Yocto Project and OpenEmbedded Training

Free Electrons

Latest update: June 11, 2015.

Document updates and sources:
`http://free-electrons.com/doc/training/yocto`

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

Rights to copy

Hyperlinks in the document

There are many hyperlinks in the document

- Regular hyperlinks:
  http://kernel.org/
- Kernel documentation links:
  Documentation/kmemcheck.txt
- 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

Free Electrons at a glance

- Engineering company created in 2004
  (not a training company!)
- Locations: Orange, Toulouse, Lyon (France)
- Serving customers all around the world
  See `http://free-electrons.com/company/customers/`
- Head count: 9
  Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel, Android Free Software / Open Source for embedded and real-time systems.
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

Free Electrons on-line resources

- All our training materials:
  http://free-electrons.com/docs/
- Technical blog:
  http://free-electrons.com/blog/
- Quarterly newsletter:
  http://lists.free-electrons.com/mailman/listinfo/newsletter
- News and discussions (Google +):
  https://plus.google.com/+FreeElectronsDevelopers
- News and discussions (LinkedIn):
  http://linkedin.com/groups/Free-Electrons-4501089
- Quick news (Twitter):
  http://twitter.com/free_electrons
- Linux Cross Reference - browse Linux kernel sources on-line:
  http://lxr.free-electrons.com

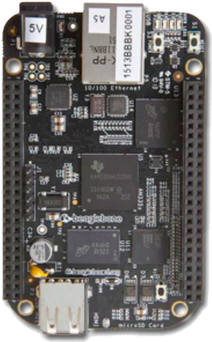
Generic course information

Free Electrons

Hardware used in this training session

BeagleBone Black, from CircuitCo

- Texas Instruments AM335x (ARM Cortex-A8)
- Powerful CPU, with 3D acceleration, additional processors (PRUs) and lots of peripherals.
- 512 MB of RAM
- 2 GB of on-board eMMC storage (4 GB in Rev C)
- USB host and USB device ports
- microSD slot
- HDMI port
- 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 `http://beagleboardtoys.com/`.



Do not damage your BeagleBone Black!

- Do not remove power abruptly:
  - Boards components have been damaged by removing the power or USB cable in an abrupt way, not leaving the PMIC the time to switch off the components in a clean way. See `http://bit.ly/1FWHNZi`
  - Reboot (`reboot`) or shutdown (`halt`) the board in software when Linux is running.
  - You can also press the `RESET` button to reset and reboot.
  - When there is no software way, you can also switch off the board by pressing the `POWER` button for 8 seconds.
- Do not leave your board powered on a metallic surface (like a laptop with a metal finish).

Course outline - Day 1

First dive into the Yocto Project.

- Overview of an embedded Linux system architecture.
- Organization of the Yocto Project source tree.
- Customizing an image.
- Building an image.

Labs: download the Yocto project sources, compile an image and flash the development board.

Course outline - Day 2

Recipes and layers details: write, use, customize.

- Recipes syntax. Writing a recipe.
- Development workflow in the Yocto Project with BitBake.
- Adding packages to the generated image.
- The Yocto Project layers. Adding a new layer.

Labs: add a custom application and its recipe to the build system, create a new layer.

Course outline - Day 3

The Yocto Project as a BSP provider.

- Extending a recipe.
- Writing your own machine configuration.
- Adding a custom image.
- Using the SDK with Eclipse.

Labs: integrate kernel changes into the build system, write a machine configuration, create a custom image, experiment with the 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 / Android 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 copy and paste from the PDF slides.
  The slides contain UTF-8 characters that look the same as ASCII ones, but won't be understood by shells or compilers.

Cooperate!

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

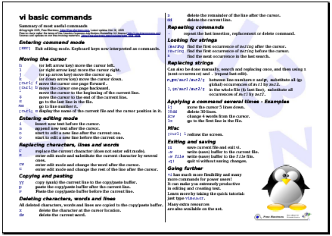
- If you complete your labs before other people, don't hesitate to help other people and investigate the issues they face. The faster we progress as a group, the more time we have to explore extra topics.
- Explain what you understood to other participants when needed. It also helps to consolidate your 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 `http://free-electrons.com/doc/training/embedded-linux/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 `http://free-electrons.com/doc/training/embedded-linux/vi_memento.pdf`
- You can also take the quick tutorial by running vimtutor. This is a worthy investment!



vi basic commands

Summary of most useful commands

Entering command mode

[ESC] Exit editing mode. Keyboard keys now interpreted as commands.

Moving the cursor

h (or left arrow key) move the cursor left.
l (or right arrow key) move the cursor right.
j (or up arrow key) move the cursor up.
k (or down arrow key) move the cursor down.
[Ctrl] f move the cursor one page forward .
[Ctrl] b move the cursor one page backward.
^ move the cursor to the beginning of the current line.
$ move the cursor to the end of the current line.
G go to the last line in the file.
nG go to line number n.
[Ctrl] g display the name of the current file and the cursor position in it.

Entering editing mode

i insert new text before the cursor.
a append new text after the cursor.
o start to edit a new line after the current one.
O start to edit a new line before the current one.

Replacing characters, lines and words

r replace the current character (does not enter edit mode).
s enter edit mode and substitute the current character by several ones.
cw enter edit mode and change the word after the cursor.
C enter edit mode and change the rest of the line after the cursor.

Copying and pasting

yy copy (yank) the current line to the copy/paste buffer.
p paste the copy/paste buffer after the current line.
P Paste the copy/paste buffer before the current line.

Deleting characters, words and lines

All deleted characters, words and lines are copied to the copy/paste buffer.

x delete the character at the cursor location.
dw delete the current word.
D delete the remainder of the line after the cursor.
dd delete the current line.

Repeating commands

. repeat the last insertion, replacement or delete command.

Looking for strings

/string find the first occurrence of string after the cursor.
?string find the first occurrence of string before the cursor.
n find the next occurrence in the last search.

Replacing strings

Can also be done manually, search and replacing once, and then using n (next occurrence) and . (repeat last edit).

n,ps/str1/str2/g between line numbers n and p, substitute all (g: global) occurrences of str1 by str2.
1,$s/str1/str2/g in the whole file ($: last line), substitute all occurrences of str1 by str2.

Applying a command several times - Examples

5j move the cursor 5 lines down.
30dd delete 30 lines.
4cw change 4 words from the cursor.
1G go to the first line in the file.

Misc

[Ctrl] l redraw the screen.

Exiting and saving

ZZ save current file and exit vi.
:w write (save) buffer to the current file.
:w file write (save) buffer to the file file.
:q! quit vi without saving changes.

Going further

vi has much more flexibility and many more commands for power users!
It can make you extremely productive in editing and creating text.
Learn more by taking the quick tutorial: just type vimtutor.
Many extra resources are also available on the net.

Prepare your lab environment

- Download the lab archive
- Enforce correct permissions

Introduction to embedded Linux build systems

Free Electrons

Embedded Linux distribution projects

Build system definition

- Purposes of a build system:
  - Compiling or cross-compiling applications.
  - Packaging applications.
  - Testing output binaries and ecosystem compatibility.
  - Deploying generated images.

Available system building tools

Large choice of tools

- **Buildroot**, developed by the community
  http://www.buildroot.org
- **PTXdist**, developed by Pengutronix
  http://pengutronix.de/software/ptxdist/
- **OpenWRT**, originally a fork of Buildroot for wireless routers, now a more generic project
  http://www.openwrt.org
- **OpenEmbedded** based build systems
  http://www.openembedded.org:
  - Poky (from the Yocto Project)
  - Arago Project
  - Ångström
- Vendor specific tools (silicon vendor or embedded Linux vendor)

Comparison of distribution projects

- Buildroot
  - Simple to use.
  - Adapted for small embedded devices.
  - Not perfect if you need advanced functionalities and multiple machines support.
  - `http://buildroot.org/`

Comparison of distribution projects

- OpenWRT
  - Based on Buildroot.
  - Primarily used for embedded network devices like routers.
  - `http://openwrt.org/`

Comparison of distribution projects

- Poky
  - Part of the Yocto Project.
  - Using OpenEmbedded.
  - Suitable for more complex embedded systems.
  - Allows lots of customization.
  - Can be used for multiple targets at the same time.
  - `http://yoctoproject.org/`

Build system benefits

Working without a build system

- Each application has to be built manually, or using custom and non stable scripts.
- The root file system has to be created from scratch.
- The applications configurations have to be done by hand.
- Each dependency has to be matched manually.
- Integrating softwares from different teams is painful.

Benefits

- Build systems automate the process of building a target system, including the kernel, and sometimes the toolchain.
- They automatically download, configure, compile and install all the components in the right order, sometimes after applying patches to fix cross-compiling issues.
- They make sure all the application dependencies are matched.
- They already contain a large number of packages, that should fit your main requirements, and are easily extensible.
- The build becomes reproducible, which allows to easily change the configuration of some components, upgrade them, fix bugs, etc.
- Several configurations can be handled in the same project. It is possible to generate the same root file system for different hardware targets or to have a debug image based on the production one, with some more flags or debugging applications.

Workflow

- Development of each application is done **out of** the build system!
  - Development is done on an external repository.
  - The build system downloads sources from this repository and start the build following the instructions.
- The build system is used to build the full system and to provide a working image to the customer.

Yocto Project and Poky reference system overview

Free Electrons

The Yocto Project overview

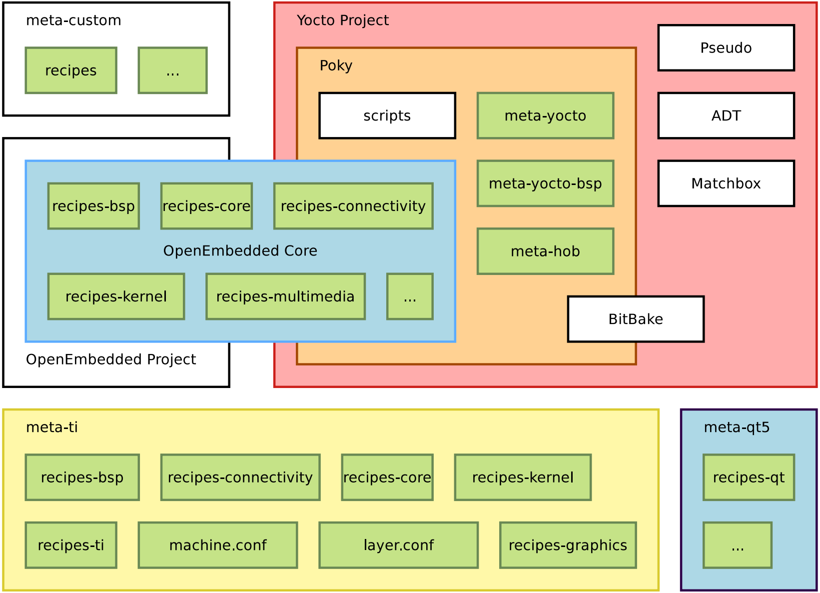
About

- 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 packages 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.
- All modifications are made in the *meta-ti* layer. Editing Poky is a **no-go**!
- We will set up this environment in the lab.

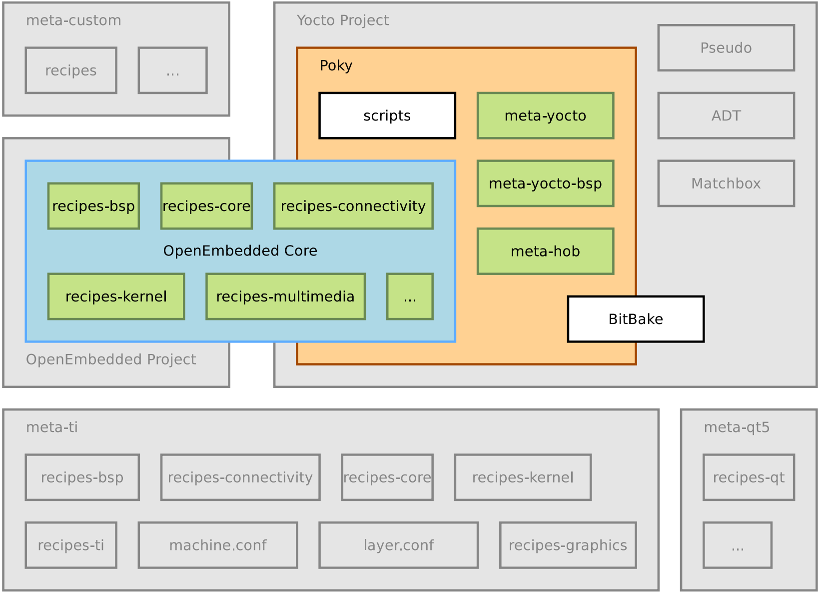
The Poky reference system overview

Download the Poky reference system

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

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

Poky


meta-custom
recipes
...
Yocto Project
Pseudo
ADT
Matchbox
Poky
scripts
meta-yocto
meta-yocto-bsp
meta-hob
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
...

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

Using Yocto Project - basics

Free Electrons

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: Includes development headers and libraries to develop directly on the target.

  adt-installer: Build the application development toolkit installer.

  meta-ide-support: Generates the cross-toolchain. Useful when working with the SDK.

Exported environment variables

BUILDDIR
: Absolute path of the build directory.

BB_ENV_EXTRAWHITE
: List of environment variables to load from the user's environment into BitBake data store.

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 packages (download, configure, compile...).
bitbake-*	Various specific commands related to the BitBake build engine.
yocto-layer	Command to create a new generic layer.
yocto-bsp	Command to create a new generic BSP.

The build/ directory 1/2

- **conf/** Configuration files. Image specific and layer configuration.
- **downloads/** Downloaded upstream tarballs of the packages 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 packages 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.

  PARALLEL_MAKE How many processes should be used when compiling.

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

  PACKAGE_CLASSES Packages format (deb, ipk or rpm).

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

Free Electrons

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 packages.
- 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 `build/conf/local.conf`

Methods and conditions 1/3

- 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).
- 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`.

Methods and conditions 2/3

- 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 3/3

- 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 `build/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`: eglibc, uclibc
  - `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.

Format

- By default, Poky uses the RPM package format
- OpenEmbedded-Core uses the IPK package format
- To select the generated package formats, use the `PACKAGE_CLASSES` variable
- Valid values are: `package_rpm`, `package_deb`, `package_ipk` (multiple values are OK)

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 packages
`-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 package `virtual/kernel`. Tasks are prefixed with `do_`.
- `bitbake -c menuconfig virtual/kernel`
  - Execute the task `menuconfig` on the kernel package.
- `bitbake -f dropbear`
  - Force the `dropbear` package to be rebuilt from scratch.
- `bitbake -c fetchall world`
  - Download all package 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>
```

Network usage

Keeping package sources on a local network

- When fetching source code, BitBake will first search in the local download directory (DL_DIR).
- It will then try to access mirrors:
  - First it tries locations from the PREMIRRORS variable, usually set by the distribution layer. Poky uses:

```
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"
```

  - Then it tries locations from the MIRRORS variable
- Use the own-mirrors class to set your mirrors:

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

- If all the mirrors fail, the build will fail.

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"

Practical lab - Advanced Yocto configuration

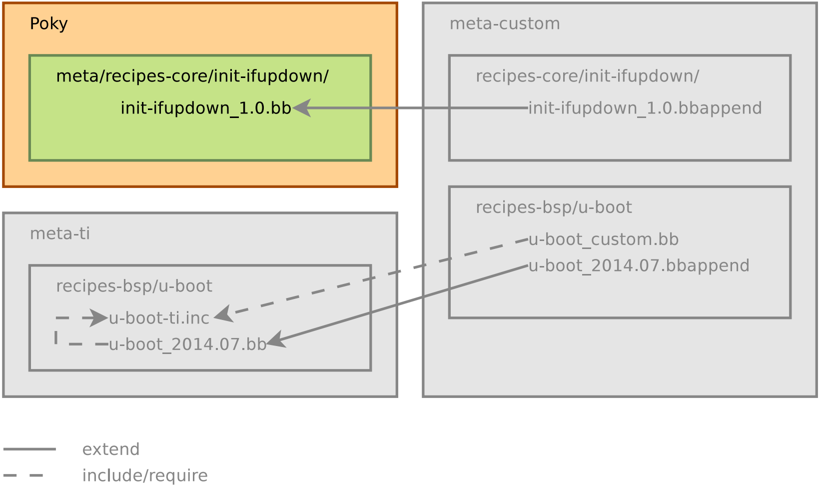
- Modify the build configuration
- Customize the package selection
- Experiment with BitBake
- Mount the root file system over NFS

Writing recipes - basics

Free Electrons

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
extend
include/require

Basics

- Recipes describe how to handle a given package.
- 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 `<package-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> <package>` command?

Common variables

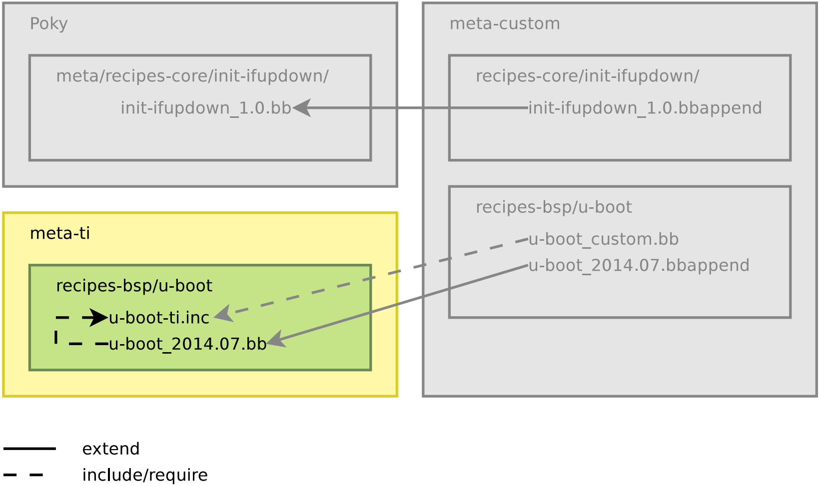
- To make it easier to write a recipe, some variables are automatically available:

PN	package name, as specified in the recipe file name
PV	package version, as specified in the recipe file name
PR	package release, defaults to `r0`
PE	package epoch: used to reorder package versions when the numbering scheme has changed

- 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 packages 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:
  - `<package>.inc`: version agnostic metadata.
  - `<package>_<version>.bb`: `require <package>.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 package:

  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 package's license

The source locations 1/3

- We need to retrieve both the raw sources from an official location and the resources needed to configure, patch or install the package.
- `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`
  - `file://joystick-support.patch`
  - `${SOURCEFORGE_MIRROR}/<project-name>/${PN}-${PV}.tar.gz`
  - `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 source locations 2/3

- For the local files, the searched paths are defined in the `FILESPATH` variable, custom ones can be added using `FILESEXTRAPATHS`. BitBake will also search in subfolders listed in the `OVERRIDES` variables in those paths.
- Prepend the paths, as the order matters.
- All local files found in `SRC_URI` are copied into the package's working directory, in `build/tmp/work/`.
- Files ending in `.patch`, `.diff` or having the `apply=yes` parameter will be applied after the sources are retrieved and extracted.
- Patches are applied in the order they are found.
- `LIC_FILES_CHKSUM` defines the URI pointing to the license file in the source code as well as its checksum. 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.
  - `LIC_FILES_CHKSUM = "file://gpl.txt;md5=393a5ca..."`

The source locations 3/3

- When the sources are provided in a tarball:
  - An md5 or an sha256 sum must be provided:
    - SRC_URI[md5sum] = "97b2c3fb082241ab5c56ab728522622b"
    - SRC_URI[sha256sum] = "..."
  - BitBake expects to find the extracted sources in a directory called <package-name>-<version>. This is controlled by the S variable. If the directory has another name, you must explicitly define S.
- By default, sources are fetched in $BUILDDIR/downloads. Change it with the DL_DIR variable in conf/local.conf
- Use the PREMIRRORS and MIRRORS variables if you want to use local mirrors or fallback mirrors.
- To ensure no network access will be done when building, use BB_NO_NETWORK = "1".

Dependencies

- A package can have dependencies during the build or at runtime. To reflect these requirements in the recipe, two variables are used:

  DEPENDS List of the package build-time dependencies.

  RDEPENDS List of the package runtime dependencies. Must be package specific (e.g. with `_${PN}`).

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

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 1/3

- 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 package's working directory

- Syntax of a task:

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

Writing tasks 2/3

- Example:

```
do_compile() {
    ${CC} ${CFLAGS} ${LDFLAGS} -o hello ${WORKDIR}/hello.c
}

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

Writing tasks 3/3

- Or using a Makefile:

```
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 0755 hello.conf ${D}${sysconfdir}
}
```

Adding new tasks

Tasks can be added with `addtask`

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

addtask mkimage after do_compile before do_install
```

Example of a recipe

Hello world recipe

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

SRC_URI = "file://hello.c"

do_compile() {
    ${CC} ${CFLAGS} ${LDFLAGS} -o hello ${WORKDIR}/hello.c
}

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 = "http://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"
```

```
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

Free Electrons

Extending a recipe

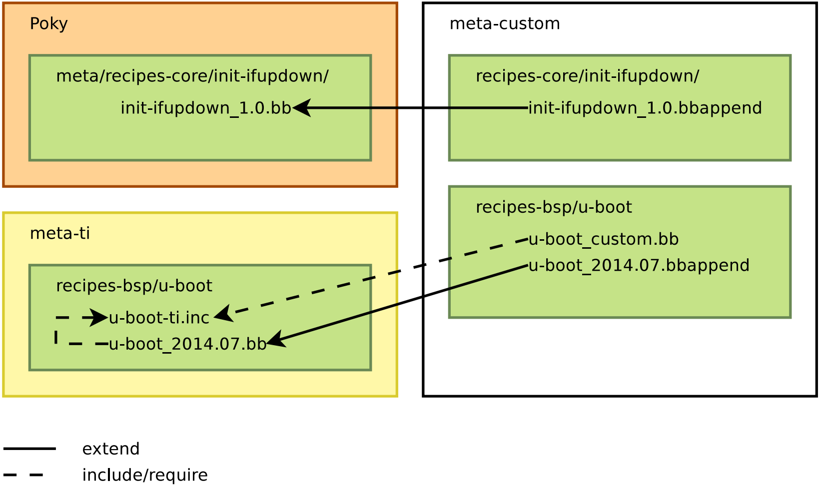
Introduction to recipe extensions

- 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.

Extend a recipe: compatibility

- When using a Yocto Project release **older than 1.5**, the Metadata revision number must explicitly be incremented in each append file.
- The revision number is stored in the `PRINC` variable.
- At the end of the recipe, you must increment it:
  - `PRINC := "${@int(PRINC) + 1"}`
- Since version 1.5, `PRINC` is automatically taken care of unless you are building on multiple machines. In that case, use the PR server, with `bitbake-prserv`

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 packages 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 packages.
- 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.
- Packages can use this common code by inheriting a class:
  - `inherits <class>`

Common classes

- Common classes can be found in `meta/classes/`
  - `base.bbclass`
  - `kernel.bbclass`
  - `autotools.bbclass`
  - `update-alternatives.bbclass`
  - `useradd.bbclass`
  - ...

The base class 1/2

- Every recipe inherits the base class automatically.
- Contains a set of basic common tasks to fetch, unpack or compile packages.
- 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 `fetchall`.

The base class 2/2

- Defines `oe_runmake`, using `EXTRA_OEMAKE` to use custom arguments.
- In Poky, `EXTRA_OEMAKE` defaults to `-e MAKEFLAGS=`.
- The `-e` option to give variables taken from the environment precedence over variables from makefiles.
  - Upstream libraries or softwares often embed their own `Makefile`.
  - Helps not using hardcoded `CC` or `CFLAGS` variables in makefiles.

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 packages 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"

inherits autotools
```

The update-alternative class

- Allows to install multiple binaries having the same functionality, avoiding conflicts by renaming the binaries.
- Four variables are used to configure the class:

  ALTERNATIVE_NAME The name of the binary.
  ALTERNATIVE_LINK The path of the resulting binary.
  ALTERNATIVE_PATH The path of the real binary.
  ALTERNATIVE_PRIORITY The alternative priority.
- The command with the highest priority will be used.

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..."

inherits 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.

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`

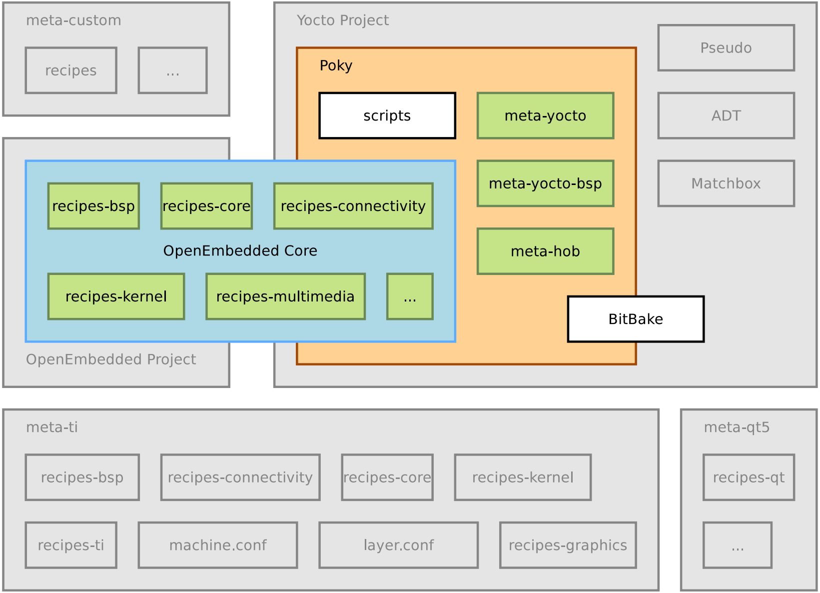
Layers

Free Electrons

Introduction to layers

Layers' principles

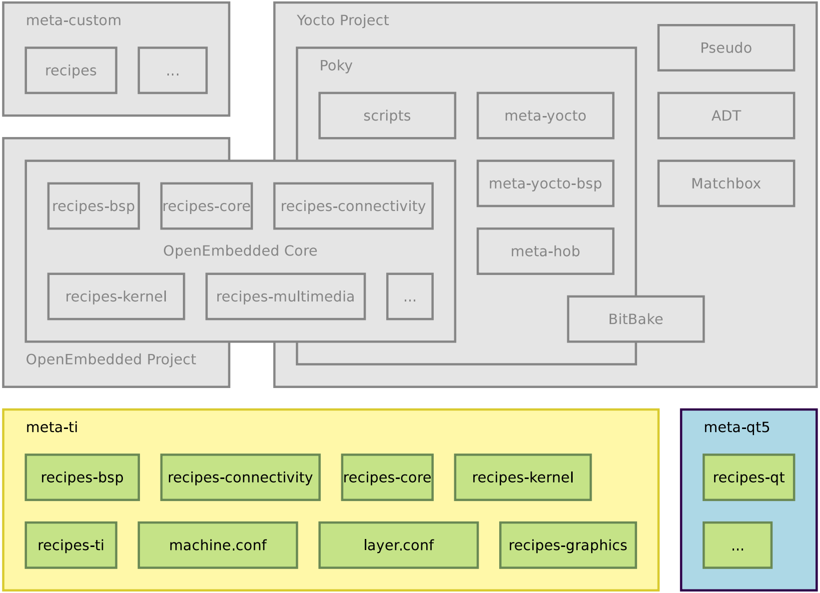
- The OpenEmbedded *build system* manipulates *metadata*.
- Layers allow to isolate and organize the metadata.
  - A layer is a collection of packages and build tasks.
- It is a good practice to begin a layer name with the prefix `meta-`.


meta-custom
recipes
...
Yocto Project
Poky
scripts
meta-yocto
meta-yocto-bsp
meta-hob
BitBake
Pseudo
ADT
Matchbox
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
...

Layers in Poky

- The Poky *reference system* is a set of basic common layers:
  - meta
  - meta-oe
  - meta-skeleton
  - meta-yocto
  - 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 packages, it is a good practice not to modify Poky. Instead you can create your own layers!

Poky



meta-custom
recipes
...
Yocto Project
Pseudo
Poky
scripts
meta-yocto
ADT
recipes-bsp
recipes-core
recipes-connectivity
meta-yocto-bsp
Matchbox
OpenEmbedded Core
meta-hob
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
...

Integrate and use a layer 1/2

- A list of existing and maintained layers can be found at `http://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 2/2

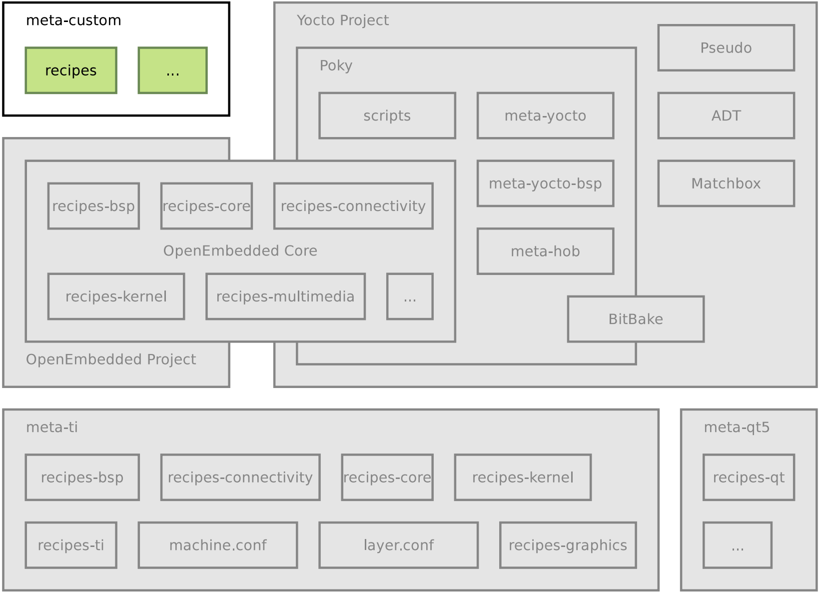
- The location where a layer is saved on the disk doesn't matter.
  - But a good practice is to save it where all others layers are stored.
- The only requirement is to let BitBake know about the new layer:
  - The list of layers BitBake uses is defined in `build/conf/bblayers.conf`
  - To include a 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.

Some useful layers

- Many SoC specific layers are available, providing support for the boards using these SoCs. Some examples: `meta-ti`, `meta-fsl-arm` and `meta-raspberrypi`.
- Other layers offer to support packages 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...

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 `yocto-layer` command helps us create new layers and ensures this is done right.
- `meta-` is automatically prepended to the layer name.
- By default `yocto-layer` creates the new layer in the current directory.
- `yocto-layer create <layer_name> -o <dest_dir>`

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.

Practical lab - Create a custom layer

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

Practical lab - Extend a recipe

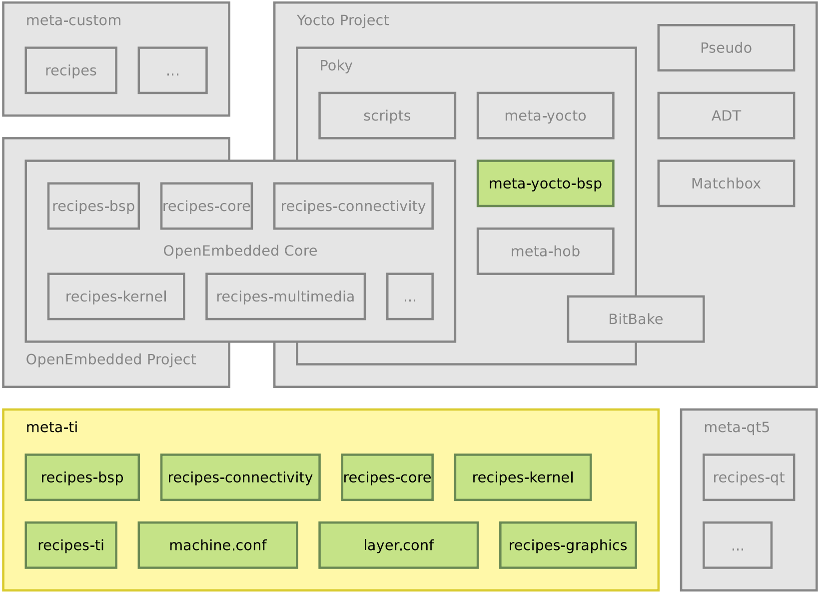
- Apply patches to an existing recipe
- Use a custom configuration file for an existing recipe
- Extend a recipe to fit your needs

BSP Layers

Free Electrons

Introduction to BSP layers in the Yocto Project

BSP layers


meta-custom
recipes
...
Yocto Project
Poky
scripts
meta-yocto
meta-yocto-bsp
meta-hob
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.

Generating a new BSP layer

Creating a new BSP 1/3

- A dedicated command is provided to create BSP layers: `yocto-bsp`.
- As for the layers, `meta-` is automatically prepended to the BSP layer's name.
- `yocto-bsp create <name> <karch>`
- `karch` stands for "kernel architecture". You can dump a list of the available ones by running: `yocto-bsp list karch`.
- `yocto-bsp create felabs arm`

Creating a new BSP 2/3

- `yocto-bsp` will prompt a few questions to help configure the kernel, bootloader and X support if needed.
- You will also need to choose compiler tuning (cortexa9, cortexa15, cortexm3, cortexm5…).
- And enable some functionalities (keyboard and mouse support).

Creating a new BSP 3/3

- A new layer is created, named meta-<bsp_name> and contains the following information:

  binary/
  : Contains bootable images or build filesystem, if needed.

  conf/layer.conf
  : The BSP layer's configuration.

  conf/machine/
  : Holds the machine configuration files. One is created by default: <bsp_name>.conf

  recipes-*
  : A few recipes are created, thanks to the user input gathered by the `yocto-bsp` command.

  README
  : The layer's documentation. This file *needs* to be updated.

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_CONSOLE 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`

IMAGE_FSTYPES Format of the root filesystem images to be created, e.g. `tar.bz2 squashfs`

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`:
  - Adds 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/mxs-base.inc

SOC_FAMILY = "mxs:mx28:cfa10036"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-cfa"
IMAGE_BOOTLOADER = "barebox"
BAREBOX_BINARY = "barebox"
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"
IMAGE_FSTYPES ?= "tar.bz2 ext3 barebox.mxsboot-sdcard sdcard"

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@free-electrons.com>

include conf/machine/include/cfa10036.inc

KERNEL_DEVICETREE += "imx28-cfa10057.dtb"

MACHINE_FEATURES += "touchscreen"
```

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

- A new way of creating images has been introduced recently: `wic`
- It 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:

```
$ wic create mkefidisk -e core-image-base
```

Formfactor

Overview

- The `yocto-bsp` command generates a `formfactor` recipe.
- `recipes-bsp/formfactor/formfactor_0.0.bbappend`
- `formfactor` is a recipe providing information about the hardware that is not described by other sources such as as the kernel.
- This configuration is defined in the recipe in: `recipes-bsp/formfactor/formfactor/<machine>/machconfig`
- Default values are defined in: `meta/recipes-bsp/formfactor/files/config`

Formfactor example

```
HAVE_TOUCHSCREEN=1
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
DISPLAY_WIDTH_PIXELS=640
DISPLAY_HEIGHT_PIXELS=480
DISPLAY_BPP=16
DISPLAY_DPI=150
DISPLAY_SUBPIXEL_ORDER=vrgb
```

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

- By default no recipe is added to customize the bootloader.
- It is possible to do so 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 if defined with: `PREFERRED_VERSION_<kernel_provider>`

Linux Yocto 1/4

- `linux-yocto` is a generic set of recipes for building mainline Linux kernel images.
- The `yocto-bsp` tool creates basic appended recipes to allow to extend the `linux-yocto` ones.
  - `meta-<bsp_name>/recipes-kernel/linux/linux-yocto_*.bbappend`
- `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"
```

Linux Yocto 3/4

- 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/1.6.1/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"`

- `.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

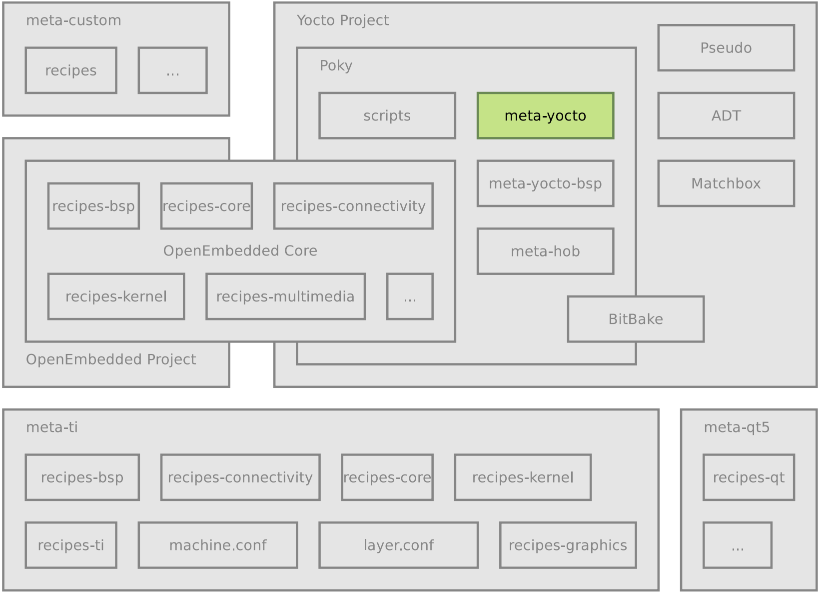
- Write a machine configuration
- Understand how the target architecture is chosen

Distro Layers

Free Electrons

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 provides 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 packages...

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 = "..."
```

Images

Free Electrons

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 debug symbols, tools and libraries.

  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"

IMAGE_PKGTYPE = "deb"

LICENSE = "MIT"
```

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
- packagegroup-core-ssh-dropbear
- packagegroup-core-ssh-openssh
- packagegroup-core-tools-debug
- packagegroup-core-tools-profile

Example

meta/recipes-core/packagegroups/packagegroup-core-nfs.bb:

```
DESCRIPTION = "NFS package groups"
LICENSE = "MIT"
PR = "r2"

inherit packagegroup

PACKAGES = "${PN}-server"

SUMMARY_${PN}-server = "NFS server"
RDEPENDS_${PN}-server = "\
    nfs-utils \
    nfs-utils-client \
    "
```

Practical lab - Create a custom image

- Write an image recipe
- Choose the packages to install

Creating and using an SDK

Free Electrons

Introduction to the 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 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 Poky SDK

- The Poky reference system is used to generate images, by building many applications and doing a lot 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 Poky 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.

Generating an SDK

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:
  - `build/tmp/deploy/sdk`
  - Example: `poky-eglibc-x86_64-meta-toolchain-i586-toolchain-1.5.3.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:
  - `build/tmp/deploy/sdk`
  - Example: `poky-eglibc-x86_64-core-image-minimal-i586-toolchain-1.5.3.sh`
- The SDK will be configured to be compatible with the specified `MACHINE`.

Use the SDK

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/1.5.3`

```
$ ./poky-eglibc-x86_64-meta-toolchain-i586-toolchain-1.5.3.sh
Enter target directory for SDK (default: /opt/poky/1.5.3):
You are about to install the SDK to "/opt/poky/1.5.3". Proceed[Y/n]?
Extracting SDK...done
Setting it up...done
SDK has been successfully set up and is ready to be used.
```

Use the SDK

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

```
$ cd /opt/poky/1.5.3
$ source ./environment-setup-i586-poky-linux
```

- 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-i586-poky-linux** Exports environment variables.
- **site-config-i586-poky-linux** Variables used during the toolchain creation
- **sysroots** SDK binaries, headers and libraries. Contains one directory for the host and one for the target.
- **version-i586-poky-linux** 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
```

Eclipse integration

The Yocto Project ADT

- Eclipse can be used as a graphical interface to the Yocto SDK.
  - To modify and develop recipes and applications.
  - To compile packages and build images.
  - To debug the system.
- This is done thanks to an Eclipse Yocto plugin.
- The SDK and the Eclipse plugin is called the Yocto Application Development Toolkit.

Installation

- The Eclipse Yocto plugin can be installed from a pre-built version available at:
  - `http://downloads.yoctoproject.org/releases/eclipse-plugin/1.6.1/kepler`
- Or it can be compiled from the official repository:
  - `git://git.yoctoproject.org/eclipse-poky-kepler`
- The full explanations to install and use the Eclipse Yocto plugin are given in the labs.

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

Practical lab - Use the Yocto SDK through Eclipse

- Install the Yocto Eclipse plugin
- Configure the plugin to work with the previously used environment
- Use Eclipse to modify recipes
- Build an image from Eclipse

Licensing

Free Electrons

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"
```

Writing recipes - going further

Free Electrons

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.

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"
```

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"
PACKAGECONFIG[bluetooth] = "--enable-bluetooth,  \
                            --disable-bluetooth, \
                            bluez4"
PACKAGECONFIG[openvpn] = "--enable-openvpn,       \
                          --with-openvpn=...,     \
                          --disable-openvpn,      \
                          ,                       \
                          openvpn"
```

Conditional features

Conditional features

- Some values can be set dynamically, thanks to a set of functions:
- `base_contains(variable, checkval, trueval, falseval, d)`: if `checkval` is found in `variable`, `trueval` is returned; otherwise `falseval` is used.
- Example:

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

Applying patches

Applying patches

- Files ending in .patch, .diff or with the apply=yes parameter will be applied during the do_patch task.
- 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.
- See meta/classes/patch.bbclass

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-yocto`.

Yocto Project Resources

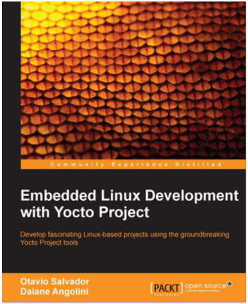
Free Electrons

Yocto Project documentation

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

Embedded Linux Development with Yocto Project, July 2014

- `https://www.packtpub.com/application-development/embedded-linux-development-yocto-project`
- 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.

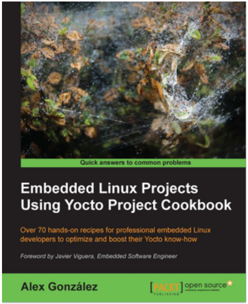


Useful Reading (2)

Embedded Linux Projects Using Yocto Project Cookbook, March 2015

- http://bit.ly/1DTvjNg
- 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: http://bit.ly/1GgVmCB



Last slides

Free Electrons

Thank you!

And may the Source be with you