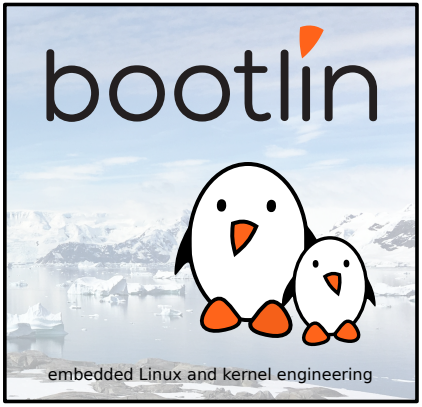
Application development



Building during development

- Buildroot is mainly a *final integration* tool: it is aimed at downloading and building **fixed** versions of software components, in a reproducible way.
- When doing active development of a software component, you need to be able to quickly change the code, build it, and deploy it on the target.
- The package build directory is temporary, and removed on `make clean`, so making changes here is not practical
- Buildroot does not automatically "update" your source code when the package is fetched from a version control system.
- Three solutions:
  - Build your software component outside of Buildroot during development. Doable for software components that are easy to build.
  - Use the `local SITE_METHOD` for your package
  - Use the `<pkg>_OVERRIDE_SRCDIR` mechanism

Building code for Buildroot

- The Buildroot cross-compiler is installed in `$(HOST_DIR)/bin`
- It is already set up to:
  - generate code for the configured architecture
  - look for libraries and headers in `$(STAGING_DIR)`
- Other useful tools that may be built by Buildroot are installed in `$(HOST_DIR)/bin`:
  - `pkg-config`, to find libraries. Beware that it is configured to return results for *target* libraries: it should only be used when cross-compiling.
  - `qmake`, when building Qt applications with this build system.
  - `autoconf`, `automake`, `libtool`, to use versions independent from the host system.
- Adding `$(HOST_DIR)/bin` to your `PATH` when cross-compiling is the easiest solution.

Building code for Buildroot: C program

Building a C program for the host

```
$ gcc -o foobar foobar.c
$ file foobar
foobar: ELF 64-bit LSB executable, x86-64, version 1...
```

Building a C program for the target

```
$ export PATH=$(pwd)/output/host/bin:$PATH
$ arm-linux-gcc -o foobar foobar.c
$ file foobar
foobar: ELF 32-bit LSB executable, ARM, EABI5 version 1...
```

Building code for Buildroot: pkg-config

Using the system `pkg-config`

```
$ pkg-config --cflags libpng
-I/usr/include/libpng12

$ pkg-config --libs libpng
-lpng12
```

Using the Buildroot `pkg-config`

```
$ export PATH=$(pwd)/output/host/bin:$PATH

$ pkg-config --cflags libpng
-I.../output/host/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/include/libpng16

$ pkg-config --libs libpng
-L.../output/host/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/lib -lpng16
```

Building code for Buildroot: autotools

- Building simple *autotools* components outside of Buildroot is easy:

```
$ export PATH=.../buildroot/output/host/bin/:$PATH
$ ./configure --host=arm-linux
```

- Passing `--host=arm-linux` tells the configure script to use the cross-compilation tools prefixed by `arm-linux-`.
- In more complex cases, some additional `CFLAGS` or `LDFLAGS` might be needed in the environment.

Building code for Buildroot: CMake

- Buildroot generates a *CMake toolchain file*, installed in `output/host/share/buildroot/toolchainfile.cmake`
- Tells *CMake* which cross-compilation tools to use
- Passed using the `CMAKE_TOOLCHAIN_FILE` *CMake* option
- https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html
- With this file, building *CMake* projects outside of Buildroot is easy:

```
$ cmake -DCMAKE_TOOLCHAIN_FILE=.../buildroot/output/host/share/buildroot/toolchainfile.cmake .
$ make
$ file app
app: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked...
```

Building code for Buildroot: Meson

- Buildroot generates a Meson *cross file*, installed in `output/host/etc/meson/cross-compilation.conf`
- Tells *Meson* which cross-compilation tools to use
- Passed using the `--cross-file` *Meson* option
- https://mesonbuild.com/Cross-compilation.html
- With this file, building *Meson* projects outside of Buildroot is easy:

```
$ mkdir build
$ meson --cross-file=.../buildroot/output/host/etc/meson/cross-compilation.conf ..
$ ninja
$ file app
app: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked...
```

Building code for Buildroot: environment-setup

- Enable BR2_PACKAGE_HOST_ENVIRONMENT_SETUP
- Installs an helper shell script output/host/environment-setup that can be sourced in the shell to define a number of useful environment variables and aliases.
- Defines: CC, LD, AR, AS, CFLAGS, LDFLAGS, ARCH, etc.
- Defines configure as an alias to run a *configure* script with the right arguments, cmake as an alias to run *cmake* with the right arguments
- Drawback: once sourced, the shell environment is really only suitable for cross-compiling with Buildroot.

Building code for Buildroot: environment-setup

```
$ source output/host/environment-setup
 _           _ _     _                 _
| |__  _   _(_) | __| |_ __ ___   ___ | |_
| '_ \| | | | | |/ _` | '__/ _ \ / _ \| __|
| |_) | |_| | | | (_| | | | (_) | (_) | |_
|_.__/ \__,_|_|_|\__,_|_|  \___/ \___/ \__|

       Making embedded Linux easy!

Some tips:
* PATH now contains the SDK utilities
* Standard autotools variables (CC, LD, CFLAGS) are exported
* Kernel compilation variables (ARCH, CROSS_COMPILE, KERNELDIR) are exported
* To configure do "./configure $CONFIGURE_FLAGS" or use
  the "configure" alias
* To build CMake-based projects, use the "cmake" alias
```

```
$ echo $CC
/home/thomas/projets/buildroot/output/host/bin/arm-linux-gcc
$ echo $CFLAGS
-D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64 -Os -D_FORTIFY_SOURCE=1
$ echo $CROSS_COMPILE
/home/thomas/projets/buildroot/output/host/bin/arm-linux-
$ alias configure
alias configure='./configure --target=arm-buildroot-linux-gnueabihf --host=arm-buildroot-linux-gnueabihf \
    --build=x86_64-pc-linux-gnu --prefix=/usr --exec-prefix=/usr --sysconfdir=/etc --localstatedir=/var \
    --program-prefix='
```

local site method

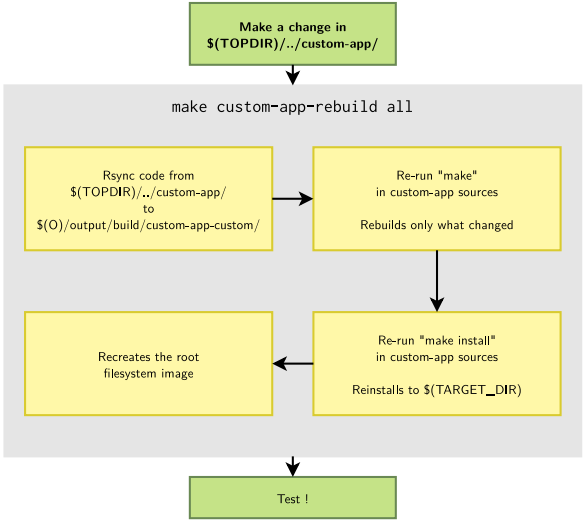
- Allows to tell Buildroot that the source code for a package is already available locally
- Allows to keep your source code under version control, separately, and have Buildroot always build your latest changes.
- Typical project organization:
  - `buildroot/`, the Buildroot source code
  - `external/`, your `BR2_EXTERNAL` tree
  - `custom-app/`, your custom application code
  - `custom-lib/`, your custom library
- In your package `.mk` file, use:

```
<pkg>_SITE = $(TOPDIR)/../custom-app
<pkg>_SITE_METHOD = local
```

Effect of `local` site method

- For the first build, the source code of your package is *rsync'ed* from `<pkg>_SITE` to the build directory, and built there.
- After making changes to the source code, you can run:
  - `make <pkg>-reconfigure`
  - `make <pkg>-rebuild`
  - `make <pkg>-reinstall`
- Buildroot will first *rsync* again the package source code (copying only the modified files) and restart the build from the requested step.

local site method workflow



<pkg>_OVERRIDE_SRCDIR

- The `local` site method solution is appropriate when the package uses this method for all developers
  - Requires that all developers fetch locally the source code for all custom applications and libraries
- An alternate solution is that packages for custom applications and libraries fetch their source code from version control systems
  - Using the `git`, `svn`, `cvs`, etc. fetching methods
- Then, locally, a user can **override** how the package is fetched using `<pkg>_OVERRIDE_SRCDIR`
  - It tells Buildroot to not *download* the package source code, but to copy it from a local directory.
- The package then behaves as if it was using the `local` site method.

Passing <pkg>_OVERRIDE_SRCDIR

- <pkg>_OVERRIDE_SRCDIR values are specified in a *package override file*, configured in BR2_PACKAGE_OVERRIDE_FILE, by default $(CONFIG_DIR)/local.mk.

Example local.mk

```
LIBPNG_OVERRIDE_SRCDIR = $(HOME)/projects/libpng
LINUX_OVERRIDE_SRCDIR = $(HOME)/projects/linux
```

Debugging: debugging symbols and stripping

- To use debuggers, you need the programs and libraries to be built with debugging symbols.
- The `BR2_ENABLE_DEBUG` option controls whether programs and libraries are built with debugging symbols
  - Disabled by default.
  - Sub-options allow to control the amount of debugging symbols (i.e. gcc options `-g1`, `-g2` and `-g3`).
- The `BR2_STRIP_strip` option allows to disable or enable stripping of binaries on the target.
  - Enabled by default.

Debugging: debugging symbols and stripping

- With `BR2_ENABLE_DEBUG=y` and `BR2_STRIP_strip=y`
  - get debugging symbols in `$(STAGING_DIR)` for libraries, and in the build directories for everything.
  - stripped binaries in `$(TARGET_DIR)`
  - Appropriate for **remote debugging**
- With `BR2_ENABLE_DEBUG=y` and `BR2_STRIP_strip` disabled
  - debugging symbols in both `$(STAGING_DIR)` and `$(TARGET_DIR)`
  - appropriate for **on-target debugging**

Debugging: remote debugging requirements

- To do remote debugging, you need:
  - A **cross-debugger**
    - With the *internal toolchain backend*, can be built using `BR2_PACKAGE_HOST_GDB=y`.
    - With the *external toolchain backend*, is either provided pre-built by the toolchain, or can be built using `BR2_PACKAGE_HOST_GDB=y`.
  - **gdbserver**
    - With the *internal toolchain backend*, can be built using `BR2_PACKAGE_GDB=y` + `BR2_PACKAGE_GDB_SERVER=y`
    - With the *external toolchain backend*, if `gdbserver` is provided by the toolchain it can be copied to the target using `BR2_TOOLCHAIN_EXTERNAL_GDB_SERVER_COPY=y` or otherwise built from source like with the internal toolchain backend.

Debugging: remote debugging setup

- On the target, start *gdbserver*
  - Use a TCP socket, network connectivity needed
  - The *multi* mode is quite convenient
  - `$ gdbserver --multi localhost:2345`
- On the host, start `<tuple>-gdb`
  - `$ ./output/host/bin/<tuple>-gdb <program>`
  - `<program>` is the path to the program to debug, with debugging symbols
- Inside *gdb*, you need to:
  - Connect to the target:
    `(gdb) target extended-remote <ip>:2345`
  - Tell the target which program to run:
    `(gdb) set remote exec-file myapp`
  - Set the path to the *sysroot* so that *gdb* can find debugging symbols for libraries:
    `(gdb) set sysroot ./output/staging/`
  - Start the program:
    `(gdb) run`

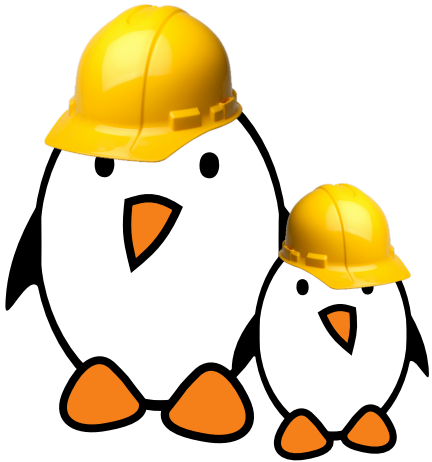
Debugging tools available in Buildroot

- Buildroot also includes a huge amount of other debugging or profiling related tools.
- To list just a few:
  - strace
  - ltrace
  - LTTng
  - perf
  - sysdig
  - sysprof
  - OProfile
  - valgrind
- Look in `Target packages` → `Debugging, profiling and benchmark` for more.

Generating a SDK for application developers

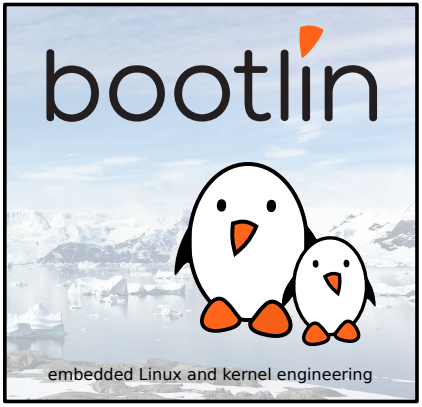
- If you would like application developers to build applications for a Buildroot generated system, without building Buildroot, you can generate a SDK.
- To achieve this:
  - Run `make sdk`, which prepares the SDK to be relocatable
  - Tarball the contents of the *host* directory, i.e `output/host`
  - Share the tarball with your application developers
  - They must uncompress it, and run the `relocate-sdk.sh` script
- **Warning**: the SDK must remain in sync with the root filesystem running on the target, otherwise applications built with the SDK may not run properly.

Practical lab - Application development



- Build and run your own application
- Remote debug your application
- Use `<pkg>_OVERRIDE_SRCDIR`

Understanding Buildroot internals



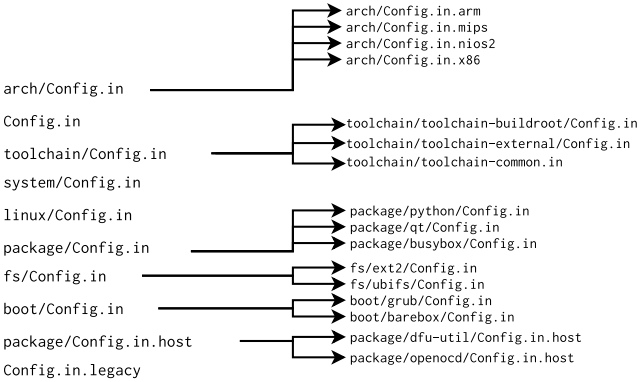
Configuration system

- 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 *config, is at the top-level of the Buildroot source tree

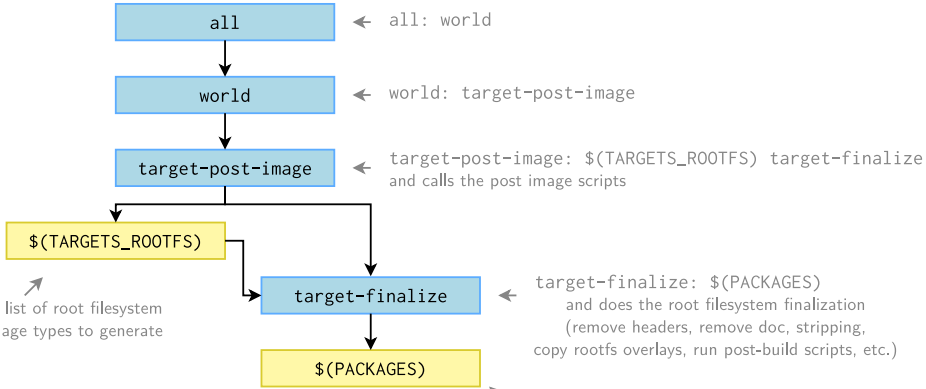
```
CONFIG_CONFIG_IN = Config.in
CONFIG = support/kconfig
BR2_CONFIG = $(CONFIG_DIR)/.config

-include $(BR2_CONFIG)

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

menuconfig: $(BUILD_DIR)/buildroot-config/mconf outputmakefile
        @$(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  --->
```



When you run make...



Note: arrows indicate a make dependency. So in practice, the build order is the reverse of the arrow direction.

Where is $(PACKAGES) 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)
PACKAGES += $(1)
endif # $(2)_KCONFIG_VAR

endef # inner-generic-package
```

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

Diving into pkg-generic.mk

- The `package/pkg-generic.mk` file is divided in two main parts:
  1. Definition of the actions done in each step of a package build process. Done through *stamp file targets*.
  2. 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)/%/.stamp_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))
        [...]
        $(foreach p,$($(PKG)_ALL_DOWNLOADS),$(call DOWNLOAD,$(p))$(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.
- `<pkg>_ALL_DOWNLOADS` lists all the files to be downloaded, which includes the ones listed 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))
        @$(call step_end,build)
        $(Q)touch $@
```

- 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/libzlib/libzlib.mk:

```
LIBZLIB_... = ...

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

- Leads to:

```
$(call inner-generic-package,libzlib,LIBZLIB,LIBZLIB,target)
$(call inner-generic-package,host-libzlib,HOST_LIBZLIB,LIBZLIB,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-libzlib

```
HOST_LIBZLIB_TYPE    = host
HOST_LIBZLIB_NAME    = host-libzlib
HOST_LIBZLIB_RAWNAME = libzlib

HOST_LIBZLIB_BASE_NAME =
  host-libzlib-$(HOST_LIBZLIB_VERSION)
HOST_LIBZLIB_DIR       =
  $(BUILD_DIR)/host-libzlib-$(HOST_LIBZLIB_VERSION)

ifndef HOST_LIBZLIB_SOURCE
 ifdef LIBZLIB_SOURCE
  HOST_LIBZLIB_SOURCE = $(LIBZLIB_SOURCE)
 else
  HOST_LIBZLIB_SOURCE ?=
   libzlib-$(HOST_LIBZLIB_VERSION).tar.gz
 endif
endif

ifndef HOST_LIBZLIB_SITE
 ifdef LIBZLIB_SITE
  HOST_LIBZLIB_SITE = $(LIBZLIB_SITE)
 endif
endif

...
```

inner-generic-package: dependencies

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

...

ifeq ($$(BR2_CCACHE),y)
ifeq ($$(filter host-tar host-skeleton host-xz host-lzip host-fakedate host-ccache,$(1)),)
$(2)_DEPENDENCIES += host-ccache
endif
endif
```

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

inner-generic-package: stamp files

```
$(2)_TARGET_INSTALL =           $$($(2)_DIR)/.stamp_installed
$(2)_TARGET_INSTALL_TARGET =    $$($(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

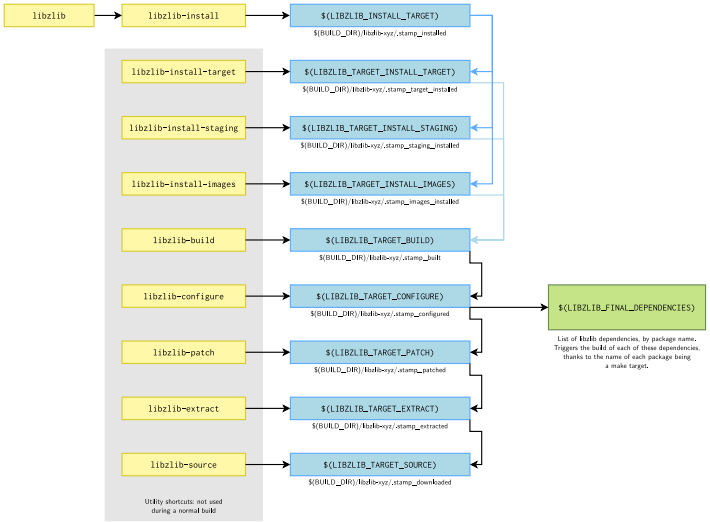
```
$$($(2)_TARGET_INSTALL):                PKG=$(2)
$$($(2)_TARGET_INSTALL_TARGET):         PKG=$(2)
$$($(2)_TARGET_INSTALL_STAGING):        PKG=$(2)
$$($(2)_TARGET_INSTALL_IMAGES):         PKG=$(2)
$$($(2)_TARGET_INSTALL_HOST):           PKG=$(2)
[...]
```

- Pass variables to the stamp file targets, especially PKG

inner-generic-package: sequencing

```
$(1):                   $(1)-install
$(1)-install:           $$($(2)_TARGET_INSTALL)

ifeq ($$($(2)_INSTALL_TARGET),YES)
$$($(2)_TARGET_INSTALL): $$($(2)_TARGET_INSTALL_TARGET)
endif
ifeq ($$($(2)_INSTALL_STAGING),YES)
$$($(2)_TARGET_INSTALL): $$($(2)_TARGET_INSTALL_STAGING)
endif
ifeq ($$($(2)_INSTALL_IMAGES),YES)
$$($(2)_TARGET_INSTALL): $$($(2)_TARGET_INSTALL_IMAGES)
endif

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

$(1)-install-staging:                   $$($(2)_TARGET_INSTALL_STAGING)
$$($(2)_TARGET_INSTALL_STAGING):        $$($(2)_TARGET_BUILD)

$(1)-install-images:            $$($(2)_TARGET_INSTALL_IMAGES)
$$($(2)_TARGET_INSTALL_IMAGES): $$($(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):        $$($(2)_TARGET_SOURCE)
$$($(2)_TARGET_EXTRACT): | $$($(2)_FINAL_EXTRACT_DEPENDENCIES)

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

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


libzlib
libzlib-install
$(LIBZLIB_INSTALL_TARGET)
$(BUILD_DIR)/libzlib-xyz/.stamp_installed
libzlib-install-target
$(LIBZLIB_TARGET_INSTALL_TARGET)
$(BUILD_DIR)/libzlib-xyz/.stamp_target_installed
libzlib-install-staging
$(LIBZLIB_TARGET_INSTALL_STAGING)
$(BUILD_DIR)/libzlib-xyz/.stamp_staging_installed
libzlib-install-images
$(LIBZLIB_TARGET_INSTALL_IMAGES)
$(BUILD_DIR)/libzlib-xyz/.stamp_images_installed
libzlib-build
$(LIBZLIB_TARGET_BUILD)
$(BUILD_DIR)/libzlib-xyz/.stamp_built
libzlib-configure
$(LIBZLIB_TARGET_CONFIGURE)
$(BUILD_DIR)/libzlib-xyz/.stamp_configured
libzlib-patch
$(LIBZLIB_TARGET_PATCH)
$(BUILD_DIR)/libzlib-xyz/.stamp_patched
libzlib-extract
$(LIBZLIB_TARGET_EXTRACT)
$(BUILD_DIR)/libzlib-xyz/.stamp_extracted
libzlib-source
$(LIBZLIB_TARGET_SOURCE)
$(BUILD_DIR)/libzlib-xyz/.stamp_downloaded
Utility shortcuts: not used during a normal build
$(LIBZLIB_FINAL_DEPENDENCIES)
List of libzlib dependencies, by package name. Triggers the build of each of these dependencies, thanks to the name of each package being a make target.

Preparation work: prepare, dependencies

pkg-generic.mk

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

- All packages have two targets in their dependencies:
  - `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, build or install step using `make <pkg>-reconfigure`, `make <pkg>-rebuild` or `make <pkg>-reinstall`

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

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

$(1)-clean-for-rebuild: $(1)-clean-for-reinstall
                        rm -f $$($(2)_TARGET_BUILD)

$(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 using a **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*, *Meson*, *Golang*, *QMake*, *kconfig*, *Rust*, *kernel-module*, *Erlang*, *Waf* packages

CMake package example: `flann`

package/flann/flann.mk

```
FLANN_VERSION = 1.9.1
FLANN_SITE = $(call github,mariusmuja,flann,$(FLANN_VERSION))
FLANN_INSTALL_STAGING = YES
FLANN_LICENSE = BSD-3-Clause
FLANN_LICENSE_FILES = COPYING
FLANN_CONF_OPTS = \
        -DBUILD_C_BINDINGS=ON \
        -DBUILD_PYTHON_BINDINGS=OFF \
        -DBUILD_MATLAB_BINDINGS=OFF \
        -DBUILD_EXAMPLES=$(if $(BR2_PACKAGE_FLANN_EXAMPLES),ON,OFF) \
        -DUSE_OPENMP=$(if $(BR2_GCC_ENABLE_OPENMP),ON,OFF) \
        -DPYTHON_EXECUTABLE=OFF \
        -DCMAKE_DISABLE_FIND_PACKAGE_HDF5=TRUE

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

CMake package infrastructure (1/2)

```
define inner-cmake-package

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

$(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) $$(HOST_DIR)/bin/cmake $$($$(PKG)_SRCDIR) \
         -DCMAKE_TOOLCHAIN_FILE="$$(HOST_DIR)/share/buildroot/toolchainfile.cmake" \
         ...
         $$($$(PKG)_CONF_OPTS) \
    )
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
        $$(TARGET_MAKE_ENV) $$($$(PKG)_MAKE_ENV) $$($$(PKG)_MAKE) $$($$(PKG)_MAKE_OPTS)
            -C $$($$(PKG)_BUILDDIR)
endef
else
... host case ...
endif
endif

... other commands ...

ifndef $(2)_INSTALL_TARGET_CMDS
define $(2)_INSTALL_TARGET_CMDS
        $$(TARGET_MAKE_ENV) $$($$(PKG)_MAKE_ENV) $$($$(PKG)_MAKE) $$($$(PKG)_MAKE_OPTS)
          $$($$(PKG)_INSTALL_TARGET_OPT) -C $$($$(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

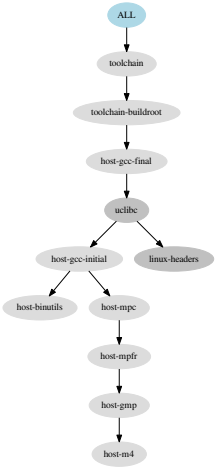
```
define AUTORECONF_HOOK
        @$$(call MESSAGE,"Autoreconfiguring")
        $$(Q)cd $$($$(PKG)_SRCDIR) && $$($$(PKG)_AUTORECONF_ENV) $$(AUTORECONF)
             $$($$(PKG)_AUTORECONF_OPTS)
        ...
endef

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

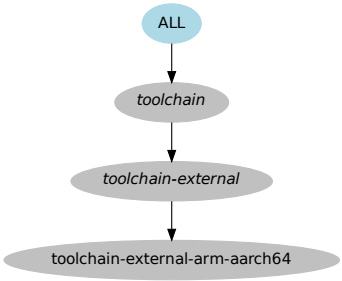
Internal toolchain back-end

- Build starts with utility host tools and libraries needed for gcc (`host-m4`, `host-mpc`, `host-mpfr`, `host-gmp`). Installed in `$(HOST_DIR)/{bin,include,lib}`
- Build goes on with the cross binutils, `host-binutils`, installed in `$(HOST_DIR)/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)/bin`



External toolchain back-end

- `toolchain-external-package` infrastructure, implementing the common logic for all external toolchains
  - Implemented in `toolchain/toolchain-external/pkg-toolchain-external.mk`
- Packages in `toolchain/toolchain-external/` are using this infrastructure
  - E.g. `toolchain-external-arm-aarch64`, `toolchain-external-bootlin`
- `toolchain-external` is a virtual package itself depends on the selected external toolchain.



External toolchain example

toolchain/toolchain-external/toolchain-external-arm-aarch64/toolchain-external-arm-aarch64.mk

```
TOOLCHAIN_EXTERNAL_ARM_AARCH64_VERSION = 2020.11
TOOLCHAIN_EXTERNAL_ARM_AARCH64_SITE = \
    https://developer.arm.com/-/media/Files/downloads/
      gnu-a/10.2-$(TOOLCHAIN_EXTERNAL_ARM_AARCH64_VERSION)/binrel

TOOLCHAIN_EXTERNAL_ARM_AARCH64_SOURCE = \
    gcc-arm-10.2-$(TOOLCHAIN_EXTERNAL_ARM_AARCH64_VERSION)-x86_64-aarch64-none-linux-gnu.tar.xz

$(eval $(toolchain-external-package))
```

toolchain-external-package logic

1. Extract the toolchain to `$(HOST_DIR)/opt/ext-toolchain`
2. Run some checks on the toolchain to verify it matches the configuration specified in *menuconfig*
3. Copy the toolchain *sysroot* (C library and headers, kernel headers) to `$(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)/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 `$(PACKAGES)` have been built, it's time to create the root filesystem images
- First, the `target-finalize` target does some cleanup of `$(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 `rootfs`, in `fs/common.mk`
- The purpose of this infrastructure is to:
  - Collect the users, permissions and device tables
  - Make a copy of `TARGET_DIR` per filesystem image
  - Generate a shell script that assigns users, permissions and invokes the filesystem image creation utility
  - Invoke the shell script under `fakeroot`

fs/common.mk, dependencies and table generation

```
ROOTFS_COMMON_DEPENDENCIES = \
        host-fakeroot host-makedevs \
        $(BR2_TAR_HOST_DEPENDENCY) \
        $(if $(PACKAGES_USERS)$(ROOTFS_USERS_TABLES),host-mkpasswd)

rootfs-common: $(ROOTFS_COMMON_DEPENDENCIES) target-finalize
        @$(call MESSAGE,"Generating root filesystems common tables")
        rm -rf $(FS_DIR)
        mkdir -p $(FS_DIR)
        $(call PRINTF,$(PACKAGES_USERS)) >> $(ROOTFS_FULL_USERS_TABLE)
        cat $(ROOTFS_USERS_TABLES) >> $(ROOTFS_FULL_USERS_TABLE)
        $(call PRINTF,$(PACKAGES_PERMISSIONS_TABLE)) > $(ROOTFS_FULL_DEVICES_TABLE)
        cat $(ROOTFS_DEVICE_TABLES) >> $(ROOTFS_FULL_DEVICES_TABLE)
        $(call PRINTF,$(PACKAGES_DEVICES_TABLE)) >> $(ROOTFS_FULL_DEVICES_TABLE)
```

fs/common.mk, rootfs infrastructure 1

```
define inner-rootfs

ROOTFS_$(2)_IMAGE_NAME ?= rootfs.$(1)
ROOTFS_$(2)_FINAL_IMAGE_NAME = $$(strip $$(ROOTFS_$(2)_IMAGE_NAME))
ROOTFS_$(2)_DIR = $$(FS_DIR)/$(1)
ROOTFS_$(2)_TARGET_DIR = $$(ROOTFS_$(2)_DIR)/target

ROOTFS_$(2)_DEPENDENCIES += rootfs-common
```

fs/common.mk, rootfs infrastructure 2

```
$$(BINARIES_DIR)/$$(ROOTFS_$(2)_FINAL_IMAGE_NAME): $$(ROOTFS_$(2)_DEPENDENCIES)
        @$$(call MESSAGE,"Generating filesystem image $$(ROOTFS_$(2)_FINAL_IMAGE_NAME)")
        [...]
        mkdir -p $$(ROOTFS_$(2)_DIR)
        rsync -auH \
                --exclude=/$$(notdir $$(TARGET_DIR_WARNING_FILE)) \
                $$(BASE_TARGET_DIR)/ \
                $$(TARGET_DIR)
        echo '#!/bin/sh' > $$(FAKEROOT_SCRIPT)
        echo "set -e" >> $$(FAKEROOT_SCRIPT)
        echo "chown -h -R 0:0 $$(TARGET_DIR)" >> $$(FAKEROOT_SCRIPT)
        PATH=$$(BR_PATH) $$(TOPDIR)/support/scripts/mkusers $$(ROOTFS_FULL_USERS_TABLE) $$(TARGET_DIR) >> $$(FAKEROOT_SCRIPT)
        echo "$$(HOST_DIR)/bin/makedevs -d $$(ROOTFS_FULL_DEVICES_TABLE) $$(TARGET_DIR)" >> $$(FAKEROOT_SCRIPT)
        [...]
        $$(call PRINTF,$$(ROOTFS_$(2)_CMD)) >> $$(FAKEROOT_SCRIPT)
        chmod a+x $$(FAKEROOT_SCRIPT)
        PATH=$$(BR_PATH) $$(HOST_DIR)/bin/fakeroot -- $$(FAKEROOT_SCRIPT)
[...]
ifeq ($$(BR2_TARGET_ROOTFS_$(2)),y)
TARGETS_ROOTFS += rootfs-$(1)
endif
endef

rootfs = $(call inner-rootfs,$(pkgname),$(call UPPERCASE,$(pkgname)))
```

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)/sbin/mkfs.ubifs -d $(TARGET_DIR) $(UBIFS_OPTS) -o $@
endef

$(eval $(rootfs))
```

Final example