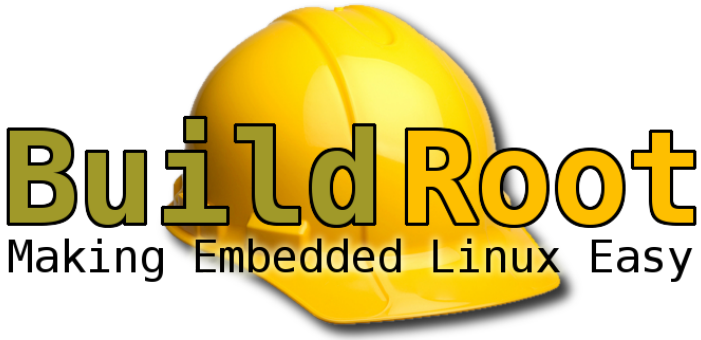
Using *Buildroot* for real projects



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

Thomas Petazzoni

- Embedded Linux engineer and trainer at Free Electrons since 2008
  - Embedded Linux development: kernel and driver development, system integration, boot time and power consumption optimization, consulting, etc.
  - Embedded Linux and driver development training, with materials freely available under a Creative Commons license.
  - `http://www.free-electrons.com`
- Major contributor to Buildroot, an open-source, simple and fast embedded Linux build system
- Living in Toulouse, south west of France

Agenda

- What is Buildroot ?
- How does it work ?
- Example systems generated by Buildroot
- Recommendations for real projects
  - Toolchain recommendations
  - Project specific configuration and files
  - Project specific packages
  - Enabling application developers
  - Misc best practices
- Features in 2011.11 making things better
- Conclusion

What is Buildroot ? (1/2)

- Buildroot is an **embedded Linux build system**
- Its goal is to build
  - a **cross-compiling toolchain**
  - a **root filesystem** with multiple cross-compiled libraries and applications
  - a **kernel image** and **bootloader** images

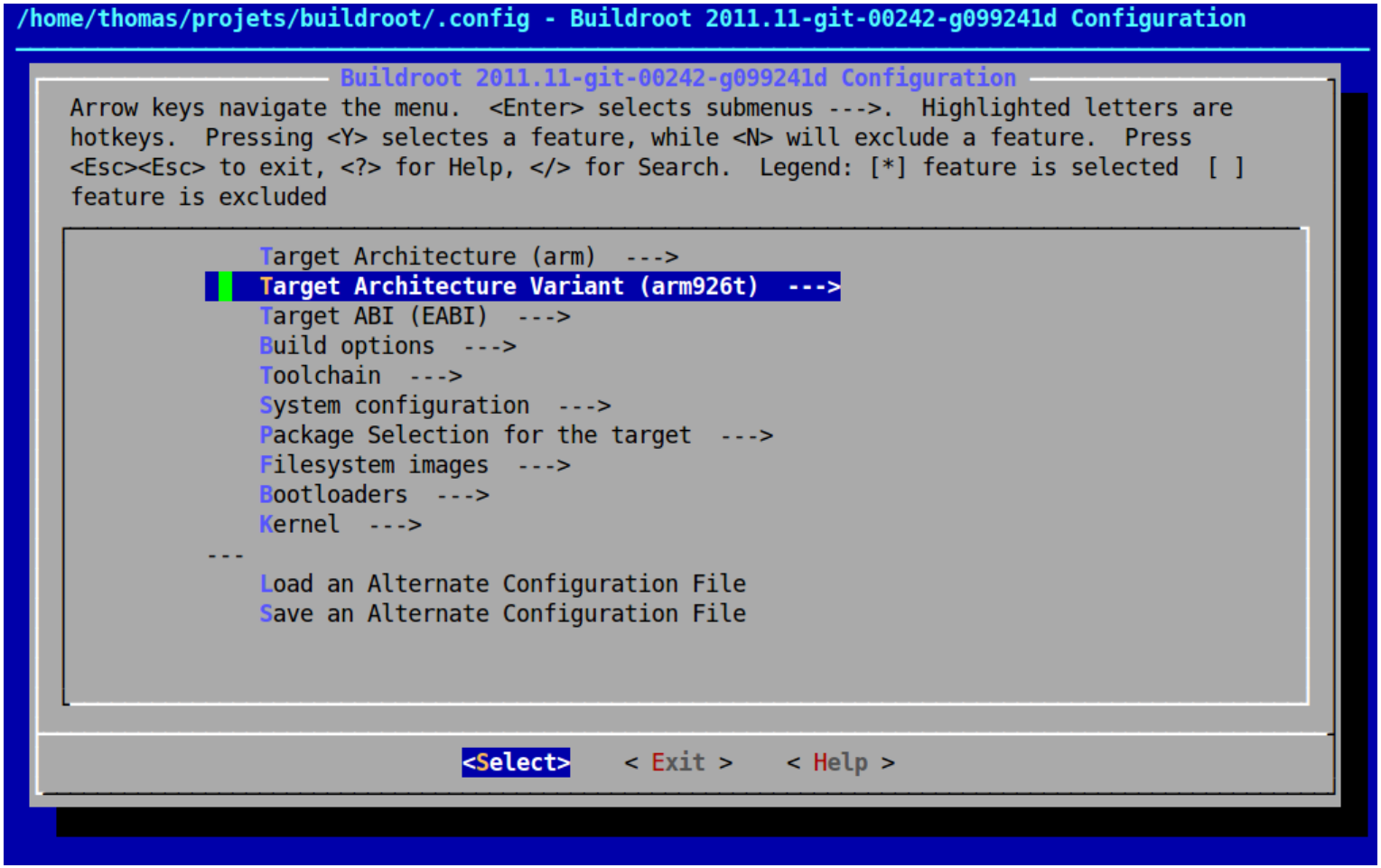
  or any combination of these
- It has a **kconfig** configuration mechanism, identical to the one used in the kernel
- It is completely written in **make**
- **It builds only what's necessary**. A base system, containing just Busybox, takes less than 2 minutes to be built from scratch.

What is Buildroot ? (2/2)

- It is designed with **simplicity** in mind
  - Standard languages used, relatively lightweight infrastructure.
  - Very easy to add packages or customize the build system behaviour.
- It is best-suited for **small to medium-sized embedded systems**
  - In the generated root filesystem, Buildroot doesn't track which source package installed what. You need to do complete rebuilds for a clean root filesystem after configuration changes.
  - Generates systems with no package management mechanism on the target (no ipkg, or apt)
- 700+ packages
- **Stable releases** published every three months
- **Active user/developer community**

Basic usage (1/2)

```
$ make menuconfig
```



Basic usage (2/2)

```
$ make
...
...
...
$ ls output/images/
dataflash_at91sam9m10g45ek.bin  rootfs.tar
rootfs.ubi                      rootfs.ubifs
u-boot.bin                      u-boot-env.bin
uImage
```

Ready to use !

How does it work ?

- Configuration options defined in `Config.in` files, and stored in a `.config` file
- Buildroot starts with the toolchain: either it generates it, or imports an existing toolchain, or uses *crosstool-NG* to generate one
- It creates a basic root filesystem from a *skeleton* (just a few configuration files)
- Once the toolchain is ready, Buildroot goes through the list of selected packages. It simply fetches, configures, builds and installs all packages, respecting their dependencies.
- It builds the kernel image and/or bootloader images, if requested
- It creates one or more root filesystem images, using *fakeroot*

Top-level source code organization (1/2)

- `board/`
  contains hardware-specific and project-specific files. Covered later.
- `boot/`
  contains config options and recipes for various bootloaders
- `configs/`
  the default configurations. Covered later.
- `docs/`
  Yes, we have some documentation.
- `fs/`
  contains config options and makefiles to generate the various filesystem images (jffs2, ubifs, ext2, iso9660, initramfs, tar and more). Also contains the root filesystem *skeleton*

Top-level source code organization (2/2)

- `linux/`
  contains the config options and makefile to generate the Linux kernel, and also the kernel part of real-time extensions (Xenomai, RTAI, etc.)
- `package/`
  contains the config options and makefiles for all userspace packages
- `support/`
  various misc stuff needed for the build (*kconfig* code, etc.)
- `target/`
  mostly historic directory. No longer contains anything useful besides the default *device table*.
- `toolchain/`
  config options and makefiles to build or import the toolchain

What does it generate?

In the output directory (`output` by default, but out-of-tree build with `O=` is supported):

- `build/`, a directory with one sub-directory per component built. The directory contains the source of that component, and this is where it is built.
- `host/`, a directory that contains the utilities built for the host machine, including the toolchain
  - A subdirectory, `host/usr/<tuple>/sysroot/` contains the toolchain *sysroot*, with all libraries and headers needed to build applications for the target.
- `staging/`, a historical symbolic link to `host/usr/<tuple>/sysroot`
- `target/`, the target root filesystem (without device files)
- `images/`, where final images are stored

Example 1: Multi-function device

- The device is an ARM AT91-based platform with GPS, RFID readers, GSM modem, Ethernet and USB.
- The Buildroot configuration:
  - CodeSourcery ARM glibc toolchain
  - *Linux kernel*
  - *Busybox* for the basic system
  - *Dropbear* for SSH access (debugging)
  - *Qt* with only *QtCore*, *QtNetwork* and *QtXml*, no GUI
  - *QExtSerialPort*
  - *zlib*, *libxml2*, *logrotate*, *pppd*, *strace*, a special RFID library, *popt* library
  - The *Qt* application
  - JFFS2 root filesystem
- Filesystem size: 11 MB. Could be reduced by using *uClibc*.
- Build time: 10 minutes on a fast build server (quad-core i7, 12 GB of RAM)

Example 2: vehicle navigation system

- An x86-based system, with an OpenGL application for vehicle navigation system.
  - External *glibc* toolchain generated with *crosstool-NG*
  - The Grub bootloader
  - *Linux kernel*, of course
  - *Busybox*
  - A part of the *X.org* stack (the server, a few drivers, and some client libraries), including *libdrm*, *Mesa*
  - The *fglrx* ATI proprietary OpenGL driver
  - *ALSA utils*, *ALSA library*, *V4L library*, *Flashrom*, *LM Sensors*, *Lua*, *Dropbear*, *Ethtool*
  - The OpenGL application and its data
- Filesystem size: 95 MB, with 10 MB of application (binary + data) and 45 MB (!) of *fglrx* driver.
- Build time: 27 minutes on a fast build server (quad-core i7, 12 GB of RAM)

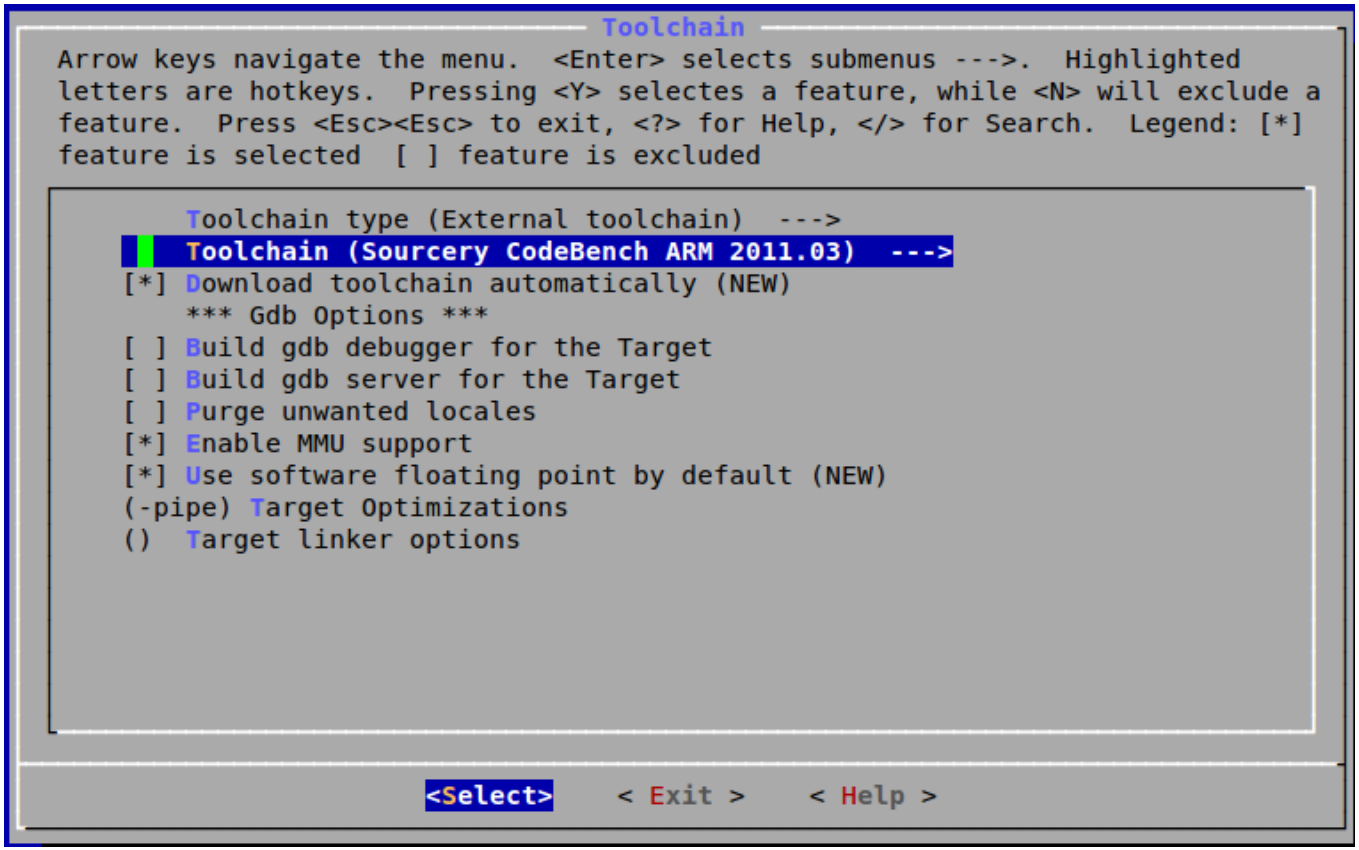
Toolchain

Three choices:

- **Buildroot builds a toolchain**. This is limited to *uClibc* based toolchains, and the toolchain is rebuilt completely at every complete Buildroot rebuild.
- **Buildroot uses crosstool-NG** as a back-end to generate the toolchain. More choices available (*glibc*, *eglibc*, *uClibc* supported, more *gcc* versions, etc.). However, requires toolchain rebuild at every complete Buildroot rebuild.
- **Buildroot can import external toolchains**. This is definitely the mechanism I recommend.
  - Allows to re-use *CodeSourcery* toolchains, or custom toolchains built with *crosstool-NG* or *Buildroot*
  - Importing the toolchain into Buildroot takes just a few seconds, which saves the toolchain build time at every Buildroot rebuild
  - The toolchain rarely needs to be changed compared to the root filesystem, so this mechanism makes sense

External toolchains: toolchain profile

Buildroot contains presets for well-known binary toolchains such as the CodeSourcery ones, for which Buildroot knows the properties (C library used, etc.) and the download location.



External toolchains: custom toolchain

Buildroot can also use custom, locally installed toolchains, in which case one must tell Buildroot a few details about the toolchain (C library being used, location, prefix, etc.)

```
                        Toolchain
Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted
letters are hotkeys.  Pressing <Y> selectes a feature, while <N> will exclude a
feature.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*]
feature is selected  [ ] feature is excluded

        Toolchain type (External toolchain)  --->
        Toolchain (Custom toolchain)  --->
    (/home/thomas/blabla/) Toolchain path
    (i686-unknown-linux-gnu) Toolchain prefix
        External toolchain C library (uClibc)  --->
    [ ] Toolchain has large file support? (NEW)
    [ ] Toolchain has IPv6 support? (NEW)
    [ ] Toolchain has RPC support? (NEW)
    [ ] Toolchain has WCHAR support? (NEW)
    [ ] Toolchain has locale support? (NEW)
    [*] Toolchain has threads support? (NEW)
    [*] Toolchain has C++ support?
    ()  Extra toolchain libraries to be copied to target
```

External toolchains HOWTO

- You can either:
  - Use an existing pre-built toolchain, available publicly from the Web
  - Generate a custom toolchain, and store it pre-built next to the Buildroot directory on the local filesystem, or in some internal FTP/HTTP server
- And then point your Buildroot configuration to this toolchain:
  - As a *custom* toolchain
  - Or by adding a new *profile* so that the toolchain gets downloaded automatically and all its configuration is already known to Buildroot
- This way, users of the Buildroot environment don't have to worry about the toolchain and don't have to wait for it to build.

Project specific files

For each project, the recommendation is to create a directory

`board/<company>/<project>/`

which will be used to store:

- Configuration files
- Root filesystem additions
- Project-specific kernel/bootloader patches
- All other project specific files

Kernel and bootloader changes

- The kernel and bootloaders often require modifications for a particular hardware platform.
- My typical workflow is the following :
  - Have a branch in a kernel Git tree in which I commit the necessary changes
  - Generate patches with `git format-patch`
  - Import them into Buildroot in the `board/<company>/<project>/linux-patches/` directory
  - Configure Buildroot so that it applies the patches before building the kernel (option `BR2_LINUX_KERNEL_PATCH`)
- The advantage is that the Buildroot environment is easy to use for others (no external dependency on a kernel Git tree)
- The drawback is that Buildroot does not directly use your kernel Git tree. This has been improved recently, covered later.

Kernel and bootloaders example (1/2)

```
$ ls board/<company>/<project>/
linux-2.6.39.config          samba-script.tcl
linux-2.6.39-patches/        u-boot-1.3.4-patches/
at91bootstrap-1.16-patches/

$ ls board/<company>/<project>/linux-2.6.39-patches/
linux-0001-foobar.patch linux-0002-barfoo.patch

$ ls board/<company>/<project>/u-boot-1.3.4-patches/
u-boot-0001-barfoo.patch u-boot-0002-foobar.patch

$ ls board/<company>/<project>/at91bootstrap-1.16-patches/
at91bootstrap-0001-bleh.patch at91bootstrap-0002-blah.patch
```

Kernel and bootloaders example (2/2)

```
BR2_TARGET_UBOOT=y
BR2_TARGET_UBOOT_BOARDNAME="company_project"
BR2_TARGET_UBOOT_1_3_4=y
BR2_TARGET_UBOOT_CUSTOM_PATCH_DIR=
  "board/company/project/u-boot-1.3.4-patches/"
# BR2_TARGET_UBOOT_NETWORK is not set
BR2_TARGET_AT91BOOTSTRAP=y
BR2_TARGET_AT91BOOTSTRAP_BOARD="at91sam9m10g45ek"
BR2_TARGET_AT91BOOTSTRAP_CUSTOM_PATCH_DIR=
  "board/company/project/at91bootstrap-1.16-patches/"
BR2_LINUX_KERNEL=y
BR2_LINUX_KERNEL_CUSTOM_VERSION=y
BR2_LINUX_KERNEL_CUSTOM_VERSION_VALUE="2.6.39"
BR2_LINUX_KERNEL_PATCH=
  "board/company/project/linux-2.6.39-patches/"
BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG=y
BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE=
  "board/company/project/linux-2.6.39.config"
```

Root filesystem customization (1/3)

The Buildroot process to create the root filesystem is:

- Copy a *skeleton* to the `output/target` directory. The default one is in `fs/skeleton`, but a different one can be used.
- Copy base libraries (C libraries and al.) from the toolchain *sysroot* into the target root filesystem
- Install all packages
- Execute a configuration-specified post build script
- Generate the filesystem images in the selected formats (jffs2, ubifs, etc.)

Root filesystem customization (2/3)

Depending on which modifications are necessary, you might need to:

- Use a **different filesystem skeleton** if the default skeleton is *really* not appropriate. If only tiny modifications are needed, do them in the post build script. I don't recommend this solution as it consists in duplicating the default skeleton, which would prevent from taking advantage of future improvements of the default skeleton.
- **Create packages** to install applications, data files, etc. The skeleton is really only for basic config files.
- **Create a post build script** that adjusts the root filesystem as needed.

Root filesystem customization (3/3)

- Create a post-build script in `board/<company>/<project>/post-build.sh` and tell Buildroot to run it with the `BR2_ROOTFS_POST_BUILD_SCRIPT`.
- This script is executed from the Buildroot main directory and receives the target filesystem directory as argument.
- Typically, my script does:
  - Tune the `/etc/inittab`, `/etc/fstab` or `/etc/securetty` files
  - Copy the contents of `board/<company>/<project>/rootfs-additions/` into the target root filesystem.
    In this directory, I put additional init scripts, configuration files for network services, or override existing files.

Root filesystem customization: example script

```
TARGETDIR=$1

# Set root password to 'root'. Password generated with
# mkpasswd, from the 'whois' package in Debian/Ubuntu.
sed -i 's%^root::%root:8kfIfYHmcyQEE:%' $TARGETDIR/etc/shadow

# Application/log file mount point
mkdir -p $TARGETDIR/applog
grep -q "^/dev/mtdblock7" $TARGETDIR/etc/fstab || \
    echo "/dev/mtdblock7\t\t/applog\tjffs2\tdefaults\t\t0\t0" \
    >> $TARGETDIR/etc/fstab

# Copy the rootfs additions
cp -a $BOARDDIR/rootfs-additions/* $TARGETDIR/
```

/dev managament

Buildroot offers four solutions for /dev management:

- **static**, where device files are created at build time. They are listed in a configurable *device table*.
- **devtmpfs**, the kernel filesystem which creates dynmically device nodes. Buildroot makes sure that your kernel has the right options selected. Recommended solution.
- **devtmpfs+mdev**, which allows to trigger scripts or applications on device insertion/removal.
- **devtmpfs+udev**, the full-fledge solution.

Whichever solution is choosen, there is still a *device table* used, which is used to set specific permissions and ownership on certain files or directories. The list of *device tables* is configured through BR2_ROOTFS_DEVICE_TABLE.

Project-specific packages (1/5)

- A project typically requires one or more specific packages, containing project-specific applications or libraries.
- In order to make the build process fully integrated, those can be integrated in Buildroot.
- In order to isolate your packages from all other packages, create a sub-directory for all your packages:
  - Create a `package/<company>/` directory
    This allows to isolate your specific packages from all other packages.
  - Put a `package/<company>/<company>.mk` files into it, with just:
    `include package/<company>/*/*.mk`
    This tells Buildroot to parse the makefiles of your packages.
- Obviously, if you need to package an existing open-source component, package it under `package/` and submit the patch to the Buildroot community.

Project-specific packages (2/5)

For each package:

- Create a `package/<company>/<yourpkg>/` directory
- Create a `package/<company>/<yourpkg>/<yourpkg>.mk` makefile that contains the recipe to build your package
- Create a `package/<company>/<yourpkg>/Config.in` file that details the configuration options of your package. At least one is mandatory, in order to enable/disable your package.
- From `package/Config.in`, source your `package/<company>/<yourpkg>/Config.in`, preferably in a company-specific menu to make merges with future Buildroot versions easier.

Project-specific packages (3/5)

To create the `.mk` file, three infrastructures are available:

- The **AUTOTARGETS** infrastructure, for autotools-based packages.
  You describe the tarball URL and version, the dependencies and configuration options, and Buildroot does all the rest.
- The **CMAKETARGETS** infrastructure, for CMake-based packages.
  Here as well, Buildroot does most of the work.
- The **GENTARGETS** infrastructure, for other packages not using a well-known build system.
  Here, Buildroot has no knowledge of the build system, so you have to specify what needs to be done to configure, build and install the component.

See the Buildroot documentation for more details on using these package infrastructures.

Project-specific packages (4/5)

- Typically, the package source code is downloaded as a tarball from HTTP/FTP or from a Git/Mercurial/Subversion repository.
- For project-specific applications or libraries, it might be useful to store them locally.
  - For small utilities, it can be directly in the package directory, in an `src/` subdirectory.
  - For larger programs or libraries having their own version control, it is possible to override the package extraction commands to copy their source code in the build directory

Project-specific packages (5/5)

The application source code is stored in `../company-application/` relative to the Buildroot source code.

```
MY_APPLICATION_VERSION = 1.0
MY_APPLICATION_DEPENDENCIES = \
  qextserialport host-pkg-config

define MY_APPLICATION_EXTRACT_CMDS
        cp -a $(TOPDIR)/../company-application/* $(@D)/
endef

define MY_APPLICATION_INSTALL_TARGET_CMDS
        cp $(@D)/myapp $(BINARIES_DIR)
endef

$(eval $(call CMAKETARGETS))
```

During application development

- Buildroot is really an *integration* utility. Once a package has been built, it is not rebuilt, even if its source code changes.
- When working on the development of a component, it is usually more convenient to build it outside of Buildroot, for a quicker compile/test/debug cycle.
- The upcoming *source directory override* feature, covered later, makes it easier to use Buildroot during development.

Using Buildroot to build external components

- The `output/host` directory is the Buildroot SDK.
- `output/host/usr/bin/ARCH-linux-*` are the cross-compilation utilities. A wrapper is used so that the compiler is passed the appropriate `--sysroot` option. From the user perspective, the compiler therefore automagically finds the libraries and headers.
- `output/host/usr/bin` also contains other useful utilities for the build process
  - `pkg-config`, which is configured to look for libraries in the right location by default.
  - `qmake`, if Qt is used, also configured with the correct paths
- `output/toolchainfile.cmake` is a *CMake* description of the toolchain. Can be passed to `cmake` using `-DCMAKE_TOOLCHAIN_FILE=...`.

Using Buildroot to build external components

A simple application

```
BRPATH/output/host/usr/bin/arm-unknown-linux-gcc -o prog prog.c
  $(BRPATH/output/host/usr/bin/pkg-config --libs --cflags glib-2.0)
```

Autotools component

```
export PATH=$PATH:BRPATH/output/host/usr/bin/
./configure --host=arm-unknown-linux
```

CMake component

```
cmake -DCMAKE_TOOLCHAIN_FILE=BRPATH/output/toolchainfile.cmake
make
```

Forcing Buildroot to rebuild a package

- Buildroot keeps **stamp files** in the package build directory `output/build/pkg-version/`. They are named `.stamp_extracted`, `.stamp_configured`, `.stamp_built`, `.stamp_target_installed`, etc.
- Removing a stamp file and re-executing `make` will **force Buildroot to do the corresponding step** again.
- Removing the complete package build directory will **force Buildroot to rebuild the package from scratch**.
- This will be improved in the upcoming Buildroot version.

Using NFS during application development

- Mounting the root filesystem over NFS is very practical during development
- The `output/target` directory created by Buildroot cannot directly be used for NFS mount, because it does not contain any device file and the permissions may not be correct. This is because Buildroot runs as non-root.
- The recommended way is:
  - Ask Buildroot to generate a tarball image of the root filesystem
  - Uncompress this tarball image, as root, in some directory exported by NFS.
  - `make && tar -xf -C /nfsroot/ output/images/rootfs.tar`
- `/nfsroot` should be exported with the NFS options `rw` and `no_root_squash` in `/etc/exports`

Storing the project configuration

- Once you have defined a Buildroot configuration for your project, you want to save it somewhere and allow others to use it.
- Just do:
  - `make savedefconfig`
    Creates a `defconfig` file in the top Buildroot source directory.
  - `mv defconfig configs/company_project_defconfig`
- Others can then load this configuration very easily:
  - `make company_project_defconfig`

Summarizing the project-specific bits

- In the end, your project-specific bits are in:
  - `board/<company>/<project>/` for the kernel and bootloader patches and configuration, post-build script and root filesystem overlay
  - `configs/company_project_defconfig` for the Buildroot configuration.
  - `package/company/` for your specific packages
- Your modifications are cleanly isolated from the Buildroot core, making it easy to upgrade Buildroot when needed.

Be prepared for offline builds

- When working on a project, you'll want to make sure that your build can be done offline in order to not depend on resources that may disappear from the Net.
- Here are some useful Buildroot features to do so:
  - `make source` will download into the Buildroot download cache *all* the files required to do the build
  - `make external-deps` will list the name of all the files (essentially tarballs) that are required to do the build. Identifies the useful tarballs in the Buildroot download cache.
  - With the `BR2_PRIMARY_SITE` configuration option, Buildroot will download files from a specified HTTP/FTP server *before* trying to reach the official site for each package. Useful to create an internal server.
  - The location of the local download cache can be configured with `BR2_DL_DIR` or with the `BUILDROOT_DL_DIR` environment variable.

In 2011.11: local method

- The upcoming 2011.11 release will natively support packages whose source code is in a local directory.
- Simply need to use:
  `MYPKG_SITE = /some/local/directory`
  `MYPKG_SITE_METHOD = local`
- Buildroot will *rsync* the source code from the specified location into its build directory in `output/build/pkg-version`.

In 2011.11: source directory override

- A package typically specifies a HTTP or FTP location for its tarball
- You might want to use a local source code for this package, instead of the official tarball
- The upcoming 2011.11 allows to specify an *override file*
  - This file is simply a *makefile* with variable assignments
  - Those variable assignments allows you to tell Buildroot: "for package foo, use the source code in this directory rather than the normal tarball location"
  - `ZLIB_OVERRIDE_SRCDIR = /home/foo/my-zlib/`
  - `LINUX_OVERRIDE_SRCDIR = /home/foo/linux-2.6/`
- The source code is *rsync'ed* from the given source directory into the Buildroot build directory.

In 2011.11: other features

- Addition of `make <pkg>-reconfigure` and `make <pkg>-rebuild`
  - Instead of manually manipulating the stamp files, those commands restart the package build process at the configuration stage or at the compilation stage.
  - Makes using *Buildroot* during kernel or application development a lot easier.
- Support for fetching from Mercurial repository
- Support for fetching using `scp`, both for packages and for the *primary download site*

Conclusion

- Buildroot has multiple features making it easy to customize the generated system
  - Addition of kernel and bootloader patches and configuration
  - Addition of project specific configuration files and scripts
  - Addition of project specific packages
- These features makes Buildroot suitable to generate embedded Linux systems for a wide range of projects, with a preference on moderately large systems.
- From our experience, Buildroot also remains simple enough to be used and understood by non-Linux experts.

Questions?

Thomas Petazzoni

thomas.petazzoni@free-electrons.com

Special thanks to the Buildroot community, including Peter Korsgaard, Baruch Siach, Thomas de Schampheleire, Arnout Vandecappelle and Will Moore for their comments and suggestions on this presentation.