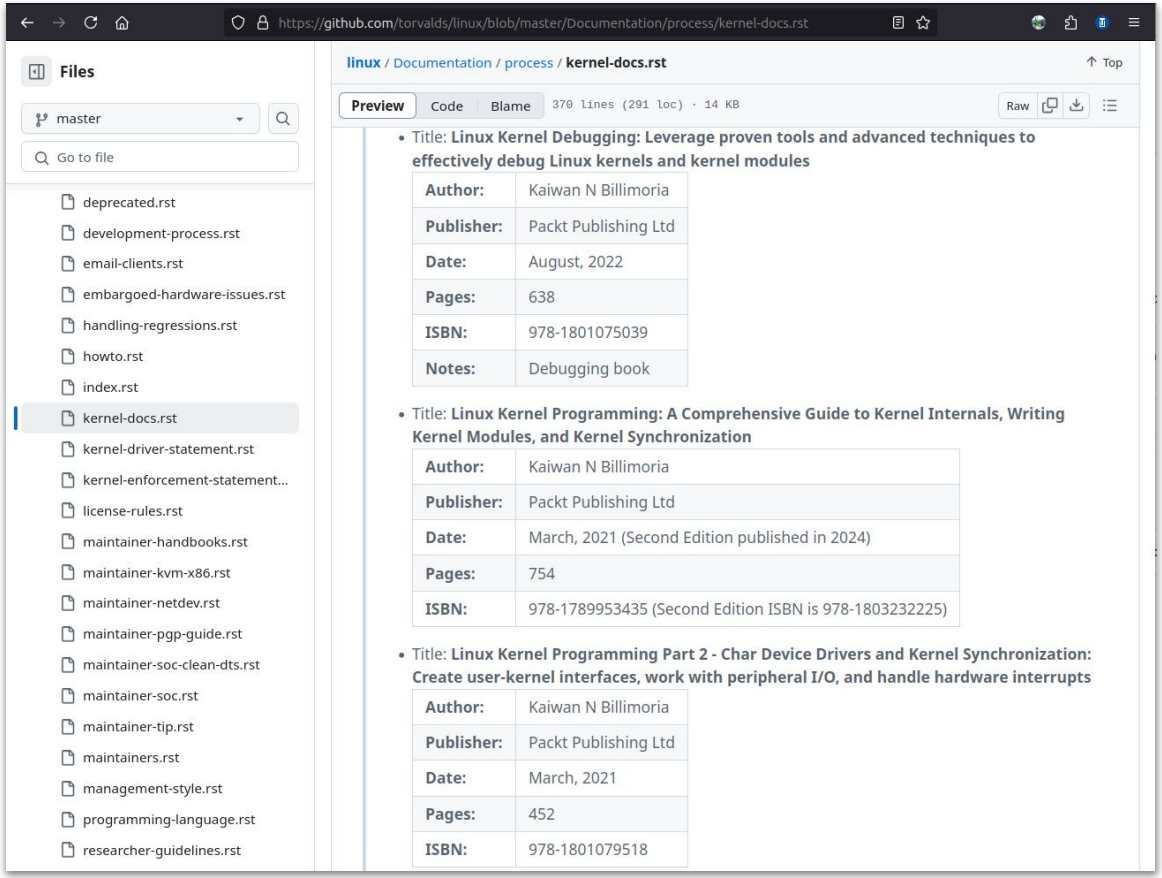
Hey Yocto, build me a custom embedded Linux! Er, no

An introduction to building a custom embedded Linux with the awesome Yocto Project

whoami

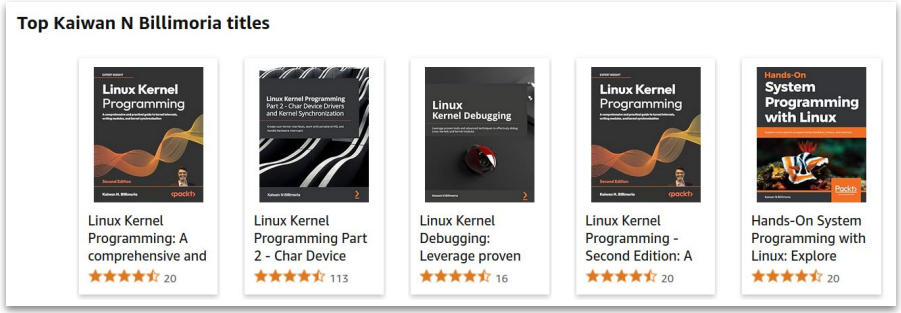
I'm Kaiwan, the author of a few of books on Linux (see next). Am happy that a few are listed in the *Published Books* section of the official kernel docs: [linux/Documentation/process/kernel-docs.rst at master]



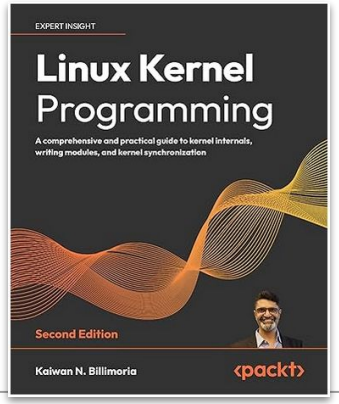
whoami

- Kaiwan N Billimoria: profile all-in-one : https://bit.ly/m/kaiwan

My Books!



★ LinkedIn public profile
★ My Amazon Author page
★ Corporate Training on Linux
  - Brief
  - More detailed
★ My GitHub repos
★ My tech blog



We provide both
- Corporate training, and
- Online Training to Individuals. Please visit https://kaiwantech.com

Agenda

- Intro to the Yocto Project
- Yocto releases
- What Yocto actually does…
- The Layer model - an overview
- Workspace setup (old way) and modern recommended >=5.3 way
- Demo embedded Linux system (Qemu AArch64) build and try
- The Layer-Recipe template - an overview
  - ‘Hello, world’ - A quick layer:recipe demo
- Pro Tips!
  - The actual Yocto philosophy: distro × image x machine , and many more!
- Appendix

GitHub repo for this presentation

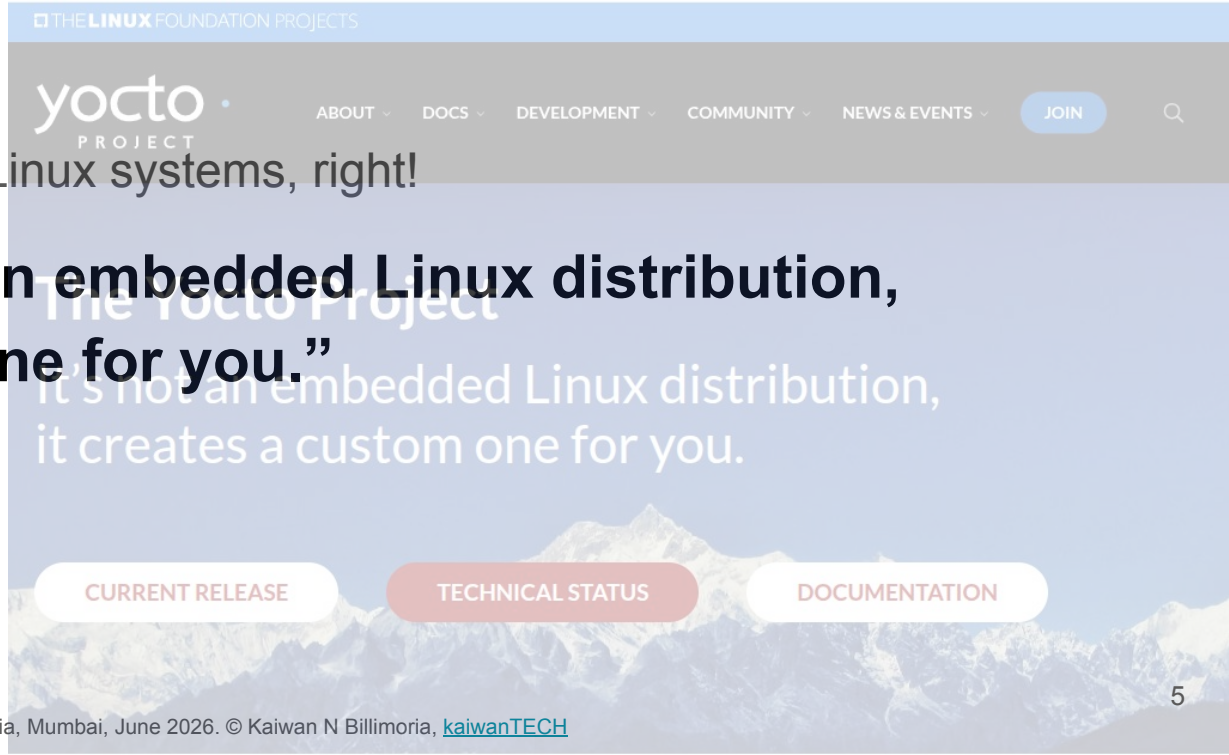
https://github.com/kaiwan/yocto_intro

(Within it are a demo layer:recipe, plus extra docs that go into (much) more detail - used in the *‘Embedded Linux with Yocto’* training programme.)

YP (Yocto Project) website: https://www.yoctoproject.org/

Why the name 'Yocto'?

- *yocto-* is the smallest SI unit prefix
- Represents 10^{-24}
- Hey, it's (also) meant to build small embedded Linux systems, right!



More correctly: **"It's not an embedded Linux distribution, it creates a custom one for you."**

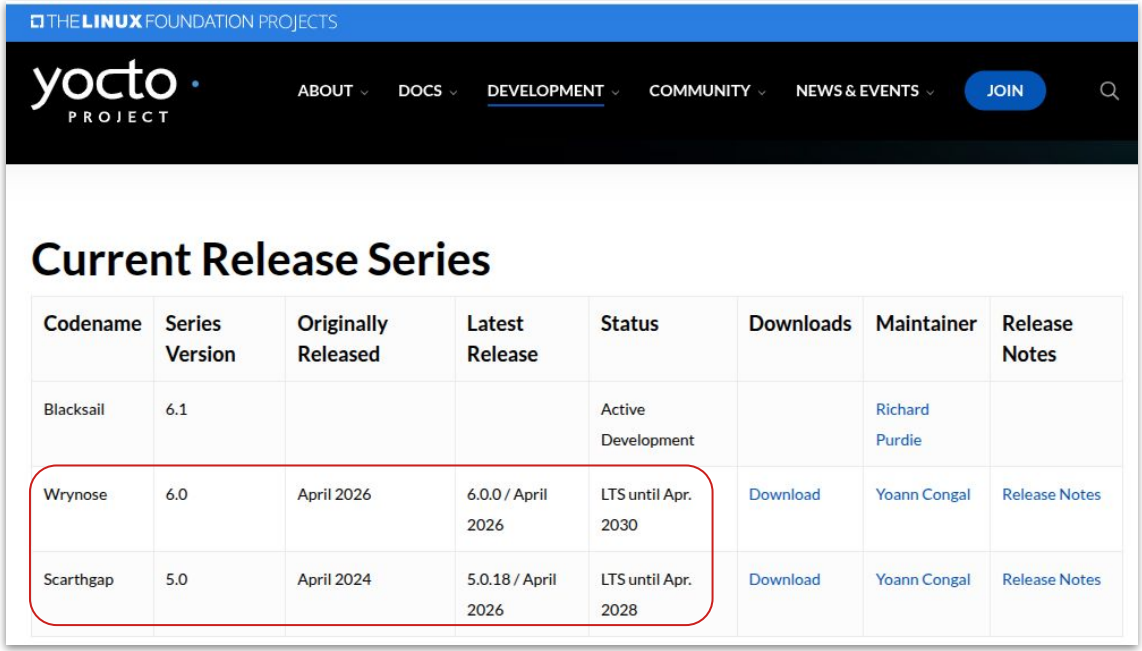
Yocto/OE Project

- 100% reproducible builds (no ‘magic script’ / ‘golden image’!)
- Many (most) known board’s - BSP layers are available
- QA tests builtin
- Details:
  - Features
  - Challenges

DO SEE these:

- The Yocto Project – introductory video 2m:12s (Oct 2011): *https://youtu.be/utZpKM7i5Z4*
- *YPS 2023.11 - 2023/11/30 - Talel BELHAJSALEM - Back to basics | The ultimate Yocto introduction* (1h08m) *- excellent!*

https://www.yoctoproject.org/development/releases/



Current Release Series

Codename	Series Version	Originally Released	Latest Release	Status	Downloads	Maintainer	Release Notes
Blacksail	6.1			Active Development		Richard Purdie	
Wrynose	6.0	April 2026	6.0.0 / April 2026	LTS until Apr. 2030	Download	Yoann Congal	Release Notes
Scarthgap	5.0	April 2024	5.0.18 / April 2026	LTS until Apr. 2028	Download	Yoann Congal	Release Notes

https://www.yoctoproject.org/development/releases/

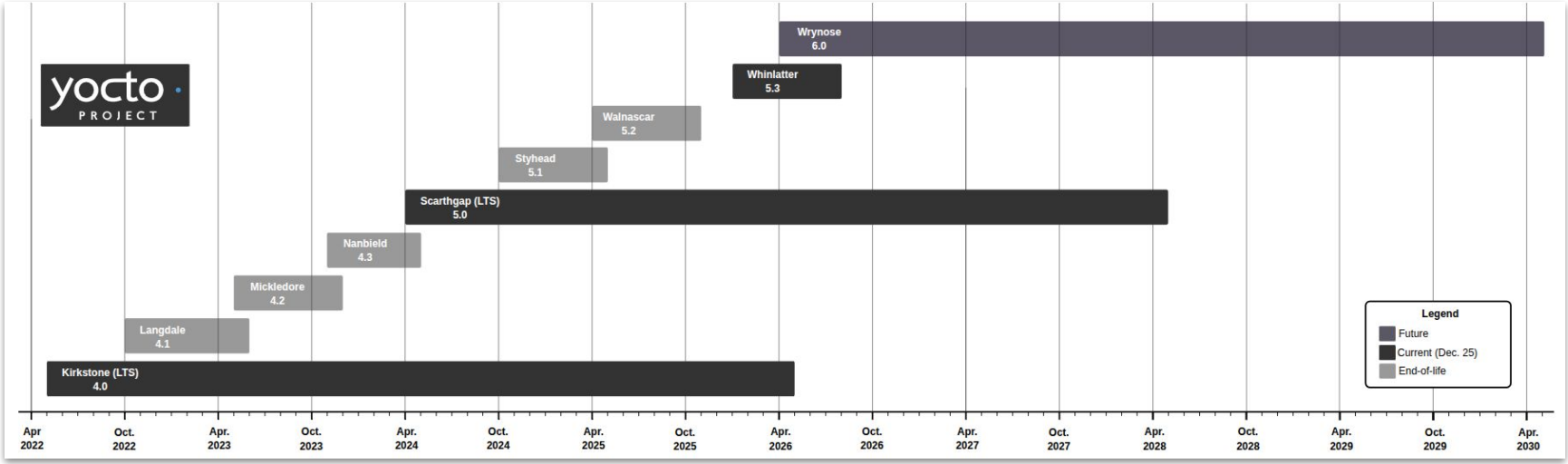
Previous Release Series

Codename	Series Version	Originally Released	Latest Release	Status	Downloads	Release Notes
Whinlatter	5.3	November 2025	5.3.4 / April 2026	EOL	Download	Release Notes
Walnascar	5.2	April 2025	5.2.4 / September 2025	EOL	Download	Release Notes
Styhead	5.1	September 2024	5.1.4 / March 2025	EOL	Download	Release Notes
Nanbield	4.3	October 2023	4.3.4 / March 2024	EOL	Download	Release Notes
Mickledore	4.2	April 2023	4.2.4 / October 2023	EOL	Download	Release Notes
Langdale	4.1	October 2022	4.1.4 / March 2023	EOL	Download	Release Notes
Kirkstone	4.0	April 2022	4.0.35 / April 2026	EOL	Download	Release Notes

Why's the Whinlatter v5.3 highlighted? As it's the first ver where Poky isn't the default distro! (We'll see…)

The dark black rectangles represent the current releases (as of Dec 2025); releases are on a 6 month cadence.

Src: https://docs.yoctoproject.org/dev/_images/releases.svg



What Yocto actually does

- Any and every Linux system must *minimally* have:
  - a BootLoader (BL) image
  - a kernel image
  - if ARM*/PPC/RiscV/etc, a Device Tree Blob (DTB) image
  - a root filesystem (rootfs) image
    - C library
    - init manager
    - utils

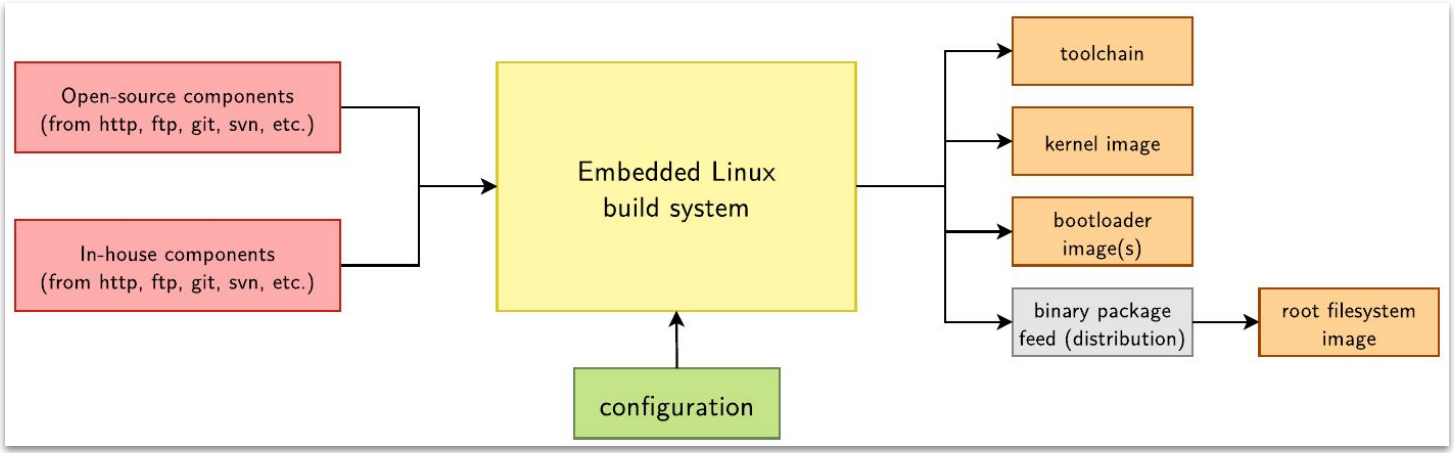
- The next two slides show that '**images**' are generated; it's the ones just mentioned!

- This is - in a nutshell - the job of Yocto: **to generate these finished images**, populating their content as specified (via the config metadata: image & distro recipes, local config and the layers & recipes)

- In order to build them, Yocto also generates the *toolchain* as required!
- They're flashed onto the target board, and voila, we (hopefully) have a running custom embedded Linux system!

What Yocto actually does

Simpler overall Yocto Project flow diagram (pic credit: Bootlin)

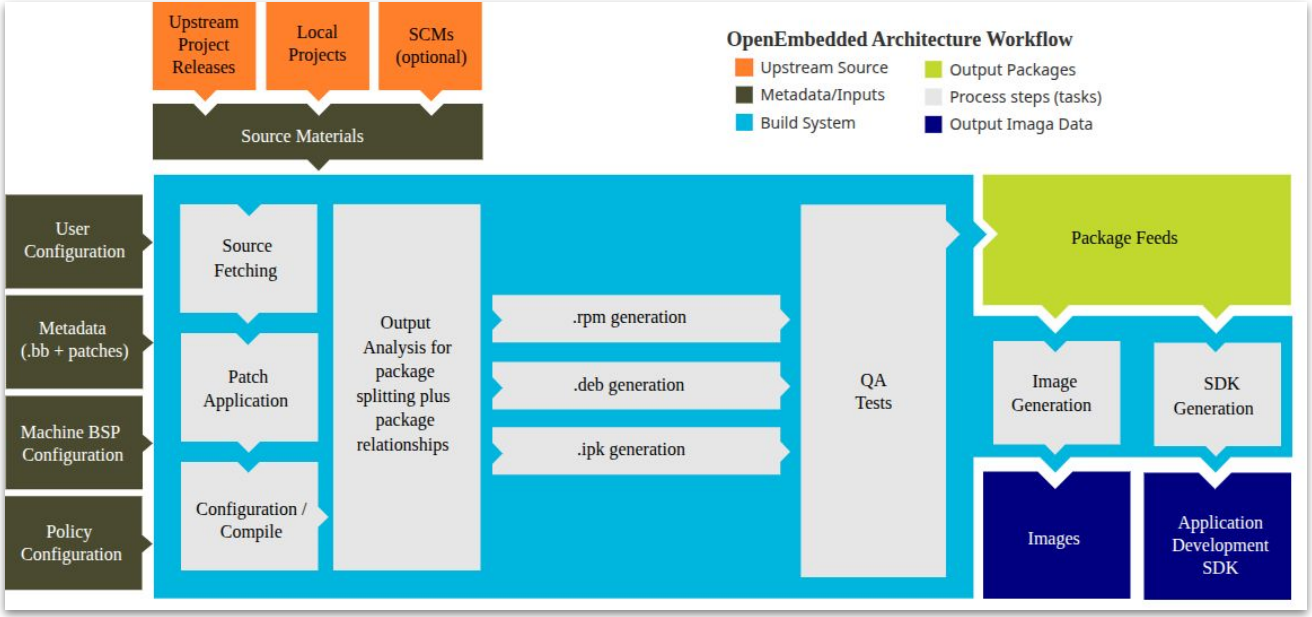


Ultimately, the Yocto Project (YP) builds **packages** (ipk/rpm/deb); the packages are then extracted to make bootable image(s)

What Yocto actually does

- The Yocto Project (YP) builds **packages** (ipk/rpm/deb); the packages are then extracted to make bootable image(s)
- ***Yocto : the canonical all-in-one diagram***:

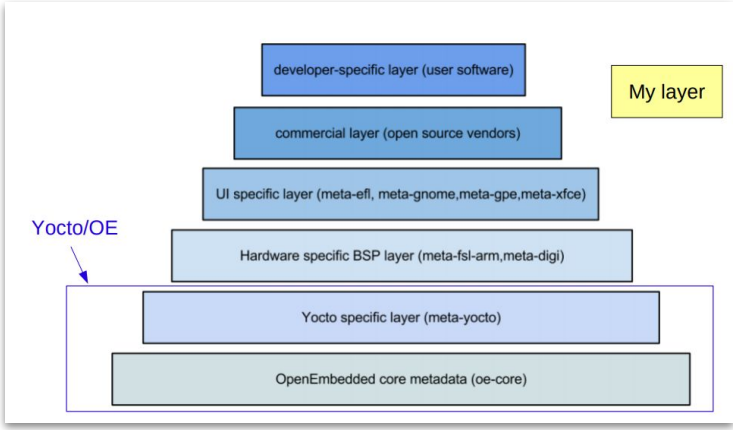
Q> How will the build system know things? Which machine (& thus toolchain) to compile for, bootloader, DTS, kernel, package versions, C lib, init manager, etc?
A> The 'configuration metadata' provides it…



The Layer Model

Source

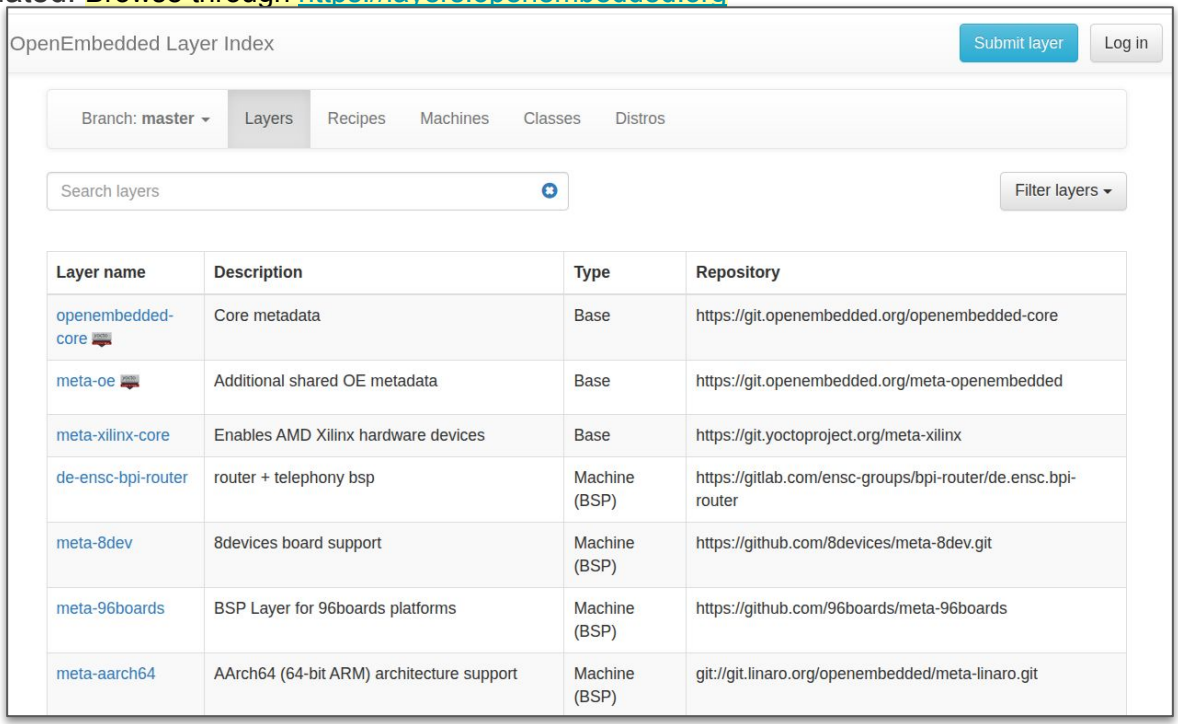
- [illegible]



Src

Familiarize yourself with the curated - tested and 'Yocto Project Compatible' certified - layers here: **YOCTO PROJECT COMPATIBLE LAYER INDEX** .

There is also the **OpenEmbedded layer index** which contains many more layers, recipes, BSPs, etc, but the content is less universally validated. Browse through https://layers.openembedded.org



Layer name	Description	Type	Repository
openembedded-core	Core metadata	Base	https://git.openembedded.org/openembedded-core
meta-oe	Additional shared OE metadata	Base	https://git.openembedded.org/meta-openembedded
meta-xilinx-core	Enables AMD Xilinx hardware devices	Base	https://git.yoctoproject.org/meta-xilinx
de-ensc-bpi-router	router + telephony bsp	Machine (BSP)	https://gitlab.com/ensc-groups/bpi-router/de.ensc.bpi-router
meta-8dev	8devices board support	Machine (BSP)	https://github.com/8devices/meta-8dev.git
meta-96boards	BSP Layer for 96boards platforms	Machine (BSP)	https://github.com/96boards/meta-96boards
meta-aarch64	AArch64 (64-bit ARM) architecture support	Machine (BSP)	git://git.linaro.org/openembedded/meta-linaro.git

Yocto workspace setup : first, the 'OLD' (Yocto < 5.3) way

1. Setup the host workspace: *link*

2. One-time: Clone the 'poky' repo: f.e.:

```
git clone -b scarthgap https://git.yoctoproject.org/poky
```

3. Initialize the work environment, the **"kitchen"** where the **"chef"** - BitBake - will cook! Run the init script - *every time:*

```
cd poky
source oe-init-build-env [build_dir]
```

Yocto workspace setup : old way, ***Poky*** *is always the default distro*

Example with Yocto 5.0.8 Scarthgap

```
$ alias ls='ls -F --color=auto'

$ ls yocto_qarm64_demo/
poky/

$ ls yocto_qarm64_demo/poky/
bitbake/        documentation/     LICENSE.MIT     meta/              meta-security/
meta-yocto-bsp/ README.md@         README.qemu.md build_qarm64/      LICENSE
MAINTAINERS.md  meta-openembedded/ meta-selftest/  oe-init-build-env  README.OE-Core.md  scripts/
contrib/           LICENSE.GPL-2.0-only  MEMORIAM     meta-poky/         meta-skeleton/
README.hardware.md@ README.poky.md@      SECURITY.md
```

In the ‘old’ way, everything’s under the ‘poky’ umbrella!

Really, let’s not!

Let's employ the modern (>= 5.3 Whinlatter) approach: where 'poky' is NOT the default distro!

- **Recommended** dir tree approach : essentially:
  - Keep a directory to hold all the upstream Yocto layers
  - Keep another directory to hold all your project's layers
- Setup via this simple Bash script:

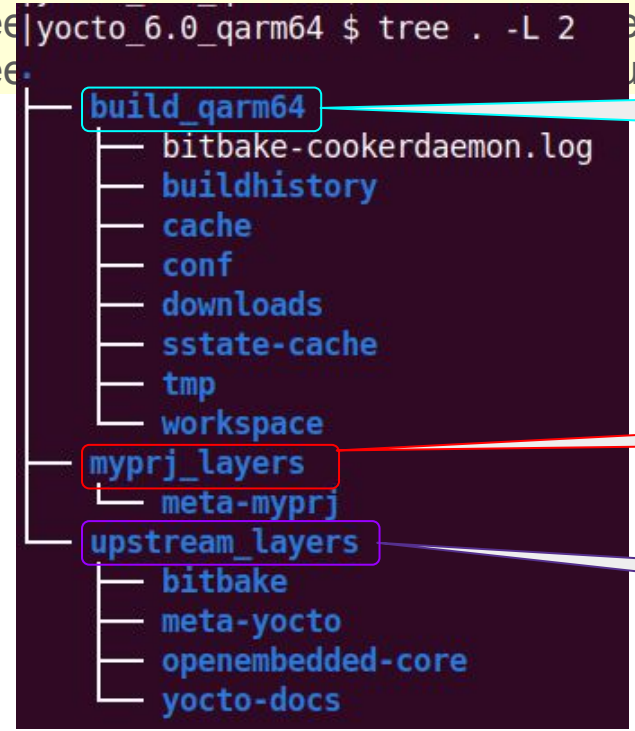
```
#!/bin/bash
# 'Correct' way to setup the Yocto >=5.3 (Whinlatter) layers directory layout
UPSTREAM_LAYERS_DIR=upstream_layers
[[ ! -d ${UPSTREAM_LAYERS_DIR} ]] && mkdir -p ${UPSTREAM_LAYERS_DIR}

git clone -b yocto-5.3.3 https://git.openembedded.org/bitbake ./${UPSTREAM_LAYERS_DIR}/bitbake
git clone -b yocto-5.3.3 https://git.openembedded.org/openembedded-core
./${UPSTREAM_LAYERS_DIR}/openembedded-core
git clone -b yocto-5.3.3 https://git.yoctoproject.org/meta-yocto ./${UPSTREAM_LAYERS_DIR}/meta-yocto
git clone -b yocto-5.3.3 https://git.yoctoproject.org/yocto-docs ./${UPSTREAM_LAYERS_DIR}/yocto-docs

# Create and maintain your own layers in another dir (which is version controlled, f.e. via git)
# f.e.
MYPRJ_LAYERS_DIR=myprj_layers
[[ ! -d ${MYPRJ_LAYERS_DIR} ]] && mkdir -p ${MYPRJ_LAYERS_DIR}
```

Hey, this is a simplified ver of the script; find the actual one here.

Let's employ the modern (>= 5.3 Whinlatter) approach: where 'poky' is NOT the default distro!

- **Recommended** dir tree approach : essentially:
  - Kee[illegible]eam Yocto layers
  - Kee[illegible]r project's layers

Only 2 levels seen:
tree . -L 2



The build dir ${BUILDDIR} (we'll come to it!)

Our project layers

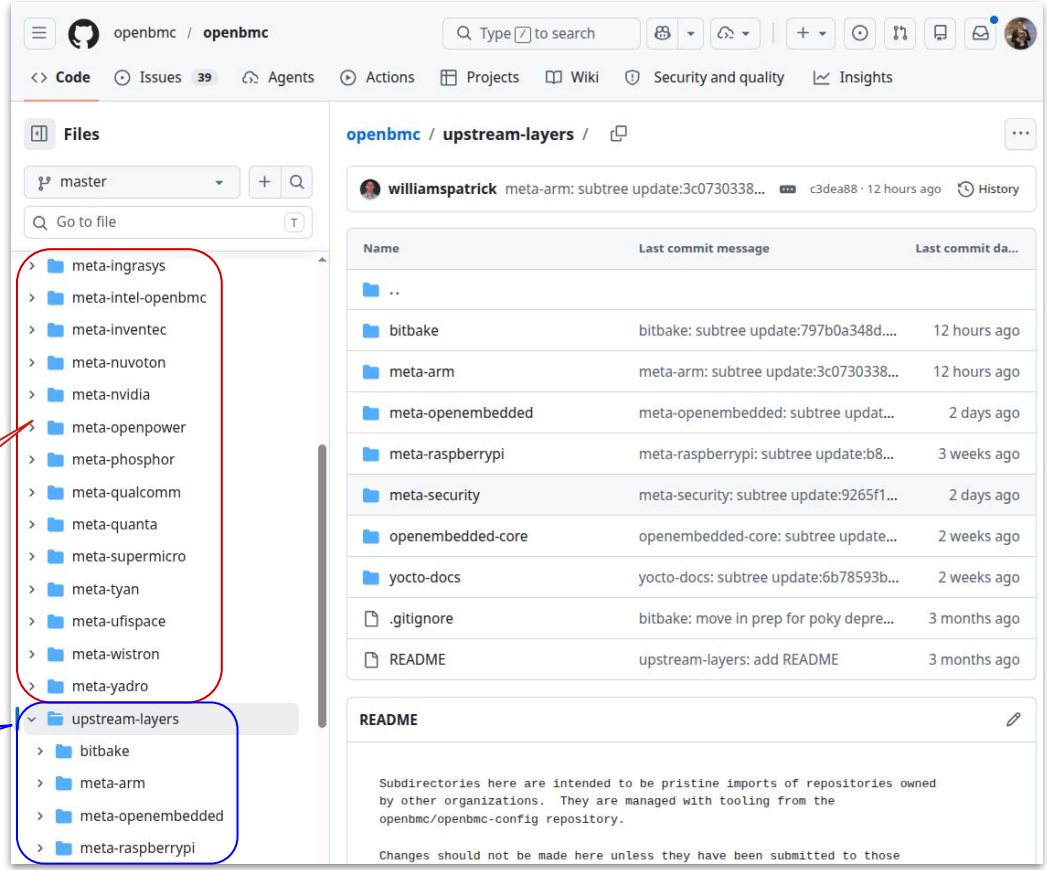
YP upstream layers

Let's employ the modern (>= 5.3 Whinlatter) approach: where 'poky' is NOT the default distro!

An example of a real-world project that does just this is the well known [*OpenBMC project!*]

openBMC layers

upstream Yocto layers



First init the work environment, the "kitchen" where the chef - BitBake! - will cook! It's the build dir, ${BUILDDIR}. How? Simple: just source a shell script (every time!):

```
|yocto_6.0_qarm64 $ source upstream_layers/openembedded-core/oe-init-build-env
build_qarm64/
```

An aside: use pre-downloaded Yocto artifacts for demo build without any internet download requirement.

To just fetch all required files – IOW, to just download everything required for the build – do this:

```
$ bitbake core-image-base --runall=fetch
…
```

core-image-base is the 'image recipe'

(Internally it performs only the *do_fetch()* task for every recipe. With the package(group) defaults for the *core-image-base* target image recipe, it downloaded close to 8 GB!)

The build …

```
$ time bitbake
    core-image-base
```

core-image-base is the ‘image recipe’

```
|build_qarm64 $ time bitbake core-image-base
Loading cache: 100% |################################################################################################
########################################| Time: 0:00:00
Loaded 1970 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "2.18.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "aarch64-oe-linux"
MACHINE              = "qemuarm64"
SDKMACHINE           = "x86_64"
DISTRO               = "nodistro"
DISTRO_VERSION       = "nodistro.0"
TUNE_FEATURES        = "aarch64 crc cortexa57"
meta                 = "HEAD:42fa856a00ac16b2a7a83d7ecfa60a5be192b16c"

Sstate summary: Wanted 2802 Local 0 Mirrors 0 Missed 2802 Current 352 (0% match, 11% complete)

Initialising tasks: 100% |###########################################################################################
########################################| Time: 0:00:02
NOTE: Executing Tasks
Setscene tasks: 3154 of 3154
Setscene tasks: 3154 of 3154
Currently 10 running tasks (1895 of 5838)  32% |#######################                                      |
0: xz-native-5.8.2-r0 do_configure - 21s (pid 211489)
1: gmp-native-6.3.0-r0 do_configure - 21s (pid 211510)
2: flex-native-2.6.4-r0 do_configure - 21s (pid 211517)
3: gperf-native-3.3-r0 do_configure - 21s (pid 211564)
4: tcl8-native-8.6.17-r0 do_compile - 12s (pid 224338)
5: libyaml-native-0.2.5-r0 do_configure - 6s (pid 233537)
6: gdbm-native-1.26-r0 do_compile - 4s (pid 237923)
7: cross-localedef-native-2.43+git-r0 do_compile - 4s (pid 237999)
8: libpcre2-native-10.47-r0 do_compile - 4s (pid 237985)
9: patch-native-2.8-r0 do_configure - 0s (pid 242146)
```

On my laptop (core i7, 12 cores, 32 GB RAM) Ubuntu 24.04.4 LTS:

- the build took ~ 2h 40m !
- … and ate up ~ 77 GB of disk space!
- … with defaults, and this image recipe (core-image-base), the target rootfs size is ~ 160 MB !

Pl use as powerful a build system as you can afford!

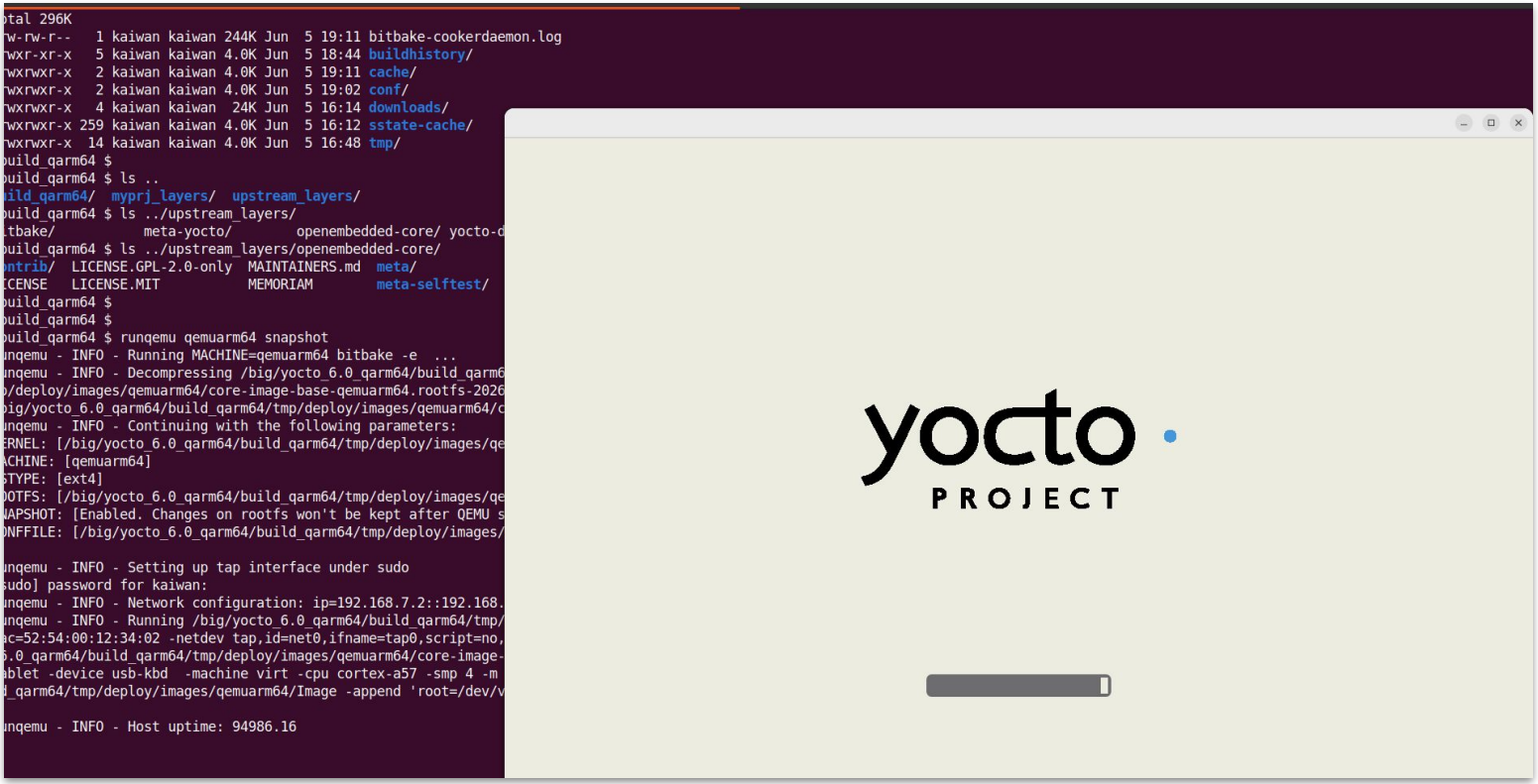
Build completed, here's the key *images/* dir:

tmp/deploy/images/qemuarm64/

It holds all the built artifacts - includes the bootloader, (possibly the DTB), kernel, root filesystem image files (among others)!

```
|build_qarm64 $ ls -lh tmp/deploy/images/qemuarm64/
total 134M
-rw-r--r-- 2 kaiwan kaiwan  53M Jun  5 18:07 core-image-base-qemuarm64.rootfs-20260605103542.ext4.zst
-rw-r--r-- 2 kaiwan kaiwan 4.9K Jun  5 18:07 core-image-base-qemuarm64.rootfs-20260605103542.manifest
-rw-r--r-- 2 kaiwan kaiwan 1.9K Jun  5 18:07 core-image-base-qemuarm64.rootfs-20260605103542.qemuboot.conf
-rw-r--r-- 2 kaiwan kaiwan 5.3M Jun  5 18:44 core-image-base-qemuarm64.rootfs-20260605103542.spdx.json
-rw-r--r-- 2 kaiwan kaiwan  53M Jun  5 18:07 core-image-base-qemuarm64.rootfs-20260605103542.tar.zst
-rw-r--r-- 2 kaiwan kaiwan 233K Jun  5 18:07 core-image-base-qemuarm64.rootfs-20260605103542.testdata.json
lrwxrwxrwx 2 kaiwan kaiwan   56 Jun  5 18:07 core-image-base-qemuarm64.rootfs.ext4.zst -> core-image-base-qemuarm64.rootfs-20260605103542.ext4.zst
lrwxrwxrwx 2 kaiwan kaiwan   56 Jun  5 18:07 core-image-base-qemuarm64.rootfs.manifest -> core-image-base-qemuarm64.rootfs-20260605103542.manifest
lrwxrwxrwx 2 kaiwan kaiwan   61 Jun  5 18:07 core-image-base-qemuarm64.rootfs.qemuboot.conf -> core-image-base-qemuarm64.rootfs-20260605103542.qemuboot.conf
lrwxrwxrwx 2 kaiwan kaiwan   57 Jun  5 18:44 core-image-base-qemuarm64.rootfs.spdx.json -> core-image-base-qemuarm64.rootfs-20260605103542.spdx.json
lrwxrwxrwx 2 kaiwan kaiwan   55 Jun  5 18:07 core-image-base-qemuarm64.rootfs.tar.zst -> core-image-base-qemuarm64.rootfs-20260605103542.tar.zst
lrwxrwxrwx 2 kaiwan kaiwan   61 Jun  5 18:07 core-image-base-qemuarm64.rootfs.testdata.json -> core-image-base-qemuarm64.rootfs-20260605103542.testdata.json
lrwxrwxrwx 2 kaiwan kaiwan   73 Jun  5 17:52 Image -> Image--6.18.24+git0+f94e250f9b_b1ba542851-r0-qemuarm64-20260605103542.bin
-rw-r--r-- 2 kaiwan kaiwan  23M Jun  5 17:52 Image--6.18.24+git0+f94e250f9b_b1ba542851-r0-qemuarm64-20260605103542.bin
lrwxrwxrwx 2 kaiwan kaiwan   73 Jun  5 17:52 Image-qemuarm64.bin -> Image--6.18.24+git0+f94e250f9b_b1ba542851-r0-qemuarm64-20260605103542.bin
-rw-r--r-- 2 kaiwan kaiwan 2.3M Jun  5 17:52 modules--6.18.24+git0+f94e250f9b_b1ba542851-r0-qemuarm64-20260605103542.tgz
lrwxrwxrwx 2 kaiwan kaiwan   75 Jun  5 17:52 modules-qemuarm64.tgz -> modules--6.18.24+git0+f94e250f9b_b1ba542851-r0-qemuarm64-20260605103542.tgz
|build_qarm64 $
```

// demo runs: **`$ runqemu qemuarm64 snapshot`**



// demo runs: alternately, in console mode:

```
$ runqemu \
  qemuarm64 snapshot nographic
```

```
[build_qarm64 $ runqemu qemuarm64 snapshot nographic
runqemu - INFO - Running MACHINE=qemuarm64 bitbake -e ...
runqemu - INFO - Decompressing /big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/core-image-base-qemuarm64.rootfs-20260605133234.ext4.zst to /big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/core-image-base-qemuarm64.rootfs-20260605133234.ext4
/big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/core-image-base-qemuarm64.rootfs-20260605133234.ext4.zst: 237587456 bytes
runqemu - INFO - Continuing with the following parameters:
KERNEL: [/big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/Image]
MACHINE: [qemuarm64]
FSTYPE: [ext4]
ROOTFS: [/big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/core-image-base-qemuarm64.rootfs-20260605133234.ext4]
SNAPSHOT: [Enabled. Changes on rootfs won't be kept after QEMU shutdown.]
CONFFILE: [/big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/core-image-base-qemuarm64.rootfs-20260605133234.qemuboot.conf]

runqemu - INFO - Setting up tap interface under sudo
runqemu - INFO - Network configuration: ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8 net.ifnames=0
runqemu - INFO - Running /big/yocto_6.0_qarm64/build_qarm64/tmp/work/x86_64-linux/qemu-helper-native/1.0/recipe-sysroot-native/usr/bin/qemu-system-aarch64 -device virtio-net-pci,netdev=net0,mac=52:54:00:12:34:02 -netdev tap,id=net0,ifname=tap0,script=no,downscript=no -object rng-random,filename=/dev/urandom,id=rng0 -device virtio-rng-pci,rng=rng0 -drive id=disk0,file=/big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/core-image-base-qemuarm64.rootfs-20260605133234.ext4,if=none,format=raw -device virtio-blk-pci,drive=disk0 -device qemu-xhci -device usb-tablet -device usb-kbd  -machine virt -cpu cortex-a57 -smp 4 -m 256 -snapshot -serial mon:stdio -serial null -nographic -device virtio-gpu-pci -kernel /big/yocto_6.0_qarm64/build_qarm64/tmp/deploy/images/qemuarm64/Image -append 'root=/dev/vda rw  mem=256M ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8 net.ifnames=0 console=ttyAMA0 console=hvc0 swiotlb=0 '

runqemu - INFO - Host uptime: 95142.15

[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x411fd070]
[    0.000000] Linux version 6.18.24-yocto-standard (oe-user@oe-host) (aarch64-oe-linux-gcc (GCC) 15.2.0, GNU ld (GNU Binutils) 2.46) #1 SMP PREEMPT Thu Apr 23 15:17:42 UTC 2026
[    0.000000] random: crng init done
[    0.000000] Machine model: linux,dummy-virt
[    0.000000] Memory limited to 256MB
[    0.000000] efi: UEFI not found.
[    0.000000] OF: reserved mem: Reserved memory: No reserved-memory node in the DT
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000040000000-0x00000004fffffff]
```

```
[  OK  ] Created slice Slice /system/psplash-start.
         Starting Start psplash boot splash screen...
[  OK  ] Finished Enable Persistent Storage in systemd-networkd.
[  OK  ] Started Start psplash boot splash screen.
[  OK  ] Listening on Load/Save RF Kill Switch Status /dev/rfkill Watch.
[  OK  ] Started Start psplash-systemd progress communication helper.
         Starting Virtual Console Setup...
[  OK  ] Finished Virtual Console Setup.

OpenEmbedded nodistro.0 qemuarm64 ttyAMA0

Type 'root' to login with superuser privileges (no password will be asked).

qemuarm64 login: root
root@qemuarm64:~# cat /etc/issue
OpenEmbedded nodistro.0 \n \l

Type 'root' to login with superuser privileges (no password will be asked).

root@qemuarm64:~# cat /proc/version
Linux version 6.18.24-yocto-standard (oe-user@oe-host) (aarch64-oe-linux-gcc (GCC) 15.2.0, GNU ld (GNU Binutils) 2.46) #1 SMP PREEMPT Thu Apr 23 15:17:42 UTC 2026
root@qemuarm64:~# ls -l /
lrwxrwxrwx    1 root     root             7 Apr  5  2011 bin -> usr/bin
drwxr-xr-x    2 root     root          1024 Apr  5  2011 boot
drwxr-xr-x   12 root     root          2860 Jun  6 05:17 dev
drwxr-xr-x   25 root     root          3072 Jun  6 05:17 etc
drwxr-xr-x    2 root     root          1024 Apr  5  2011 home
lrwxrwxrwx    1 root     root             7 Apr  5  2011 lib -> usr/lib
drwx------    2 root     root         12288 Apr  5  2011 lost+found
drwxr-xr-x    2 root     root          1024 Apr  5  2011 media
drwxr-xr-x    2 root     root          1024 Apr  5  2011 mnt
dr-xr-xr-x  158 root     root             0 Jun  6 05:17 proc
drwx------    3 root     root          1024 Jun  6 05:18 root
drwxr-xr-x   14 root     root           420 Jun  6 05:17 run
lrwxrwxrwx    1 root     root             8 Apr  5  2011 sbin -> usr/sbin
drwxr-xr-x    2 root     root          1024 Apr  5  2011 srv
dr-xr-xr-x   12 root     root             0 Jun  6 05:17 sys
drwxrwxrwt    7 root     root           140 Jun  6 05:17 tmp
drwxr-xr-x   10 root     root          1024 Apr  5  2011 usr
drwxr-xr-x    8 root     root          1024 Jun  6 05:17 var
root@qemuarm64:~#
```

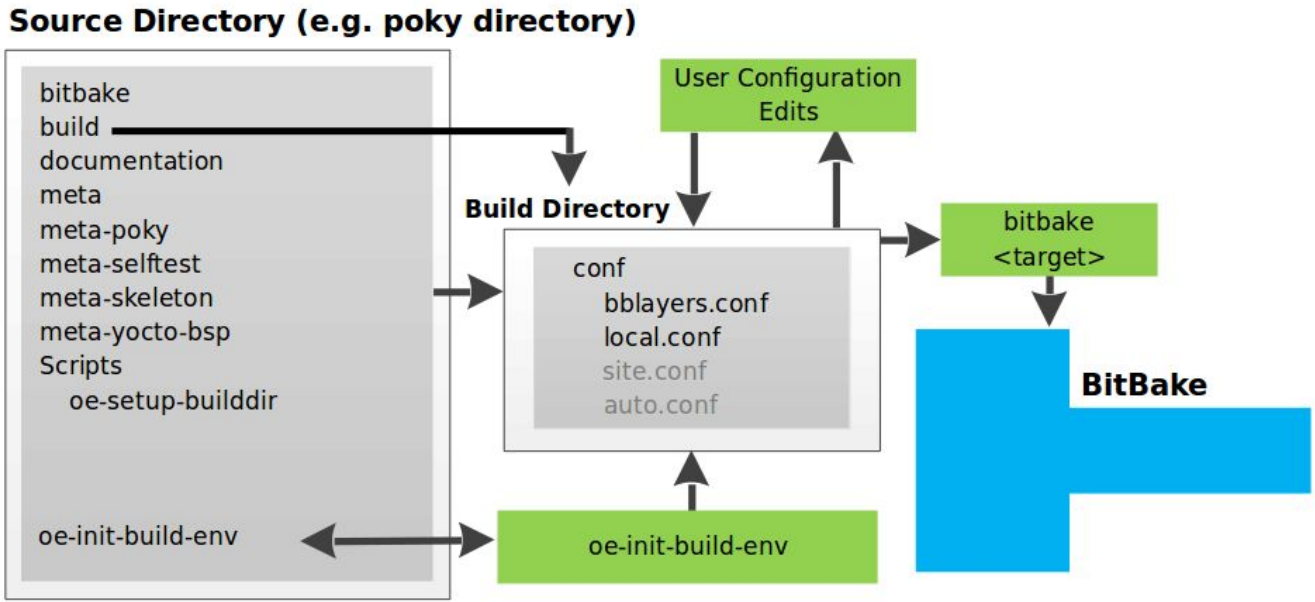
User configuration

Pic credit: Yocto official docs

- *oe-init-build-env* : the startup script to be sourced

conf:

- *bblayers.conf* : all the layers
- *local.conf* : the local machine build environment



The Layer-Recipe template

- A layer contains one or more recipes
- Recipes are key: it's how bitbake knows what to do

```
meta-<LAYER_NAME>
├── conf
│   └── layer.conf
├── <License>
├── README[.md]
└── recipes-<LAYER_NAME>*
    └── <RECIPE_NAME>
        ├── files
        │   └── <place required file(s) here>
        │    └── <as many as is required>
        └── <RECIPE_NAME>_<PV>.<PR>.bb
```

An example

```
meta-myprj
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
└── recipes-myprj
    └── myinit
        ├── files
        │   └── 0setup.sh
        └── myinit_0.1.bb
```

* The recipe folder name does Not have to be **recipes-**<LAYER_NAME> ; it can be **recipes-**<anything>

Example: layout of the *openembedded-core/meta-skeleton* layer and the recipes within it… (output truncated)

```
build_qarm64 $ tree ../upstream_layers/openembedded-core/meta-skeleton/
../upstream_layers/openembedded-core/meta-skeleton/
├── conf
│   ├── layer.conf
│   ├── multilib-example2.conf
│   └── multilib-example.conf
├── README.skeleton
├── recipes-core
│   └── busybox
│       ├── busybox
│       │   └── no_rfkill.cfg
│       └── busybox_%.bbappend
```

--snip--

```
└── recipes-skeleton
    ├── hello-autotools
    │   └── hello_2.10.bb
    ├── hello-single
    │   ├── files
    │   │   └── helloworld.c
    │   └── hello_1.0.bb
```

‘Hello, world’ - a quick layer:recipe demo

```
build_qarm64 $ tree ../myprj_layers/meta-myprj/
../myprj_layers/meta-myprj/
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
└── recipes-myprj
    ├── helloworld
    │   ├── files
    │   │   └── helloworld.c
    │   └── helloworld_0.1.bb
    └── myinit
        ├── files
        │   └── 0setup.sh
        └── myinit_0.1.bb
```

1. Setup the layer:recipe as per the template we just saw (screenshot); to do so, can use *recipetool* or *devtool* (or do it manually or use my simple *yct_recipe_gen* script)
2. Place the source file here: *files/helloworld.c*
3. Write the recipe (here):
   ./meta-myprj/recipes-myprj/helloworld/helloworld_0.1.bb
   (due to lack of space+time, see it in this presentation’s GitHub repo)
4. Verify that the recipe’s ‘seen’ by bitbake:

```
$ bitbake-layers show-recipes helloworld
…
=== Matching recipes: ===
helloworld:
  meta-myprj           0.1
```

‘Hello, world’ - a quick layer:recipe demo

5. Test: build just this recipe:

```
$ bitbake helloworld
```

6. Once it’s working, add it to the build via *conf/local.conf* (for now), thus ***“plating”*** the recipe:

```
IMAGE_INSTALL:append = " helloworld"
```

7. Rebuild:

```
bitbake core-image-base
```

8. Test:

```
runqemu qemuarm64 snapshot nographic
```

…

```
root@qemuarm64:~# helloworld
Hello, Yocto world
```

```
OpenEmbedded nodistro.0 qemuarm64 ttyAMA0

Type 'root' to login with superuser privileges (no password will be asked).

qemuarm64 login: root
root@qemuarm64:~#
root@qemuarm64:~# helloworld
Hello, Yocto world
root@qemuarm64:~#
root@qemuarm64:~# which helloworld
/bin/helloworld
root@qemuarm64:~#
```

Nice!

In addition: our useful **info_yocto** script shows some details; one of them’s the rootfs location within tmp/ :

```
$ ls -l tmp/work/qemuarm64-oe-linux/core-image-base/1.0/rootfs/bin/helloworld
-rwxr-xr-x 1 kaiwan kaiwan 67672 Mar  9  2018 tmp/work/<...>/helloworld
```

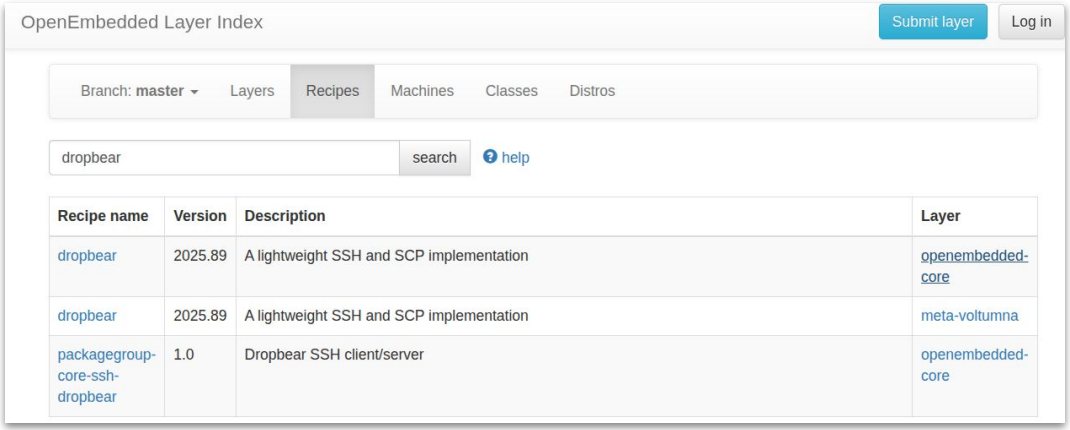
Adding a lightweight SSH / SCP implementation - dropbear !

In your ${BUILDDIR}/conf/local.conf:

```
IMAGE_INSTALL:append = " \
  helloworld \
  dropbear \
"
```

Q> How do we know the recipe name to use?
A>
1. Use the layer index!
https://layers.openembedded.org/layerindex/branch/master/recipes
Search within the 'Recipes' tab…

2. Check the 'Layer' that provides the recipe; it needs to be available.. Here, openembedded-core is always available (within the upstream layers). Else, first download and then 'install' the layer by running bitbake-layers add-layer <layer-name>

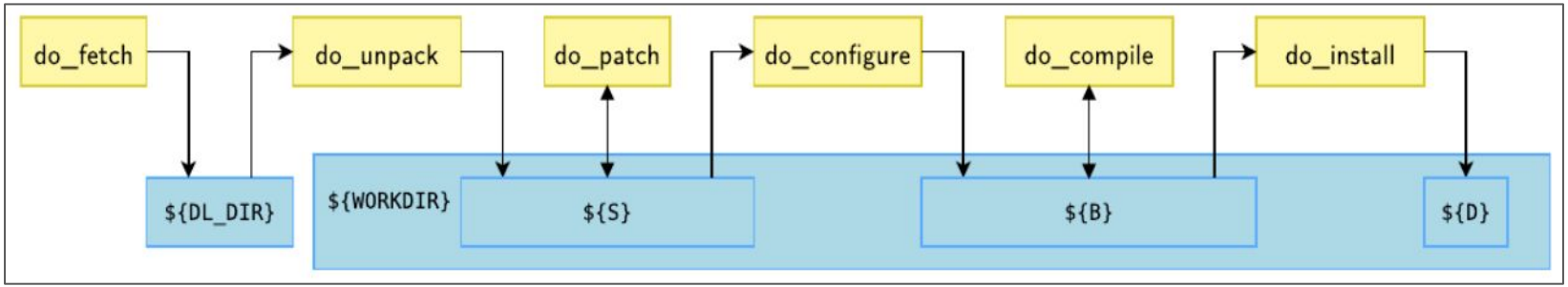


How a recipe works

Most (non-image) recipes generate one or more packages.
Every package has its 'own rootfs' under a 'work' directory; finally, the union of all package rootfs's becomes the final actual rootfs!

These are the main tasks (functions) - all optional - to be carried out by a recipe; they can be overridden or prepended or appended to!

Pic: from Bootlin's doc: "The main tasks"



DL_DIR : Download dir

${WORKDIR} : work dir (= ${TMPDIR}/work = tmp/work)

${S} : source dir

${B} : Build dir

${D} : Destination (install) dir

(FYI: ${BUILDDIR} is the build directory.)

TIP: Query a BB variable with: `bitbake-getvar [--value] <bb-variable-name>`

There's more steps than shown here: the do_package(), do_package_qa(), do_package_write_<pkgtype>(), do_populate_lic(), etc also follow!

Try
`bitbake -c listtasks helloworld`

Layout of the per-package ${WORKDIR} : ${TMPDIR}/work/…

The per-package (PN) WORKDIR value is:

WORKDIR = ${TMPDIR}/work/${PACKAGE_ARCH}-${DISTRO}-${TARGET_OS}/${PN}/${PV}-${PR} ; where

- ${TMPDIR} : typically named tmp/
- *NOTE*: ${PACKAGE_ARCH} is the processor family (f.e. Cortex A57), not the processor arch (f.e. if you're building Yocto for the Qemu arm64, then the processor type, PACKAGE_ARCH, is *cortexa57* (and not aarch64 or qemuarm64). (Tip: it's often the same as the TUNE_PKGARCH value.)
- ${DISTRO} is the distro name of course. Later, you'll likely want to customize and specify your very own custom distro! (we cover this in a later section). When you do so, replace it's name here… (It's often 'oe' - openembedded)
- ${TARGET_OS} is the target OS (typically 'linux')
- **${PN}** is the Package Name; usually matches the recipe name. F.e. for our 'helloworld' recipe, it's 'helloworld' of course!
  ${PV} is the Package Version
  ${PR} is the Package Release

So for our '**helloworld**' recipe, the **WORKDIR** value is: **tmp/work/cortexa57-oe-linux/helloworld/0.1/**

```
$ ls -F tmp/work/cortexa57-oe-linux/helloworld/0.1/
debugsources.list   deploy-source-date-epoch/  license-destdir/   packages-split/   pkgdata-pdata-input/
pseudo/             recipe-sysroot-native/      sources/          sysroot-destdir/
deploy-ipks/        image/                      package/          pkgdata/          pkgdata-sysroot/
recipe-sysroot/     source-date-epoch/          spdx/             temp/
```

Layout of ${TMPDIR}/${WORKDIR} : tmp/work/…

As well:

- Source dir S (or build dir B):

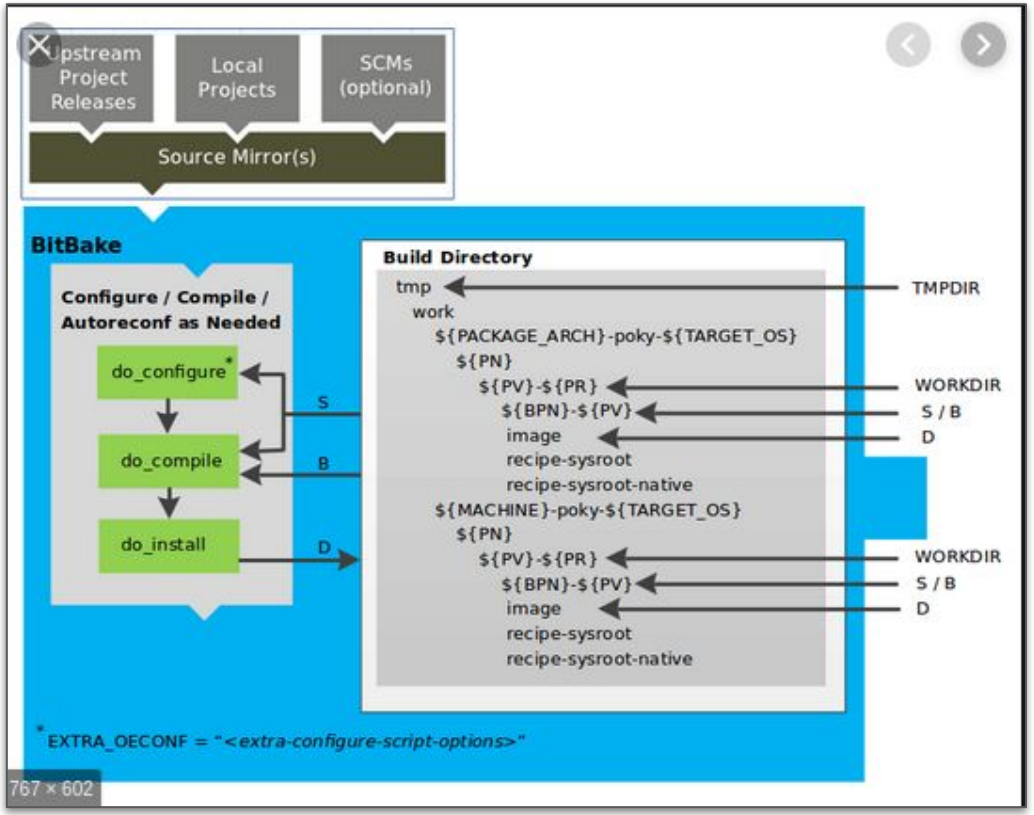
```
S = ${WORKDIR}/${BPN}-${PV}
```

- Destination dir D:

```
D = ${WORKDIR}/image
```

PRO / DEBUG TIP:
Under `${WORKDIR}/temp` you'll find v useful files: `log.do_*`, `run.do_*` (and more)!

The `run.do_*` files are per BB recipe action; f.e. `run.do_compile` (a soft link) will show precisely how compilation was done!



PRO TIPS

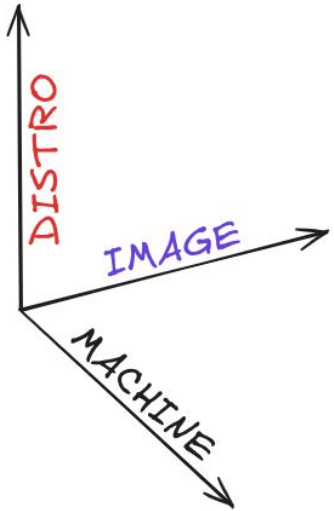
- *Must-read:*
  - *The extensive Yocto Documentation, the manuals*
    - Yocto Project Quick Build
    - What I wish I'd known about Yocto Project
- *Never* modify anything in the upstream layers (or poky); write your own layer:<append>recipe to make changes as required!
- Don't misunderstand how Yocto works; you CANNOT do anything (besides adding/removing a package) that makes a material difference on the rootfs *without a layer and recipe…* this includes creating, editing or removing even a single file (*want a script, a ~/.ssh/authorized_keys, etc? You need to get it in place!*)
  - READ THIS thread: *[yocto] recipe to clean up files from rootfs*
  - Can't change what another recipe does
  - CAN:
    - add to what a recipe does, via an append-style (.bbappend) recipe
    - postprocess the rootfs image creation.

PRO TIPS

When new, we simply use *conf/local.conf*.

The actual Yocto philosophy: **distro × image x machine are three orthogonal axes**

- The **DISTRO**
  - the top of this hierarchy; defines global policy
  - defines the distribution: the init manager, login manager, C library, toolchain settings, and more
  - akin to ‘capability bits’; they show what’s possible
  - set via the **DISTRO_FEATURES** bitbake variable
- The **IMAGE**
  - the mid-tier in this hierarchy; defines what actually goes into the rootfs (which packages); f.e. Ubuntu is a single distro but can have diff ‘flavours’ or ‘images’: desktop, server, coreOS, real-time, minimal, etc
  - ultimately generates an image artifact, the rootfs (ext[34] / wic / tar.bz2 / etc)
  - set via the **IMAGE_FEATURES** bitbake variable (expands to IMAGE_ISTALL + config)
- The **MACHINE**
  - the lowest level of this hierarchy; defines which **MACHINE** to build for (BSP layers)
  - which ‘distribution’ or Linux flavour to favour (**DISTRO**)
  - set via the IMAGE_INSTALL bitbake variable



PRO TIPS

You can add as many of the following as you like by adding them to your `${TOPDIR}/conf/local.conf` `EXTRA_IMAGE_FEATURES` variable (shown alphabetically below; they're described here):

```
allow-empty-password      allow-root-login
bash-completion-pkgs
dbg-pkgs                  debug-tweaks           dev-pkgs  doc      doc-pkgs
eclipse-debug             empty-root-password
hwcodecs
lic-pkgs
nfs-client                nfs-server
package-management        post-install-logging   ptest-pkgs
read-only-rootfs          read-only-rootfs-delayed-postinsts
splash src-pkgs           ssh-server-dropbear    ssh-server-openssh
stateless-rootfs          staticdev-pkgs
tools-debug               tools-profile          tools-sdk
tools-testapps
weston
x11                       x11-base               x11-sato
```

F.e.: `EXTRA_IMAGE_FEATURES ?= "allow-root-login allow-empty-password empty-root-password"`

There's (much) more, but no time to cover it all! :-)

So, please see the

Appendix

at the end of this presentation (a few more slides).

To conclude:

Building a custom embedded Linux with Yocto is not *that* difficult, nor is it that simple!

So:

“Hey Yocto, build me a custom embedded Linux!”

“Er, no”

Thank you!

Done *

* No, we're never really 'done'. Especially with Yocto, we've just seen the tip of the proverbial iceberg!

Q. What's the biggest room in the world?

A. The room for improvement!

Appendix

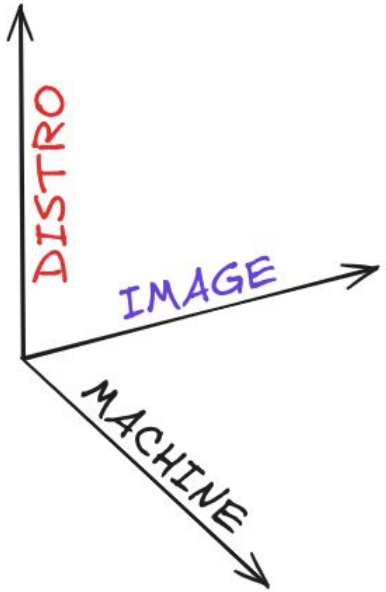
The stuff I had no time to cover :-) follows

PRO TIPS

The actual Yocto philosophy: **distro × image x machine are three orthogonal axes**

How a package X makes it into the target root filesystem: there are a few ‘gates’ to get through:

if `DISTRO_FEATURES` (or `REQUIRED_DISTRO_FEATURES`) includes it AND
  `IMAGE_FEATURES` includes it OR it’s explicitly added into the build via
    `IMAGE_INSTALL` (via the image recipe or local.conf)
then
  it’s packaged in
fi



F.e. if Wayland’s required (and not the old X11):

```
DISTRO_FEATURES:append = " wayland"
DISTRO_FEATURES:remove = " x11"
IMAGE_INSTALL:append = " weston weston-init" # weston is the reference
                              # implementation of a Wayland compositor
```

Defaults: if Yocto ver >= 5.3: use ‘nodistro’ else (older < 5.3) poky is the default distro; ‘image’ is typically core-image-{base|minimal}

PRO TIPS

The actual Yocto philosophy: **distro × image x machine are three orthogonal axes**

Conclusion: **What actually determines the installed package set:**

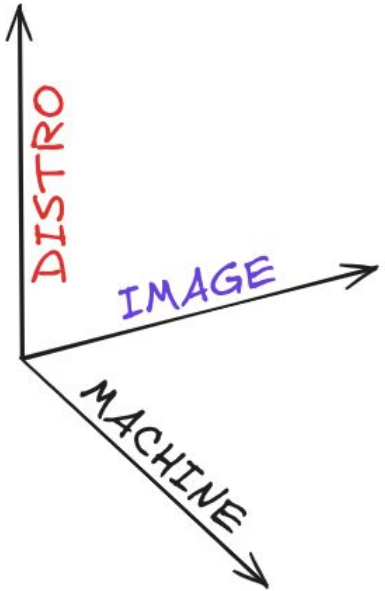
1. **The image recipe** : IMAGE_INSTALL, the core-image class, IMAGE_FEATURES. This is the real "package manifest." F.e., core-image-minimal pulls in packagegroup-core-boot + ${CORE_IMAGE_EXTRA_INSTALL}. The concept of **package groups** - literally, groups of packages - is key here (*link*).

2. **IMAGE_FEATURES / EXTRA_IMAGE_FEATURES** : toggles like debug-tweaks, ssh-server-dropbear, tools-debug that expand into package groups.

3. **Dependencies**: runtime deps (RDEPENDS), recommends (RRECOMMENDS), and PACKAGE_INSTALL resolution can drag in additional packages as required.

4. **Distro config**: influences the set indirectly: **DISTRO_FEATURES** gates whether features like systemd, wayland, pam are even available to be pulled in, and PREFERRED_* picks providers/versions. With newer Yocto ver >= 5.3's nodistro you just get OE-Core's 'un-opinionated' default DISTRO_FEATURES (unlike poky's 'opinionated' selections :-)).

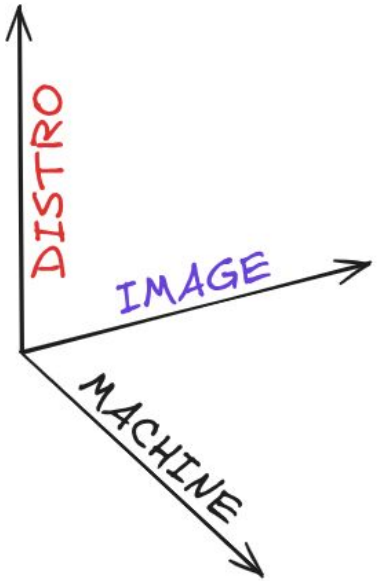
5. The *local* config: ${BUILDDIR}/conf/local.conf 's IMAGE_INSTALL 's.



PRO TIPS

The actual Yocto philosophy: **distro × image x machine are three orthogonal axes**

- **Custom distro config:** `meta-mydistro/conf/distro/mydistro.conf`
  - Meant for project's - all distro-policy variables / settings
  - `DISTRO_NAME, DISTRO_VERSION`
    F.e.: `DISTRO_FEATURES`:append = " … …", specifies init manager (`systemd` / `sysvinit`),
    `DISTRO_FEATURES:remove = "wayland x11", …`
  - Details - Yocto official docs: *Creating Your Own Distribution*
- **Custom Image recipe (image-specific, determines rootfs content):**
  `meta-mydistro/recipes-core/images/myimage.bb`
  - Meant for project's - determines the final rootfs content
  - Build: `bitbake myimage`
  - Contains image-specific settings
  - F.e.: `IMAGE_FEATURES += " … …" ;`
    `EXTRA_IMAGE_FEATURES ?= "debug-tweaks"`
    `IMAGE_INSTALL:append = " … …"`
    `BAD_RECOMMENDATIONS += "…"`
    `IMAGE_FSTYPES += "wic ext4 …"`
  - Learn about Yocto *packagegroups* - *Customizing Images Using Custom Package Groups*
- **Local config:** `${BUILDDIR}/conf/local.conf`
  - the lowest tier on the reproducibility and shareability ladder
  - holds the immediate '**machine**' build environment
    F.e.: `MACHINE, buildhistory, BB_DISKMON_DIRS` (limits on disk, memory, etc), identifies the `DISTRO` to build against
  - dev / immediate stuff / local hacking only



PRO TIPS

- Leverage the BitBake toolset
  - bitbake-layers
  - recipetool and devtool
  - bitbake-getvar, debugging
  - bitbake -c <cmd> (f.e. bitbake -c devshell busybox)
  - bitbake-setup (new, Yocto >= 5.3)
- Security
  - Yocto docs: *Making Images More Secure*
  - conf/local.conf: `require conf/distro/include/security_flags.inc`
- Release Notes
  - Very useful…
  - F.e. here's the Release Notes for yocto-6.0 (wrynose): https://downloads.yoctoproject.org/releases/yocto/yocto-6.0/RELEASENOTES

PRO TIPS

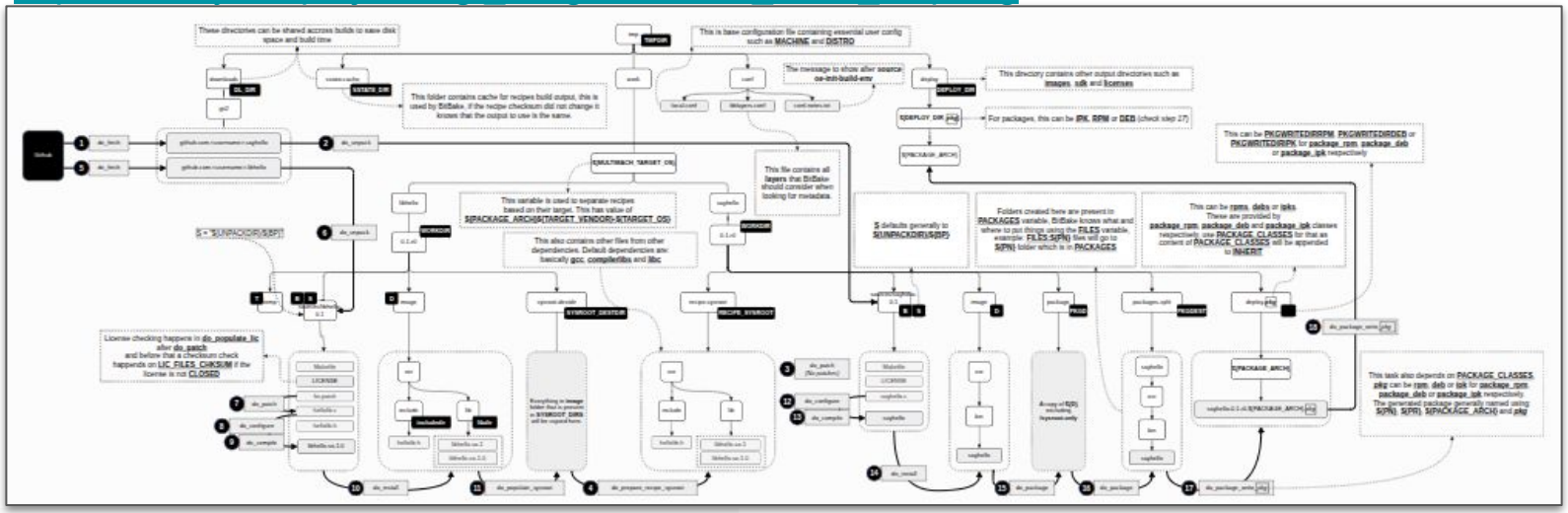
- When writing a recipe, do NOT assume that any pathname (file or directory), even a 'system' one, is present; always generate it via the install utility:
  `install -d -m <mode> <dir>`
- **Migration guides** are Very important! Must-read; f.e.:
  https://docs.yoctoproject.org/dev/migration-guides/migration-5.3.html

  F.e.
  "**WORKDIR changes**
  `S = ${WORKDIR}/something` **no longer supported**
  If a recipe has S set to be `${WORKDIR}/something`, this is no longer supported, and an error will be issued. The recipe should be changed to:
  ```
  S = "${UNPACKDIR}/something"
  ```

- My yocto_tools GitHub repo ; YMMV

PRO TIPS

- The complete, *very* detailed ***Bitbake Tasks Map: doc; image:*** https://docs.yoctoproject.org/_images/bitbake_tasks_map.svg



- Reminder: pl see this excellent presentation: *YPS 2023.11 - 2023/11/30 - Talel BELHAJSALEM - Back to basics | The ultimate Yocto introduction* (1h08m)

<End document>