



Creating Friendly Layers, 2022 Edition

Paul Barker, SanCloud Ltd


Yocto Project Summit, 2022.11

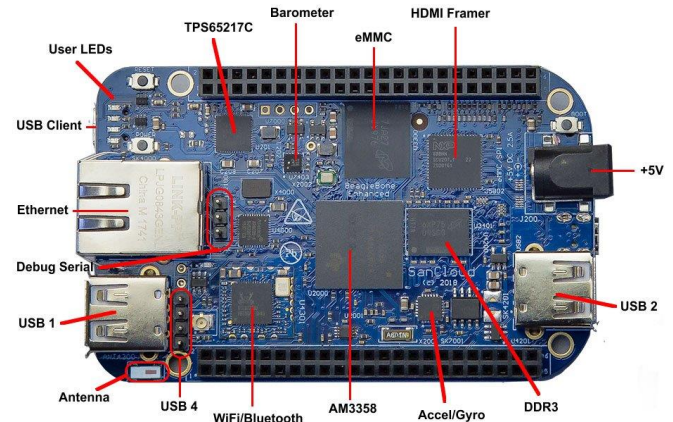
About Me

- Involved in Yocto Project since 2013
- Work across the whole embedded stack
- Principal Engineer @ SanCloud Ltd
-  paul@pbarker.dev
-  [@pbarker@social.afront.org](https://social.afront.org/@pbarker)



About SanCloud

- Custom & Off-The-Shelf Embedded/IoT Hardware
- Cloud hosted or on-site IoT Platform 
- Open Source focussed
- Global customer base
- UK head office



BeagleBone Enhanced (BBE)

About This Talk

- For layer creator & maintainers
- What & Why?
- Best Practices
 - Layers to learn from
 - Methods
 - Examples
- Parsing details of `bblayers.conf` and `layer.conf` files

Updated Edition
For Kirkstone & Langdale releases

There Shall Be No Victims

- I won't be showing examples of bad practice today
- Sorry to disappoint!



What is a Friendly Layer?

- Simply adding the layer doesn't change functionality
- Doesn't assume MACHINE, DISTRO, etc
- Careful use of bbappends
- Avoid clashing with recipe names in existing layers
- Place python helpers in a lib directory
 - Avoid littering the global namespace




Why Should You Care?

- **Yocto Project Compatible badge requires this**
- **Makes it easier to integrate with other layers**
 - Less likely to cause conflicts
- **Easier to test and debug builds**
 - Can quickly turn features on and off
- **Can reduce the number of layers you need to create**
 - Check MACHINE instead of having one layer per machine
 - Check features instead of having one layer per feature
- **Actually simplifies development of your layer**

But Can't You Just Dynamically Set BBLAYERS?

- **Not in a multiconfig**
- **Not based on variables in local.conf or some layer**
 - So you may not even know MACHINE, DISTRO, etc
- **Not even very easily in bblayers.conf**
 - Parsing limitations discussed later
- **Dynamically creating bblayers.conf for each build means another script to maintain**

Build a Friendly Community

- Provide documentation
- Provide clear contribution guidelines
 - How to send patches
 - Where to report issues
 - If needed, adopt a Code of Conduct
- Use inclusive language   

Layers To Learn From

- **meta-virtualization**
- **meta-clang**
- **meta-security**
- **meta-raspberrypi**

Documenting Your Layer

- You need a README
- Consider adding a 'docs' folder at the top level
 - Sphinx (<http://www.sphinx-doc.org>) is a good choice
 - Can publish to Read the Docs (<https://readthedocs.org>)
- Also clearly identify
 - Licensing
 - How to contribute
 - Support forums, mailing lists or email addresses

Keep layer.conf Simple

- **Settings in layer.conf apply to all recipes**
 - Not just those in your layer
- **Often difficult to override things set in layer.conf**
- **Parsed very early**
 - Details covered in appendix slides
 - Parsed in BBLAYERS order not BBFILE_PRIORITY order

Adding New Content in Layers

- **New content is typically safe to add**
 - New recipes
 - New classes
 - New machines
 - New distros
- **Watch out for name clashes**
 - Search the layer index first: <https://layers.openembedded.org/>

Modifying Existing Recipes

- This is where you can cause problems
- Don't indiscriminately modify variables and tasks
- Use overrides and conditionals
- Check MACHINE, DISTRO, feature variables, etc

Avoid Network Access Outside do_fetch

- Network access disabled by default outside do_fetch in recent releases
- Do not override this!
 - Especially not for do_configure/do_compile/do_install
 - Likely to break license compliance tooling, source archival and many other tools!

Use `:remove` With Caution

- `:remove` takes precedence over `:append`
- `:remove` cannot be undone easily!
- Avoid it if at all possible

Using Overrides

- Extend **OVERRIDES** based on a variable
- Use override syntax in variable assignments
- Document your new variable
- For example, if you support option `a` and option `b`:

```
OVERRIDES =. "option-${OPTION}"
```

```
SRC_URI:append:option-a = "file://a.patch"
```

```
SRC_URI:append:option-b = "file://b.patch file://b.conf"
```

Example: Toolchain Override in meta-clang

- In `clang.bbclass`:

```
OVERRIDES =. "${@[', 'toolchain-${TOOLCHAIN}:' ]['${TOOLCHAIN}' != '']}"
```

```
CC:toolchain-clang = "..."
```

```
CXX:toolchain-clang = "..."
```

```
CPP:toolchain-clang = "..."
```

```
CCLD:toolchain-clang = "..."
```

```
CLANG_TIDY_EXE:toolchain-clang = "..."
```

```
RANLIB:toolchain-clang = "..."
```

```
AR:toolchain-clang = "..."
```

```
NM:toolchain-clang = "..."
```

Using Features

- Much tidier than messing with overrides
- Three classes of feature variables:
 - DISTRO_FEATURES
 - MACHINE_FEATURES
 - IMAGE_FEATURES
- Also have **COMBINED_FEATURES**
 - Intersection of DISTRO_FEATURES & MACHINE_FEATURES

Conditional Syntax

- Python expressions
 - Can call a function `fn` with the syntax `\${@fn()}`
- Commonly used condition function:
 - `bb.utils.contains` – is `checkvalues` a subset of `variable`?

```
def contains(variable, checkvalues, truevalue, falsevalue, d):  
    if checkvalues.issubset(variable):  
        return truevalue  
    else:  
        return falsevalue
```

*Actual code is slightly more complex

Conditional Inclusion

- You can use Python expressions in include and require statements

- Example:

```
require ${@bb.utils.contains('DISTRO_FEATURES', ...)}
```

- You can have a simple .inc file without conditionals if you have many changes to make based on one condition

Include vs Require Statements

- ``require`` errors on missing files
 - You almost always want this
- ``include`` silently ignores missing files
 - Useful for optional configs
 - Useful when including something from another optional layer

Example: Distro Features in meta-virtualization

- **README**

The bbappend files for some recipes (e.g. linux-yocto) in this layer need to have 'virtualization' in DISTRO_FEATURES to have effect. To enable them, add in configuration file the following line.

```
DISTRO_FEATURES:append = " virtualization"
```

- **linux-%.bbappend**

```
include ${@bb.utils.contains('DISTRO_FEATURES', 'virtualization', '...', '', d)}
```

- **No DISTRO_FEATURES conditionals needed in the .inc file**

Example: Conditional Inheritance in meta-integrity

- **linux_ima.inc**

```
inherit ${@bb.utils.contains('DISTRO_FEATURES', 'modsign',  
                             'kernel-modsign', '', d)}
```

- **No DISTRO_FEATURES conditionals needed in kernel-modsign.bbclass**

Adding Build-time Checks

- Add a handler for `bb.event.SanityCheck`
 - Ensures your check only runs once
- Raise a flag if things look wrong
 - `bb.warn()`
 - `bb.error()`
 - `bb.fatal()` if you really can't continue
- Use this if you really must limit supported values of **MACHINE**, **DISTRO**, etc

Example: Checks in meta-virtualization

- **sanity-meta-virt.bbclass**

```
addhandler virt_bbappend_distrocheck
virt_bbappend_distrocheck[eventmask] = "bb.event.SanityCheck"

python virt_bbappend_distrocheck() {

    skip_check = e.data.getVar('SKIP_META_VIRT_SANITY_CHECK') == "1"

    if 'virtualization' not in e.data.getVar('DISTRO_FEATURES').split()
        and not skip_check:

        bb.warn("...")
}
```

Using Anonymous Python Functions

- Useful when more complex conditionals are needed
 - Full support for python if statements, for statements, etc
- Executed at parse time
- Can use `d.getVar()` to check variables
- Can use `d.setVar()` to modify variables
- **Example:**

```
python() {  
    if d.getVar('SOMEVAR').startswith('prefix'):  
        d.setVar('SOMEOTHERVAR', '1')  
}
```

Using Classes to Modify Recipes

- Define a new class in your layer
- Do not set INHERIT in layer.conf or elsewhere
- Document that your functionality is enabled by adding the new class to INHERIT in local.conf or a distro conf
- Useful if you have similar modifications to make to many recipes

Modifying BBCLASSEXTEND

- Appending to BBCLASSEXTEND in a bbappend is relatively safe
- No need for conditionals here
- May be used to add ``-native`` variant of an existing recipe
 - Can then be used in the build of another recipe

yocto-check-layer Script

- Layer compatibility test script
- Checks recipe signatures with and without the layer present
- Also checks for other common requirements:
 - Does the layer have a README?
 - Does everything parse correctly?
 - Is `LAYERSERIES_COMPAT` set?
 - Can we get signatures for `bitbake world`?
 - Actual build is not performed

In Summary: Think About Downstream Developers

- How can they extend configuration?
- How can they disable things?
 - Don't force them to use :remove
- **Don't assume distro, machine or target image**
 - If support really is limited, add a sanity check

Appendix: bblayers.conf Parsing Details

- **Parsed first**
 - Before any layer.conf
 - Before local.conf or other user config files
 - Before base.bbclass
- **BBLAYERS is iterated as soon as bblayers.conf is fully parsed**
 - Can't depend on variables from any of the above files
- **No access to python lib directories from any layer**
 - Can't `import oe` or any submodules
 - Can't use `oe.utils.conditional()`, use `bb.utils.contains()` instead

Appendix: layer.conf Parsing Details

- Parsed in sequence of BBLAYERS immediately after bblayers.conf
- Still before local.conf, base.bbclass, etc
- Still no access to python lib directories from any layer
 - Including the current layer!



yocto
PROJECT

THE
LINUX
FOUNDATION