Yocto Project and OpenEmbedded Training

Based on the material from http://bootlin.com
Latest update: November 3, 2020.

Corrections, suggestions, contributions and translations are welcome!
Send them to nikola.jelic@zuehlke.com

Rights to copy

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

Hyperlinks in the document

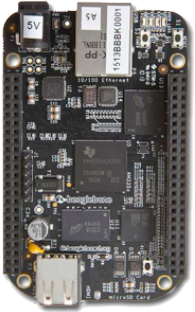
There are many hyperlinks in the document

- Regular hyperlinks:
  `https://kernel.org/`
- Kernel documentation links:
  `dev-tools/kasan`
- Links to kernel source files and directories:
  `drivers/input/`
  `include/linux/fb.h`
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  `platform_get_irq()`
  `GFP_KERNEL`
  `struct file_operations`

Generic course information

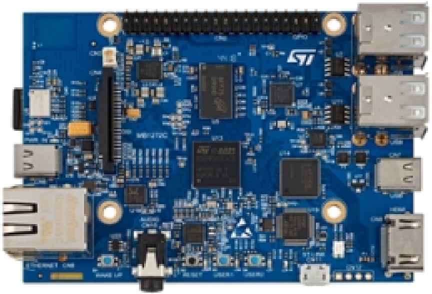
Two supported hardware platforms

There are two variants of the practical labs for this course, one supporting the Beaglebone Black board, the other supporting the STMicroelectronics STM32MP157A-DK1 Discovery board.



Beaglebone Black

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

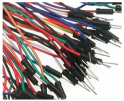


STM32MP157A-DK1 Discovery

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

Shopping list: hardware for this course

- Main board, choice 1:
  - STMicroelectronics STM32MP157A-DK1 Discovery kit (Mouser: 65 EUR + VAT)
  - USB-C cable (power supply)
  - USB-A to micro B cable (serial console)
- Main board, choice 2:
  - Beaglebone Black or Beaglebone Black Wireless (Mouser: 73 EUR + VAT), USB-A to micro B power cable included
  - USB Serial Cable - 3.3 V - female ends: Olimex: https://frama.link/zWJDToXP
- Nintendo Nunchuk with UEXT connector: Olimex: https://j.mp/1dTYLfs
- Breadboard jumper wires - Male ends: Olimex: https://bit.ly/2pSiIPs
- SD card with 8 GB capacity

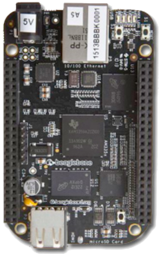


Supported hardware

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.

beagleboard.org



open source hardware

Supported hardware

STMicroelectronics STM32MP157A-DK1 Discovery board

- STM32MP157A (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
- Currently sold at 65 EUR + VAT at Mouser

Board and CPU documentation, design files, software:
https://www.st.com/en/evaluation-tools/stm32mp157a-dk1.html

Course outline - Day 1

First dive into the Yocto Project. Introduction to recipes.

- Overview of an embedded Linux system architecture.
- Organization of the Yocto Project source tree.
- Building an image.
- Adding packages to the generated image.
- Recipes syntax. Writing a recipe.

Labs: first Yocto Project build, advanced Yocto configuration, add a custom application.

Course outline - Day 2

Recipes and layers details: write, use, customize. Machine configuration.

- Advanced recipes concepts.
- Extending existing recipes.
- The Yocto Project layers. Adding a new layer.
- Writing your own machine configuration.

Labs: create a Yocto layer, create a custom machine configuration.

Course outline - Day 3

Images and packagegroups details, advanced concepts, SDK and runtime package management.

- Adding a custom image.
- Further recipes concepts.
- Rootfs generation.
- Using the Yocto Project SDK.
- Runtime package management.

Labs: create a custom image, develop your application in the Poky SDK.

Participate!

During the lectures...

- Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- This helps the trainer to detect any explanation that wasn't clear or detailed enough.
- Don't hesitate to share your experience, for example to compare Linux with other operating systems used in your company.
- Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- Your participation can make our session more interactive and make the topics easier to learn.

Practical lab guidelines

During practical labs...

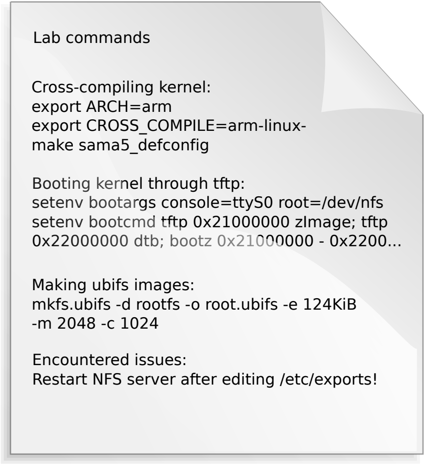
- We cannot support more than 8 workstations at once (each with its board and equipment). Having more would make the whole class progress slower, compromising the coverage of the whole training agenda (exception for public sessions: up to 10 people).
- So, if you are more than 8 participants, please form up to 8 working groups.
- Open the electronic copy of your lecture materials, and use it throughout the practical labs to find the slides you need again.
- Don't hesitate to copy and paste commands from the PDF slides and labs.

Advise: write down your commands!

During practical labs, write down all your commands in a text file.

- You can save a lot of time re-using commands in later labs.
- This helps to replay your work if you make significant mistakes.
- You build a reference to remember commands in the long run.
- That's particular useful to keep kernel command line settings that you used earlier.
- Also useful to get help from the instructor, showing the commands that you run.

```
gedit ~/lab-history.txt
```



Cooperate!

As in the Free Software and Open Source community, cooperation during practical labs is valuable in this training session:

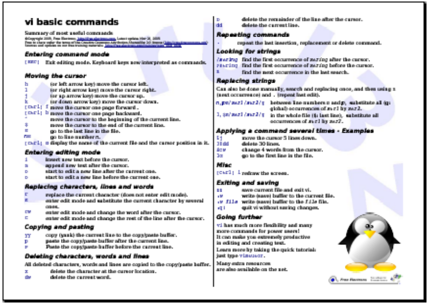
- If you complete your labs before other people, don't hesitate to help other people and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- Explain what you understood to other participants when needed. It also helps to consolidate your knowledge.
- Don't hesitate to report potential bugs to your instructor.
- Don't hesitate to look for solutions on the Internet as well.

Command memento sheet

- This memento sheet gives command examples for the most typical needs (looking for files, extracting a tar archive...)
- It saves us 1 day of UNIX / Linux command line training.
- Our best tip: in the command line shell, always hit the Tab key to complete command names and file paths. This avoids 95% of typing mistakes.
- Get an electronic copy on https://bootlin.com/doc/legacy/command-line/command_memento.pdf

vi basic commands

- The vi editor is very useful to make quick changes to files in an embedded target.
- Though not very user friendly at first, vi is very powerful and its main 15 commands are easy to learn and are sufficient for 99% of everyone's needs!
- Get an electronic copy on https://bootlin.com/doc/legacy/command-line/vi_memento.pdf
- You can also take the quick tutorial by running vimtutor. This is a worthy investment!


vi basic commands

Entering command mode

Moving the cursor

Entering editing mode

Replacing characters, lines and words

Copying and pasting

Deleting characters, words and lines

Repeating commands

Looking for strings

Replacing strings

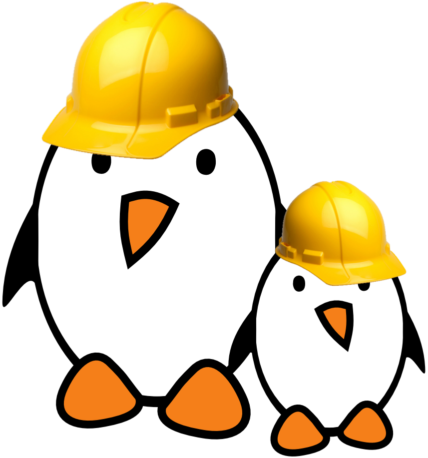
Applying a command several times - Examples

Misc

Exiting and saving

Going further

Practical lab - Training Setup

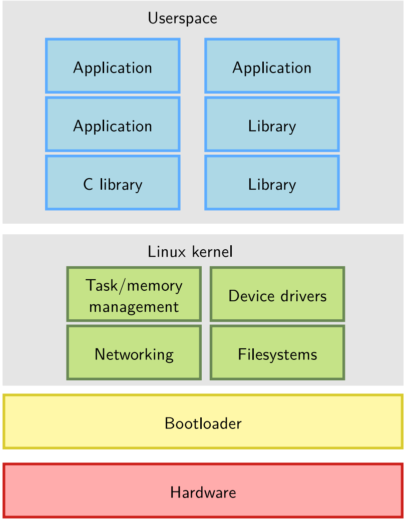


Prepare your lab environment

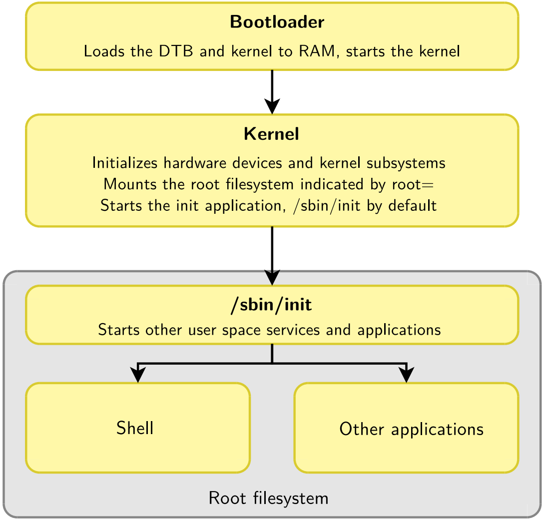
- Download and extract the lab archive

Introduction to Embedded Linux

Simplified Linux system architecture



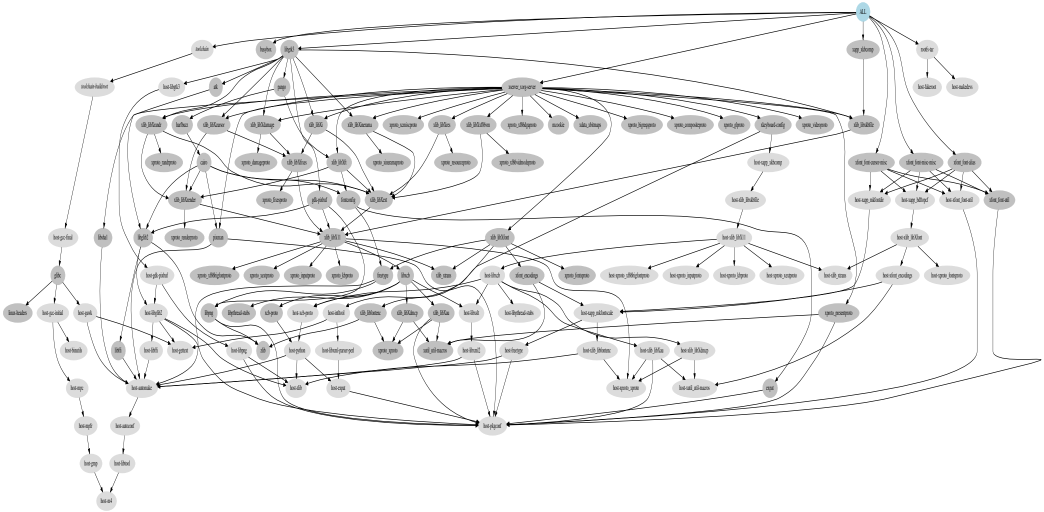
Overall Linux boot sequence



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.

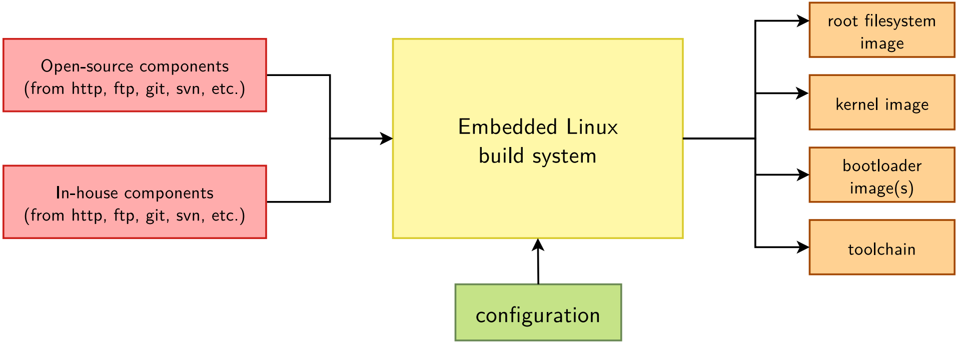
Complexity of user space integration



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

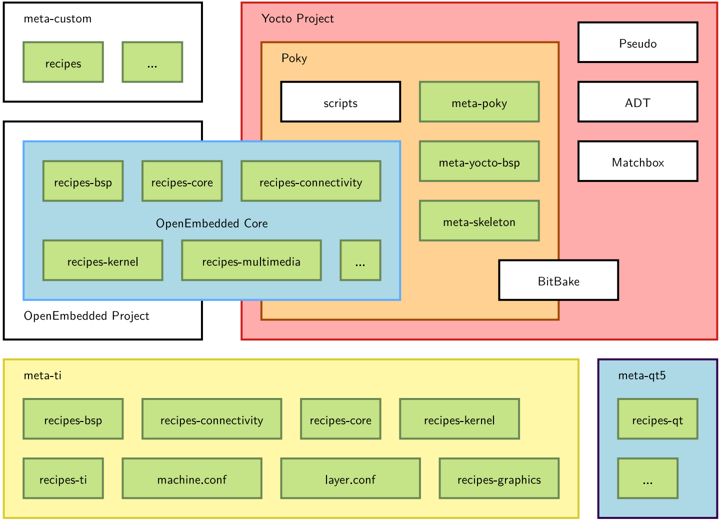
About

- The Yocto Project is a set of templates, tools and methods that allow to build custom embedded Linux-based systems.
- It is an open source project initiated by the Linux Foundation in 2010 and is still managed by one of its fellows: Richard Purdie.

The Yocto Project lexicon

- The core components of the Yocto Project are:
  - BitBake, the *build engine*. It is a task scheduler, like `make`. It interprets configuration files and recipes (also called *metadata*) to perform a set of tasks, to download, configure and build specified applications and filesystem images.
  - OpenEmbedded-Core, a set of base *layers*. It is a set of recipes, layers and classes which are shared between all OpenEmbedded based systems.
  - Poky, the *reference system*. It is a collection of projects and tools, used to bootstrap a new distribution based on the Yocto Project.

The Yocto Project lexicon



The Yocto Project lexicon

- Organization of OpenEmbedded-Core:
  - *Recipes* describe how to fetch, configure, compile and package applications and images. They have a specific syntax.
  - *Layers* are sets of recipes, matching a common purpose. For Texas Instruments board support, the *meta-ti* layer is used.
  - Multiple layers are used within a same distribution, depending on the requirements.
  - It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures.
  - It supports QEMU emulated machines for these architectures.

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.
- The main required element is **Poky**, the reference system which includes OpenEmbedded-Core. Other available tools are optional, but may be useful in some cases.

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* layer, a set of Texas Instruments specific recipes.
  - The *meta-arm* layers, a set of ARM architecture specific layers.
- All modifications are made in the *meta-ti* layer. Editing Poky is a **no-go**!
- We will set up this environment in the lab.

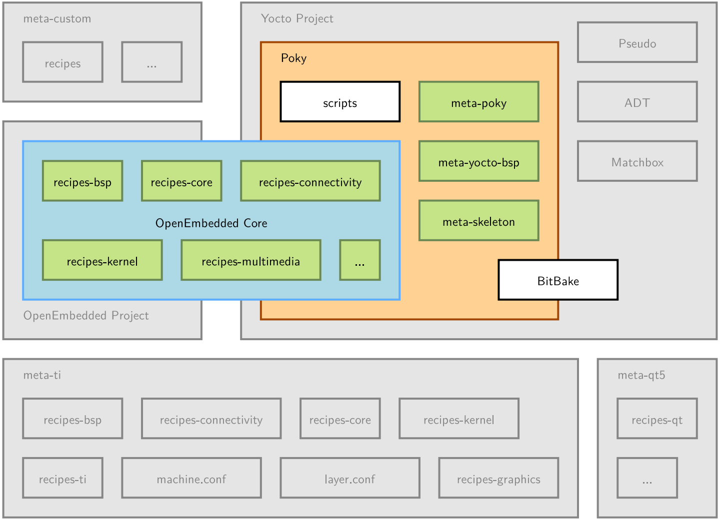
The Poky reference system overview

Download the Poky reference system

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

```
git clone -b dunfell git://git.yoctoproject.org/poky.git
```

Poky


meta-custom
recipes
...
Yocto Project
Poky
scripts
meta-poky
meta-yocto-bsp
meta-skeleton
Pseudo
ADT
Matchbox
recipes-bsp
recipes-core
recipes-connectivity
OpenEmbedded Core
recipes-kernel
recipes-multimedia
...
BitBake
OpenEmbedded Project
meta-ti
recipes-bsp
recipes-connectivity
recipes-core
recipes-kernel
recipes-ti
machine.conf
layer.conf
recipes-graphics
meta-qt5
recipes-qt
...

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. It takes an optional parameter which is the build directory name. By default, this is `build`. This script has to be sourced because it changes environment variables.

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://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html
- Variables in particular are described in the variable glossary: https://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html#ref-variables-glossary

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`.

Common targets

- Common targets are listed when sourcing the script:

  core-image-minimal A small image to boot a device and have access to core command line commands and services.

  core-image-sato Image with Sato support. Sato is a GNOME mobile-based user interface.

  meta-toolchain Generates the cross-toolchain in an installable format.

  meta-ide-support Generates the cross-toolchain and additional tools (gdb, qemu, ...) for IDE integration.

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.

The build/ directory 1/2

conf/	Configuration files. Image specific and layer configuration.
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 2/2

tmp/buildstats/ Build statistics for all packages built (CPU usage, elapsed time, host, timestamps…).

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/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.

Configuring the build system

The build/conf/ directory

- The conf/ directory in the build one holds build specific configuration.

  bblayers.conf Explicitly list the available layers.

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

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 CPUs on the system.

  PARALLEL_MAKE How many processes should be used when compiling. Defaults to the number of CPUs on the system.

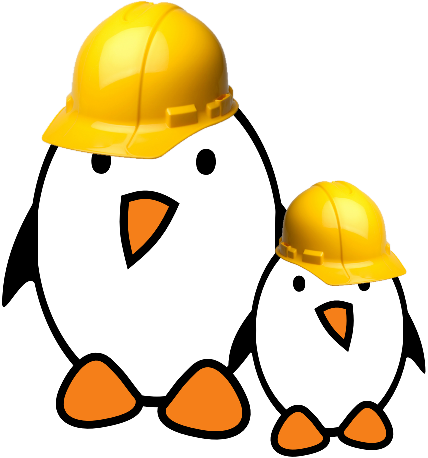
  MACHINE The machine the target is built for, e.g. beaglebone.

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.

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

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

A little reminder

- *Recipes* describe how to fetch, configure, compile and install applications.
- These tasks can be run independently (if their dependencies are met).
- All available packages in Poky are not selected by default in the images.
- Some packages may provide the same functionality, e.g. OpenSSH and Dropbear.

Advanced configuration

Overview

- The OpenEmbedded build system uses configuration variables to hold information.
- Configuration settings are in upper-case by convention, e.g. `CONF_VERSION`
- To make configuration easier, it is possible to prepend, append or define these variables in a conditional way.
- All variables can be overridden or modified in `$BUILDDIR/conf/local.conf`

Methods and conditions 1/4

- Append the keyword `_append` to a configuration variable to add values **after** the ones previously defined (without space).
  - `IMAGE_INSTALL_append = " dropbear"` adds `dropbear` to the packages installed on the image.
- Append the keyword `_prepend` to add values **before** the ones previously defined (without space).
  - `FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"` adds the folder to the set of paths where files are located (in a recipe).

Methods and conditions 2/4

- Append the keyword `_remove` to a configuration variable to remove all occurrences of a value within a configuration variable.
  - `IMAGE_INSTALL_remove = "i2c-tools"`
- Append the machine name to only define a configuration variable for a given machine. It tries to match with values from `MACHINEOVERRIDES` which include `MACHINE` and `SOC_FAMILY`.
  - `KERNEL_DEVICETREE_beaglebone = "am335x-bone.dtb"` tells to use the kernel device tree `am335x-bone.dtb` only when the machine is `beaglebone`.

- 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.

Methods and conditions 4/4

- The most specific variable 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"`

Operators 1/2

- Various operators can be used to assign values to configuration variables:

=	expand the value when using the variable
:=	immediately expand the value
+=	append (with space)
=+	prepend (with space)
.=	append (without space)
=.	prepend (without space)
?=	assign if no other value was previously assigned
??=	same as previous, with a lower precedence

Operators 2/2

- Avoid using `+=`, `=+`, `.=` and `=.` in `$BUILDDIR/conf/local.conf` due to ordering issues.
  - If `+=` is parsed before `?=`, the latter will be discarded.
  - Using `_append` unconditionally appends the value.

Packages 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, 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_linux-yocto = "3.10\%"`
  - `PREFERRED_VERSION_python = "2.7.3"`

Packages

Selection

- 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.

Exclusion

- The list of packages to install is also filtered using the PACKAGE_EXCLUDE variable.
- However, if a package needs installing to satisfy a dependency, it will still be selected.

The power of BitBake

Common BitBake options

- BitBake can be used to run a full build for a given target with `bitbake [target]`.
- But it can be more precise, with optional options:

  `-c <task>` execute the given task

  `-s` list all locally available packages and their versions

  `-f` force the given task to be run by removing its stamp file

  `world` keyword for all recipes

  `-b <recipe>` execute tasks from the given `recipe` (without resolving dependencies).

BitBake examples

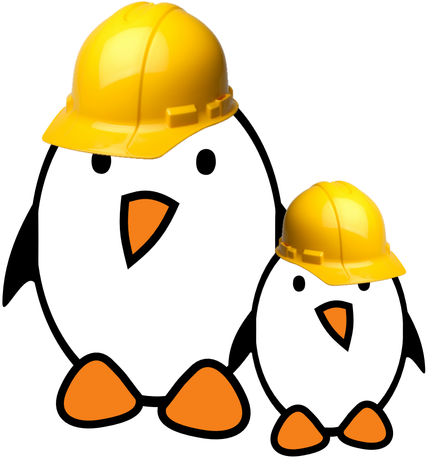
- `bitbake -c listtasks virtual/kernel`
  - Gives a list of the available tasks for the recipe providing the package `virtual/kernel`. Tasks are prefixed with `do_`.
- `bitbake -c menuconfig virtual/kernel`
  - Execute the task `menuconfig` on the recipe providing the `virtual/kernel` package.
- `bitbake -f dropbear`
  - Force the `dropbear` recipe to run all tasks.
- `bitbake world --runall=fetch`
  - 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. Its location is controlled by the SSTATE_DIR variable.
- This cache is use to speed up compilation.
- Over time, as you compile more recipes, it can grow quite big. It is possible to clean old data with:

```
$ ./scripts/sstate-cache-management.sh --remove-duplicated -d \
  --cache-dir=<SSTATE_DIR>
```

Practical lab - Advanced Yocto configuration



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

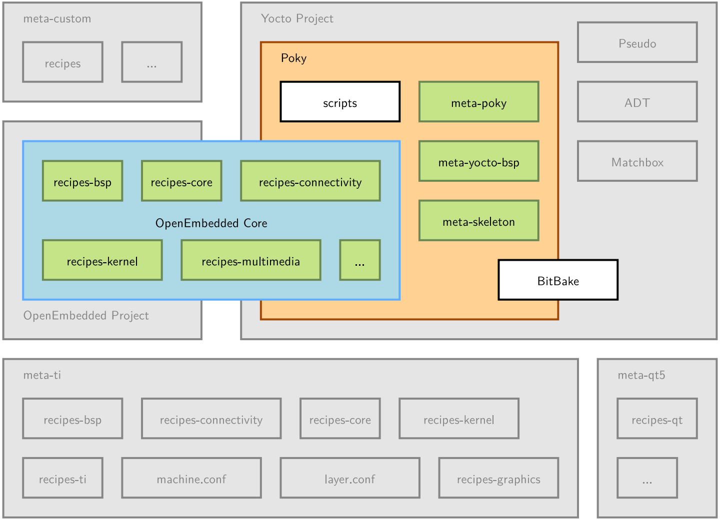
Layers

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-`.

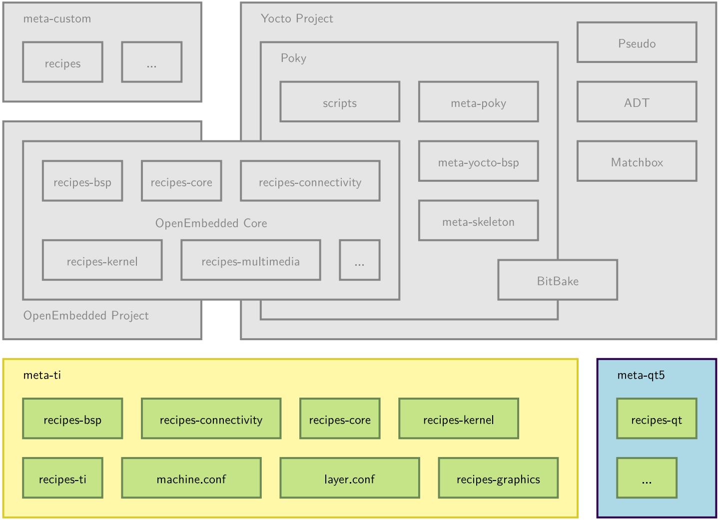
Layers in Poky



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 not to modify Poky. Instead you can create your own layers!

Poky


meta-custom
recipes
...
Yocto Project
Poky
scripts
meta-poky
Pseudo
ADT
Matchbox
meta-yocto-bsp
meta-skeleton
BitBake
recipes-bsp
recipes-core
recipes-connectivity
OpenEmbedded Core
recipes-kernel
recipes-multimedia
...
OpenEmbedded Project
meta-ti
recipes-bsp
recipes-connectivity
recipes-core
recipes-kernel
recipes-ti
machine.conf
layer.conf
recipes-graphics
meta-qt5
recipes-qt
...

Integrate and use a layer 1/3

- A list of existing and maintained layers can be found at https://layers.openembedded.org/layerindex/branch/master/layers/
- 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`

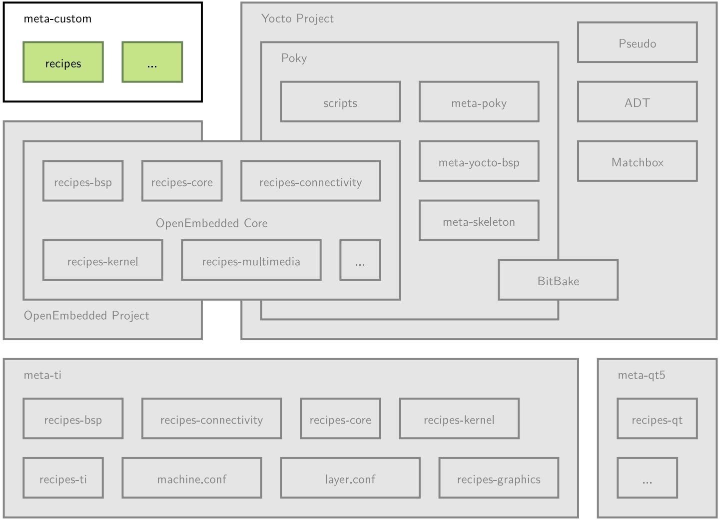
Some useful layers

- Many SoC specific layers are available, providing support for the boards using these SoCs. Some examples: `meta-ti`, `meta-freescale` and `meta-raspberrypi`.
- 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-gstreamer10`: support for GStreamer 1.0.
  - `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. The `meta-browser` did not have a krogoth branch, for example.

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>`

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.

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

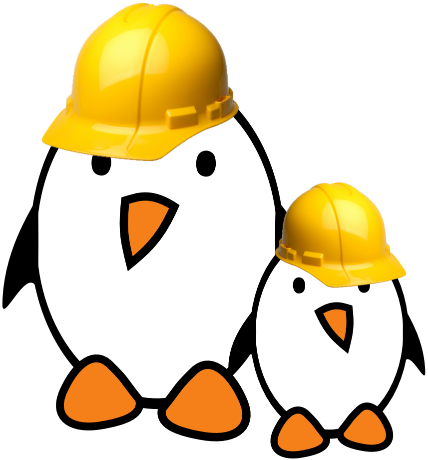
  README
  : A basic description of the layer. Contains a contact e-mail to update.

- By default, all metadata matching `./recipes-*/*/*.bb` will be parsed by the BitBake *build engine*.

Use 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, in `OEROOT`.
- 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

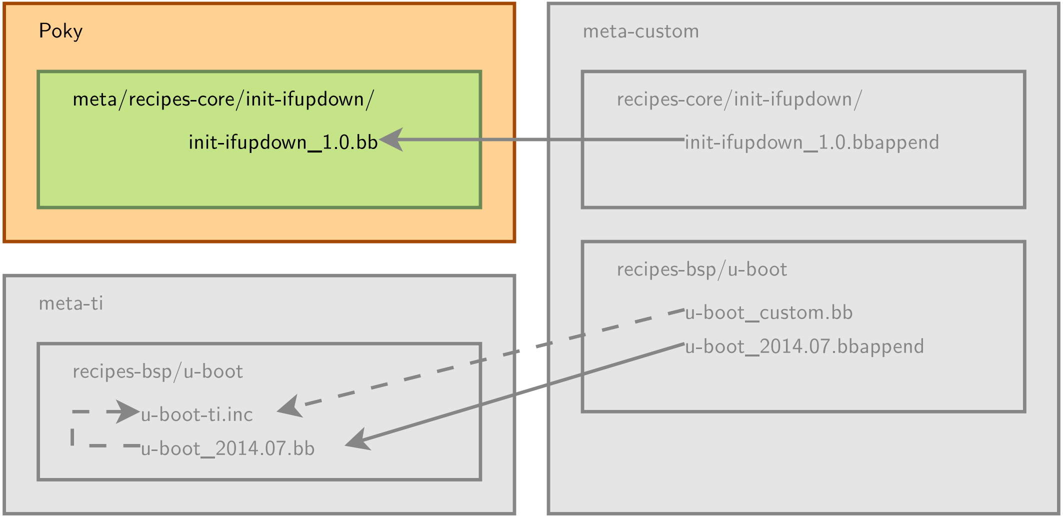


- Create a layer from scratch
- Add recipes to the new layer
- Integrate it to the build

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
recipes-bsp/u-boot
u-boot-ti.inc
u-boot_2014.07.bb

Basics

- Recipes describe how to handle a given application.
- A recipe is a set of instructions to describe how to retrieve, patch, compile, install and generate binary packages for a given application.
- It also defines what build or runtime dependencies are required.
- The recipes are parsed by the BitBake build engine.
- The format of a recipe file name is `<application-name>_<version>.bb`

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

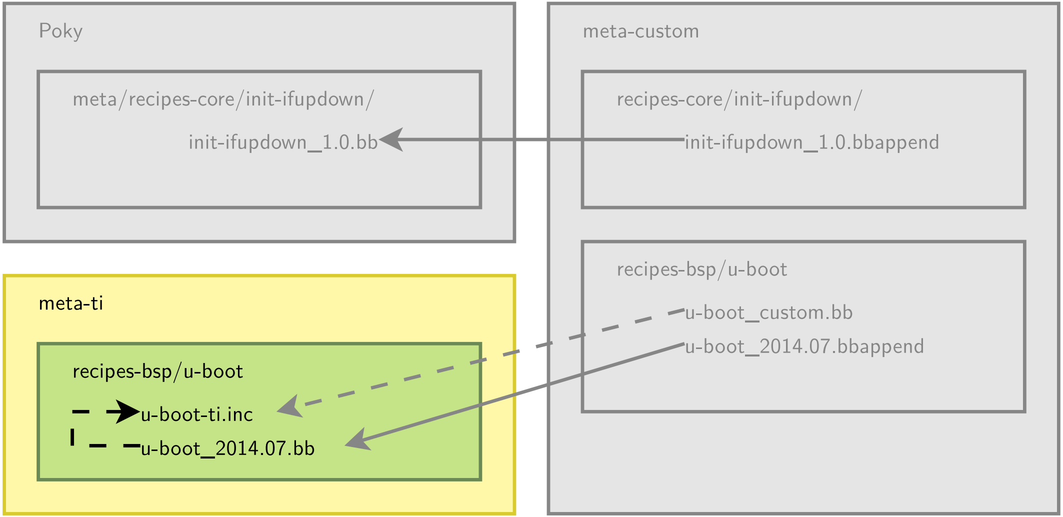
- To make it easier to write a recipe, some variables are automatically available:

PN	package name, as specified in the recipe file name
PV	package version, as specified in the recipe file name
PR	package revision, defaults to `r0`

- The recipe name and version usually match the upstream ones.
- When using the recipe `bash_4.2.bb`:
  - `${PN} = "bash"`
  - `${PV} = "4.2"`

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` and 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:
  - **DESCRIPTION** describes what the software is about
  - **HOMEPAGE** URL to the project's homepage
  - **PRIORITY** defaults to `optional`
  - **SECTION** package category (e.g. `console/utils`)
  - **LICENSE** the application's license

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 `git` scheme:
  - `git://<url>;protocol=<protocol>;branch=<branch>`
  - When using git, it is necessary to also define `SRCREV`. If `SRCREV` is a hash or a tag not present in master, the branch parameter is mandatory. When the tag is not in any branch, it is possible to use `nobranch=1`
- 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>/${PN}-${PV}.tar.gz`
  - See `meta/conf/bitbake.conf`

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

- All local files found in SRC_URI are copied into the recipe's working directory, in $BUILDDIR/tmp/work/.
- The searched paths are defined in the FILESPATH variable.

```
FILESPATH = "${@base_set_filespath([
                "${FILE_DIRNAME}/${PN}",
                "${FILE_DIRNAME}/${PN}-${PV}",
                "${FILE_DIRNAME}/files"], d)}"

FILESOVERRIDES = "${MACHINEOVERRIDES}:${DISTROOVERRIDES}"
```

- The base_set_filespath(path) function uses its path parameter, FILESEXTRAPATHS and FILESOVERRIDES to fill the FILESPATH variable.
- Custom paths and files can be added using FILESEXTRAPATHS and FILESOVERRIDES.
- Prepend the paths, as the order matters.

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_configure` task depends on the `do_populate_sysroot` task of recipe-b.
- `RDEPENDS_${PN} = "recipe-b"`: the local `do_build` task depends on the `do_package_write_<archive-format>` task of recipe b.

Dependencies 2/2

- Sometimes a recipe have 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 exists, 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
```

Writing tasks 1/2

- Functions use the sh shell syntax, with available OpenEmbedded variables and internal functions available.

  D The destination directory (root directory of where the files are installed, before creating the image).

  WORKDIR the recipe's working directory

- Syntax of a task:

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

Writing tasks 2/2

- Example:

```
do_compile() {
    oe_runmake
}

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

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}
}
```

Adding new tasks

Tasks can be added with addtask

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

addtask do_mkimage after do_compile before do_install
```

Applying patches

Patches 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.
  - In certain simple cases the `-e` option of `make` can be used.
  - The `-e` option gives variables taken from the environment precedence over variables from `Makefiles`.
  - Helps when an upstream `Makefile` uses hardcoded `CC` and/or `CFLAGS`.
- To apply patches before they get their way into the upstream version.

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.

```
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

```
DESCRIPTION = "Hello world program"
HOMEPAGE = "http://example.net/hello/"
PRIORITY = "optional"
SECTION = "examples"
LICENSE = "GPLv2"

SRC_URI = "git://git.example.com/hello;protocol=https"
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 = "GPLv2"
LIC_FILES_CHKSUM =  \
  "file://COPYING;md5=59530bdf33659b29e73d4adb9f9f6552"

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

SRC_URI[md5sum] = "c6c4f1c075dbf0f75c29737faa58f290"
```

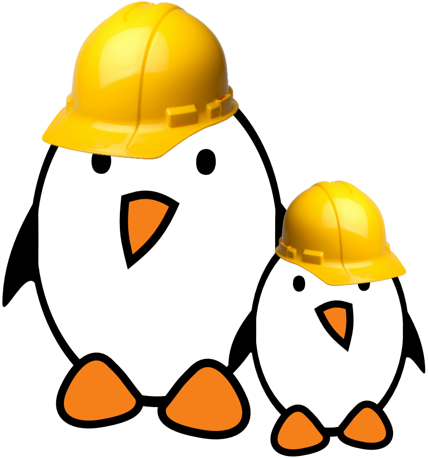
tar_1.26.bb

```
require tar.inc

LICENSE = "GPLv3"
LIC_FILES_CHKSUM = \
  "file://COPYING;md5=d32239bcb673463ab874e80d47fae504"

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

Practical lab - Add a custom application



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

Writing recipes - advanced

Extending a recipe

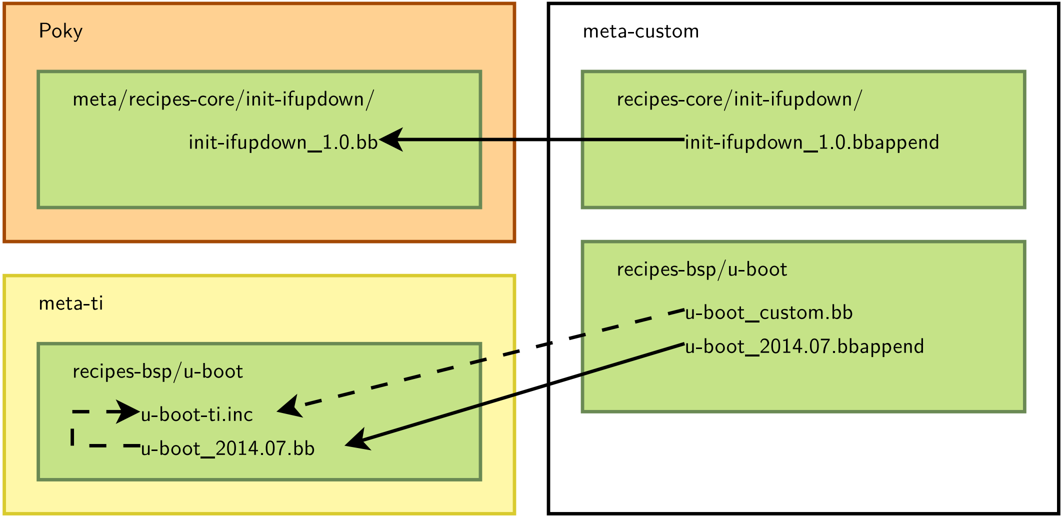
Introduction to recipe extensions

- It is a good practice **not** to modify recipes available in Poky.
- But it is sometimes useful to modify an existing recipe, to apply a custom patch 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

- Metadata can be changed, added or appended.
- Tasks can be added or appended.
- Operators are used extensively, to add, append, prepend or assign values.

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.
  - `example_0.1.bbappend` applies to `example_0.1.bb`
- Append files are **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

Hello world append file

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"

SRC_URI += "file://custom-modification-0.patch \
            file://custom-modification-1.patch \
            "
```

Advanced recipe configuration

Advanced configuration

- In the real word, more complex configurations are often needed because recipes may:
  - Provide virtual packages
  - Inherit generic functions from classes

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 are located in the `classes` folder of a layer.
- Recipes can use this common code by inheriting a class:
  - `inherit <class>`

Common classes

- Common classes can be found in `meta/classes/`
  - `base.bbclass`
  - `kernel.bbclass`
  - `autotools.bbclass`
  - `autotools-brokensep.bbclass`
  - `cmake.bbclass`
  - `native.bbclass`
  - `systemd.bbclass`
  - `update-rc.d.bbclass`
  - `useradd.bbclass`
  - ...

The base class

- Every recipe inherits the base class automatically.
- Contains a set of basic common tasks to fetch, unpack or compile applications.
- Inherits other common classes, providing:
  - Mirrors definitions: `DEBIAN_MIRROR`, `GNU_MIRROR`, `KERNELORG_MIRROR`...
  - The ability to filter patches by `SRC_URI`
  - Some tasks: `clean`, `listtasks` or `fetch`.
- 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.
- 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/"
PRIORITY = "optional"
SECTION = "examples"
LICENSE = "GPLv3"

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. Defines the list of packages which needs the user.
- 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"
PRIORITY = "optional"
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 = "-u 1000 -d /home/user0 -s /bin/bash user0"

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

Binary packages

Specifics for binary packages

- It is possible to install binaries into the generated root filesystem.
- Set the `LICENSE` to `CLOSED`.
- Use the `do_install` task to copy the binaries into the root file system.

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` directories found in `BBPATH`.
- It is possible to include a class conditionally using a variable: `inherit ${FOO}`

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: `include ninvaders.inc`
- To include a file from another location (which could be in another layer): `include path/to/file.inc`

Debugging recipes

Debugging recipes

- For each task, logs are available in the `temp` directory in the work folder of a recipe.
- 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.

  - `./scripts/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 Poky

```
PREMIRRORS ??= "\
bzr://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
cvs://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
git://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
hg://.*/.*    http://downloads.yoctoproject.org/mirror/sources/ \n \
osc://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
p4://.*/.*    http://downloads.yoctoproject.org/mirror/sources/ \n \
svk://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
svn://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n"

MIRRORS =+ "\
ftp://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
http://.*/.*  http://downloads.yoctoproject.org/mirror/sources/ \n \
https://.*/.* http://downloads.yoctoproject.org/mirror/sources/ \n"
```

Configuring the mirrors

- It's possible to prepend custom mirrors, using the PREMIRRORS variable:

```
PREMIRRORS_prepend = "\
 git://.*/.* http://www.yoctoproject.org/sources/   \n \
 ftp://.*/.* http://www.yoctoproject.org/sources/   \n \
 http://.*/.* http://www.yoctoproject.org/sources/  \n \
 https://.*/.* http://www.yoctoproject.org/sources/ \n"
```

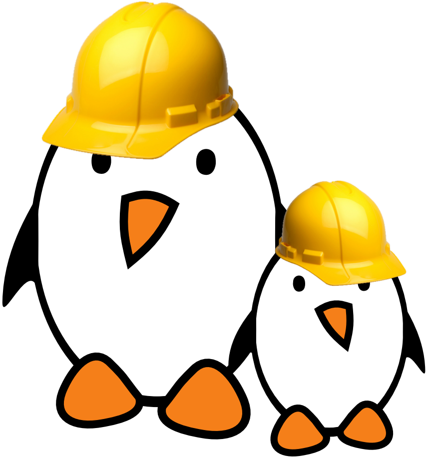
- Another solution is to use the own-mirrors class:

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

Forbidding network access

- You can use `BB_GENERATE_MIRROR_TARBALLS = "1"` to generate tarballs of the git repositories in `DL_DIR`
- You can also completely disable network access using `BB_NO_NETWORK = "1"`
- Or restrict BitBake to only download files from the `PREMIRRORS`, using `BB_FETCH_PREMIRRORONLY = "1"`

Practical lab - Extend a recipe

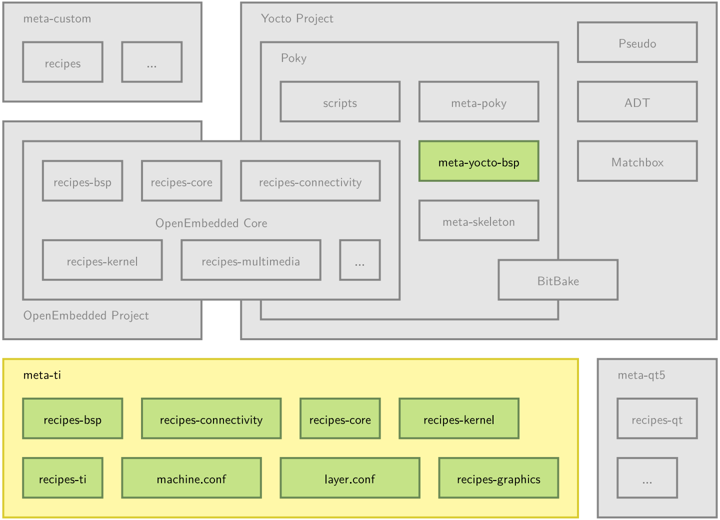


- Apply patches to an existing recipe
- Use a custom configuration file for an existing recipe
- Extend a recipe to fit your needs

BSP Layers

Introduction to BSP layers in the Yocto Project

BSP layers



meta-custom
recipes
...
Yocto Project
Poky
scripts
meta-poky
meta-yocto-bsp
meta-skeleton
Pseudo
ADT
Matchbox
BitBake
recipes-bsp
recipes-core
recipes-connectivity
OpenEmbedded Core
recipes-kernel
recipes-multimedia
...
OpenEmbedded Project
meta-ti
recipes-bsp
recipes-connectivity
recipes-core
recipes-kernel
recipes-ti
machine.conf
layer.conf
recipes-graphics
meta-qt5
recipes-qt
...

Overview

- BSP layers are device specific layers. They hold metadata with the purpose of supporting specific hardware devices.
- BSP layers describe the hardware features and often provide a custom kernel and bootloader with the required modules and drivers.
- BSP layers can also provide additional software, designed to take advantage of the hardware features.
- As a layer, it is integrated into the build system as we previously saw.
- A good practice is to name it `meta-<bsp_name>`.

BSP layers Specifics

- BSP layers are a subset of the layers.
- In addition to package recipes and build tasks, they often provide:
  - Hardware configuration files (`machines`).
  - Bootloader, kernel and display support and configuration.
  - Pre-built user binaries.

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/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 console to attach. Passed to the kernel as the `console` parameter, e.g. `115200;ttyS0`

KERNEL_IMAGETYPE The type of kernel image to build, e.g. `zImage`

MACHINE_FEATURES

- 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 feature `bluetooth`:
  - Asks the `bluez` daemon to be built and added to the image.
  - Enables bluetooth support in ConnMan.

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_CONSOLE = "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>

include conf/machine/include/cfa10036.inc

KERNEL_DEVICETREE += "imx28-cfa10057.dtb"

MACHINE_FEATURES += "touchscreen"
```

Bootloader

Default bootloader 1/2

- By default 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 basically two ways of compiling a kernel in the Yocto Project:
  - By using the `linux-yocto` packages, provided in Poky.
  - By using a fully custom kernel recipe.
- 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 1/4

- `linux-yocto` is a generic set of recipes for building mainline Linux kernel images.
- `PREFERRED_PROVIDER_virtual/kernel = "linux-yocto"`
- `PREFERRED_VERSION_linux-yocto = "3.14\%"`

Linux Yocto 2/4

- Like other appended recipes, patches can be added by filling SRC_URI with .patch and/or .diff files.
- The kernel configuration must also be provided, and the file containing it must be called defconfig.
  - This can be generated from a Linux source tree, by using make savedefconfig
  - The configuration can be split in several files, by using the .cfg extension. It is the best practice when adding new features:

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

- Configuration fragments can be generated directly with the `bitbake` command:
  1. Configure the kernel following its recipe instructions:
     `bitbake -c kernel_configme linux-yocto`
  2. Edit the configuration: `bitbake -c menuconfig linux-yocto`
  3. Save the configuration differences: `bitbake -c diffconfig linux-yocto`
     - The differences will be saved at `$WORKDIR/fragment.cfg`
- After integrating configuration fragments into the appended recipe, you can check everything is fine by running:
  `bitbake -c kernel_configcheck -f linux-yocto`

Linux Yocto 4/4

- 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.
- The full documentation can be found at https://www.yoctoproject.org/docs/2.5/kernel-dev/kernel-dev.html#kernel-dev-advanced

Linux Yocto: Kernel Metadata 1/4

- 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 2/4

- Kernel Metadata can be stored in the `linux-yocto` recipe space.
- It must be under `$FILESEXTRAPATHS`. A best practice is to follow this directory hierarchy:

  bsp/
  cfg/
  features/
  ktypes/
  patches/

- Kernel Metadata are divided into 3 file types:
  - Description files, ending in `.scc`
  - Configuration fragments
  - Patches

Linux Yocto: Kernel Metadata 3/4

- Kernel Metadata description files have their own syntax, used to describe the feature provided and which patches and configuration fragments to use.
- Simple example, `features/smp.scc`

```
define KFEATURE_DESCRIPTION "Enable SMP"

kconf hardware smp.cfg
patch smp-support.patch
```

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

Linux Yocto: Kernel Metadata 4/4

- `.scc` syntax description:

  `branch <ref>` Create a new branch relative to the current one.

  `define` Defines variables.

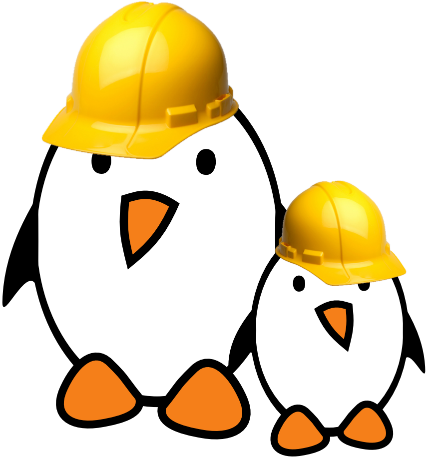
  `include <scc file>` Include another description file. Parsed inline.

  `kconf [hardware|non-hardware] <cfg file>` Queues a configuration fragment, to merge it into Linux's `.config`

  `git merge <branch>` Merge `branch` into the current git branch.

  `patch <patch file>` Applies `patch file` to the current git branch.

Practical lab - Create a custom machine configuration

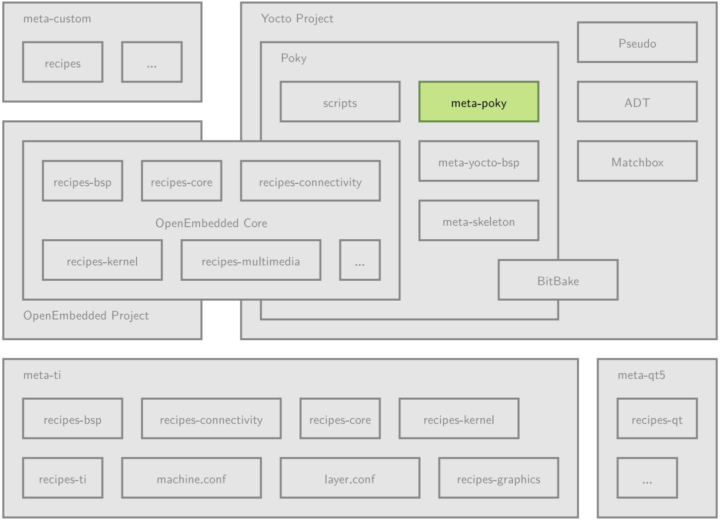


- Write a machine configuration
- Understand how the target architecture is chosen

Distro Layers

Distro Layers

Distro layers



Distro layers

- You can create a new distribution by using a Distro layer.
- This allows to change the defaults that are used by `Poky`.
- It is useful to distribute changes that have been made in `local.conf`

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.
  - 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 = "..."
```

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.

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/`.
- The `TEMPLATECONF` variable controls where to find the samples.
- It is set in `${OEROOT}/.templateconf`.

Distribute the distribution

- 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 `manifest` file, usually available in its own `git` repository.

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="dunfell" 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.

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

- Common images are:

  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-rt `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 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.

  IMAGE_ROOTFS_SIZE The final root filesystem size.

  IMAGE_FEATURES List of features to enable in the image.

  IMAGE_FSTYPES List of formats the OpenEmbedded build system will use to create images.

  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.

Example of an image

```
require recipes-core/images/core-image-minimal.bb

DESCRIPTION = "Example image"

IMAGE_INSTALL += "ninvaders"

IMAGE_FSTYPES = "tar.bz2 cpio squashfs"

LICENSE = "MIT"
```

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 delivered in Poky, thanks to `meta/classes/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>`.

wic

- wic is a tool that can create a flashable image from the compiled packages and artifacts.
- It can create partitions.
- It can select which files are located in which partition through the use of plugins.
- The final image layout is described in a .wks file.
- It can be extended in any layer.
- Usage example:

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

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.
- They can be found under `meta*/recipes-core/packagegroups/`
- A package group is yet another recipe.
- The prefix `packagegroup-` is always used.

Common package groups

- packagegroup-core-boot
- packagegroup-core-buildessential
- packagegroup-core-nfs-client
- packagegroup-core-nfs-server
- packagegroup-core-tools-debug
- packagegroup-core-tools-profile

Example

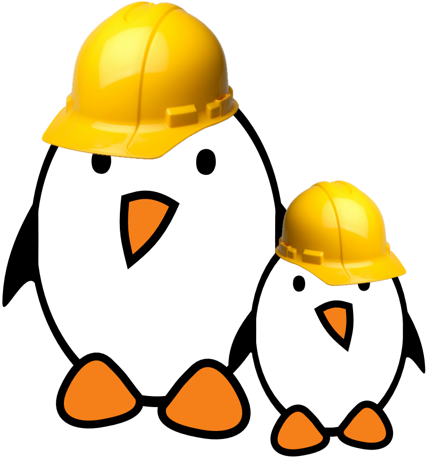
./meta/recipes-core/packagegroups/packagegroup-core-tools-debug.bb:

```
SUMMARY = "Debugging tools"
LICENSE = "MIT"

inherit packagegroup

RDEPENDS_${PN} = "\
    gdb \
    gdbserver \
    strace"
```

Practical lab - Create a custom image



- Write an image recipe
- Choose the packages to install

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 = "GPLv3"
```

- 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_WHITELIST` variable.
- Example, `gst-plugins-ugly`:

```
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly"
```

Manifest of packages licenses

- A summary of all the packages and their licenses is available in:
  `$BUILDDIR/tmp/deploy/licenses/<image>/license.manifest`

Writing recipes - going further

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 four 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.
- 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"
```

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.
- Example:

```
PACKAGECONFIG ??= "                                                    \
    ${@bb.utils.contains('DISTRO_FEATURES', 'wifi','wifi', '', d)}          \
    ${@bb.utils.contains('DISTRO_FEATURES', 'bluetooth','bluetooth', '', d)} \
    ${@bb.utils.contains('DISTRO_FEATURES', '3g','3g', '', d)}"
```

Python tasks

Tasks in Python

- Tasks can be written in Python when using the keyword `python`.
- The `d` variable is accessible, and represents the BitBake datastore (where variables are stored).
- 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 keyword `import`.
- Anonymous Python functions are executed during parsing.

Accessing the datastore with Python

- The `d` variable is accessible within Python tasks.

`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)` Expend variables in `expression`.

Python task example

```
# Anonymous function
python () {
    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()))
}
```

Variable flags

Variable flags

- *Variable flags* are used to store extra information on tasks and variables.
- They are used to control task functionalities.
- A number of these flags are already used by BitBake:
  - `dirs`: directories that should be created before the task runs. The last one becomes the work directory for the task.
  - `noexec`: disable the execution of the task.
  - `nostamp`: do not create a *stamp* file when running the task. The task will always be executed.
  - `doc`: task documentation displayed by *listtasks*.

```
do_settime[noexec] = "1"
do_compile[nostamp] = "1"
do_settime[doc] = "Set the current time in ${TIME}"
```

Root filesystem creation

Files and directories selection

- The FILES variable controls the list of files and directories to be placed into packages.
- It must be package specific (e.g. with _${PN}).
- 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"
```

- To prevent configuration files to be overwritten during the Package Management System update process, use CONFFILES.

Root filesystem generation

- Image generation overview:
  1. The rootfs is created using packages.
  2. One or more images files are created, depending on the IMAGE_FSTYPES value.
- The rootfs 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/rootfs_${IMAGE_PKGTYPE}.bbclass

Example: rootfs creation with .deb packages

```
rootfs_deb_do_rootfs () {
    [...]

    export INSTALL_ROOTFS_DEB="${IMAGE_ROOTFS}"

    [...]

    apt update
    apt ${APT_ARGS} install ${package_to_install}  \
        --force-yes --allow-unauthenticated

    [...]
}
```

Splitting packages

Benefits

- Packages can be split.
- Useful when a single remote repository provides multiple binaries or libraries.
- The list of packages to provide is defined by the `PACKAGES` variable.
- The `FILES` variable is often used to split the output into packages.

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"
SRC_URI[sha256sum] = "..."

PACKAGES =+ "kexec kdump"

FILES_kexec = "${sbindir}/kexec"
FILES_kdump = "${sbindir}/kdump"
```

Default packages

- By default several packages are produced automatically:

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

- For each of these packages a `FILES` variable is defined in `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 ${base_libdir}/*.la"
FILES_${PN}-dbg = "/usr/lib/debug /usr/src/debug"
```

Application development workflow

Recommended workflows

- Different development workflows are possible given the needs:
  - Low-level application development (bootloader, kernel).
  - Application development.
  - Temporary modifications on an external project (bug fixes, security fixes).
- Three workflows exists for theses needs: the SDK, `devtool` and `quilt`.

The Yocto Project SDK

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.

Advantages

- The Yocto Project is often used to build images for embedded targets.
  - This often requires a special toolchain, to cross compile the software.
  - Some libraries headers may be specific to the target and not available on the developers' computers.
- A self-sufficient environment makes development easier and avoids many errors.
- Long manuals are not necessary, the only thing required is the SDK!
- Using the SDK to develop an application limits the risks of dependency issues when running it on the target.

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.
    - Its toolchain 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-2.5.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-2.5.sh`
- The SDK will be configured to be compatible with the specified `MACHINE`.

SDK format

- Both SDKs are distributed as bash 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/2.5`

```
$ ./poky-glibc-x86_64-meta-toolchain-cortexa8hf-neon-toolchain-2.5.sh
Poky (Yocto Project Reference Distro) SDK installer version 2.5
===============================================================
Enter target directory for SDK (default: /opt/poky/2.5):
You are about to install the SDK to "/opt/poky/2.5". 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/2.5/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/2.5
$ 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 variables is set to be used with the C compiler (gcc).
  - When building the Linux kernel, unset this variable.

```
$ unset LDFLAGS
$ make menuconfig
$ make
```

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.

devtool usage 1/3

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 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.

Quilt

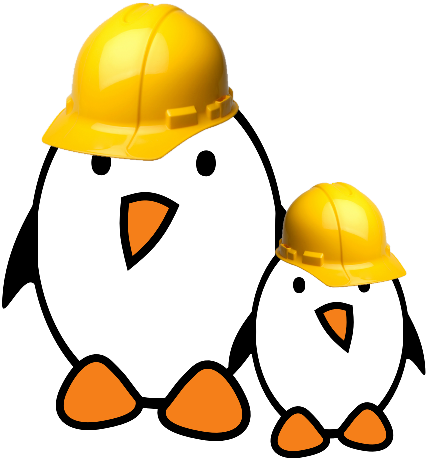
Overview

- 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.

Practical lab - Create and use a Poky SDK



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

Runtime Package Management

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 (yum).

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
```

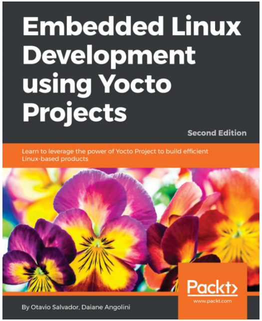
Yocto Project Resources

Yocto Project documentation

- https://www.yoctoproject.org/documentation
- Wiki: https://wiki.yoctoproject.org/wiki/Main_Page
- http://recipes.yoctoproject.org

Embedded Linux Development with Yocto Project - Second Edition, Nov 2017

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

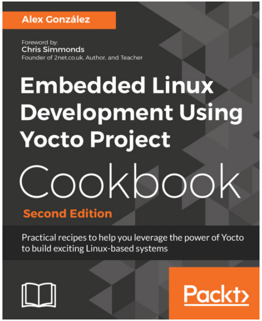


Useful Reading (2)

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

Corrections, suggestions, contributions and translations are welcome!

Last slide

Thank you!

And may the Source be with you