Yocto Project and OpenEmbedded Training Course

Training course

Michael Opdenacker

Root Commit

June 9, 2025

Introduction

License of these training materials

Attribution-ShareAlike 4.0 International

You are free to:

- Share — copy and redistribute the material in any medium or format for any purpose, even commercially.
- Adapt — remix, transform, and build upon the material for any purpose, even commercially.
- The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- Attribution — You must give appropriate credit , provide a link to the license, and indicate if changes were made . You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
- No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

https://creativecommons.org/licenses/by-sa/4.0/

Sources: `https://gitlab.com/rootcommit/training-materials/`

Credits

Swaminathan K, Johannes Zink, Martin Herren

You can help making this course better and add your name to the above list by sending suggestions, testing the instructions and reporting typos and bugs.

Michael Opdenacker

Embedded Linux consultant and trainer

- https://rootcommit.com/about/michael-opdenacker/
- Former founder of Bootlin
- New founder of Root Commit
- Offering embedded Linux training courses with a focus on practical activities, interactivity and learning techniques. https://rootcommit.com/training/
- Free Software enthusiast and advocate (member of April.org)

Yocto Project and OpenEmbedded Experience

root commit

- First used it in 2004 — Very close to the beginning
- Conducted several customer projects
- Shared experience through multiple technical presentations. See videos too.
- Yocto Project trainer since 2023
- 2021–2024: Official documentation maintainer for the Yocto Project
- Contributions to BitBake, Openembedded Core and Meta Openembedded.
- Contributor to the Yocto Project advocacy group — Try to find me on `https://www.yoctoproject.org/about/project-overview/`!

Introduction

First demo

BeagleBone Black Demo

- BeagleBone Black board from BeagleBoard.org
- ARM32 Cortex A8 CPU, the only board officially supported by Yocto at the moment
- Using Yocto Styhead 5.1 (latest stable)

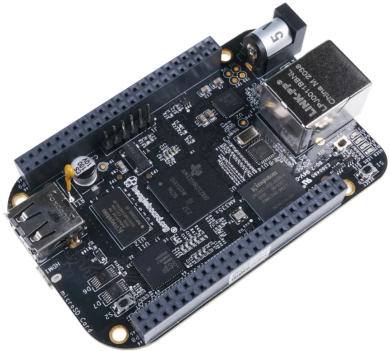


Image credits: BeagleBoard.org

Introduction

Learning Techniques

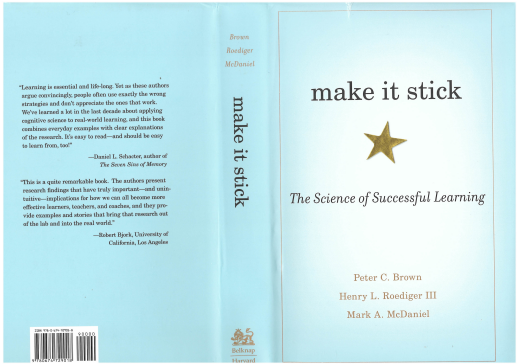
Experience from 20+ years of training

root commit

- Need challenging labs (good)
- But too long series of lectures, people passive for too long
- And tendency to explain the whole theory (as *exhaustively* as possible), before letting people experiment
- And quiz only at the end
- You forget quickly if you stop using

What I'm trying to do

- More practical lab time
- Challenging labs
  - You don't learn from labs that are too easy
  - If you don't know how to do something, it's often because you missed something in the lectures
  - Making mistakes is very positive: that's how you build experience and correct misconceptions
  - And the instructor is here to avoid staying stuck for too long
- Online sessions: people do their labs instead of just watching demos
- More interaction with the audience
- Lectures should never exceed 30 minutes (except if many questions)
- More self-testing (quizzes)



https://rootcommit.com/2024/make-it-stick-book-review/

Lab — Preparing Your Environment

- Check PC requirements
- Check your GNU/Linux distribution
- Download your lab data

Introduction

Embedded Linux

Question

What's common between...

Thermoplan coffee machine

BMW In Vehicle Infotainment

Ikea Dirigera smart home hub

Stream Unlimited audio hardware modules

Comcast set top boxes

FLIR C5 infrared camera

Answer

root commit

- They are all using Embedded Linux

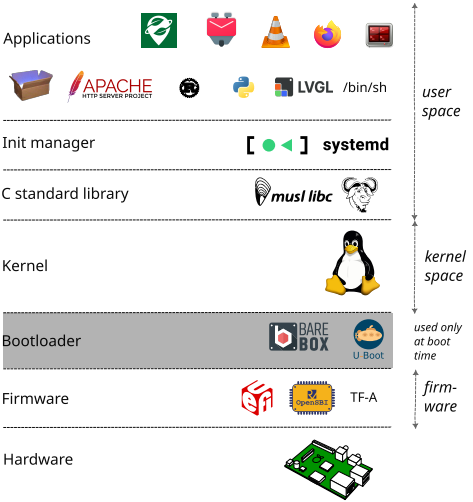
Answer

root commit

- They are all using Embedded Linux
- And their system image was built by Yocto

Answer

- They are all using Embedded Linux
- And their system image was built by Yocto
- Like tens of thousands of other devices `https://wiki.yoctoproject.org/wiki/Project_Users`



Differences with regular Linux systems

Regular "GNU/Linux" systems

- Runs on a high-end processor (x86 or ARM)
- High RAM and disk space requirements, expensive hardware
- Standard distribution, maintained by the distro vendor (Red Hat, Debian...)
- Standard and versatile software stack
- Linux kernel with a standard configuration, supported by the distro vendor, close to mainline.
- Very frequent security updates, managed by the distro vendor.

Embedded Linux systems

- Runs on a less powerful, cheaper processors (mostly ARM and RISC-V)
- Low RAM and disk space requirements, cheaper hardware
- Most often, custom root filesystem, built and maintained independently
- Most often, dedicated software stack ("just include what you need")
- Linux kernel with a custom configuration, often maintained by the hardware vendor and with custom changes.
- Less frequent or no security updates, managed by the system vendor.

How the system is built

Manually

- Build everything from source
- Great for learning (!), manageable for very simple or demo systems
- But not reproducible (!)

Copyright: Dargaud (Lucky Luke)

Using automated tools

- Buildroot
- Yocto Project

Most popular solution!

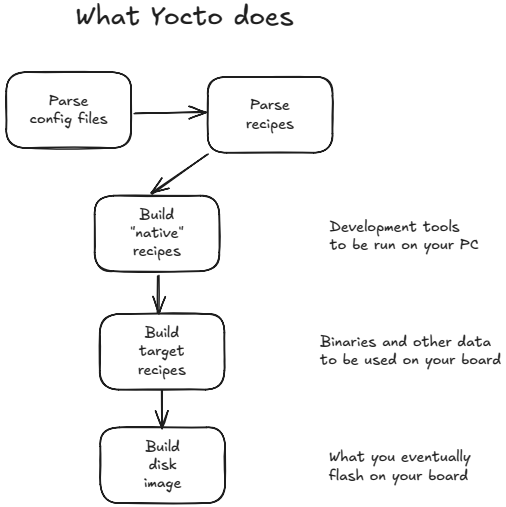
Using a binary distribution

- Regular desktop/server distributions: **Debian**, Ubuntu, Fedora, OpenSUSE
- Embedded friendly distributions: Alpine

Yocto in a Nutshell

Main components

- **BitBake**: task scheduler
- **Recipes**: how to build specific components from source
- **Layers**: collections of recipes

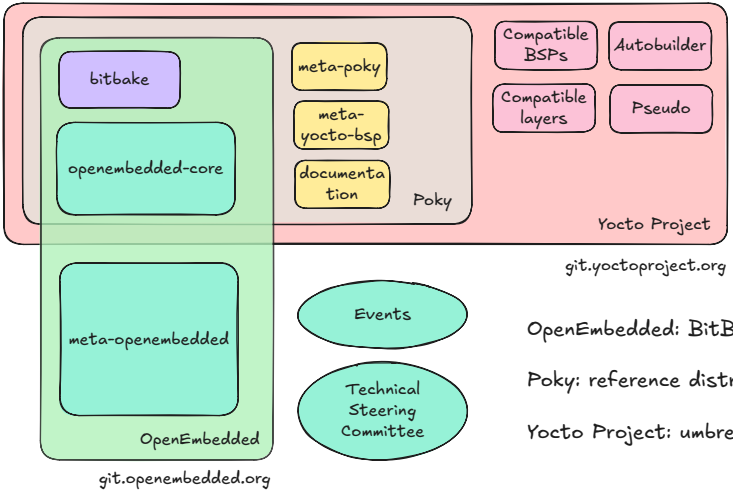


Yocto History

root commit

- 2003: creation of OpenEmbedded, merging the efforts of OpenZaurus, Familiar Linux and OpenSIMpad.
- 2010: creation of the Yocto Project by the Linux Foundation (project number 2!), with BitBake and core OpenEmbedded recipes as foundations. A lot of money invested from the LF and project members (development, support, documentation, events...).
- 2020: First Long Term Support release.
- Today: You're investing in your Yocto skills!

Yocto sub-projects and terminology

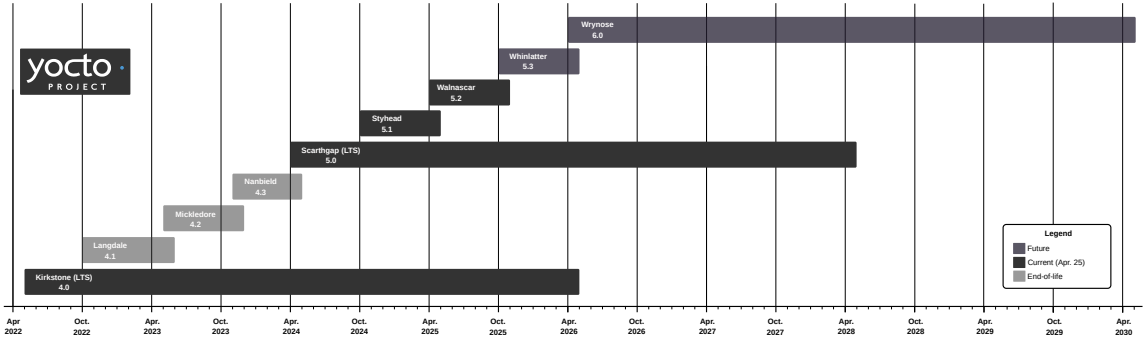


OpenEmbedded: BitBake scheduler + standard recipes + tools

Poky: reference distribution

Yocto Project: umbrella project

Yocto releases

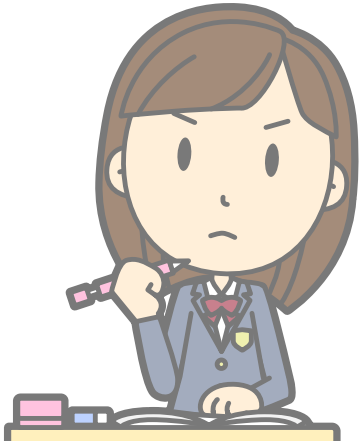


Source: `https://git.yoctoproject.org/yocto-docs/tree/documentation/ref-manual/svg/releases.svg`

Lab — First Yocto Build

- Get the source code
- Setup the environment
- Build your first image, for the BeaglePlay board.

Quiz — Embedded Linux Introduction Quiz

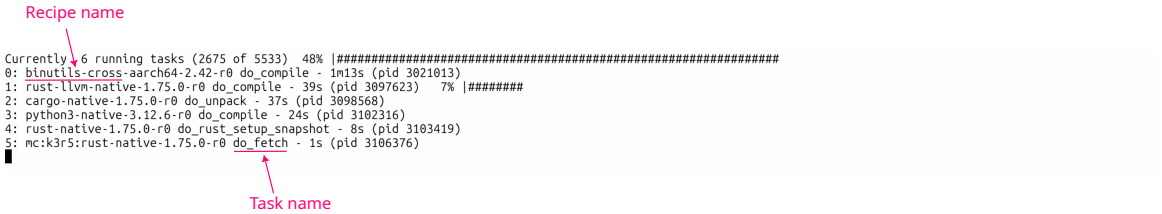


→Click here←

Introduction

Learning from BitBake output

Lab observations: recipes and tasks



What to understand:

- Each component to build (program, library, kernel, image) is represented by a **recipe**.
- Each recipe describes a set of **tasks**: fetch, unpack, compile, install...
- BitBake **manages build dependencies at the task level**, not at the recipe level.

Lab observations: versions and revisions

root commit

```
Currently  8 running tasks (0 of 5533)   0% |
0: glibc-locale-2.39+git-r0 do_create_runtime_spdx_setscene - 0s (pid 2949294)
1: libseccomp-2.5.5-r0 do_populate_lic_setscene - 0s (pid 2950204)
2: m4-1.4.1-r0 do_populate_lic_setscene - 0s (pid 2950239)
3: libcap-ng-0.8.5-r0 do_populate_lic_setscene - 0s (pid 2950266)
4: make-4.4.1-r0 do_package_write_rpm_setscene - 0s (pid 2950268)
5: expat-native-2.6.4-r0 do_create_runtime_spdx_setscene - 0s (pid 2950205)
6: update-rc.d-0.8+git-r0 do_create_runtime_spdx_setscene - 0s (pid 2950347)
7: grep-3.11-r0 do_package_qa_setscene - 0s (pid 2950310)
```

package version

package revision

Recipes are deployed through **packages** (see `tmp/deploy/[rpm|ipk|deb]`)

- Packages allow to split components into several parts: binaries, binaries with debug info, headers, data, documentation... We can just install what we need.
- Packages can be removed too, useful to have the ability to remove no-longer needed stuff without regenerating everything (hello Buildroot 😜)
- Each package has a **version**, which coincides with the recipe version.
- Each package has a **revision**, typically useful to generate different binaries from the same sources (different configuration, patches applied)

Lab observations: native recipes

"native" recipe

```
Currently  8 running tasks (0 of 5533)   0% |
0: glibc-locale-2.39+git-r0 do_create_runtime_spdx_setscene - 0s (pid 2949294)
1: libseccomp-2.5.5-r0 do_populate_lic_setscene - 0s (pid 2950204)
2: make-4.4.1-r0 do_populate_lic_setscene - 0s (pid 2950239)
3: libcap-ng-0.8.5-r0 do_populate_lic_setscene - 0s (pid 2950266)
4: make-4.4.1-r0 do_package_write_rpm_setscene - 0s (pid 2950268)
5: expat-native-2.6.4-r0 do_create_runtime_spdx_setscene - 0s (pid 2950205)
6: update-rc.d-0.8+git-r0 do_create_runtime_spdx_setscene - 0s (pid 2950347)
7: grep-3.11-r0 do_package_qa_setscene - 0s (pid 2950310)
```

Some recipes have the `-native` suffix:

- **native** recipes generate code for the **host** machine (your PC running Yocto)
- Yocto prefers to build the tools it needs by itself rather than rely on distribution provided ones. This brings reproducibility!
- Examples: compilers, CMake, QEMU...
- Exceptions: Python (needed to run BitBake), git, gcc (needed to build the native gcc)...

Lab observations: setscene tasks

```
Sstate summary: Wanted 2406 Local 2403 Mirrors 0 Missed 3 Current 0 (99% match, 0% complete)#################################################################
Initialising tasks: 100% |#####################################################################################################################################################
NOTE: Executing Tasks
Setscene tasks: 570 of 2406
Currently  8 running tasks (0 of 5533)   0% |
0: glibc-locale-2.39+git-r0 do_create_runtime_spdx_setscene - 0s (pid 2949294)
1: libseccomp-2.5.5-r0 do_populate_lic_setscene - 0s (pid 2950204)
2: make-4.4.1-r0 do_populate_lic_setscene - 0s (pid 2950239)
3: libcap-ng-0.8.5-r0 do_populate_lic_setscene - 0s (pid 2950266)
4: make-4.4.1-r0 do_package_write_rpm_setscene - 0s (pid 2950268)
5: expat-native-2.6.4-r0 do_create_runtime_spdx_setscene - 0s (pid 2950205)
6: update-rc.d-0.8+git-r0 do_create_runtime_spdx_setscene - 0s (pid 2950347)
7: grep-3.11-r0 do_package_qa_setscene - 0s (pid 2950310)
```

Some tasks have the `_setscene` suffix:

- They correspond to tasks which were already built before in the same conditions (same input).
- To save time, their output is retrieved from the **Shared State Cache** ("Sstate").

Lab observations: did you notice?

```
Currently  6 running tasks (2675 of 5533)  48% |###################################################################
0: binutils-cross-aarch64-2.42-r0 do_compile - 1m13s (pid 3021013)
1: rust-llvm-native-1.75.0-r0 do_compile - 39s (pid 3097623)   7% |########
2: cargo-native-1.75.0-r0 do_unpack - 37s (pid 3098568)
3: python3-native-3.12.6-r0 do_compile - 24s (pid 3102316)
4: rust-native-1.75.0-r0 do_rust_setup_snapshot - 8s (pid 3103419)
5: mc:k3r5:rust-native-1.75.0-r0 do_fetch - 1s (pid 3106376)
```

Additional configuration

Here, we have a special task prefixed by `mc:k3r5`

- `mc` means **multiconfig**
- In the case of the BeaglePlay, multiconfig means generating two images for the board
  - The default configuration to generate the image for the Cortex-A53 cores in the AM625 SoC.
  - The `k3r5` configuration to generate another image for the Cortex-R5 processor in AM625.

Lab observations: build output

```
Build Configuration (mc:default):
BB_VERSION           = "2.8.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-24.04"
TARGET_SYS           = "aarch64-poky-linux"
MACHINE              = "beagleplay-tl"
DISTRO               = "poky"
DISTRO_VERSION       = "5.0.8"
TUNE_FEATURES        = "aarch64"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp       = "scarthgap:8f74fa4073d4b2ba8e0d9559aa654f3cafcf373a"
meta-arm-toolchain
meta-arm             = "scarthgap:3cadb81ffaa9f03b92e302843cb22a9cd41df34b"
meta-ti-bsp          = "scarthgap:05609ed6e6441c5549496e31b6a28da3a105a7bf"


Build Configuration:
BB_VERSION           = "2.8.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-24.04"
TARGET_SYS           = "arm-poky-eabi"
MACHINE              = "beagleplay-tl-k3r5"
DISTRO               = "poky"
DISTRO_VERSION       = "5.0.8"
TUNE_FEATURES        = "arm armv7a vfp thumb callconvention-hard"
TARGET_FPU           = "hard"
meta
meta-poky
meta-yocto-bsp       = "scarthgap:8f74fa4073d4b2ba8e0d9559aa654f3cafcf373a"
meta-arm-toolchain
meta-arm             = "scarthgap:3cadb81ffaa9f03b92e302843cb22a9cd41df34b"
meta-ti-bsp          = "scarthgap:05609ed6e6441c5549496e31b6a28da3a105a7bf"

Sstate summary: Wanted 2406 Local 2403 Mirrors 0 Missed 3 Current 0 (99% match, 0% complete)#####################################################
Initialising tasks: 100% |#########################################################################################################################
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 5533 tasks of which 5520 didn't need to be rerun and all succeeded.
```

arm 64 bit main CPU

arm 32 bit Cortex-R5

Configurations are also shown in the build output.

Lab observations: key takeaways

- BitBake processes **recipes** to build components
- Recipes describe multiple **tasks**: fetching sources, configuring, compiling, installing
- BitBake schedules tasks, managing dependencies at task level.
- BitBake also builds **native recipes**: tools for your build machine.
- **Setscene** tasks: tasks which output can be retrieved from the **Shared State Cache**.
- Components are deployed through **packages**.
- With BitBake, you can generate multiple images at once (**multiconfig** capability).

Lab — First Image Boot

- Prepare your board
- Flash your new image
- Boot the board

Getting Started

Getting Started

Basic Variable Syntax and Operations

BitBake variables

Two types of variables:

- **Global variables**:
  Defined in configuration files (`.conf`)
- **Local variables**:
  Defined in recipe files.
  Can also be accessed from configuration files.
  Recipes can also access global variables.

- Variables names are uppercase names by convention. Example: `KERNEL_LOCALVERSION`
- Variable values are strings. Example: `ZSTD_LEGACY_SUPPORT ??= "4"`

All official Yocto variables: `https://docs.yoctoproject.org/genindex.html`

Variable assignments

Basic variable assignment:

```
VAR = "value"
```

To set an empty string value:

```
VAR = ""
```

To set a multi-line value:

```
VAR = " Yocto \
        Is like \
        A layered cake"
```

Note: \n is not interpreted as a newline.

Single quotes are also allowed but not common:

```
VAR = 'value'
```

Except for specifying a value with double quotes:

```
VAR = 'I have a " in my value'
```

Checking variable value — bitbake-getvar (1)

root commit

Plain command for global variables:

```
$ bitbake-getvar MACHINE
NOTE: Starting bitbake server...
#
# $MACHINE [3 operations]
#   set /home/mike/yocto-labs/poky/build/conf/local.conf:40
#     [_defaultval] "qemux86-64"
#   set /home/mike/yocto-labs/poky/build/conf/local.conf:290
#     "beagleplay-ti"
#   set /home/mike/yocto-labs/poky/meta/conf/documentation.conf:280
#     [doc] "Specifies the target device for which the image is built. You define MACHINE in the conf/local.conf file in the Build Directory
# pre-expansion value:
#   "beagleplay-ti"
MACHINE="beagleplay-ti"
```

```
$ bitbake-getvar IMAGE_INSTALL
NOTE: Starting bitbake server...
The variable 'IMAGE_INSTALL' is not defined
```

Checking variable value — bitbake-getvar (2)

root commit

Use the -r option to get a variable local to a recipe:

```
$ bitbake-getvar -r core-image-minimal IMAGE_INSTALL
#
# $IMAGE_INSTALL [5 operations]
#   set /home/mike/yocto-labs/poky/meta/conf/documentation.conf:218
#     [doc] "Specifies the packages to install into an image. Image recipes set IMAGE_INSTALL to specify the packages to install into..
#   set /home/mike/yocto-labs/poky/meta/recipes-core/images/core-image-minimal.bb:3
#     "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"
#   set? /home/mike/yocto-labs/poky/meta/classes-recipe/core-image.bbclass:84
#     "${CORE_IMAGE_BASE_INSTALL}"
#   set? /home/mike/yocto-labs/poky/meta/classes-recipe/image.bbclass:84
#     ""
#   set /home/mike/yocto-labs/poky/meta/classes-recipe/image.bbclass:85
#     [type] "list"
# pre-expansion value:
#   "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"
IMAGE_INSTALL="packagegroup-core-boot "
```

Checking variable value — bitbake -e

Another possibility to check variable values is `bitbake -e`

- `bitbake -e` shows the global environment: variables and tasks
- `bitbake -e <recipe>` show the environment for a specific recipe

`bitbake -e` is useful when you don't know the variable(s) you're looking for.

Default and weak variable assignments

root commit

?= — default assignment

- Useful to set a default value, applied when the variable is still undefined

```
BITBAKE_PATIENCE ?= "Running low..."
```

Unlike = statements, it is applied at parsing time; the first one of this kind wins.

??= — weak assignment

- Last recourse value when no default value has been set.
- Applied at parsing time: the last one of this kind wins

```
WATCHDOG_TIMEOUT ??= "60"
```

Other operators

- `:=` — Assigned immediately
- `+=` — Append with a space
- `=+` — Prepend with a space
- `.=` — Append without a space
- `=.` — Prepend without a space

- `A := "keeps"`
- `A += "the doctor"`
- `A =+ "a day"`
- `A .= " away"`
- `A =. "An apple "`

All these operators, like `?=` and `??=`, are applied immediately (at parsing time).

Variable expansion

Variables can be expanded in other variables

```
A = "Thomas Edison"
B = "Nikola Tesla"
C = "${A} vs ${B}"
```

- `C := "${A} vs ${B}"` is evaluated immediately
- `C = "${A} vs ${B}"` is evaluated when `C` is accessed

- Curly braces are mandatory. `$A` won't get expanded.
- If `A` doesn't exist, `${A}` is kept as is in the string.

"Override" style operators

3 additional operators are available, using the "override" syntax (explained later)

:append: add to the end of a variable (no space added)

```
R = "Liberty"
R:append = " Pursuit of \
           Happiness"
```

:prepend: add to the end of a variable (no space added)

```
R:prepend = "Life "
```

:remove: remove all instances of a substring

```
R:remove = "Liberty"
```

Before version 4.0 (Kirkstone), a different override syntax was used (_ instead of :)

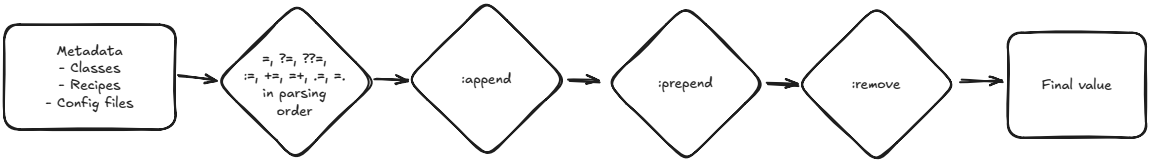
Parsing and evaluation order

root commit

The parsing order is hard to predict, so don't try to make assumptions.
Instead, try to follow these rules:

- Don't use immediately evaluated operations in `conf/local.conf`. Use = and override operators instead.
- In recipes and even more in classes (explained later), keep in mind to leave the user the possibility to modify the default values. Therefore, beware of = and prefer ?= and ??=.

Evaluation order

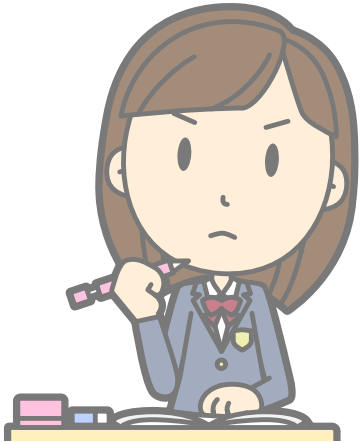


Lab — Basic BitBake variable operators

root commit

- Experiment with the various operators
- Try to solve a few challenges

Quiz — BitBake Variable Operators



→Click here←

Lab — Setting up networking

root commit

To enable networking between the PC and board

- Host-side networking setup
- Board-side networking setup. Configuration changes to enable a static IP address

Getting Started

Adding Packages to an Image

Using the IMAGE_INSTALL variable

Use `IMAGE_INSTALL` to add **packages** to an existing image

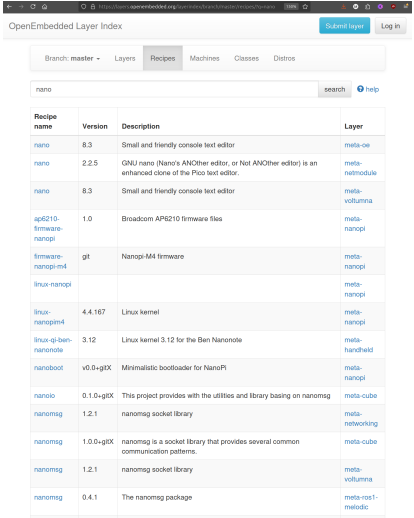
- Remember that packages are installed, not recipes
- You don't need all the packages produced by a recipe
- However, very often the main binary package name coincides with the recipe name
- Typically appended in `conf/local.conf`. Examples:

```
IMAGE_INSTALL:append = " os-release curl"
```

```
IMAGE_INSTALL:append = " libflac"  # Just libflac, not the flac executable
```

Find the recipe(s) you need

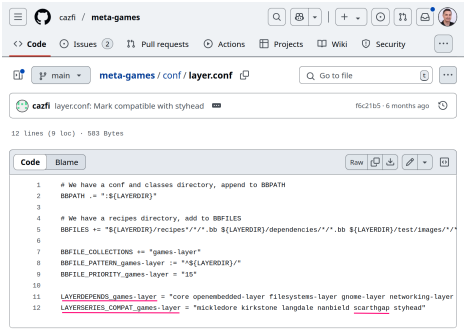
- Go to `https://layers.openembedded.org/`
- Click on the `Recipes` tab
- Make a search by recipe name
- You get the matching recipes and the layer they belong to.



Install additional layers

- Stay on `https://layers.openembedded.org/`, and visit the link for the layer your recipe belongs to.
- Follow the link in the web repo button.
- In the web repository, follow the link to the `conf/layer.conf` file.
- Check the `LAYERDEPENDS` and `LAYERSERIES_COMPAT` for additional layers, and for the compatibility with the Yocto/OE branch you're using.
- Clone the source code of your layer.
- Add your layer to `conf/bblayers.conf`:

```
$ bitbake-layers add-layer <path-to-layer>
```



Find the package(s) you need

root commit

You first need to build the recipe you are interested in:

```
bitbake flac
```

Then, you can query the packages that have been built:

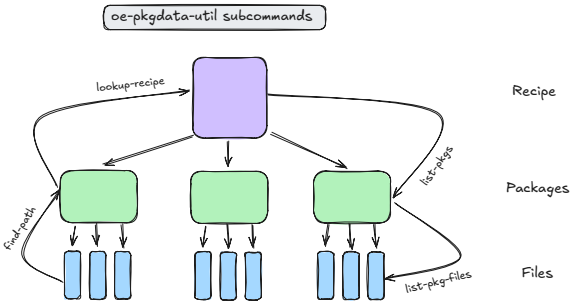
```
$ oe-pkgdata-util list-pkgs -p flac
flac
flac-dbg
flac-dev
flac-doc
flac-src
libflac
libflac++
```

oe-pkgdata-util script

`oe-pkgdata-util`:
Very handy script to query package information
(without having to install a package manager on the target).

- `oe-pkgdata-util find-path <path>`
  Find the package providing a path in the image
- `oe-pkgdata-util lookup-recipe <package>`
  Find the recipe implementing the package
- `oe-pkgdata-util list-pkgs -p <recipe>`
  List packages built by a recipe
- `oe-pkgdata-util list-pkg-files <package>`
  List files belonging to a package

If you forget: `oe-pkgdata-util --help`



oe-pkgdata-util script — Example

```
$ oe-pkgdata-util list-pkgs -p flac
flac
flac-dbg
flac-dev
flac-doc
flac-src
libflac
libflac++
```

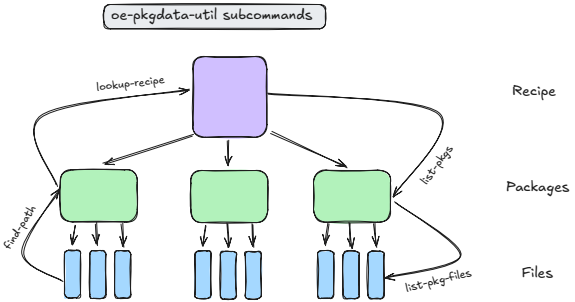
```
$ oe-pkgdata-util list-pkg-files libflac
libflac:
 /usr/lib/libFLAC.so.12
 /usr/lib/libFLAC.so.12.1.0
```

```
$ oe-pkgdata-util find-path /usr/lib/libFLAC.so.12
libflac: /usr/lib/libFLAC.so.12
```

```
$ oe-pkgdata-util lookup-recipe libflac
flac
```

Add Packages to an Image — Key Takeaways

- `IMAGE_INSTALL` takes **package names**, not recipe names
- One recipe can generate multiple packages: binaries, libraries, documentation...
- `oe-pkgdata-util` allows to navigate between recipes, packages and files



Lab — Adding Packages to an Image

root commit

To add software that we will need in later labs

- Look for recipes
- Look for packages to install

Getting Started

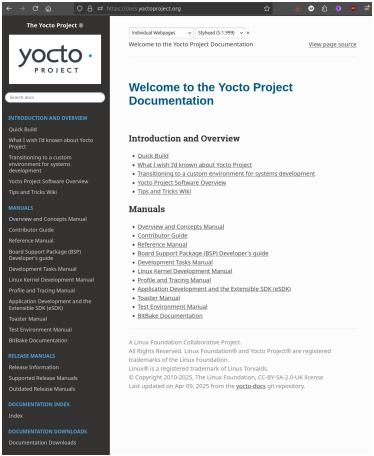
Documentation

Yocto Manuals Walkthrough

root commit

- You're talking to the right person (former docs maintainer)
- Multiple manuals:
  - Yocto manuals
  - BitBake manual
- Variable references in these materials link to the manuals. Example: `CFLAGS`

`https://docs.yoctoproject.org`



Let's take a tour!

Lab — Build Image for an Emulated ARM64 Machine

root commit

- Build for another machine: `genericarm64`
- Also on another release
- Test the image with QEMU

Recipes

Recipes

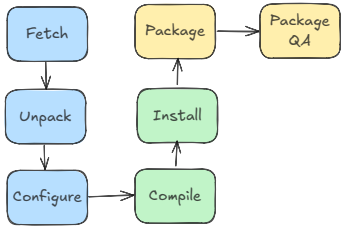
BitBake Recipes — Part 1

What is a Recipe?

A recipe defines:

- Primarily all the **tasks** to fetch sources, configure, build, install and deploy a given software component
  - Many components share the same **build system**: GNU autotools, CMake, Meson, Python, Rust, Go...
  - Therefore, their tasks have a lot in common. The shared code is handled by inheriting **classes** that take care of defining the common tasks.
  - All the recipe has to do is set variables interpreted by these classes.
- Generic information about a package component: description, homepage, license...

Main tasks in a recipe



Recipe Example — flac (part 1)

root commit

meta/recipes-multimedia/flac/flac_1.5.0.bb

```
SUMMARY = "Free Lossless Audio Codec"
DESCRIPTION = "FLAC stands for Free Lossless Audio Codec, a lossless audio compression format."
HOMEPAGE = "https://xiph.org/flac/"
BUGTRACKER = "https://github.com/xiph/flac/issues"
SECTION = "libs"
LICENSE = "GFDL-1.3 & GPL-2.0-or-later & LGPL-2.1-or-later & BSD-3-Clause"
LIC_FILES_CHKSUM = "file://COPYING.FDL;md5=802e79e394e372d01e863e3f4058cf40 \
                    file://src/Makefile.am;beginline=1;endline=17;md5=9c882153132df8f3a1cb1a8ca1f2350f \
                    file://COPYING.GPL;md5=b234ee4d69f5fce4486a80fdaf4a4263 \
                    file://src/flac/main.c;beginline=1;endline=18;md5=1e826b5083ba1e028852fe7ceec6a8ad \
                    file://COPYING.LGPL;md5=fbc093901857fcd118f065f900982c24 \
                    file://COPYING.Xiph;md5=78a131b2ea50675d245d280ccc34f8b6 \
                    file://include/FLAC/all.h;beginline=65;endline=70;md5=39aaf5e03c7364363884c8b8ddda8eea \
                    "

SRC_URI = "http://downloads.xiph.org/releases/flac/${BP}.tar.xz \
           file://0001-API-documentation-replace-modules.html-by-topics.htm.patch"

SRC_URI[sha256sum] = "f2c1c76592a82ffff8413ba3c4a1299b6c7ab06c734dee03fd88630485c2b920"

CVE_PRODUCT = "libflac flac"

inherit autotools gettext
```

Recipe Example — flac (part 2)

root commit

meta/recipes-multimedia/flac/flac_1.5.0.bb

```
EXTRA_OECONF = "--disable-oggtest \
                --without-libiconv-prefix \
                "

PACKAGECONFIG ??= " \
    ogg \
"
PACKAGECONFIG[avx] = "--enable-avx,--disable-avx"
PACKAGECONFIG[ogg] = "--enable-ogg --with-ogg-libraries=${STAGING_LIBDIR} --with-ogg-includes=${STAGING_INCDIR},--disable-ogg,libogg"

PACKAGES += "libflac libflac++"
FILES:${PN} = "${bindir}/*"
FILES:libflac = "${libdir}/libFLAC.so.*"
FILES:libflac++ = "${libdir}/libFLAC++.so.*"

do_install:append() {
    # make the links in documentation relative to avoid buildpaths reproducibility problem
    sed -i "s#${S}/include#${includedir}#g" ${D}${docdir}/flac/FLAC.tag ${D}${docdir}/flac/api/*.html
    # there is also one root path without trailing slash
    sed -i "s#${S}#/#g" ${D}${docdir}/flac/api/*.html
}
```

Recipe File Path and Naming Scheme

`<layer>/recipes-<type>/<name>_<version>.bb`

Examples:

```
meta/recipes-multimedia/ffmpeg/ffmpeg_7.1.1.bb
meta/recipes-core/musl/musl_git.bb
meta/recipes-core/images/core-image-minimal.bb
```

- A recipe can support multiple versions
- Version independent files can be included:

```
meta/recipes-support/vim/vim.inc
meta/recipes-support/vim/vim_9.1.bb
meta/recipes-support/vim/vim-tiny_9.1.bb
```

Recipe Header Fields

Some of them correspond to package metadata fields:

- `SUMMARY` Short description of the binary packages, at most 72 characters
- `HOMEPAGE` Website with more information about the software the recipe is building
- `DESCRIPTION` The package description used by package managers. If not set, `DESCRIPTION` takes the value of `SUMMARY`.
- `SECTION` The section in which packages should be categorized.
- `BUGTRACKER` URL for an upstream bug tracking website for a recipe.

Example: `meta/recipes-kernel/dtc/dtc_1.7.2.bb`

```
SUMMARY = "Device Tree Compiler"
HOMEPAGE = "https://devicetree.org/"
DESCRIPTION = "The Device Tree Compiler is a toolchain for
working with device tree source and binary files."
SECTION = "bootloader"
LICENSE = "GPL-2.0-only | BSD-2-Clause"
```

LICENSE field

An important field for license compliance, especially to know your obligations when you ship a product.

- Must be set using one of the SPDX license identifiers (listed on `https://spdx.org/licenses/`).
- Example:

```
LICENSE = "BSD-3-Clause"
```

- A source package can have components with different licenses. Example (`libgit2`):

```
LICENSE = "GPL-2.0-with-GCC-exception & MIT & OpenSSL & BSD-3-
Clause & Zlib & ISC & LGPL-2.1-or-later & CC0-1.0 & BSD-2-Clause"
```

LIC_FILES_CHKSUM field

- Mandatory checksum for license files (except if `LICENSE = "CLOSED"`)
- Necessary to detect a license change in the sources. If this happens, the recipe won't build anymore, to catch the attention of the recipe maintainer.
- Example (`nettle`):

```
LICENSE = "LGPL-3.0-or-later | GPL-2.0-or-later"

LIC_FILES_CHKSUM = "file://COPYING.LESSERv3;md5=6a6a8e020838b23406c81b19c1d46df6 \
                    file://COPYINGv2;md5=b234ee4d69f5fce4486a80fdaf4a4263 \
                    file://serpent-decrypt.c;beginline=14;endline=36;md5=ca0d220bc413e1842ecc507690ce416e \
                    file://serpent-set-key.c;beginline=14;endline=36;md5=ca0d220bc413e1842ecc507690ce416e"
```

Automatic Recipe Variables

root commit

`PN` Originally *Package Name*? Confusing naming. Name part extracted from the recipe file, without the version information. Examples: `flac`, `cmake-native`

`BPN` `PN` value with common prefixes and suffixes removed, such as `nativesdk-`, `-cross`, `-native`... Examples: `flac`, `cmake`

`PV` Version of the recipe extracted from the recipe file name. Example: `3.31.6` for `cmake_3.31.6.bb`. Overridden in recipes building from development versions (git sources). Example: `meta/recipes-support/sass/sassc_git.bb`: `PV = "3.6.2"`.

`BP` Equals `${BPN}-${PV}`. Mostly useful in source URLs. Examples: `meta/recipes-support/lzop/lzop_1.04.bb`: `SRC_URI = "http://www.lzop.org/download/${BP}.tar.gz`

SRC_URI: Fetching Sources (1)

root commit

- Can be a fixed release archive:

meta/recipes-multimedia/flac/flac_1.5.0.bb

```
SRC_URI = "http://downloads.xiph.org/releases/flac/${BP}.tar.xz \
           file://0001-API-documentation-replace-modules.html-by-topics.htm.patch"

SRC_URI[sha256sum] = "f2c1c76592a82ffff8413ba3c4a1299b6c7ab06c734dee03fd88630485c2b920"
```

- The checksum is needed in case the upstream server is compromised.

SRC_URI: Fetching Sources (2)

- Can be the URL of a source repository:

meta/recipes-support/bmaptool/bmaptool_git.bb

```
SRC_URI = "git://github.com/yoctoproject/${BPN};branch=main;protocol=https"
SRCREV = "2ff5750b8a3e0b36a9993c20e2ea10a07bc62085"
S = "${WORKDIR}/git"
BASEVER = "3.8.0"
PV = "${BASEVER}+git"
```

- SRCREV is necessary to know which commit to fetch
- Also need to set S to point to the cloned directory
- PV should be set too if it can't be extracted from the recipe filename.

Lab — Create your first recipe

root commit

Of course for a `hello world` application

- Create your first layer
- Create your recipe using `devtool`
- Deploy and test it on the target

Recipes

BitBake Recipes — Part 2

Adding patches and other files

root commit

As you could see SRC_URI supports multiple **fetchers**:

- http:// or https:// for source archives
- git:// and other fetchers for source control repositories
- file:// for local files, patches in particular:

meta/recipes-support/boost/boost_1.87.0.bb

```
SRC_URI += "file://boost-math-disable-pch-for-gcc.patch \
```

The files are found in the recipes themselves, typically in the BPN (recipe name) subdirectory. We will cover the actual search path later.

Defining packages

root commit

Let's look at the `flac` recipe again:

```
PACKAGES += "libflac libflac++"
FILES:${PN} = "${bindir}/*"
FILES:libflac = "${libdir}/libFLAC.so.*"
FILES:libflac++ = "${libdir}/libFLAC++.so.*"
```

Defining packages

Let's look at the `flac` recipe again:

```
PACKAGES += "libflac libflac++"
FILES:${PN} = "${bindir}/*"
FILES:libflac = "${libdir}/libFLAC.so.*"
FILES:libflac++ = "${libdir}/libFLAC++.so.*"
```

- `PACKAGES`: defines the list of packages to generate.
- Default value for `PACKAGES`:

  `${PN}-src ${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN}`
- `FILES`: defines the contents of the packages from the output of the `install` task.
- Packages with no matching files are not generated.

Dependency Fields

Two types of dependencies

`DEPENDS` : list of **build-time** dependencies (recipe names)

`RDEPENDS` : list of **run-time** dependencies (package names)
These correspond to dependencies between packages. Installing a package will automatically require the installation of its dependencies.

It's rare to need to set `RDEPENDS` in recipes, shared library dependencies are automatically figured out by BitBake. Only dependencies not deducible at compile time must be added (external programs, data files...)

Example: `meta/recipes-kernel/perf/perf.bb`

```
DEPENDS = " \
    virtual/${MLPREFIX}libc \
    ${MLPREFIX}elfutils \
    ${MLPREFIX}binutils \
    bison-native flex-native xz \
"
...
RDEPENDS:${PN} += "elfutils bash"
RDEPENDS:${PN}-archive =+ "bash"
RDEPENDS:${PN}-python =+ "bash python3 python3-
modules ${@bb.utils.contains('PACKAGECONFIG', 'audit', 'audit-
python', '', d)}"
RDEPENDS:${PN}-perl =+ "bash perl perl-modules"
RDEPENDS:${PN}-tests =+ "python3 bash perl"
```

Per recipe sysroots

root commit

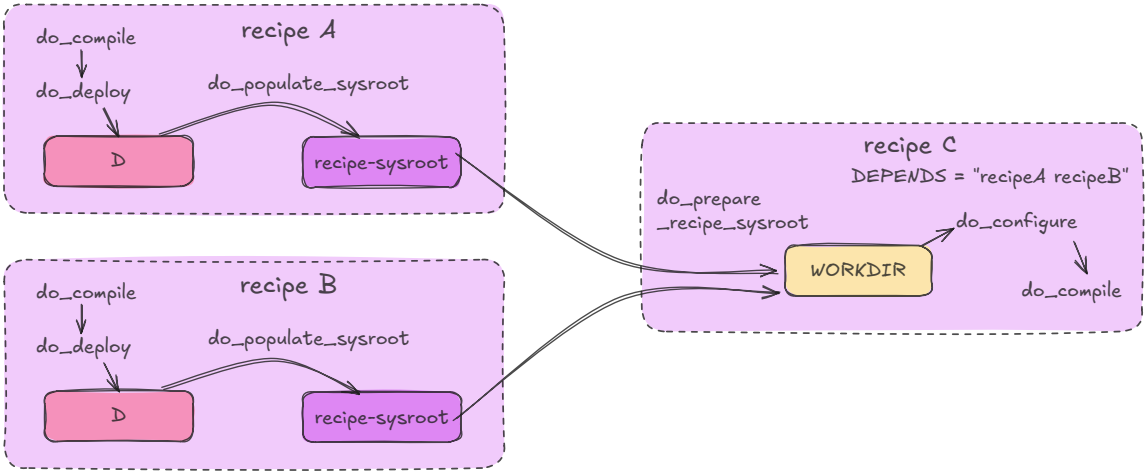
Sysroot

- Everything a C/C++ program needs to compile and link with other software:
  - C headers (`.h` files)
  - Shared libraries (`.so` files)

Instead of having a global sysroot with everything that was built, BitBake uses **per-recipe sysroots**:

- During the `do_populate_sysroot`, each recipe stores its headers and libraries in its own **output sysroot**.
- When a recipe depends on others, its `do_prepare_recipe_sysroot` task fetches the output sysroots of such recipes into its own sysroot.
- This way, success doesn't depend on luck (whether something was built before or not), and missing dependencies are detected immediately.

Per recipe sysroots illustrated



Lab — Create a recipe for the MyMan ASCII game

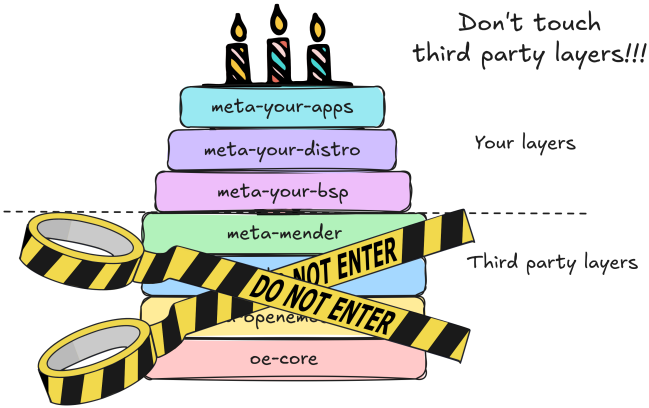
root commit

- A more complicated recipe with dependencies
- Figure out the license too
- Multiple packages needed

Recipes

Modifying Recipes

Most Important Yocto Law



BitBake bbappend mechanism

.bbappend files can be used to extend or override existing recipes.
To be applied, a matching .bb file must be found:

- busybox_1.37.0.bbappend overrides busybox_1.37.0.bb
- busybox_1.37.%.bbappend overrides all busybox_1.37.*.bb recipes
- busybox_%.bbappend overrides all busybox_*.bb recipes

bbappends are typically stored in the same directory structure as the original recipe (recipes-kernel/<recipe>/, recipes-multimedia/<recipe>/...)

Giving precedence to files supplied by `bbappends`

root commit

Some `bbappends` want to add or even replace files in the original recipes

- This is done by modifying the search path for files, putting the `bbappend`'s directories first.
- In most cases, done by (assuming the files are stored in `files/`):

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
```

bbappend example

root commit

meta-raspberrypi/recipes-bsp/u-boot/u-boot_%.bbappend

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"

SRC_URI:append:rpi = " \
    file://fw_env.config \
"

SRC_URI:append:rpi = " file://0001-rpi-always-set-fdt_addr-with-firmware-provided-FDT-address.patch"
SRC_URI:append:raspberrypi4 = " file://maxsize.cfg"

DEPENDS:append:rpi = " u-boot-default-script"

do_install:append:rpi () {
    install -d ${D}${sysconfdir}
    install -m 0644 ${UNPACKDIR}/fw_env.config ${D}${sysconfdir}/fw_env.config
}
```

Note: `:rpi` and `:raspberrypi4` are **overrides** (covered later).

They allow to apply some settings only on specific conditions.

Checking bbappends

Checking all bbappends:

```
$ bitbake-layers show-appends
```

Checking one bbappend:

```
$ bitbake-layers show-appends
bitbake-layers show-appends psplash
...
=== Matched appended recipes ===
psplash_git.bb:
  /home/mike/yocto-labs/poky/meta-poky/recipes-core/psplash/psplash_git.bbappend
```

Check what this particular bbappend does!

Lab — Tweaking the Device Tree

root commit

- Connect an I2C device to the board
- Override the U-Boot recipe to apply a patch that adds the I2C device to the board device tree.

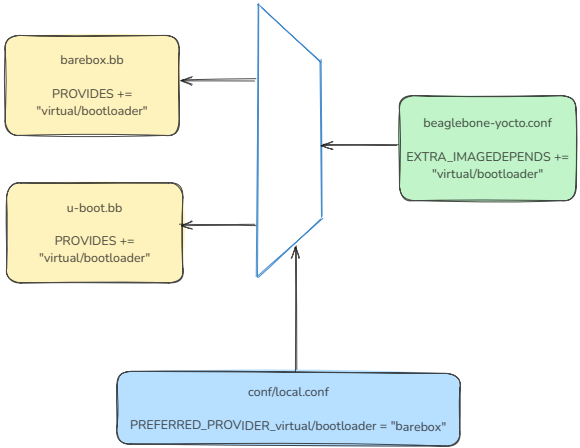
Recipes

Virtual Packages

Need for Virtual Packages

Typically to express a generic dependency to something that can have multiple implementations. Examples:

- `virtual/kernel`: `linux-yocto`, `linux-yocto-rt`, `linux-yocto-tiny`, `linux-dummy`...
- `virtual/bootloader`: Barebox, U-Boot...
- `virtual/libc`: Musl, GNU libc...
- `virtual/cross-cc`: Clang, GCC...
- And a few more



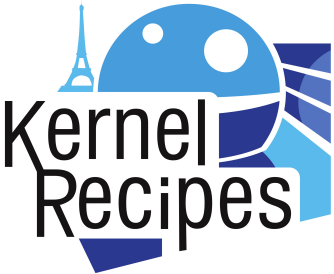
Recipes

Kernel Recipes

Multiple Kernel Recipes

Thanks to the `virtual/kernel` virtual package, multiple options are available:

- `linux-yocto`: the default kernel recipe in Poky
- Your own kernel recipes based on `kernel.bbclass`
- SOC or board vendor recipes, e.g. `linux-ti-staging`, `linux-ti-mainline`...
- `linux-dummy`: simple placeholder recipe, for kernel built outside of Yocto.



https://kernel-recipes.org

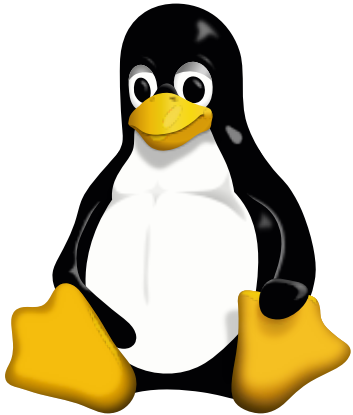
linux-yocto

`linux-yocto` is a Linux kernel recipe with some additional features not found in `kernel.bbclass`

- Integrated with Yocto special kernel tooling, kernel featuresets and config fragments
  - This allows to apply specific configuration settings or patches according to features on the platform (like Bluetooth or sound support).
- Several variants: `linux-yocto-dev`, `linux-yocto-rt`, `linux-yocto-tiny`
- Managed through a special repository: `https://git.yoctoproject.org/linux-yocto/`
  - But the `master` branch is 5,570 commits ahead of mainline `master` (from `git rebase -i`, on Apr. 18, 2025).
  - That's difficult to maintain: most unpatched vulnerabilities are in `linux-yocto`. See `https://autobuilder.yocto.io/pub/non-release/patchmetrics/`
  - My own experience: that's not sustainable to maintain your own kernel tree with a significant delta vs mainline. Desktop distributions are very close to mainline.

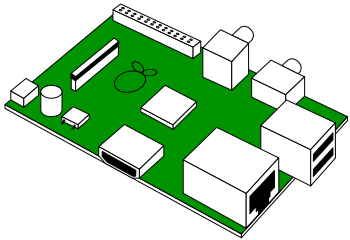
Own Kernel Recipe with `kernel.bbclass`

- Typically based on the stable mainline Linux kernels
- Very easy to create — See presentation at OE Workshop 2025: `https://rootcommit.com/pub/conferences/2025/oe-workshop/yocto-mainline-linux-uboot/`
- Alternative: use the `meta-linux-mainline` layer: `https://github.com/betafive/meta-linux-mainline`
- Need to include a `defconfig` configuration file for each machine
  - You can easily get one:
    ```
    $ bitbake -c menuconfig
    $ bitbake -c savedefconfig
    ```
- You very quickly get the latest vulnerability fixes from the mainline stable trees.
- Find out by yourself in our practical labs!



SoC or Board Vendor Kernel Recipe

- Convenient to get early support for recent hardware.
- But full of unfixed vulnerabilities (beyond the vendor's capabilities)
- But may be replaced by a mainline kernel on mature platforms
  - You can keep the vendor BSP layer just for the bootloader, firmware and specific tools!
  - Consider switching, could be pretty cheap
  - Will be less costly if vulnerabilities are considered
  - Upgrading and access to new features will be easier



Lab — Switching to the Mainline Linux Kernel

root commit

- Create a mainline kernel recipe based on `kernel.bbclass`
- Build the image with this kernel
- Customize the kernel configuration to support the I2C gamepad

Recipes

devtool

devtool — Why it exists

root commit

- The build system is great at generating reproducible builds from sources and metadata
- However, it's not adapted to software development.
  - You could try to directly work with sources under `tmp/work`
  - But this leads to all sorts of complications and the build system may erase your changes doing that.
- devtool is the cornerstone of the BitBake Extensible Software Development Kit (eSDK)
  - It makes it easy to build experimental code, taking care of all the tedious recipe management tasks.
  - It can help new developers create new recipes
  - Even developers without BitBake can build applications for a given product.

devtool — What you can do with it

root commit

Devtool can be used to

- Create a new recipe from the sources of a component
  - It guesses the recipe name
  - It automatically recognizes the build system for the program
  - It detects the license files
  - It figures out some dependencies
  - The generated recipe needs human review, but a big part of the job is done
- Check whether a recipe has an update upstream
- Propose patches to upgrade a recipe to a newer upstream
- Modify an existing recipe
- Create an IDE configuration for your recipes
- Compile applications that you are developing
- Even without having BitBake (while using an *Extended SDK*)
- Deploy your applications to your target system
- Build an image with recipes under development

devtool subcommands (1)

```
$ devtool --help
NOTE: Starting bitbake server...
usage: devtool [--basepath BASEPATH] [--bbpath BBPATH] [-d] [-q] [--color COLOR] [-h] <subcommand> ...

OpenEmbedded development tool

options:
  --basepath BASEPATH   Base directory of SDK / build directory
  --bbpath BBPATH       Explicitly specify the BBPATH, rather than getting it from the metadata
  -d, --debug           Enable debug output
  -q, --quiet           Print only errors
  --color COLOR         Colorize output (where COLOR is auto, always, never)
  -h, --help            show this help message and exit

subcommands:
  Beginning work on a recipe:
    add                   Add a new recipe
    modify                Modify the source for an existing recipe
    upgrade               Upgrade an existing recipe
  Getting information:
    status                Show workspace status
    search                Search available recipes
    latest-version        Report the latest version of an existing recipe
    check-upgrade-status  Report upgradability for multiple (or all) recipes
```

devtool subcommands (2)

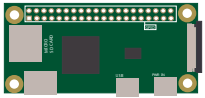
```
  Working on a recipe in the workspace:
    ide-sdk               Setup the SDK and configure the IDE
    build                 Build a recipe
    rename                Rename a recipe file in the workspace
    edit-recipe           Edit a recipe file
    find-recipe           Find a recipe file
    configure-help        Get help on configure script options
    update-recipe         Apply changes from external source tree to recipe
    reset                 Remove a recipe from your workspace
    finish                Finish working on a recipe in your workspace
  Testing changes on target:
    deploy-target         Deploy recipe output files to live target machine
    undeploy-target       Undeploy recipe output files in live target machine
    build-image           Build image including workspace recipe packages
  Advanced:
    create-workspace      Set up workspace in an alternative location
    export                Export workspace into a tar archive
    extract               Extract the source for an existing recipe
    sync                  Synchronize the source tree for an existing recipe
    import                Import exported tar archive into workspace
    menuconfig            Alter build-time configuration for a recipe
Use devtool <subcommand> --help to get help on a specific command
```

devtool subcommand help

```
devtool check-upgrade-status --help
NOTE: Starting bitbake server...
usage: devtool check-upgrade-status [-h] [--all] [recipe ...]

Prints a table of recipes together with versions currently provided by recipes, and latest upstream versions, when there is a later
version available

arguments:
  recipe      Name of the recipe to report (omit to report upgrade info for all recipes)

options:
  -h, --help  show this help message and exit
  --all, -a   Show all recipes, not just recipes needing upgrade
```

devtool workspace

root commit



devtool subcommands

deploy-target
undeploy-target

finish

your layers

modify

upgrade

configure-help build build-image

devtool workspace

find-recipe status

rename reset

add

check-upgrade-status

status

Image credits:
https://openclipart.org/detail/35389/tango-applications-internet
https://openclipart.org/detail/299799/raspberry-pi-zero

The core machinery to work on recipes — used by `devtool`

- Used by `devtool add` to create recipes
- Can also be used:
  - to set variables in a recipe from a script
  - to create and update `bbappends` files

devtool workspace details

- `workspace` is a layer, with priority 99 (see `conf/layer.conf`)
- `appends`: bbappends for the recipes in the workspace
- `sources`: copy of the recipe sources — You can directly edit them!
  Each subdirectory is a git repository

~/yocto-labs/poky/build/workspace/sources/myman ᛦ devtool

- `attic`: backup of sources after being removed from the workspace by `devtool reset` or `devtool finish`

~/yocto-labs/poky/build/workspace ᛦ scarthgap tree -d -L 3

```
.
├── appends
├── attic
│   └── sources
│       └── u-boot-ti-staging.20250416074047
├── conf
└── sources
    ├── hello
    │   ├── build-aux
    │   ├── contrib
    │   ├── doc
    │   ├── lib
    │   ├── m4
    │   ├── man
    │   ├── po
    │   ├── src
    │   └── tests
    └── myman
        ├── autom4te.cache
        ├── chr
        ├── debian
        ├── efilibc
        ├── fink
        ├── gfx
        ├── inc
        ├── lvl
        ├── m4
        ├── mygetopt
        ├── oe-logs -> /home/mike/yocto-labs/poky/build/tmp/work/aarch64-poky-linux/myman/0.1+git/temp
        ├── oe-workdir -> /home/mike/yocto-labs/poky/build/tmp/work/aarch64-poky-linux/myman/0.1+git
        ├── patches
        ├── sfx
        ├── spr
        ├── src
        ├── utl
        └── website

36 directories
```

devtool command sample output

```
$ devtool status  # Information about recipes in your workspace
hello: /home/mike/yocto-labs/poky/build/workspace/sources/hello
myman: /home/mike/yocto-labs/poky/build/workspace/sources/myman
```

```
$ devtool search i2c  # Smart search for recipes (name, description, package contents...)
i2c-tools             Set of i2c tools for linux
linux-libc-headers    Sanitized set of kernel headers for the C library's use
linux-mainline
i2cdev                i2c dev tools for Linux
```

```
$ devtool find-recipe i2c-tools  # Find path to recipe
/home/mike/yocto-labs/poky/meta/recipes-devtools/i2c-tools/i2c-tools_4.3.bb
```

SSH host definitions

Useful for `devtool deploy-target` and `devtool undeploy-target`

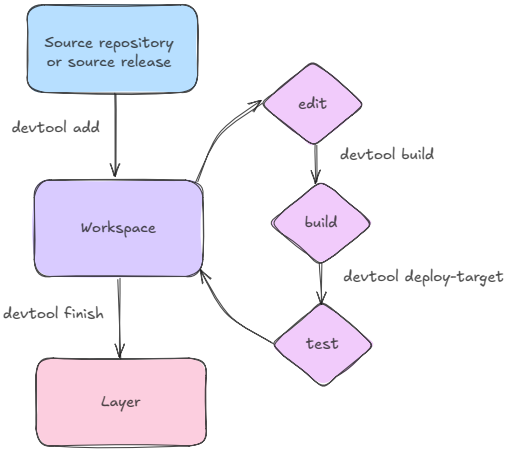
- Need an SSH server (like `openssh`) in the target image
- Usage `devtool deploy-target <recipe> <target>`
- `devtool deploy-target` needs the recipe to be built in the workspace
- `devtool undeploy-target` is optional before one more `devtool deploy-target`, but useful to remove no longer wanted files.

Entry in $HOME/.ssh/config

```
Host beagleplay
                User root
                Hostname 172.24.0.2
                Port 22
                StrictHostKeyChecking no
                UserKnownHostsFile /dev/null
```

devtool add workflow

1. Create recipe in workspace from remote or local sources:
   `devtool add https://ftp.gnu.org/gnu/hello/hello-2.12.1.tar.gz`
2. Review and fix the recipe in the workspace
3. Build and test recipe on the target:
   `devtool build hello`
   `devtool deploy-target hello beagleplay`
4. Copy new recipe to layer and clean workspace:
   `devtool finish hello ../../meta-homebrew -N` (dry run)
   `devtool finish hello ../../meta-homebrew`
   `devtool finish hello`



devtool upgrade workflow

1. Check whether a new release exists upstream:

```
$ devtool check-upgrade-status alsa-lib
alsa-lib                  1.2.13          1.2.14          Michael Opdenacker <michael@opdenacker.org>
```

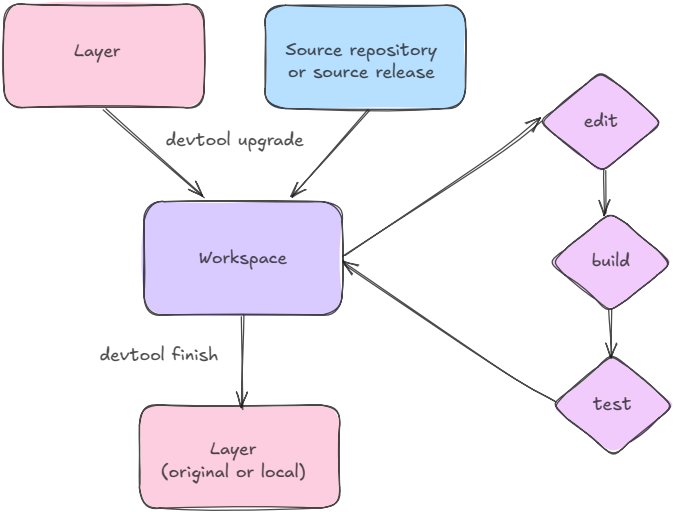
2. Import recipe from layer and try to upgrade it:

```
$ devtool upgrade alsa-lib
INFO: Upgraded source extracted to /home/mike/work/yocto/poky/build/workspace/sources/alsa-lib
INFO: New recipe is /home/mike/work/yocto/poky/build/workspace/recipes/alsa-lib/alsa-lib_1.2.14.bb
```

3. Review, build and test new release version
4. Publish new version to layer:

```
$ devtool finish -f alsa-lib ../../meta-testlayer
```

devtool upgrade workflow — Summary



devtool modify workflow

1. Import an existing recipe into the workspace

```
$ devtool modify myman
```

2. Modify sources or recipe:
3. Build and test the sources

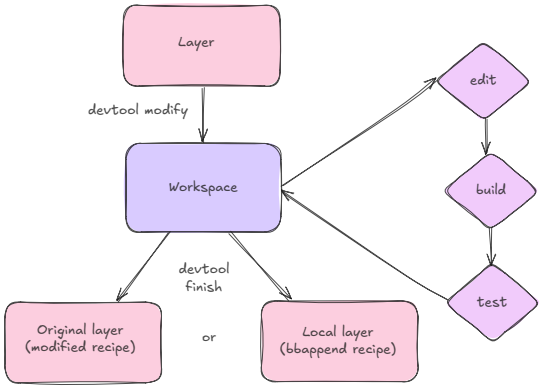
```
$ devtool build myman
$ devtool deploy-target myman beagleplay
```

4. Commit your changes

```
$ pushd workspace/sources/myman
$ git commit -as
$ popd
```

5. Publish your changes to a layer

```
$ devtool finish -f myman ../../meta-homebrew
```



devtool — Key Takeaways

- Use `devtool add` to create new recipes
- Very smart logic: produced recipes are almost ready
- Use `devtool modify` and `devtool upgrade` to modify recipes
- The `workspace` directory is a top priority layer
- Very easy to compile applications under development (`devtool build`) and test them on the target (`devtool deploy-target`) without having to commit your changes so that they can be used by a formal recipe.

Lab — Modify the `myman` recipe

- Add joystick support to MyMan
- Compile and test your changes without committing them
- Generate a patch and update the recipe
- Play the game with your joystick!

Recipes

BitBake Overrides

Overrides are a way to make variable definitions conditional

Example

```
OVERRIDES = "architecture:os:machine"
TEST = "default"
TEST:machine = "machine specific"
TEST:os = "os specific"
TEST:architecture = "architecture specific"
```

After parsing this, when accessing TEST:

- Foreach key in OVERRIDES from right to left:
  - If, in the current project, key has the same value as in the TEST:<key> statement, then give TEST the matching value. Exit.
- If no key matched, TEST gets its default value.

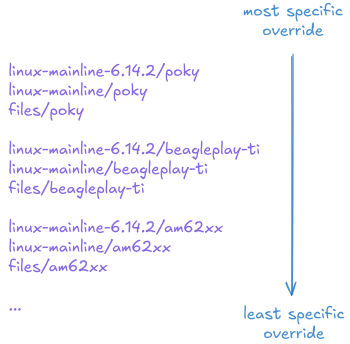
Overrides in File Search Paths

root commit

Overrides are particularly useful in the search file for files (`file://`)

```
$ bitbake-getvar -r linux-mainline OVERRIDES
# pre-expansion value:
#   "${TARGET_OS}:${TRANSLATED_TARGET_ARCH}:pn-${PN}:layer-${FILE_LAYERNAME}:
#   ${MACHINEOVERRIDES}:${DISTROOVERRIDES}:${CLASSOVERRIDE}
#   ${LIBCOVERRIDE}:forcevariable"
OVERRIDES="linux:aarch64:pn-linux-mainline:layer-meta-homebrew:bsp-ti-6_12:aarch64:ti-
soc:k3:am62xx:beagleplay-ti:poky:class-target:libc-glibc:forcevariable"
```

```
bitbake-getvar -r linux-mainline FILESPATH
# pre-expansion value:
#   "${@base_set_filesath(["${FILE_DIRNAME}/${BP}", "${FILE_DIRNAME}/${BPN}",
#   "${FILE_DIRNAME}/files"], d)}"
FILESPATH="/home/mike/yocto-labs/meta-homebrew/recipes-kernel/linux-mainline/linux-
mainline-6.14.2/poky:<...>/linux-mainline/poky:<...>/files/poky:<...>/linux-
mainline-6.14.2/beagleplay-ti:<...>/linux-mainline/beagleplay-
ti:<...>/files/beagleplay-ti:<...>/linux-mainline-6.14.2/am62xx:<...>/linux-
mainline/am62xx:<...>/files/am62xx:<...>/linux-mainline-6.14.2/k3:<...>/linux-
mainline/k3:<...>/files/k3:<...>/linux-mainline-6.14.2/ti-soc:<...>/linux-
mainline/ti-soc:<...>/files/ti-soc:<...>/linux-mainline-6.14.2/aarch64:<...>/linux-
mainline/aarch64:<...>/files/aarch64:<...>/linux-mainline-6.14.2/bsp-ti-
6_12:<...>/linux-mainline/bsp-ti-6_12:<...>/files/bsp-ti-6_12:<...>/linux-mainline-
6.14.2/aarch64:<...>/linux-mainline/aarch64:<...>/files/aarch64:<...>/linux-mainline-
6.14.2/:<...>/linux-mainline/:<...>/files/"
```



Yocto overrides quiz 1

root commit

In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:genericarm64 += "file://defconfig"
```

Value of `SRC_URI`?

Yocto overrides quiz 1

root commit

In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:genericarm64 += "file://defconfig"
```

Value of `SRC_URI`?

```
SRC_URI = " file://defconfig"
```

Why?

Yocto overrides quiz 1

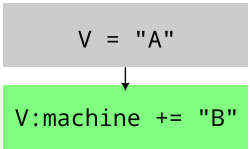
In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:genericarm64 += "file://defconfig"
```

Value of `SRC_URI`?

```
SRC_URI = " file://defconfig"
```

Why?



Evaluations steps:

V = "A"

V = " B"

+= applies to V:machine (initially "")

Yocto overrides quiz 2

root commit

In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:genericarm64:append = " file://defconfig"
```

Value of `SRC_URI`?

Yocto overrides quiz 2

root commit

In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:genericarm64:append = " file://defconfig"
```

Value of `SRC_URI`?

```
SRC_URI = " file://defconfig"
```

Why?

Yocto overrides quiz 2

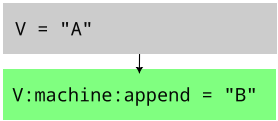
In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:genericarm64:append = " file://defconfig"
```

Value of `SRC_URI`?

```
SRC_URI = " file://defconfig"
```

Why?



Evaluations steps:

V = "A"

V = " B"

append applies to `V:machine` (initially "")

Yocto overrides quiz 3

root commit

In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:append:genericarm64 = " file://defconfig"
```

Value of `SRC_URI`?

Yocto overrides quiz 3

In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:append:genericarm64 = " file://defconfig"
```

Value of `SRC_URI`?

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https file://defconfig"
```

Why?

Yocto overrides quiz 3

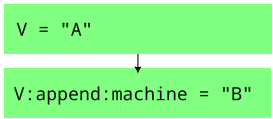
In a context where `MACHINE = "genericarm64"`

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https"
SRC_URI:append:genericarm64 = " file://defconfig"
```

Value of `SRC_URI`?

```
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git;branch=linux-
6.13.y;protocol=https file://defconfig"
```

Why?



Evaluations steps:

V = "A"

V = "A B" append applies to V in the machine case

Overrides — What to remember

- Overrides allow to specialize a variable (or even a task) for a given machine, architecture, C library or distribution.
- Use the `bitbake-getvar` command to track the various sources contributing to the final value of a variable.
- Overrides can be overridden 🤣 using the `OVERRIDES` variable!
- When appending a machine specific setting to a variable, place `append` **before** the override.

Blog post: `https://rootcommit.com/2025/yocto-variable-overrides-tricks/`

Lab — Smarter Kernel Recipe

root commit

- Create a recipe supporting two kernel versions, and two machines
- Avoid code duplication
- Have version and machine specific configurations

Recipes

Task Details

- Network access is only enabled during the `do_fetch` task.
- This guarantees that nothing else downloads code without the supervision of the build system.

If your component uses a `configure` script

- You often need to pass specific options to `configure`
- Do it with the `EXTRA_OECONF` variable

meta/recipes-support/sqlite/sqlite3.inc

```
EXTRA_OECONF = " \
    --enable-shared \
    --enable-threadsafe \
    --disable-static-shell \
"
```

The configuration options are also configurable 😉

- You can set PACKAGECONFIG entries in your recipes
- Set the PACKAGECONFIG variable to the list of features you want to enable in your recipe

```
PACKAGECONFIG ??= "f1 f2 f3 ..."
PACKAGECONFIG[f1] = "\
    --with-f1, \
    --without-f1, \
    build-deps-for-f1, \
    runtime-deps-for-f1, \
    runtime-recommends-for-f1, \
    packageconfig-conflicts-for-f1"
PACKAGECONFIG[f2] = "\
     ... and so on and so on ...
```

meta/recipes-support/sqlite/sqlite3.inc

```
PACKAGECONFIG ?= "fts4 fts5 rtree dyn_ext"
PACKAGECONFIG:class-native ?= "fts4 fts5 rtree dyn_ext"

PACKAGECONFIG[editline] = "--enable-editline,--disable-editline,libedit"
PACKAGECONFIG[readline] = "--enable-readline,--disable-readline,readline ncurses"
PACKAGECONFIG[fts3] = "--enable-fts3,--disable-fts3"
PACKAGECONFIG[fts4] = "--enable-fts4,--disable-fts4"
PACKAGECONFIG[fts5] = "--enable-fts5,--disable-fts5"
PACKAGECONFIG[rtree] = "--enable-rtree,--disable-rtree"
PACKAGECONFIG[session] = "--enable-session,--disable-session"
PACKAGECONFIG[dyn_ext] = "--enable-dynamic-extensions,--disable-dynamic-extensions"
PACKAGECONFIG[zlib] = ",,zlib"
```

- PACKAGECONFIG can also be modified from a local or distribution configuration file:

```
PACKAGECONFIG:append:pn-recipename = " f4"
```

- Some applications with simple Makefiles hardcode some variables, such as `CC`, `CFLAGS` and `LDFLAGS`
- This bypasses the settings from the build system, and prevents from cross-compiling
- Workaround: invoke `make` with the `-e` option, so that variables from the environment take precedence.

meta/recipes-extended/ed/ed_1.21.1.bb

```
EXTRA_OEMAKE = "-e MAKEFLAGS="
```

do_install

- Installing files is normally taken care of by the classes you're using (`autotools`, `cmake`...)
- If files must be installed "manually", you have to create the target directories, because they are recipe specific.

meta/recipes-devtools/mmc/mmc-utils_git.bb

```
do_install() {
    install -d ${D}${bindir}
    install -m 0755 mmc ${D}${bindir}
}
```

- This task runs many sanity checks on the generated packages
- Whole list of checks on `https://docs.yoctoproject.org/ref-manual/classes.html#ref-classes-insane`
- Examples
  - `buildpaths`: detect paths on the build host
  - `already-stripped`: detect executables stripped before the build system gets a chance to make a `-dbg` package with the unstripped version
  - `staticdev`: detect static library files (`*.a`) in non `-staticdev` packages.
- Sometimes, some of these checks are irrelevant and need to be skipped (per package definition):

meta/recipes-connectivity/openssl/openssl_3.4.1.bb

```
INSANE_SKIP:${PN} = "already-stripped"
```

Recipes

Debugging Recipes

Useful variables

root commit

- `S`: directory containing the component sources
- `T`: directory where BitBake stores temporary files for a recipe
  - Contains scripts created to run tasks (e.g. `run.do_configure`). You could run them directly!
  - Contains the log files of such tasks (e.g. `log.do_configure`)
- `B`: directory where the recipe is built. Usually equal to `${S}`.
- `FILE`: exact recipe file used to build the recipe

Don't try to remember the internal paths used by BitBake. Use `bitbake-getvar`.

Other useful features for debugging

root commit

- devshell task:
  Sets all the variables so that you can manually run commands such as configure and make.
  See https://docs.yoctoproject.org/dev-manual/development-shell.html

  ```
  $ bitbake -c devshell myman
  ```

- pydevshell task:
  Allows to execute Python functions as if you were in the BitBake environment. Great for checking Python code, access variables, run tasks manually.
  See https://docs.yoctoproject.org/dev-manual/python-development-shell.html

  ```
  $ bitbake -c pydevshell myman
  ```

- buildhistory class:
  Records information about the contents of each package and image and stores it into a local Git repository. Useful for tracking unexpected regressions. See https://docs.yoctoproject.org/dev-manual/build-quality.html
  Set this in conf/local.conf:

  ```
  INHERIT += "buildhistory"
  BUILDHISTORY_COMMIT = "1"
  ```

Layers

Layers

BSP Layers

Purpose of BSP Layers

Goal: support hardware specific features

- Define **machines**: `conf/machine/*.conf`
- Provide Linux kernel recipes
- Provide bootloader recipes: U-Boot, Barebox, Grub... (`recipes-bsp`)
- Provide recipes to build firmware (`recipes-bsp`)
- Plus other customizations and configurations to standard software (usually `bbappends`)

Often contains `bsp` in the layer name.

```
meta-raspberrypi
├── classes
├── conf
├── COPYING.MIT
├── docs
├── dynamic-layers
├── files
├── img
├── kas-poky-rpi.yml
├── lib
├── README.md
├── recipes-bsp
├── recipes-connectivity
├── recipes-core
├── recipes-devtools
├── recipes-graphics
├── recipes-kernel
├── recipes-multimedia
├── recipes-sato
├── SECURITY.md
└── wic
```

U-Boot Recipes

Unlike Linux kernel recipes, there is no special class for U-Boot recipes yet

- You have to include includes from the standard U-Boot recipe
- No support for `defconfig` configuration either
  - Need to provide a patch to implement a specific configuration
  - To be chosen via the `UBOOT_MACHINE` setting

Mainline U-Boot Recipe Example

root commit

u-boot-mainline_2025.01.bb

```
require recipes-bsp/u-boot/u-boot-common.inc
require recipes-bsp/u-boot/u-boot.inc
PROVIDES = "virtual/bootloader"
RPROVIDES:${PN} = "u-boot"

# U-Boot 2025.01
SRCREV = "6d41f0a39d6423c8e57e92ebbe9f8c0333a63f72"
UBOOT_MACHINE = "custom_defconfig"

SRC_URI:append:genericarm64 = "file://add-custom-defconfig.patch"
SRC_URI:append:beaglebone-yocto = "file://add-custom-defconfig.patch"

DEPENDS += "bc-native dtc-native gnutls-native python3-pyelftools-native"
```

Source: Presentation at OE Workshop 2025 — https://rootcommit.com/pub/conferences/2025/oe-workshop/

Difference between `include` and `require`

- Both are used to "include" other files
- But only `require` aborts with an error when the file to include is not found.

Example Machine Definition (1)

root commit

conf/machine/beagleboard.conf — Part 1

```
#@TYPE: Machine
#@NAME: Beagleboard machine

PREFERRED_PROVIDER_virtual/xserver ?= "xserver-xorg"
MACHINE_EXTRA_RRECOMMENDS = "kernel-modules"
EXTRA_IMAGEDEPENDS += "virtual/bootloader"

DEFAULTTUNE ?= "cortexa8hf-neon"
include conf/machine/include/arm/armv7a/tune-cortexa8.inc

IMAGE_FSTYPES += "tar.bz2 jffs2 wic wic.bmap"
WKS_FILE ?= "beagleboard.wks"
MACHINE_ESSENTIAL_EXTRA_RDEPENDS += "kernel-image kernel-devicetree"
do_image_wic[depends] += "mtools-native:do_populate_sysroot dosfstools-native:do_populate_sysroot virtual/bootloader:do_deploy"

SERIAL_CONSOLES ?= "115200;ttyS2"
```

Example Machine Definition (2)

root commit

conf/machine/beagleboard.conf — Part 2

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-mainline"
PREFERRED_VERSION_linux-mainline ?= "6.13.11"
KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "ti/omap/omap3-beagle.dtb"

PREFERRED_PROVIDER_virtual/bootloader = "u-boot-mainline"
PREFERRED_VERSION_u-boot-mainline = "2024.07"
SPL_BINARY = "MLO"
UBOOT_SUFFIX = "img"

DTB_FILES = "omap3-beagle.dtb"
IMAGE_BOOT_FILES ?= "u-boot.${UBOOT_SUFFIX} ${SPL_BINARY} ${KERNEL_IMAGETYPE} ${DTB_FILES}"

MACHINE_FEATURES = "usbgadget usbhost vfat alsa"
```

Source: `https://www.youtube.com/shorts/w-N3yh5U8rw` (Yocto on BeagleBoard)

Machine definitions — Key takeaways

root commit

Place for some settings otherwise in `conf/local.conf`:

- `PREFERRED_PROVIDER`
- `PREFERRED_VERSION`
- Device tree and other bootloader settings

Other settings:

- Disk image layout: `WKS_FILE`
- `SERIAL_CONSOLES`
- Toolchain settings: `DEFAULTTUNE` and includes
- `MACHINE_FEATURES` (see next page)

List of hardware features supported by the machine:

- alsa, bluetooth, usbhost, keyboard, screen...
- Example for beagleplay-ti:

```
MACHINE_FEATURES="apm usbgadget usbhost vfat ext2 alsa pci efi screen gpu"
```

- They can be used by recipes
  - To control kernel configuration options
  - To define packages to install to an image:

meta/recipes-core/packagegroups/packagegroup-base.bb

```
${@bb.utils.contains("MACHINE_FEATURES", "alsa", "packagegroup-base-alsa", "", d)} \
```

  - To define configuration settings or dependencies:

meta/recipes-sato/matchbox-panel-2/matchbox-panel-2_2.12.bb

```
EXTRA_OECONF += " ${@bb.utils.contains("MACHINE_FEATURES", "acpi", "--with-battery=acpi", "",d)}"
```

Create Your Own MACHINE

root commit

Don't hesitate to create your own `MACHINE`

- To eliminate some settings in `conf/local.conf`
- You can include some other machine definitions or their includes in your SOC vendor BSP layer.
- Advice: easier not to redefine existing machine names.

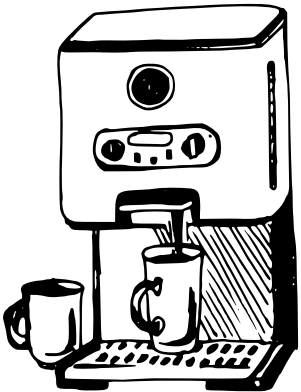


Image credits:
`https://openclipart.org/detail/183040/coffee-machine`

BSP Layers: Key Takeaways

For all hardware related features

- Create yours reusing the SoC vendor BSP layer
- Define machines in `conf/machine/<machine>.conf`
- Define kernel, bootloader, and firmware recipes
- Set preferred kernel and bootloader versions
- Include all hardware specific customizations of normal recipes

Lab — Create a New Machine

- Create a new `beagleplay-gaming` machine
- Remove settings from codeconf/local.conf

Layers

Images

Role of an Image

root commit

An image is supposed to describe which **packages** should be installed on the target root filesystem.

- Regardless of the `MACHINE` setting
- Regardless of the distribution choices (C library, init manager...)
- It should also describe the supported output formats: tar archive(s), disk image(s)...
- As well as the image size (including free space)

Layers

Image Recipes

Getting Image Contents

- `bitbake-getvar` doesn't help here,
  because images are defined by **packagegroups** (explained soon):

```
$ bitbake-getvar -r core-image-minimal IMAGE_INSTALL
# pre-expansion value:
#   "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL} u-boot"
IMAGE_INSTALL="packagegroup-core-boot  u-boot"
```

- The best way is to look at the contents of the image manifest in `tmp/deploy/images/<machine>/<image>-<machine>.rootfs.manifest`

```
base-files genericarm64 3.0.14
base-passwd armv8a 3.6.4
busybox armv8a 1.36.1
busybox-hwclock armv8a 1.36.1
busybox-syslog armv8a 1.36.1
busybox-udhcpc armv8a 1.36.1
eudev armv8a 3.2.14
grub-bootconf genericarm64 1.00
grub-common armv8a 2.12
grub-editenv armv8a 2.12
grub-efi armv8a 2.12
initt-ifupdown genericarm64 1.0
initt-system-helpers-service armv8a 1.66
initscripts armv8a 1.0
initscripts-functions armv8a 1.0
kbd armv8a 2.6.4
kernel genericarm64 6.13.11
kernel-6.13.11 genericarm64 6.13.11
kernel-image-6.13.11 genericarm64 6.13.11
kernel-image-image-6.13.11 genericarm64 6.13.11
keymaps genericarm64 1.0
kmod armv8a 33
ldconfig armv8a 2.40+git0+626c048f32
libblkid1 armv8a 2.40.2
libc6 armv8a 2.40+git0+626c048f32
libcrypto3 armv8a 3.3.1
libkmod2 armv8a 33
liblzma5 armv8a 5.6.2
libz1 armv8a 1.3.1
modutils-initscripts armv8a 1.0
netbase noarch 6.4
openssl-conf armv8a 3.3.1
openssl-ossl-module-legacy armv8a 3.3.1
packagegroup-core-boot genericarm64 1.0
sysvinit armv8a 3.04
sysvinit-inittab genericarm64 2.88dsf
sysvinit-pidof armv8a 3.04
ttyrun armv8a 2.34.0
u-boot genericarm64 2024.07
u-boot-env genericarm64 2024.07
update-alternatives-opkg armv8a 0.7.0
update-rc.d noarch 0.8+git0+b8f9501050
```

Manifest for `core-image-minimal` + `u-boot` on release Styhead

Standard Images

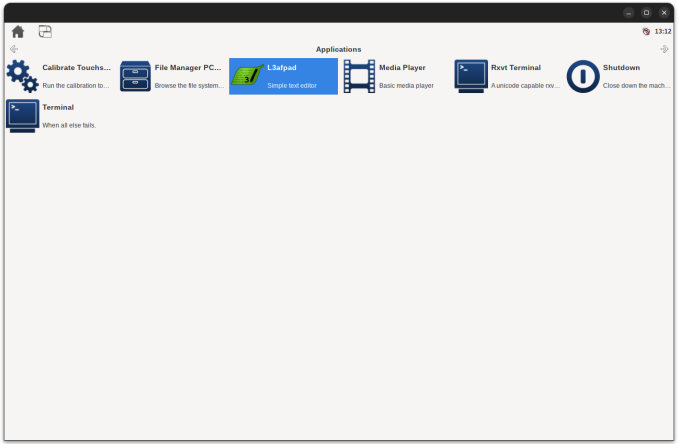
Image sizes given for Styhead with the `genericarm64` machine

- `core-image-minimal`: a minimal image just allowing to boot a machine. Mainly kernel (without modules), BusyBox and a few other utilities (62 MB, 42 packages)
- `core-image-minimal-dev`: + development packages (headers) to develop applications (84 MB, 83 packages)
- `core-image-minimal-initramfs`: - kernel + `udev` and utilities for use in an initramfs (33 MB, 38 packages)
- `core-image-base`: minimal base + kernel modules + associated userspace tools (172 MB, 876 packages)

Additional images, mainly for testing:

- `core-image-weston`: basic Wayland image with basic graphical applications (369 MB, 1046 packages)
- `core-image-x11`: basic X11 image (364 MB, 1066 packages)
- `core-image-sato`: outdated Sato multimedia desktop image, but useful for testing (448 MB, 1504 packages)

Sato Desktop Screenshot



Definitely useful only for testing 🤣

Images: Useful Variables

Images are defined as regular recipes. Example: `meta/recipes-core/images/core-image-minimal.bb`

- `IMAGE_BASENAME`: base name of the image output file. Defaults to `${PN}`.
- `IMAGE_INSTALL`: list of packages and package groups to include in the image

  ```
  IMAGE_INSTALL += "openssh"
  ```

- `IMAGE_FEATURES`: list of features features to include in an image (see next page)

  ```
  IMAGE_FEATURES += "package-management"
  ```

- `IMAGE_LINGUAS`: list of locales to install in an image

  ```
  IMAGE_LINGUAS = "pt-br de-de"
  ```

- `IMAGE_ROOTFS_SIZE`: size in KB for the generated image. See also `IMAGE_ROOTFS_EXTRA_SPACE` and `IMAGE_OVERHEAD_FACTOR`

  ```
  IMAGE_ROOTFS_SIZE = "262144"
  ```

- `IMAGE_FSTYPES`: Output image file formats

  ```
  # From conf/machine/beaglebone-yocto.conf
  IMAGE_FSTYPES += "tar.bz2 jffs2 wic wic.bmap"
  ```

- `EXTRA_IMAGEDEPENDS`: list of recipes not meant for installing into the root filesystem. Typical case: bootloader.

  ```
  # From meta/conf/machine/qemuriscv64.conf
  EXTRA_IMAGEDEPENDS += "u-boot"
  ```

- `IMAGE_POSTPROCESS_COMMAND`: list of functions to call after generating the image

  ```
  IMAGE_POSTPROCESS_COMMAND +=
        "buildhistory_get_imageinfo"
  ```

Image Features

Correspond to features impacting the creation of the image, or the way some recipes are built.

- `allow-empty-password`: allows Drop bear and OpenSSH to accept logins with an empty password
- `allow-root-login`: allows Dropbear and OpenSSH to accept root logins
- `empty-root-password`: speaks for itself
- `post-install-logging`: logs the execution of package post install scripts
- `debug-tweaks`: shortcut for the above features. Removed in Walnascar (5.2), to avoid the risk to keep it in production.
- `dbg-pkgs`: installs debug symbol packages
- `package-management`: installs package management tools and database. See `PACKAGE_CLASSES`
- `read-only-rootfs`: sets up a read-only root filesystem
- `splash`: enable a splashscreen during boot
- `overlayfs-etc`: configures the `/etc` directory to be in overlayfs (read-only root filesystem)

To be set through `IMAGE_FEATURES` (image recipes) or through `EXTRA_IMAGE_FEATURES` (configuration files)

Image Recipe Example

root commit

meta/recipes-core/images/core-image-minimal.bb (Apr. 2025)

```
SUMMARY = "A small image just capable of allowing a device to boot."

IMAGE_INSTALL = "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"

IMAGE_LINGUAS = " "

LICENSE = "MIT"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"
IMAGE_ROOTFS_EXTRA_SPACE:append = "${@bb.utils.contains("DISTRO_FEATURES", "systemd", " + 4096",
"", d)}"
```

LICENSE is optional in image recipes.

Layers

Package Groups

Package Groups

- Package Groups are also regular recipes (`LICENSE` optional)
- They get package installed by setting `RDEPENDS` and `RRECOMMENDS`

Package Group Example

root commit

meta/recipes-core/packagegroups/packagegroup-core-boot.bb (Apr. 2025)

```
SUMMARY = "Minimal boot requirements"
DESCRIPTION = "The minimal set of packages required to boot the system"
PACKAGE_ARCH = "${MACHINE_ARCH}"

inherit packagegroup

EFI_PROVIDER ??= "grub-efi"
SYSVINIT_SCRIPTS = "${@bb.utils.contains('MACHINE_FEATURES', 'rtc', '${VIRTUAL-RUNTIME_base-utils-hwclock}', '', d)} \
                    modutils-initscripts \
                    ${VIRTUAL-RUNTIME_initscripts} \
                   "
RDEPENDS:${PN} = "\
    base-files \
    base-passwd \
    ${VIRTUAL-RUNTIME_base-utils} \
    ${@bb.utils.contains("DISTRO_FEATURES", "sysvinit", "${SYSVINIT_SCRIPTS}", "", d)} \
    ${@bb.utils.contains("MACHINE_FEATURES", "keyboard", "${VIRTUAL-RUNTIME_keymaps}", "", d)} \
    ${@bb.utils.contains("MACHINE_FEATURES", "efi", "${EFI_PROVIDER} kernel", "", d)} \
    netbase \
    ${VIRTUAL-RUNTIME_login_manager} \
    ${VIRTUAL-RUNTIME_init_manager} \
    ${VIRTUAL-RUNTIME_dev_manager} \
    ${VIRTUAL-RUNTIME_update-alternatives} \
    ${MACHINE_ESSENTIAL_EXTRA_RDEPENDS}"
...
```

Layers

Making Disk Images

.wks files

root commit

- Disk images are typically created by a tool called **Wic**
- They are generated from specifications in a `.wks` or `.wks.in` file

meta-ti-bsp/wic/sdimage-2part.wks

```
# short-description: Create SD card image with 2 partitions
# long-description: Creates a partitioned SD card image for TI platforms.
# Boot files are located in the first vfat partition with extra reserved space.

part --source bootimg-partition --fstype=vfat --label boot --active --align 1024 --use-uuid --fixed-
size 128M
part / --source rootfs --fstype=ext4 --label root --align 1024 --use-uuid
```

- Use `WKS_FILE` to select such a file:

meta-ti-bsp/conf/machine/include/k3.inc

```
WKS_FILE ?= "${@bb.utils.contains("MACHINE_FEATURES", "efi", "sdimage-2part-efi.wks.in", "sdimage-
2part.wks", d)}"
```

- You can override `WKS_FILE` in your own layer.

Images — Key Takeaways

- Image contents are sets of **packages**
- Specified by `IMAGE_INSTALL` and **package groups**
- Images and package groups are defined by regular recipes
- `IMAGE_FEATURES` alters the creation of the image (like allowing for empty root password), and the way some recipes are built.
- You can create your own partition scheme through a custom `WKS_FILE`

Lab — Create your own image

- Create a new `core-image-games` image
- Move some settings from `conf/local.conf`
- Add space to your root partition

Layers

Distro Layers

Purpose of Distro Layers

root commit

Typically for everything that neither belongs in a BSP layer nor in an image recipe 😉

- The choice of the C library: Glibc, Musl
- The choice of the Init Manager: SystemV Init, systemd, BusyBox init...
- The choice of Package management system: `rpm`, `deb`, `ipk`
- The choice of display server: X11, Wayland...
- Anything that can implement a distribution policy: splash screen, init scripts, image signing keys and scripts, some preferred versions, security settings, SBoM generation, QA checks, exclusion of GPLv3 components...

A distro should work whatever the machine and image contents.

How to set the distro?

root commit

- The distro is meant to be set via the `DISTRO` variable in `conf/local.conf`
- There needs to be a matching `conf/distro/<DISTRO>.conf` file in a layer
- If `DISTRO` is `""`, then `meta/conf/distro/defaultsetup.conf` is used.

The Poky distro

root commit

- `poky`: default distro with SystemV Init and RPM packages
- `poky-altcfg`: distro with systemd and IPK packages
- `poky-tiny`: distro with SystemV Init, IPK packages, `-Os` optimizations and some features and packages disabled

```
WARNING: Poky is a reference Yocto Project distribution that should be used for
testing and development purposes only. It is recommended that you create your
own distribution for production use.
```

What's wrong with Poky in production?

root commit

- Poky is made to test a wide variety of features and packages.
  In an embedded system, you usually want to reduce the footprint and attack surface.
- Poky has many debug, development and test features enabled. It basically enabled everything!
- Poky is not secure by default, doesn't have a hardened kernel, doesn't use a secure init manager, runs services as root (for example `lighttpd`). However, it includes hardened compiling options.
- Poky still uses the outdated SysV Init manager.

Similar to MACHINE_FEATURES and IMAGE_FEATURES, impacting the way components are configured

- Some examples: alsa, bluetooth, ipv4, ipv6, pulseaudio, nfs, wayland, x11...
- There is some overlap with MACHINE_FEATURES and IMAGE_FEATURES:
  - bluetooth is in DISTRO_FEATURES, causing applications to be configured with bluez library support
  - bluetooth is also in MACHINE_FEATURES, meaning the kernel should have Bluetooth support and the Bluetooth modules need to be included too.
  - You could have a machine that supports Bluetooth but a distribution choosing not to enable it.
  - See COMBINED_FEATURES for specifying features both in DISTRO_FEATURES and in MACHINE_FEATURES

Toolchain related options

root commit

- `TCLIBC`: allows to choose the C library: `glibc` (default), `musl`, `newlib` or `baremetal`. Not all packages support options other than `glibc`.
- `TCMODE`: can be used to use an external toolchain, but this reduces reproducibility and traceability
- `TC` means "ToolChain"!

Init Manager

`INIT_MANAGER`: important distribution setting. Several choices:

- `sysvinit`: traditional init system based on scripts but outdated. Poky's default.
- `systemd`: modern init system with Udev for hardware event management, powerful security capabilities and other features. Starts more services by default (impact on boot-time unless tweaked).
- `mdev-busybox`: BusyBox init system with `mdev` for hardware event management. Lightweight and often sufficient for simple and low-resource systems.

See `https://docs.yoctoproject.org/dev-manual/init-manager.html#init-manager`

Create your own distribution

- Once again, Poky is not recommended in production
- Create a new one in your layer: conf/distro/<distro>.conf
- Can use the TEMPLATECONF variable to provide custom templates for conf/local.conf and conf/bblayers.conf
- Then, set the same DISTRO setting in conf/local.conf:

```
DISTRO = "geniux"
```

https://github.com/carlesfernandez/meta-gnss-sdr/blob/master/conf/distro/geniux.conf

```
DISTRO = "geniux"

DISTRO_NAME = "Geniux"
DISTRO_CODENAME = "master"
DISTRO_VERSION = "${DISTRO_CODENAME}-25.04.${GENIUX_CONF_VERSION}"

SDK_VENDOR = "-geniuxsdk"
SDK_VERSION = "${DISTRO_VERSION}"
MAINTAINER = "cfernandez@cttc.es"

TARGET_VENDOR = "-geniux"
```

Distro include files

Your distro can include standard include files:

Contents of `meta/conf/distro/include`

cve-extra-exclusions.inc	lto.inc	tclibc-musl.inc
default-distrovars.inc	maintainers.inc	tclibc-newlib.inc
default-providers.inc	no-gplv3.inc	tclibc-picolibc.inc
default-versions.inc	no-static-libs.inc	tcmode-default.inc
distro_alias.inc	ptest-packagelists.inc	time64.inc
init-manager-mdev-busybox.inc	rust_security_flags.inc	uninative-flags.inc
init-manager-none.inc	security_flags.inc	yocto-space-optimize.inc
init-manager-systemd.inc	tclibc-baremetal.inc	yocto-uninative.inc
init-manager-sysvinit.inc	tclibc-glibc.inc	

See `meta/conf/distro/defaultsetup.conf`

Distro layers: key takeaways

- The distribution sets policies for how your distribution starts, what C library it uses, the features it supports, how packages are generated...
- Poky is meant for Yocto testing, not for production

Lab — Create your own distro

root commit

- Create new distro — Ditch Poky
- Try BusyBox init
- Switch to systemd
- Change login message

Layers

BSP vs Distro vs Image

Where to go for...

BSP Layers

- Machine definitions
- Kernel recipes and config
- Bootloader recipes
- Firmware recipes
- Custom Hardware utilities
- Instruction set definitions
- Machine features
(supported HW drivers)
- Machine specific bbappends

Images

- List of packages
- Package groups
- Image size and free space
- Partitioning scheme
- Image features
(root login...)

Distro Layers

- Package policies
- Toolchain
- C standard library
- Init manager
- Distro features
(ipv 6 , sound server...)

Suggestion: create https://OpenLayerMap.org

Yocto in Projects

Yocto in Projects

Binary Distributions

Yocto and Binary Distributions

Binary distributions:

- Like standard GNU/Linux distributions on desktop and servers
- Can be updated through packages. In most cases, no need to reboot
- Don't need a full reflash
- Packages can be added and removed too
- Support preserving modified configuration files across updates

Yocto supports generating binary distributions:

- But not enabled by default
- Not all built packages are installed in the image
- Supported package formats: `rpm`, `deb` and `ipk`
- Can generate **package feeds**
- Best known OE/Yocto built distro: Angstrom (defunct)

Available Package Formats

rpm

- Users: Fedora, Red Hat, Poky (by default)
- Low level tool: `rpm`
- Front-end: `yum`
- Test:
  - 1.3 GB of packages
  - 45 MB .tar.bz2 image

deb

- Users: Ubuntu, Debian, Poky (non-default)
- Low-level tool: `dpkg`
- Front-end: `apt`
- Test:
  - 1.1 GB of packages
  - 27 MB .tar.bz2 image

ipk

- Users: OpenWRT, Poky (non-default)
- Simplified version of deb
- Maintained by Yocto
- Only one tool: `opkg`
- Test:
  - 1.8 GB of packages
  - 19 MB .tar.bz2 image

Test: core-image-minimal, poky master (Sep. 11, 2024), with package management

Why are ipk packages bigger than deb ones?

root commit

- deb package contents are compressed with `xz`
- ipk package contents are compressed with `zstd`
- `zstd` compresses less than `xz`, but is much less CPU intensive. Much better for low-end CPUs.

Choosing the package format

root commit

Set PACKAGE_CLASSES in conf/local.conf or in distro.conf:

```
PACKAGE_CLASSES ?= "package_deb"
PACKAGE_CLASSES ?= "package_deb package_ipk package_rpm"
```

Though packages are generated for all PACKAGE_CLASSES, only the first setting is actually used to generate the image.

Enabling package management in image

root commit

- Though OpenEmbedded uses packages to install applications and other files, by default there is no package manager on Poky's `core-image-minimal` image.
- If you want to be able to use package management at run time:
  - Add to `conf/local.conf`:

    ```
    EXTRA_IMAGE_FEATURES += "package-management"
    ```
  - Or to an image recipe:

    ```
    IMAGE_FEATURES += "package-management"
    ```
- See `EXTRA_IMAGE_FEATURES` and `IMAGE_FEATURES`.

Create a package feed

root commit

- Package feeds are created automatically in `tmp/deploy/[rpm|deb|ipk]` when you build packages
- The package indexes (catalogs of packages and versions) are not created by default. You need to create them with:

```
$ bitbake package-index
```

Publish a package feed

root commit

- Your package feed contents are in `tmp/deploy/<format>`
- You may copy that to a directory shared by a web server
- For development and testing, the quickest way is to run a local server from the command line. No need to set up an Apache server! 😏

```
$ cd tmp/deploy/ipk/
$ python3 -m http.server
```

This starts an HTTP server on local TCP port 8000

Use a package feed

- You need to configure the package manager in the image to let it know the HTTP(S) server details.
- Set the `PACKAGE_FEED_URIS`, `PACKAGE_FEED_BASE_PATHS`, and `PACKAGE_FEED_ARCHS` variables in `conf/local.conf`

Example:

```
PACKAGE_FEED_URIS = "https://example.com/packagerepos/release \
                    https://example.com/packagerepos/updates"
PACKAGE_FEED_BASE_PATHS = "rpm rpm-dev"
PACKAGE_FEED_ARCHS = "all core2-64"
```

Given these settings, the resulting package feeds are as follows:

```
https://example.com/packagerepos/release/rpm/all
https://example.com/packagerepos/release/rpm/core2-64
https://example.com/packagerepos/release/rpm-dev/all
https://example.com/packagerepos/release/rpm-dev/core2-64
https://example.com/packagerepos/updates/rpm/all
https://example.com/packagerepos/updates/rpm/core2-64
https://example.com/packagerepos/updates/rpm-dev/all
https://example.com/packagerepos/updates/rpm-dev/core2-64
```

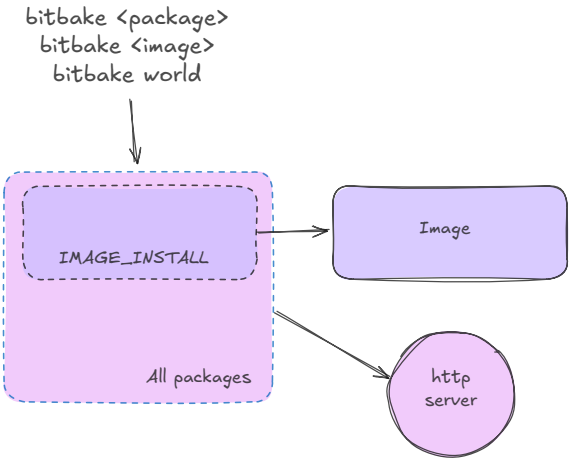
Image and package feed contents

- What goes into the image?
  - The list of packages defined by `IMAGE_INSTALL` and the image that you build:

    ```
    $ bitbake core-image-minimal
    ```

- What goes into the package feed?
  - The list of packages that you build:

    ```
    $ bitbake hello
    $ bitbake world
    ...
    ```



Package managers: quick reference

rpm

- Configuration:

  `/etc/yum.repos.d/`

- Commands:

```
dnf update
dnf install
dnf remove
dnf upgrade
```

deb

- Configuration:

  `/etc/apt`

- Commands:

```
apt update
apt list --upgradable
apt upgrade
```

ipk

- Configuration:

  `/etc/opkg`

- Commands:

```
opkg update
opkg install <package>
opkg remove <package>
opkg upgrade -noaction
opkg upgrade
```

PR value

- `PR` = Package Revision
- Only needed when applying package updates
- Example:
  - Currently installed package: `myapp-1.0-r0`
  - Available bugfix update: `myapp-1.0-r1`
- This makes sure that the update prevails and gets installed. Not necessary when there is a version number increase.

1.0-r0
PV-rPR

`https://docs.yoctoproject.org/ref-manual/variables.html#term-PR`

PR server

- A PR server is a process which increases the PR (revision) value when a new package output hash is found. Therefore, also needs a Hash Equivalence Server to work properly.
- Can either be a local server:

```
PRSERV_HOST = "localhost:0"
```

- Or a server shared by multiple builders:

```
PRSERV_HOST = "192.168.1.17:8585"
```

Hash Equivalence Server:
Detects when two instances of a task how different input hash but the same output hash. This could come from differences in comments or in other types of unused code.
This equivalence avoids propagating changes all the way down the dependency chain.

https://docs.yoctoproject.org/dev-manual/packages.html#working-with-a-pr-service

Binary Distributions — Key Takeaways

- BitBake / OE support generating binary distributions
- However, Poky images don't include package management by default: not possible to add and remove packages.
- Difference between built packages and those which are added to an image
- Enable package management with:

```
IMAGE_FEATURES += "package-management"
```

- Also need to generate package indexes for built packages:

```
$ bitbake package-index
```

- Then start an HTTP server to serve files in `tmp/deploy/[rpm|deb|ipk]`

Lab — Setting up a binary distribution

root commit

- Build a new package
- Enable package management
- Remove a package
- Generate a package feed
- Start an HTTP server
- Install a new package

Yocto in Projects

Addressing Vulnerabilities

Vulnerability landscape

root commit

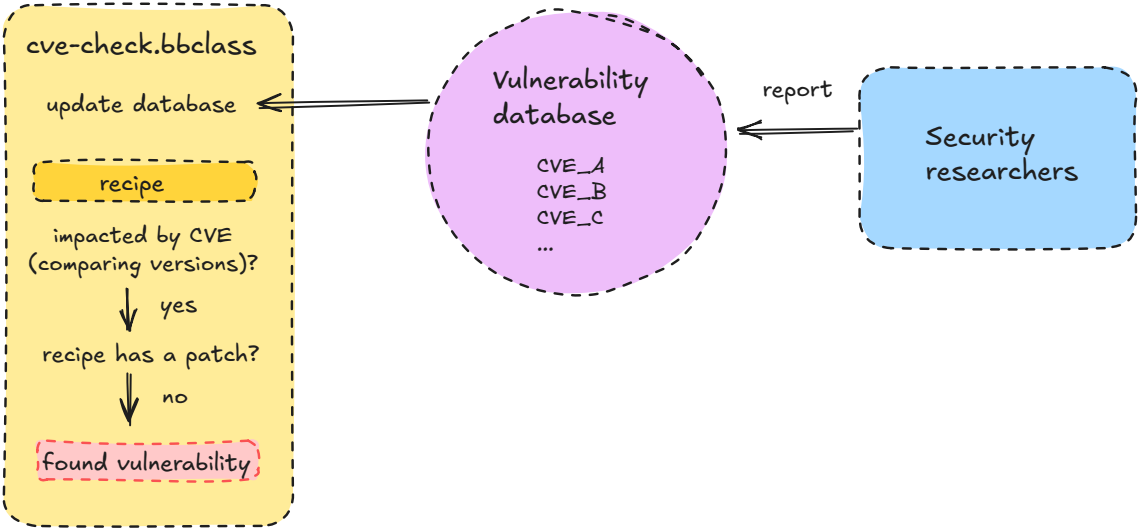
Constraints

- Don't want to leave a product with unfixed vulnerabilities. Could be very costly for your customers and your reputation.
- Embedded device makers have a tendency to "ship and run".
- Regulations (like the CRA in the EU) are going to make it mandatory to have proper security policies and vulnerability management.

A bumpy ride

- Vulnerability databases are originally maintained by US government bodies
- However, the quality of service has severely degraded since 2024 (stalled updates) for multiple reasons, especially budget cuts (in particular in 2025)
- The Yocto Project is doing its best to maintain working tools in this unstable environment.

How vulnerability checking works



Enable vulnerability checking

root commit

- Add this to `conf/local.conf`:
  ```
  INHERIT += "cve-check"
  ```
- This will add a CVE task to the recipes you're building
- You may also want to ignore CVEs that are irrelevant to Poky and OE-core:
  `include conf/distro/include/cve-extra-exclusions.inc`
- Then, `bitbake` your regular image,
  and the checks will be run (without running the other tasks if not necessary)
- You can also run checks on specific recipes:
  ```
  $ bitbake -c cve_check linux-yocto
  ```

Exploiting vulnerability report

- Simple command:

```
$ grep Unpatched /home/mike/work/yocto/poky/build/tmp/log/cve/cve-summary.json | wc
30
```

- You can also parse the JSON report with your own tooling
- Unfortunately, don't know good / standard tools to browse such vulnerabilities
  - VulnScout (`https://github.com/savoirfairelinux/vulnscout`) is promising but had issues making it work properly (ongoing)
  - No tools recommended in Yocto's documentation yet

Addressing vulnerabilities in your products

root commit

If a fix is found (typically upstream), add the patch to your recipe

- Include the CVE identifier in the patch file name (recommended)
- Add a `CVE:<id>` line to the patch
- Also set an `Upstream-Status:` field. `https://docs.yoctoproject.org/contributor-guide/recipe-style-guide.html#patch-upstream-status`
- Share your patch with the Yocto community!

Of course, another option is to upgrade to a newer version of upstream (if available).

meta/recipes-bsp/grub/files/CVE-2025-0622-01.patch

```
From 2123c5bca7e21fbeb0263df4597ddd7054700726 Mon Sep 17 00:00:00 2001
From: B Horn <b@horn.uk>
Date: Fri, 1 Nov 2024 19:24:29 +0000
Subject: [PATCH 1/3] commands/pgp: Unregister the "check_signatures" hooks on
 module unload

If the hooks are not removed they can be called after the module has
been unloaded leading to an use-after-free.

Fixes: CVE-2025-0622

Reported-by: B Horn <b@horn.uk>
Signed-off-by: B Horn <b@horn.uk>
Reviewed-by: Daniel Kiper <daniel.kiper@oracle.com>

CVE: CVE-2025-0622
Upstream-Status: Backport [https://git.savannah.gnu.org/cgit/grub.git/commit/?id=2123c5bca7e21fbeb...]
Signed-off-by: Peter Marko <peter.marko@siemens.com>
---
 grub-core/commands/pgp.c | 2 ++
 1 file changed, 2 insertions(+)

diff --git a/grub-core/commands/pgp.c b/grub-core/commands/pgp.c
index c6766f044..5fadc33c4 100644
--- a/grub-core/commands/pgp.c
+++ b/grub-core/commands/pgp.c
@@ -1010,6 +1010,8 @@ GRUB_MOD_INIT(pgp)

 GRUB_MOD_FINI(pgp)
 {
+  grub_register_variable_hook ("check_signatures", NULL, NULL);
+  grub_env_unset ("check_signatures");
   grub_verifier_unregister (&grub_pubkey_verifier);
   grub_unregister_extcmd (cmd);
   grub_unregister_extcmd (cmd_trust);
```

Ignoring vulnerabilities in your products

root commit

You can also modify the recipe to mark some vulnerabilities as irrelevant:

meta/recipes-devtools/rust/rust-source.inc

```
CVE_STATUS[CVE-2024-24576] = "not-applicable-platform: Issue only applies on Windows"
```

You can also group vulnerabilities that can be ignored in the same way:

meta/recipes-extended/logrotate/logrotate_3.22.0.bb

```
CVE_STATUS_GROUPS = "CVE_STATUS_RECIPE"
CVE_STATUS_RECIPE = "CVE-2011-1548 CVE-2011-1549 CVE-2011-1550"
CVE_STATUS_RECIPE[status] = "not-applicable-platform: CVE is debian, gentoo or SUSE specific on the way logrotate was installed/used"
```

Managing vulnerabilities — Key Takeaways

root commit

- Yocto hides the complexity of managing vulnerability database changes
- Add this to `conf/local.conf`:
  ```
  INHERIT += "cve-check"
  ```
- You will get CVE reports when you generate your image
- To check single recipe:
  ```
  $ bitbake -c cve_check linux-yocto
  ```
- Fix vulnerabilities by adding patches or marking issues as irrelevant

See the Yocto manual:
`https://docs.yoctoproject.org/dev-manual/vulnerabilities.html`

Lab — Managing Vulnerabilities

root commit

- Enable vulnerability checks
- Fetch a copy of the vulnerability database
- Reduce the vulnerability count
- Mark a vulnerability as ignored

Yocto in Projects

Software Supply Chain — Software Bill of Materials (SBoM)

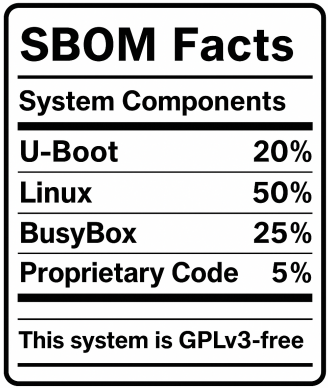
What's an SBoM?

A description of the software contents in a product

- Components, their versions and sources (hashes)
- Patches applied to fix vulnerabilities
- Licenses of components
- Dependencies between components
- Tools used to build the components
- Can include the full sources too

Two main open formats:

- SPDX (Linux Foundation): `https://spdx.dev/`
- CycloneDX (Open Worldwide Application Security Project): `https://cyclonedx.org/`



Usefulness of SBoM?

Vulnerability assessment

- Especially for your customers, who don't have the build system and could run checks based on just the versions of the components and their dependencies.
- In particular years after the product was released

License compliance

- SPDX SBoM itself already meets some of the requirements
- Allows to run all sorts of checks without having the build system.

Generating SBoM from Yocto

root commit

- Yocto now supports
  - Kirkstone (4.0), Scarthgap (5.0): SPDX 2.2
  - Styhead, Walnascar (5.1+): SPDX 3.0
  - CycloneDX not supported

- Except in version Kirkstone (4.0), toplevel SPDX output is generated by default, in `tmp/deploy/images/<machine>/<image>-<machine>.rootfs.spdx.json`
- Some individual SPDX files are available in `tmp/deploy/spdx`
- Add this to `conf/local.conf` to make the output human readable:

  ```
  SPDX_PRETTY = "1"
  ```

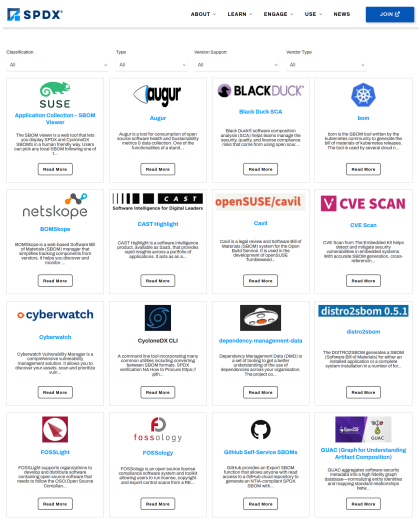
Consuming SPDX produced by Yocto

root commit

That's the fizzy part...

- The SPDX standard is recent
- "Jungle" of tools in `https://spdx.dev/use/spdx-tools/`
- Haven't heard about one tool gaining substantial adoption yet

That's why we don't have a lab on this topic yet 😏



Working with licenses

Free Software is not public domain, it also carries obligations, like:

- Keeping the copyright notices
- Sharing the original and modified sources ("copyleft" licenses)
- GPLv3 software: should allow users to run modified software on the device itself (very strong obligation!)

A few tricks:

- `LICENSE_FLAGS_ACCEPTED` may be necessary to build components that are not always suitable for all purposes, like Gstreamer "Ugly" plugins that may not be used in some countries because of software patents.

```
LICENSE_FLAGS_ACCEPTED = "commercial"
```

- `INCOMPATIBLE_LICENSE` allows to make sure software with specific licenses (such as GPLv3) is not built:

```
INCOMPATIBLE_LICENSE = "GPL-3.0* LGPL-3.0* AGPL-3.0*"
```

See `https://docs.yoctoproject.org/dev-manual/licenses.html`

SBoM and licenses — Key Takeaways

- Yocto now generates SPDX SBoM
- It contains everything you need to run vulnerability checks without the build system
- SPDX SBoM also helps with some license compliance requirements
- Tools for consuming SPDX are being developed but no "winner" has emerged yet.

Configuration Management

Managing Layers

root commit

People are using several tools to manage the various layers and versions in their projects.

- Google Repo — Originally created for Android
  `https://gerrit.googlesource.com/git-repo`
- Kas — Created by Siemens
  Dedicated tool for Yocto
  Fetches the right sources, sets configuration options and runs BitBake:
  Homepage: `https://github.com/siemens/kas`
  Documentation: `https://kas.readthedocs.io/en/latest/`

That's particularly useful to automate tasks, in particular in CI jobs!

Using Kas (1)

root commit

- First install Kas. Example on Ubuntu:

  ```
  $ sudo apt install kas
  ```

- Then create a YAML file describing your project, or use a provided one
- `target` is the recipe you want to build, typically an image
- Then build your project:

  ```
  $ mkdir build
  $ cd $build
  $ kas build project.yml
  ```

kas/examples/openembedded.yml

```yaml
header:
  version: 19

# Optionally provide keys to verify signed repositories
signers:
  YoctoBuildandRelease:
    fingerprint: 2AFB13F28FBBB0D1B9DAF63087EB3D32FB631AD9
    gpg_keyserver: keyserver.ubuntu.com

machine: qemux86-64
distro: poky

target: zlib-native

repos:
  poky:
    url: https://git.yoctoproject.org/poky.git
    # when specifying a tag, optionally provide a commit hash
    tag: yocto-5.1.1
    commit: 7e081bd98fdc5435e850d1df79a5e0f1e30293d0
    signed: true
    allowed_signers:
      - YoctoBuildandRelease
    layers:
      meta:
      meta-poky:
```

Using Kas (2)

- You can also include shared files through the `includes` section
- You can also specify one or multiple configurations through the `local_conf_header` section.

meta-mender-community/kas/beagleplay-ti.yml

```
header:
  version: 14
  includes:
  - kas/include/mender-full.yml
  - kas/include/arm.yml
  - kas/include/ti.yml

machine: beagleplay-ti

local_conf_header:
  beagleplay: |
    MENDER_FEATURES_ENABLE:append = " mender-image-sd"
    MENDER_FEATURES_DISABLE:append = "mender-image-uefi"
    MENDER_STORAGE_DEVICE = "/dev/mmcblk1"
    MENDER_BOOT_PART_SIZE_MB = "128"
    MENDER_PARTITION_ALIGNMENT = "1048576"
    IMAGE_FSTYPES:remove = "wic wic.bmap mender.bmap sdimg.bmap"
```

Kas override files

root commit

- You can add your own settings on top of a standard YAML file →
- And then run a command like:

```
$ kas build qemux86.yml:my-mender.yml
```

See quick Kas tutorial from Josef Holzmayr:
`https://hub.mender.io/t/using-kas-to-reproduce-your-yocto-builds/`

my-mender.yml

```
header:
  version: 13

local_conf_header:
  mender: |
    MENDER_SERVER_URL = "https://hosted.mender.io"
    MENDER_TENANT_TOKEN = "..."
```

Lab — Using Kas

root commit

- Turn your local layers into Git repositories
- Create a YAML description of your current project
- Regenerate it entirely with Kas

Yocto References

Books

Difficult to recommend books

- Several books are available
- But important to pick up a recent one, as many changes happened in the recent versions
- The most recent ones are pretty short and very expensive. The table of contents don't seem to go very deep either.

Image credits:
https://openclipart.org/detail/174860/bookworm-penguin

Yocto and BitBake Manuals

root commit

Our favorite parts:

- Yocto and BitBake Variable Index:
  `https://docs.yoctoproject.org/genindex.html`
- Development Tasks Manual:
  `https://docs.yoctoproject.org/dev-manual/`
  A gold mine for typical tasks!
- Reference Manual — Classes:
  `https://docs.yoctoproject.org/ref-manual/classes.html`
- Migration and Release Notes:
  `https://docs.yoctoproject.org/migration-guides/index.html`

Constantly updated by the Yocto Project — Contributions welcome too!

Further Online Resources

- Bootlin free Yocto training materials
  Also CC-BY-SA licensed. Michael Opdenacker contributed to them
  `https://bootlin.com/doc/training/yocto/`
- Yocto Project videos on YouTube:
  `https://www.youtube.com/@TheYoctoProject`
- Yocto Project on LinkedIn:
  `https://www.linkedin.com/company/yocto-project/`
- Yocto Project on Mastodon:
  `https://fosstodon.org/@yoctoproject`

Specific Resources

root commit

- Viktor Petersson's podcast: Inside the Yocto Project's Evolving Tools: SBOMs, SPDX 3.0 and Secure Embedded Systems
  Solid introduction to Yocto, and to software supply chain security in particular.
  `https://vpetersson.com/podcast/S02E09.html`

Conferences and Events

All announced on
`https://www.yoctoproject.org/community/events/`

- Yocto Project Summit
  Once a year, virtual event, usually in November
- OpenEmbedded Workshop
  Organized in Brussels in February right after FOSDEM
- Other events: dev-days and workshops

Image:
`https://www.yoctoproject.org/blog/2023/08/04/yocto-project-at-embedded-open-source-summit-2023/`

In Technical Conferences

- Embedded Linux Conference
  North America (Spring – Summer) and Europe (Summer – Autumn)
  Strong Yocto presence, often a booth too.
  `https://embeddedlinuxconference.com/`
- FOSDEM
  The biggest FOSS conference
  Brussels, February, free and available online
  Some Yocto talks and attended by many developers
  `https://fosdem.org`
- Embedded World
  Big trade show in March in Nuremberg, Germany
  A very well attended booth
  `https://www.embedded-world.de/`

Get Involved

The best way to hone your skills!

- Yocto and OpenEmbedded are a very welcoming community
- Check out our contributor guide: `https://docs.yoctoproject.org/contributor-guide/`
- Subscribe to our mailing lists: `https://www.yoctoproject.org/community/mailing-lists/`
- Join our weekly virtual meetings: `https://www.yoctoproject.org/community/get-involved/#virtual-meetings`

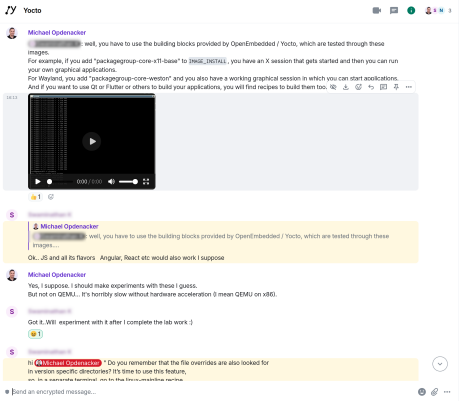
Image: https://www.flickr.com/photos/linuxfoundation/53053474040/sizes/h/

Stay in Touch

Thank you for participating to this course

- As a participant to our course, you have unlimited access to our Matrix chatroom.
  - Don't hesitate to ask questions from real life projects
- You may also be interested in other courses from Root Commit:
  - Embedded Linux
  - Linux Kernel, Board Support and Driver Development
  - Embedded Linux Boot Time Reduction

`https://rootcommit.com/training/`



Lab — Final Challenge: Media Player

root commit

- Fix all errors in a `meta-broken` layer
- See basic graphics in action: splashscreen, videoplayer
- Get codes to claim your completion certificate