

# Buildroot vs Yocto: Differences for Your Daily Job

---

Luca Ceresoli — AIM Sportline

luca@lucaceresoli.net

<http://lucaceresoli.net>

ELC-E 2018

# About me



- Embedded Linux engineer at AIM Sportline  
<http://www.aim-sportline.com/>
  - Develop products on custom hardware
  - Kernel, drivers, bootloader, FPGA
  - Integration, build system
- Open source enthusiast
  - Contributor to Buildroot, the Linux kernel and a few other projects

# Introduction

---

## This is not...

- This is not a tutorial

## This is not...

- This is not a tutorial
- This is not a feature comparison, not a selection guide

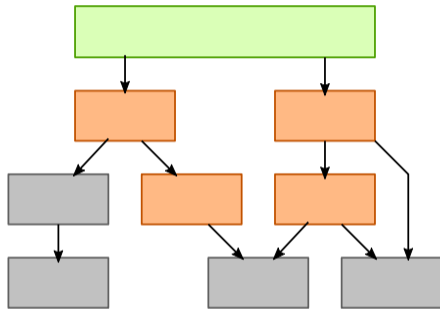
## This is not...

- This is not a tutorial
- This is not a feature comparison, not a selection guide
- If you need one:
  - Buildroot vs. OpenEmbedded/Yocto: A Four Hands Discussion, Belloni and Petazzoni, ELC 2016 (slides and video online)
  - <http://www.jumpnowtek.com/linux/Choosing-an-embedded-linux-build-system.html>
  - <https://opensource.com/article/18/6/embedded-linux-build-tools>

## This is not...

- This is not a tutorial
- This is not a feature comparison, not a selection guide
- If you need one:
  - Buildroot vs. OpenEmbedded/Yocto: A Four Hands Discussion, Belloni and Petazzoni, ELC 2016 (slides and video online)
  - <http://www.jumpnowtek.com/linux/Choosing-an-embedded-linux-build-system.html>
  - <https://opensource.com/article/18/6/embedded-linux-build-tools>
- Fact: both tools have pros and cons

Similar...

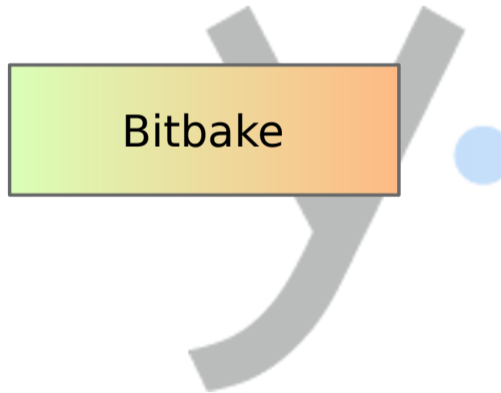
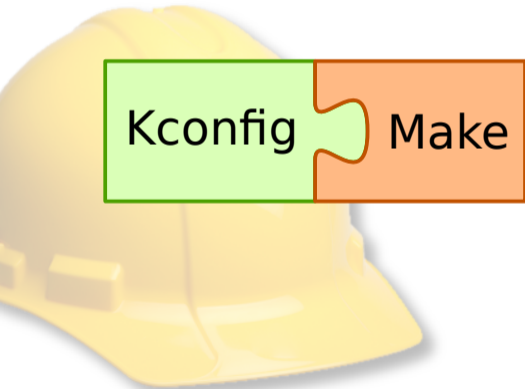


In a nutshell:  
a dependency graph  
with actions to build each node.

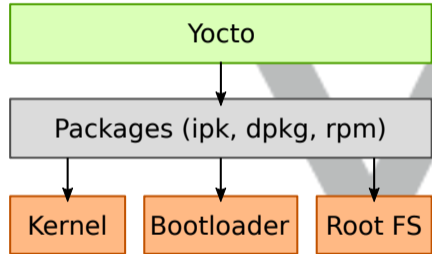
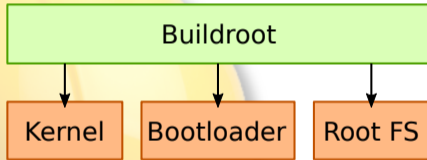




...but different — based on different tools



## ...but different — root filesystem VS distribution



- Bootstrapping
- Naming
- Writing recipes
- Layers / external trees
- Building
- Understanding what's going on
- Customizing the root filesystem
- Tweaking recipes

# Bootstrapping

---

- Get the sources
  - `git clone git://git.buildroot.net/buildroot; cd buildroot`



# Ingredients

1. Get the Poky sources (bitbake, oe-core)
  - `git clone -b sumo git://git.yoctoproject.org/poky; cd poky`
2. You'll probably need more recipes
  - `git clone -b sumo git://git.openembedded.org/meta-openembedded`
3. Additional layers can be useful
  - SoC/board vendor BSP layer, additional software, ...
  - <http://layers.openembedded.org/layerindex/branch/master/layers/>

# Configure

- Smooth start: find a defconfig for a similar board
  - `make list-defconfigs` # minimal booting configs
  - `make similar_board_defconfig`
- Or from scratch
  - Find kernel and U-Boot sources that work for your SoC
  - `make menuconfig`
    - Target: architecture, CPU features
    - Kernel: where to fetch it from, defconfig, dtbs
    - U-Boot: where to fetch it from, defconfig



Kernel

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> selects a feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] feature is selected [ ] feature is excluded

[\*] Linux Kernel

\*\*\* Linux kernel in thumb mode may be broken with binutils >= 2.29 \*\*\*

Kernel version (Custom version) --->

(4.17.4) Kernel version

() Custom kernel patches

Kernel configuration (Using an in-tree defconfig file) --->

(imx\_v6\_v7) Defconfig name

() Additional configuration fragment files

Kernel binary format (zImage) --->

Kernel compression format (gzip compression) --->

[\*] Build a Device Tree Blob (DTB)

(imx6q-sabresd imx6dl-sabresd imx6qp-sabresd) In-tree Device Tree Source file names

() Out-of-tree Device Tree Source file paths

[ ] Install kernel image to /boot in target

[\*] Needs host OpenSSL

[ ] Needs host libelf

Linux Kernel Extensions --->

Linux Kernel Tools --->

<Select>

< Exit >

< Help >

< Save >

< Load >



- `. oe-init-build-env` # creates and enters the build/ dir
- Smooth start: find a defconfig for a similar board
  - `ls conf/machine/` in your SoC vendor layer
  - Set `MACHINE` `?= "<similar_machine>"` in `conf/local.conf`

- make
  - Without parameters builds “all”



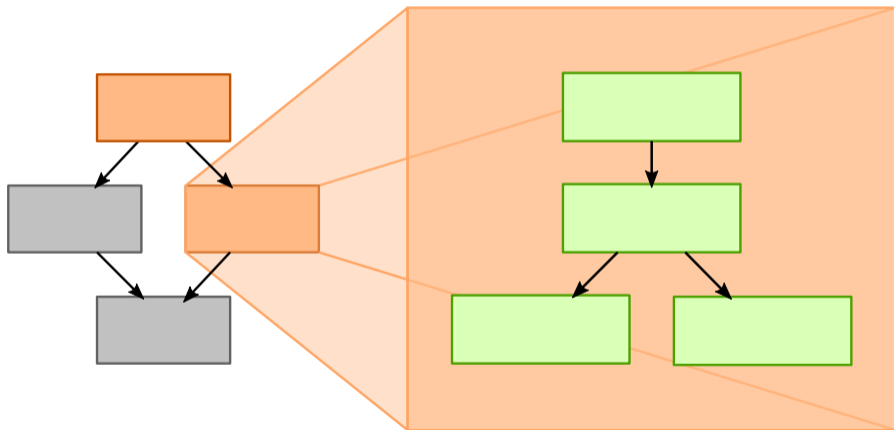
- `bitbake <IMAGE>`
- `bitbake core-image-minimal`



# Naming

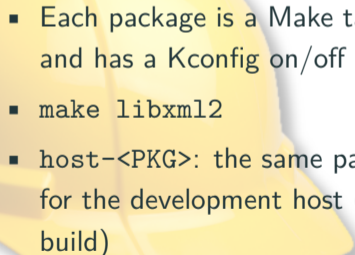
---


## Building items



# Package == Recipe

Rules to download and “build” a single program, library or other  
(e.g. binutils, busybox, gcc, libxml2)

- 
- Each package is a Make target and has a Kconfig on/off knob
  - `make libxml2`
  - `host-<PKG>`: the same package built for the development host (native build)

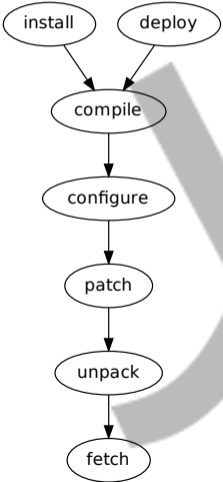
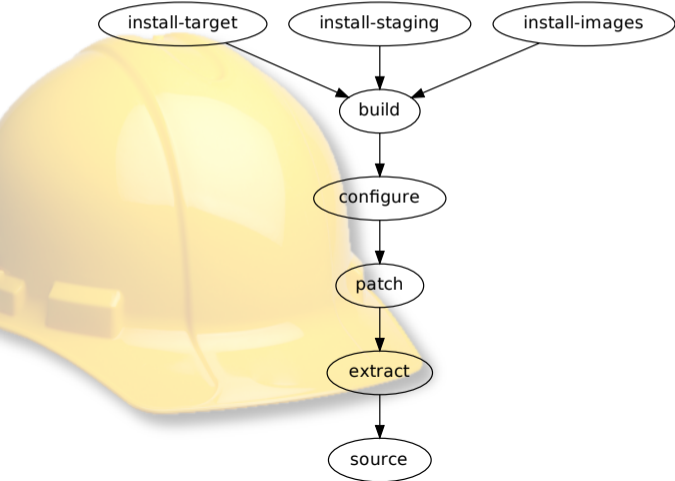
- 
- Each package is a Bitbake target
  - `bitbake libxml2`
  - `<PKG>-native`: the same package built for the development host (native build)

## Step == Task

Each package requires several steps to be built

- No formal name, usually called just *steps*
- `source`, `extract`, `patch`, `configure`, `build`, `install`, ...
- Each step is also a make target
- The special `<PKGNAME>` make target depends on all other normal tasks required to 'build' a recipe
- `make libxml2-configure`  
`host-binutils-build busybox`
- Called *tasks* (often prefixed with `do_`)
- `fetch`, `unpack`, `patch`, `configure`, `compile`, `install`, `deploy`, ...
- First-class citizens in bitbake
- The special `build` task depends on all other normal tasks required to 'build' a recipe
- `bitbake -c configure`  
`libxml2 busybox`

# Default steps/tasks





## Naming side-by-side



Package	Recipe
Package Step	Recipe Task
host-<PKG>	<PKG>-native
pkg-generic.mk	base.bbclass
source	fetch
extract	unpack
patch	patch
configure	configure
build	compile
install-{target,staging}	install
install-images	deploy



## Layers / external trees

---

The preferred way to add features: layers

conf/bblayers.conf

```
BBLAYERS ?= " \  
  /home/murray/devel/poky/meta \  
  /home/murray/devel/poky/meta-poky \  
  /home/murray/devel/poky/meta-yocto-bsp \  
  <...path to other layers...> \  
"
```

## Yocto: adding layers

```
BBLAYERS ?= " \  
  /home/murray/devel/poky/meta \  
  /home/murray/devel/poky/meta-poky \  
  /home/murray/devel/poky/meta-yocto-bsp \  
+  ${TOPDIR}/../meta-my-soc-vendor \  
+  ${TOPDIR}/../meta-openembedded/meta-oe \  
"
```

- Suggestion: use relative paths

- .bbappend files are appended to the .bb file while parsing
  - Change variable values
  - Append/prepend to tasks
- The resulting myrecipe is a concatenation of:
  - <LAYER1>/\*/\*/myrecipe.bb
  - <LAYER2>/\*/\*/myrecipe.bbappend
  - <LAYER3>/\*/\*/myrecipe.bbappend



- Some SoC vendor layers augment the buildsystem, at times creating problems
- Conflict between layers (e.g. in gstreamer)
- Suggestion: add layers one by one, bottom-up, test each time
- Problems?
  - Fix the offending code in your layer (`.bbappend`)
  - disable the recipe (`PNBLACKLIST`) and provide an alternative
  - Don't use the layer, copy only what you need



# Yocto: your top-level layer

- Add your top-level layer
  - Your machine configuration
  - Your proprietary packages
  - `.bbappends` and other files to modify the behaviour of lower layers



## Buildroot: BR2\_EXTERNAL

- BR2\_EXTERNAL is technically similar to Yocto layers, but simpler
- The goal is to add, not modify
- Typical use: add your own product customizations
  - packages
  - Kconfig options
  - defconfigs
  - boards
  - patches
  - ...
- Need to fix/improve a Buildroot package?
  - Suggested policy: do it in the Buildroot code, then submit your improvements upstream





```
$ make BR2_EXTERNAL=~/.devel/myext:~/devel/myext2 menuconfig
```

- The list of your externals is saved in `.config`
- The top-level Makefile will include each external Makefile
- The same for `Config.in` files



## Writing recipes

---

## A simple Yocto package: the .bb file

<MYLAYER>/recipes-app/corporate-apps/foo\_1.0.bb

```
SRC_URI = "http://www.foo.org/download/foo-${PV}.tar.xz"
DEPENDS = "libbar-native libusb"

do_compile() {
    oe_runmake all
}

do_install() {
    install -D -m 0755 ${B}/foo ${D}${bindir}/foo
}
```

## A simple Buildroot package: the makefile

package/foo/foo.mk

```
FOO_VERSION = 1.0
FOO_SITE = http://www.foo.org/download
FOO_DEPENDENCIES = host-libbar libusb

define FOO_BUILD_CMDS
    $(MAKE) $(TARGET_CONFIGURE_OPTS) -C $(@D) all
endef

define FOO_INSTALL_TARGET_CMDS
    $(INSTALL) -D -m 0755 $(@D)/foo $(TARGET_DIR)/usr/bin/foo
endef

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



## A simple Buildroot package: Config.in

- Shows the package in the Kconfig interfaces
- Uses the Kconfig language

package/foo/Config.in

```
config BR2_PACKAGE_FOO
    bool "foo"
    select BR2_PACKAGE_LIBUSB
    help
        A brief description.
```



- *classes* implement common features for reuse in recipes
  - .bbclass files
  - There are classes for the most common build tools: Autotools, CMake

<MYLAYER>/recipes-app/corporate-apps/foo\_1.0.bb

```
SRC_URI = "http://www.foo.org/download/foo-${PV}.tar.xz"
```

```
DEPENDS = "libbar-native libusb"
```

```
inherit autotools
```

# Buildroot package infrastructures

- *package infrastructures* are classes of packages that use the same build tool
  - Autotools, CMake, Python, LuaRocks, Perl/CPAN ...
- Most commands have a default

package/foo/foo.mk

```
FOO_VERSION = 1.0
FOO_SITE = http://www.foo.org/download
FOO_DEPENDENCIES = host-libbar libusb

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



- With classes the common `do_<TASK>` functions are already set
- Customizable via infrastructure-specific variables

```
EXTRA_OECONF += "--enable-warp-speed"
```

- Can be extended with

- `do_<TASK>_prepend`
- `do_<TASK>_append`

```
do_install_append() {  
    touch ${D}${sysconfdir}/foo.conf  
}
```





# Buildroot package infrastructures

- With package infrastructures `FOO_<STEP>_CMDS` are already set
- Customizable via infrastructure-specific variables

```
FOO_CONF_OPTS = --enable-warp-speed
```

- To extend them define hooks


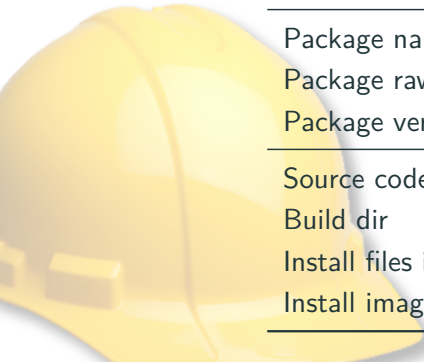
- `FOO_PRE_<STEP>_HOOKS`
- `FOO_POST_<STEP>_HOOKS`

```
define FOO_CREATE_CONF_FILE
    touch $(TARGET_DIR)/etc/foo.conf
endef
FOO_POST_INSTALL_HOOKS += FOO_CREATE_CONF_FILE
```



## Predefined variables


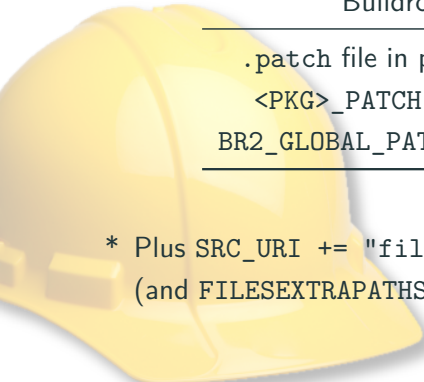
Lots of predefined variables can (and should) be user in rules. The most widely used:



	Buildroot	Yocto
Package name	<PKG>_NAME	PN
Package raw name	<PKG>_RAWNAME	BPN
Package version	<PKG>_VERSION	PV
Source code dir	@D	S
Build dir	@D	B
Install files in (*)	TARGET_DIR	D
Install images in (*)	BINARIES_DIR	DEPLOYDIR

\* The final dirs in Buildroot, temp dirs in Yocto.

## Adding patches



Buildroot	Yocto
.patch file in package dir	.patch file in recipe subdir (*)
<PKG>_PATCH = <URL>	SRC_URI += <URL>
BR2_GLOBAL_PATCH_DIR tree	Your layer

\* Plus SRC\_URI += "file://foo.patch"  
(and FILESEXTRAPATHS\_prepend = "<DIR>:")

## Overall recipe directory layout

<BUILDR00T>

└ package

└ mypackage

├ Config.in

├ mypackage.mk

├ mypackage.hash

└ 0001-fix-bug.patch

<LAYER>

└ recipes-\*

└ \*

├ myrecipe\_1.0.bb

└ files

└ fix-bug.patch

# Building

---

---

Buildroot

`make [all]`

`make busybox`

`make busybox-configure`

---

`make busybox-reconfigure`

`make clean`

`make busybox-dirclean`

---

Yocto

`bitbake <IMAGE>`

`bitbake busybox`

`bitbake -c configure busybox`

---

`bitbake -C configure busybox`

`bitbake -c clean world`

`bitbake -c clean busybox`

---

# Tuning resource usage



---

## Buildroot

---

`BR2_JLEVEL=2 make`

—

Build options → Enable compiler cache

—

---

## Yocto

---

`PARALLEL_MAKE="-j 2" bitbake ...`

`BB_NUMBER_THREADS=2 bitbake ...`

—

`SSTATE_DIR ?= " /.sstate-cache"`

---

## Buildroot: out-of-tree builds

- `make O=foo foo_defconfig`
- `make O=bar bar_defconfig`
- `cd foo; make`
  - Build in `foo/*` instead of `output/*`





## Buildroot: out-of-tree builds

- `make O=foo foo_defconfig`
- `make O=bar bar_defconfig`
- `cd foo; make`
  - Build in `foo/*` instead of `output/*`
- `cd bar; make`
  - Can run in parallel



- `bitbake core-image-minimal`
- `bitbake my-image-huge`
  - Recycles common artifacts



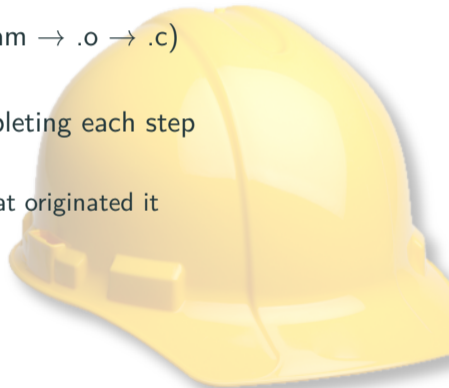
# Yocto: multiple machines and images

- `bitbake core-image-minimal`
- `bitbake my-image-huge`
  - Recycles common artifacts
- `MACHINE=another-board bitbake my-image-huge`
  - Remember to use `?=` to set `MACHINE` in your conf file



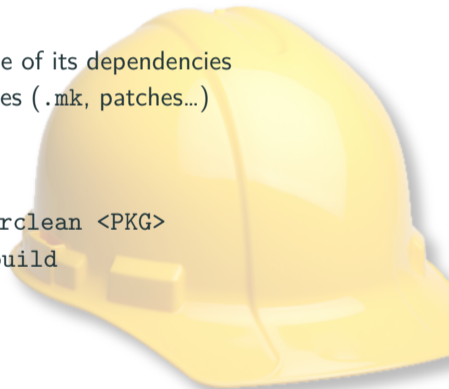
## Buildroot dependency tracking: stamp files

- Dependency tracking is at the core of Make (program → .o → .c)
  - Does not fit completely the needs of a buildsystem
- Internally Buildroot touches a *stamp file* after completing each step
  - An empty file
  - Tracks successful step completion, not the rules that originated it
  - If the rules change, Buildroot is unaware



## Buildroot dependency tracking: stamp files

- You need to manually trigger a rebuild when:
  - You changed the configuration of the package or one of its dependencies
  - You're developing the package and changed the rules (`.mk`, patches...)
- How to rebuild
  - The safe option: `make clean; make`
  - If you know what you really need: `make <PKG>-dirclean <PKG>`
  - Or `make <PKG>-reconfigure / make <PKG>-rebuild`



- Bitbake stores a hash for each task
- Hash content:
  - All the recipe variables and task code (`bitbake -e`)
  - Content of all files stored in `SRC_URI`
- Automatically detect recipe changes and rebuilds what's needed
- Result stores in the *sstate cache* for later reuse



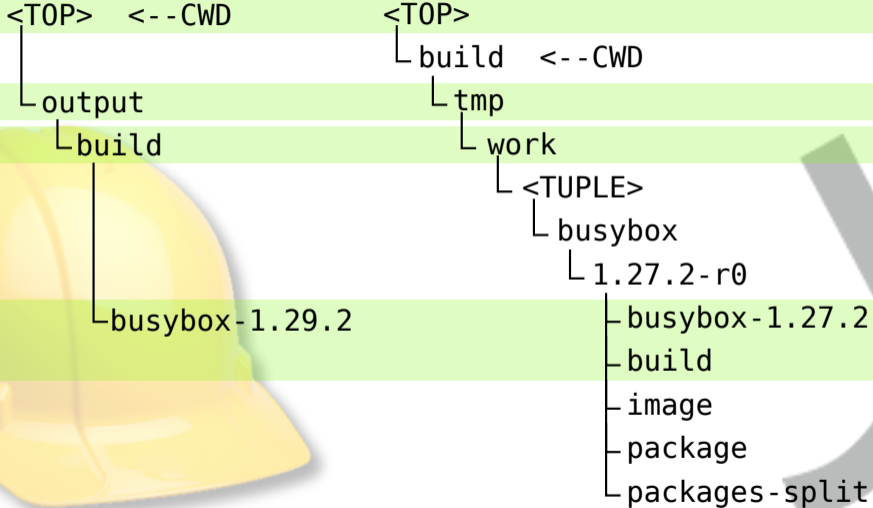
- Still want to force a task?
  - `bitbake -f -c configure <PKG>`
  - `-f` forces to run tasks even when not needed



**Where are my output files?**



## Work directory layout



# Root filesystem generation

<TOP> <-- CWD

└ output

└ target

└ bin, usr...

<TOP>

└ build <-- CWD

└ tmp

└ work

└ <MACHINE TUPLE>

└ core-image-minimal

└ 1.0-r0

└ rootfs

└ bin, usr...

# Output directory layout

```
<TOP>  <-- CWD
```

```
└─ output
```

```
└─ images
```

```
├─ u-boot.*
```

```
├─ *.dtb
```

```
├─ *Image
```

```
├─ rootfs.<EXT>
```

```
└─ sdcard.img
```

```
<TOP>
```

```
└─ build  <-- CWD
```

```
└─ tmp
```

```
└─ deploy
```

```
└─ images
```

```
└─ <MACHINE>
```

```
├─ u.boot.*
```

```
├─ *.dtb
```

```
├─ *Image
```

```
└─ <IMG> - <MACHINE> . <EXT>
```

# Understanding what's going on

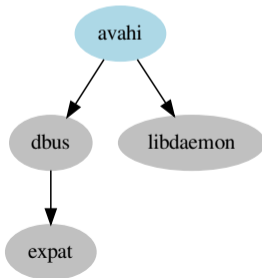
---

**What will it build?**

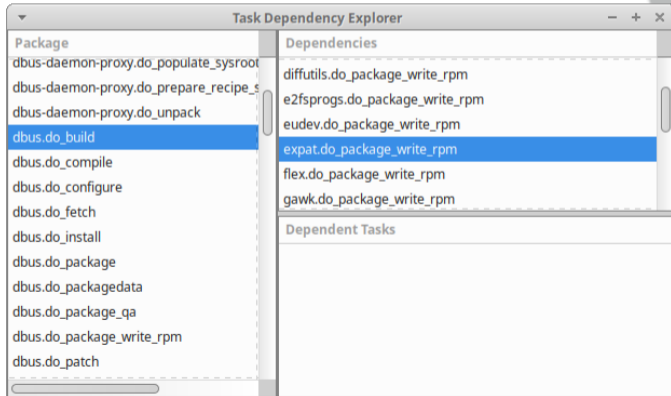


## Buildroot: graph-depends

- Build a per-package graph: `<PKG>-graph-depends`
- Set `BR2_GRAPH_DEPS_OPTS` in the environment to control the output
- `BR2_GRAPH_DEPS_OPTS="--exclude=host" make avahi-graph-depends`
- Produces `output/graphs/avahi-graph-depends.pdf`



- Generating dot graphs not really usable
- Task Explorer: `bitbake -g -u taskexp world`
- Shows dependencies between tasks (not recipes)





**What does it do?**

**What went wrong?**

## Buildroot: default output

```
$ make
...
>>> host-e2fsprogs 1.44.2 Extracting
xzcat /home/murray/src/e2fsprogs/e2fsprogs-1.44.2.tar.xz...
>>> host-e2fsprogs 1.44.2 Patching
>>> host-e2fsprogs 1.44.2 Configuring
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
...
```

- “>>>” marks the started tasks
- The whole output of each step follows
- Failure? Look at the last lines



## Buildroot: concise output

```
$ ./utils/brmake
```

```
...
```

```
2018-10-06T16:15:58 >>> host-zlib Patching
```

```
2018-10-06T16:15:58 >>> host-zlib Configuring
```

```
2018-10-06T16:15:58 >>> host-zlib Building
```

```
2018-10-06T16:15:58 >>> host-zlib Installing to host directory
```

```
2018-10-06T16:15:58 >>> host-util-linux 2.32.1 Patching
```

```
...
```

- Adds step start time
- Verbose output saved in `br.log`



## Yocto: default output

- The default output shows the current status, no logs
- Hides completed tasks

```
Currently 4 running tasks (119 of 2503) 4% |## |
0: glibc-initial-2.27-r0 do_fetch (pid 5216) 38% |##### | 4.09M/s
1: glibc-2.27-r0 do_fetch - 4s (pid 5261)
2: ncurses-native-6.0+20171125-r0 do_fetch (pid 6147) | <=> |
3: elfutils-native-0.170-r0 do_fetch (pid 7143) 11% |### | 2.49M/s
```

## Yocto: concise “log” output

- To see the completed tasks:
- `bitbake ... | cat`

```
NOTE: Running task 119 of 2645 (.../binutils/binutils-cross_2.30.bb:do_unpack)
NOTE: Running task 232 of 2645 (virtual:native:...lzo/lzo_2.10.bb:do_compile)
NOTE: recipe binutils-cross-arm-2.30-r0: task do_unpack: Started
NOTE: recipe binutils-cross-arm-2.30-r0: task do_prepare_recipe_sysroot: Started
NOTE: recipe elfutils-native-0.170-r0: task do_prepare_recipe_sysroot: Started
NOTE: recipe lzo-native-2.10-r0: task do_compile: Started
NOTE: recipe elfutils-native-0.170-r0: task do_prepare_recipe_sysroot: Succeeded
NOTE: Running task 247 of 2645 (virtual:native:.../elfutils_0.170.bb:do_configure)
NOTE: recipe binutils-cross-arm-2.30-r0: task do_prepare_recipe_sysroot: Succeeded
```

- Failure?
- For each task a log file is saved
  - in `tmp/work/<TUPLE>/<RECIPE>/<VERSION>/temp/log.do_<TASK>`
  - e.g. `tmp/work/x86_64-linux/gmp-native/6.1.2-r0/temp/log.do_configure`
- Or re-run the failed task with verbose output to see its output on your terminal
  - `bitbake -v -f -c configure gmp-native`

**What is it thinking?**

## Buildroot: printvars

- `make -s printvars`
  - Print all variables
- `make -s VARS=BUSYBOX_% printvars`
  - Only variables matching a pattern
- `make -qp`
  - Print the whole Make database
  - Variables (before expansion) and the file where they were set
  - Rules (target + prerequisites + actions)





- `bitbake -e`
  - Show the global environment
  - Variables and the files where they were set
- `bitbake -e <RECIPE>`
  - Show the per-recipe environment
  - Variables and the files where they were set
  - Tasks actions



## Customizing the root filesystem

---

- The same configuration system as the kernel, Busybox, U-Boot, Barebox...
- `make menuconfig`, `make xconfig`
- `.config` is your current configuration
- `make savedefconfig` updates your `defconfig` with the new values



- Your “configuration” is in several .bb files.
- A common layout:
  - Build options, toolchain, MACHINE: a conf file in your top layer (or `build/conf/local.conf`)
  - Target options, kernel and bootloader selection: in your layer `conf/machine/<MACHINE>.bb`
  - System configuration: various recipes, other places
  - Packages to put in rootfs: image recipe (see later)



## Buildroot: adding packages

- `make menuconfig` → Packages
  - Search, add, remove, change packages
- `make clean` (if you changed or remove packages)
- `make`



- Find the package you need
  - `bitbake-layers show-recipes`
  - <http://layers.openembedded.org/layerindex/branch/master/layers/>
- Create your own image recipe
  - Image = list of packages to put in rootfs (a subset of all the packages)



## Yocto: adding packages

- Create an image recipe (<MYLAYER>/recipes-\*/images/\*-image-\*.bb)

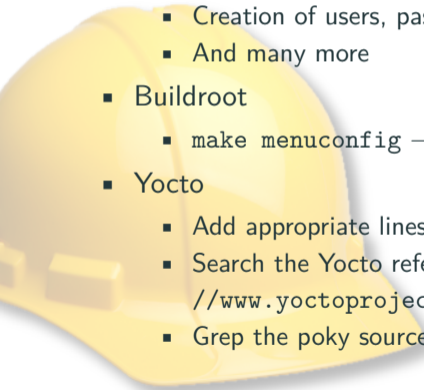
```
require recipes-core/images/core-image-minimal.bb
DESCRIPTION = "My own root filesystem"
LICENSE = "MIT"
IMAGE_FSTYPES = "tar.gz"
IMAGE_INSTALL += "htop packagegroup-debug"
```

- Package groups (<MYLAYER>/recipes-\*/packagegroups/packagegroup-\*.bb)

```
inherit packagegroup
RDEPENDS_${PN} = "gdb strace"
```

# Typical root filesystem customizations

- And embedded systems needs customizations
  - High-level choices: init system, /dev management, locales...
  - Creation of users, passwords, assorted files, ...
  - And many more
- Buildroot
  - `make menuconfig` → System configuration
- Yocto
  - Add appropriate lines to your conf, board or image files
  - Search the Yocto reference manual <https://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html>
  - Grep the poky source code





## Some system settings, side-by-side

Buildroot	Yocto
System hostname	<code>hostname_pn-base-files = "mybox"</code>
System banner	<code>DISTRO_NAME_pn-base-files = "Welcome", DISTRO_VERSION_pn-base-files = ""</code>
Init system	<code>VIRTUAL-RUNTIME_init_manager</code>
/dev management	<code>VIRTUAL-RUNTIME_dev_manager</code>
Root password	<code>IMAGE_FEATURES += "empty-root-password"</code>
Users tables	<code>inherit extrausers; EXTRA_USERS_PARAMS = "usermod -P 1876*18 root;"</code>

## Other rootfs customizations

- Buildroot: System configuration menu:
  - Root filesystem overlay directories
  - Post-build and post-image scripts
- Yocto
  - `ROOTFS_POSTPROCESS_COMMAND` and `IMAGE_POSTPROCESS_COMMAND`

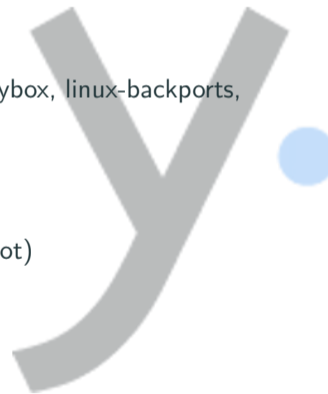
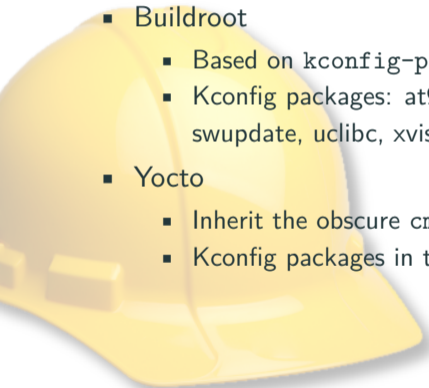


## **Tweaking recipes**

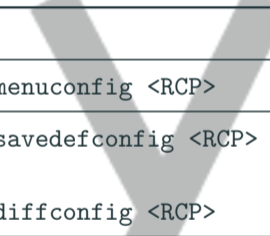
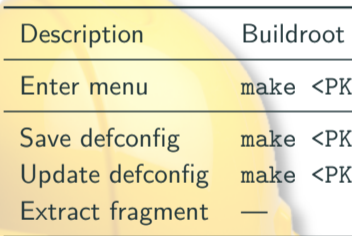
---

# Configuring Kconfig-based packages

- Buildroot
  - Based on kconfig-package
  - Kconfig packages: at91bootstrap3, barebox, uboot, linux, busybox, linux-backports, swupdate, uclibc, xvisor
- Yocto
  - Inherit the obscure cm11 class
  - Kconfig packages in the Poky layer: linux, busybox (not U-Boot)



## Configuring Kconfig-based packages



Description	Buildroot	Yocto
Enter menu	<code>make &lt;PKG&gt;-menuconfig</code>	<code>bitbake -c menuconfig &lt;RCP&gt;</code>
Save defconfig	<code>make &lt;PKG&gt;-savedefconfig</code>	<code>bitbake -c savedefconfig &lt;RCP&gt;</code>
Update defconfig	<code>make &lt;PKG&gt;-update-defconfig</code>	—
Extract fragment	—	<code>bitbake -c diffconfig &lt;RCP&gt;</code>

# Yocto: (re)assigning variables

- Assignments
  - `F := "foo-${A}"` — Immediate expansion
  - `F = "foo-${A}"` — Expansion on usage
- Weak assignments: used for values the user is supposed to customize
  - Base layer `.bb: VAR ??= "white"`
  - Middle layer `.bbappend: VAR ?= "black"`
  - Top-level layer `.bbappend: VAR = "green"`
  - The recipe will use `VAR = "green"`
- Append or prepend
  - `VAR += "val", VAR =+ "val"` (adds spaces)
  - `VAR_append = "val", VAR_prepend = "val"` (does not add spaces)
  - `VAR_remove = "val"`

## Buildroot: (re)assigning variables

- It's a Makefile, use the Make syntax
- Assignments
  - `F := "foo-$(VER)"` — Immediate expansion
  - `F = "foo-$(VER)"` — Expansion on usage
- Append or prepend
  - `VAR = "$(VAR) extra", VAR = "extra $(VAR)"`



# More string processing

- Buildroot
  - Make has several functions for transforming text
  - Example: `VAR = $(filter-out bug, foo bug bar)`
- Yocto
  - If Bitbake is not enough, use Python
  - `PV_x = "${@'.'.join('${PV}'.split('.')[0:2] + ['x'])}"`  
"10.11.12" → "10.11.x"





# Yocto: changing task code

```
do_conf_append() {  
    echo CONFIG_ACS >>${D}/.config  
}  
  
do_install_prepend() {  
    mkdir -p ${D}${bindir}  
}
```

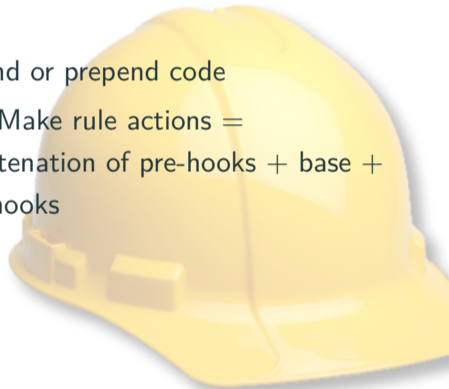
- Append or prepend code
- Final task code = concatenation of prepends + base + appends
  - Don't mix Bash and Python

## Buildroot: changing task code

```
define FOO_ENABLE_ACS
    echo CONFIG_ACS >>$(@D)/.config
endef
FOO_POST_CONFIGURE_HOOKS += FOO_ENABLE_ACS

define FOO_CREATE_BIN_DIR
    mkdir -p $(TARGET_DIR)/bin
endef
FOO_PRE_INSTALL_HOOKS += FOO_CREATE_BIN_DIR
```

- Append or prepend code
- Final Make rule actions = concatenation of pre-hooks + base + post-hooks



## Conclusions

---

# Questions?

Ask now...

...or during my Office Hour  
Wednesday, October 24  
from 10:45 to 11:45  
Level -2 Built-In Seating  
(near Lennox)

Thank you for your attention!

Luca Ceresoli

luca@lucaceresoli.net

<http://lucaceresoli.net>

© Copyright 2018, Luca Ceresoli

Slides released under

Creative Commons Attribution - Share Alike 3.0 License

<https://creativecommons.org/licenses/by-sa/3.0/>

## Extra slides

---

## Working with local sources

- Use sources from a local directory
  - Not managed by the build system
  - Useful during application development
- Buildroot
  - `<PKG>_OVERRIDE_SRCDIR=/my/src/tree make`
  - Skips source, extract, patch
  - rsyncs from `/my/src/tree` before building
- Yocto
  - `inherit externalsrc`
  - `EXTERNALSRC = "/my/src/tree"`
  - `fetch, unpack, patch`
  - Points `S` to `/my/src/tree`