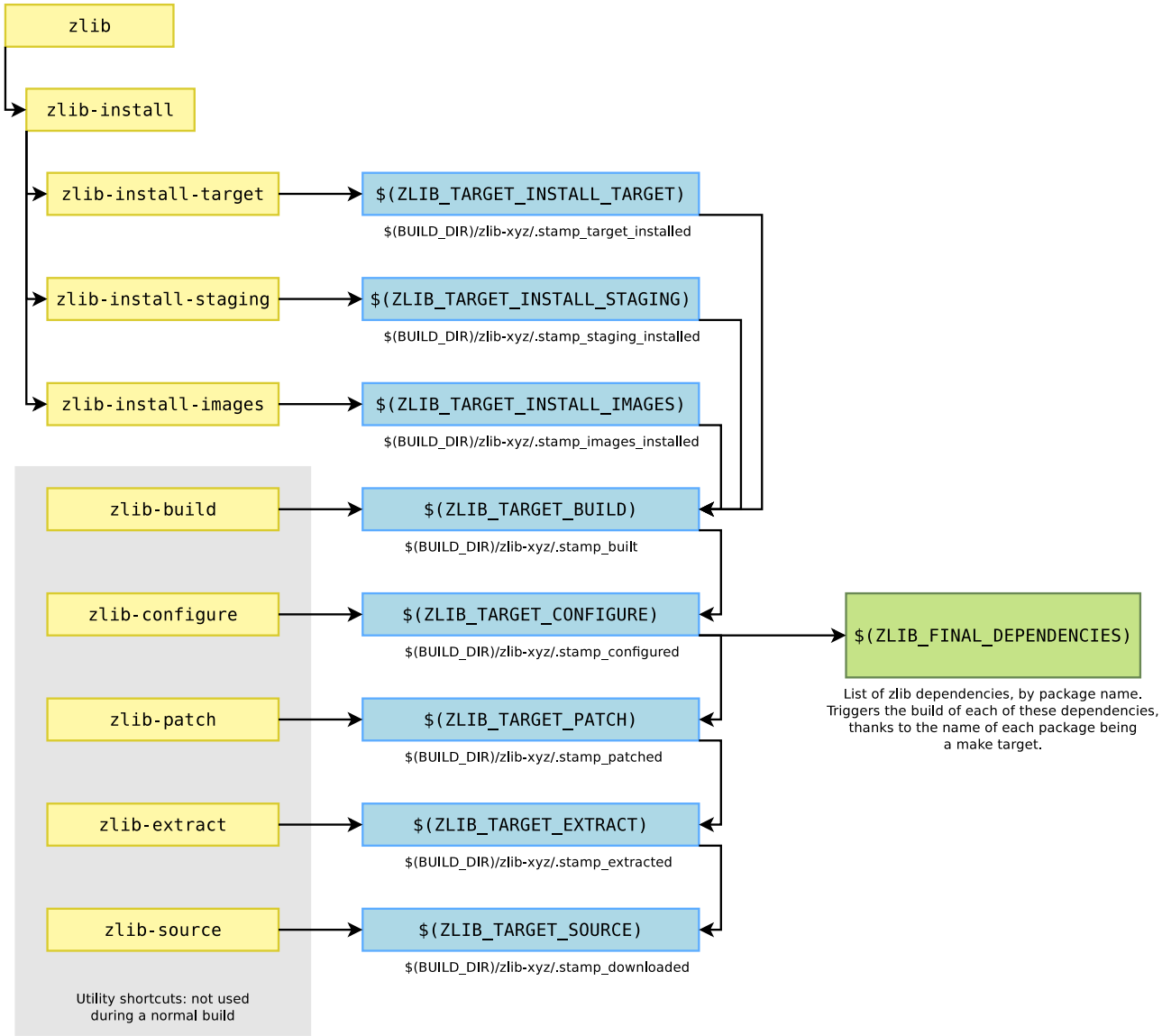
- Pass variables to the stamp file targets, especially PKG

inner-generic-package: sequencing

Step sequencing for target packages

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

$(1)-install:           $(1)-install-staging $(1)-install-target $(1)-install-images

$(1)-install-target:            $$($(2)_TARGET_INSTALL_TARGET)
$$($(2)_TARGET_INSTALL_TARGET): $$($(2)_TARGET_BUILD)

$(1)-build:             $$($(2)_TARGET_BUILD)
$$($(2)_TARGET_BUILD):  $$($(2)_TARGET_CONFIGURE)

$(1)-configure:                 $$($(2)_TARGET_CONFIGURE)
$$($(2)_TARGET_CONFIGURE):      | $$($(2)_FINAL_DEPENDENCIES)
$$($(2)_TARGET_CONFIGURE):      $$($(2)_TARGET_PATCH)

$(1)-patch:             $$($(2)_TARGET_PATCH)
$$($(2)_TARGET_PATCH):  $$($(2)_TARGET_EXTRACT)

$(1)-extract:                   $$($(2)_TARGET_EXTRACT)
$$($(2)_TARGET_EXTRACT):        $$($(2)_TARGET_SOURCE)

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

$$($(2)_TARGET_SOURCE): | dirs prepare
$$($(2)_TARGET_SOURCE): | dependencies
```

inner-generic-package: sequencing diagram



Example of package build

```
>>> zlib 1.2.8 Downloading
... here it wgets the tarball ...

>>> zlib 1.2.8 Extracting
xzcat /home/thomas/dl/zlib-1.2.8.tar.xz | tar ...

>>> zlib 1.2.8 Patching

>>> zlib 1.2.8 Configuring
(cd /home/thomas/projets/buildroot/output/build/zlib-1.2.8;
   ...
   ./configure --shared --prefix=/usr)

>>> zlib 1.2.8 Building
/usr/bin/make -j1 -C /home/thomas/projets/buildroot/output/build/zlib-1.2.8

>>> zlib 1.2.8 Installing to staging directory
/usr/bin/make -j1 -C /home/thomas/projets/buildroot/output/build/zlib-1.2.8
  DESTDIR=/home/thomas/projets/buildroot/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot
  LDCONFIG=true install

>>> zlib 1.2.8 Installing to target
/usr/bin/make -j1 -C /home/thomas/projets/buildroot/output/build/zlib-1.2.8
  DESTDIR=/home/thomas/projets/buildroot/output/target
  LDCONFIG=true install
```

Preparation work: dirs, prepare, dependencies

pkg-generic.mk

```
$$($(2)_TARGET_SOURCE): | dirs prepare
$$($(2)_TARGET_SOURCE): | dependencies
```

- All package have in their dependencies three targets:
  - `dirs`: creates the main directories (`BUILD_DIR`, `TARGET_DIR`, `HOST_DIR`, etc.). As part of creating `TARGET_DIR`, the root filesystem skeleton is copied into it
  - `prepare`: generates a kconfig-related `auto.conf` file
  - `dependencies`: triggers the check of Buildroot system dependencies, i.e things that must be installed on the machine to use Buildroot

Rebuilding packages?

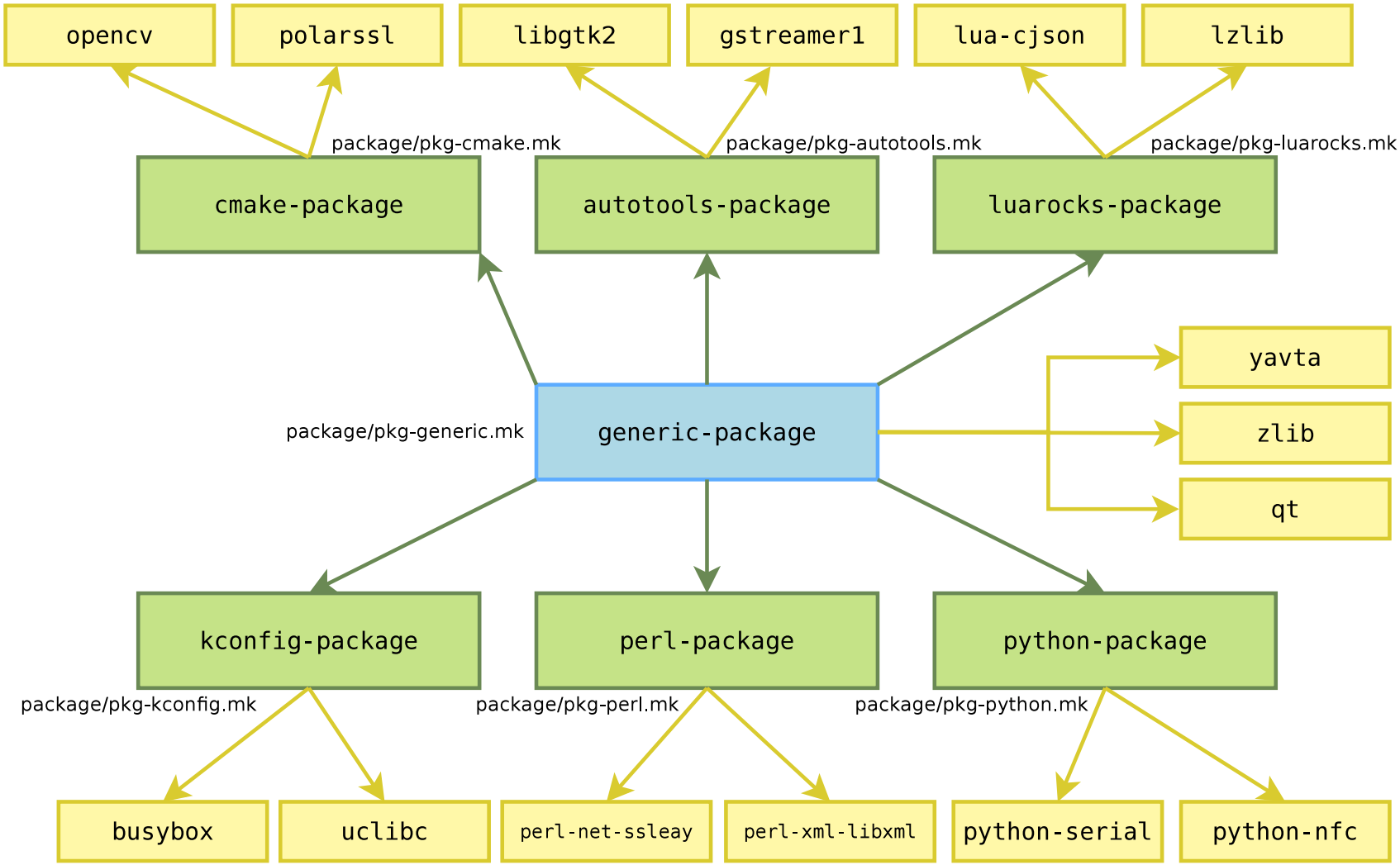
- Once one step of a package build process has been done, it is never done again due to the *stamp file*
- Even if the package configuration is changed, or the package is disabled → Buildroot doesn't try to be smart
- One can force rebuilding a package from its configure step or build step using `make <pkg>-reconfigure` or `make <pkg>-rebuild`

```
$(1)-clean-for-rebuild:
                        rm -f $$($(2)_TARGET_BUILD)
                        rm -f $$($(2)_TARGET_INSTALL_STAGING)
                        rm -f $$($(2)_TARGET_INSTALL_TARGET)
                        rm -f $$($(2)_TARGET_INSTALL_IMAGES)
                        rm -f $$($(2)_TARGET_INSTALL_HOST)

$(1)-rebuild:           $(1)-clean-for-rebuild $(1)

$(1)-clean-for-reconfigure: $(1)-clean-for-rebuild
                        rm -f $$($(2)_TARGET_CONFIGURE)

$(1)-reconfigure:       $(1)-clean-for-reconfigure $(1)
```

Specialized package infrastructures

- The `generic-package` infrastructure is fine for packages having a **custom** build system
- For packages having **well-known build system**, we want to factorize more logic
- Specialized **package infrastructures** were created to handle these packages, and reduce the amount of duplication
- For *autotools*, *CMake*, *Python*, *Perl*, *Lua* and *kconfig* packages

Specialized package infrastructures



CMake package example: `flann`

package/flann/flann.mk

```
FLANN_VERSION = d0c04f4d290ebc3aa9411a3322992d298e51f5aa
FLANN_SITE = $(call github,mariusmuja,flann,$(FLANN_VERSION))
FLANN_INSTALL_STAGING = YES
FLANN_LICENSE = BSD-3c
FLANN_LICENSE_FILES = COPYING
FLANN_CONF_OPT = \
        -DBUILD_C_BINDINGS=ON \
        -DBUILD_PYTHON_BINDINGS=OFF \
        -DBUILD_MATLAB_BINDINGS=OFF \
        -DBUILD_EXAMPLES=$(if $(BR2_PACKAGE_FLANN_EXAMPLES),ON,OFF) \
        -DBUILD_TESTS=OFF \
        -DBUILD_DOC=OFF \
        -DUSE_OPENMP=$(if $(BR2_GCC_ENABLE_OPENMP),ON,OFF) \
        -DPYTHON_EXECUTABLE=OFF

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

CMake package infrastructure (1/2)

```
define inner-cmake-package

$(2)_CONF_ENV                   ?=
$(2)_CONF_OPT                   ?=
...

$(2)_SRCDIR                     = $$($(2)_DIR)/$$($(2)_SUBDIR)
$(2)_BUILDDIR                   = $$($(2)_SRCDIR)

ifndef $(2)_CONFIGURE_CMDS
ifeq ($(4),target)
define $(2)_CONFIGURE_CMDS
    (cd $$($$(PKG)_BUILDDIR) && \
     $$($$(PKG)_CONF_ENV) $$(HOST_DIR)/usr/bin/cmake $$($$(PKG)_SRCDIR) \
         -DCMAKE_TOOLCHAIN_FILE="$$(HOST_DIR)/usr/share/buildroot/toolchainfile.cmake" \
         ...
         $$($$(PKG)_CONF_OPT) \
    )
endef
else
define $(2)_CONFIGURE_CMDS
... host case ...
endef
endif
endif
```

CMake package infrastructure (2/2)

```
$(2)_DEPENDENCIES += host-cmake

ifndef $(2)_BUILD_CMDS
ifeq ($(4),target)
define $(2)_BUILD_CMDS
        $$(TARGET_MAKE_ENV) $$($$(PKG)_MAKE_ENV) $$($$(PKG)_MAKE) $$($$(PKG)_MAKE_OPT)
            -C $$($$(PKG)_BUILDDIR)
endef
else
... host case ...
endif
endif

... other commands ...

ifndef $(2)_INSTALL_TARGET_CMDS
define $(2)_INSTALL_TARGET_CMDS
        $$(TARGET_MAKE_ENV) $$($$(PKG)_MAKE_ENV) $$($$(PKG)_MAKE) $$($$(PKG)_MAKE_OPT)
          $$($$(PKG)_INSTALL_TARGET_OPT) -C $$($$(PKG)_BUILDDIR)
endef
endif

$(call inner-generic-package,$(1),$(2),$(3),$(4))

endef

cmake-package = $(call inner-cmake-package,$(pkgname),...,target)
host-cmake-package = $(call inner-cmake-package,host-$(pkgname),...,host)
```

Autoreconf in pkg-autotools.mk

- Package infrastructures can also add additional capabilities controlled by variables in packages
- For example, with the autotools-package infra, one can do FOOBAR_AUTORECONF = YES in a package to trigger an *autoreconf* before the *configure* script is executed
- Implementation in pkg-autotools.mk

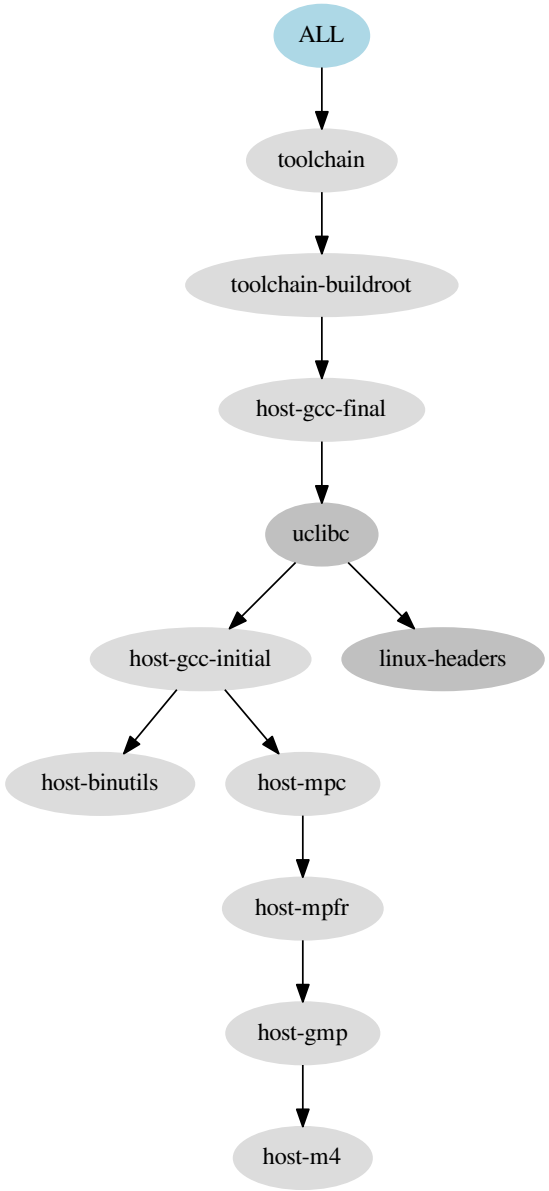
```
define AUTORECONF_HOOK
        @$$(call MESSAGE,"Autoreconfiguring")
        $$(Q)cd $$($$(PKG)_SRCDIR) && $$($$(PKG)_AUTORECONF_ENV) $$(AUTORECONF)
             $$($$(PKG)_AUTORECONF_OPTS)
        ...
endef

ifeq ($$($(2)_AUTORECONF),YES)
...
$(2)_PRE_CONFIGURE_HOOKS += AUTORECONF_HOOK
$(2)_DEPENDENCIES += host-automake host-autoconf host-libtool
endif
```

Toolchain support

- One *virtual package*, `toolchain`, with two implementations in the form of two packages: `toolchain-buildroot` and `toolchain-external`
- `toolchain-buildroot` implements the **internal toolchain back-end**, where Buildroot builds the cross-compilation toolchain from scratch. This package simply depends on `host-gcc-final` to trigger the entire build process
- `toolchain-external` implements the **external toolchain back-end**, where Buildroot uses an existing pre-built toolchain

Internal toolchain back-end

- Build starts with utility host tools and libraries needed for gcc (`host-m4`, `host-mpc`, `host-mpfr`, `host-gmp`). Installed in `$(HOST_DIR)/usr/{bin,include,lib}`
- Build goes on with the cross binutils, `host-binutils`, installed in `$(HOST_DIR)/usr/bin`
- Then the first stage compiler, `host-gcc-initial`
- We need the `linux-headers`, installed in `$(STAGING_DIR)/usr/include`
- We build the C library, `uclibc` in this example. Installed in `$(STAGING_DIR)/lib`, `$(STAGING_DIR)/usr/include` and of course `$(TARGET_DIR)/lib`
- We build the final compiler `host-gcc-final`, installed in `$(HOST_DIR)/usr/bin`



External toolchain back-end

- Implemented as one package, `toolchain-external`
- Knows about well-known toolchains (CodeSourcery, Linaro, etc.) or allows to use existing custom toolchains (built with Buildroot, Crosstool-NG, etc.)
- Core logic:
  1. Extract the toolchain to `$(HOST_DIR)/opt/ext-toolchain`
  2. Run some checks on the toolchain
  3. Copy the toolchain *sysroot* (C library and headers, kernel headers) to `$(STAGING_DIR)/usr/{include,lib}`
  4. Copy the toolchain libraries to `$(TARGET_DIR)/usr/lib`
  5. Create symbolic links or wrappers for the compiler, linker, debugger, etc from `$(HOST_DIR)/usr/bin/<tuple>-<tool>` to `$(HOST_DIR)/opt/ext-toolchain/bin/<tuple>-<tool>`
  6. A wrapper program is used for certain tools (gcc, ld, g++, etc.) in order to ensure a certain number of compiler flags are used, especially `--sysroot=$(STAGING_DIR)` and target-specific flags.

Root filesystem image generation

- Once all the targets in `$(TARGETS)` have been built, it's time to create the root filesystem images
- First, the `target-finalize` target does some cleanup of `$(TARGET_DIR)` by removing documentation, headers, static libraries, etc.
- Then the root filesystem image targets listed in `$(ROOTFS_TARGETS)` are processed
- These targets are added by the common filesystem image generation infrastructure, in `fs/common.mk`
- The purpose of this infrastructure is to factorize the preparation logic, and then call *fakeroot* to create the filesystem image

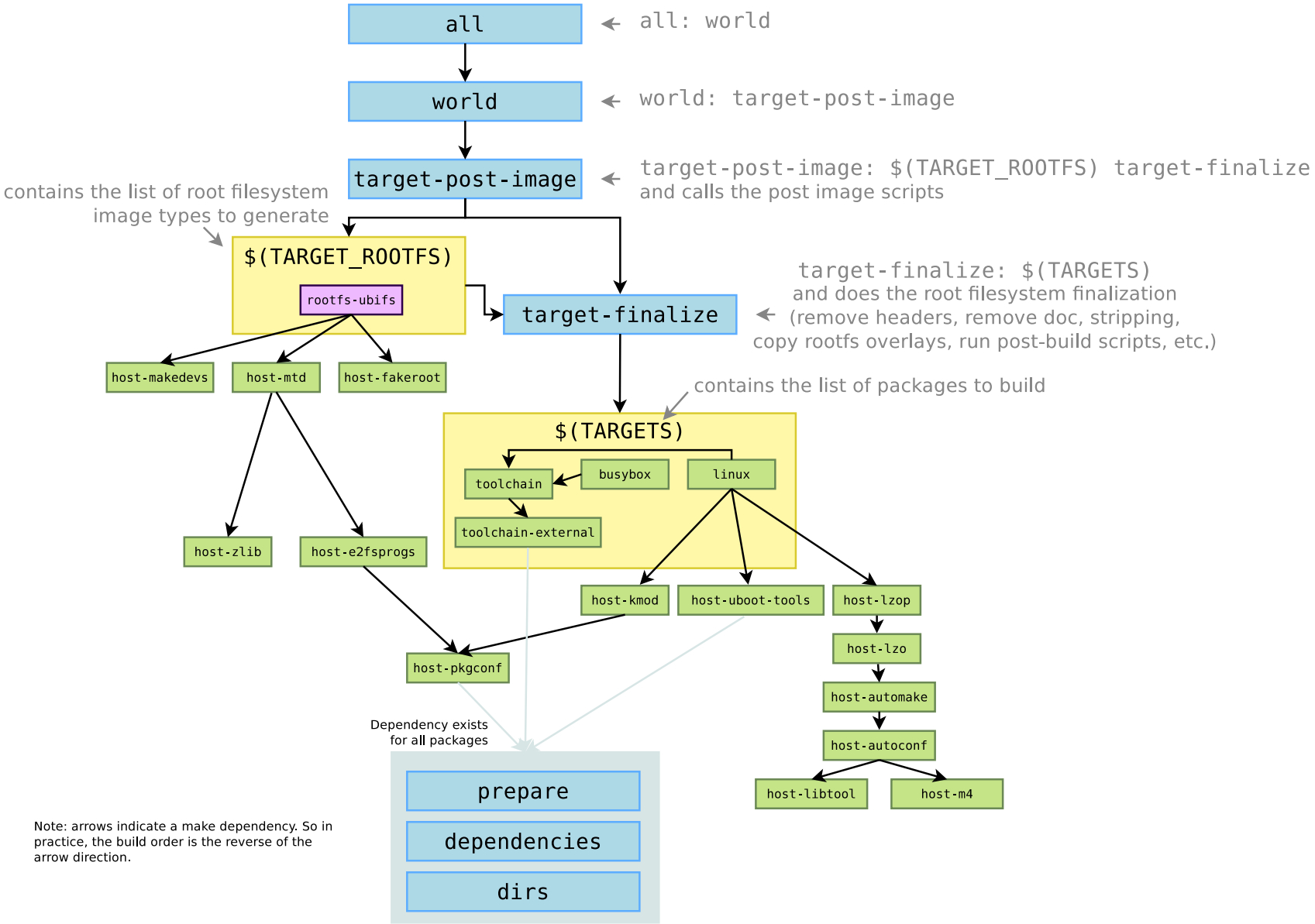
fs/common.mk

```
define ROOTFS_TARGET_INTERNAL

ROOTFS_$(2)_DEPENDENCIES += host-fakeroot host-makedevs \
        $$(if $$(PACKAGES_USERS),host-mkpasswd)

$$(BINARIES_DIR)/rootfs.$(1): target-finalize $$(ROOTFS_$(2)_DEPENDENCIES)
        @$$(call MESSAGE,"Generating root filesystem image rootfs.$(1)")
        $$(foreach hook,$$(ROOTFS_$(2)_PRE_GEN_HOOKS),$$(call $$(hook))$$(sep))
        ...
        echo "chown -h -R 0:0 $$(TARGET_DIR)" >> $$(FAKEROOT_SCRIPT)
        echo "$$(HOST_DIR)/usr/bin/makedevs -d $$(FULL_DEVICE_TABLE) $$(TARGET_DIR)" >> \
              $$(FAKEROOT_SCRIPT)
        echo "$$(ROOTFS_$(2)_CMD)" >> $$(FAKEROOT_SCRIPT)
        chmod a+x $$(FAKEROOT_SCRIPT)
        PATH=$$(BR_PATH) $$(HOST_DIR)/usr/bin/fakeroot -- $$(FAKEROOT_SCRIPT)
        ...

rootfs-$(1): $$(BINARIES_DIR)/rootfs.$(1) $$(ROOTFS_$(2)_POST_TARGETS)

ifeq ($$(BR2_TARGET_ROOTFS_$(2)),y)
TARGETS_ROOTFS += rootfs-$(1)
endif
endef

define ROOTFS_TARGET
$(call ROOTFS_TARGET_INTERNAL,$(1),$(call UPPERCASE,$(1)))
endef
```

fs/ubifs/ubifs.mk

```
UBIFS_OPTS := -e $(BR2_TARGET_ROOTFS_UBIFS_LEBSIZE) \
              -c $(BR2_TARGET_ROOTFS_UBIFS_MAXLEBCNT) \
              -m $(BR2_TARGET_ROOTFS_UBIFS_MINIOSIZE)

ifeq ($(BR2_TARGET_ROOTFS_UBIFS_RT_ZLIB),y)
UBIFS_OPTS += -x zlib
endif
...

UBIFS_OPTS += $(call qstrip,$(BR2_TARGET_ROOTFS_UBIFS_OPTS))

ROOTFS_UBIFS_DEPENDENCIES = host-mtd

define ROOTFS_UBIFS_CMD
        $(HOST_DIR)/usr/sbin/mkfs.ubifs -d $(TARGET_DIR) $(UBIFS_OPTS) -o $@
endef

$(eval $(call ROOTFS_TARGET,ubifs))
```

Final example



http://buildroot.org
http://buildroot.org/downloads/manual/manual.html

Questions?

thomas.petazzoni@free-electrons.com

http://free-electrons.com/pub/conferences/2014/elce/petazzoni-dive-into-buildroot-core/