

Creating Yocto Project compatible layers

Paul Barker, The Yocto Project

(C) 2026, CC-BY-SA 4.0

Source

Slides -->



Who am I?

- I work full time on Yocto Project
- Act as a leverage function, help to coordinate our activities and help the project to scale effectively
- Still get my hands dirty with a bit of implementation and debugging



Previous talks

I've spoken about "Friendly layers" before:

- 2022 talk
- 2019 talk

The 2022 talk is still a great resource, I'll try not to repeat too much from this.

Contents

Introduction to the Yocto Project compatibility program

The `yocto-check-layer` tool

Further best practices and advice

Introduction to the Yocto Project compatibility program



What is the Yocto Project compatibility program?

- Allows YP member companies to advertise compatibility
- Reduces likelihood of issues with different layer combinations
    - Less time wasted debugging issues

Why should you care?

- There are *many* layers available, leading to a multitude of possible combinations
- Poorly designed layers can interfere with eachother and cause bugs
- Following best practices, and passing `yocto-check-layer` tests makes your layer safe
    - For your development team
    - For your downstream users

YP Compatible criteria

- Open to Yocto Project members at any tier
    - OpenEmbedded can sponsor community layers
- Layer must pass `yocto-check-layer` tests
- README file with accurate dependency list
- Builds without errors
- All dependency layers must be YP Compatible

YP Compatible criteria, continued

- BSP and distro layers must be separate
- If you re-package bitbake and/or openembedded-core:
  - Must be clearly identified and isolated
  - Any changes must be submitted to the project
- Alignment with Yocto Project goals

What's new in version 3?

- Program relaunch
- Criteria have changed slightly over the last 5 years
- Recent criteria additions
  - SECURITY file
  - Cannot disable QA checks
  - Network access only in do_fetch, via bitbake fetchers

Compatibility is branch specific

- When submitting, specify YP releases you support
- We will track support for new YP releases
  - If you support master
  - If you create a new branch for the new YP release
- Updates for Wrynose are about to happen

The `yocto-check-layer` tool



What `yocto-check-layer` does

- Checks one or more layers, including dependencies
- Validates layer compatibility
- Enforces best practices

Preparation

- You need a build directory where you can run bitbake commands
  - You should be able to use `bitbake-setup`, `kas shell`, etc
- Remove the layers to be tested from `bblayers.conf`
- Keep `local.conf` and other config files simple

Running `yocto-check-layer`

For a layer with only dependency on openembedded-core:

```
yocto-check-layer ../layers/meta-openembedded/meta-oe
```

For a layer with additional dependencies:

```
yocto-check-layer ../layers/meta-virtualization --no-auto-dependency --dependency \
    ../layers/meta-openembedded/meta-{oe,python,filesystems,networking}
```

Running `yocto-check-layer` for BSP layers

For a BSP layer, we need to specify which MACHINEs to test:

```
yocto-check-layer ../layers/meta-yocto/meta-yocto-bsp --machines genericx86-64 genericarm64
```

For maximum coverage, test all layers with at least two MACHINEs specified (e.g. `qemux86-64` and `qemuarm64`). This may be required in the future.

Build directory interaction

- Uses existing configuration (`bblayers.conf` , `local.conf` , etc) and build directory
  - Setting some variables may impact validity of the tests
- Layers to be tested *must not* be in `bblayers.conf` already
- **Recommendation**: Set up a fresh build directory and limited config for running `yocto-check-layer`

Test automation

- You should run `yocto-check-layer` regularly in CI
- We can run this on our autobuilder for member companies and community layers

User interface improvements

`yocto-check-layer` is currently difficult to run correctly.

- Argument handling can be improved (this would be a breaking change)
- Should use a separate build directory
- Should use bblayers.conf to find layers and dependencies
- Should enumerate MACHINEs in a BSP layer and test them all

See YOCTO #16266 for more info.

Basic test cases

- Checks for README and SECURITY file presence
- Checks that recipes parse (`bitbake -p`)
- Checks that we can evaluate the bitbake environment (`bitbake -e`)
- Checks that `LAYERSERIES_COMPAT` is set
- Checks that no required QA tests are skipped (see `CHECKLAYER_REQUIRED_TESTS`)

Patch upstreaming

- Looks at all `.patch` files, regardless of whether they are used in `SRC_URI` or not
- Checks that patch files have valid `Upstream-Status` tags
- Currently allows 'Pending' patches without explanation, but this may change

Signature tests

- Checks that simply adding a layer doesn't change task signatures
- For BSP layers, checks that changing `MACHINE` does not change machine-independent task signatures

Signature tests: what does this mean?

What does this mean?

- A task signature is essentially the hash of a task, its inputs and dependencies
    - Changes to any of these will change the signature

So, be careful with `layer.conf`, bbappends, upgraded versions of recipes, etc.

- See previous "Friendly layers" talks for advice

Future test cases

- Check DISTROs added by a layer
- Stricter patch validity checks
- Check that parsing is possible without network access
- Check for valid license

We're open to other ideas!

- Test cases need to be lightweight, cannot actually build recipes.

Limitations to remove

- Cannot currently handle multiconfig builds or dependencies
- Cannot currently handle bare-metal MACHINEs
  - We don't want to have to set `COMPATIBLE_HOST` in every recipe!
- Should be possible to modify specific variables unconditionally
  - E.g. `IMAGE_FEATURES[validitems]`

Further advice and best practices



Anonymous Python functions

You can run Python code at parse time.

- Very powerful
- Be careful of impact on parse time
- Evaluated late, after `inherit`, `inherit_defer`, `include`, etc

Anonymous Python function example

```
python () {
    kver = (int(d.getVar("LINUX_VMAJOR")), int(d.getVar("LINUX_VMINOR")))
    kver_minimum = (int(d.getVar("LINUX_MINIMUM_VMAJOR")), int(d.getVar("LINUX_MINIMUM_VMINOR")))

    # Handle checksums of older COPYING files
    if kver <= (4, 17):
        d.setVar("LIC_FILES_CHKSUM", "file://COPYING;md5=d7810fab7487fb0aad327b76f1be7cd7")
    elif kver <= (5, 5):
        d.setVar("LIC_FILES_CHKSUM", "file://COPYING;md5=bbea815ee2795b2f4230826c0c6b8814")
}
```

(this particular example is now obsolete but it's still instructive)

Expensive variable assignments

Another way to impact parse time if you're not careful!

- Will be evaluated multiple times during parsing
    - Every d.setVar() call causes bitbake to refresh the data store

```
MY_VARIABLE = "${@expensive_function_call()}"
```

Event handlers

A good way to control exactly when evaluation happens.

Example from `create-spdx-3.0.bbclass`:

```
python spdx30_build_started_handler () {
    import oe.spdx30_tasks
    d = e.data.createCopy()
    oe.spdx30_tasks.write_bitbake_spdx(d)
}

addhandler spdx30_build_started_handler
spdx30_build_started_handler[eventmask] = "bb.event.BuildStarted"
```

See `lib/bb/event.py` in bitbake for a list of built-in events.

Thank you!

