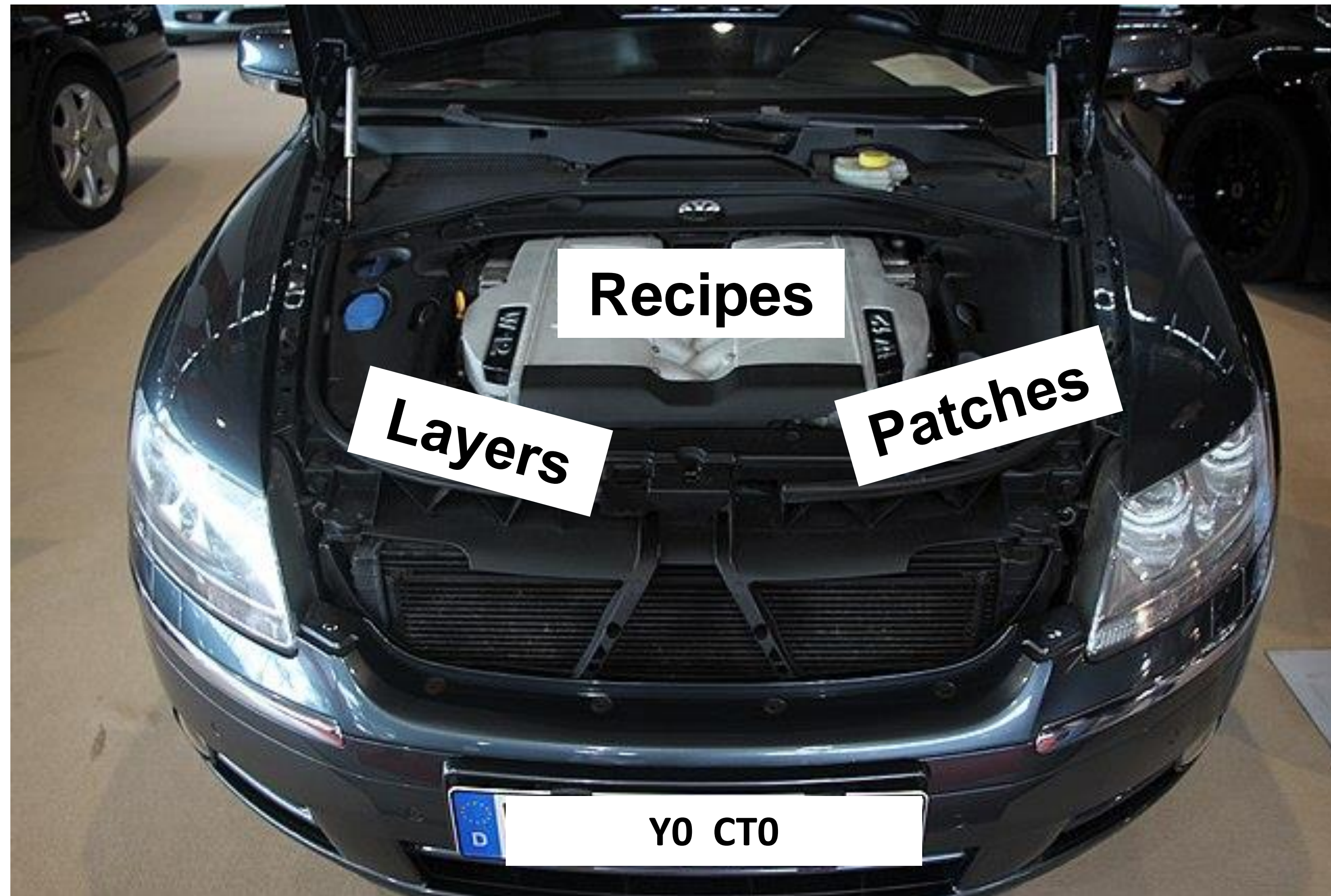
Building Linux boot files using templates for multiple SoC projects

Giulio Muscarello (ATLAS L1CT)

~~Petalinux~~ Yocto under the hood

Alexander Migl, CC BY-SA 4.0, via Wikimedia Commons

~~Petalinux~~ Yocto under the hood



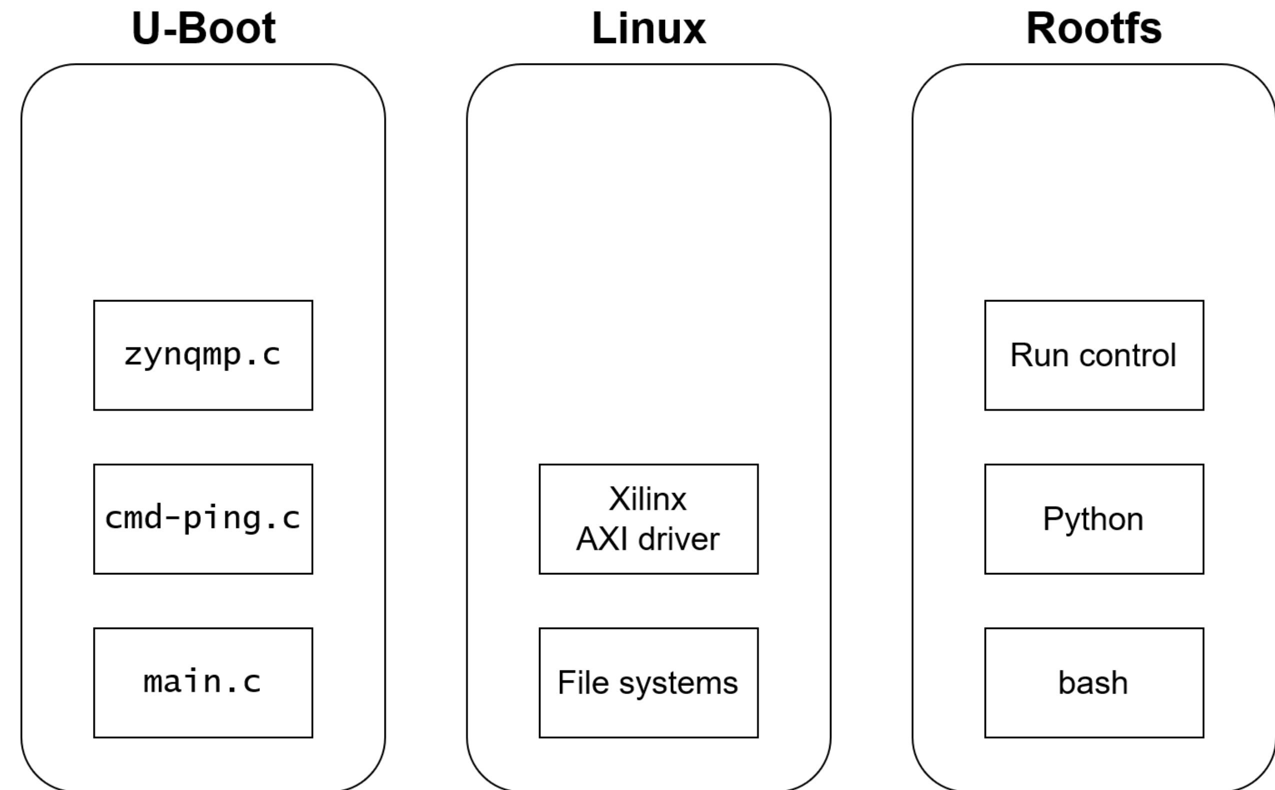
Wikisympathisant, CC BY-SA 4.0, via Wikimedia Commons

~~Petalinux~~ Yocto under the hood

You might think of Yocto as a collection of software packages that Just Works™, and perhaps where you drop your files into (e.g. “run control” in the diagram)

This is mostly true. Yocto **recipes** are software packages that comprise the full source code of an application, plus the instructions for compiling it in the BitBake language (more on this later).

But… it’s not the whole story.



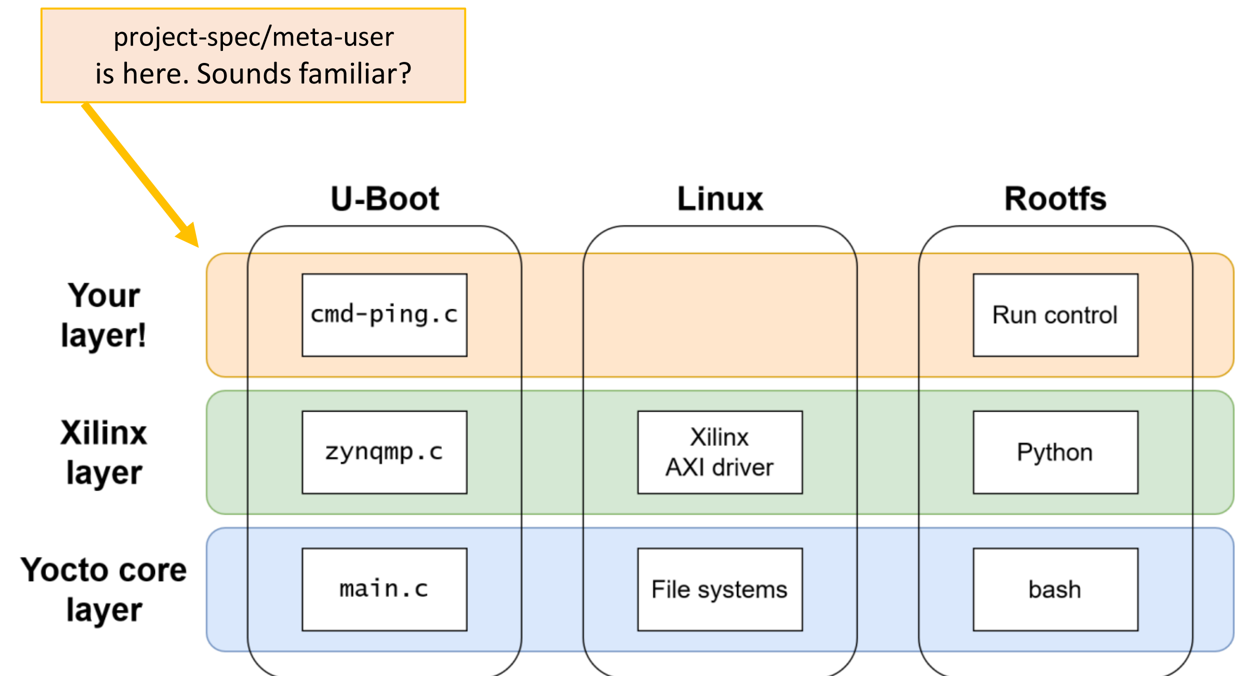
~~Petalinux~~ Yocto under the hood

In a typical Petalinux project, Yocto looks more like this.

Layers are stacked on top of each other. Each new layer on top modifies the recipes below – adding some files, removing others, changing build instructions. There are layers from the Yocto team, layers from Xilinx, and even layers from CERN!

This means that **recipes are not monolithic**, rather, they are the sum of many different contributions.

Only at build time layers are flattened into monolithic recipes ready for compilation.



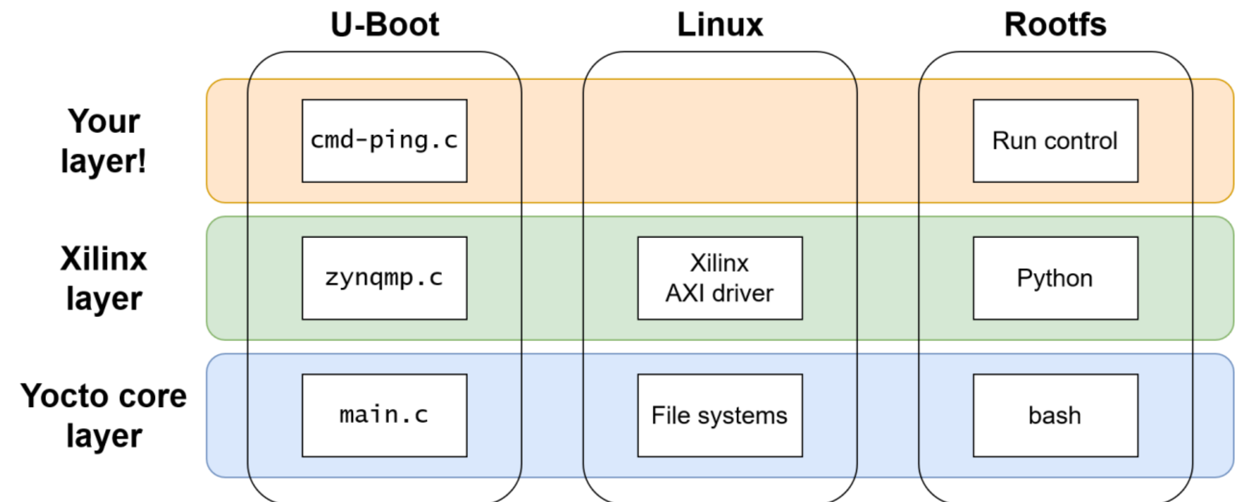
** Simplified representation: Yocto has tens of layers, Xilinx has ~10, and the user can have several*

~~Petalinux~~ Yocto under the hood

The advantage of this design is that layers are **decoupled**: different vendors can work independently

- The Yocto team curates generic versions of U-Boot, Linux kernel, etc., doesn't go mad trying to support every SoC that exists
- Xilinx only needs to maintain patches for its SoCs, doesn't care about the rest of the codebase
- Likewise, you can (mostly!) just concentrate on your patches and not worry about U-Boot internals, Xilinx driver details, etc.

Because layers are composable, you can also **share and reuse** them across teams: more on this later…

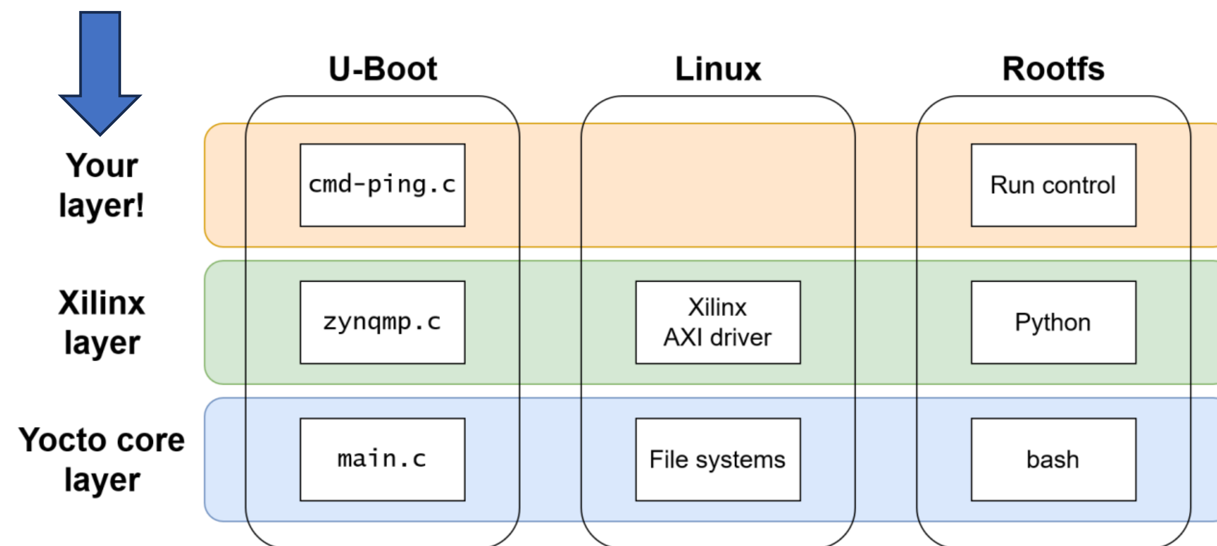


Tutorial goals

After this tutorial you should be able to:

- Get started **writing your own** Yocto layer
- **Share** your layer and **reuse** others' in a standalone project
- Use the SoC IG Petalinux template to **simplify** layer management

This is not a full guide to Yocto: refer to documentation!



What does a layer look like?

What does a layer look like?

A layer is a collection of patches with some metadata:

- `layer.conf`: name, priority
- Individual recipe directories
  - `[recipe].bbappend`: modifies build instructions
  - .patch files: modify source code
  - .c, .h, … files: add source code
  - .cfg files: set configuration values (can be overridden by upper layers!)

Not all recipes have all kinds of files.

```
[gmuscare@pcphese91 layers]$ tree 1-u-boot-sipl/
1-u-boot-sipl/
├── conf
│   └── layer.conf
├── recipes-bsp
│   └── u-boot-xlnx
│       ├── u-boot-xlnx_%.bbappend
│       ├── Add-SIPL-support-by-R.-Spiwoks.patch
│       └── sipl.cfg
├── recipes-kernel
│   └── linux
│       ├── iptables.cfg
│       ├── phy.cfg
│       ├── qspi.cfg
│       ├── uio.cfg
│       └── linux-xlnx_%.bbappend
├── README.md
└── ...
```

What does a layer look like? – .bbappend

Original recipe

Recipes are written in the BitBake language, but for most purposes they use a very simple set of features and operations:

Our .bbappend

Suppose we want to add a new feature. How would we change each step?

Step	Original recipe	Our .bbappend
1. **Fetch**	Clone a Git repository; copy a list of local files (`SRC_URI` variable) into the build directory	Add our C files/patches `SRC_URI += "hello.c hello.h i2c.patch"`
2. **Configure**	Prepare project for building, typically copying files around and applying patches	Move our files from the build directory to the correct place: `mv hello-world.h ${S}/include/` .patch files are applied automatically!
3. **Compile**	Call `make` (typically). Yocto/BitBake does not manage the compilation itself!	(No changes to compilation *commands*: if anything, you'll want to patch the *Makefile or CMakeLists.txt* in step 2)

What does a layer look like? – Patches

When modifying source code you typically deal with .patch files

Patches:

- Describe what files and lines to modify
- Contain the modifications
  - + Lines with a plus are additions
  - - Lines with a minus are removals
  - **Other lines are context**
- Are typically generated automatically

Can you think of a tool that makes it easy to work with diffs?

```
--- a/lib/misc_init_r.c
+++ b/lib/misc_init_r.c
@@ -1,3 +1,4 @@
+#include "i2c_init.h"
 #ifdef CONFIG_SIPL
 #include <sipl_init.h>
 #endif
@@ -11,6 +12,7 @@
 #ifdef CONFIG_SIPL
     res |= sipl_misc_init_r();
 #endif
+    res |= i2c_misc_init_r();
     return res;
 }
```

Filename

Line numbers

Modifications

What does a layer look like? – Git patches

A **Git** commit is just a patch to which we give a title and a parent hash

Indeed, it is much more practical to use Git commits to represent changes:

- There are more practical tools to work with them
- People are more used to working with Git

You can trivially convert a git commit to a patch (`git show HASH > file.patch`) and vice versa (`patch; git add; git commit`)

```
--- a/lib/misc_init_r.c
+++ b/lib/misc_init_r.c
@@ -1,3 +1,4 @@
+#include "i2c_init.h"
 #ifdef CONFIG_SIPL
 #include <sipl_init.h>
 #endif
@@ -11,6 +12,7 @@
 #ifdef CONFIG_SIPL
     res |= sipl_misc_init_r();
 #endif
+    res |= i2c_misc_init_r();
     return res;
 }
```

Filename

Line numbers

Modifications

What does a layer look like? – Git patches

In the configuration stage, Yocto will:

1. Convert raw patches to Git commits
2. Apply the commits in order
3. Copy any additional files from SRC_URI
4. Build the project

If the patches don't apply (typically because the "base" source has changed in an update), Yocto will throw an error and warn the user

How to adapt commits/patches for new Petalinux release? We discuss this later

What does a layer look like? – Configuration

Yocto uses a simple language to describe configuration options: **Kconfig** (originally used in the Linux kernel, now used by many C projects)

1. Developers write Kconfig files to describe what configuration options their software accepts
2. Users write .cfg files (“**fragments**”) to configure their software, either using petalinux-config or manually:

   ```
   CONFIG_SIPL=y
   ```

3. Kconfig translates these values to #define options for C code:

   ```
   #define CONFIG_SIPL
   ```

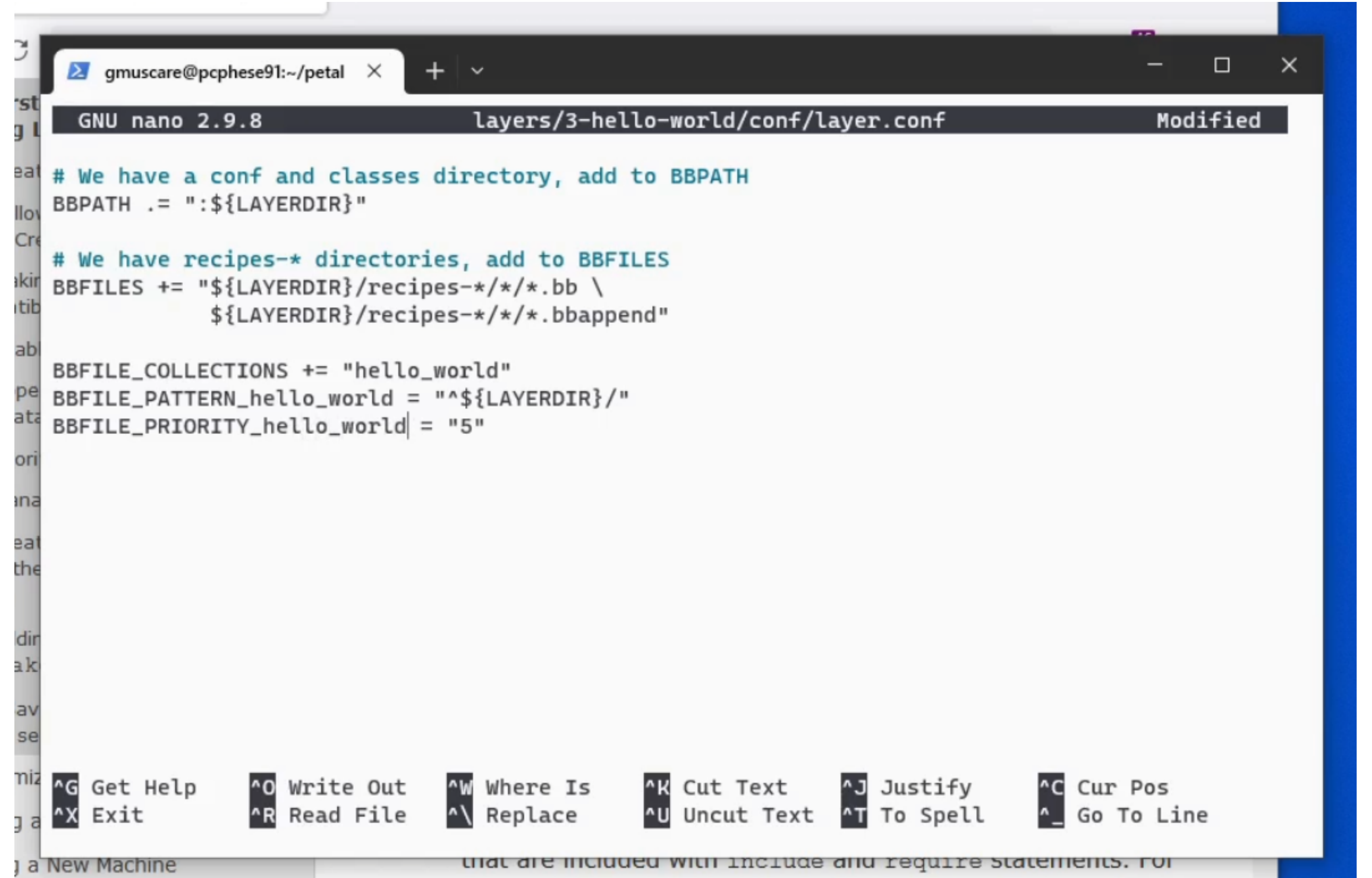
```
index 807a4c6ade..a66c86da09 100644
--- a/lib/Kconfig
+++ b/lib/Kconfig
@@ -216,6 +216,12 @@ config RBTREE
 config BITREVERSE
    bool "Bit reverse library from Linux"

+config SIPL
+   bool "Enable software support for the SIPL-TM protocol"
+   help
+     Enables software support for the SIPL-TM protocol over a serial
+          interface to the IPMC.
+
 config TRACE
    bool "Support for tracing of function calls and timing"
    imply CMD_TRACE
```

```
.config - U-Boot 2022.01 Configuration
> Library routines
                         Library routines
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or
  empty submenus ----).  Highlighted letters are hotkeys.  Pressing
  <Y> includes, <N> excludes, <M> modularizes features.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*]
          ^(-)
          [ ] Bit reverse library from Linux
          [ ] Enable software support for the SIPL-TM protocol
          [ ] Support for tracing of function calls and timing
          [ ] Support the 'dhry' command to run the dhrystone benchmark
              Security support  --->
              Android Verified Boot  --->
              Hashing Support  --->
```

How to create a layer: **layer.conf**

Watch on Indico: [link]()



What does a layer look like? **Git patches, .bbappend**

Watch on Indico: link

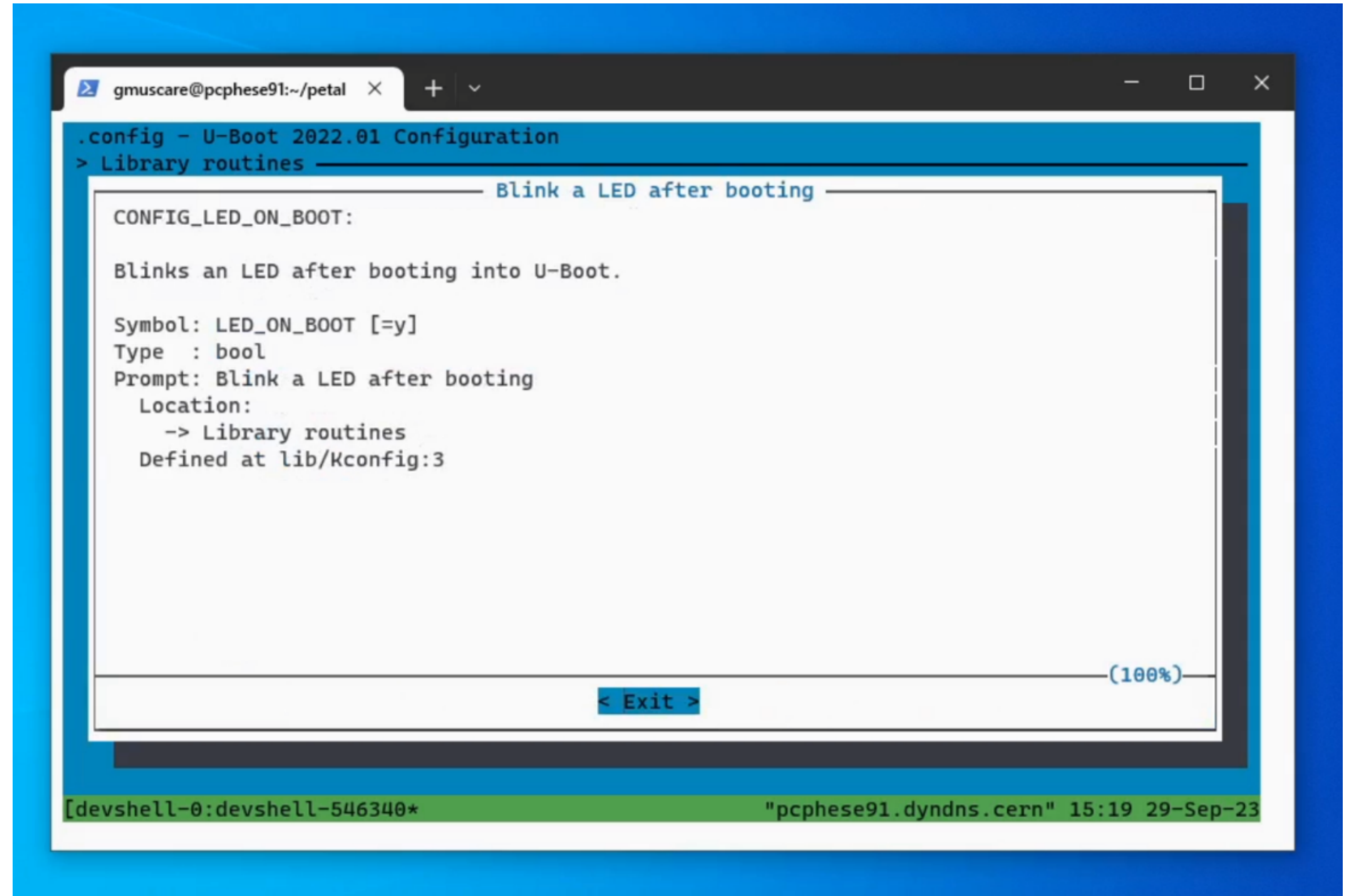
```
gmuscare@pcphese91:~/petal

INFO: Handling branch devtool-override-pn-arm-trusted-firmware...
INFO: No patches or local source files needed updating
INFO: Handling branch devtool-override-zynq...
NOTE: Writing append file /home/gmuscare/petalinux/layers/3-hello-world/recipes-bsp/u-boot/u-b
oot-xlnx_%.bbappend
NOTE: Copying 0001-Fix-QSPI-env-vars-with-JTAG-booting-zynq.patch to /home/gmuscare/petalinux/
layers/3-hello-world/recipes-bsp/u-boot/u-boot-xlnx/0001-Fix-QSPI-env-vars-with-JTAG-booting-z
ynq.patch
INFO: Handling branch devtool-override-zynqmp...
INFO: No patches or local source files needed updating
INFO: Cleaning sysroot for recipe u-boot-xlnx...
INFO: Leaving source tree /home/gmuscare/petalinux/l1ct-peta/components/yocto/workspace/source
s/u-boot-xlnx as-is; if you no longer need it then please delete it manually
[gmuscare@pcphese91 l1ct-peta]$ tree ../layers/3-hello-world/
../layers/3-hello-world/
├── conf
│   └── layer.conf
└── recipes-bsp
    └── u-boot
        ├── u-boot-xlnx
        │   ├── 0001-Add-LED-on-boot-functionality.patch
        │   └── 0001-Fix-QSPI-env-vars-with-JTAG-booting-zynq.patch
        └── u-boot-xlnx_%.bbappend

4 directories, 4 files
[gmuscare@pcphese91 l1ct-peta]$
```

What does a layer look like? **Configuration**

Watch on Indico: [link]



Maintaining layers

Petalinux releases new versions twice a year, and sometimes they add **breaking changes**

- Yocto is based on Git, so you can use familiar tools (shell, Visual Studio Code, …) to solve a familiar problem: **rebasing git commits** on a new branch and fix conflicts
  Also, git is forgiving: it will try to apply a patch as long as it can find the context, even if the line numbers changed
- You can also not use Git patches and just replace existing files entirely using SRC_URI.
  It's simpler to develop, but when updates introduce breaking changes it will be difficult to understand what changes are to be carried over. **Tradeoff** to consider!

Maintaining layers

The BitBake language rarely changes, but there was a breaking change in the Yocto release in Petalinux 2022.1:

```
SRC_URI_append = …      ->      SRC_URI:append = …
```

Also, recipes may be renamed in Petalinux releases (also in 2022.1).

Moral of the story – **read release notes** before upgrading

Table: Updated Yocto Recipe Names

Old Recipe File Name	New Recipe File Name
fsbl.bb	fsbl-firmware.bb
plm.bb	plm-firmware.bb

Sharing layers

Sharing layers

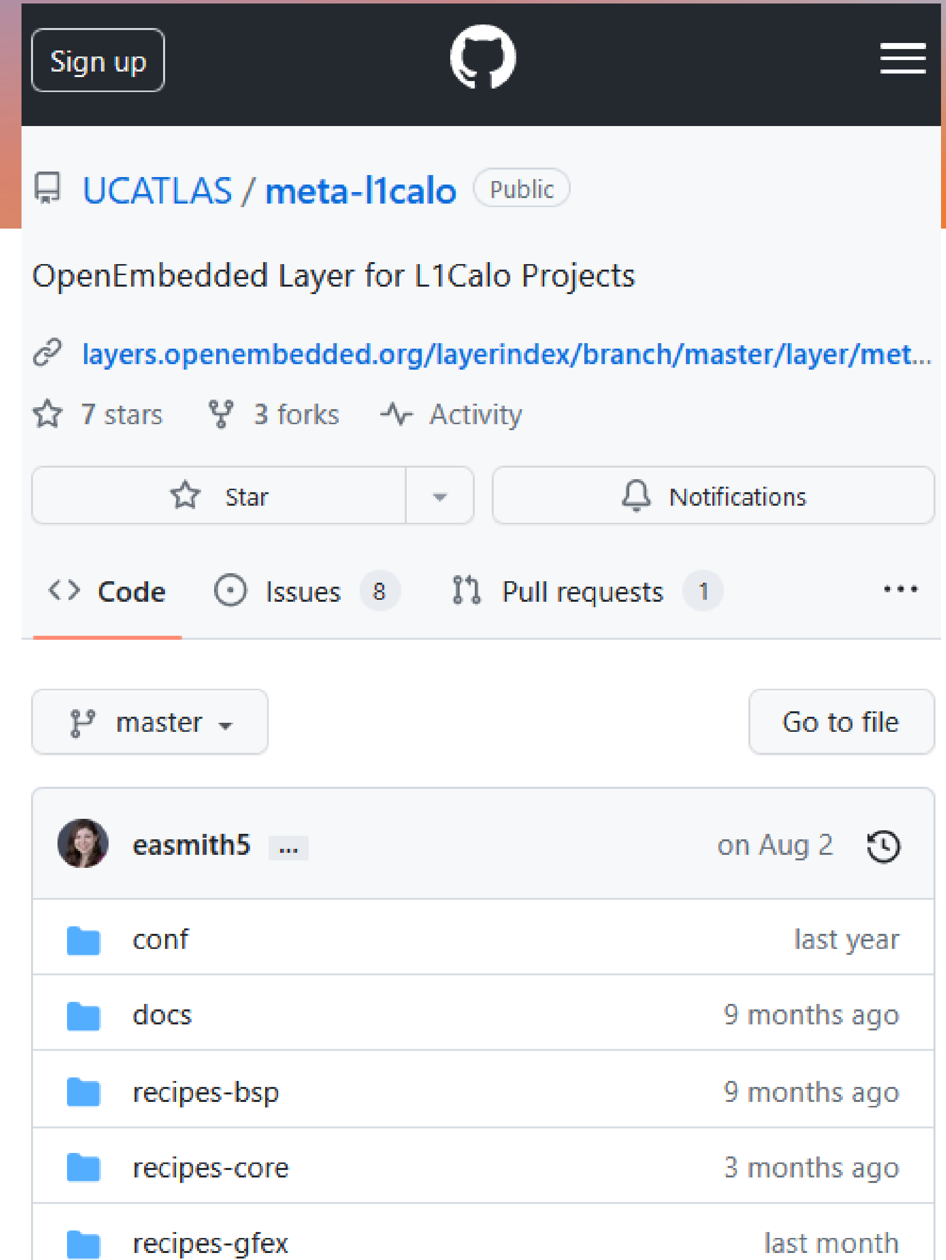
To share a layer, simply **distribute** your directory, e.g. via Git. See L1Calo layer on the right

Vice versa, you can **import** other people's layers by simply copying them on your machine and adding them from `petalinux-config`:

```
/home/gmuscare/petalinux/l1ct-peta/project-spec/configs/config - misc/config System Cor
→ Yocto Settings → User Layers
                              User Layers
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N>
  excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help,
  </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < > module

      (/home/gmuscare/petalinux/layers/0-soc-ig-common) user layer 0
      (/home/gmuscare/petalinux/layers/1-u-boot-sipl) user layer 1
      (/home/gmuscare/petalinux/layers/2-l1ct-common) user layer 2
      (/home/gmuscare/petalinux/boards/muctpi-v4) user layer 3
      ()  user layer 4

            <Select>    < Exit >    < Help >    < Save >    < Load >
```



Sharing layers: importing

Watch on Indico: link

gmuscare@pcphese91:~/petal

```
/home/gmuscare/petalinux/l1ct-peta/project-spec/configs/config - misc/config System Configura
t→ Yocto Settings
                              Yocto Settings
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
  ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
  modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

        (zynqmp-generic) YOCTO_MACHINE_NAME
            TMPDIR Location  --->
            Devtool Workspace Location  --->
            Parallel thread execution  --->
            Add pre-mirror url   --->
            Local sstate feeds settings  --->
        [*] Enable Network sstate feeds
              Network sstate feeds URL  --->
        [ ] Enable BB NO NETWORK
        [*] Enable Buildtools Extended
            User Layers  --->

         <Select>    < Exit >    < Help >    < Save >    < Load >
```

Activity

recipes-bsp   move to UCATLAS/meta-l1calo   9 months ago

Sharing layers: good practices

Some good practices when creating layers for external usage:

- #1 - write **documentation**!
  Documentation prevents software rot, speeds up the onboarding of newcomers, and helps potential users understand your code
- Have **one feature per layer**
  Every feature increases complexity, potential overhead, etc. often unnecessarily. Don't package many unrelated changes in one layer
- Strongly **prefer Git** patches over raw patches
  Git patches embed documentation and authorship, and have a clear ordering
- **Prefer patches** over SRC_URI override
  SRC_URI override masks what changes are actually part of the layer. Only override empty files

Sharing layers: the SoC Interest Group

We as SoC Interest Group make available two layers:

1. `soc-ig-common`, adding generic Client ID to U-Boot (*: how to retrieve it is left to you)
   See previous talks on Client ID and scalable booting
2. `u-boot-sipl`, adding SIPL support to U-Boot
3. … your layer here? Feel free to submit proposals!

You can use them directly, but…

Sharing layers: the SoC Interest Group

We also provide a template to **get started and simplify** the use of layers: just provide your .xsa and copy any additional layers in a directory

Try it out: soc/petalinux-template on CERN Gitlab

- Meant to be a starting point for your Petalinux projects
- Comes with a core set of features for Client ID
- You're invited to fork it and add features, set up a CI workflow, etc.
- Has scripts to automate building your project
- Scales well to multiple hardware projects with a common set of patches
  (Interested? Stay around for the next tutorial on multi-board projects!)

Currently in use in the ATLAS L1CT group, kindly tested by ATLAS gFEX and CMS

Sharing layers: the **SoC Interest Group**

Watch on Indico: [link]

```
gmuscare@pcphese91:~/petal

example
[gmuscare@pcphese91 petalinux]$ cp -r boards/example/ boards/l1ct-muctpi
[gmuscare@pcphese91 petalinux]$ tree boards/l1ct-muctpi/
boards/l1ct-muctpi/
├── bitfile
├── conf
│   └── layer.conf
├── recipes-bsp
│   ├── device-tree
│   │   └── files
│   │       └── system-user.dtsi
│   └── u-boot
│       ├── files
│       │   └── sipl.cfg
│       └── u-boot-xlnx_%.bbappend
└── xsa

6 directories, 6 files
[gmuscare@pcphese91 petalinux]$ ln -svf ~/my-hardware-design.xsa boards/l1ct-muctpi/xsa
'boards/l1ct-muctpi/xsa' -> '/home/gmuscare/my-hardware-design.xsa'
[gmuscare@pcphese91 petalinux]$ vim boards/l1ct-muctpi/recipes-bsp/device-tree/files/system
-user.dtsi
[gmuscare@pcphese91 petalinux]$
```

Conclusions

Conclusions

- The Yocto model, based on a composition of layers, lets people cooperate and work independently on individual features
- Yocto makes clever use of git: changes are packaged in .patch files that are easy to update
- The layers architecture makes it easy to share, reuse and collaborate: make use of it!
- SoC IG provides some layers
- SoC IG provides a template to simplify Petalinux builds