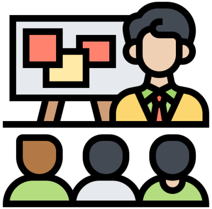
Yocto Project and OpenEmbedded system development training

Latest update: July 03, 2025.

Document updates and training details:
https://bootlin.com/training/yocto

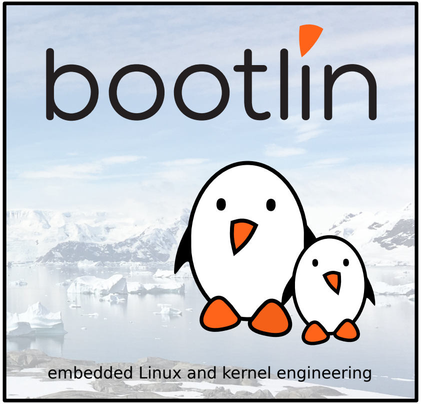
Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com

- These slides are the training materials for Bootlin's *Yocto Project and OpenEmbedded system development* training course.
- If you are interested in following this course with an experienced Bootlin trainer, we offer:
  - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
  - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
  - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- Details and registrations: https://bootlin.com/training/yocto
- Contact: training@bootlin.com



Icon by Eucalyp, Flaticon

About Bootlin



Corrections, suggestions, contributions and translations are welcome!

Bootlin introduction

- Engineering company
  - In business since 2004
  - Before 2018: *Free Electrons*
- Team based in France and Italy
- Serving **customers worldwide**
- **Highly focused and recognized expertise**
  - Embedded Linux
  - Linux kernel
  - Embedded Linux build systems
- **Strong open-source** contributor
- Activities
  - **Engineering** services
  - **Training** courses
- https://bootlin.com

Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox, OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded, Buildroot, ...

Embedded Linux integration

Boot time, real-time, security, multimedia, networking

Open-source upstreaming

Get code integrated in upstream Linux, U-Boot, Yocto, Buildroot, ...

Bootlin training courses

Course	On-site	Online
Embedded Linux system development	4 or 5 days	7 * 4 hours
Linux kernel driver development	5 days	7 * 4 hours
Yocto Project system development	3 days	4 * 4 hours
Buildroot system development	3 days	5 * 4 hours
Embedded Linux networking	3 days	4 * 4 hours
Understanding the Linux graphics stack	2 days	4 * 4 hours
Embedded Linux audio	2 days	4 * 4 hours
Real-Time Linux with PREEMPT_RT	2 days	3 * 4 hours
Linux debugging, tracing, profiling and performance analysis	3 days	4 * 4 hours

All our training materials are freely available
under a free documentation license (CC-BY-SA 3.0)
See https://bootlin.com/training/

Bootlin, an open-source contributor

- Strong contributor to the **Linux** kernel
  - In the top 30 of companies contributing to Linux worldwide
  - Contributions in most areas related to hardware support
  - Several engineers maintainers of subsystems/platforms
  - 9000 patches contributed
  - https://bootlin.com/community/contributions/kernel-contributions/
- Contributor to **Yocto Project**
  - Maintainer of the official documentation
  - Core participant to the QA effort
- Contributor to **Buildroot**
  - Co-maintainer
  - 6000 patches contributed
- Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- Fully **open-source training materials**

Bootlin on-line resources

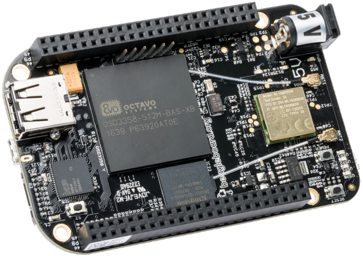
- Website with a technical blog:
  https://bootlin.com
- Engineering services:
  https://bootlin.com/engineering
- Training services:
  https://bootlin.com/training
- LinkedIn:
  https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line:
  https://elixir.bootlin.com

Icon by Freepik, Flaticon

Generic course information

Three supported hardware platforms

Three variants for this course, each using a different hardware platform.

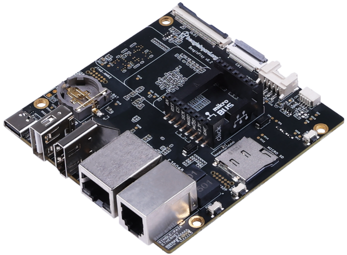


Beaglebone Black

https://bootlin.com/doc/training/yocto/

STM32MP157D-DK1 Discovery

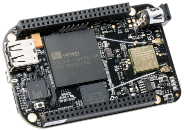
https://bootlin.com/doc/training/yocto-stm32/

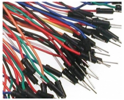


BeaglePlay

https://bootlin.com/doc/training/yocto-beagleplay/

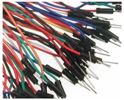
- Beaglebone Black or Beaglebone Black Wireless, USB-A to micro B power cable included
  https://www.mouser.fr/ProductDetail/BeagleBoard-by-GHI/BBBWL-SC-562?qs=k%2Fsw%252B3Yi%2FUbELBjXQpiBUQ%3D%3D
- USB Serial Cable - 3.3 V - female ends (for serial console)
  https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-SERIAL-F/
- Nintendo Nunchuk with UEXT connector
  https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/
- Breadboard jumper wires - Male ends (to connect to Nunchuk) https://www.olimex.com/Products/Breadboarding/JUMPER-WIRES/JW-110x10/
- Micro SD card with 8 GB capacity





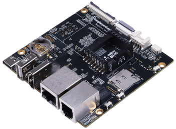

Shopping list: STM32MP1 Discovery Kit variant

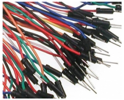
- STMicroelectronics STM32MP157D-DK1 Discovery kit
  https://www.st.com/en/evaluation-tools/stm32mp157d-dk1.html#sample-buy
- USB-C cable for the power supply
- USB-A to micro B cable for the serial console
- RJ45 cable for networking
- Nintendo Nunchuk with UEXT connector
  https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/
- Breadboard jumper wires - Male ends
  https://www.olimex.com/Products/Breadboarding/JUMPER-WIRES/JW-110x10/
- Micro SD card with 8 GB capacity



Shopping list: BeaglePlay variant

- BeaglePlay https://www.mouser.com/ProductDetail/BeagleBoard-by-Seeed-Studio/102110762?qs=amGC7iS6iy%252BiSone%2FBpwhg%3D%3D
- USB-C cable for the power supply
- USB Serial Cable - 3.3 V - female ends (for serial console) https://www.olimex.com/Products/Components/Cables/USB-Serial-Cable/USB-SERIAL-F/
- Nintendo Nunchuk with UEXT connector https://www.olimex.com/Products/Modules/Sensors/MOD-WII/MOD-Wii-UEXT-NUNCHUCK/
- Breadboard jumper wires - Male ends (to connect to Nunchuk) https://www.olimex.com/Products/Breadboarding/JUMPER-WIRES/JW-110x10/
- Micro SD card with 8 GB capacity

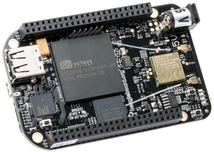




Supported hardware

BeagleBone Black or BeagleBone Black Wireless, from BeagleBoard.org

- Texas Instruments AM335x (ARM Cortex-A8 CPU)
- SoC with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- 512 MB of RAM
- 4 GB of on-board eMMC storage
- USB host and USB device, microSD, micro HDMI
- WiFi and Bluetooth (wireless version), otherwise Ethernet
- 2 x 46 pins headers, with access to many expansion buses (I2C, SPI, UART and more)
- A huge number of expansion boards, called *capes*. See https://elinux.org/Beagleboard:BeagleBone_Capes.



Supported hardware

Discovery Kits from STMicroelectronics: STM32MP157A-DK1, STM32MP157D-DK1, STM32MP157C-DK2 or STM32MP157F-DK2

- STM32MP157 (Dual Cortex-A7 + Cortex-M4) CPU from STMicroelectronics
- 512 MB DDR3L RAM
- Gigabit Ethernet port
- 4 USB 2.0 host ports, 1 USB-C OTG port
- 1 Micro SD slot
- On-board ST-LINK/V2-1 debugger
- Misc: buttons, LEDs, audio codec
- LCD touchscreen (DK2 only)

DK1 Discovery Kit

Board and CPU documentation, design files, software: A-DK1, D-DK1, C-DK2, F-DK2

Training quiz and certificate

- You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.

Participate!

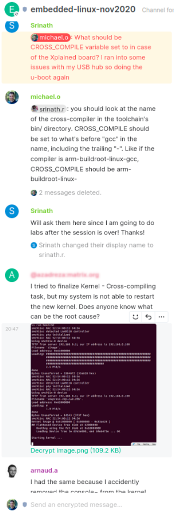
During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- In on-line sessions
  - Please always keep your camera on!
  - Also make sure your name is properly filled.
  - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.

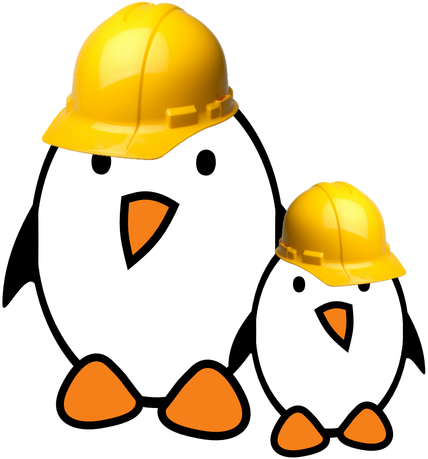
Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- Use the dedicated Matrix channel for this session to add questions.
- If your session offers practical labs, you can also report issues, share screenshots and command output there.
- Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.



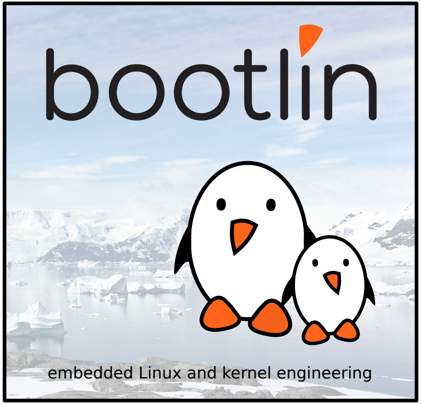
Practical lab - Training Setup



Prepare your lab environment

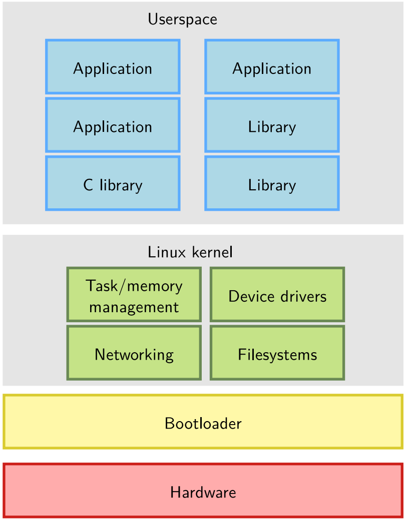
- Download and extract the lab archive

Introduction to Embedded Linux

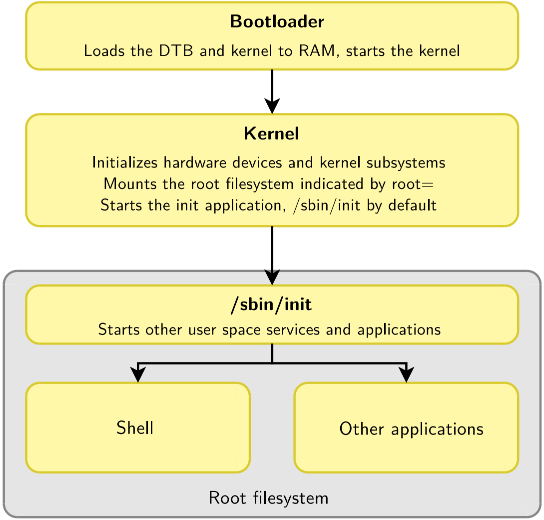


Corrections, suggestions, contributions and translations are welcome!

Simplified Linux system architecture



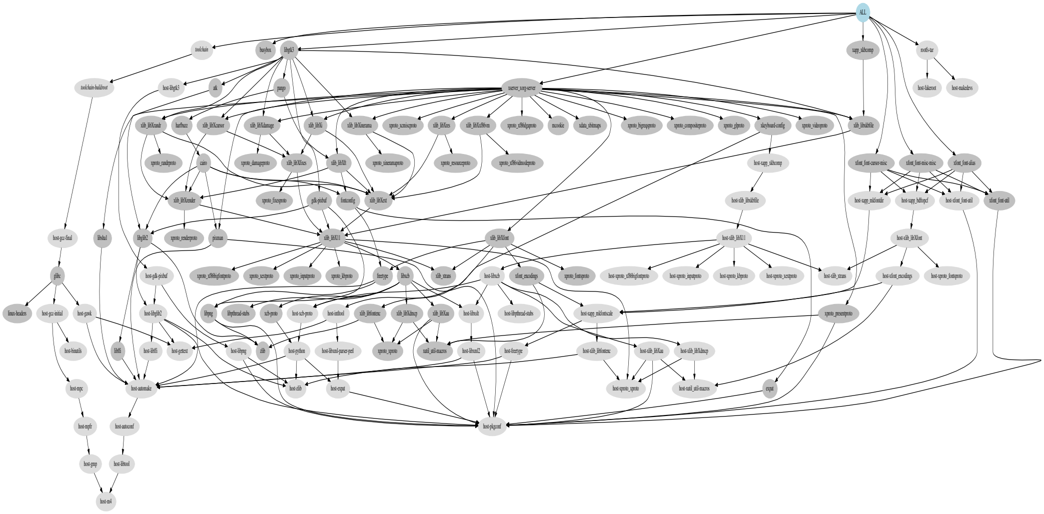
Overall Linux boot sequence



Embedded Linux work

- **BSP work**: porting the bootloader and Linux kernel, developing Linux device drivers.
- **system integration work**: assembling all the user space components needed for the system, configure them, develop the upgrade and recovery mechanisms, etc.
- **application development**: write the company-specific applications and libraries.

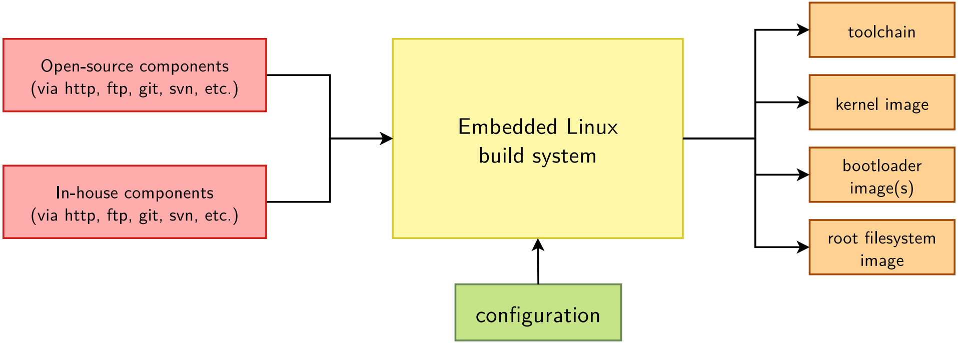
Complexity of user space integration



System integration: several possibilities

	Pros	**Cons**
Building everything manually	Full flexibility Learning experience	Dependency hell Need to understand a lot of details Version compatibility Lack of reproducibility
Binary distribution Debian, Ubuntu, Fedora, etc.	Easy to create and extend	Hard to customize Hard to optimize (boot time, size) Hard to rebuild the full system from source Large system Uses native compilation (slow) No well-defined mechanism to generate an image Lots of mandatory dependencies Not available for all architectures
Build systems Buildroot, Yocto, PTXdist, etc.	Nearly full flexibility Built from source: customization and optimization are easy Fully reproducible Uses cross-compilation Have embedded specific packages not necessarily in desktop distros Make more features optional	Not as easy as a binary distribution Build time

Embedded Linux build system: principle



- Building from source → lot of flexibility
- Cross-compilation → leveraging fast build machines
- Recipes for building components → easy

Embedded Linux build system: tools

- A wide range of solutions: Yocto/OpenEmbedded, PTXdist, Buildroot, OpenWRT, and more.
- Today, two solutions are emerging as the most popular ones
  - **Yocto/OpenEmbedded**
    Builds a complete Linux distribution with binary packages. Powerful, but somewhat complex, and quite steep learning curve.
  - **Buildroot**
    Builds a root filesystem image, no binary packages. Much simpler to use, understand and modify.

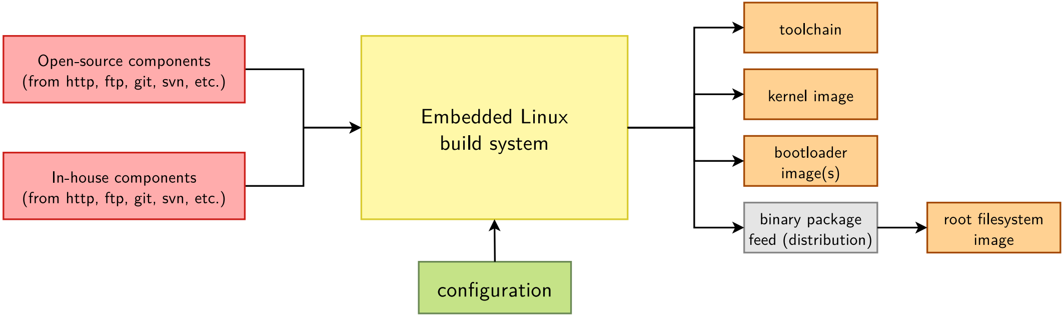
Yocto Project and Poky reference system overview

The Yocto Project overview

About

- The Yocto Project is an open source collaboration project that allows to build custom embedded Linux-based systems.
- Established by the Linux Foundation in 2010 and still managed by one of its fellows: Richard Purdie.

Yocto: principle



- Yocto always builds binary packages (a "distribution")
- The final root filesystem is generated from the package feed
- The big picture is way more complex

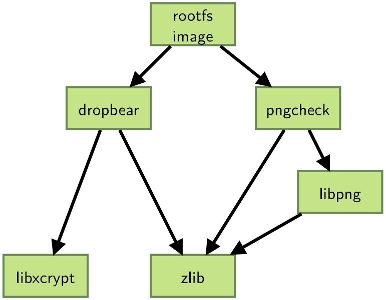
Lexicon: `bitbake`

In Yocto / OpenEmbedded, the *build engine* is implemented by the `bitbake` program

- `bitbake` is a task scheduler, like `make`
- `bitbake` parses text files to know what it has to build and how
- It is written in Python (need Python 3 on the development host)

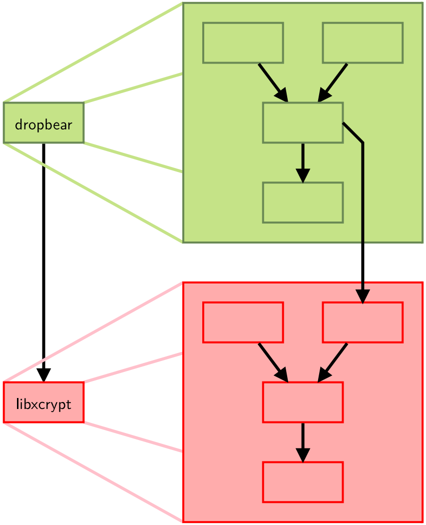
Lexicon: recipes

- The main kind of text file parsed by `bitbake` is *recipes*, each describing a specific software component
- Each *Recipe* describes how to fetch and build a software component: e.g. a program, a library or an image
- They have a specific syntax
- `bitbake` can be asked to build any recipe, building all its dependencies automatically beforehand



Lexicon: tasks

- The build process implemented by a recipe is split in several *tasks*
- Each task performs a specific step in the build
- Examples: fetch, configure, compile, package
- Tasks can depend on other tasks (including on tasks of other recipes)



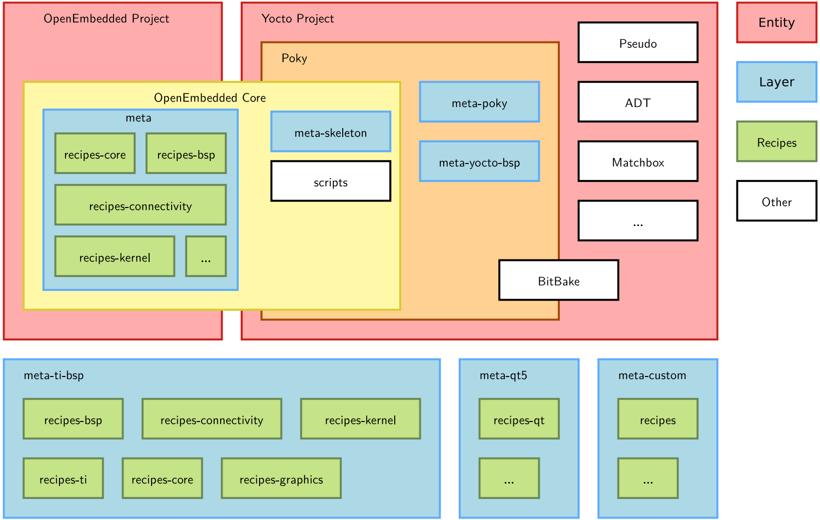
Lexicon: metadata and layers

- The input to `bitbake` is collectively called *metadata*
- Metadata includes *configuration files*, *recipes*, *classes* and *include files*
- Metadata is organized in *layers*, which can be composed to get various components
  - A layer is a set of recipes, configurations files and classes matching a common purpose
    - For Texas Instruments board support, the *meta-ti-bsp* layer is used
  - Multiple layers are used for a project, depending on the needs
- *openembedded-core* is the core layer
  - All other layers are built on top of openembedded-core
  - It supports the ARM, MIPS (32 and 64 bits), PowerPC, RISC-V and x86 (32 and 64 bits) architectures
  - It supports QEMU emulated machines for these architectures

Lexicon: Poky

- The word *Poky* has several meanings
- Poky is a git repository that is assembled from other git repositories: bitbake, openembedded-core, yocto-docs and meta-yocto
- poky is the *reference distro* provided by the Yocto Project
- meta-poky is the layer providing the poky reference distribution

The Yocto Project lexicon



The Yocto Project lexicon

- The Yocto Project is **not used as** a finite set of layers and tools.
- Instead, it provides a **common base** of tools and layers on top of which custom and specific layers are added, depending on your target.

Example of a Yocto Project based BSP

- To build images for a BeagleBone Black, we need:
  - The Poky reference system, containing all common recipes and tools.
  - The *meta-ti-bsp* layer, a set of Texas Instruments specific recipes.
- All modifications are made in your own layer. Editing Poky or any other third-party layer is a **no-go**!
- We will set up this environment in the lab.

The Poky reference system overview

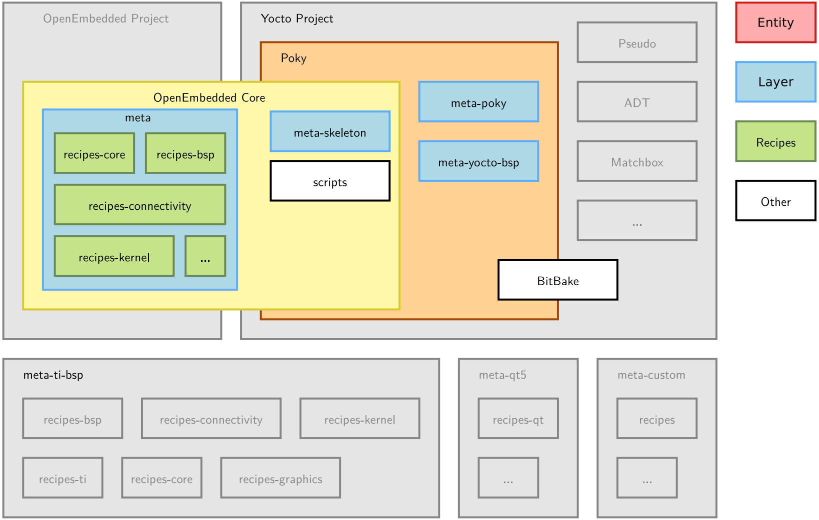
Getting the Poky reference system

- All official projects part of the Yocto Project are available at `https://git.yoctoproject.org/`
- To download the Poky reference system:

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

- A new version is released every 6 months, and maintained for 7 months
- **LTS** versions are maintained for 4 years, and announced before their release.
- Each release has a codename such as `kirkstone` or `scarthgap`, corresponding to a release number.
  - A summary can be found at `https://wiki.yoctoproject.org/wiki/Releases`

Poky



OpenEmbedded Project
Yocto Project
Poky
OpenEmbedded Core
meta
recipes-core
recipes-bsp
recipes-connectivity
recipes-kernel
...
meta-skeleton
scripts
meta-poky
meta-yocto-bsp
BitBake
Pseudo
ADT
Matchbox
...
Entity
Layer
Recipes
Other
meta-ti-bsp
recipes-bsp
recipes-connectivity
recipes-kernel
recipes-ti
recipes-core
recipes-graphics
meta-qt5
recipes-qt
...
meta-custom
recipes
...

Poky source tree 1/2

bitbake/ Holds all scripts used by the `bitbake` command. Usually matches the stable release of the BitBake project.

documentation/ All documentation sources for the Yocto Project documentation. Can be used to generate nice PDFs.

meta/ Contains the OpenEmbedded-Core metadata.

meta-skeleton/ Contains template recipes for BSP and kernel development.

Poky source tree 2/2

- **meta-poky/** Holds the configuration for the Poky reference distribution.
- **meta-yocto-bsp/** Configuration for the Yocto Project reference hardware board support package.
- **LICENSE** The license under which Poky is distributed (a mix of GPLv2 and MIT).
- **oe-init-build-env** Script to set up the OpenEmbedded build environment. It will create the build directory.
- **scripts/** Contains scripts used to set up the environment, development tools, and tools to flash the generated images on the target.

Documentation

- Documentation for the current sources, compiled as a "mega manual", is available at: https://docs.yoctoproject.org/singleindex.html
- Variables in particular are described in the variable glossary: https://docs.yoctoproject.org/genindex.html

Using Yocto Project - basics

Environment setup

Environment setup

- All Poky files are left unchanged when building a custom image.
- Specific configuration files and build repositories are stored in a separate build directory.
- A script, `oe-init-build-env`, is provided to set up the build directory and the environment variables (needed to be able to use the `bitbake` command for example).

oe-init-build-env

- Modifies the environment: has to be sourced!
- Adds environment variables, used by the build engine.
- Allows you to use commands provided in Poky.
- `source ./oe-init-build-env [builddir]`
- Sets up a basic build directory, named `builddir` if it is not found. If not provided, the default name is `build`.

The initial `build/` directory

- The `oe-init-build-env` script creates the `build` directory with only one subdirectory in it:

  conf Configuration files. Image specific and layer configuration.

Exported environment variables

BUILDDIR
: Absolute path of the build directory.

PATH
: Contains the directories where executable programs are located.
  Absolute paths to `scripts/` and `bitbake/bin/` are prepended.

Available commands

bitbake	The main build engine command. Used to perform tasks on available recipes (download, configure, compile…).
bitbake-*	Various specific commands related to the BitBake build engine.

Configuring the build system

The `build/conf/` directory

- The `conf/` directory in the `build` one holds two mandatory build-specific configuration files:

  bblayers.conf Explicitly list the layers to use.

  local.conf Set up the configuration variables relative to the current user for the build. Configuration variables can be overridden there.

- Additional optional configuration files can be used:

  site.conf Similar to `local.conf` but intended to be used for site-specific settings, such as network mirrors and CPU/memory resource usage limits.

Configuring the build

The conf/local.conf configuration file holds local user configuration variables:

- BB_NUMBER_THREADS: How many tasks BitBake should perform in parallel. Defaults to the number of CPU threads on the system.
- PARALLEL_MAKE: How many processes should be used when compiling. Defaults to the number of CPU threads on the system.
- MACHINE: The machine the target is built for, e.g. beaglebone.

Building an image

Compilation

- The compilation is handled by the BitBake *build engine*.
- Usage: `bitbake [options] [recipename/target ...]`
- To build a target: `bitbake [target]`
- Building a minimal image: `bitbake core-image-minimal`
  - This will run a full build for the selected target.
- The `oe-init-build-env` script lists some more example targets

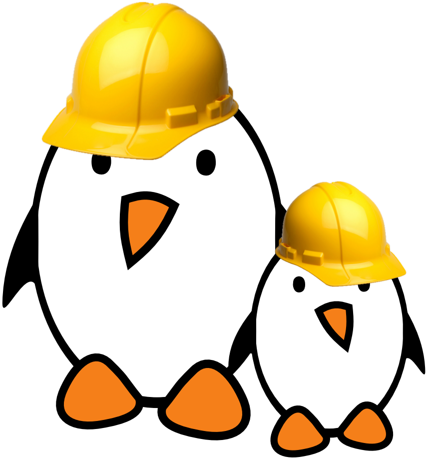
The `build/` directory after the build 1/2

conf/	Configuration files, as before, not touched by the build.
downloads/	Downloaded upstream tarballs of the recipes used in the builds.
sstate-cache/	Shared state cache. Used by all builds.
tmp/	Holds all the build system outputs.

The `build/` directory after the build 2/2

- tmp/work/ Set of specific work directories, split by architecture. They are used to unpack, configure and build the packages. Contains the patched sources, generated objects and logs.
- tmp/sysroots/ Shared libraries and headers used to compile applications for the target but also for the host.
- tmp/deploy/ Final output of the build.
- tmp/deploy/images/ Contains the complete images built by the OpenEmbedded build system. These images are used to flash the target.
- tmp/buildstats/ Build statistics for all packages built (CPU usage, elapsed time, host, timestamps...).

Practical lab - First Yocto build



- Download the sources
- Set up the environment
- Configure the build
- Build an image

Using Yocto Project - advanced usage

Advanced build usage and configuration

- Variable operators and overrides.
- Select package variants.
- Manually add packages to the generated image.
- Run specific tasks with BitBake.

A little reminder

- A *Recipe* describes how to fetch, configure, compile and install a software component (application, library, ...).
- These tasks can be run independently (if their dependencies are met).
- All the available packages in the project layer are not selected by default to be built and included in the images.
- Some packages may provide the same functionality, e.g. OpenSSH and Dropbear.

Variables

Overview

- The OpenEmbedded build system uses configuration *variables* to hold information.
- Variable *names* are in upper-case by convention, e.g. `CONF_VERSION`
- Variable *values* are strings
- To make configuration easier, it is possible to prepend, append or define these variables in a conditional way.
- Variables defined in **Configuration Files** have a **global** scope
  - Files ending in `.conf`
- Variables defined in **Recipes** have a **local** scope
  - Files ending in `.bb`, `.bbappend` and `.bbclass`
- Recipes can also access the global scope

Operators: basic assignment

- `VAR = "value"` simply assigns a value
- Re-assigning overwrites variable value

```
VAR = "this"
VAR = "that"
```

Result: `VAR = "that"`

- Newlines need to be escaped (this does not apply to functions)

```
LIST = "this \
        and that"
```

- Variable assignments can contain expansion of other variables

```
COLOUR = "blue"
SKY = "the sky is ${COLOUR}"
```

Operators: immediate expansion

- With `=`, expansion happens when the variable is used
- Use `:=` for immediate expansion

```
COLOUR = "blue"
SKY = "the sky is ${COLOUR}"
COLOUR = "grey"
PHRASE = "Look, ${SKY}"
```

Result:

`PHRASE = "Look, the sky is grey"`

```
COLOUR  = "blue"
SKY := "the sky is ${COLOUR}"
COLOUR  = "grey"
PHRASE = "Look, ${SKY}"
```

Result:

`PHRASE = "Look, the sky is blue"`

- Normal expansion is correct in most cases. Only use `:=` when really needed.

Operators: appending and prepending

- Variable values can be modified by composition:

  `+=` append (with space)
  `=+` prepend (with space)
  `.=` append (without space)
  `=.` prepend (without space)

Operators: default and weak default values

- The ?= operator assigns a value only if the variable has not been assigned when the statement is parsed

```
COLOUR ?= "unknown"
COLOUR = "blue"
```

Result: `COLOUR = "blue"`

```
COLOUR ?= "unknown"
```

Result: `COLOUR = "unknown"`

- The ??= operator assigns a value only if the variable has not been assigned when the statement is parsed, not even using a ?= operator

Operators caveats

- The operators apply their effect during parsing
- Example:

```
VAR ?= "a"
VAR += "b"
```

Result: `VAR = "a b"`

```
VAR += "b"
VAR ?= "a"
```

Result: `VAR = " b"`

- The parsing order of files is difficult to predict, no assumption should be made about it.
- To avoid the problem, avoid using `+=`, `=+`, `.=` and `=.` in `$BUILDDIR/conf/local.conf`. Always use overrides (see following slides).

Overrides

- Bitbake **overrides** allow appending, prepending or modifying a variable at expansion time, when the variable's value is read
- Overrides are written as `<VARIABLE>:<override> = "some_value"`
- A different syntax was used before **Honister** (3.4), with no retrocompatibility: `<VARIABLE>_<override> = "some_value"`

Overrides to modify variable values

- The `append` override adds **at the end** of the variable (without space).
  - `IMAGE_INSTALL:append = " dropbear"` adds `dropbear` to the packages installed on the image.
- The `prepend` override adds **at the beginning** of the variable (without space).
  - `PATH:prepend = "${COREBASE}/scripts/native-intercept:"` adds a path before the ones already present.
- The `remove` override removes all occurrences of a value within a variable.
  - `IMAGE_INSTALL:remove = "i2c-tools"`

Overrides for conditional assignment

- Append the machine name to only define a configuration variable for a given machine.
- It tries to match with values from OVERRIDES which includes MACHINE, SOC_FAMILY, and more.
- If the override is in OVERRIDES, the assignment is applied, otherwise it is ignored.

```
OVERRIDES="arm:armv7a:ti-soc:ti33x:beaglebone:poky"

KERNEL_DEVICETREE:beaglebone = "am335x-bone.dtb" # This is applied
KERNEL_DEVICETREE:dra7xx-evm = "dra7-evm.dtb"    # This is ignored
```

Overrides for conditional assignment: precedence

- The most specific assignment takes precedence.
- Example:

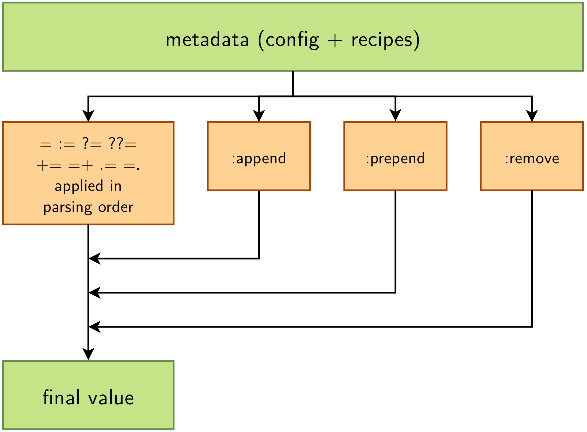
```
IMAGE_INSTALL:beaglebone = "busybox mtd-utils i2c-tools"
IMAGE_INSTALL = "busybox mtd-utils"
```

- If the machine is `beaglebone`:
  - `IMAGE_INSTALL = "busybox mtd-utils i2c-tools"`
- Otherwise:
  - `IMAGE_INSTALL = "busybox mtd-utils"`

Combining overrides

- The previous methods can be combined.
- If we define:
  - `IMAGE_INSTALL = "busybox mtd-utils"`
  - `IMAGE_INSTALL:append = " dropbear"`
  - `IMAGE_INSTALL:append:beaglebone = " i2c-tools"`
- The resulting configuration variable will be:
  - `IMAGE_INSTALL = "busybox mtd-utils dropbear i2c-tools"` if the machine being built is `beaglebone`.
  - `IMAGE_INSTALL = "busybox mtd-utils dropbear"` otherwise.

Order of variable assignment



1. All the operators are applied, in parsing order
2. `:append` overrides are applied
3. `:prepend` overrides are applied
4. `:remove` overrides are applied

bitbake-getvar

- bitbake-getvar can be used to understand and debug how variables are assigned
- bitbake-getvar <VARIABLE>
- Lists each configuration file touching the variable, the pre-expansion value and the final value

```
$ bitbake-getvar DEPLOY_DIR
NOTE: Starting bitbake server...
#
# $DEPLOY_DIR [2 operations]
#   set? /home/user/yocto-labs/poky/meta/conf/bitbake.conf:440
#     "${TMPDIR}/deploy"
#   set /home/user/yocto-labs/poky/meta/conf/documentation.conf:137
#     [doc] "Points to the general area that the OpenEmbedded build system uses to place images, [...]"
# pre-expansion value:
#   "${TMPDIR}/deploy"
DEPLOY_DIR="/home/user/yocto-labs/build/tmp/deploy"
$
```

Package variants

Introduction to package variants

- Some packages have the same purpose, and only one can be used at a time.
- The build system uses **virtual packages** to reflect this. A virtual package describes functionalities and several packages may provide it.
- Only one of the packages that provide the functionality will be compiled and integrated into the resulting image.

Variant examples

- The virtual packages are often in the form `virtual/<name>`
- Example of available virtual packages with some of their variants:
  - `virtual/bootloader`: u-boot, u-boot-ti-staging...
  - `virtual/kernel`: linux-yocto, linux-yocto-tiny, linux-yocto-rt, linux-ti-staging...
  - `virtual/libc`: glibc, musl, newlib
  - `virtual/xserver`: xserver-xorg

Package selection

- Variants are selected thanks to the PREFERRED_PROVIDER configuration variable.
- The package names **have to** suffix this variable.
- Examples:
  - PREFERRED_PROVIDER_virtual/kernel ?= "linux-ti-staging"
  - PREFERRED_PROVIDER_virtual/libgl = "mesa"

Version selection

- By default, Bitbake will try to build the provider with the highest version number, from the highest priority layer, unless the recipe defines `DEFAULT_PREFERENCE = "-1"`
- When multiple package versions are available, it is also possible to explicitly pick a given version with `PREFERRED_VERSION`.
- The package names **have to** suffix this variable.
- **%** can be used as a wildcard.
- Example:
  - `PREFERRED_VERSION_nginx = "1.20.1"`
  - `PREFERRED_VERSION_linux-yocto = "5.14%"`

Selection of packages to install

Selection of packages to install

- The set of packages installed into the image is defined by the target you choose (e.g. `core-image-minimal`).
- It is possible to have a custom set by defining our own target, and we will see this later.
- When developing or debugging, adding packages can be useful, without modifying the recipes.
- Packages are controlled by the `IMAGE_INSTALL` configuration variable.

The power of BitBake

Common BitBake options

- BitBake can be used to run a full build for a given target with `bitbake [target]`
  - `target` is a recipe name, possibly with modifiers, e.g. `-native`
  - `bitbake ncurses`
  - `bitbake ncurses-native`
- But it can be more precise, with additional options:

`-c <task>`	execute the given task
`-s`	list all available recipes and their versions
`-f`	force the given task to be run by removing its stamp file
`world`	keyword for all recipes

BitBake examples

- `bitbake -c listtasks virtual/kernel`
  - Gives a list of the available tasks for the recipe providing the package `virtual/kernel`. Tasks are prefixed with `do_`.
- `bitbake -c menuconfig virtual/kernel`
  - Execute the task `menuconfig` on the recipe providing the `virtual/kernel` package.
- `bitbake -f dropbear`
  - Force the `dropbear` recipe to run all tasks.
- `bitbake --runall=fetch core-image-minimal`
  - Download all recipe sources and their dependencies.
- For a full description: `bitbake --help`

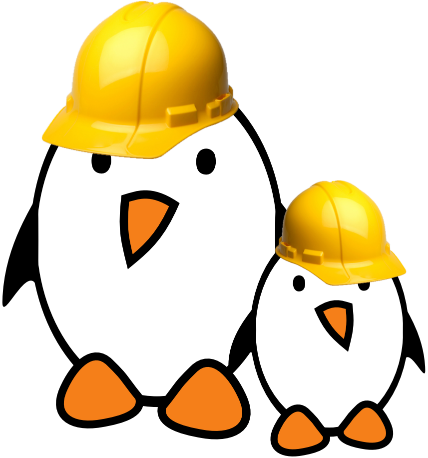
shared state cache

- BitBake stores the output of each task in a directory, the shared state cache.
- This cache is used to speed up compilation.
- Its location is defined by the SSTATE_DIR variable and defaults to build/sstate-cache.
- Over time, as you compile more recipes, it can grow quite big. It is possible to clean old data with:

```
$ find sstate-cache/ -type f -atime +30 -delete
```

This removes all files that have last been accessed more than 30 days ago (for example).

Practical lab - Advanced Yocto configuration

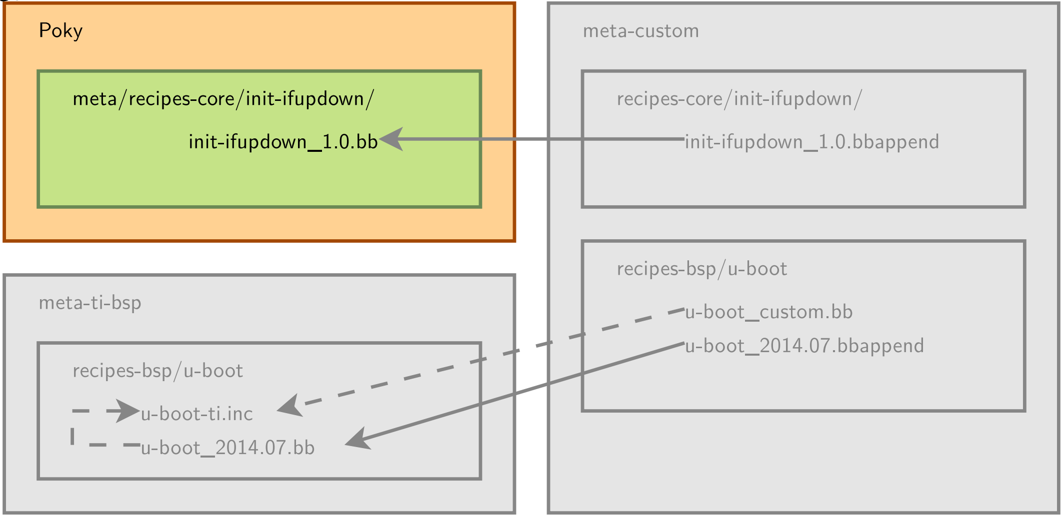


- Modify the build configuration
- Customize the package selection
- Experiment with BitBake
- Mount the root file system over NFS

Writing recipes - basics

Recipes: overview

Recipes


Poky
meta/recipes-core/init-ifupdown/
init-ifupdown_1.0.bb
meta-custom
recipes-core/init-ifupdown/
init-ifupdown_1.0.bbappend
recipes-bsp/u-boot
u-boot_custom.bb
u-boot_2014.07.bbappend
meta-ti-bsp
recipes-bsp/u-boot
u-boot-ti.inc
u-boot_2014.07.bb

Basics

- A recipe describes how to handle a given software component (application, library, …).
- It is a set of instructions to describe how to retrieve, patch, compile, install and generate binary packages.
- It also defines what build or runtime dependencies are required.
- Recipes are parsed by the `bitbake` build engine.
- The format of a recipe file name is `<application-name>_<version>.bb`
- The output product of a recipe is a set of binary packages (rpm, deb or ipk): typically `<recipename>`, `<recipename>-doc`, `<recipename>-dbg` etc.

Content of a recipe

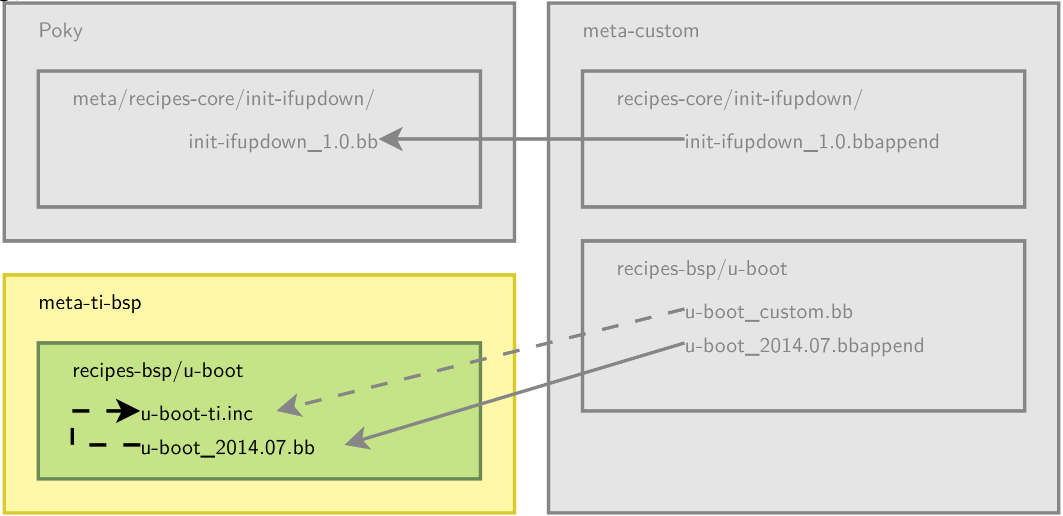
- A recipe contains configuration variables: name, license, dependencies, path to retrieve the source code…
- It also contains functions that can be run (fetch, configure, compile…) which are called **tasks**.
- Tasks provide a set of actions to perform.
- Remember the `bitbake -c <task> <target>` command?

Common variables

- To make it easier to write a recipe, some variables are automatically available:
  - BPN: recipe name, extracted from the recipe file name
  - PN: BPN potentially with prefixes or suffixes added such as `nativesdk-`, or `-native`
  - PV: package version, extracted from the recipe file name
  - BP: defined as `${BPN}-${PV}`
- The recipe name and version usually match the upstream ones.
- When using the recipe `bash_5.1.bb`:
  - `${BPN} = "bash"`
  - `${PV} = "5.1"`

Organization of a recipe

Organization of a recipe



Organization of a recipe

- Many applications have more than one recipe, to support different versions. In that case the common metadata is included in each version specific recipe and is in a `.inc` file:
  - `<application>.inc`
    - version agnostic metadata
  - `<application>_<version>.bb`
    - `require <application>.inc`
    - any version specific metadata
- We can divide a recipe into three main parts:
  - The header: what/who
  - The sources: where
  - The tasks: how

The header

Configuration variables to describe the application:

- `SUMMARY`: short descrition for the package manager
- `DESCRIPTION`: describes what the software is about
- `HOMEPAGE`: URL to the project's homepage
- `SECTION`: package category (e.g. `console/utils`)
- `LICENSE`: the application's license, using SPDX identifiers (https://spdx.org/licenses/)

The source locations: overview

- We need to retrieve both the raw sources from an official location and the resources needed to configure, patch or install the application.
- `SRC_URI` defines where and how to retrieve the needed elements. It is a set of URI schemes pointing to the resource locations (local or remote).
- URI scheme syntax: `scheme://url;param1;param2`
- `scheme` can describe a local file using `file://` or remote locations with `https://`, `git://`, `svn://`, `hg://`, `ftp://`...
- By default, sources are fetched in `$BUILDDIR/downloads`. Change it with the `DL_DIR` variable in `conf/local.conf`

The source locations: remote files 1/2

- The `http`, `https` and `ftp` schemes:
  - `https://example.com/application-1.0.tar.bz2`
  - A few variables are available to help pointing to remote locations: `${SOURCEFORGE_MIRROR}`, `${GNU_MIRROR}`, `${KERNELORG_MIRROR}`...
  - Example: `${SOURCEFORGE_MIRROR}/<project-name>/${BPN}-${PV}.tar.gz`
  - See `meta/conf/bitbake.conf`
- The `git` scheme:
  - `git://<url>;protocol=<protocol>;branch=<branch>`
  - When using git, it is necessary to also define SRCREV. It has to be a commit hash and not a tag to be able to do offline builds (a git tag can change, you then need to connect to the repository to check for a possible update). The `branch` parameter is mandatory as a safety check that SRCREV is on the expected branch.

The source locations: remote files 2/2

- An md5 or an sha256 sum must be provided when the protocol used to retrieve the file(s) does not guarantee their integrity. This is the case for `https`, `http` or `ftp`.

```
SRC_URI[md5sum] = "97b2c3fb082241ab5c56ab728522622b"
SRC_URI[sha256sum] = "..."
```

- It's possible to use checksums for more than one file, using the `name` parameter:

```
SRC_URI = "http://example.com/src.tar.bz2;name=tarball \
           http://example.com/fixes.patch;name=patch"

SRC_URI[tarball.md5sum] = "97b2c3fb082241ab5c56..."
SRC_URI[patch.md5sum]   = "b184acf9eb39df794ffd..."
```

The source locations: local files

- SRC_URI items using the file:// scheme are *local files*
- They are not downloaded, but rather copied from the layer to the work directory
- The searched paths are defined in the FILESPATH variable
- FILESPATH is a colon-separated list of paths to look for files
- The order matters: when a file is found in a path, the search ends

FILESPATH 1/3

- FILESPATH is generated with all combinations of:
- Base paths
  - `${FILE_DIRNAME}/${BP}` (e.g. BP = dropbear-2020.81)
  - `${FILE_DIRNAME}/${BPN}` (e.g. BPN = dropbear)
  - `${FILE_DIRNAME}/files`
  - Items in FILESEXTRAPATHS (none by default)
  - `${FILE_DIRNAME}` is the directory containing the `.bb` file
- The overrides in FILESOVERRIDES
  - Set as `${TRANSLATED_TARGET_ARCH}:${MACHINEOVERRIDES}:${DISTROOVERRIDES}`
  - E.g. `arm:armv7a:ti-soc:ti33x:beaglebone:poky`
  - Applied right to left

FILESPATH 2/3

- This results in a long list, including:
  - `/.../meta/recipes-core/dropbear/dropbear-2020.81/poky`
  - `/.../meta/recipes-core/dropbear/dropbear/poky`
  - `/.../meta/recipes-core/dropbear/files/poky`
  - `/.../meta/recipes-core/dropbear/dropbear-2020.81/beaglebone`
  - `/.../meta/recipes-core/dropbear/dropbear/beaglebone`
  - `/.../meta/recipes-core/dropbear/files/beaglebone`
  - `/.../meta/recipes-core/dropbear/dropbear-2020.81/ti33x`
  - `/.../meta/recipes-core/dropbear/dropbear/ti33x`
  - `/.../meta/recipes-core/dropbear/files/ti33x`
  - ...
  - `/.../meta/recipes-core/dropbear/dropbear-2020.81/armv7a`
  - `/.../meta/recipes-core/dropbear/dropbear/armv7a`
  - `/.../meta/recipes-core/dropbear/files/armv7a`
  - ...
  - `/.../meta/recipes-core/dropbear/dropbear-2020.81/`
  - `/.../meta/recipes-core/dropbear/dropbear/`
  - `/.../meta/recipes-core/dropbear/files/`

FILESPATH 3/3

- This complex logic allows to use different files without conditional code
- Example: with a single item in SRC_URI:

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

a different defconfig can be used for different MACHINE values:

```
recipes-kernel/
    linux
        my-linux
            mymachine1
                defconfig <-- used when MACHINE="mymachine1"
            mymachine2
                defconfig <-- used when MACHINE="mymachine2"
            defconfig     <-- used for any other MACHINE value
        my-linux_6.4.bb
```

The source locations: tarballs

- When extracting a tarball, `bitbake` expects to find the extracted files in a directory named `<application>-<version>`. This is controlled by the `S` variable. If the directory has another name, you must explicitly define `S`.
- If the scheme is `git`, `S` must be set to `${WORKDIR}/git`

The source locations: license files

- License files must have their own checksum.
- LIC_FILES_CHKSUM defines the URI pointing to the license file in the source code as well as its checksum.

```
LIC_FILES_CHKSUM = "file://gpl.txt;md5=393a5ca..."
LIC_FILES_CHKSUM = \
    "file://main.c;beginline=3;endline=21;md5=58e..."
LIC_FILES_CHKSUM = \
    "file://${COMMON_LICENSE_DIR}/MIT;md5=083..."
```

- This allows to track any license update: if the license changes, the build will trigger a failure as the checksum won't be valid anymore.

Dependencies 1/2

- A recipe can have dependencies during the build or at runtime. To reflect these requirements in the recipe, two variables are used:
  - `DEPENDS`: List of the recipe build-time dependencies.
  - `RDEPENDS`: List of the package runtime dependencies. Must be package specific (e.g. with `:${PN}`).
- `DEPENDS = "recipe-b"`: the local `do_prepare_recipe_sysroot` task depends on the `do_populate_sysroot` task of recipe-b.
- `RDEPENDS:${PN} = "package-b"`: the local `do_build` task depends on the `do_package_write_<archive-format>` task of recipe b.

Dependencies 2/2

- Sometimes a recipe has dependencies on specific versions of another recipe.
- `bitbake` allows to reflect this by using:
  - `DEPENDS = "recipe-b (>= 1.2)"`
  - `RDEPENDS:${PN} = "recipe-b (>= 1.2)"`
- The following operators are supported: `=`, `>`, `<`, `>=` and `<=`.
- A graphical tool can be used to explore dependencies or reverse dependencies:
  - `bitbake -g -u taskexp core-image-minimal`

Tasks

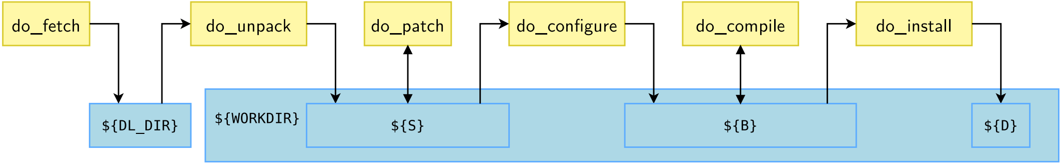
Default tasks already exist, they are defined in classes:

- do_fetch
- do_unpack
- do_patch
- do_configure
- do_compile
- do_install
- do_package
- do_rootfs

You can get a list of existing tasks for a recipe with:

`bitbake <recipe> -c listtasks`

The main tasks



Writing tasks 1/2

- Syntax of a task:

```
do_task() {
    action0
    action1
    ...
}
```

- Functions use the sh shell syntax, with available OpenEmbedded variables and internal functions available.
  - `WORKDIR`: the recipe's working directory
  - `S`: The directory where the source code is extracted
  - `B`: The directory where `bitbake` places the objects generated during the build
  - `D`: The destination directory (root directory of where the files are installed, before creating the image).

Writing tasks 2/2

- Example:

```
do_compile() {
    oe_runmake
}

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

Adding new tasks

- Tasks can be added with addtask

```
do_mkimage () {
    uboot-mkimage ...
}

addtask do_mkimage after do_compile before do_install
```

- Tasks are commonly added by classes

Applying patches

Patch use cases

Patches can be applied to resolve build-system problematics:

- To support old versions of a software: bug and security fixes.
- To fix cross-compilation issues.
- To apply patches before they make their way into the upstream version.

However, there are cases when patching a `Makefile` is unnecessary:

- For example, when an upstream `Makefile` uses hardcoded `CC` and/or `CFLAGS`.
- You can call `make` with the `-e` option which gives precedence to variables taken from the environment:

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

The source locations: patches

- Files ending in .patch, .diff or having the apply=yes parameter will be applied after the sources are retrieved and extracted, during the do_patch task.
  - Compressed patches with .gz, .bz2, .xz or .Z suffix are automatically decompressed

```
SRC_URI += "file://joystick-support.patch \
            file://smp-fixes.diff \
           "
```

- Patches are applied in the order they are listed in SRC_URI.
- It is possible to select which tool will be used to apply the patches listed in SRC_URI variable with PATCHTOOL.
- By default, PATCHTOOL = 'quilt' in Poky.
- Possible values: git, patch and quilt.

Resolving conflicts

- The `PATCHRESOLVE` variable defines how to handle conflicts when applying patches.
- It has two valid values:
  - `noop`: the build fails if a patch cannot be successfully applied.
  - `user`: a shell is launched to resolve manually the conflicts.
- By default, `PATCHRESOLVE = "noop"` in `meta-poky`.

Example of a recipe

Hello world recipe

```
SUMMARY = "Hello world program"
DESCRIPTION = "Hello world program"
HOMEPAGE = "http://example.net/hello/"
SECTION = "examples"
LICENSE = "GPL-2.0-or-later"

SRC_URI = "git://git.example.com/hello;protocol=https;branch=master"
SRCREV = "2d47b4eb66e705458a17622c2e09367300a7b118"
S = "${WORKDIR}/git"
LIC_FILES_CHKSUM = "file://hello.c;beginline=3;endline=21;md5=58e..."

do_compile() {
    oe_runmake
}
do_install() {
    install -d ${D}${bindir}
    install -m 0755 hello ${D}${bindir}
}
```

Example of a recipe with a version agnostic part

tar.inc

```
SUMMARY = "GNU file archiving program"
HOMEPAGE = "https://www.gnu.org/software/tar/"
SECTION = "base"

SRC_URI = "${GNU_MIRROR}/tar/tar-${PV}.tar.bz2"

do_configure() { ... }

do_compile() { ... }

do_install() { ... }
```

tar_1.17.bb

```
require tar.inc

LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM =  \
  "file://COPYING;md5=59530bdf33659b29e73d4adb9f9f6552"

SRC_URI += "file://avoid_heap_overflow.patch"

SRC_URI[md5sum] = "c6c4f1c075dbf0f75c29737faa58f290"
```

tar_1.26.bb

```
require tar.inc

LICENSE = "GPL-3.0-only"
LIC_FILES_CHKSUM = \
  "file://COPYING;md5=d32239bcb673463ab874e80d47fae504"

SRC_URI[md5sum] = "2cee42a2ff4f1cd4f9298eeeb2264519"
```

Debugging recipes

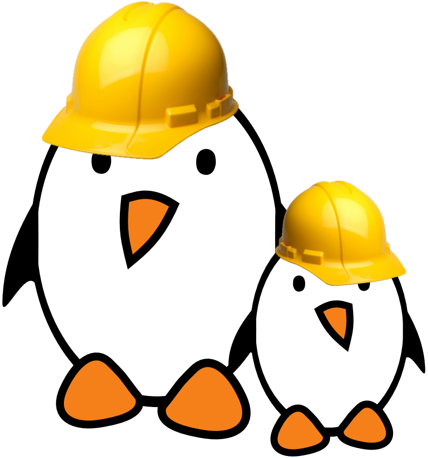
Log and run files

- For each task, these files are generated in the `temp` directory under the recipe work directory
- `run.do_<taskname>`
  - the script generated from the recipe content and executed to run the task
- `log.do_<taskname>`
  - the output of the task execution
- These can be inspected to understand what is being done by the tasks

Debugging variable assignment

- `bitbake-getvar` can dump the per-recipe variable value using the `-r` option
  - `bitbake-getvar -r ncurses SRC_URI`
- Similarly, `bitbake -e` dumps the entire environment, and also the task code
  - `bitbake -e`
  - `bitbake -e ncurses`

Practical lab - Add a custom application



- Write a recipe for a custom application
- Integrate it in the image

Writing recipes - advanced

Extending a recipe

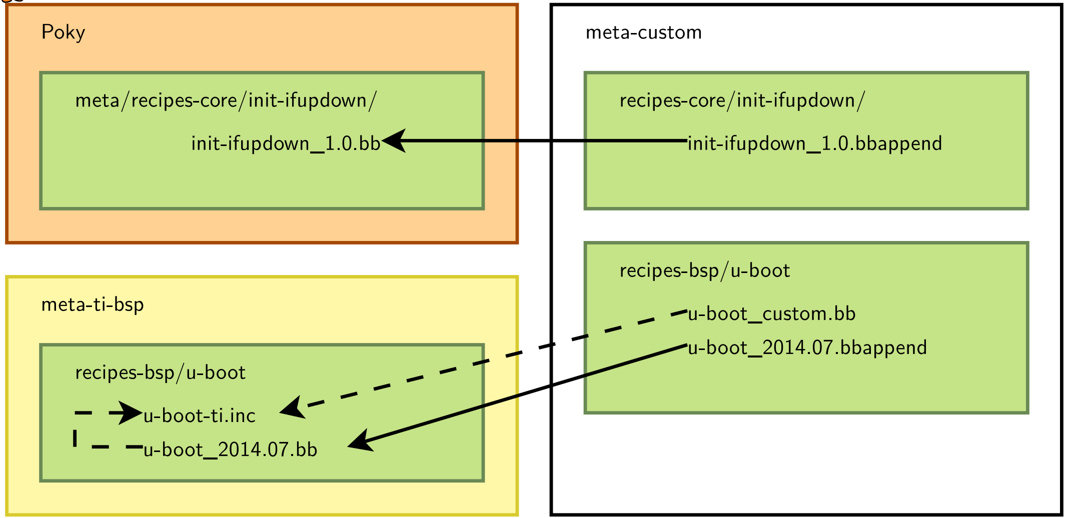
Introduction to recipe extensions

- It is a good practice to avoid modifying recipes available in third party layers so it is easy to update.
- But it is sometimes useful to apply a custom patch or add a configuration file for example.
- The `bitbake` *build engine* allows to modify a recipe by extending it.
- Multiple extensions can be applied to a recipe.

Introduction to recipe extensions

- Variable values can be modified
  - Using operators to set, append or prepend
  - Using overrides to append, prepend or remove
- Tasks can be modified
  - Using overrides to append or prepend task code
  - Adding new tasks

Extend a recipe



Extend a recipe

- The recipe extensions end in `.bbappend`
- Append files must have the same root name as the recipe they extend, but can also use wildcards.
  - `example_0.1.bbappend` applies to `example_0.1.bb`
  - `example_0.%.bbappend` applies to `example_0.1.bb` and `example_0.2.bb` but not `example_1.0.bb`
  - The `%` works only just before the `.bbappend` suffix
- Append files should be **version specific**. If the recipe is updated to a newer version, the append files must also be updated.
- If adding new files, the path to their directory must be prepended to the `FILESEXTRAPATHS` variable.
  - Files are looked up in paths referenced in `FILESEXTRAPATHS`, from left to right.
  - Prepending a path makes sure it has priority over the recipe's one. This allows to override recipes' files.

Append file example

`linux-yocto_6.12.bbappend`:

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

SRC_URI += "file://defconfig \
           file://fix-memory-leak.patch \
           "
```

File organization:

```
meta-custom/
`-- recipes-kernel/
    `-- linux/
        |-- files/
        |   |-- defconfig
        |   `-- fix-memory-leak.patch
        `-- linux-yocto_6.12.bbappend
```

Modifying existing tasks

Tasks can be extended with `:prepend` or `:append`

```
do_install:append() {
    install -d ${D}${sysconfdir}
    install -m 0644 hello.conf ${D}${sysconfdir}
}
```

Overrides can also be used for extending tasks:

```
do_install:append:beaglebone() {
    install -d ${D}${nonarch_base_libdir}/firmware
    install -m 0644 firmware.bin ${D}${nonarch_base_libdir}/firmware
}
```

Providing virtual packages

Providing virtual packages

- `bitbake` allows to use virtual names instead of the actual package name. We saw a use case with *package variants*.
- The virtual name is specified through the `PROVIDES` variable.
- Several recipes can provide the same virtual name. Only one will be built and installed into the generated image.
- `PROVIDES += "virtual/kernel"`

Classes

Introduction to classes

- Classes provide an abstraction to common code, which can be re-used in multiple recipes.
- Common tasks do not have to be re-developed!
- Any metadata and task which can be put in a recipe can be used in a class.
- Classes extension is `.bbclass`
- Classes must be located in the `classes-recipe`, `classes-global`, or `classes` folders of a layer.

Using classes

- Recipes can use the classes located in the `classes-recipe` folder:
  - `inherit class1 class2 ...`
- A recipe can inherit from multiple classes.
- Classes in `classes-global` can be inherited from configuration files with `INHERIT`:
  - `INHERIT:append = " class1 class2"`
- Classes in `INHERIT` will be used by every built recipe.
- Classes in the `classes` folder can be used with `inherit` or `INHERIT`, as their usage is not clearly defined.

Common classes

- classes-global/base.bbclass
- classes-recipe/kernel.bbclass
- classes-recipe/autotools.bbclass
- classes-recipe/autotools-brokensep.bbclass
- classes-recipe/cmake.bbclass
- classes-recipe/meson.bbclass
- classes-recipe/native.bbclass
- classes-recipe/systemd.bbclass
- classes-recipe/update-rc.d.bbclass
- classes/useradd.bbclass
- ...

The base class

- Every recipe inherits the `base` class automatically.
- Defines basic common tasks with a default implementation:
  - `fetch`, `unpack`, `patch`
  - `configure`, `compile`, `install`
  - Utility tasks such as: `clean`, `listtasks`
- Automatically applies patch files listed in `SRC_URI`
- Defines mirrors: `SOURCEFORGE_MIRROR`, `DEBIAN_MIRROR`, `GNU_MIRROR`, `KERNELORG_MIRROR`...
- Defines `oe_runmake`, using `EXTRA_OEMAKE` to use custom arguments.

The kernel class

- Used to build Linux kernels.
- Defines tasks to configure, compile and install a kernel and its modules.
- Automatically applies a `defconfig` listed in `SRC_URI`

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

- The kernel is divided into several packages: `kernel`, `kernel-base`, `kernel-dev`, `kernel-modules`...
- Automatically provides the virtual package `virtual/kernel`.
- Configuration variables are available:
  - `KERNEL_IMAGETYPE`, defaults to `zImage`
  - `KERNEL_EXTRA_ARGS`
  - `INITRAMFS_IMAGE`

The autotools class

- Defines tasks and metadata to handle applications using the autotools build system (autoconf, automake and libtool):
  - `do_configure`: generates the configure script using `autoreconf` and loads it with standard arguments or cross-compilation.
  - `do_compile`: runs `make`
  - `do_install`: runs `make install`
- Extra configuration parameters can be passed with `EXTRA_OECONF`.
- Compilation flags can be added thanks to the `EXTRA_OEMAKE` variable.

Example: use the autotools class

```
DESCRIPTION = "Print a friendly, customizable greeting"
HOMEPAGE = "https://www.gnu.org/software/hello/"
SECTION = "examples"
LICENSE = "GPL-3.0-or-later"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"
SRC_URI[md5sum] = "67607d2616a0faaf5bc94c59dca7c3cb"
SRC_URI[sha256sum] = "ecbb7a2214196c57ff9340aa71458e1559abd38f6d8d169666846935df191ea7"
LIC_FILES_CHKSUM = "file://COPYING;md5=d32239bcb673463ab874e80d47fae504"

inherit autotools
```

The useradd class

- This class helps to add users to the resulting image.
- Adding custom users is required by many services to avoid running them as root.
- `USERADD_PACKAGES` must be defined when the `useradd` class is inherited. It defines the individual packages produced by the recipe that need users or groups to be added.
- Users and groups will be created before the packages using it perform their `do_install`.
- At least one of the two following variables must be set:
  - `USERADD_PARAM`: parameters to pass to `useradd`.
  - `GROUPADD_PARAM`: parameters to pass to `groupadd`.

Example: use the useradd class

```
DESCRIPTION = "useradd class usage example"
SECTION = "examples"
LICENSE = "MIT"

SRC_URI = "file://file0"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/files/common-licenses/MIT;md5=0835ade698e0bc..."

inherit useradd

USERADD_PACKAGES = "${PN}"
USERADD_PARAM:${PN} = "-u 1000 -d /home/user0 -s /bin/bash user0"

FILES:${PN} = "/home/user0/file0"

do_install() {
    install -d ${D}/home/user0/
    install -m 644 file0 ${D}/home/user0/
    chown user0:user0 ${D}/home/user0/file0
}
```

The bin_package class

- In some cases you only need to install pre-built files into the generated root filesystem
  - E.g.: firmware blobs
- `bin_package.bbclass` simplifies this
  - Disables `do_configure` and `do_compile`
  - Provides a default `do_install` that copies whatever is in `S` (useful e.g. when extracting a pre-built RPM/DEB)
- Additionally you probably need:
  - Remember to set the `LICENSE` to `CLOSED` if applicable
  - You probably should also `inherit allarch`

BitBake file inclusions

Locate files in the build system

- Metadata can be shared using included files.
- `BitBake` uses the BBPATH to find the files to be included. It also looks into the current directory.
- Three keywords can be used to include files from recipes, classes or other configuration files:
  - `inherit`
  - `include`
  - `require`

The `inherit` keyword

- `inherit` can be used in recipes or classes, to inherit the functionalities of a class.
- To inherit the functionalities of the `kernel` class, use: `inherit kernel`
- `inherit` looks for files ending in `.bbclass`, in `classes`, `classes-recipe` or `classes-global` directories found in BBPATH.
- It is possible to include a class conditionally using a variable: `inherit ${FOO}`
- Inheriting in configuration files is based on the INHERIT variable instead:
  - `INHERIT += "rm_work"`
  - This inherits the class globally (i.e. for all recipes)

The `include` and `require` keywords

- `include` and `require` can be used in all files, to insert the content of another file at that location.
- If the path specified on the `include` (or `require`) path is relative, `bitbake` will insert the first file found in BBPATH.
- `include` does not produce an error when a file cannot be found, whereas `require` raises a parsing error.
- To include a local file: `require ninvaders.inc`
- To include a file from another location (which could be in another layer): `require path/to/file.inc`

More recipe debugging tools

More recipe debugging tools

- A development shell, exporting the full environment can be used to debug build failures:

```
$ bitbake -c devshell <recipe>
```

- To understand what a change in a recipe implies, you can activate build history in `local.conf`:

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

Then use the `buildhistory-diff` tool to examine differences between two builds.

  - `buildhistory-diff`

Network usage

Source fetching

- bitbake will look for files to retrieve at the following locations, in order:
  1. DL_DIR (the local download directory).
  2. The PREMIRRORS locations.
  3. The upstream source, as defined in SRC_URI.
  4. The MIRRORS locations.
- If all the mirrors fail, the build will fail.

Mirror configuration in OpenEmbedded-Core

meta/classes-global/mirrors.bbclass

```
PREMIRRORS += "git://sourceware.org/git/glibc.git        https://downloads.yoctoproject.org/mirror/sources/ \
               git://sourceware.org/git/binutils-gdb.git https://downloads.yoctoproject.org/mirror/sources/"

MIRRORS += "\
svn://.*/.*     http://sources.openembedded.org/ \
git://.*/.*     http://sources.openembedded.org/ \
https?://.*/.*  http://sources.openembedded.org/ \
ftp://.*/.*     http://sources.openembedded.org/ \
...
"
```

Configuring the premirrors

- It is easy to add a custom mirror to the PREMIRRORS by using the own-mirrors class (only one URL supported):

```
INHERIT += "own-mirrors"
SOURCE_MIRROR_URL = "http://example.com/my-mirror"
```

- For a more complex setup, prepend custom mirrors to the PREMIRRORS variable:

```
PREMIRRORS:prepend = "\
 git://.*/.*   http://example.com/my-mirror-for-git/ \
 svn://.*/.*   http://example.com/my-mirror-for-svn/ \
 http://.*/.*  http://www.yoctoproject.org/sources/  \
 https://.*/.* http://www.yoctoproject.org/sources/  "
```

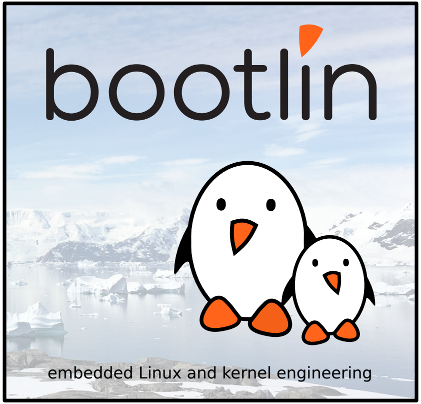
Creating a local mirror

- The download directory can be exposed on the network to create a local mirror
  - Except for sources fetched via an SCM a tarball of the repository is needed, not the bare git repository that is created by default
  - You can use `BB_GENERATE_MIRROR_TARBALLS = "1"` to generate tarballs of the git repositories in `DL_DIR`

Forbidding network access

- Since Kirkstone (4.0), network access is only enabled in the `do_fetch` task, to make sure no untraced sources are fetched.
- You can also completely disable network access using `BB_NO_NETWORK = "1"`
  - To download all the sources before disabling network access use `bitbake --runall=fetch core-image-minimal`
- Or restrict `bitbake` to only download files from the `PREMIRRORS`, using `BB_FETCH_PREMIRRORONLY = "1"`

Layers

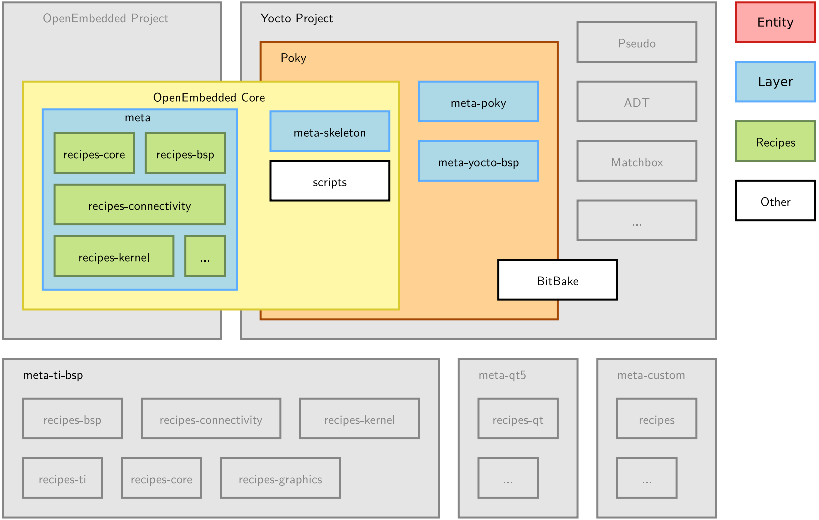


Introduction to layers

Layers principles

- The OpenEmbedded *build system* manipulates *metadata*.
- Layers allow to isolate and organize the metadata.
  - A layer is a collection of recipes.
- It is a good practice to begin a layer name with the prefix `meta-`.

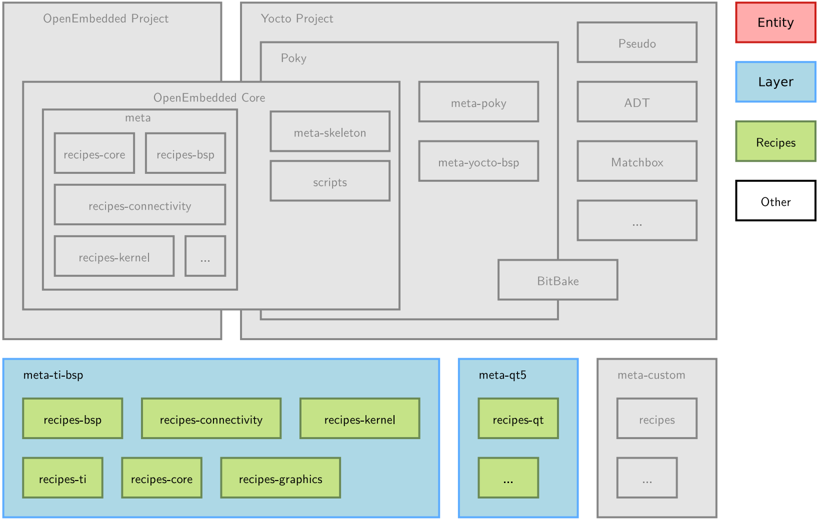
Layers in Poky



Layers in Poky

- The Poky *reference system* is a set of basic common layers:
  - meta
  - meta-skeleton
  - meta-poky
  - meta-yocto-bsp
- Poky is not a final set of layers. It is the common base.
- Layers are added when needed.
- When making modifications to the existing recipes or when adding new ones, it is a good practice to avoid modifying Poky. Instead you can create your own layers!

Third party layers



Integrate and use a layer 1/3

- A list of existing and maintained layers can be found at `https://layers.openembedded.org`
- Instead of redeveloping layers, always check the work hasn't been done by others.
- It takes less time to download a layer providing a package you need and to add an append file if some modifications are needed than to do it from scratch.

Integrate and use a layer 2/3

- The location where a layer is saved on the disk doesn't matter.
  - But a good practice is to save it where all others layers are stored.
- The only requirement is to let BitBake know about the new layer:
  - The list of layers BitBake uses is defined in `$BUILDDIR/conf/bblayers.conf`
  - To include a new layer, add its absolute path to the BBLAYERS variable.
  - BitBake parses each layer specified in BBLAYERS and adds the recipes, configurations files and classes it contains.

Integrate and use a layer 3/3

- The `bitbake-layers` tool is provided alongside `bitbake`.
- It can be used to inspect the layers and to manage `$BUILDDIR/conf/bblayers.conf`:
  - `bitbake-layers show-layers`
  - `bitbake-layers add-layer meta-custom`
  - `bitbake-layers remove-layer meta-qt5`

Some useful layers

- Many SoC specific layers are available, providing support for the boards using these SoCs. Some examples: `meta-ti-bsp`, `meta-freescale` and `meta-st-stm32mp`.
- Other layers offer to support applications not available in the Poky reference system:
  - `meta-browser`: web browsers (Chromium, Firefox).
  - `meta-filesystems`: support for additional filesystems.
  - `meta-java` and `meta-oracle-java`: Java support.
  - `meta-linaro-toolchain`: Linaro toolchain recipes.
  - `meta-qt5`: QT5 modules.
  - `meta-realtime`: real time tools and test programs.
  - `meta-telephony` and many more...

  Notice that some of these layers do not come with all the Yocto branches. `meta-realtime` layer does not have a `honister` (3.4) branch, for example.

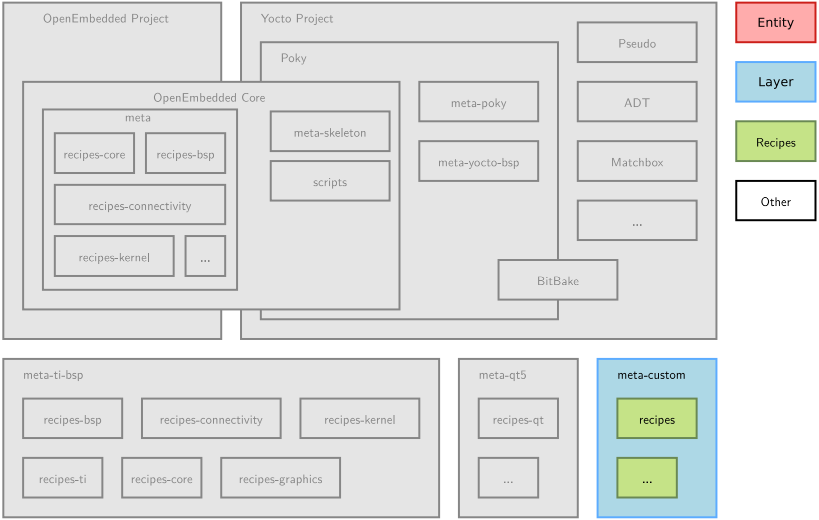
Layer recommendations

Layer recommendations

- Keep your build system simple: use very few layers initially
- Then add layers when needed based on benefit/cost ratio
  - The quality of several board/SoM layers is questionable
  - The quality of SoC vendor layers is varying
- A working example:
  - https://github.com/bootlin/simplest-yocto-setup
  - Minimal dependencies: OE-core, meta-arm
  - One company-specific layer: meta-kiss
  - Custom distro and machine configurations, image recipes
  - Can be used as a starting point for your project
  - Introduced at Yocto Project Summit 2023 (slides, video)

Creating a layer

Custom layer



Create a custom layer 1/2

- A layer is a set of files and directories and can be created by hand.
- However, the `bitbake-layers create-layer` command helps us create new layers and ensures this is done right.
- `bitbake-layers create-layer -p <PRIORITY> <layer>`
- The **priority** is used to select which recipe to use when multiple layers contains the same recipe
- The layer priority takes precedence over the recipe version number ordering. This allows to downgrade a recipe in a layer.

Create a custom layer 2/2

- The layer created will be pre-filled with the following files:

  conf/layer.conf The layer's configuration. Holds its priority and generic information. No need to modify it in many cases.
  - Mandatory, this is the entry point for the layer.

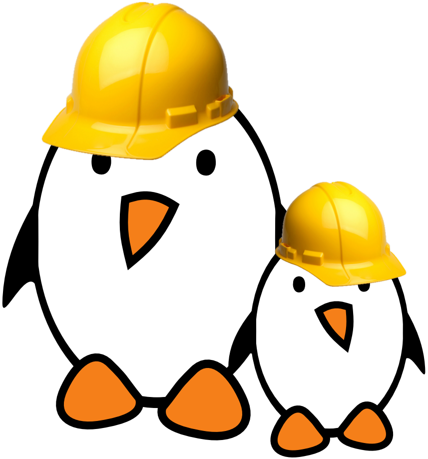
  COPYING.MIT The license under which a layer is released. By default MIT.

  README A basic description of the layer. Contains a contact e-mail to update.
- By default, all metadata matching `./recipes-*/*/*.bb` will be parsed by the BitBake *build engine*.

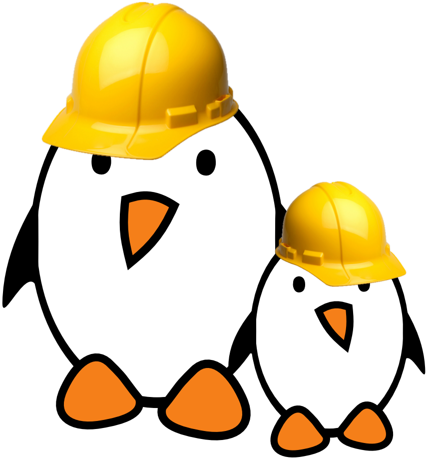
Creating a layer: best practices

- Do not copy and modify existing recipes from other layers. Instead use append files.
- Avoid duplicating files. Use append files or explicitly use a path relative to other layers.
- Save the layer alongside other layers.
- Use `LAYERDEPENDS` to explicitly define layer dependencies.
- Use `LAYERSERIES_COMPAT` to define the Yocto version(s) with which the layer is compatible.

Practical lab - Create a custom layer



- Create a layer from scratch
- Add recipes to the new layer
- Integrate it to the build

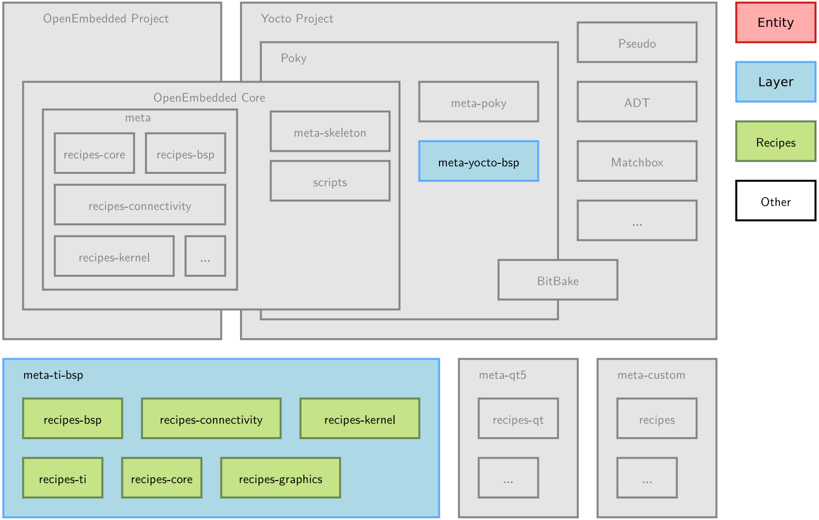


- Apply patches to an existing recipe
- Use a custom configuration file for an existing recipe
- Extend a recipe to fit your needs

BSP Layers

Introduction to BSP layers in the Yocto Project

BSP layers



OpenEmbedded Project
Yocto Project
Poky
Pseudo
OpenEmbedded Core
meta
recipes-core
recipes-bsp
recipes-connectivity
recipes-kernel
...
meta-skeleton
scripts
meta-poky
meta-yocto-bsp
ADT
Matchbox
...
BitBake
meta-ti-bsp
recipes-bsp
recipes-connectivity
recipes-kernel
recipes-ti
recipes-core
recipes-graphics
meta-qt5
recipes-qt
...
meta-custom
recipes
...
Entity
Layer
Recipes
Other

BSP layers overview

- BSP layers are a subset of the layers.
- They hold metadata with the purpose of supporting a specific class of hardware devices.
- They usually provide:
  - Hardware configuration files (`machines`)
  - Custom kernel and bootloader recipes and configurations
  - Modules and drivers to enable specific hardware features (e.g. multimedia accelerators)
  - Pre-built user binaries and firmware
- A good practice is to name it `meta-<bsp_name>`.
- Examples: `meta-ti-bsp`, `meta-st-stm32mp`.

Hardware configuration files

Overview 1/2

- A layer provides one machine file (hardware configuration file) per machine it supports.
- These configuration files are stored under `meta-<bsp_name>/conf/machine/*.conf`
- The file names correspond to the values set in the `MACHINE` configuration variable.
  - `meta-ti/meta-ti-bsp/conf/machine/beaglebone.conf`
  - `MACHINE = "beaglebone"`
- Each machine should be described in the `README` file of the BSP.

Overview 2/2

- The hardware configuration file contains configuration variables related to the architecture and to the machine features.
- Some other variables help customize the kernel image or the filesystems used.

Machine configuration

- TARGET_ARCH: The architecture of the device being built.
- PREFERRED_PROVIDER_virtual/kernel: The default kernel.
- MACHINE_FEATURES: List of hardware features provided by the machine, e.g. usbgadget usbhost screen wifi
- SERIAL_CONSOLES: Speed and device for the serial consoles to attach. Used to configure getty, e.g. 115200;ttyS0
- KERNEL_IMAGETYPE: The type of kernel image to build, e.g. zImage

- Lists the hardware features provided by the machine.
- These features are used by package recipes to enable or disable functionalities.
- Some packages are automatically added to the resulting root filesystem depending on the feature list.
  - The machine feature `keyboard` adds the `keymaps` to the image.

conf/machine/include/cfa10036.inc

```
# Common definitions for cfa-10036 boards
include conf/machine/include/imx-base.inc
include conf/machine/include/tune-arm926ejs.inc

SOC_FAMILY = "mxs:mx28:cfa10036"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-cfa"
PREFERRED_PROVIDER_virtual/bootloader ?= "barebox"
IMAGE_BOOTLOADER = "barebox"
BAREBOX_BINARY = "barebox"
IMAGE_FSTYPES:mxs = "tar.bz2 barebox.mxsboot-sdcard sdcard.gz"
IMXBOOTLETS_MACHINE = "cfa10036"

KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "imx28-cfa10036.dtb"
# we need the kernel to be installed in the final image
IMAGE_INSTALL:append = " kernel-image kernel-devicetree"
SDCARD_ROOTFS ?= "${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.ext3"
SERIAL_CONSOLES = "115200;ttyAMA0"
MACHINE_FEATURES = "usbgadget usbhost vfat"
```

conf/machine/cfa10057.conf

```
#@TYPE: Machine
#@NAME: Crystalfontz CFA-10057
#@SOC: i.MX28
#@DESCRIPTION: Machine configuration for CFA-10057, also called CFA-920
#@MAINTAINER: Alexandre Belloni <alexandre.belloni@bootlin.com>

require conf/machine/include/cfa10036.inc

KERNEL_DEVICETREE += "imx28-cfa10057.dtb"

MACHINE_FEATURES += "touchscreen"
```

Bootloader

Default bootloader 1/2

- By default (on ARM) the bootloader used is the mainline version of `U-Boot`, with a fixed version (per Poky release).
- All the magic is done in `meta/recipes-bsp/u-boot/u-boot.inc`
- Some configuration variables used by the U-Boot recipe can be customized, in the machine file.

Default bootloader 2/2

- SPL_BINARY: If an SPL is built, describes the name of the output binary. Defaults to an empty string.
- UBOOT_SUFFIX: `bin` (default) or `img`.
- UBOOT_MACHINE: The target used to build the configuration.
- UBOOT_ENTRYPOINT: The bootloader entry point.
- UBOOT_LOADADDRESS: The bootloader load address.
- UBOOT_MAKE_TARGET: Make target when building the bootloader. Defaults to `all`.

Customize the bootloader

- It is possible to support a custom U-Boot by creating an extended recipe and to append extra metadata to the original one.
- This works well when using a mainline version of U-Boot.
- Otherwise it is possible to create a custom recipe.
  - Try to still use `meta/recipes-bsp/u-boot/u-boot.inc`

Kernel

Linux kernel recipes in Yocto

- There are mainly two ways of compiling a kernel:
  - By creating a custom kernel recipe, inheriting `kernel.bbclass`
  - By using the `linux-yocto` packages, provided in Poky, for very complex needs
- The kernel used is selected in the machine file thanks to: `PREFERRED_PROVIDER_virtual/kernel`
- Its version is defined with: `PREFERRED_VERSION_<kernel_provider>`

Linux Yocto 1/3

- `linux-yocto` is a set of recipes with advanced features to build a mainline kernel
- `PREFERRED_PROVIDER_virtual/kernel = "linux-yocto"`
- `PREFERRED_VERSION_linux-yocto = "5.14%"`

Linux Yocto 2/3

- Automatically applies configuration fragments listed in SRC_URI with a .cfg extension

```
SRC_URI += "file://defconfig          \
            file://nand-support.cfg \
            file://ethernet-support.cfg"
```

Linux Yocto 3/3

- Another way of configuring `linux-yocto` is by using *Advanced Metadata*.
- It is a powerful way of splitting the configuration and the patches into several pieces.
- It is designed to provide a very configurable kernel, at the cost of higher complexity.
- The full documentation can be found at https://docs.yoctoproject.org/kernel-dev/advanced.html#working-with-advanced-metadata-yocto-kernel-cache

Linux Yocto: Kernel Metadata 1/2

- Kernel Metadata is a way to organize and to split the kernel configuration and patches in little pieces each providing support for one feature.
- Two main configuration variables help taking advantage of this:
  - `LINUX_KERNEL_TYPE`: `standard` (default), `tiny` or `preempt-rt`
    - `standard`: generic Linux kernel policy.
    - `tiny`: bare minimum configuration, for small kernels.
    - `preempt-rt`: applies the `PREEMPT_RT` patch.
  - `KERNEL_FEATURES`: List of features to enable. Features are sets of patches and configuration fragments.

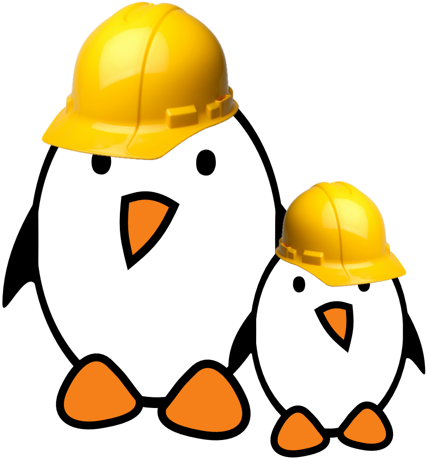
Linux Yocto: Kernel Metadata 2/2

- Kernel Metadata description files have their own syntax to describe an optional kernel feature
- A basic feature is defined as a patch to apply and a configuration fragment to add
- Simple example, `features/nunchuk.scc`

```
define KFEATURE_DESCRIPTION "Enable Nunchuk driver"

kconf hardware enable-nunchuk-driver.cfg
patch Add-nunchuk-driver.patch
```

- To integrate the feature into the kernel image:
  `KERNEL_FEATURES += "features/nunchuk.scc"`

Practical lab - Create a custom machine configuration

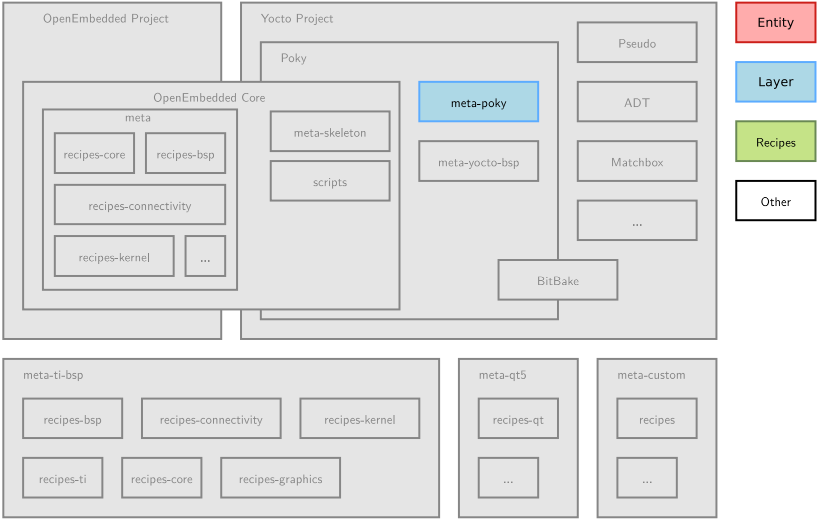


- Write a machine configuration
- Understand how the target architecture is chosen

Distro Layers

Distro Layers

Distro layers



Distro layers

- You can create a new distribution by using a Distro layer.
- This corresponds to the settings that have an impact on your packages. You can also decide to use Musl or Glibc, Wayland or X11, systemd or sysvinit…
- A distribution layer allows to change the defaults that are provided by `openembedded-core` or `poky`.
- It is useful to distribute changes that have been made in `local.conf`
- Note: Poky is a rather bloated distribution, mainly meant to be used for testing. It's not necessarily a good starting point to optimize the root filesystem for your own platform.

Best practice

- A distro layer is used to provide policy configurations for a custom distribution.
- It is a best practice to separate the distro layer from the custom layers you may create and use.
- It often contains:
  - Configuration files.
  - Specific classes (for example to sign images)
  - Distribution specific recipes: initialization scripts, splash screen…

Creating a Distro layer

- The configuration file for the distro layer is `conf/distro/<distro>.conf`
- This file must define the `DISTRO` variable.
- It is possible to inherit configuration from an existing distro layer.
- You can also use all the `DISTRO_*` variables.
- Use `DISTRO = "<distro>"` in `local.conf` to use your distro configuration.

```
require conf/distro/poky.conf

DISTRO = "distro"
DISTRO_NAME = "distro description"
DISTRO_VERSION = "1.0"

MAINTAINER = "..."
```

- Lists the features the distribution will enable (SSL, WiFi, Bluetooth...).
- As for MACHINE_FEATURES, this is used by package recipes to enable or disable functionalities.
- For example, the bluetooth feature:
  - Asks the bluez daemon to be built and added to the image.
  - Enables bluetooth support in ConnMan.
- COMBINED_FEATURES provides the list of features that are enabled in both MACHINE_FEATURES and DISTRO_FEATURES.

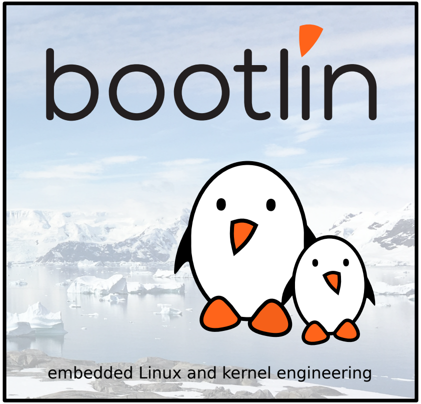
Toolchain selection

- The toolchain selection is controlled by the `TCMODE` variable.
- It defaults to `"default"`.
- The `conf/distro/include/tcmode-${TCMODE}.inc` file is included.
  - This configures the toolchain to use by defining preferred providers and versions for recipes such as `gcc`, `binutils`, `*libc`...
- The providers' recipes define how to compile or/and install the toolchain.
- Toolchains can be built by the build system or external (rarely used because toolchains are fast to rebuild thanks to the shared state cache).

Sample files

- A distro layer often contains `sample files`, used as templates to build key configurations files.
- Example of `sample files`:
  - `bblayers.conf.sample`
  - `local.conf.sample`
- In Poky, they are in `meta-poky/conf/templates/default/`.
- The TEMPLATECONF environment variable controls where to find the samples and can be exported before sourcing `oe-init-build-env`.
- `##OEROOT##` can be used in `bblayers.conf.sample` and will be replaced by the path to the directory containing the `oe-init-build-env` script, after `oe-init-build-env` is sourced.
- `bitbake-layers save-build-conf` can be used to save the current configuration.

Images



Introduction to images

Overview 1/3

- An `image` is the top level recipe and is used alongside the `machine` definition.
- Whereas the `machine` describes the hardware used and its capabilities, the `image` is architecture agnostic and defines how the root filesystem is built, with what packages.
- By default, several images are provided in Poky:
  - `meta*/recipes*/images/*.bb`

Overview 2/3

- Here are a few common images:

  core-image-base Console-only image, with full support of the hardware.

  core-image-minimal Small image, capable of booting a device.

  core-image-minimal-dev Small image with extra tools, suitable for development.

  core-image-x11 Image with basic X11 support.

  core-image-weston Image with basic Wayland support.

  core-image-rt Like `core-image-minimal` with real time tools and test suite.

Overview 3/3

- An `image` is no more than a recipe.
- It has a description, a license (optional) and inherits the `core-image` class.

Organization of an image recipe

Some special configuration variables are used to describe an image:

- `IMAGE_BASENAME`: The name of the output image files. Defaults to `${PN}`.
- `IMAGE_INSTALL`: List of packages and package groups to install in the generated image (only toplevel packages, dependencies unnecessary)
- `IMAGE_ROOTFS_SIZE`: The final root filesystem size.
- `IMAGE_FEATURES`: List of features to enable in the image (e.g. `allow-root-login`).
- `IMAGE_FSTYPES`: List of formats the OpenEmbedded build system will use to create images. Could be set in machine definitions too (machine dependent).
- `IMAGE_LINGUAS`: List of the locales to be supported in the image.
- `IMAGE_PKGTYPE`: Package type used by the build system. One of `deb`, `rpm`, `ipk` and `tar`.
- `IMAGE_POSTPROCESS_COMMAND`: Shell commands to run at post process.
- `EXTRA_IMAGEDEPENDS`: Recipes to be built with the image, but which do not install anything in the root filesystem (e.g. the bootloader).

Example of an image

```
SUMMARY = "Example image"
IMAGE_INSTALL = "packagegroup-core-boot dropbear ninvaders"
IMAGE_LINGUAS = " "

inherit core-image
```

Note: unlike other recipes, image recipes don't need to set `LICENSE`.

Root filesystem generation

- Image generation overview:
  1. An empty directory is created for the root filesystem.
  2. Packages from IMAGE_INSTALL are installed into it using the package manager.
  3. One or more images files are created, depending on the IMAGE_FSTYPES value.
- Root filesystem creation is specific to the IMAGE_PKGTYPE value. It should be defined in the image recipe, otherwise the first valid package type defined in PACKAGE_CLASSES is used.
- All the magic is done in `meta/classes-recipe/rootfs_${IMAGE_PKGTYPE}.bbclass`

Image types

IMAGE_FSTYPES

- Configures the resulting root filesystem image format.
- If more than one format is specified, one image per format will be generated.
- Image formats instructions are provided by `openembedded-core`, in `meta/classes-recipe/image_types.bbclass`
- Common image formats are: `ext2`, `ext3`, `ext4`, `squashfs`, `squashfs-xz`, `cpio`, `jffs2`, `ubifs`, `tar.bz2`, `tar.gz`…

Creating an image type

- If you have a particular layout on your storage (for example bootloader location on an SD card), you may want to create your own image type.
- This is done through a class that inherits from `image_types`.
- It has to define a function named `IMAGE_CMD:<type>`.
- Append it to `IMAGE_TYPES`

Creating an image conversion type

- Common conversion types are: `gz`, `bz2`, `sha256sum`, `bmap`...
- This is done through a class that inherits from `image_types`.
- It has to define a function named `CONVERSION_CMD:<type>`.
- Append it to `CONVERSIONTYPES`
- Append valid combinations to `IMAGE_TYPES`

- wic is a tool that can create a flashable image from the compiled packages and artifacts.
- It can create partitions (but doesn't support raw flash partitions and filesystems)
- It can select which files are located in which partition through the use of plugins.
- The final image layout is described in a .wks or .wks.in file.
- It can be extended in any layer.
- Usage example:

```
WKS_FILE = "imx-uboot-custom.wks.in"
IMAGE_FSTYPES = "wic.bmap wic"
```

- Note: bmaptool is an alternative to dd, skipping uninitialized contents in partitions.

imx-uboot-custom.wks.in

```
part u-boot --source rawcopy --sourceparams="file=imx-boot" --no-table --align ${IMX_BOOT_SEEK}
part /boot --source bootimg-partition --use-uuid --fstype=vfat --label boot --active --align 8192 --size 64
part / --source rootfs --use-uuid --fstype=ext4 --label root --exclude-path=home/ --exclude-path=opt/ --align 8192
part /home --source rootfs --rootfs-dir=${IMAGE_ROOTFS}/home --use-uuid --fstype=ext4 --label home --align 8192
part /opt --source rootfs --rootfs-dir=${IMAGE_ROOTFS}/opt --use-uuid --fstype=ext4 --label opt --align 8192

bootloader --ptable msdos
```

- Copies `imx-boot` from `$DEPLOY_DIR_IMAGE` in the image, aligned on (and so at that offset) `${IMX_BOOT_SEEK}`.
- Creates a first partition, formatted in FAT32, with the files listed in the `IMAGE_BOOT_FILES` variable.
- Creates an `ext4` partition with the contents on the root filesystem, excluding the content of `/home` and `/opt`
- Creates two `ext4` partitions, one with the content of `/home`, the other one with the content of `/opt`, from the image root filesystem.

Package groups

Overview

- Package groups are a way to group packages by functionality or common purpose.
- Package groups are used in image recipes to help building the list of packages to install.
- A package group is yet another recipe.
  - Using the `packagegroup` class.
  - The generated binary packages do not install any file, but they require other packages.
- Be careful about the `PACKAGE_ARCH` value:
  - Set to the value `all` by default,
  - Must be explicitly set to `${MACHINE_ARCH}` when there is a machine dependency.

Common package groups

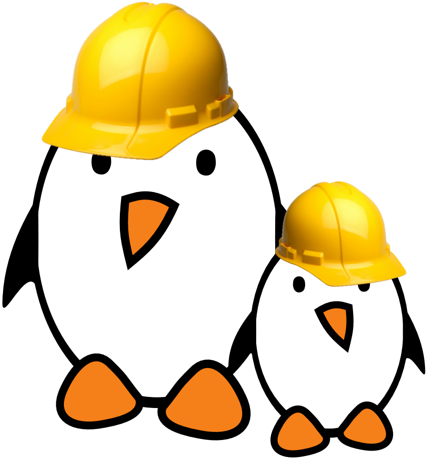
- packagegroup-base
  - Adds many core packages to the image based on MACHINE_FEATURES and DISTRO_FEATURES
- packagegroup-core-boot
- packagegroup-core-buildessential
- packagegroup-core-nfs-client
- packagegroup-core-nfs-server
- packagegroup-core-tools-debug
- packagegroup-core-tools-profile

Example

./meta/recipes-core/packagegroups/packagegroup-core-tools-debug.bb:

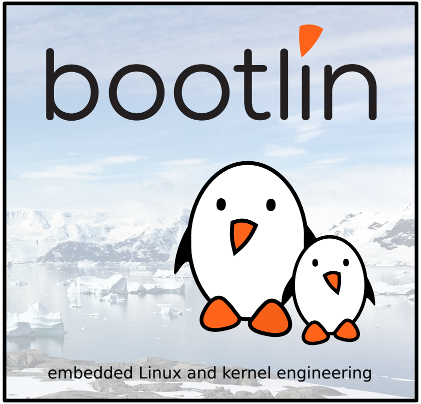
```
SUMMARY = "Debugging tools"

inherit packagegroup

RDEPENDS:${PN} = "\
    gdb \
    gdbserver \
    strace"
```

Practical lab - Create a custom image



- Write an image recipe
- Choose the packages to install

Writing recipes - going further



The per-recipe sysroot

Sysroot

- The *sysroot* is the logical root directory for headers and libraries
- Where *gcc* looks for headers, and *ld* looks for libraries
- Contains:
  - The kernel headers
  - The C library and headers
  - Other libraries and their headers

Optional dependencies

- Most software have a configure script that checks for libraries:
  - Fails if mandatory dependencies are not found
  - Fails if explicitly requested optional dependencies are not found
  - Enables more features if automatic dependencies are found
- Problem with automatic dependencies: result can depend on build order. Example:
  - `libpcap` built *before* `bluez5`

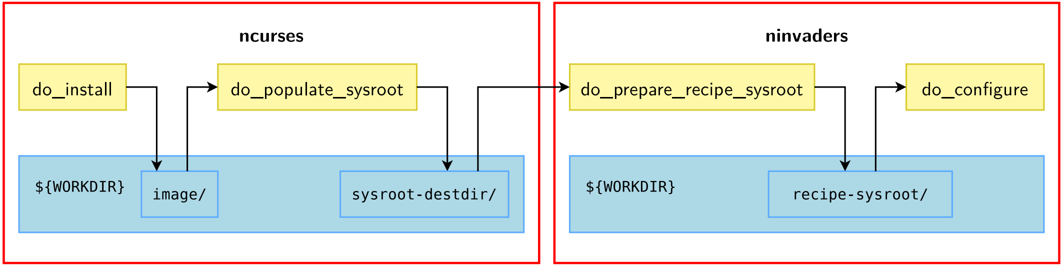
```
libpcap$ ./configure
...
checking for bluetooth/bluetooth.h... no
configure: Bluetooth sniffing is not supported; install bluez-lib devel to enable it
```

  - `libpcap` built *after* `bluez5`

```
libpcap$ ./configure
...
checking for bluetooth/bluetooth.h... yes
configure: Bluetooth sniffing is supported
```

Per-recipe sysroot

- Instead of a global sysroot, `bitbake` implements a *per-recipe sysroot*
- Main goal: reproducible build even with automatic dependencies



Per-recipe sysroot

- Before the actual build, each recipe prepares its own sysroot
  - Contains libraries and headers *only* for the recipes it `DEPENDS` on
  - Ensures the configuration stage will not detect libraries not explicitly listed in `DEPENDS` but already built for other reasons
  - `${WORKDIR}/recipe-sysroot` for target recipes
  - `${WORKDIR}/recipe-sysroot-native` for native recipes
- At the end of the build, each recipe produces its destination sysroot
  - Its own slice of sysroot, with the libraries and headers it directly provides
  - Used as input for other recipes to generate their recipe-sysroot
  - `${WORKDIR}/sysroot-destdir`

The complete sysroot

- A complete sysroot is available:
  - For each image
    - In `${WORKDIR}/recipe-sysroot` just like any recipe
  - In the SDK
    - Covered later

Using Python code in metadata

Tasks in Python

- Tasks can be written in Python when using the `python` keyword.
- Two modules are automatically imported:
  - `bb`: to access `bitbake`'s internal functions.
  - `os`: Python's operating system interfaces.
- You can import other modules using the `import` keyword.
- Anonymous Python functions are executed during parsing.
- Short Python code snippets can be written inline with the `${@<python code>}` syntax.

Accessing the datastore with Python

- The `d` variable is accessible within Python tasks.
- `d` represents the `bitbake` datastore (where variables are stored).

`d.getVar("X", expand=False)` Returns the value of `X`.

`d.setVar("X", "value")` Set `X`.

`d.appendVar("X", "value")` Append `value` to `X`.

`d.prependVar("X", "value")` Prepend `value` to `X`.

`d.expand(expression)` Expand variables in `expression`.

Python code examples

```
# Anonymous function (automatically called at parsing time)
python __anonymous() {
    if d.getVar("FOO", True) == "example":
        d.setVar("BAR", "Hello, World.")
}
# Task
python do_settime() {
    import time
    d.setVar("TIME", time.strftime('%Y%m%d', time.gmtime()))
}

# Inline Python code
do_install() {
    echo "Build OS: ${@os.uname()[0].lower()}"
}
```

Real life example of anonymous function:

https://github.com/linux4sam/meta-atmel/blob/scarthgap/recipes-kernel/linux/linux.inc

Variable flags

Variable flags

- *Variable flags*, or *varflags*, are used to store extra information on tasks and variables.
- They are used to control task functionalities.
- A typical example:

  `SRC_URI[md5sum] = "97b2c3fb082241ab5c56ab728522622b"`
- See the list of varflags supported by bitbake.
- More varflags can be added freely.

Variable flags examples

- dirs: directories that should be created before the task runs. The last one becomes the work directory for the task. Example: do_fetch in base.bbclass.

  ```
  do_compile[dirs] = "${B}"
  ```
- noexec: disable the execution of the task.

  ```
  do_settime[noexec] = "1"
  ```
- nostamp: do not create a *stamp* file when running the task. The task will always be executed.

  ```
  do_menuconfig[nostamp] = "1"
  ```
- doc: task documentation displayed by *listtasks*.

  ```
  do_settime[doc] = "Set the current time in ${TIME}"
  ```
- depends: add a dependency between specific tasks

  ```
  do_patch[depends] = "quilt-native:do_populate_sysroot"
  ```

Packages features

Benefits

- Features can be built depending on the needs.
- This allows to avoid compiling all features in a software component when only a few are required.
- A good example is ConnMan: Bluetooth support is built only if there is Bluetooth on the target.
- The `PACKAGECONFIG` variable is used to configure the build on a per feature granularity, for packages.

PACKAGECONFIG

- PACKAGECONFIG takes the list of features to enable.
- PACKAGECONFIG[<feature>] takes up to six arguments, separated by commas:
  1. Argument used by the configuration task if the feature is enabled (EXTRA_OECONF).
  2. Argument added to EXTRA_OECONF if the feature is disabled.
  3. Additional build dependency (DEPENDS), if enabled.
  4. Additional runtime dependency (RDEPENDS), if enabled.
  5. Additional runtime recommendations (RRECOMMENDS), if enabled.
  6. Any conflicting PACKAGECONFIG settings for this feature.
- Unused arguments can be omitted or left blank.

Example: from ConnMan

```
PACKAGECONFIG ??= "wifi openvpn"

PACKAGECONFIG[wifi] = "--enable-wifi,                    \
                       --disable-wifi,                   \
                       wpa-supplicant,                   \
                       wpa-supplicant"
PACKAGECONFIG[bluez] = "--enable-bluetooth,              \
                        --disable-bluetooth,             \
                        bluez5,                          \
                        bluez5"
PACKAGECONFIG[openvpn] = "--enable-openvpn,              \
                          --disable-openvpn,             \
                          ,                              \
                          openvpn"
```

Enabling PACKAGECONFIG features

- In a .bbappend of the recipe, just append to PACKAGECONFIG

```
PACKAGECONFIG += "<feature>"
PACKAGECONFIG += "tui"
```

- In a config file (e.g. distro conf)

```
PACKAGECONFIG:append:pn-<recipename> = " <feature>"
PACKAGECONFIG:append:pn-gdb = " tui"
```

Inspecting available PACKAGECONFIG flags

- ${POKY_DIR}/scripts/contrib/list-packageconfig-flags.py shows the PACKAGECONFIG varflags available for each recipe:

```
$ ../poky/scripts/contrib/list-packageconfig-flags.py
RECIPE NAME    PACKAGECONFIG FLAGS
==================================
alsa-plugins   aaf jack libav maemo-plugin maemo-resource-manager pulseaudio samplerate speexdsp
connman        3g bluez client iptables l2tp nfc nftables openvpn pptp systemd tist vpnc wifi ...
gdb            babeltrace debuginfod python readline tui xz
...
```

- The -a flag shows all the details:

```
$ ../poky/scripts/contrib/list-packageconfig-flags.py -a
connman-1.41
/home/murray/w/yocto-stm32-labs/poky/meta/recipes-connectivity/connman/connman_1.41.bb
PACKAGECONFIG wispr iptables client                    3g wifi                    bluez
PACKAGECONFIG[wifi] --enable-wifi, --disable-wifi, wpa-supplicant, wpa-supplicant
PACKAGECONFIG[bluez] --enable-bluetooth, --disable-bluetooth, bluez5, bluez5
PACKAGECONFIG[openvpn] --enable-openvpn --with-openvpn=${sbindir}/openvpn,--disable-openvpn,,openvpn
...
```

Conditional features

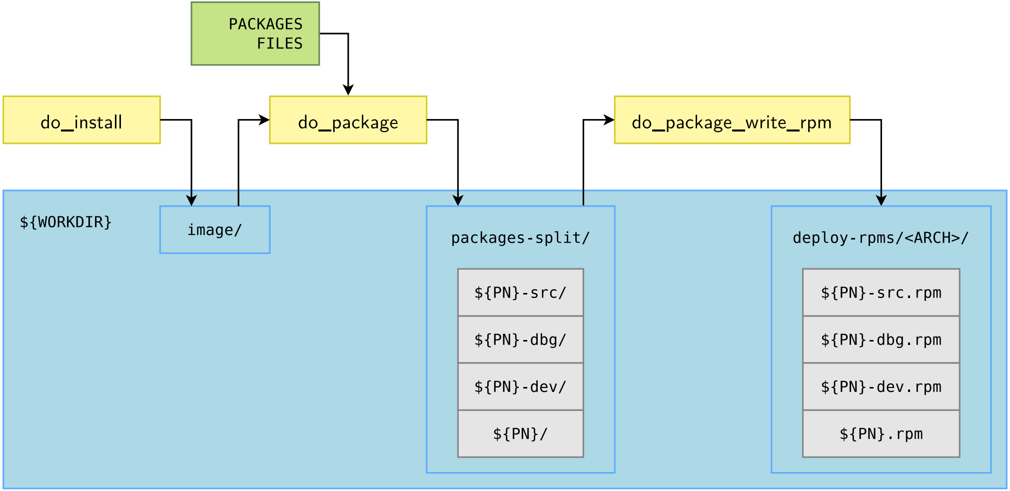
Conditional features

- Some values can be set dynamically, thanks to a set of functions:
- `bb.utils.contains(variable, checkval, trueval, falseval, d)`: if `checkval` is found in `variable`, `trueval` is returned; otherwise `falseval` is used. `d` is the BitBake datastore.
- `bb.utils.filter(variable, checkvalues, d)`: returns all the words in the variable that are present in the checkvalues.
- Example (`meta/recipes-connectivity/connman/connman.inc`):

```
PACKAGECONFIG ??= "wispr iptables client\
                   ${@bb.utils.filter('DISTRO_FEATURES', '3g systemd', d)} \
                   ${@bb.utils.contains('DISTRO_FEATURES', 'bluetooth', 'bluez', '', d)} \
                   ${@bb.utils.contains('DISTRO_FEATURES', 'wifi', 'wifi ${WIRELESS_DAEMON}', '', d)} \
"
```

Package splitting

Package splitting



Package splitting

- `do_install` copies *all* files in the `D` directory (`${WORKDIR}/image`).
- `do_package` splits files in several packages in `${WORKDIR}/packages-split`
  - based on the `PACKAGES` and `FILES` variables.
- `do_package_write_rpm` generates RPM packages

PACKAGES

- PACKAGES lists the packages to be built:

```
PACKAGES = "${PN}-src ${PN}-dbg ${PN}-staticdev ${PN}-dev \
    ${PN}-doc ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN}"
```

- More packages can be added to the default list
  - Useful when a single remote repository provides multiple binaries or libraries.
  - The order matters. PACKAGE_BEFORE_PN allows to pick files normally included in the default package in another.
- PACKAGES_DYNAMIC allows to check dependencies when optional packages are satisfied.
- ALLOW_EMPTY allows to produce a package even if it is empty.
- To prevent configuration files from being overwritten during the Package Management System update process, use CONFFILES.

- For each package a `FILES` variable lists the files to include.
- It must be package specific (e.g. with `:${PN}`, `:${PN}-dev`, dots).
- Defaults from `meta/conf/bitbake.conf`:

```
FILES:${PN}-dev = \
    "${includedir} ${FILES_SOLIBSDEV} ${libdir}/*.la \
     ${libdir}/*.o ${libdir}/pkgconfig ${datadir}/pkgconfig \
     ${datadir}/aclocal ${base_libdir}/*.o \
     ${libdir}/${BPN}/*.la ${base_libdir}/*.la \
     ${libdir}/cmake ${datadir}/cmake"
FILES:${PN}-dbg = \
    "/usr/lib/debug /usr/lib/debug-static \
     /usr/src/debug"
```

- The package named just ${PN} is the one that gets installed in the root filesystem.
- In Poky, defaults to:

```
FILES:${PN} = \
    "${bindir}/* ${sbindir}/* ${libexecdir}/* ${libdir}/lib*${SOLIBS} \
     ${sysconfdir} ${sharedstatedir} ${localstatedir} \
     ${base_bindir}/* ${base_sbindir}/* \
     ${base_libdir}/*${SOLIBS} \
     ${base_prefix}/lib/udev/rules.d ${prefix}/lib/udev/rules.d \
     ${datadir}/${BPN} ${libdir}/${BPN}/* \
     ${datadir}/pixmaps ${datadir}/applications \
     ${datadir}/idl ${datadir}/omf ${datadir}/sounds \
     ${libdir}/bonobo/servers"
```

Example

- The kexec tools provides kexec and kdump:

```
require kexec-tools.inc
export LDFLAGS = "-L${STAGING_LIBDIR}"
EXTRA_OECONF = " --with-zlib=yes"

SRC_URI[md5sum] = "b9f2a3ba0ba9c78625ee7a50532500d8"

PACKAGES =+ "kexec kdump"

FILES:kexec = "${sbindir}/kexec"
FILES:kdump = "${sbindir}/kdump"
```

Inspecting packages

`oe-pkgdata-util` is a tool that can help inspecting packages:

- Which package is shipping a file:

```
$ oe-pkgdata-util find-path /bin/busybox
busybox: /bin/busybox
```

- Which files are shipped by a package:

```
$ oe-pkgdata-util list-pkg-files busybox
busybox:
    /bin/busybox
    /bin/busybox.nosuid
    /bin/busybox.suid
    /bin/sh
```

- Which recipe is creating a package:

```
$ oe-pkgdata-util lookup-recipe kdump
kexec-tools
$ oe-pkgdata-util lookup-recipe libtinfo5
ncurses
```

Dependencies in detail

DEPENDS

- `DEPENDS` describes a build-time dependency
- Typical case: a program needs the library and headers files from a library to be configured and/or built
- In other words: it needs the library in its *sysroot*
- In `ninvaders.bb`, the line
  `DEPENDS = "ncurses"`
  creates a dependency
  - Of `ninvaders.do_prepare_recipe_sysroot`
  - On `ncurses.do_populate_sysroot`

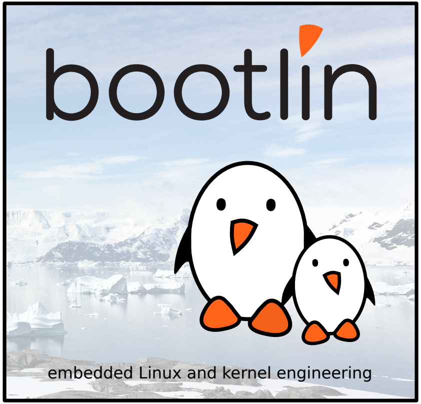
RDEPENDS

- RDEPENDS describes a runtime dependency
- Typical case: a program uses another program at runtime via sockets, DBUS, etc, or simply executes it
- It does not need it at build time
- In `inetutils_2.4.bb`, the line
  `RDEPENDS:${PN}-ftpd += "xinetd"`
  creates a dependency
  - Of `inetutils.do_build`
  - On `xinetd.do_package_write_rpm`
- And adds in the `inetutils-ftpd` RPM package a dependency on the `xinetd` RPM package

RRECOMMENDS

- RRECOMMENDS is similar to RDEPENDS
- But if the dependency package is not built it will just be skipped instead of failing the build
- Typical cases:
  - A package extends the features of a program, but its build has been disabled explicitly (e.g. via BAD_RECOMMENDATIONS)
  - Depending on a kernel module that might also be built-in in the kernel Image
- In `watchdog_5.16.bb`, the line
  `RRECOMMENDS:${PN} += "kernel-module-softdog"`
  does nothing if the `softdog` kernel module is not built by the kernel (could be builtin)

Licensing



Managing licenses

Tracking license changes

- The license of an external project may change at some point.
- The LIC_FILES_CHKSUM tracks changes in the license files.
- If the license's checksum changes, the build will fail.
  - The recipe needs to be updated.

```
LIC_FILES_CHKSUM = "                                   \
    file://COPYING;md5=...                             \
    file://src/file.c;beginline=3;endline=21;md5=..."
```

- LIC_FILES_CHKSUM is mandatory in every recipe, unless LICENSE is set to CLOSED.

Package exclusion

- We may not want some packages due to their licenses.
- To exclude a specific license, use INCOMPATIBLE_LICENSE
- To exclude all GPLv3 packages:

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

- License names are the ones used in the LICENSE variable.

Commercial licenses

- By default the build system does not include commercial components.
- Packages with a commercial component define:

```
LICENSE_FLAGS = "commercial"
```

- To build a package with a commercial component, the package must be in the LICENSE_FLAGS_ACCEPTED variable.
- Example, gst-plugins-ugly:

```
LICENSE_FLAGS_ACCEPTED = "commercial_gst-plugins-ugly"
```

Listing licenses

OpenEmbbedded will generate a manifest of all the licenses of the software present on the target image in `$BUILDDIR/tmp/deploy/licenses/<image>/license.manifest`

```
PACKAGE NAME: busybox
PACKAGE VERSION: 1.31.1
RECIPE NAME: busybox
LICENSE: GPL-2.0-only & bzip2-1.0.4

PACKAGE NAME: dropbear
PACKAGE VERSION: 2019.78
RECIPE NAME: dropbear
LICENSE: MIT & BSD-3-Clause & BSD-2-Clause & PD
```

You can also include the manifest and individual licenses in the root filesystem:

- Either use `COPY_LIC_DIRS = "1"` and `COPY_LIC_MANIFEST = "1"`
- Or use `LICENSE_CREATE_PACKAGE = "1"` to generate and install packages including the license files.

Providing sources

OpenEmbedded provides the `archiver` class to generate tarballs of the source code, to meet the requirements of some licenses:

- Use `INHERIT += "archiver"`
- Set the `ARCHIVER_MODE` variable, the default is to provide patched sources. To provide configured sources:

```
ARCHIVER_MODE[src] = "configured"
```

- The `archiver` class stores all source archives, even for software whose license does not require to provide the sources
  - The documentation provides a sample script to filter only GPL licenses

Generating a Software Bill of Materials (SBoM)

Instead of generating license information and source tarballs separately, OpenEmbedded can actually generate an SBoM, describing:

- **Sources** for target and host components
- **Licenses** of such components
- **Dependencies** between such components
- **Applied changes**, in particular fixes for **known vulnerabilities**.

This SBoM is generated in the standard SPDX format, which you can feed to tools supporting SPDX.

Usefulness of SPDX SBoM

SPDX SBoM can be attached to a software delivery, and used for:

- License compliance assessment
- Vulnerability assessment. You can use the SBoM to check whether your software supply chain is impacted by currently known vulnerabilities, both in host and target packages.

The US government is pushing for having such information in all software it procures and will probably make it mandatory soon.

How to create SPDX SBoM with OpenEmbedded

- Add `INHERIT += "create-spdx"` to your configuration file
- JSON SPDX files will be generated in `tmp/deploy/images/MACHINE/`
- You can then set optional variables:
  - `SPDX_PRETTY`: Make generated files more human readable (newlines, indentation)
  - `SPDX_ARCHIVE_PACKAGED`: Add compressed archives of the files in generated target packages.
  - `SPDX_INCLUDE_SOURCES`: Add descriptions of the source files for host tools and target packages.
  - `SPDX_ARCHIVE_SOURCES`: Add archives of these source files themselves (when `SPDX_INCLUDE_SOURCES` is set).

Example IMAGE-MACHINE.spdx.json output


```
{
  "SPDXID": "SPDXRef-DOCUMENT",
  "creationInfo": {
       "comment": "This document was created by analyzing the source of the Yocto recipe during the build.",
       "created": "2022-10-25T12:32:13Z",
       "creators": [
       "Tool: OpenEmbedded Core create-spdx.bbclass",
       "Organization: OpenEmbedded ()",
       "Person: N/A ()"
       ],
       "licenseListVersion": "3.14"
  },
  "dataLicense": "CC0-1.0",
  "documentNamespace": "http://spdx.org/spdxdoc/core-image-minimal-qemux86-64-20221025122556-f686f4f3-...",
  "externalDocumentRefs": [
       {
       "checksum": {
       "algorithm": "SHA1",
       "checksumValue": "f6de08ea7fa026f480fd80cf7862a5c99c4d7a2b"
       },
       "externalDocumentId": "DocumentRef-base-files",
       "spdxDocument": "http://spdx.org/spdxdoc/base-files-ee9424e3-1d7e-5739-b9cd-237a1a6f843f"
       },
...
```

SPDX output details

```
...
"externalDocumentRefs": [
 {
   ...
   "spdxDocument":
"http://spdx.org/spdxdoc/base-files-ee9
424e3-1d7e-5739-b9cd-237a1a6f843f"
 },
 {
   ...
   "spdxDocument":
"http://spdx.org/spdxdoc/runtime-base-
files-2c77c531-02ab-5600-a436-27b79
9b75d94"
 },
 ...
```

```
{
 "documents": [
   {
     "documentNamespace":
"http://spdx.org/spdxdoc/base-files-ee9424e3-1
d7e-5739-b9cd-237a1a6f843f",
     "filename": "base-files.spdx.json",
     ...
   },
   {
     "documentNamespace":
"http://spdx.org/spdxdoc/runtime-base-files-2c
77c531-02ab-5600-a436-27b799b75d94",
     "filename": "runtime-base-files.spdx.json",
     ...
   }
  ...
```

```
-rw-r--r-- root/root  11466 2011-04-06 01:00
base-files.spdx.json
-rw-r--r-- root/root  1077 2011-04-06 01:00
runtime-base-files.spdx.json
...
```

Further resources about SPDX SBoM

- Yocto project documentation:
  https://docs.yoctoproject.org/dev/dev-manual/sbom.html
- Joshua Watt: Automated SBoM generation with OpenEmbedded and the Yocto Project (FOSDEM 2023)
  https://youtu.be/Q5UQUM6zxVU
- SPDX project homepage:
  https://spdx.dev

The Yocto Project SDK

Corrections, suggestions, contributions and translations are welcome!

Overview

- An SDK (Software Development Kit) is a set of tools allowing the development of applications for a given target (operating system, platform, environment, …).
- It generally provides a set of tools including:
  - Compilers or cross-compilers.
  - Linkers.
  - Library headers.
  - Debuggers.
  - Custom utilities.

The Yocto Project SDK

- The Poky reference system is used to generate images, by building many applications and doing a lot of configuration work.
  - When developing an application, we only care about the application itself.
  - We want to be able to develop, test and debug easily.
- The Yocto Project SDK is an application development SDK, which can be generated to provide a full environment compatible with the target.
- It includes a toolchain, libraries headers and all the needed tools.
- This SDK can be installed on any computer and is self-contained. The presence of Poky is not required for the SDK to fully work.

Available SDKs

- Two different SDKs can be generated:
  - A generic SDK, including:
    - A toolchain.
    - Common tools.
    - A collection of basic libraries.
  - An image-based SDK, including:
    - The generic SDK.
    - The sysroot matching the target root filesystem.
- The toolchain in the SDKs is self-contained (linked to an SDK embedded libc).
- The SDKs generated with Poky are distributed in the form of a shell script.
- Executing this script extracts the tools and sets up the environment.

The generic SDK

- Mainly used for low-level development, where only the toolchain is needed:
  - Bootloader development.
  - Kernel development.
- The recipe `meta-toolchain` generates this SDK:
  - `bitbake meta-toolchain`
- The generated script, containing all the tools for this SDK, is in:
  - `$BUILDDIR/tmp/deploy/sdk`
  - Example:
    `poky-glibc-x86_64-meta-toolchain-cortexa8hf-neon-toolchain-5.0.sh`
- The SDK will be configured to be compatible with the specified `MACHINE`.

The image-based SDK

- Used to develop applications running on the target.
- One task is dedicated to the process. The task behavior can vary between the images.
  - `populate_sdk`
- To generate an SDK for `core-image-minimal`:
  - `bitbake -c populate_sdk core-image-minimal`
- The generated script, containing all the tools for this SDK, is in:
  - `$BUILDDIR/tmp/deploy/sdk`
  - Example:
    `poky-glibc-x86_64-core-image-minimal-cortexa8hf-neon-toolchain-5.0.sh`
- The SDK will be configured to be compatible with the specified `MACHINE`.

Adding packages to the SDK

- Two variables control what will be installed in the SDK:
  - TOOLCHAIN_TARGET_TASK: List of target packages to be included in the SDK
  - TOOLCHAIN_HOST_TASK: List of host packages to be included in the SDK
- Both can be appended to install more tools or libraries useful for development.
- Example: to have native curl on the SDK:

```
TOOLCHAIN_HOST_TASK:append = " nativesdk-curl"
```

SDK format

- Both SDKs are distributed as shell scripts.
- These scripts self extract themselves to install the toolchains and the files they provide.
- To install an SDK, retrieve the generated script and execute it.
  - The script asks where to install the SDK. Defaults to `/opt/poky/<version>`
  - Example: `/opt/poky/5.0`

```
$ ./poky-glibc-x86_64-meta-toolchain-cortexa8hf-neon-toolchain-5.0.sh
Poky (Yocto Project Reference Distro) SDK installer version 5.0
===============================================================
Enter target directory for SDK (default: /opt/poky/5.0):
You are about to install the SDK to "/opt/poky/5.0". Proceed[Y/n]?
Extracting SDK................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source
the environment setup script e.g.
 $ . /opt/poky/5.0/environment-setup-cortexa8hf-neon-poky-linux-gnueabi
```

Use the SDK

- To use the SDK, a script is available to set up the environment:

```
$ cd /opt/poky/5.0
$ source ./environment-setup-cortexa8hf-neon-poky-linux-gnueabi
```

- The PATH is updated to take into account the binaries installed alongside the SDK.
- Environment variables are exported to help using the tools.

SDK installation

environment-setup-cortexa8hf-neon-poky-linux-gnueabi
: Exports environment variables.

site-config-cortexa8hf-neon-poky-linux-gnueabi
: Variables used during the toolchain creation

sysroots
: SDK binaries, headers and libraries. Contains one directory for the host and one for the target.

version-cortexa8hf-neon-poky-linux-gnueabi
: Version information.

SDK environment variables

- CC: Full path to the C compiler binary.
- CFLAGS: C flags, used by the C compiler.
- CXX: C++ compiler.
- CXXFLAGS: C++ flags, used by CPP
- LD: Linker.
- LDFLAGS: Link flags, used by the linker.
- ARCH: For kernel compilation.
- CROSS_COMPILE: For kernel compilation.
- GDB: SDK GNU Debugger.
- OBJDUMP: SDK objdump.

To see the full list, open the environment script.

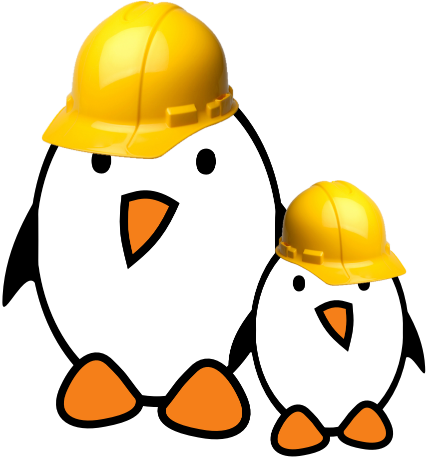
Examples

- To build an application for the target:

```
$ $CC -o example example.c
```

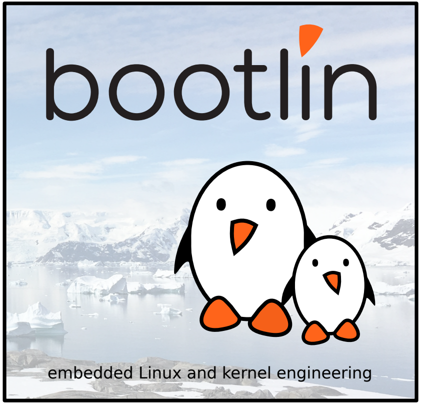
- The LDFLAGS variable is set to be used with the C compiler (gcc).

Practical lab - Create and use a Poky SDK



- Generate an SDK
- Compile an application for the target in the SDK

Devtool



Overview

- `Devtool` is a set of utilities to ease the integration and the development of OpenEmbedded recipes.
- It can be used to:
  - Generate a recipe for a given upstream application.
  - Modify an existing recipe and its associated sources.
  - Upgrade an existing recipe to use a newer upstream application.
- `Devtool` adds a new layer, automatically managed, in `$BUILDDIR/workspace/`.
- It then adds or appends recipes to this layer so that the recipes point to a local path for their sources. In `$BUILDDIR/workspace/sources/`.
  - Local sources are managed by `git`.
  - All modifications made locally should be commited.

There are three ways of creating a new `devtool` project:

- To create a new recipe: `devtool add <recipe> <fetchuri>`
  - Where `recipe` is the recipe's name.
  - `fetchuri` can be a local path or a remote *uri*.
- To modify the source for an existing recipe: `devtool modify <recipe>`
- To upgrade a given recipe: `devtool upgrade -V <version> <recipe>`
  - Where `version` is the new version of the upstream application.

`devtool` usage 2/3

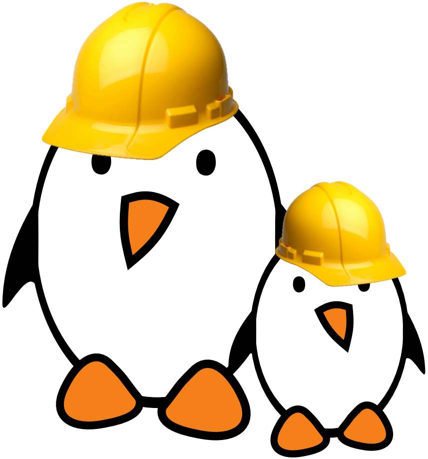
Once a `devtool` project is started, commands can be issued:

- `devtool edit-recipe <recipe>`: edit `recipe` in a text editor (as defined by the `EDITOR` environment variable).
- `devtool build <recipe>`: build the given `recipe`.
- `devtool build-image <image>`: build `image` with the additional `devtool` recipes' packages.

devtool usage 3/3

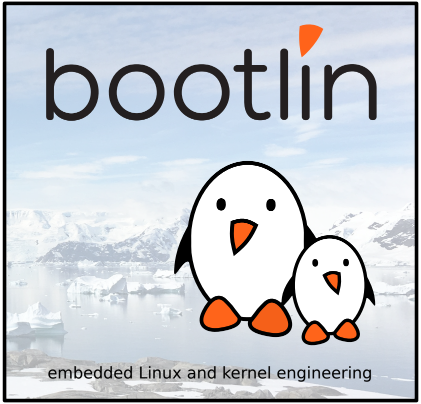
- `devtool deploy-target <recipe> <target>`: upload the `recipe`'s packages on `target`, which is a live running target with an SSH server running (`user@address`).
- `devtool update-recipe <recipe>`: generate patches from git commits made locally.
- `devtool reset <recipe>`: remove `recipe` from the control of `devtool`. Standard layers and remote sources are used again as usual.

Practical lab - Using devtool



- Generate a new recipe
- Modify a recipe to add a new patch
- Upgrade a recipe to a newer version

Automating layer management



Release management

There are multiple tasks that OE/bitbake based projects let you do on your own to ensure build reproducibility:

- Code distribution and project setup.
- Release tagging

A separate tool is needed for that, usual solutions are:

- git submodules + setup script. Great example in YOE: https://github.com/YoeDistro/yoe-distro
- repo and `templateconf` or setup script
- kas

Google repo

- A good way to distribute a distribution (Poky, custom layers, BSP, `.templateconf`...) is to use Google's `repo`.
- `Repo` is used in Android to distribute its source code, which is split into many `git` repositories. It's a wrapper to handle several `git` repositories at once.
- The only requirement is to use `git`.
- The `repo` configuration is stored in a `manifest` file, usually available in its own `git` repository.
- It could also be in a specific branch of your custom layer.
- It only handles fetching code, handling `local.conf` and `bblayers.conf` is done separately

Manifest example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote name="yocto-project" fetch="git.yoctoproject.org" />
  <remote name="private" fetch="git.example.net" />

  <default revision="scarthgap" remote="private" />

  <project name="poky" remote="yocto-project" />
  <project name="meta-ti" remote="yocto-project" />
  <project name="meta-custom" />
  <project name="meta-custom-bsp" />
  <project path="meta-custom-distro" name="distro">
    <copyfile src="templateconf" dest="poky/.templateconf" />
  </project>
</manifest>
```

Retrieve the project using repo

```
$ mkdir my-project; cd my-project
$ repo init -u https://git.example.net/manifest.git
$ repo sync -j4
```

- `repo init` uses the `default.xml` manifest in the repository, unless specified otherwise.
- You can see the full `repo` documentation at https://source.android.com/source/using-repo.html.

repo: release

To tag a release, a few steps have to be taken:

- Optionally tag the custom layers
- For each project entry in the manifest, set the revision parameter to either a tag or a commit hash.
- Commit and tag this version of the manifest.

kas

- Specific tool developed by Siemens for OpenEmbedded: https://github.com/siemens/kas
- Will fetch layers and build the image in a single command
- Uses a single JSON or YAML configuration file part of the custom layer
- Can generate and run inside a Docker container
- Can setup `local.conf` and `bblayers.conf`

kas configuration

```yaml
header:
  version: 8
machine: mymachine
distro: mydistro
target:
  - myimage

repos:
  meta-custom:

  bitbake:
    url: "https://git.openembedded.org/bitbake"
    # tag 2.0
    commit: c212b0f3b542efa19f15782421196b7f4b64b0b9
    layers:
      .: excluded

  openembedded-core:
    url: "https://git.openembedded.org/openembedded-core"
    branch: scarthgap
    layers:
      meta:
```

kas configuration

```yaml
meta-freescale:
  url: "https://github.com/Freescale/meta-freescale"
  branch: scarthgap

meta-openembedded:
 url: https://git.openembedded.org/meta-openembedded
 branch: scarthgap
 layers:
   meta-oe:
   meta-python:
   meta-networking:
```

- Then a single command will build all the listed targets for the machine:
  `$ kas build meta-custom/mymachine.yaml`
- Or, alternatively, invoke `bitbake` commands:
  `$ kas shell /path/to/kas-project.yml -c 'bitbake dosfsutils-native'`

Runtime Package Management

Corrections, suggestions, contributions and translations are welcome!

Introduction

- BitBake always builds packages selected in `IMAGE_INSTALL`.
- The packages are used to generate the root filesystem.
- It is also possible to update the system at runtime using these packages, for many use cases:
  - In-field security updates.
  - System updates over the wire.
  - System, packages or configuration customization at runtime.
  - Remote debugging.
- Using the Runtime Package Management is an optional feature.
- We'll use the IPK package format as an example in the following slides.

Requirements

- First of all, you need a server to serve the packages to a private subnet or over the Internet. Packages are typically served over `https` or `http`.
- Specific tools are also required on the target, and must be shipped on the product. They should be included into the images generated by the build system.
- These tools will be specific to the package type used.
  - This is similar to Linux distributions: Debian is using `.deb` related tools (dpkg, apt...) while Fedora uses `.rpm` related ones (rpm, dnf).

Build configuration

Build configuration 1/2

- The PACKAGE_CLASSES variable controls which package format to use. More than one can be used.
- Valid values are package_rpm, package_deb, package_ipk.
- By default Poky uses the RPM format, while OpenEmbedded-Core uses the IPK one.
- Example:
  - PACKAGE_CLASSES = "package_ipk"
  - PACKAGE_CLASSES = "package_rpm package_deb"

Build configuration 2/2

To install the required tools on the target, there are two possible solutions:

- By adding `package-management` to the images features.
  - The required tool will be installed on the target.
  - The package database corresponding to the build will be installed as well.
- Or by manually adding the required tools in `IMAGE_INSTALL`. For example, to use the IPK format we need `opkg`.

Build considerations

- The Runtime Package Management uses package databases to store information about available packages and their version.
- Whenever a build generates a new package or modifies an existing one, the package database must be updated.
- `$ bitbake package-index`
- Be careful: BitBake does not properly schedule the `package-index` target. You must use this target alone to have a consistent package database.
    - `$ bitbake ninvaders package-index` won't necessarily generate an updated package database.

Package server configuration

Apache2 example setup

Apache2 HTTP setup for IPK packages. This should go in `/etc/apache2/sites-enabled/package-server.conf`.

```
<VirtualHost *:80>
    ServerName packages.example.net

    DocumentRoot /path/to/build/tmp/deploy/ipk
    <Directory /path/to/build/tmp/deploy/ipk>
        Options +Indexes
        Options Indexes FollowSymLinks
        Order allow,deny
        allow from all
        AllowOverride None
        Require all granted
    </Directory>
</VirtualHost>
```

Target configuration

The IPK runtime management software

- The IPK runtime management software is opkg.
- It can be configured using configurations files ending in `.conf` in `/etc/opkg/`.
- This configuration helps opkg to find the package databases you want to use.
- For example, with our previously configured package server:

```
src/gz all http://packages.example.net/all
src/gz armv7a http://packages.example.net/armv7a
src/gz beaglebone http://packages.example.net/beaglebone
```

- This can be automatically generated by defining the `PACKAGE_FEED_URIS`, `PACKAGE_FEED_BASE_PATHS` and `PACKAGE_FEED_ARCHS` variables

opkg usage

- `opkg update`: fetch and update the package databases, from the remote package servers.
- `opkg list`: list available packages.
- `opkg upgrade`: upgrade all installed packages.
- `opkg upgrade <package>`: upgrade one package explicitly.
- `opkg install <package>`: install a specific package.

opkg upgrade over an unstable network

- To avoid upgrade issues when downloading packages from a remote package server using an unstable connection, you can first download the packages and then proceed with the upgrade.
- To do this we must use a cache, which can be defined in the opkg configuration with: `option cache /tmp/opkg-cache`.

```
# opkg update
# opkg --download-only upgrade
# opkg upgrade
```

Yocto Project Resources

Yocto Project documentation

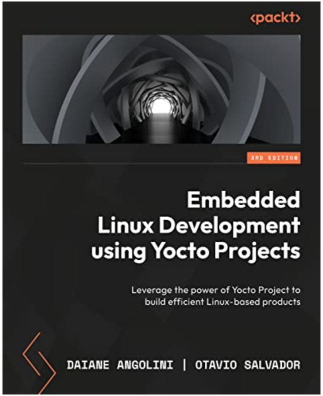
- https://docs.yoctoproject.org/
- Wiki: https://wiki.yoctoproject.org/wiki/Main_Page
- https://layers.openembedded.org/

Contributing

- Friendly community to newcomers of the project
- Mailing list based upstreaming workflow:
  https://docs.yoctoproject.org/contributor-guide
- Ask questions and help people on IRC:
  https://web.libera.chat/?channels=#yocto
- Try to solve newcomer bugs reported on the official Bugzilla:
  https://wiki.yoctoproject.org/wiki/Newcomers#Newcomer_Bugs
- The Wiki contains a lot of useful information:
  https://wiki.yoctoproject.org/wiki/Main_Page

Embedded Linux Development Using Yocto Project - Third Edition, April 2023

- https://www.packtpub.com/product/embedded-linux-development-using-yocto-project-third-edition
- By Daiane Angolini and Otavio Salvador
- From basic to advanced usage, helps writing better, more flexible recipes. A good reference to jumpstart your Yocto Project development.

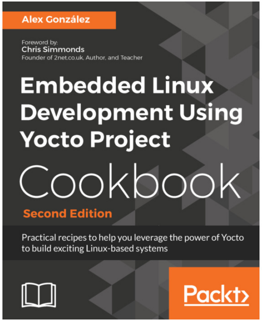


Useful Reading (2)

Embedded Linux Projects Using Yocto Project Cookbook - Second Edition, January 2018

- https://www.packtpub.com/virtualization-and-cloud/embedded-linux-development-using-yocto-project-cookbook-second-edition
- By Alex González
- A set of recipes that you can refer to and solve your immediate problems instead of reading it from cover to cover.

See our review: https://bit.ly/1GgVmCB



Last slides

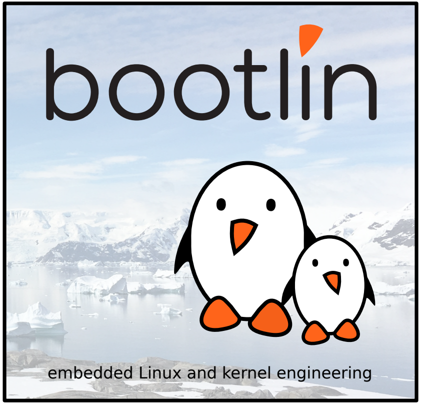
Thank you!

And may the Source be with you

Rights to copy

Document sources: https://github.com/bootlin/training-materials/

Extra slides



Quilt

Overview

- Quilt is a utility to manage patches which can be used without having a clean source tree.
- It can be used to create patches for recipes already available in the build system.
- Be careful when using this workflow: the modifications won't persist across builds!

Using Quilt

1. Find the recipe working directory in `$BUILDDIR/tmp/work/`.
2. Create a new `Quilt` patch: `$ quilt new topic.patch`
3. Add files to this patch: `$ quilt add file0.c file1.c`
4. Make the modifications by editing the files.
5. Test the modifications: `$ bitbake -c compile -f recipe`
6. Generate the patch file: `$ quilt refresh`
7. Move the generated patch into the recipe's directory.