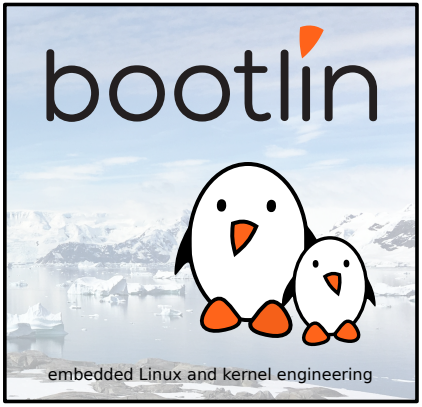
10 best practices for Yocto

Jérémie Dautheribes
jeremie.dautheribes@bootlin.com



Jérémie Dautheribes

- Embedded Linux engineer at Bootlin
  - Yocto **expertise**: i.MX6/7/8, Nvidia Jetson Nano, TI AM62x, Xilinx ZynqMP, ...
  - Development, consulting and **training**
  - Strong open-source focus
- Living in **Toulouse**, France

What is this talk about?

What is this talk about?

- Yocto: tool kit for building custom Linux images for embedded systems
- De facto industry standard
- Very powerful and (too?) flexible -> Often used badly
- Hard to fully master
- Feedback from my experience and colleagues'

Keep your layers updated

Keep your layers updated

- Always use a supported Yocto release!
- Usual workflow: use a branch named after a supported Yocto release
- Update your layer to newer minor version on a regular basis
- Use Long Term Support (LTS) versions if you need stability (supported 4 years)

Keep your layers updated

Codename	Yocto Project Version	Release Date	Current Version	Support Level	Poky Version	BitBake branch	Maintainer
Walnascar (aka Walna)	5.2	April 2025		Future	N/A	2.12	Richard Purdie <richard.purdie@linuxfoundation.org>
Styhead (like 'try head')	5.1	October 2024	5.1 (October 2024)	Support for 7 months (until May 2025)	N/A	2.10	Steve Sakoman <steve@sakoman.com>
Scarthgap	5.0	April 2024	5.0.5 (November 2024)	Long Term Support (until April 2028)	N/A	2.8	Steve Sakoman <steve@sakoman.com>
Nanbield (like 'man field')	4.3	November 2023	4.3.4 (April 2024)	EOL	N/A	2.6	Steve Sakoman <steve@sakoman.com>
Mickledore	4.2	May 2023	4.2.4 (December 2023)	EOL	N/A	2.4	Steve Sakoman <steve@sakoman.com>
Langdale	4.1	October 2022	4.1.4 (May 2023)	EOL	N/A	2.2	Steve Sakoman <steve@sakoman.com>
Kirkstone (like 'kirk stun')	4.0	May 2022	4.0.23 (November 2024)	Long Term Support (Apr. 2026[1])	N/A	2.0	Steve Sakoman <steve@sakoman.com>
Honister	3.4	October 2021	3.4.4 (May 2022)	EOL	N/A	1.52	Anuj Mittal <anuj.mittal@intel.com>
Hardknott	3.3	April 2021	3.3.6 (April 2022)	EOL	N/A	1.50	Anuj Mittal <anuj.mittal@intel.com>
Gatesgarth	3.2	Oct 2020	3.2.4 (May 2021)	EOL	N/A	1.48	Anuj Mittal <anuj.mittal@intel.com>
Dunfell	3.1	April 2020	3.1.33 (May 2024)	EOL - LTS[1]	23.0	1.46	Steve Sakoman <steve@sakoman.com>

Screenshot from https://wiki.yoctoproject.org/wiki/Releases

Don't overuse local.conf

Don't overuse local.conf

- Useful for development and quick testing
- Should be specific to your **local build**: thread limitations, network configuration, build log management
- Options you want to share should be moved to **distro** and **machine** configuration files or to **image** recipes
- For company specific needs: `site.conf`
- Possibility to use a template `local.conf.sample` (see advice 4)

Don't use Poky in production

Don't use Poky in production

- Poky:
  - Reference Yocto distribution
  - Meant for learning, testing and optionally early development

> **Note**
>
> While Poky is a "complete" distribution specification and is tested and put through QA, you cannot use it as a product "out of the box" in its current form.

- Create your own distro:
  - Very easy to create in `<layer>/conf/distro/<distro>.conf`
  - Distribute changes from `local.conf`
  - Configure options that impact your global build: toolchain, libc implementation, init system, Wayland compositor
  - Can contain specific classes (ex: signing image)
  - Can provide sample files: `bblayers.conf.sample`, `local.conf.sample`

Minimal distro example

In <layer>/conf/distro/myDistro.conf:

```
DISTRO = "myDistro"
DISTRO_NAME = "My Wonderful Linux Distribution"
DISTRO_VERSION = "1.0"

# Add basic features
DISTRO_FEATURES = "acl alsa argp ipv4 ipv6 largefile xattr vfat"

# Add specific features
DISTRO_FEATURES:append = " rauc"

DEFAULT_TIMEZONE = "Europe/Paris"

# Default settings
# TCLIBC = "glibc"
# TCMODE = "default" (i.e. gcc)
# VIRTUAL-RUNTIME_init_manager = "sysvinit"
```

More distro example

In <layer>/conf/distro/myDistro.conf:

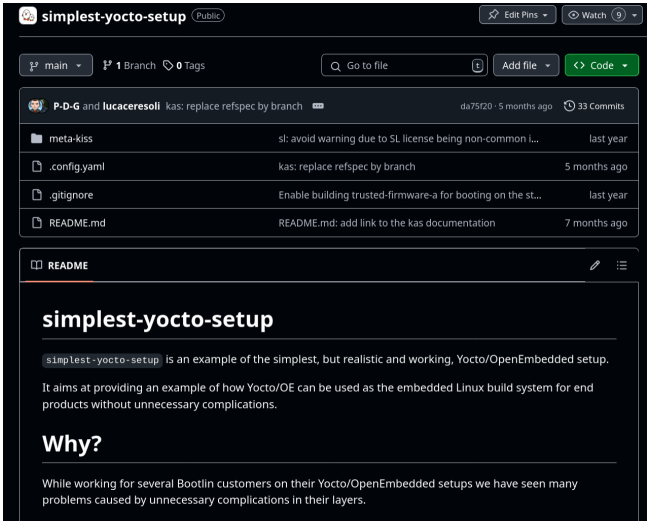
```
DISTRO = "myDistro"
DISTRO_NAME = "My Powerful Linux Distribution"
DISTRO_VERSION = "1.0"

DISTRO_FEATURES = "acl pam polkit rauc seccomp systemd usrmerge xattr \
    ${@bb.utils.contains('HOST_ARCH', 'aarch64', 'selinux', '', d)} \
    ${@bb.utils.contains('HOST_ARCH', 'x86_64', 'selinux', '', d)} \
"

TCLIBC = "musl"
VIRTUAL-RUNTIME_init_manager = "systemd"
SDK_NAME = "${DISTRO}-${TCLIBC}-${SDK_ARCH}-${IMAGE_BASENAME}-${TUNE_PKGARCH}"
INHERIT += "create-spdx"
ACCEPT_FSL_EULA = "1"
```

Select third-party layers carefully

Select third-party layers carefully

- Keep your number of layer low (as much as you can)
- Quality of various third-party layers is dubious
- BSP layers: quality of SoC vendor layers is varying, for SoM vendor layers it's questionable
- If supported in mainline, you can even drop BSP third-party layers
- Use third-party layers for standard complex stacks: meta-qt6, meta-flutter, ...
- Always estimate benefit/cost ratio

Example of a simple layer



https://github.com/bootlin/simplest-yocto-setup/

Manage carefully your layer(s)

Manage carefully your custom layer(s)

- Important to understand what should go in `image`, `DISTRO` and `MACHINE` files
- Fine to have only 1 layer if correctly managed (small team?)
- If required split it in at least 3 layers: BSP, distro, custom apps
- Might be more flexible and easier to maintain depending on your environment

Don't put source code or binary inside your layers

- Don't use your layers as source-control management for your packages
- Yocto is a big wrapper around existing projects
- Meant to build everything from scratch
- Might be fine to have some configuration files
- Avoid pre-compiled binaries, if you can't (firmwares, proprietary librairies, ...):
  - Write recipes fetching those binaries from remote locations
  - Use `bin_package.bbclass`

Leverage existing Yocto functionalities

Leverage existing Yocto functionalities

- Yocto provides a lot of features, use them!
- Non-exhaustive list:
  - bbclass: autotools/cmake/meson, kernel, fitImage, ...
  - Bootloader: u-boot includes
  - `DISTRO_FEATURES`, `MACHINE_FEATURES`, `COMBINED_FEATURES`
  - `IMAGE_FEATURES`
  - Overriding mechanism, `FILESPATH`, `PACKAGECONFIG`

bbclass example

Example: `fatresize` recipe

```
SUMMARY = "Resize FAT partitions using libparted"
SECTION = "console/tools"
LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM = "file://COPYING;md5=d32239bcb673463ab874e80d47fae504"

SRC_URI = "git://salsa.debian.org/parted-team/fatresize.git;protocol=https;branch=master \
           file://0001-build-Do-not-build-.sgml-file.patch \
           file://0001-configure-Do-not-add-D_FILE_OFFSET_BITS-to-CFLAGS.patch \
          "
SRCREV = "12da22087de2ec43f0fe5af1237389e94619c483"

S = "${WORKDIR}/git"

DEPENDS = "parted"

inherit autotools pkgconfig
```

Override mechanism

```
OVERRIDES="arm:armv7a:ti-soc:ti33x:beaglebone:poky"

KERNEL_DEVICETREE:beaglebone = "am335x-bone.dtb" # This is applied
KERNEL_DEVICETREE:dra7xx-evm = "dra7-evm.dtb"    # This is ignored
```

Note: `OVERRIDES` is automatically generated, but can be easily customized

Be careful with variable scope

Be careful with variable scope

- **Global scope**: variables defined in global configuration files (distro, machine, local.conf, ...), will impact the build globally
- **Local scope**: the rest, i.e. variables in recipes, include files, bbclasses
- Bitbake provides a set of common variables (`PV`, `S`, `FILEPATHS`, ...), each recipe will have its **own copy**
- .bbclass and images are "just" recipes, i.e. local scope (except for classes inherited globally)
- Avoid using operators (+=, .=, ...) within **global configuration files** due to parsing order, **prefer overridings** (:append, :prepend, ...) which provide much more reliable results

Tip: use `bitbake-getvar` tool and `buildhistory` feature

bitbake-getvar

```
$ bitbake-getvar PREFERRED_PROVIDER_virtual/kernel
NOTE: Starting bitbake server...
#
# $PREFERRED_PROVIDER_virtual/kernel [2 operations]
#   set /home/jd/sources/meta-st-stm32mp/conf/machine/include/st-machine-providers-stm32mp.inc:4
#     "linux-stm32mp"
#   override[bootlinlabs]:set /home/jd/build/conf/local.conf:1
#     "linux-dummy"
# pre-expansion value:
#   "linux-dummy"
PREFERRED_PROVIDER_virtual/kernel="linux-dummy"
```

buildhistory

- See official doc for how to configure it
- Example when removing `htop` package from `IMAGE_INSTALL`:

```
# Inside build/buildhistory
$ git show

diff --git a/images/bootlinlabs/glibc/bootlinlabs-image-minimal/image-info.txt b/images/...
index 911e216..070d52e 100644
--- a/images/bootlinlabs/glibc/bootlinlabs-image-minimal/image-info.txt
+++ b/images/bootlinlabs/glibc/bootlinlabs-image-minimal/image-info.txt
...
-IMAGE_INSTALL = packagegroup-core-boot packagegroup-bootlinlabs-games htop
+IMAGE_INSTALL = packagegroup-core-boot packagegroup-bootlinlabs-games
-IMAGESIZE = 15868
+IMAGESIZE = 15392
...
```

buildhistory

```
...
diff --git a/images/bootlinlabs/glibc/bootlinlabs-image-minimal/files-in-image.txt b/images/...
index 9bf03e7..eab4e8d 100644
--- a/images/bootlinlabs/glibc/bootlinlabs-image-minimal/files-in-image.txt
+++ b/images/bootlinlabs/glibc/bootlinlabs-image-minimal/files-in-image.txt

...

--rw-r--r-- root       root             1647 ./etc/ld.so.cache
+-rw-r--r-- root       root             1448 ./etc/ld.so.cache
--rwxr-xr-x root       root           184872 ./usr/bin/htop
--rwxr-xr-x root       root           128252 ./lib/libncursesw.so.5.9
-lrwxrwxrwx root       root               18 ./lib/libncursesw.so.5 -> libncursesw.so.5.9

...
```

Use KAS

Use KAS

- Tool developped by Siemens for automating setting up Yocto and build images
- Very easy to use, especially for non-Yocto people
- Support containers out of the box
- Note: bypass `template` mechanism, overwrite `local.conf` and `bblayers.conf`

KAS configuration example:

```
header:
  version: 8
machine: mymachine
distro: mydistro
target:
  - myimage

repos:
  meta-custom:

  bitbake:
    url: "https://git.openembedded.org/bitbake"
    # tag 2.0
    commit: c212b0f3b542efa19f15782421196b7f4b64b0b9
    layers:
      .: excluded

  openembedded-core:
    url: "https://git.openembedded.org/openembedded-core"
    branch: kirkstone
    layers:
      meta:
```

KAS configuration example

```
meta-freescale:
  url: "https://github.com/Freescale/meta-freescale"
  branch: kirkstone

meta-openembedded:
 url: https://git.openembedded.org/meta-openembedded
 branch: kirkstone
 layers:
   meta-oe:
   meta-python:
   meta-networking:

 local_conf_header:
  common-conf: |
   RM_OLD_IMAGE = "1"
   BB_NO_NETWORK = "1"
```

- Build in a single command:
  `$ kas build meta-custom/mymachine.yaml`
- Or build inside a container:
  `$ kas-container build meta-custom/mymachine.yaml`

Work with the community

Work with the community

- Read the official documentation
- Contribute to the doc (maintained by Bootlin), follow the contributor guide
- Niche community, very friendly and easy to talk to
- Mailing list based upstreaming workflow
- Third-party layers: usually through Git web interfaces (Github, Gitlab)

Questions? Suggestions? Comments?

Jérémie Dautheribes
jeremie.dautheribes@bootlin.com

Bootlin "Yocto Project and OpenEmbedded development" training
https://bootlin.com/training/yocto/