# Buildroot

## *free electrons*

Embedded Linux
Experts


Latest update: December 18, 2015.

Document updates and sources:
`http://free-electrons.com/doc/training/buildroot`

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@free-electrons.com

# Rights to copy

# Hyperlinks in the document

There are many hyperlinks in the document

- Regular hyperlinks:
  `http://kernel.org/`
- Kernel documentation links:
  `Documentation/kmemcheck.txt`
- Links to kernel source files and directories:
  `drivers/input`
  `include/linux/fb.h`
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  `platform_get_irq()`
  `GFP_KERNEL`
  `struct file_operations`

# Free Electrons at a glance

- Engineering company created in 2004
  (not a training company!)
- Locations: Orange, Toulouse, Lyon (France)
- Serving customers all around the world
  See `http://free-electrons.com/company/customers/`
- Head count: 9
  Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel, Android Free Software / Open Source for embedded and real-time systems.
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

# Free Electrons on-line resources

- All our training materials:
  http://free-electrons.com/docs/
- Technical blog:
  http://free-electrons.com/blog/
- Quarterly newsletter:
  http://lists.free-electrons.com/mailman/listinfo/newsletter
- News and discussions (Google +):
  https://plus.google.com/+FreeElectronsDevelopers
- News and discussions (LinkedIn):
  http://linkedin.com/groups/Free-Electrons-4501089
- Quick news (Twitter):
  http://twitter.com/free_electrons
- Linux Cross Reference - browse Linux kernel sources on-line:
  http://lxr.free-electrons.com

# Generic course information

***free electrons***

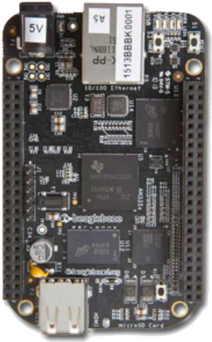
Embedded Linux
Experts


Corrections, suggestions, contributions and translations are welcome!

# Hardware used in this training session

## BeagleBone Black, from CircuitCo

- Texas Instruments AM335x (ARM Cortex-A8)
- Powerful CPU, with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- 512 MB of RAM
- 2 GB of on-board eMMC storage (4 GB in Rev C)
- USB host and USB device ports
- microSD slot
- HDMI port
- 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- A huge number of expansion boards, called *capes*. See `http://beagleboardtoys.com/`.



# Do not damage your BeagleBone Black!

- Do not remove power abruptly:
  - Boards components have been damaged by removing the power or USB cable in an abrupt way, not leaving the PMIC the time to switch off the components in a clean way. See `http://bit.ly/1FWHNZi`
  - Reboot (`reboot`) or shutdown (`halt`) the board in software when Linux is running.
  - You can also press the `RESET` button to reset and reboot.
  - When there is no software way, you can also switch off the board by pressing the `POWER` button for 8 seconds.
- Do not leave your board powered on a metallic surface (like a laptop with a metal finish).

# Participate!

During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- This helps the trainer to detect any explanation that wasn't clear or detailed enough.
- Don't hesitate to share your experience, for example to compare Linux / Android with other operating systems used in your company.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- Your participation can make our session more interactive and make the topics easier to learn.

# Practical lab guidelines

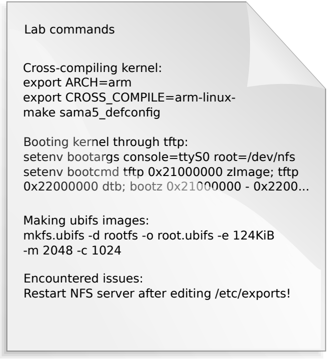
During practical labs...

- We cannot support more than 8 workstations at once (each with its board and equipment). Having more would make the whole class progress slower, compromising the coverage of the whole training agenda (exception for public sessions: up to 10 people).
- So, if you are more than 8 participants, please form up to 8 working groups.
- Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- Don't hesitate to copy and paste commands from the PDF slides and labs.

# Advise: write down your commands!

During practical labs, write down all your commands in a text file.

- You can save a lot of time re-using commands in later labs.
- This helps to replay your work if you make significant mistakes.
- You build a reference to remember commands in the long run.
- That's particular useful to keep kernel command line settings that you used earlier.
- Also useful to get help from the instructor, showing the commands that you run.


Lab commands

Cross-compiling kernel:
export ARCH=arm
export CROSS_COMPILE=arm-linux-
make sama5_defconfig

Booting kernel through tftp:
setenv bootargs console=ttyS0 root=/dev/nfs
setenv bootcmd tftp 0x21000000 zImage; tftp 0x22000000 dtb; bootz 0x21000000 - 0x2200...

Making ubifs images:
mkfs.ubifs -d rootfs -o root.ubifs -e 124KiB -m 2048 -c 1024

Encountered issues:
Restart NFS server after editing /etc/exports!

```
gedit ~/lab-history.txt
```

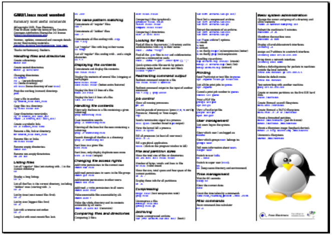
# Cooperate!

As in the Free Software and Open Source community, cooperation during practical labs is valuable in this training session:

- If you complete your labs before other people, don't hesitate to help other people and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- Explain what you understood to other participants when needed. It also helps to consolidate your knowledge.
- Don't hesitate to report potential bugs to your instructor.
- Don't hesitate to look for solutions on the Internet as well.

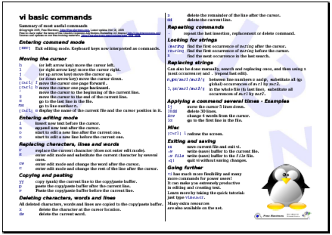
# Command memento sheet

- This memento sheet gives command examples for the most typical needs (looking for files, extracting a tar archive...)
- It saves us 1 day of UNIX / Linux command line training.
- Our best tip: in the command line shell, always hit the Tab key to complete command names and file paths. This avoids 95% of typing mistakes.
- Get an electronic copy on `http://free-electrons.com/doc/training/embedded-linux/command_memento.pdf`



# vi basic commands

- The `vi` editor is very useful to make quick changes to files in an embedded target.
- Though not very user friendly at first, `vi` is very powerful and its main 15 commands are easy to learn and are sufficient for 99% of everyone's needs!
- Get an electronic copy on `http://free-electrons.com/doc/training/embedded-linux/vi_memento.pdf`
- You can also take the quick tutorial by running `vimtutor`. This is a worthy investment!


vi basic commands

Entering command mode

Moving the cursor

Entering editing mode

Replacing characters, lines and words

Copying and pasting

Deleting characters, words and lines

Repeating commands

Looking for strings

Replacing strings

Applying a command several times - Examples

Misc

Exiting and saving

Going further

# Practical lab - Training Setup

Prepare your lab environment

- Download the lab archive
- Enforce correct permissions

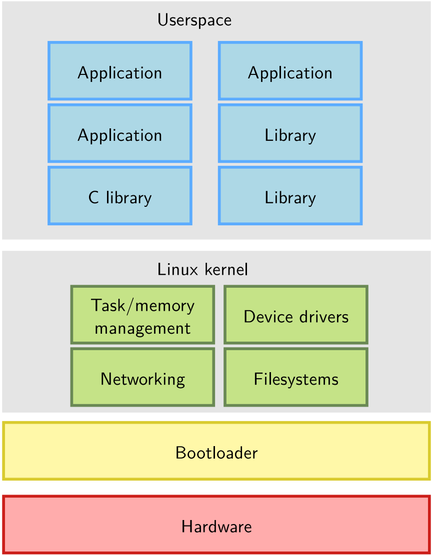
# Introduction to Embedded Linux

***free electrons***

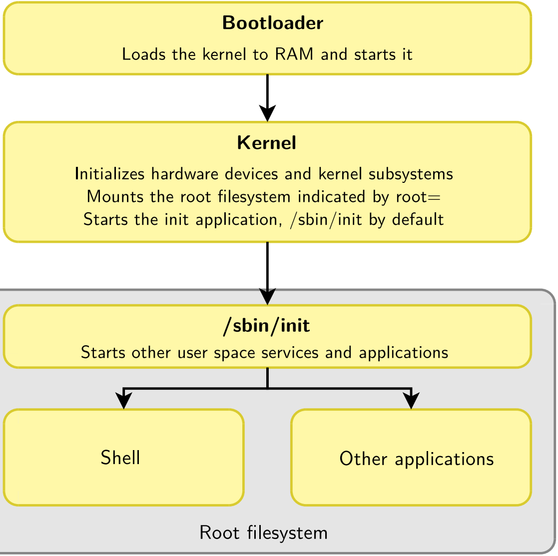
Embedded Linux
Experts


Corrections, suggestions, contributions and translations are welcome!

# Simplified Linux system architecture



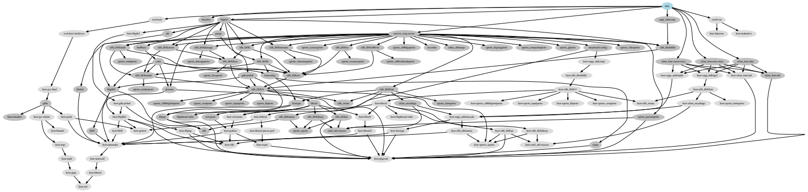
# Overall Linux boot sequence



# Embedded Linux work

- **BSP work**: porting the bootloader and Linux kernel, developing Linux device drivers.
- **system integration work**: assembling all the user space components needed for the system, configure them, develop the upgrade and recovery mechanisms, etc.
- **application development**: write the company-specific applications and libraries.

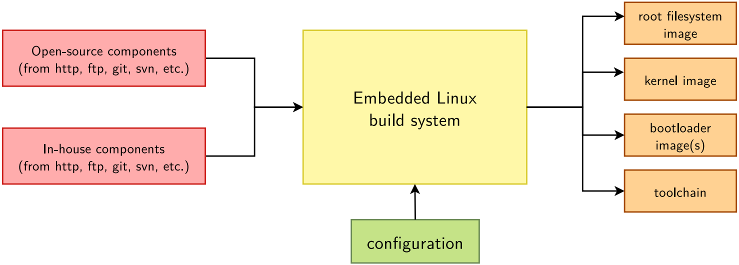
# Complexity of user space integration



# System integration: several possibilities

| | **Pros** | **Cons** |
|---|---|---|
| **Building everything manually** | Full flexibility<br>Learning experience | Dependency hell<br>Need to understand a lot of details<br>Version compatibility<br>Lack of reproducibility |
| **Binary distribution**<br>Debian, Ubuntu, Fedora, etc. | Easy to create and extend | Hard to customize<br>Hard to optimize (boot time, size)<br>Hard to rebuild the full system from source<br>Large system<br>Uses native compilation (slow)<br>No well-defined mechanism to generate an image<br>Lots of mandatory dependencies<br>Not available for all architectures |
| **Build systems**<br>Buildroot, Yocto, PTXdist, etc. | Nearly full flexibility<br>Built from source: customization and optimization are easy<br>Fully reproducible<br>Uses cross-compilation<br>Have embedded specific packages not necessarily in desktop distros<br>Make more features optional | Not as easy as a binary distribution<br>Build time |

# Embedded Linux build system: principle




- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines
- Recipes for building components → easy

# Embedded Linux build system: tools

- A wide range of solutions: Yocto/OpenEmbedded, PTXdist, Buildroot, LTIB, OpenBricks, OpenWRT, and more.
- Today, two solutions are emerging as the most popular ones
  - **Yocto/OpenEmbedded**
    Builds a complete Linux distribution with binary packages. Powerful, but somewhat complex, and quite steep learning curve.
  - **Buildroot**
    Builds a root filesystem image, no binary packages. Much simpler to use, understand and modify.

# Introduction to Buildroot

***free electrons***

Embedded Linux Experts


Corrections, suggestions, contributions and translations are welcome!

# Buildroot at a glance

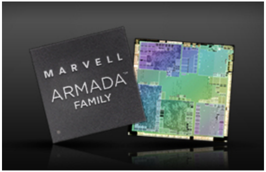
- Can build a toolchain, a rootfs, a kernel, a bootloader
- **Easy to configure**: menuconfig, xconfig, etc.
- **Fast**: builds a simple root filesystem in a few minutes
- Easy to understand: written in make, extensive documentation
- **Small** root filesystem, starting at 2 MB
- **1600+ packages** for user space libraries/apps available
- **Many architectures** supported
- **Well-known technologies**: *make* and *kconfig*
- Vendor neutral
- Active community, regular releases
  - The present slides cover *Buildroot 2015.08*. There may be some differences if you use older or newer Buildroot versions.
- `http://buildroot.org`

# Buildroot design goals

- Buildroot is designed with a few key goals:
  - Simple to use
  - Simple to customize
  - Reproducible builds
  - Small root filesystem
  - Relatively fast boot
  - Easy to understand
- Some of these goals require to not necessarily support all possible features
- They are some more complicated and featureful build systems available (Yocto Project, OpenEmbedded)

# Who's using Buildroot?

- **System makers**
  - Google
  - Barco
  - Rockwell Collins
- **Processor vendors**
  - Imagination Technologies
  - Marvell
  - Atmel
  - Analog Devices
- Many companies when doing *R&D* on products
- Many, many **hobbyists** on development boards: Raspberry Pi, BeagleBone Black, etc.


# Getting Buildroot

- Stable Buildroot releases are published every three months.
- Tarballs are available for each stable release
  - `http://buildroot.org/downloads/`
- However, it is generally more convenient to clone the Git repository
  - Allows to clearly identify the changes you make to the Buildroot source code
  - Simplifies the upstreaming of the Buildroot changes
  - `git clone git://git.busybox.net/buildroot`
  - Git tags available for every stable release.

# Using Buildroot

- Implemented in `make`
  - With a few helper shell scripts
- All interaction happens by calling `make` in the main Buildroot sources directory.

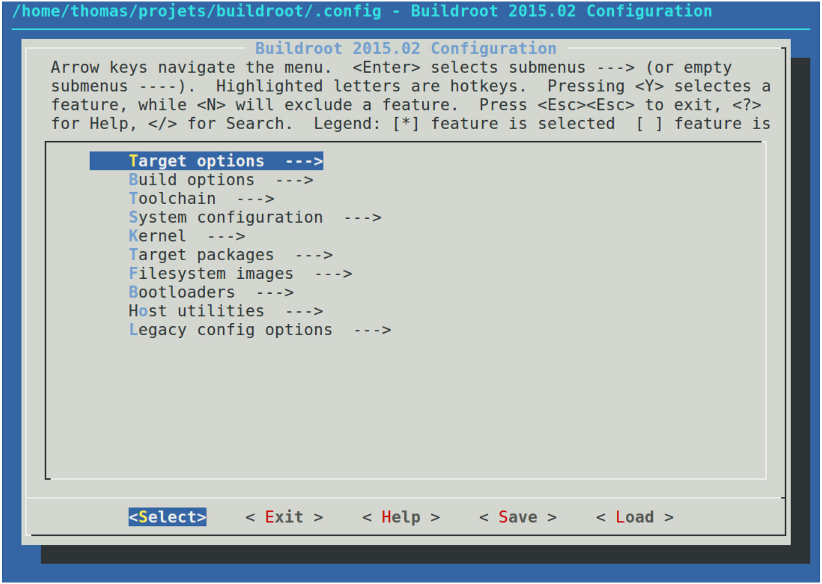
```
$ cd buildroot/
$ make help
```

- No need to run as `root`, Buildroot is designed to be executed with normal user privileges.
  - Running as root is even strongly discouraged!

# Configuring Buildroot

- Like the Linux kernel, uses *Kconfig*
- A choice of configuration interfaces:
  - `make menuconfig`
  - `make nconfig`
  - `make xconfig`
  - `make gconfig`
- Make sure to install the relevant libraries in your system (*ncurses* for menuconfig/nconfig, *Qt* for xconfig, *Gtk* for gconfig)

# Main menuconfig menu



# Running the build

- As simple as:

```
$ make
```

- Often useful to keep a log of the build output, for analysis or investigation:

```
$ make 2>&1 | tee build.log
```

# Build results

- The build results are located in `output/images`
- Depending on the configuration, this directory will contain:
  - One or several root filesystem images, in various formats
  - One kernel image, possibly one or several Device Tree blobs
  - One or several bootloader images
- There is no standard way to install the images on any given device
  - Those steps are very device specific
  - Buildroot provides some tools for specific platforms (e.g.: SAM-BA for Atmel, imx-usb-loader for i.MX6, etc.)

# Practical lab - Basic Buildroot usage

- Get Buildroot
- Configure a minimal system with Buildroot for the BeagleBone Black
- Do the build
- Prepare the BeagleBone Black for usage
- Flash and test the generated system

# Managing the build and the configuration

***free electrons***

Embedded Linux Experts

# Default build organization

- By default:
  - All the build output goes into a directory called `output/` within the top-level Buildroot source directory.
    - `O = output`
  - The configuration file is stored as `.config` in the top-level Buildroot source directory.
    - `CONFIG_DIR = $(TOPDIR)`
    - `TOPDIR = $(shell pwd)`
  - `buildroot/`
    - `.config`
    - `arch/`
    - `package/`
    - `output/`
    - `fs/`
    - ...

# Out of tree build: introduction

- Out of tree build allows to use an output directory different than `output/`
- Useful to build different Buildroot configurations with the same source tree.
- Customization of the output directory done by passing `O=/path/to/directory` on the command line.
- Configuration file stored inside the `$(O)` directory, as opposed to inside the Buildroot sources for the in-tree build case.
- `project/`
  - `buildroot/`, Buildroot sources
  - `foo-output/`, output of a first project
    - `.config`
  - `bar-output/`, output of a second project
    - `.config`

# Out of tree build: using

- To start an out of tree build, two solutions:
  - From the Buildroot source tree, simplify specify a `O=` variable:

```
make O=../foo-output/ menuconfig
```

  - From an empty output directory, specify `O=` and the path to the Buildroot source tree:

```
make -C ../buildroot/ O=$(pwd) menuconfig
```

- Once one out of tree operation has been done (`menuconfig`, loading a defconfig, etc.), Buildroot creates a small wrapper `Makefile` in the output directory.
- This wrapper `Makefile` then avoids the need to pass `O=` and the path to the Buildroot source tree.

# Out of tree build: example

1. You are in your Buildroot source tree:

```
$ ls
arch board boot ... Makefile ... package ...
```

2. Create a new output directory, and move to it:

```
$ mkdir ../foobar-output
$ cd ../foobar-output
```

3. Start a new Buildroot configuration:

```
$ make -C ../buildroot O=$(pwd) menuconfig
```

4. Start the build (passing `O=` and `-C` no longer needed thanks to the wrapper):

```
$ make
```

5. Adjust the configuration again, restart the build, clean the build:

```
$ make menuconfig
$ make
$ make clean
```

# Full config file vs. *defconfig*

- The `.config` file is a *full* config file: it contains the value for all options (except those having unmet dependencies)
- The default `.config`, without any customization, has 2655 lines (as of Buildroot 2015.08)
  - Not very practical for reading and modifying by humans.
- A *defconfig* stores only the values for options for which the non-default value is chosen.
  - Much easier to read
  - Can be modified by humans
  - Can be used for automated construction of configurations

# *defconfig*: example

- For the default Buildroot configuration, the *defconfig* is empty: everything is the default.
- If you change the architecture to be ARM, the *defconfig* is just one line:

```
BR2_arm=y
```

- If then you also enable the `stress` package, the *defconfig* will be just two lines:

```
BR2_arm=y
BR2_PACKAGE_STRESS=y
```

# Using and creating a *defconfig*

- To use a *defconfig*, copying it to `.config` is not sufficient as all the missing (default) options need to be expanded.
- Buildroot allows to load *defconfig* stored in the `configs/` directory, by doing: `make <foo>_defconfig`
  - It overwrites the current `.config`, if any
- To create a *defconfig*, run:
  `make savedefconfig`
  - Saved in the file pointed by the `BR2_DEFCONFIG` configuration option
  - By default, points to `defconfig` in the current directory if the configuration was started from scratch, or points to the original *defconfig* if the configuration was loaded from a defconfig.
  - Move it to `configs/` to make it easily loadable with `make <foo>_defconfig`.

# Existing *defconfigs*

- Buildroot comes with a number of existing *defconfigs* for various publicly available hardware platforms:
  - RaspberryPi, BeagleBone Black, CubieBoard, Atmel evaluation boards, Minnowboard, various i.MX6 boards
  - QEMU emulated platforms
- List them using `make help` (changed to `make list-defconfigs` since Buildroot 2015.05)
- Minimal *defconfigs*: only build a toolchain, bootloader, kernel and minimal root filesystem.

```
$ make qemu_arm_vexpress_defconfig
$ make
```

- Additional instructions often available in `board/<boardname>`, e.g.: `board/qemu/arm-vexpess/readme.txt`.

# Assembling a *defconfig* (1/2)

- *defconfigs* are trivial text files, one can use simple concatenation to assemble them from fragments.

platform1.frag

```
BR2_arm=y
BR2_TOOLCHAIN_BUILDROOT_WCHAR=y
BR2_GCC_VERSION_4_9_X=y
```

platform2.frag

```
BR2_mipsel=y
BR2_TOOLCHAIN_EXTERNAL=y
BR2_TOOLCHAIN_EXTERNAL_CODESOURCERY_MIPS201405=y
```

packages.frag

```
BR2_PACKAGE_STRESS=y
BR2_PACKAGE_MTD=y
BR2_PACKAGE_LIBCONFIG=y
```

# Assembling a *defconfig* (2/2)

debug.frag

```
BR2_ENABLE_DEBUG=y
BR2_PACKAGE_STRACE=y
```

Build a release system for *platform1*

```
$ ./support/kconfig/merge_config.sh platform1.frag packages.frag > \
        .config
$ make olddefconfig
$ make
```

Build a debug system for *platform2*

```
$ ./support/kconfig/merge_config.sh platform2.frag packages.frag \
        debug.frag > .config
$ make olddefconfig
$ make
```

- `olddefconfig` expands a minimal *defconfig* to a full `.config`
- Saving fragments is not possible; it must be done manually from an existing *defconfig*

# Other building tips

- Cleaning targets
  - Cleaning all the build output, but keeping the configuration file:

```
$ make clean
```

  - Cleaning everything, including the configuration file, and downloaded file if at the default location:

```
$ make distclean
```

- Verbose build
  - By default, Buildroot hides a number of commands it runs during the build, only showing the most important ones.
  - To get a fully verbose build, pass V=1:

```
$ make V=1
```

  - Passing V=1 also applies to packages, like the Linux kernel, busybox...

# Buildroot source and build trees

***free electrons***

Embedded Linux Experts

# Source tree

# Source tree (1/5)

- `Makefile`
  - top-level `Makefile`, handles the configuration and general orchestration of the build
- `Config.in`
  - top-level `Config.in`, main/general options. Includes many other `Config.in` files
- `arch/`
  - `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 (2/5)

- `toolchain/`
  - 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
- `system/`
  - `skeleton/` the rootfs skeleton
  - `Config.in`, options for system-wide features like init system, `/dev` handling, etc.
- `linux/`
  - `linux.mk`, the Linux kernel package

# Source tree (3/5)

- package/
  - all the user space packages (1600+)
  - busybox/, gcc/, qt5/, etc.
  - pkg-generic.mk, core package infrastructure
  - pkg-cmake.mk, pkg-autotools.mk, pkg-perl.mk, etc. Specialized package infrastructures
- fs/
  - logic to generate filesystem images in various formats
  - common.mk, common logic
  - cpio/, ext2/, squashfs/, tar/, ubifs/, etc.
- boot/
  - bootloader packages
  - at91bootstrap3/, barebox/, grub/, syslinux/, uboot/, etc.

# Source tree (4/5)

- `configs/`
  - default configuration files for various platforms
  - similar to kernel defconfigs
  - `atmel_xplained_defconfig`, `beaglebone_defconfig`, `raspberrypi_defconfig`, etc.
- `board/`
  - board-specific files (kernel configuration files, kernel patches, image flashing scripts, etc.)
  - typically go together with a *defconfig* in `configs/`
- `support/`
  - misc utilities (kconfig code, libtool patches, download helpers, and more.)

# Source tree (5/5)

- docs/
  - Buildroot documentation
  - Written in AsciiDoc, can generate HTML, PDF, TXT versions: `make manual`
  - 90 pages PDF document
  - Also available pre-generated online.
  - http://buildroot.org/downloads/manual/manual.html

# Build tree

# Build tree: $(O)

- `output/`
- 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)

# Build tree: $(O)/build

- output/
  - build/
    - buildroot-config/
    - busybox-1.22.1/
    - host-pkgconf-0.8.9/
    - kmod-1.18/
    - build-time.log
  - 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

# Build tree: $(O)/host

- output/
  - 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
  - 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`
  - `<tuple>` is an identifier of the architecture, vendor, operating system, C library and ABI. E.g: `arm-unknown-linux-gnueabihf`.
  - Variable for the *sysroot*: `STAGING_DIR`

# Build tree: $(O)/staging

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

# Build tree: $(O)/target

- output/
  - target/
    - bin/
    - etc/
    - lib/
    - usr/bin/
    - usr/lib/
    - usr/share/
    - usr/sbin/
    - THIS_IS_NOT_YOUR_ROOT_FILESYSTEM
    - ...
  - 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

# Build tree: $(O)/images

- output/
  - images/
    - zImage
    - armada-370-mirabox.dtb
    - rootfs.tar
    - rootfs.ubi
  - Contains the final images: kernel image, bootloader image, root filesystem image(s)
  - Variable: BINARIES_DIR

# Build tree: $(O)/graphs

- `output/`
  - `graphs/`
  - Visualization of Buildroot operation: dependencies between packages, time to build the different packages
  - `make graph-depends`
  - `make graph-build`
  - Variable: `GRAPHS_DIR`
  - See the section *Analyzing the build* later in this training.

# Build tree: $(O)/legal-info

- output/
  - 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

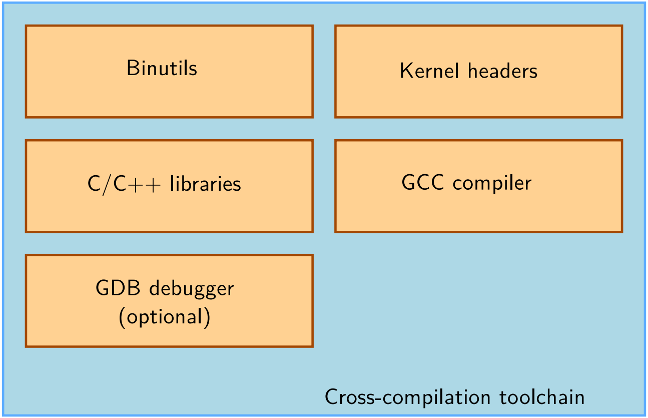
# Toolchains in Buildroot

***free electrons***

Embedded Linux Experts

# What is a cross-compilation toolchain?

- A set of tools to build and debug code for a target architecture, from a machine running a different architecture.
- Example: building code for ARM from a x86-64 PC.



# Two possibilities for the toolchain

- Buildroot offers two choices for the toolchain, called **toolchain backends**:
  - The **internal toolchain** backend, where Buildroot builds the toolchain entirely from source
  - The **external toolchain** backend, where Buildroot uses a existing pre-built toolchain
- Selected from `Toolchain` → `Toolchain type`.

# Internal toolchain backend

- Makes Buildroot build the entire cross-compilation toolchain from source.
- Provides a lot of flexibility in the configuration of the toolchain.
  - Kernel headers version
  - C library: Buildroot supports uClibc, (e)glibc and musl
    - (e)glibc, the standard C library. Good choice if you don't have tight space constraints ($>=$ 10 MB)
    - uClibc and musl, smaller C libraries. uClibc supports non-MMU architectures. Good for very small systems ($<$ 10 MB).
  - Different versions of binutils and gcc. Keep the default versions unless you have specific needs.
  - Numerous toolchain options: C++, LTO, OpenMP, libmudflap, graphite, and more depending on the selected C library.
- Building a toolchain takes quite some time: 15-20 minutes on moderately recent machines.

# Internal toolchain backend: result

- host/usr/bin/<tuple>-<tool>, the cross-compilation tools: compiler, linker, assembler, and more.
- host/usr/<tuple>/
  - sysroot/usr/include/, the kernel headers and C library headers
  - sysroot/lib/ and sysroot/usr/lib/, C library and gcc runtime
  - include/c++/, C++ library headers
  - lib/, host libraries needed by gcc/binutils
- target/
  - lib/ and usr/lib/, C and C++ libraries
- The compiler is configured to:
  - generate code for the architecture, variant, FPU and ABI selected in the Target options
  - look for libraries and headers in the *sysroot*
  - no need to pass weird gcc flags!

# External toolchain backend possibilities

- Allows to re-use existing pre-built toolchains
- Great to:
  - save the build time of the toolchain
  - use vendor provided toolchain that are supposed to be reliable
- Several options:
  - Use an existing toolchain profile known by Buildroot
  - Download and install a custom external toolchain
  - Directly use a pre-installed custom external toolchain

# Existing external toolchain profile

- Buildroot already knows about a wide selection of publicly available toolchains.
- Toolchains from Linaro (ARM and AArch64), Mentor Graphics (ARM, MIPS, NIOS-II, PowerPC, SuperH, x86, x86-64), Analog Devices (Blackfin) and the musl project.
- In such cases, Buildroot is able to download and automatically use the toolchain.
- It already knows the toolchain configuration: C library being used, kernel headers version, etc.
- Additional profiles can easily be added.

# Custom external toolchains

- If you have a custom external toolchain, select `Custom toolchain` in `Toolchain`.
- Buildroot can download and extract it for you
  - Convenient to share toolchains between several developers
  - Option `Toolchain to be downloaded and installed` in `Toolchain origin`
  - The URL of the toolchain tarball is needed
- Or Buildroot can use an already installed toolchain
  - Option `Pre-installed toolchain` in `Toolchain origin`
  - The local path to the toolchain is needed.
- In both cases, you will have to tell Buildroot the configuration of the toolchain: C library, kernel headers version, etc.
  - Buildroot needs this information to know which packages can be built with this toolchain
  - Buildroot will check those values at the beginning of the build

# External toolchain example configuration

```
                                     Toolchain
Enter> selects submenus ---> (or empty submenus ----).  Highlighted letters are hotkeys.  Pressir
ll exclude a feature.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*] featu
d

    Toolchain type (External toolchain)  --->
    Toolchain (Custom toolchain)  --->
    Toolchain origin (Toolchain to be downloaded and installed)  --->
(http://autobuild.buildroot.org/toolchains/tarballs/br-arm-full-2015.02.tar.bz2) Toolchain URL
($(ARCH)-linux) Toolchain prefix (NEW)
    External toolchain kernel headers series (3.18.x)  --->
    External toolchain C library (uClibc)  --->
-*- Toolchain has WCHAR support?
[*] Toolchain has locale support?
[*] Toolchain has threads support? (NEW)
[ ]   Toolchain has threads debugging support?
[*]   Toolchain has NPTL threads support? (NEW)
[ ] Toolchain has SSP support? (NEW)
[*] Toolchain has RPC support?
[*] Toolchain has C++ support?
()  Extra toolchain libraries to be copied to target (NEW)
[ ] Copy gdb server to the Target (NEW)
[ ] Build cross gdb for the host (NEW)
[ ] Purge unwanted locales (NEW)
[*] Enable MMU support (NEW)
()  Target Optimizations (NEW)
()  Target linker options (NEW)
[ ] Register toolchain within Eclipse Buildroot plug-in (NEW)
```

# External toolchain: result

- `host/opt/ext-toolchain`, where the original toolchain tarball is extracted. Except when a local pre-installed toolchain is used.
- `host/usr/bin/<tuple>-<tool>`, symbolic links to the cross-compilation tools in their original location. Except the compiler, which points to a wrapper program.
- `host/usr/<tuple>/`
  - `sysroot/usr/include/`, the kernel headers and C library headers
  - `sysroot/lib/` and `sysroot/usr/lib/`, C library and gcc runtime
  - `include/c++/`, C++ library headers
- `target/`
  - `lib/` and `usr/lib/`, C and C++ libraries
- The wrapper takes care of passing the appropriate flags to the compiler.
  - Mimics the internal toolchain behavior

# Kernel headers version

- One option in the toolchain menu is particularly important: the kernel headers version.
- When building user space programs, libraries or the C library, kernel headers are used to know how to interface with the kernel.
- This kernel/user space interface is **backward compatible**, but can introduce new features.
- It is therefore important to use kernel headers that have a version **equal or older** than the kernel version running on the target.
- With the internal toolchain backend, choose an appropriate kernel headers version.
- With the external toolchain backend, beware when choosing your toolchain.

# Other toolchain menu options

- The toolchain menu offers a few other options:
  - *Purge unwanted locales*
    - This allows to get rid of translation files, when not needed. They consume quite a lot of disk space.
  - *Target optimizations*
    - Allows to pass additional compiler flags when building target packages
    - Do not pass flags to select a CPU or FPU, these are already passed by Buildroot
    - Be careful with the flags you pass, they affect the entire build
  - *Target linker options*
    - Allows to pass additional linker flags when building target packages
  - gdb and Eclipse related options
    - Covered in our *Application development* section later.

# Managing the Linux kernel configuration

***free electrons***

Embedded Linux Experts


Corrections, suggestions, contributions and translations are welcome!

# Introduction

- The Linux kernel itself uses *kconfig* to define its configuration
- Buildroot cannot replicate all Linux kernel configuration options in its `menuconfig`
- Defining the Linux kernel configuration therefore needs to be done in a special way.
- Note: while described with the example of the Linux kernel, this discussion is also valid for other packages using *kconfig*: `barebox`, `uclibc`, `busybox` and in the near future, `u-boot`.

# Defining the configuration

- In the `Kernel` menu in `menuconfig`, after selecting the kernel version, you have two options to define the kernel configuration:
  - `Use a defconfig`
    - Will use a *defconfig* provided within the kernel sources
    - Available in `arch/<ARCH>/configs` in the kernel sources
    - Used unmodified by Buildroot
    - Good starting point
  - `Use a custom config file`
    - Allows to give the path to either a full `.config`, or a minimal *defconfig*
    - Usually what you will use, so that you can have a custom configuration

# Changing the configuration

- Running one of the Linux kernel configuration interfaces:
  - `make linux-menuconfig`
  - `make linux-nconfig`
  - `make linux-xconfig`
  - `make linux-gconfig`
- Will load either the defined kernel *defconfig* or custom configuration file, and start the corresponding Linux kernel configuration interface.
- Changes made are only made in `$(O)/build/linux-<version>/`, i.e. they are not preserved across a clean rebuild.
- To save them:
  - `make linux-update-config`, to save a full config file
  - `make linux-update-defconfig`, to save a minimal defconfig. Available since Buildroot 2015.05.
  - Only works if a *custom configuration file* is used

# Typical flow

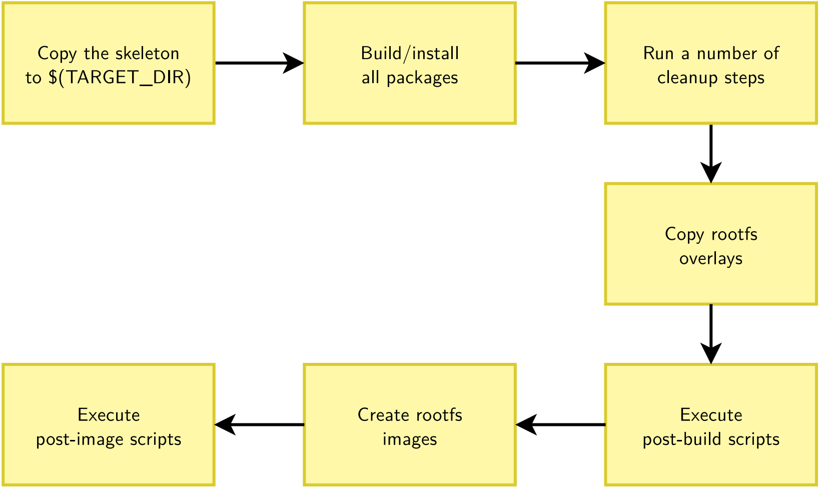
1. `make menuconfig`
   - Start with a *defconfig* from the kernel, say `mvebu_v7_defconfig`
2. Run `make linux-menuconfig` to customize the configuration
3. Do the build, test, tweak the configuration as needed.
4. You cannot do `make linux-update-config,defconfig`, since the Buildroot configuration points to a kernel *defconfig*
5. `make menuconfig`
   - Change to a custom configuration file. There's no need for the file to exist, it will be created by Buildroot.
6. `make linux-update-defconfig`
   - Will create your custom configuration file, as a minimal *defconfig*

# Root filesystem in Buildroot

***free electrons***

Embedded Linux Experts

# Overall rootfs construction steps



# Root filesystem skeleton

- The base of a Linux root filesystem: Unix directory hierarchy, a few configuration files and scripts in `/etc`. No programs or libraries.
- First thing to get copied to `$(TARGET_DIR)` at the beginning of the build.
- By default (`BR2_ROOTFS_SKELETON_DEFAULT=y`), the one in `system/skeleton` is used.
- A custom *skeleton* can be used, through the `BR2_ROOTFS_SKELETON_CUSTOM` and `BR2_ROOTFS_SKELETON_CUSTOM_PATH` options.
  - Not recommended though: the *skeleton* is only copied once at the beginning of the build, and the base is usually good for most projects.
  - Use *rootfs overlays* or *post-build scripts* for customization.

# Installation of packages

- All the selected target packages will be built (can be Busybox, Qt, OpenSSH, lighttpd, and many more)
- Most of them will install files in `$(TARGET_DIR)`: programs, libraries, fonts, data files, configuration files, etc.
- This is really the step that will bring the vast majority of the files in the root filesystem.
- Covered in more details in the section about creating your own Buildroot packages.

# Cleanup step

- Once all packages have been installed, a cleanup step is executed to reduce the size of the root filesystem.
- It mainly involves:
  - Removing header files, pkg-config files, CMake files, static libraries, man pages, documentation.
  - Stripping all the programs and libraries using `strip`, to remove unneeded information. Depends on `BR2_ENABLE_DEBUG` and `BR2_STRIP_*` options.
  - Additional specific clean up steps: clean up unneeded Python files when Python is used, etc. See `TARGET_FINALIZE_HOOKS` in the Buildroot code.

# Root filesystem overlay

- To customize the contents of your root filesystem, to add configuration files, scripts, symbolic links, directories or any other file, one possible solution is to use a **root filesystem overlay**.
- A *root filesystem overlay* is simply a directory whose contents will be **copied over the root filesystem**, after all packages have been installed. Overwriting files is allowed.
- The option `BR2_ROOTFS_OVERLAY` contains a space-separated list of overlay paths.

```
$ grep ^BR2_ROOTFS_OVERLAY .config
BR2_ROOTFS_OVERLAY="board/myproject/rootfs-overlay"
$ find -type f board/myproject/rootfs-overlay
board/myproject/rootfs-overlay/etc/ssh/sshd_config
board/myproject/rootfs-overlay/etc/init.d/S99myapp
```

# Post-build scripts

- Sometimes a *root filesystem overlay* is not sufficient: you can use **post-build scripts**.
- Can be used to **customize existing files**, **remove unneeded files** to save space, add **new files that are generated dynamically** (build date, etc.)
- Executed before the root filesystem image is created. Can be written in any language, shell scripts are often used.
- `BR2_ROOTFS_POST_BUILD_SCRIPT` contains a space-separated list of post-build script paths.
- `$(TARGET_DIR)` path passed as first argument, additional arguments can be passed in the `BR2_ROOTFS_POST_SCRIPT_ARGS` option.
- Various environment variables are available:
  - `BR2_CONFIG`, path to the Buildroot .config file
  - `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`, `BUILD_DIR`, `BINARIES_DIR`, `BASE_DIR`

# Post-build script: example

board/myproject/post-build.sh

```
#!/bin/sh
TARGET_DIR=$1
BOARD_DIR=board/myproject/

# Generate a file identifying the build (git commit and build date)
echo $(git describe) $(date +%Y-%M-%d-%H:%m:%S) > \
    $TARGET_DIR/etc/build-id

# Create /applog mountpoint, and adjust /etc/fstab
mkdir -p $TARGET_DIR/applog
grep -q "^/dev/mtdblock7" $TARGET_DIR/etc/fstab || \
    echo "/dev/mtdblock7\t\t/applog\tjffs2\tdefaults\t\t0\t0" >> \
    $TARGET_DIR/etc/fstab

# Remove unneeded files
rm -rf $TARGET_DIR/usr/share/icons/bar
```

Buildroot configuration

```
BR2_ROOTFS_POST_BUILD_SCRIPT="board/myproject/post-build.sh"
```

# Generating the filesystem images

- In the `Filesystem images` menu, you can select which filesystem image formats to generate.
- To generate those images, Buildroot will generate a shell script that:
  - **Changes the owner** of all files to `0:0` (root user)
  - Takes into account the global **permission and device tables**, as well as the per-package ones.
  - Takes into account the **global and per-package users tables**.
  - Runs the **filesystem image generation utility**, which depends on each filesystem type (`genext2fs`, `mkfs.ubifs`, `tar`, etc.)
- This script is executed using a tool called *fakeroot*
  - Allows to fake being root so that permissions and ownership can be modified, device files can be created, etc.

# Permission table

- By default, all files are owned by the `root` user, and the permissions with which they are installed in `$(TARGET_DIR)` are preserved.
- To customize the ownership or the permission of installed files, one can create one or several **permission tables**
- `BR2_ROOTFS_DEVICE_TABLE` contains a space-separated list of permission table files. The option name contains *device* for backward compatibility reasons only.
- The `system/device_table.txt` file is used by default.
- Packages can also specify their own permissions. See the *Advanced package aspects* section for details.

Permission table example

```
#<name>    <type>  <mode>  <uid>   <gid>   <major> <minor> <start> <inc>   <count>
/dev       d       755     0       0       -       -       -       -       -
/tmp       d       1777    0       0       -       -       -       -       -
/var/www   d       755     33      33      -       -       -       -       -
```

# Device table

- When the system is using a static /dev, one may need to create additional *device nodes*
- Done using one or several **device tables**
- BR2_ROOTFS_STATIC_DEVICE_TABLE contains a space-separated list of device table files.
- The system/device_table_dev.txt file is used by default.
- Packages can also specify their own device files. See the *Advanced package aspects* section for details.

Device table example

```
# <name>        <type> <mode> <uid>  <gid>  <major> <minor> <start> <inc>  <count>
/dev/mem        c      640    0      0      1       1       0       0      -
/dev/kmem       c      640    0      0      1       2       0       0      -
/dev/i2c-       c      666    0      0      89      0       0       1      4
```

# Users table

- One may need to add specific Unix users and groups in addition to the ones available in the default skeleton.
- BR2_ROOTFS_USERS_TABLES is a space-separated list of user tables.
- Packages can also specify their own users. See the *Advanced package aspects* section for details.

Users table example

```
# <username> <uid> <group> <gid> <password> <home>    <shell> <groups>    <comment>
foo          -1    bar     -1    !=blabla   /home/foo /bin/sh alpha,bravo Foo user
test         8000  wheel   -1    =          -         /bin/sh -           Test user
```

# Post-image scripts

- Once all the filesystem images have been created, at the very end of the build, **post-image** scripts are called.
- They allow to do any custom action at the end of the build. For example:
  - Extract the root filesystem to do NFS booting
  - Generate a final firmware image
  - Start the flashing process
- `BR2_ROOTFS_POST_IMAGE_SCRIPT` is a space-separated list of *post-image* scripts to call.
- Post-image scripts are called:
  - from the Buildroot source directory
  - with the `$(BINARIES_DIR)` path as first argument
  - with the contents of the `BR2_ROOTFS_POST_SCRIPT_ARGS` as other arguments
  - with a number of available environment variables: `BR2_CONFIG`, `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`, `BUILD_DIR`, `BINARIES_DIR` and `BASE_DIR`.

# Init mechanism

- Buildroot supports multiple *init* implementations:
  - **Busybox init**, the default. Simplest solution.
  - **sysvinit**, the old style featureful *init* implementation
  - **systemd**, the new generation init system
- Selecting the *init* implementation in the `System configuration` menu will:
  - Ensure the necessary packages are selected
  - Make sure the appropriate init scripts or configuration files are installed by packages. See *Advanced package aspects* for details.

# /dev management method

- Buildroot supports four methods to handle the `/dev` directory:
  - Using **devtmpfs**. `/dev` is managed by the kernel *devtmpfs*, which creates device files automatically. Requires kernel 2.6.32+. Default option.
  - Using **static /dev**. This is the old way of doing `/dev`, not very practical.
  - Using **mdev**. `mdev` is part of Busybox and can run custom actions when devices are added/removed. Requires *devtmpfs* kernel support.
  - Using **eudev**. Forked from `systemd`, allows to run custom actions. Requires *devtmpfs* kernel support.
- When *systemd* is used, the only option is *udev* from *systemd* itself.

# Other customization options

- There are various other options to customize the root filesystem:
  - **getty** options, to run a login prompt on a serial port or screen
  - **hostname** and **banner** options
  - **DHCP network** on one interface (for more complex setups, use an *overlay*)
  - **root password**
  - **timezone** installation and selection

# Deploying the images

- By default, Buildroot simply stores the different images in `$(O)/images`
- It is up to the user to deploy those images to the target device.
- Possible solutions:
  - For removable storage (SD card, USB keys):
    - manually create the partitions and extract the root filesystem as a tarball to the appropriate partition.
    - use a tool like `genimage` to create a complete image of the media, including all partitions
  - For NAND flash:
    - Transfer the image to the target, and flash it.
  - NFS booting
  - initramfs

# Deploying the image: NFS booting

- Many people try to use `$(O)/target` directly for NFS booting
  - This cannot work, due to permissions/ownership being incorrect
  - Clearly explained in the `THIS_IS_NOT_YOUR_ROOT_FILESYSTEM` file.
- Generate a tarball of the root filesystem
- Use `sudo tar -C /nfs -xf output/images/rootfs.tar` to prepare your NFS share.

# Deploying the image: initramfs

- Another common use case is to use an *initramfs*, i.e. a root filesystem fully in RAM.
  - Convenient for small filesystems, fast booting or kernel development
- Two solutions:
  - `BR2_TARGET_ROOTFS_CPIO=y` to generate a *cpio* archive, that you can load from your bootloader next to the kernel image.
  - `BR2_TARGET_ROOTFS_INITRAMFS=y` to directly include the *initramfs* inside the kernel image. Only available when the kernel is built by Buildroot.

# Practical lab - Root filesystem construction

- Explore the build output
- Customize the root filesystem using a rootfs overlay
- Use a post-build script
- Customize the kernel with patches and additional configuration options
- Add more packages
- Use defconfig files and out of tree build

# Download infrastructure in Buildroot

***free electrons***

Embedded Linux Experts

# 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`

# 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.

# DL_DIR

- Once a file has been downloaded by Buildroot, it is cached in the directory pointed by `$(DL_DIR)`
- By default, `$(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.

# 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 stored in a temporary location
- 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
- `make source-check`
  - Checks whether the upstream site of all downloads needed for the current configuration are still available.

# GNU Make 101

***free electrons***

Embedded Linux
Experts

# 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*
- http://www.gnu.org/software/make/manual/make.html
- http://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`, prepend 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)/usr/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

***free electrons***

Embedded Linux Experts


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.
- 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_WCHAR # util-linux
        depends on BR2_USE_MMU # util-linux
        depends on BR2_TOOLCHAIN_HAS_THREADS
        select BR2_PACKAGE_ACL
        select BR2_PACKAGE_ATTR
        select BR2_PACKAGE_E2FSPROGS
        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/in...

comment "btrfs-progs needs a toolchain w/ wchar, threads"
        depends on BR2_USE_MMU
        depends on !BR2_USE_WCHAR || \
               !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_USE_WCHAR` and `depends on BR2_TOOLCHAIN_HAS_THREADS`, because the package requires wide-char and 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

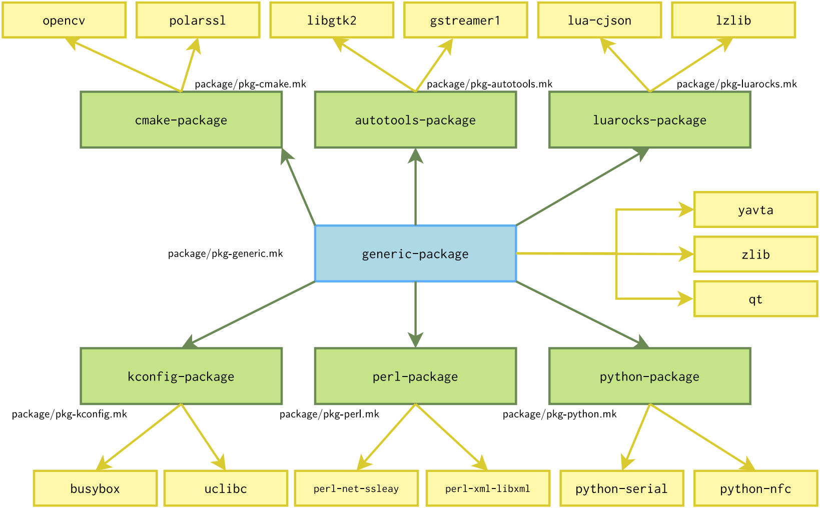
endif
```

# 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*, *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.

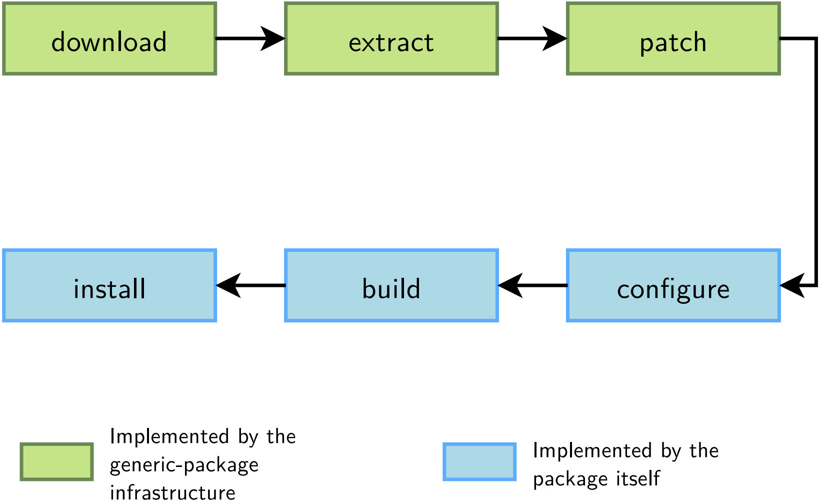
# Package infrastructures



# `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.

# generic-package: steps



# 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 = http://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)
```

# 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
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: example (1)

eeprog.mk

```
EEPROG_VERSION = 0.7.6
EEPROG_SITE = http://www.codesink.org/download
EEPROG_LICENSE = GPLv2+
EEPROG_LICENSE_FILES = eeprog.c

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

define EEPROG_INSTALL_TARGET_CMDS
        $(INSTALL) -m 0755 -D $(@D)/eeprog $(TARGET_DIR)/usr/bin/eeprog
endef

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

# Describing actions: example (2)

zlib.mk

```
ZLIB_VERSION = 1.2.8
ZLIB_SOURCE = zlib-$(ZLIB_VERSION).tar.xz
ZLIB_SITE = http://downloads.sourceforge.net/project/libpng/zlib/$(ZLIB_VERSION)
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

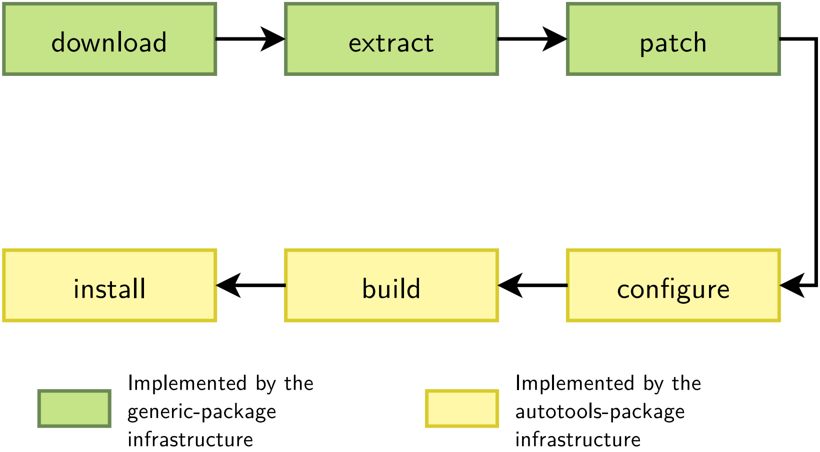
$(eval $(generic-package))
```

# 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_VERSION = 0.1.6
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

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

# More complicated autotools-package example

```
POPPLER_VERSION = 0.32.0
POPPLER_SOURCE = poppler-$(POPPLER_VERSION).tar.xz
POPPLER_SITE = http://poppler.freedesktop.org
POPPLER_DEPENDENCIES = fontconfig
POPPLER_LICENSE = GPLv2+
POPPLER_LICENSE_FILES = COPYING
POPPLER_INSTALL_STAGING = YES
POPPLER_CONF_OPTS = \
   --with-font-configuration=fontconfig

ifeq ($(BR2_PACKAGE_LCMS2),y)
POPPLER_CONF_OPTS += --enable-cms=lcms2
POPPLER_DEPENDENCIES += lcms2
else
POPPLER_CONF_OPTS += --enable-cms=none
endif

ifeq ($(BR2_PACKAGE_TIFF),y)
POPPLER_CONF_OPTS += --enable-libtiff
POPPLER_DEPENDENCIES += tiff
else
POPPLER_CONF_OPTS += --disable-libtiff
endif

[...]
```

```
[...]

ifeq ($(BR2_PACKAGE_POPPLER_QT),y)
POPPLER_DEPENDENCIES += qt
POPPLER_CONF_OPTS += --enable-poppler-qt4
else
POPPLER_CONF_OPTS += --disable-poppler-qt4
endif

ifeq ($(BR2_PACKAGE_OPENJPEG),y)
POPPLER_DEPENDENCIES += openjpeg
POPPLER_CONF_OPTS += \
   -enable-libopenjpeg=openjpeg1
else
POPPLER_CONF_OPTS += -enable-libopenjpeg=none
endif

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

# python-package infrastructure

# Python package infrastructure: basics

- Modules for the Python language often use *distutils* or *setuptools* as their build/installation system.
- Buildroot provides a `python-package` infrastructure for such packages.
- Supports all the `generic-package` metadata information (source, site, license, etc.)
- Adds a mandatory variable `<pkg>_SETUP_TYPE`, which must be set to either `distutils` or `setuptools`
- And several optional variables to further adjust the build: `<pkg>_ENV`, `<pkg>_BUILD_OPTS`, `<pkg>_INSTALL_TARGET_OPTS`, `<pkg>_INSTALL_STAGING_OPTS`, `<pkg>_INSTALL_OPTS`, `<pkg>_NEEDS_HOST_PYTHON`.

# Python package: simple example

python-serial.mk

```
PYTHON_SERIAL_VERSION = 2.6
PYTHON_SERIAL_SOURCE = pyserial-$(PYTHON_SERIAL_VERSION).tar.gz
PYTHON_SERIAL_SITE = http://pypi.python.org/packages/source/p/pyserial
PYTHON_SERIAL_LICENSE = Python Software Foundation License
PYTHON_SERIAL_LICENSE_FILES = LICENSE.txt
PYTHON_SERIAL_SETUP_TYPE = distutils

$(eval $(python-package))
```

# Python package: more complicated example

python-serial.mk

```
PYTHON_LXML_VERSION = 3.4.2
PYTHON_LXML_SITE = http://lxml.de/files
PYTHON_LXML_SOURCE = lxml-$(PYTHON_LXML_VERSION).tgz
[...]
PYTHON_LXML_SETUP_TYPE = setuptools
PYTHON_LXML_DEPENDENCIES = libxml2 libxslt zlib

PYTHON_LXML_BUILD_OPTS = \
        --with-xslt-config=$(STAGING_DIR)/usr/bin/xslt-config \
        --with-xml2-config=$(STAGING_DIR)/usr/bin/xml2-config

$(eval $(python-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 in the package infrastructure (1)

- 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 in the package infrastructure (2)

- For many variables, when `HOST_BAZ_<var>` is not defined, the package infrastructure uses `BAZ_<var>` instead: source, site, version, license, etc.
  - E.g. defining `<PKG>_SITE` once is sufficient.
- But not for all variables, especially commands
  - E.g. `HOST_<PKG>_BUILD_CMDS` is not inherited from `<PKG>_BUILD_CMDS`
- `HOST_<PKG>_DEPENDENCIES` is handled specially:
  - Derived automatically from `<PKG>_DEPENDENCIES`, after prepending `host-` to all dependencies.
  - `FOO_DEPENDENCIES = bar host-baz` → `HOST_FOO_DEPENDENCIES = host-bar host-baz`.
  - Can be overridden if the dependencies of the host variant are different than the ones of the target variant.

# 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 = 3.0.4
BISON_SOURCE = bison-$(BISON_VERSION).tar.xz
BISON_SITE = $(BR2_GNU_MIRROR)/bison
BISON_LICENSE = GPLv3+
BISON_LICENSE_FILES = COPYING
HOST_BISON_DEPENDENCIES = host-m4

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

# Example 2: a flashing utility

- dfu-util, to reflash devices support the USB DFU protocol. Typically used on a development PC.
- Not used as a build dependency of another package → visible in menuconfig.

package/dfu-util/Config.in.host

```
config BR2_PACKAGE_HOST_DFU_UTIL
        bool "host dfu-util"
        help
          Dfu-util is the host side implementation of the DFU 1.0
          specification of the USB forum. DFU is intended to download
          and upload firmware to devices connected over USB.

          http://dfu-util.gnumonks.org/
```

package/dfu-util/dfu-util.mk

```
DFU_UTIL_VERSION = 0.6
DFU_UTIL_SITE = http://dfu-util.gnumonks.org/releases
DFU_UTIL_LICENSE = GPLv2+
DFU_UTIL_LICENSE_FILES = COPYING

HOST_DFU_UTIL_DEPENDENCIES = host-libusb

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

# Example 3: target and host of the same package

package/e2tools/e2tools.mk

```
E2TOOLS_VERSION = 3158ef18a903ca4a98b8fa220c9fc5c133d8bdf6
E2TOOLS_SITE = $(call github,ndim,e2tools,$(E2TOOLS_VERSION))

# Source coming from GitHub, no configure included.
E2TOOLS_AUTORECONF = YES
E2TOOLS_LICENSE = GPLv2
E2TOOLS_LICENSE_FILES = COPYING
E2TOOLS_DEPENDENCIES = e2fsprogs
E2TOOLS_CONF_ENV = LIBS="-lpthread"
HOST_E2TOOLS_CONF_ENV = LIBS="-lpthread"

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

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

# Advanced package aspects

***free electrons***

Embedded Linux Experts


Corrections, suggestions, contributions and translations are welcome!

# Licensing report

# Licensing report: introduction

- A key aspect of embedded Linux systems is **license compliance**.
- Embedded Linux systems integrate together a number of open-source components, each distributed under its own license.
- The different open-source licenses may have **different requirements**, that must be met before the product using the embedded Linux system starts shipping.
- Buildroot helps in this license compliance process by offering the possibility of generating a number of **license-related information** from the list of selected packages.
- Generated using:

```
$ make legal-info
```

# Licensing report: contents of `legal-info`

- `sources/`, all the source files that are redistributable (tarballs, patches, etc.)
- `buildroot.config`, the Buildroot `.config` file
- `host-manifest.csv`, a CSV file with the list of *host packages*, their version, license, etc.
- `host-licenses/<pkg>/`, the full license text of all *host packages*, per package
- `host-licenses.txt`, the full license text of all *host packages*, in a single file
- `licenses.txt`, the full license text of all *target packages*, in a single file
- `README`
- `licenses/`, the full license text of all *target packages*, per package
- `manifest.csv`, a CSV file with the list of *target packages*, their version, license, etc.

# Including licensing information in packages

- <pkg>_LICENSE
  - Comma-separated **list of license(s)** under which the package is distributed.
  - Free form string, but should if possible use the license codes from `https://spdx.org/licenses/`
  - Can indicate which part is under which license (programs, tests, libraries, etc.)
- <pkg>_LICENSE_FILES
  - Space-separated **list of file paths** from the package source code containing the license text and copyright information
  - Paths relative to the package top-level source directory
- <pkg>_REDISTRIBUTE
  - Boolean indicating whether the package source code can be redistributed or not (part of the `legal-info` output)
  - Defaults to `YES`, can be overridden to `NO`
  - If `NO`, source code is not copied when generating the licensing report

# Licensing information examples

linux.mk

```
LINUX_LICENSE = GPLv2
LINUX_LICENSE_FILES = COPYING
```

acl.mk

```
ACL_LICENSE = GPLv2+ (programs), LGPLv2.1+ (libraries)
ACL_LICENSE_FILES = doc/COPYING doc/COPYING.LGPL
```

owl-linux.mk

```
OWL_LINUX_LICENSE = PROPRIETARY
OWL_LINUX_LICENSE_FILES = LICENSE
OWL_LINUX_REDISTRIBUTE = NO
```

# Patching packages

# Patching packages: why?

- In some situations, it might be needed to patch the source code of certain packages built by Buildroot.
- Useful to:
  - Fix cross-compilation issues
  - Backport bug or security fixes from upstream
  - Integrate new features or fixes not available upstream, or that are too specific to the product being made
- Patches are automatically applied by Buildroot, during the *patch* step, i.e. after extracting the package, but before configuring it.
- Buildroot already comes with a number of patches for various packages, but you may need to add more for your own packages, or to existing packages.

# Patch application ordering

- Overall the patches are applied in this order:
  1. Patches mentioned in the `<pkg>_PATCH` variable of the package `.mk` file. They are automatically downloaded before being applied.
  2. Patches present in the package directory `package/<pkg>/*.patch`
  3. Patches present in the *global patch directories*
- In each case, they are applied:
  - In the order specified in a `series` file, if available
  - Otherwise, in alphabetic ordering

# Patch conventions

- There are a few conventions and best practices that the Buildroot project encourages to use when managing patches
- Their name should start with a sequence number that indicates the ordering in which they should be applied.

ls package/nginx/*.patch

```
0001-auto-type-sizeof-rework-autotest-to-be-cross-compila.patch
0002-auto-feature-add-mechanism-allowing-to-force-feature.patch
0003-auto-set-ngx_feature_run_force_result-for-each-featu.patch
0004-auto-lib-libxslt-conf-allow-to-override-ngx_feature_.patch
0005-auto-unix-make-sys_nerr-guessing-cross-friendly.patch
```

- Each patch should contain a description of what the patch does, and if possible its upstream status.
- Each patch should contain a `Signed-off-by` that identifies the author of the patch.

# Patch example

```
From 81289d1d1adaf5a767a4b4d1309c286468cfd37f Mon Sep 17 00:00:00 2001
From: Samuel Martin <s.martin49@gmail.com>
Date: Thu, 24 Apr 2014 23:27:32 +0200
Subject: [PATCH 1/5] auto/type/sizeof: rework autotest to be cross-compilation
 friendly

Rework the sizeof test to do the checks at compile time instead of at
runtime. This way, it does not break when cross-compiling for a
different CPU architecture.

Signed-off-by: Samuel Martin <s.martin49@gmail.com>
---
 auto/types/sizeof | 42 ++++++++++++++++++++++++++++--------------
 1 file changed, 28 insertions(+), 14 deletions(-)

diff --git a/auto/types/sizeof b/auto/types/sizeof
index 9215a54..c2c3ede 100644
--- a/auto/types/sizeof
+++ b/auto/types/sizeof
@@ -14,7 +14,7 @@ END

 ngx_size=

-cat << END > $NGX_AUTOTEST.c
+cat << _EOF > $NGX_AUTOTEST.c
[...]
```

# Global patch directories

- You can include patches for the different packages in their package directory, package/<pkg>/.
- However, doing this involves changing the Buildroot sources themselves, which may not be appropriate for some highly specific patches.
- The *global patch directories* mechanism allows to specify additional locations where Buildroot will look for patches to apply on packages.
- BR2_GLOBAL_PATCH_DIR specifies a space-separated list of directories containing patches.
- These directories must contain sub-directories named after the packages, themselves containing the patches to be applied.

# Global patch directory example

## Patching *strace*

```
$ ls package/strace/*.patch
0001-linux-aarch64-add-missing-header.patch

$ find ~/patches/
~/patches/
~/patches/strace/
~/patches/strace/0001-Demo-strace-change.patch

$ grep ^BR2_GLOBAL_PATCH_DIR .config
BR2_GLOBAL_PATCH_DIR="$(HOME)/patches"

$ make strace
[...]
>>> strace 4.10 Patching

Applying 0001-linux-aarch64-add-missing-header.patch using patch:
patching file linux/aarch64/arch_regs.h

Applying 0001-Demo-strace-change.patch using patch:
patching file README
[...]
```

# Generating patches

- To generate the patches against a given package source code, there are typically two possibilities.
- Use the upstream version control system, often *Git*
- Use a tool called `quilt`
  - Useful when there is no version control system provided by the upstream project
  - http://savannah.nongnu.org/projects/quilt

# Generating patches: with Git

Needs to be done outside of Buildroot: you cannot use the Buildroot package build directory.

1. Clone the upstream Git repository
   `git clone git://...`
2. Create a branch starting on the tag marking the stable release of the software as packaged in Buildroot
   `git checkout -b buildroot-changes v3.2`
3. Import existing Buildroot patches (if any)
   `git am /path/to/buildroot/package/<foo>/*.patch`
4. Make your changes and commit them
   `` git commit -s -m ``this is a change'' ``
5. Generate the patches
   `git format-patch v3.2`

# Generating patches: with Quilt

1. Extract the package source code:
   `tar xf /path/to/dl/<foo>-<version>.tar.gz`
2. Inside the package source code, reate a directory for patches
   `mkdir patches`
3. Import existing Buildroot patches
   `quilt import /path/to/buildroot/package/<foo>/*.patch`
4. Apply existing Buildroot patches
   `quilt push -a`
5. Create a new patch
   `quilt new 0001-fix-header-inclusion.patch`
6. Edit a file
   `quilt edit main.c`
7. Refresh the patch
   `quilt refresh`

# User, permission and device tables

# Package-specific users

- The default skeleton in `system/skeleton/` has a number of default users/groups.
- Packages can define their own custom users/groups using the `<pkg>_USERS` variable:

```
define <pkg>_USERS
	username uid group gid password home shell groups comment
endef
```

- Examples:

```
define AVAHI_USERS
	avahi -1 avahi -1 * - - -
endef
```

```
define MYSQL_USERS
	mysql -1 nogroup -1 * /var/mysql - - MySQL daemon
endef
```

# File permissions and ownership

- By default, before creating the root filesystem images, Buildroot changes the ownership of all files to `0:0`, i.e. `root:root`
- Permissions are preserved as is, but since the build is executed as non-root, it is not possible to install setuid applications.
- A default set of permissions for certain files or directories is defined in `system/device_table.txt`.
- The `<pkg>_PERMISSIONS` variable allows packages to define special ownership and permissions for files and directories:

```
define <pkg>_PERMISSIONS
name type mode uid gid major minor start inc count
endef
```

- The `major`, `minor`, `start`, `inc` and `count` fields are not used.

# File permissions and ownership: examples

- sudo needs to be installed *setuid root*:

```
define SUDO_PERMISSIONS
        /usr/bin/sudo f 4755 0 0 - - - - -
endef
```

- /var/lib/nginx needs to be owned by www-data, which has UID/GID 33 defined in the skeleton:

```
define NGINX_PERMISSIONS
        /var/lib/nginx d 755 33 33 - - - - -
endef
```

# Devices

- Defining devices only applies when the chosen /dev management strategy is *Static using a device table*. In other cases, *device files* are created dynamically.
- A default set of *device files* is described in system/device_table_dev.txt and created by Buildroot in the root filesystem images.
- When packages need some additional custom devices, they can use the <pkg>_DEVICES variable:

```
define <pkg>_DEVICES
name type mode uid gid major minor start inc count
endef
```

- Becoming less useful, since most people are using a dynamic /dev nowadays.

# Devices: example

xenomai.mk

```
define XENOMAI_DEVICES
/dev/rtheap  c  666  0  0  10  254  0  0  -
/dev/rtscope c  666  0  0  10  253  0  0  -
/dev/rtp     c  666  0  0  150 0    0  1  32
endef
```

# Init scripts and systemd unit files

# Init scripts, systemd unit files

- Buildroot supports several main init systems: *sysvinit*, *Busybox* and *systemd*
- When packages want to install a program to be started at boot time, they need to install either a startup script (*sysvinit*/*Busybox*) or a *systemd service* file.
- They can do so with the `<pkg>_INSTALL_INIT_SYSV` and `<pkg>_INSTALL_INIT_SYSTEMD` variables, which contain a list of shell commands.
- Buildroot will execute either the `<pkg>_INSTALL_INIT_SYSV` or the `<pkg>_INSTALL_INIT_SYSTEMD` commands of all enabled packages depending on the selected init system.

# Init scripts, systemd unit files: example

bind.mk

```
define BIND_INSTALL_INIT_SYSV
  $(INSTALL) -m 0755 -D package/bind/S81named \
     $(TARGET_DIR)/etc/init.d/S81named
endef

define BIND_INSTALL_INIT_SYSTEMD
  $(INSTALL) -D -m 644 package/bind/named.service \
     $(TARGET_DIR)/usr/lib/systemd/system/named.service

  mkdir -p $(TARGET_DIR)/etc/systemd/system/multi-user.target.wants

  ln -sf /usr/lib/systemd/system/named.service \
    $(TARGET_DIR)/etc/systemd/system/[...]/named.service
endef
```

# Config scripts

# Config scripts: introduction

- Libraries not using `pkg-config` often install a **small shell script** that allows applications to query the compiler and linker flags to use the library.
- Examples: `curl-config`, `freetype-config`, etc.
- Such scripts will:
  - generally return results that are **not appropriate for cross-compilation**
  - be used by other cross-compiled Buildroot packages that use those libraries
- By listing such scripts in the `<pkg>_CONFIG_SCRIPTS` variable, Buildroot will **adapt the prefix, header and library paths** to make them suitable for cross-compilation.
- Paths in `<pkg>_CONFIG_SCRIPTS` are relative to `$(STAGING_DIR)/usr/bin`.

# Config scripts: examples

libpng.mk

```
LIBPNG_CONFIG_SCRIPTS = \
	libpng$(LIBPNG_SERIES)-config libpng-config
```

imagemagick.mk

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

ifeq ($(BR2_INSTALL_LIBSTDCPP)$(BR2_USE_WCHAR),yy)
IMAGEMAGICK_CONFIG_SCRIPTS += Magick++-config
endif
```

# Config scripts: effect

Without <pkg>_CONFIG_SCRIPTS

```
$ ./output/staging/usr/bin/libpng-config --cflags --ldflags
-I/usr/include/libpng16
-L/usr/lib -lpng16
```

With <pkg>_CONFIG_SCRIPTS

```
$ ./output/staging/usr/bin/libpng-config --cflags --ldflags
-I.../buildroot/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/include/libpng16
-L.../buildroot/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot/usr/lib -lpng16
```

# Hooks

# Hooks: principle (1)

- Buildroot *package infrastructure* often implement a default behavior for certain steps:
  - `generic-package` implements for all packages the download, extract and patch steps
  - Other infrastructures such as `autotools-package` or `cmake-package` also implement the configure, build and installations steps
- In some situations, the package may want to do **additional actions** before or after one these steps.
- The **hook** mechanism allows packages to add such custom actions.

# Hooks: principle (2)

- There are **pre** and **post** hooks available for all steps of the package compilation process:
  - download, extract, rsync, patch, configure, build, install, install staging, install target, install images, legal info
  - `<pkg>_(PRE|POST)_<step>_HOOKS`
  - Example: `CMAKE_POST_INSTALL_TARGET_HOOKS`, `CVS_POST_PATCH_HOOKS`, `BINUTILS_PRE_PATCH_HOOKS`
- Hook variables contain a list of make macros to call at the appropriate time.
  - Use `+=` to register an additional hook to a hook point
- Those make macros contain a list of commands to execute.

# Hooks: examples

libungif.mk: remove unneeded binaries

```
define LIBUNGIF_BINS_CLEANUP
        rm -f $(addprefix $(TARGET_DIR)/usr/bin/,$(LIBUNGIF_BINS))
endef

LIBUNGIF_POST_INSTALL_TARGET_HOOKS += LIBUNGIF_BINS_CLEANUP
```

vsftpd.mk: adjust configuration

```
define VSFTPD_ENABLE_SSL
        $(SED) 's/.*VSF_BUILD_SSL/#define VSF_BUILD_SSL/' \
                $(@D)/builddefs.h
endef

ifeq ($(BR2_PACKAGE_OPENSSL),y)
VSFTPD_DEPENDENCIES += openssl
VSFTPD_LIBS += -lssl -lcrypto
VSFTPD_POST_CONFIGURE_HOOKS += VSFTPD_ENABLE_SSL
endif
```

# Overriding commands

# Overriding commands: principle

- In other situations, a package may want to completely **override** the default implementation of a step provided by a package infrastructure.
- A package infrastructure will in fact only implement a given step **if not already defined by a package**.
- So defining `<pkg>_EXTRACT_CMDS` or `<pkg>_BUILDS_CMDS` in your package `.mk` file will override the package infrastructure implementation (if any).

# Overriding commands: examples

jquery: source code is only one file

```
JQUERY_SITE = http://code.jquery.com
JQUERY_SOURCE = jquery-$(JQUERY_VERSION).min.js

define JQUERY_EXTRACT_CMDS
	cp $(DL_DIR)/$(JQUERY_SOURCE) $(@D)
endef
```

tftpd: install only what's needed

```
define TFTPD_INSTALL_TARGET_CMDS
	$(INSTALL) -D $(@D)/tftp/tftp $(TARGET_DIR)/usr/bin/tftp
	$(INSTALL) -D $(@D)/tftpd/tftpd $(TARGET_DIR)/usr/sbin/tftpd
endef

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

# Legacy handling

# Legacy handling: `Config.in.legacy`

- When a `Config.in` option is removed, the corresponding value in the `.config` is silently removed.
- Due to this, when users upgrade Buildroot, they generally don't know that an option they were using has been removed.
- Buildroot therefore adds the removed config option to `Config.in.legacy` with a description of what has happened.
- If any of these legacy options is enabled then Buildroot refuses to build.

# Virtual packages

# Virtual packages

- There are situations where different packages provide an implementation of the same interface
- The most useful example is OpenGL
  - OpenGL is an API
  - Each HW vendor typically provides its own OpenGL implementation, each packaged as separate Buildroot packages
- Packages using the OpenGL interface do not want to know which implementation they are using: they are simply using the OpenGL API
- The mechanism of *virtual packages* in Buildroot allows to solve this situation.
  - `libgles` is a virtual package offering the OpenGL ES API
  - Eight packages are *providers* of the OpenGL ES API: `gpu-amd-bin-mx51`, `gpu-viv-bin-mx6q`, `mesa3d`, `nvidia-driver`, `nvidia-tegra23-binaries`, `rpi-userland`, `sunxi-mali`, `ti-gfx`

# Virtual package definition: Config.in

libgles/Config.in

```
config BR2_PACKAGE_HAS_LIBGLES
        bool

config BR2_PACKAGE_PROVIDES_LIBGLES
        depends on BR2_PACKAGE_HAS_LIBGLES
        string
```

- `BR2_PACKAGE_HAS_LIBGLES` is a hidden boolean
  - Packages needing OpenGL ES will `depends on` it.
  - Packages providing OpenGL ES will `select` it.
- `BR2_PACKAGE_PROVIDES_LIBGLES` is a hidden string
  - Packages providing OpenGL ES will define their name as the variable value
  - The `libgles` package will have a build dependency on this provider package.

# Virtual package definition: .mk

libgles/libgles.mk

```
$(eval $(virtual-package))
```

- Nothing to do: the `virtual-package` infrastructure takes care of everything, using the `BR2_PACKAGE_HAS_<name>` and `BR2_PACKAGE_PROVIDES_<name>` options.

# Virtual package provider

sunxi-mali/Config.in

```
config BR2_PACKAGE_SUNXI_MALI
        bool "sunxi-mali"
        select BR2_PACKAGE_HAS_LIBEGL
        select BR2_PACKAGE_HAS_LIBGLES

config BR2_PACKAGE_PROVIDES_LIBGLES
        default "sunxi-mali"
```

sunxi-mali/sunxi-mali.mk

```
[...]
SUNXI_MALI_PROVIDES = libegl libgles
[...]
```

- The variable <pkg>_PROVIDES is only used to detect if two providers for the same virtual package are enabled.

# Virtual package user

qt5/qt5base/Config.in

```
config BR2_PACKAGE_QT5BASE_OPENGL_ES2
        bool "OpenGL ES 2.0+"
        depends on BR2_PACKAGE_HAS_LIBGLES
        help
          Use OpenGL ES 2.0 and later versions.
```

qt5/qt5base/qt5base.mk

```
ifeq ($(BR2_PACKAGE_QT5BASE_OPENGL_DESKTOP),y)
QT5BASE_CONFIGURE_OPTS += -opengl desktop
QT5BASE_DEPENDENCIES   += libgl
else ifeq ($(BR2_PACKAGE_QT5BASE_OPENGL_ES2),y)
QT5BASE_CONFIGURE_OPTS += -opengl es2
QT5BASE_DEPENDENCIES   += libgles
else
QT5BASE_CONFIGURE_OPTS += -no-opengl
endif
```

# Practical lab - Advanced packages

- Package an application with a mandatory dependency and an optional dependency
- Package a library, hosted on GitHub
- Use *hooks* to tweak packages
- Add a patch to a package

# Analyzing the build

***free electrons***

Embedded Linux
Experts

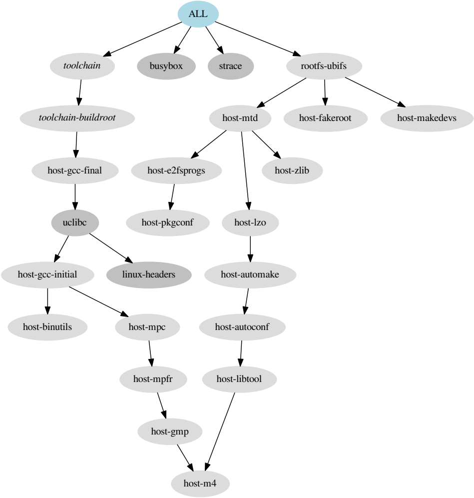
# Analyzing the build: available tools

- Buildroot provides several useful tools to analyze the build:
  - The **licensing report**, covered in a previous section, which allows to analyze the list of packages and their licenses.
  - The **dependency graphing** tools
  - The **build time graphing** tools
- A tool to analyze the contribution of each package to the filesystem size is under development, it should be merged in Buildroot 2015.08. Patches are already available.
- Additional tools can be constructed using **instrumentation scripts**

# Dependency graphing

- Exploring the dependencies between packages is useful to understand
  - why a particular package is being brought into the build
  - if the build size and duration can be reduced
- `make graph-depends` to generate a full dependency graph, which can be huge!
- `make <pkg>-graph-depends` to generate the dependency graph of a given package
- The graph is done according to the current Buildroot configuration.
- Resulting graphs in `$(O)/graphs/`

# Dependency graph example



# Dependency graphing: advanced

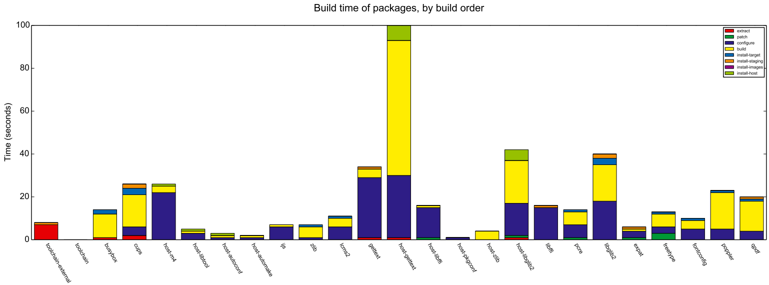
- Variable BR2_GRAPH_OUT, to select the output format. Defaults to pdf, can be png or svg for example.
- Internally, the graph is generated by the Python script support/scripts/graph-depends
- All options that this script supports can be passed using the BR2_GRAPH_DEPS_OPTS variable when calling make graph-depends
- Example
  - Generate a PNG graph of the openssh package dependencies
  - Custom colors
  - Stop graphing on the host-automake package, to remove a part of the graph we're not interested in

```
BR2_GRAPH_OUT=png \
    BR2_GRAPH_DEPS_OPTS="--colours red,blue,green --stop-on=host-automake" \
    make openssh-graph-depends
```

# Build time graphing

- When the generated embedded Linux system grows bigger and bigger, the build time also increases.
- It is sometimes useful to analyze this build time, and see if certain packages are particularly problematic.
- Buildroot collects build duration data in the file `$(O)/build/build-time.log`
- `make graph-build` generates several graphs in `$(O)/graphs/`:
  - `build.hist-build.pdf`, build time in build order
  - `build.hist-duration.pdf`, build time by duration
  - `build.hist-name.pdf`, build time by package name
  - `build.pie-packages.pdf`, pie chart of the per-package build time
  - `build.pie-steps.pdf`, pie chart of the per-step build time
- Note: only works properly after a complete clean rebuild.

# Build time graphing: example



# Instrumentation scripts

- Additional analysis tools can be constructed using the **instrumentation scripts** mechanism.
- `BR2_INSTRUMENTATION_SCRIPTS` is an environment variable, containing a space-separated list of scripts, that will be called before and after each step of the build of all packages.
- Three arguments are passed to the scripts:
  1. `start` or `stop` to indicate whether it's the beginning or end of the step
  2. the name of the step
  3. the name of the package

# Instrumentation scripts: example

### instrumentation.sh

```
#!/bin/sh
echo "${3} now ${1}s ${2}"
```

### Output

```
$ make BR2_INSTRUMENTATION_SCRIPTS="./instrumentation.sh"
strace now starts extract
>>> strace 4.10 Extracting
xzcat /home/thomas/dl/strace-4.10.tar.xz | tar --strip-components=1 \
      -C /home/thomas/projets/buildroot/output/build/strace-4.10  -xf -
strace now ends extract
strace now starts patch
>>> strace 4.10 Patching

Applying 0001-linux-aarch64-add-missing-header.patch using patch:
patching file linux/aarch64/arch_regs.h
>>> strace 4.10 Updating config.sub and config.guess
for file in config.guess config.sub; do for i in $(find \
    /home/thomas/projets/buildroot/output/build/strace-4.10 -name $file); do \
       cp support/gnuconfig/$file $i; done; done
>>> strace 4.10 Patching libtool
strace now ends patch
strace now starts configure
>>> strace 4.10 Configuring
```

# Advanced topics

***free electrons***

Embedded Linux Experts


Corrections, suggestions, contributions and translations are welcome!

# BR2_EXTERNAL: principle

- Storing your custom packages, custom configuration files and custom *defconfigs* inside the Buildroot tree may not be the most practical solution
  - Doesn't cleanly separate open-source parts from proprietary parts
  - Makes it harder to upgrade Buildroot
- The `BR2_EXTERNAL` mechanism allows to store your own package recipes, *defconfigs* and other artefacts **outside** of the Buildroot source tree.
- Note: can only be used to add new packages, not to override existing Buildroot packages

# BR2_EXTERNAL: example organization

- `project/`
  - `buildroot/`
    - The Buildroot source code, cloned from Git, or extracted from a release tarball.
  - `external/`
    - Your *external tree*, with your own custom packages and *defconfigs*
  - `output-build1/`
  - `output-build2/`
    - Several *output* directories, to build various configurations
  - `custom-app/`
  - `custom-lib/`
    - The source code of your custom applications and libraries.

# BR2_EXTERNAL: mechanism

- Specify `BR2_EXTERNAL` on the command line when building.
- Buildroot will:
  - include `$(BR2_EXTERNAL)/Config.in` in the configuration menu, under a new menu called `User-provided options`
  - include `$(BR2_EXTERNAL)/external.mk` in the make logic
  - include `$(BR2_EXTERNAL)/configs/` in the list of *defconfigs*

# BR2_EXTERNAL: recommended structure

```
+-- board/
|   +-- <company>/
|       +-- <boardname>/
|           +-- linux.config
|           +-- busybox.config
|           +-- <other configuration files>
|           +-- post_build.sh
|           +-- post_image.sh
|           +-- rootfs_overlay/
|           |   +-- etc/
|           |   +-- <some file>
|           +-- patches/
|               +-- foo/
|               |   +-- <some patch>
|               +-- libbar/
|                   +-- <some other patches>
|
+-- configs/
|   +-- <boardname>_defconfig
|
+-- package/
|   +-- <company>/
|       +-- package1/
|       |    +-- Config.in
|       |    +-- package1.mk
|       +-- package2/
|           +-- Config.in
|           +-- package2.mk
|
+-- Config.in
+-- external.mk
```

# BR2_EXTERNAL/Config.in

- Custom configuration options
- Configuration options for the BR2_EXTERNAL packages
- The $BR2_EXTERNAL variable is available

Example $(BR2_EXTERNAL)/Config.in

```
source "$BR2_EXTERNAL/package/package1/Config.in"
source "$BR2_EXTERNAL/package/package2/Config.in"
```

# BR2_EXTERNAL/external.mk

- Can include custom *make* logic
- Generally only used to include the package .mk files

Example $(BR2_EXTERNAL)/external.mk

```
include $(sort $(wildcard $(BR2_EXTERNAL)/package/*/*.mk))
```

# Using BR2_EXTERNAL

- Not a configuration option, only an **environment variable** to be passed on the command line

```
make BR2_EXTERNAL=/path/to/external
```

- **Automatically saved** in the hidden `.br-external` file in the output directory
  - no need to pass `BR2_EXTERNAL` at every make invocation
  - can be changed at any time by passing a new value, and removed by passing an empty value
- Can be either an **absolute** or a **relative** path, but if relative, important to remember that it's relative to the Buildroot source directory

# Use BR2_EXTERNAL in your configuration

- In your Buildroot configuration, don't use absolute paths for the *rootfs overlay*, the *post-build scripts*, *global patch directories*, etc.
- If they are located in your BR2_EXTERNAL, you can use $(BR2_EXTERNAL) in your Buildroot configuration options.
- With the recommended structure shown before, a Buildroot configuration would look like:

```
BR2_GLOBAL_PATCH_DIR="$(BR2_EXTERNAL)/board/<company>/<boardname>/patches/"
...
BR2_ROOTFS_OVERLAY="$(BR2_EXTERNAL)/board/<company>/<boardname>/rootfs_overlay/"
...
BR2_ROOTFS_POST_BUILD_SCRIPT="$(BR2_EXTERNAL)/board/<company>/<boardname>/post_build.sh"
BR2_ROOTFS_POST_IMAGE_SCRIPT="$(BR2_EXTERNAL)/board/<company>/<boardname>/post_image.sh"
...
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="$(BR2_EXTERNAL)/board/<company>/<boardname>/linux.config"
```

# Package-specific targets: basics

- Internally, each package is implemented through a number of package-specific *make targets*
  - They can sometimes be useful to call directly, in certain situations.
- The targets used in the normal build flow of a package are:
  - `<pkg>`, fully build and install the package
  - `<pkg>-source`, just download the source code
  - `<pkg>-extract`, download and extract
  - `<pkg>-patch`, download, extract and patch
  - `<pkg>-configure`, download, extract, patch and configure
  - `<pkg>-build`, download, extract, patch, configure and build
  - `<pkg>-install-staging`, download, extract, patch, configure and do the staging installation (target packages only)
  - `<pkg>-install-target`, download, extract, patch, configure and do the target installation (target packages only)
  - `<pkg>-install`, download, extract, patch, configure and install

# Package-specific targets: example (1)

```
$ make strace
>>> strace 4.10 Extracting
>>> strace 4.10 Patching
>>> strace 4.10 Updating config.sub and config.guess
>>> strace 4.10 Patching libtool
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ make strace-build
... nothing ...
$ make ltrace-patch
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Extracting
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Patching
$ make ltrace
>>> argp-standalone 1.3 Extracting
>>> argp-standalone 1.3 Patching
>>> argp-standalone 1.3 Updating config.sub and config.guess
>>> argp-standalone 1.3 Patching libtool
[...]
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Configuring
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Autoreconfiguring
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Patching libtool
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Building
>>> ltrace 0896ce554f80afdcba81d9754f6104f863dea803 Installing to target
```

# Package-specific targets: advanced

- Additional useful targets
  - `make <pkg>-show-depends`, show the package dependencies
  - `make <pkg>-graph-depends`, generates a dependency graph
  - `make <pkg>-dirclean`, completely remove the package source code directory. The next `make` invocation will fully rebuild this package.
  - `make <pkg>-reinstall`, force to re-execute the installation step of the package
  - `make <pkg>-rebuild`, force to re-execute the build and installation steps of the package
  - `make <pkg>-reconfigure`, force to re-execute the configure, build and installation steps of the package.

# Package-specific targets: example (2)

```
$ make strace
>>> strace 4.10 Extracting
>>> strace 4.10 Patching
>>> strace 4.10 Updating config.sub and config.guess
>>> strace 4.10 Patching libtool
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ ls output/build/
strace-4.10 [...]
$ make strace-dirclean
rm -Rf /home/thomas/projets/buildroot/output/build/strace-4.10
$ ls output/build/
[... no strace-4.10 directory ...]
```

# Package-specific targets: example (3)

```
$ make strace
>>> strace 4.10 Extracting
>>> strace 4.10 Patching
>>> strace 4.10 Updating config.sub and config.guess
>>> strace 4.10 Patching libtool
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ make strace-rebuild
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
$ make strace-reconfigure
>>> strace 4.10 Configuring
>>> strace 4.10 Building
>>> strace 4.10 Installing to target
```

# Understanding rebuilds (1)

- Doing a **full rebuild** is achieved using:

```
$ make clean all
```

  - It will completely remove all build artefacts and restart the build from scratch
- Buildroot **does not try to be smart**
  - once the system has been built, if a configuration change is made, the next `make` will **not apply all the changes** made to the configuration.
  - being smart is very, very complicated if you want to do it in a reliable way.

# Understanding rebuilds (2)

- When a package has been built by Buildroot, Buildroot keeps a **hidden file** telling that the package has been built.
  - Buildroot will therefore *never* rebuild that package, unless a **full rebuild is done**, or this specific package is **explicitly rebuilt**.
  - Buildroot does not *recurse* into each package at each `make` invocation, it would be too time-consuming. So if you change one source file in a package, Buildroot does not know it.
- When `make` is invoked, Buildroot **will always**:
  - Build the packages that have not been built in a previous build and install them to the target
  - Cleanup the target root filesystem from useless files
  - Run *post-build* scripts, copy *rootfs overlays*
  - Generate the root filesystem images
  - Run *post-image* scripts

# Understanding rebuilds: scenarios (1)

- If you enable a new package in the configuration, and run `make`
  - Buildroot will build it and install it
  - However, other packages that may benefit from this package will not be rebuilt automatically
- If you remove a package from the configuration, and run `make`
  - Nothing happens. The files installed by this package are not removed from the target filesystem.
  - Buildroot does not track which files are installed by which package
  - Need to do a full rebuild to get the new result. Advice: do it only when really needed.
- If you change the sub-options of a package that has already been built, and run `make`
  - Nothing happens.
  - You can force Buildroot to rebuild this package using `make <pkg>-reconfigure` or `make <pkg>-rebuild`.

# Understanding rebuilds: scenarios (2)

- If you make a change to a *post-build* script, a *rootfs overlay* or a *post-image* script, and run `make`
  - This is sufficient, since these parts are re-executed at every `make` invocation.
- If you change a fundamental system configuration option: architecture, type of toolchain or toolchain configuration, init system, etc.
  - You **must do a full rebuild**
- If you change some source code in `output/build/<foo>-<version>/` and issue `make`
  - The package will not be rebuilt automatically: Buildroot has a *hidden file* saying that the package was already built.
  - Use `make <pkg>-reconfigure` or `make <pkg>-rebuild`
  - And remember that doing changes in `output/build/<foo>-<version>/` can only be temporary: this directory is removed during a `make clean`.

# Tips for building faster

- Build time is often an issue, so here are some tips to help
  - Use fast hardware: lots of RAM, and SSD
  - Do not use virtual machines
  - You can enable the `ccache` *compiler cache* using `BR2_CCACHE`
  - Use external toolchains instead of internal toolchains
  - Learn about rebuilding only the few packages you actually care about
  - Build everything locally, do not use NFS for building
  - Remember that you can do several independent builds in parallel in different output directories

- Use legal-info for legal information extraction
- Use graph-depends for dependency graphing
- Use graph-build for build time graphing
- Use BR2_EXTERNAL to isolate the project-specific changes (packages, configs, etc.)

# Application development

***free electrons***

Embedded Linux Experts


Corrections, suggestions, contributions and translations are welcome!

# Building code for Buildroot

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

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

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

Note: too long lines have been splitted.

# Building code for Buildroot: autotools

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

```
$ export PATH=.../buildroot/output/host/usr/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.

# The <pkg>_OVERRIDE_SRCDIR mechanism

- Very often, you don't build packages manually: Buildroot builds them for you.
- But Buildroot also downloads them for you, and keeps the source code in the package build directory.
- Not very practical during development:
  - The build directory is temporary, gets removed when doing a `make clean` or `make <pkg>-dirclean`
  - The build directory isn't checked out from your version control system.
- Buildroot should, for certain packages, pick up the source from a local directory.
- This is exactly what `<pkg>_OVERRIDE_SRCDIR` allows to do.

# Without <pkg>_OVERRIDE_SRCDIR

- The normal package build process, when `<pkg>_OVERRIDE_SRCDIR` is not used, is:
  1. Package gets downloaded as a tarball, or from a VCS repository (in which case a tarball is generated)
  2. The tarball is extracted in `$(O)/build/<pkg>-<version>`
  3. Then the configure, build and installs steps are executed
- `$(O)/build/<pkg>-<version>/` does not contain any VCS metadata, and is a temporary directory.
- Running `make <pkg>-reconfigure`, `make <pkg>-rebuild`, `make <pkg>-reinstall` only restarts the build process from one of the corresponding steps.

# Effect of <pkg>_OVERRIDE_SRCDIR

- For each package, you can define a `<pkg>_OVERRIDE_SRCDIR` variable that points to a local directory containing the source code for this package.
- Instead of downloading and extracting the original source, Buildroot will *rsync* the source from the specified directory to the build directory.
- Invoking `make <pkg>-reconfigure`, `make <pkg>-rebuild`, `make <pkg>-reinstall` will retrigger a *rsync*.

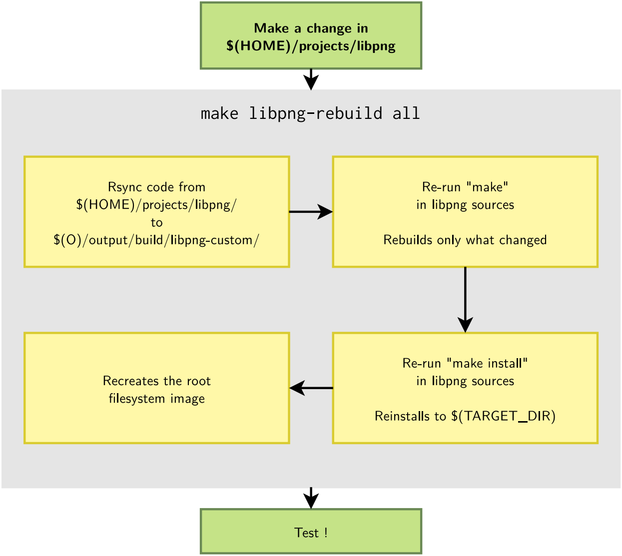
# Passing <pkg>_OVERRIDE_SRCDIR

- <pkg>_OVERRIDE_SRCDIR can be specified:
  - In the package .mk file itself. Not ideal solution, and identical to <pkg>_SITE_METHOD = local
  - 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
```

# <pkg>_OVERRIDE_SRCDIR workflow



# 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_none` and `BR2_STRIP_strip` options allow to disable or enable stripping of binaries on the target.

# 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_none=y`
  - 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/usr/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 remote-extended <ip>:2345`
  - 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:
  - Customize the `BR2_HOST_DIR` option to a path like `/opt/project-sdk/`.
  - Do a full build from scratch. Due to the value of `BR2_HOST_DIR`, the cross-compiler and the sysroot with all its libraries will be installed in `/opt/project-sdk/` instead of the normal `$(O)/host`.
  - Tarball the `/opt/project-sdk/` and share it with the developers.
- Warnings:
  - The SDK is not relocatable: it must remain in `/opt/project-sdk/`
  - The SDK must remain in sync with the root filesystem running on the target, otherwise applications built with the SDK may not run properly.

# Eclipse plug-in

- For application developers interested in using the Eclipse IDE, a Buildroot-specific plugin has been developed.
- It integrates the toolchain(s) generated by Buildroot into the Eclipse C/C++ Development Environment.
- Allows Eclipse projects to easily use the compiler, linker and debugger provided by Buildroot
- In Buildroot, enable the `BR2_ECLIPSE_REGISTER` option.
- In Eclipse, install the *Buildroot* plugin, and follow the instructions available from the plugin website.
- See `https://github.com/mbats/eclipse-buildroot-bundle/wiki` for download, installation and usage details.

- Build and run your own application
- Remote debug your application
- Use <pkg>_OVERRIDE_SRCDIR
- Set up Eclipse for Buildroot application development

# Understanding Buildroot internals

***free electrons***

Embedded Linux Experts


Corrections, suggestions, contributions and translations are welcome!

# 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

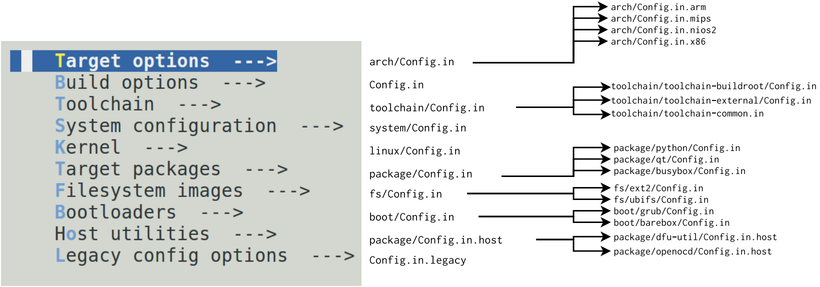
```
CONFIG_CONFIG_IN = Config.in
CONFIG = support/kconfig
BR2_CONFIG = $(CONFIG_DIR)/.config

-include $(BR2_CONFIG)

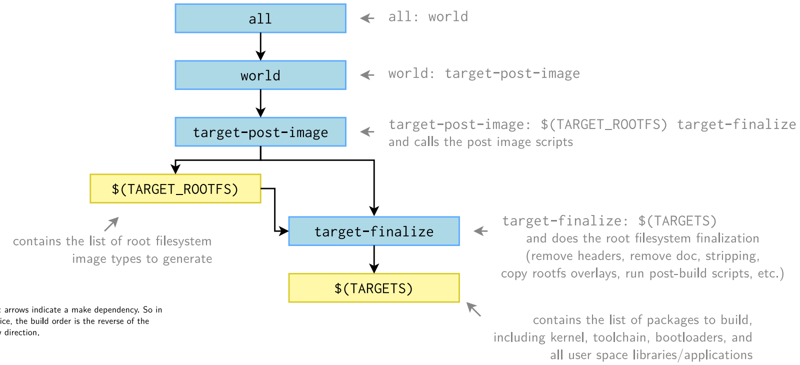
$(BUILD_DIR)/buildroot-config/%onf:
        mkdir -p $(@D)/lxdialog
        $(MAKE) ... -C $(CONFIG) -f Makefile.br $(@F)

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

# Configuration hierarchy



# 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 $(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
- Note: as of Buildroot 2015.05, `TARGETS` has been renamed to `PACKAGES`

# 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)

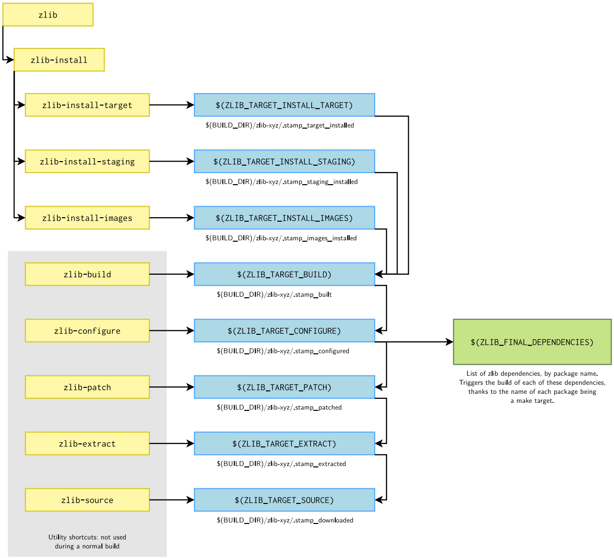
$(1)-patch:             $$($(2)_TARGET_PATCH)
$$($(2)_TARGET_PATCH):  $$($(2)_TARGET_EXTRACT)

$(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 packages have three targets in their dependencies:
  - `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 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* and *kconfig* packages

# 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

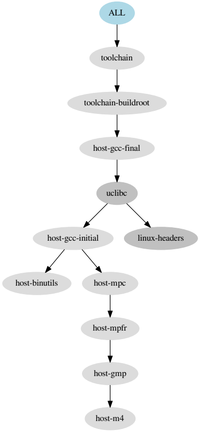
ifeq ($$($(2)_AUTORECONF),YES)
...
$(2)_PRE_CONFIGURE_HOOKS += AUTORECONF_HOOK
$(2)_DEPENDENCIES += host-automake host-autoconf host-libtool
endif
```

# 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
...

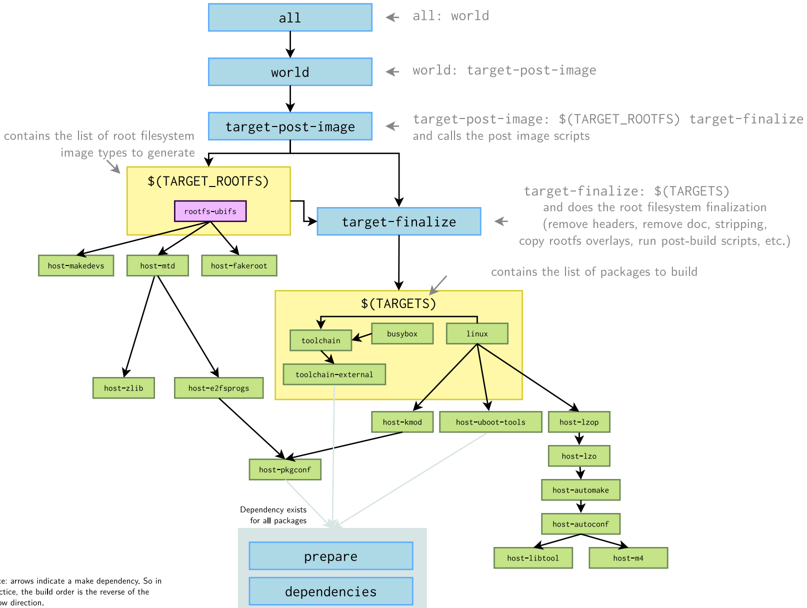
UBIFS_OPTS += $(call qstrip,$(BR2_TARGET_ROOTFS_UBIFS_OPTS))

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




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

# Buildroot community: support and contribution

***free electrons***

Embedded Linux
Experts

# Documentation

- Buildroot comes with its own documentation
- Pre-built versions available at `http://buildroot.org/docs.html` (PDF, HTML, text)
- Source code of the manual located in `docs/manual` in the Buildroot sources
  - Written in *Asciidoc* format
- The manual can be built with:
  - `make manual`
  - or just `make manual-html`, `make manual-pdf`, `make manual-epub`, `make manual-text`, `make manual-split-html`
  - A number of tools need to be installed on your machine, see the manual itself.

# Getting support

- Free support
  - The *mailing list* for e-mail discussion
    http://lists.busybox.net/mailman/listinfo/buildroot
    1300+ subscribers, quite heavy traffic.
  - The IRC channel, #buildroot on the Freenode network, for interactive discussion
    100+ people, most available during European daylight hours
  - Bug tracker
    https://bugs.busybox.net/buglist.cgi?product=buildroot
- Commercial support
  - A number of embedded Linux services companies, including Free Electrons, can provide commercial services around Buildroot.

# Tips to get free support

- If you have a build issue to report:
  - Make sure to reproduce after a `make clean all` cycle
  - Include the Buildroot version, Buildroot `.config` that reproduces the issue, and last 100-200 lines of the build output in your report.
  - Use *pastebin* sites like `http://code.bulix.org` when reporting issues over IRC.
- The community will be much more likely to help you if you use a recent Buildroot version.

# Release schedule

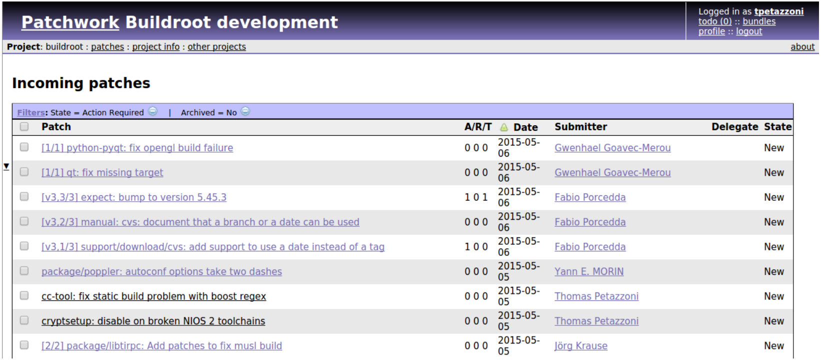
- The Buildroot community publishes stable releases every three months.
- YYYY.02, YYYY.05, YYYY.08 and YYYY.11 every year.
- The three months cycle is split in two periods
  - Two first months of active development
  - One month of stabilization before the release
- At the beginning of the stabilization phase, `-rc1` is released.
- Several `-rc` versions are published during this stabilization phase, until the final release.
- Development not completely stopped during the stabilization, a `next` branch is opened.

# Contribution process

- Contributions are made in the form of patches
- Created with `git` and sent by e-mail to the mailing list
  - Use `git send-email` to avoid issues
- The patches are reviewed, tested and discussed by the community
  - You may be requested to modify your patches, and submit updated versions
- Once ready, they are applied by the project maintainer Peter Korsgaard, or the interim maintainer Thomas Petazzoni.
- Some contributions may be rejected if they do not fall within the Buildroot principles/ideas, as discussed by the community.

# Patchwork

- Tool that records all patches sent on the mailing list
- Allows the community to see which patches need review/testing, and the maintainers which patches can be applied.
- Everyone can create an account to manage his own patches
- `http://patchwork.buildroot.org/`



Patchwork Buildroot development

Logged in as tpetazzoni
todo (0) :: bundles
profile :: logout

Project: buildroot : patches : project info : other projects — about

## Incoming patches

Filters: State = Action Required | Archived = No

| Patch | A/R/T | Date | Submitter | Delegate | State |
|---|---|---|---|---|---|
| [1/1] python-pyqt: fix opengl build failure | 0 0 0 | 2015-05-06 | Gwenhael Goavec-Merou | | New |
| [1/1] qt: fix missing target | 0 0 0 | 2015-05-06 | Gwenhael Goavec-Merou | | New |
| [v3,3/3] expect: bump to version 5.45.3 | 1 0 1 | 2015-05-06 | Fabio Porcedda | | New |
| [v3,2/3] manual: cvs: document that a branch or a date can be used | 0 0 0 | 2015-05-06 | Fabio Porcedda | | New |
| [v3,1/3] support/download/cvs: add support to use a date instead of a tag | 1 0 0 | 2015-05-06 | Fabio Porcedda | | New |
| package/poppler: autoconf options take two dashes | 0 0 0 | 2015-05-05 | Yann E. MORIN | | New |
| cc-tool: fix static build problem with boost regex | 0 0 0 | 2015-05-05 | Thomas Petazzoni | | New |
| cryptsetup: disable on broken NIOS 2 toolchains | 0 0 0 | 2015-05-05 | Thomas Petazzoni | | New |
| [2/2] package/libtirpc: Add patches to fix musl build | 0 0 0 | 2015-05-05 | Jörg Krause | | New |

# Automated build testing

- The enormous number of configuration options in Buildroot make it very difficult to test all combinations.
- Random configurations are therefore built 24/7 by multiple machines.
  - Random choice of architecture/toolchain combination from a pre-defined list
  - Random selection of packages using `make randpackageconfig`
  - Random enabling of features like static library only, or `BR2_ENABLE_DEBUG=y`
- Scripts and tools publicly available at http://git.buildroot.net/buildroot-test/
- Results visible at http://autobuild.buildroot.org/
- Daily e-mails with the build results of the past day

# autobuild.buildroot.org

## Buildroot tests

| Date | Status | Commit ID | Submitter | Arch | Failure reason | Data |
|---|---|---|---|---|---|---|
| 2015-05-06 18:12:18 | OK | 1f55934c | Peter Korsgaard *(gcc10)* | arm | none | dir, end log, config, defconfig |
| 2015-05-06 18:11:08 | OK | 1f55934c | Thomas Petazzoni *(gcc75)* | arm | none | dir, end log, config, defconfig |
| 2015-05-06 18:04:57 | OK | 1f55934c | Thomas Petazzoni *(gcc75)* | arm | none | dir, end log, config, defconfig |
| 2015-05-06 17:56:09 | OK | 1f55934c | Thomas Petazzoni *(Free Electrons server)* | xtensa | none | dir, end log, config, defconfig |
| 2015-05-06 17:53:08 | OK | 1f55934c | Thomas Petazzoni *(gcc75)* | arc | none | dir, end log, config, defconfig |
| 2015-05-06 17:45:56 | OK | 1f55934c | Peter Korsgaard *(gcc10)* | arm | none | dir, end log, config, defconfig |
| 2015-05-06 17:32:35 | OK | 1f55934c | Thomas Petazzoni *(gcc75)* | arm | none | dir, end log, config, defconfig |
| 2015-05-06 17:25:19 | OK | 1f55934c | Richard Braun *(sceen.net)* | xtensa | none | dir, end log, config, defconfig |
| 2015-05-06 17:18:30 | OK | 1f55934c | Peter Korsgaard *(gcc20)* | arc | none | dir, end log, config, defconfig |
| 2015-05-06 17:16:22 | OK | 1f55934c | Thomas Petazzoni *(Free Electrons server)* | i686 | none | dir, end log, config, defconfig |
| 2015-05-06 17:12:27 | NOK | 1f55934c | Richard Braun *(sceen.net)* | arm | libsigsegv-2.10 | dir, end log, config, defconfig |
| 2015-05-06 17:10:07 | OK | 1f55934c | Peter Korsgaard *(gcc20)* | nios2 | none | dir, end log, config, defconfig |
| 2015-05-06 17:09:27 | OK | 1f55934c | Thomas Petazzoni *(gcc75)* | mips | none | dir, end log, config, defconfig |
| 2015-05-06 16:59:38 | NOK | 1f55934c | Peter Korsgaard *(gcc20)* | arm | snmppp-3.3.4 | dir, end log, config, defconfig |

# Autobuild daily reports

```
From: Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
To: buildroot@uclibc.org
Subject: [Buildroot] [autobuild.buildroot.net] Build results for 2015-05-05
Date: Wed,  6 May 2015 08:30:17 +0200 (CEST)

Build statistics for 2015-05-05
===============================

        success : 301
       failures : 50
       timeouts : 1
          TOTAL : 352

Classification of failures by reason
====================================

freerdp-770c67d340d5f0a7b48... | 6
              postgresql-9.4.1 | 5
            python-pyqt-4.11.3 | 5

Detail of failures
===================

     powerpc | boost-1.57.0 | NOK | http://autobuild.buildroot.net/results/b64fd94a8ccff7fa8...
        bfin | cc-tool-0.26 | NOK | http://autobuild.buildroot.net/results/5f84d5696a52c7541...
      xtensa | cc-tool-0.26 | NOK | http://autobuild.buildroot.net/results/d971db839e84480a5...
```

# What's new in Buildroot?

***free electrons***

Embedded Linux Experts


Corrections, suggestions, contributions and translations are welcome!

# What's new in Buildroot

- The major improvements in each release are summarized in the file named `CHANGES` in the Buildroot source tree
- Always mentions changes that could cause backward compatibility problems
- The following slides summarize the major new features added in each release between 2013.08 and 2015.08.
- All new Buildroot versions come with new packages, and many updates to the existing packages
  - Such package additions and updates are not listed in the following slides.

# In 2013.08 (1)

- **Architectures:** improved support for floating point on ARM and Thumb/Thumb2, support for ARM OABI removed
- **Toolchains:**
  - support added for Sourcery CodeBench ARM and MIPS 2013.05
  - Linaro ARM and Aarch64 toolchains updated
  - support added for the Arago ARMv5 and ARMv7 toolchains
  - gcc 4.8.x version bumped
  - support for installing both FDPIC and FLAT libraries on Blackfin
  - support for uClibc 0.9.31 removed
  - convert the internal toolchain backend to use the package infrastructure
  - support added for eglibc in the internal toolchain backend
  - toolchain components for the ARC architecture updated and gdb for ARC added.
  - support for Blackfin in the internal toolchain fixed

# In 2013.08 (2)

- **Defconfigs:** BeagleBone defconfig updated, new defconfig for CubieBoard, for Olimex mx233 Olinuxino, for Calao Systems TNY-A9G20-LPW
- **Packages:**
  - A number of packages have been fixed to use the `<pkg>_CONFIG_SCRIPTS` mechanism to get their `<pkg>-config` shell script installed and modified properly.
  - Licensing information has been added to a number of packages.
  - Use XZ tarballs for a number of packages
- **Noticeable package changes/additions**
  - The glib2/libgtk2/webkit stack has been updated to recent versions.
  - Support for Gstreamer 1.x has been added.
  - OpenGL support for TI OMAP platforms has been added.
  - OpenGL support for Allwinner platforms has been added.
  - OpenMAX support for RaspberryPi has been added.
- Top level menu names reordered and renamed for clarity

# In 2013.11

- **Architectures:** Nios-II support, MIPS arch handling fixes
- **Toolchains:**
  - glibc support
  - upstream uClibc fixes
  - uClibc 0.9.31 for avr32
  - internal crosstool-ng backend removed
  - external musl toolchain support
  - gcc 4.8.2
  - updated Linaro external toolchains
  - Fortran and objective-C support deprecated
  - mudflap support
- **Bootloaders:** U-Boot: u-boot.imx support, version bumps
- **Linux:** use kmod instead of module-init-tools
- **System:** default to devtmpfs for /dev
- **Infrastructure:** Infrastructure: Make 3.82 fixes, locales generation fixes, CVS download support, post-rsync hooks

# In 2014.02

- Support for external packages/defconfigs: `BR2_EXTERNAL`
- Cleanup of environment variable names for consistency
- **Toolchain**: new Linaro and Sourcery Codebench toolchains. x86: Support for AMD Jaguar cores, SSE4.x, SH: SH2/SH3/SH3EB variants removed, Microblaze: Internal toolchain support
- **Legal infrastructure:** Info is now split between host and target packages, large number of license annotations.
- **Lua**: selection between lua 5.1 / 5.2, luarocks support
- **Python:** package infrastructure, many new packages
- **Defconfigs:** Armadeus APF51 and Zedboard added, apf27, apf28, beaglebone, microblaze, pandaboard, qemu, raspberry pi updated.

# In 2014.05 (1)

- **Architectures:**
  - Support for MIPS o32 ABI on MIPS-64 targets has been removed (too exotic)
  - Support for the ARM A12 variant and Intel corei7
- **Defconfigs:** Minnowboard and Altera SoCkit added, QEMU updates.
- **Bootloaders:** Grub2 and gummiboot support, syslinux support extended.
- Kconfig handling for minimum kernel headers version required for packages. Now packages needing specific kernel header features can specify these requirements in Kconfig.

# In 2014.05 (2)

- **Toolchains:**
  - GCC 4.9. Glibc 2.19.
  - Support for the musl C library for internal and external toolchains.
  - GCC 4.8-R3 support for ARC
  - Internal toolchain support for Aarch64 and Microblaze
  - Toolchain tuple vendor name can now be customized.
  - Updated external Linaro ARM/Aarch64 toolchains.
  - Added external Linaro ARMEB toolchain.
  - A GDB gdbinit file is now generated for external toolchains to automatically set the correct sysroot.
- **Infrastructure:**
  - Support for (but disabled as it leads to unreproducible builds) toplevel parallel builds.
  - Python package infrastructure extended to support Python 3.x
  - Perl and virtual package infrastructure support added.
  - `PRE_*_HOOKS` support for all build steps.

# In 2014.08 (1)

- **Architectures:**
  - Powerpc64 BE/LE added, AVR32 deprecated.
  - Improved altivec / SPE /atomic instructions handling. Additional PowerPC CPU variants added.
- **Defconfigs:** Atmel SAMA5D3, Congatec QMX6, Lego ev3, TS-5x00, qemu-system-xtensa, qemu-aarch64-virt added. A number of tweaks to existing ones. lpc32xx defconfigs removed.
- **Toolchain:**
  - Microblaze support for internal musl toolchain.
  - Default to GCC 4.8 for internal toolchain, remove deprecated 4.3 and 4.6 versions.
  - External CodeSourcery / Linaro toolchain updates
  - Option to copy gconv libraries for external toolchains.

# In 2014.08 (2)

- **Infrastructure:**
  - `graph-depends` improvements
  - Download handling is now done using helper scripts.
  - Integrity of downloads can now be verified using hashes
  - Legal-info: License info of local or overridden packages are saved as well. Toolchain packages are also taken into account.
  - autotools: Static linking with libtool / v1.5 improvements
  - Gettextize support, similar to autoreconf
  - kconfig package infrastructure added
- User manual restructured / reworked

# In 2014.11

- **Toolchains:**
  - Use `-mcpu` / `-march` instead of `-mtune`
  - Support additional ARC and sparc variants
  - Updated Code sourcery and Linaro external toolchains
- **Defconfigs:** Freescale iMX6DL SabreSD, Minnowboard MAX, QEMU powerpc64 pseries added and a number of updates to the existing configurations.
- **Infrastructure:**
  - Buildroot is now less noisy when built with the silent option (`make -s`)
  - A number of package infrastructure variables have been renamed from `*_OPT` to `*_OPTS` for consistency
  - Option to choose what shell /bin/sh points to
- **Documentation:**
  - Various updates to the user manual
  - The asciidoc documentation handling has now been extended so it can be used by `BR2_EXTERNAL`

# In 2015.02 (1)

- **Static/shared library handling reworked**
  - This is now a tristate (shared only / shared and static / static only)
  - Default is now shared only to speed up the build. `BR2_PREFER_STATIC_LIB` is now called `BR2_STATIC_LIBS`
- **Toolchain:**
  - The toolchain (internal and external) will now warn when an unsafe library or header path is used
  - If `BR2_COMPILER_PARANOID_UNSAFE_PATH` is enabled under build options this instead becomes an error.
- **Architectures:** Freescale E5500 and E6500 PowerPC support added, deprecated MIPS 1/2/3/4 support removed.
- **Defconfigs:** Freescale p2020ds, MIPS creator CI20, Raspberrypi with DT, UDOO Quad

# In 2015.02 (2)

- `make <foo>_defconfig` now saves the path to the defconfig in the `.config`, so a `make savedefconfig` automatically updates it
- Infrastructure for packages using the Erlang rebar tool has been added.
- Hashes for a large number of packages have been added. Hashes are now checked for both target and host packages.
- The system menu now has an option to automatically configure a network interface through DHCP at bootup.
- The default filesystem skeleton now uses a separate `tmpfs` for `/run` instead of a symlink to `/tmp/` for security reasons / to protect against conflicts with user generated temporary files.
- BR2_EXTERNAL is now exported to post-build and post-image scripts.

# In 2015.05 (1)

- **Architectures:**
  - Removed AVR32 support, SuperH64 deprecated
  - Added support for steamroller, corei7-avx and core-avx2 x86 variants.
- **Toolchains:**
  - IPv6 and Largefile support now enforced for uClibc. Corresponding Kconfig symbols removed.
  - External CodeSourcery AMD64 2014.05 added
  - musl-cross 1.1.6 added
  - CodeSourcery SuperH 2 and Xilinx Microblaze v2/14.3 removed
  - Distro-class external toolchains are now detected and blacklisted
  - Internal toolchain support for Nios2 added, Blackfin removed.
  - Aarch64 and sh musl support.
  - uClibc-ng support added
  - Libatomic is now handled for internal and external toolchains.
  - Link time optimization (LTO) support.

# In 2015.05 (2)

- **Defconfigs:** Freescale i.MX28 EVK, i.MX31 PDK and SABRE Auto, Raspberry Pi 2, RIoTboard
- **Infrastructure:**
  - Hashes for a large number of packages have been added.
  - Missing hashes now stop the build unless explicitly disabled.
  - Spaces and colons (:) are now supported in package versions.
  - Dependencies can now be listed for the patch step (`<PKG>_PATCH_DEPENDENCIES`).
  - Kconfig and Linux kernel extensions infrastructure has been added.
  - Makedevs now has a recursive (r) option
  - `external-deps`, `legal-info`, `source`, `source-check` have been reimplemented using the package infrastructure, so their output/behaviour may differ from earlier (some packages were not included in the past).

# In 2015.08

- **Architectures:** Minimal support for ARM Cortex-M3 and AArch64 big-endian.
- **Toolchains:** Use *uClibc-ng* by default, add gcc 5.x support, update toolchain components
- **Defconfigs:** VIA VAB-820/AMOS-820, OLimex OLinuxino A20 Lime, many Atmel evaluation boards, ACME Systems Aria G25, WarPboard, Altera Cyclone 5 Development Board, Xilinx zc706, ARC AXS101 and AXS103
- **Infrastructure:**
  - Predictable permissions in the generated rootfs
  - Support for *kconfig fragments*
  - New `kernel-module` infrastructure
  - Rework of the skeleton and init scripts packaging
  - New `linux-tools` infrastructure in the `linux` package
  - GCC version dependency mechanism
- **Filesystems:** Complete rework of the ISO9660 support.

# Acknowledgements


- Free Electrons would like to thank the following members of the Buildroot community for their useful comments and reviews during the development of these training materials:
  - Thomas De Schampheleire
  - Peter Korsgaard
  - Yann E. Morin
  - Arnout Vandecappelle
  - Gustavo Zacarias

# Last slides

***free electrons***

Embedded Linux
Experts


Corrections, suggestions, contributions and translations are welcome!

# Thank you!

## And may the Source be with you