Introduction to the Yocto Project / OpenEmbedded-core

Mylène Josserand
Bootlin
mylene@bootlin.com
Mylène Josserand

- Embedded Linux engineer at Bootlin since 2016
  - Embedded Linux expertise
  - Development, consulting and training around the Yocto Project
  - One of the authors of Bootlin’ Yocto Project / OpenEmbedded training materials.
- Kernel contributor: audio driver, touchscreen, RTC and more to come!
In this talk, we will:

- Understand why we should use a build system
- How the Yocto Project / OpenEmbedded core are structured
- How we can use it
- How we can update it to fit our needs
- Give some good practices to start using the Yocto Project correctly

Allows to customize many things: it is easy to do things the wrong way!

✓ When you see a ▶️, it means it is a good practice!
In this talk, we will:

- Understand why we should use a build system
In this talk, we will:

- Understand why we should use a build system
- How the Yocto Project / OpenEmbedded core are structured
In this talk, we will:

- Understand why we should use a build system
- How the Yocto Project / OpenEmbedded core are structured
- How we can use it
- Allows to customize many things: it is easy to do things the wrong way

When you see a ✓, it means it is a good practice!
In this talk, we will:

- Understand why we should use a build system
- How the Yocto Project / OpenEmbedded core are structured
- How we can use it
- How we can update it to fit our needs
In this talk, we will:

- Understand why we should use a build system
- How the Yocto Project / OpenEmbedded core are structured
- How we can use it
- How we can update it to fit our needs
- Give some good practices to start using the Yocto Project correctly
In this talk, we will:
- Understand why we should use a build system
- How the Yocto Project / OpenEmbedded core are structured
- How we can use it
- How we can update it to fit our needs
- Give some good practices to start using the Yocto Project correctly

Allows to customize many things: it is easy to do things the wrong way!
Introduction

In this talk, we will:

- Understand why we should use a build system
- How the Yocto Project / OpenEmbedded core are structured
- How we can use it
- How we can update it to fit our needs
- Give some good practices to start using the Yocto Project correctly

- Allows to customize many things: it is easy to do things the wrong way
- When you see a ✓, it means it is a **good practice**!
Why use a build system?

In the Embedded world, we have many constraints ▶ Nice to reduce the system to a minimal one + add our custom application ▶ A build system will automate the creation of the system in a reproducible way ▶ Integration means packaging applications to create a final image

Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
Why use a build system?

- In the Embedded world, we have many constraints
Why use a build system?

- In the Embedded world, we have many constraints
- Nice to reduce the system to a **minimal** one + add our **custom application**
Why use a build system?

- In the Embedded world, we have many constraints
- Nice to reduce the system to a **minimal** one + add our **custom application**
- A build system will automate the creation of the system in a **reproducible** way
Why use a build system?

- In the Embedded world, we have many constraints
- Nice to reduce the system to a **minimal** one + add our **custom application**
- A build system will automate the creation of the system in a **reproducible** way
- **Integration** means packaging applications to create a final image
System integration: several possibilities

▶ Building everything manually:

⊕ Full flexibility
⊗ Dependency hell
⊗ Lack of reproducibility

▶ Binary distribution (Debian, Ubuntu, Fedora, etc):
⊕ Easy to create and extend
⊗ Hard to customize and optimize (boot time, size)
⊗ Hard to rebuild from source

▶ Build systems (Buildroot, the Y octo Project, etc):
⊗ Not as easy as a binary distribution
⊕ Nearly full flexibility
⊕ Built from source: customization and optimization are easy
⊕ Fully reproducible
⊕ Cross-compilation
Building everything manually:

- Full flexibility
Building everything manually:

- Full flexibility
- Dependency hell

- Building everything manually:

- Full flexibility
- Dependency hell
Building everything manually:

- Full flexibility
- Dependency hell
- Lack of reproducibility
System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - ✗ Dependency hell
  - ✗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊖ Not as easy as a binary distribution
  - ⊕ Nearly full flexibility
  - ⊕ Built from source: customization and optimization are easy
  - ⊕ Fully reproducible
  - ⊖ Native-compilation
  - ⊖ Not available for all architectures
  - ⊖ Hard to rebuild from source
  - ⊖ Hard to customize and optimize (boot time, size)
System integration: several possibilities

- **Building everything manually:**
  - ☑ Full flexibility
  - ✗ Dependency hell
  - ✗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ☑ Easy to create and extend

System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - ⊗ Dependency hell
  - ⊗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊕ Easy to create and extend
  - ⊗ Hard to customize and optimize (boot time, size)
System integration: several possibilities

- **Building everything manually:**
  - ☑ Full flexibility
  - ☒ Dependency hell
  - ☒ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ☑ Easy to create and extend
  - ☒ Hard to customize and optimize (boot time, size)
  - ☒ Hard to rebuild from source
System integration: several possibilities

▶ Building everything manually:
  ⊕ Full flexibility
  ✗ Dependency hell
  ✗ Lack of reproducibility

▶ Binary distribution (Debian, Ubuntu, Fedora, etc):
  ⊕ Easy to create and extend
  ✗ Hard to customize and optimize (boot time, size)
  ✗ Hard to rebuild from source
  ✗ Native-compilation
System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - ⊗ Dependency hell
  - ⊗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊕ Easy to create and extend
  - ⊗ Hard to customize and optimize (boot time, size)
  - ⊗ Hard to rebuild from source
  - ⊗ Native-compilation
  - ⊗ Not available for all architectures
System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - ⊗ Dependency hell
  - ⊗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊕ Easy to create and extend
  - ⊗ Hard to customize and optimize (boot time, size)
  - ⊗ Hard to rebuild from source
  - ⊗ Native-compilation
  - ⊗ Not available for all architectures

- **Build systems** (Buildroot, the Yocto Project, etc):
System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - ⊗ Dependency hell
  - ⊗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊕ Easy to create and extend
  - ⊗ Hard to customize and optimize (boot time, size)
  - ⊗ Hard to rebuild from source
  - ⊗ Native-compilation
  - ⊗ Not available for all architectures

- **Build systems** (Buildroot, the Yocto Project, etc):
  - ⊗ Not as easy as a binary distribution
System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - × Dependency hell
  - × Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊕ Easy to create and extend
  - × Hard to customize and optimize (boot time, size)
  - × Hard to rebuild from source
  - × Native-compilation
  - × Not available for all architectures

- **Build systems** (Buildroot, the Yocto Project, etc):
  - × Not as easy as a binary distribution
  - ⊕ Nearly full flexibility
System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - ⊗ Dependency hell
  - ⊗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊕ Easy to create and extend
  - ⊗ Hard to customize and optimize (boot time, size)
  - ⊗ Hard to rebuild from source
  - ⊗ Native-compilation
  - ⊗ Not available for all architectures

- **Build systems** (Buildroot, the Yocto Project, etc):
  - ⊗ Not as easy as a binary distribution
  - ⊕ Nearly full flexibility
  - ⊕ Built from source: customization and optimization are easy
System integration: several possibilities

- **Building everything manually:**
  - ⊕ Full flexibility
  - ⊗ Dependency hell
  - ⊗ Lack of reproducibility

- **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ⊕ Easy to create and extend
  - ⊗ Hard to customize and optimize (boot time, size)
  - ⊗ Hard to rebuild from source
  - ⊗ Native-compilation
  - ⊗ Not available for all architectures

- **Build systems** (Buildroot, the Yocto Project, etc):
  - ⊗ Not as easy as a binary distribution
  - ⊕ Nearly full flexibility
  - ⊕ Built from source: customization and optimization are easy
  - ⊕ Fully reproducible
System integration: several possibilities

▶ **Building everything manually:**
  - ✗ Full flexibility
  - ✗ Dependency hell
  - ✗ Lack of reproducibility

▶ **Binary distribution** (Debian, Ubuntu, Fedora, etc):
  - ✗ Easy to create and extend
  - ✗ Hard to customize and optimize (boot time, size)
  - ✗ Hard to rebuild from source
  - ✗ Native-compilation
  - ✗ Not available for all architectures

▶ **Build systems** (Buildroot, the Yocto Project, etc):
  - ✗ Not as easy as a binary distribution
  - ✗ Nearly full flexibility
  - ✗ Built from source: customization and optimization are easy
  - ✗ Fully reproducible
  - ✗ Cross-compilation
Representation in the Yocto Project

- Common tasks defined in OpenEmbedded core
- Many recipes available for many applications: organized in layers
- Allow to build custom embedded Linux-based systems

This is the aim of the Yocto Project
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine

▶ Common **tasks** defined in **OpenEmbedded core**
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine

▶ Common **tasks** defined in **OpenEmbedded core**
▶ Many recipes availables for many applications: organized in **layers**
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine

- Common tasks defined in **OpenEmbedded core**
- Many recipes availables for many applications: organized in **layers**
- Allow to build custom embedded Linux-based systems
Representation in the Yocto Project

- Download the source
- Configure the build
- Satisfy the dependencies when needed
- Compile the application using autotools, CMake, make, ...
- Install the binary on your machine

- Common tasks defined in **OpenEmbedded core**
- Many recipes availables for many applications: organized in **layers**
- Allow to build custom embedded Linux-based systems

⇒ This is the aim of the Yocto Project
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of base layer with recipes and classes
- It is the core of all the magic
- It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU
- Reference distribution of the Yocto Project
- Contains everything you need to start a project:
  - OpenEmbedded-core
  - Bitbake
  - Additional layers
- Also contains some useful tools to ease recipes and layers' creation

- Kernel, drivers and embedded Linux
- Development, consulting, training and support - https://bootlin.com
Co-maintained by the Yocto Project and OpenEmbedded Project
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of **base layer** with recipes and classes

It is the core of all the magic
It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU

Reference distribution of the Yocto Project
Contains everything you need to start a project:
- OpenEmbedded-core
- Bitbake
- Additional layers

Also contains some useful tools to ease recipes and layers' creation
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of **base layer** with recipes and classes
- It is the **core** of all the magic
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of base layer with recipes and classes
- It is the core of all the magic
- It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of **base layer** with recipes and classes
- It is the **core** of all the magic
- It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of **base layer** with recipes and classes
- It is the **core** of all the magic
- It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU

- Reference distribution of the Yocto Project
Co-maintained by the Yocto Project and Open Embedded Project

Set of **base layer** with recipes and classes

It is the **core** of all the magic

It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU

Reference distribution of the Yocto Project

Contains everything you need to start a project:
Co-maintained by the Yocto Project and OpenEmbedded Project

Set of **base layer** with recipes and classes

It is the **core** of all the magic

It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU

Reference distribution of the Yocto Project

Contains everything you need to start a project:
  - OpenEmbedded-core
Co-maintained by the Yocto Project and OpenEmbedded Project

Set of base layer with recipes and classes

It is the core of all the magic

It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU

Reference distribution of the Yocto Project

Contains everything you need to start a project:

- OpenEmbedded-core
- Bitbake
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of **base layer** with recipes and classes
- It is the **core** of all the magic
- It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU

- Reference distribution of the Yocto Project
- Contains everything you need to start a project:
  - OpenEmbedded-core
  - Bitbake
  - Additional layers
OpenEmbedded core & Poky

- Co-maintained by the Yocto Project and OpenEmbedded Project
- Set of **base layer** with recipes and classes
- It is the **core** of all the magic
- It supports the ARM, MIPS (32 and 64 bits), PowerPC and x86 (32 and 64 bits) architectures + QEMU

- Reference distribution of the Yocto Project
- Contains everything you need to start a project:
  - OpenEmbedded-core
  - Bitbake
  - Additional layers
- Also contains some useful tools to ease recipes and layers' creation
Workflow - General

User/Developer actions

Download → Configure → Compile
Workflow - General

User/Developer actions

- Download
- Configure
- Compile
Workflow - Users/Developers actions

User/Developer actions

Download → Configure → Compile

Developer's work

Create layer:
- Create recipe
  - Extend recipe
- Create image
- Create machine
- Create distro
Workflow - 1. Download

- Find which version you want to use:
  - Support level: Development, Stable, Community
  - A codename corresponds to a Poky and Bitbake versions
    - Pyro = Yocto Project v2.3
    - Poky v17.0 & Bitbake v1.34
- How to download:
  - `git clone -b pyro git://git.yoctoproject.org/poky.git`

[Link to Bootlin website: https://bootlin.com]
Find which **version** you want to use:

**Workflow - 1. Download**

- **Find which version you want to use:**

  ![Image](https://wiki.yoctoproject.org/wiki/Releases)

  **Support level:** Development, Stable, Community

  A codename corresponds to a Poky and Bitbake versions

  - **Pyro** = Yocto Project v2.3
    - Poky v17.0 & Bitbake v1.34

  **How to download:**

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

**Release Activity**

<table>
<thead>
<tr>
<th>Codename</th>
<th>Yocto Project Version</th>
<th>Release Date</th>
<th>Current Version</th>
<th>Support Level</th>
<th>Poky Version</th>
<th>BitBake branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocko</td>
<td>2.4</td>
<td>Fall 2017</td>
<td>Development</td>
<td>18.0</td>
<td>1.36</td>
<td></td>
</tr>
<tr>
<td>Pyro</td>
<td>2.3</td>
<td>May 2017</td>
<td>2.3.1</td>
<td>Stable</td>
<td>17.0</td>
<td>1.34</td>
</tr>
<tr>
<td>Morty</td>
<td>2.2</td>
<td>Nov 2016</td>
<td>2.2.1</td>
<td>Stable</td>
<td>16.0</td>
<td>1.32</td>
</tr>
<tr>
<td>Kroeth</td>
<td>2.1</td>
<td>Apr 2016</td>
<td>2.1.3</td>
<td>Community</td>
<td>15.0</td>
<td>1.30</td>
</tr>
</tbody>
</table>

**Figure:** [https://wiki.yoctoproject.org/wiki/Releases](https://wiki.yoctoproject.org/wiki/Releases)
Find which **version** you want to use:

<table>
<thead>
<tr>
<th>Codename</th>
<th>Yocto Project Version</th>
<th>Release Date</th>
<th>Current Version</th>
<th>Support Level</th>
<th>Poky Version</th>
<th>BitBake branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocko</td>
<td>2.4</td>
<td>Fall 2017</td>
<td></td>
<td>Development</td>
<td>18.0</td>
<td>1.36</td>
</tr>
<tr>
<td>Pyro</td>
<td>2.3</td>
<td>May 2017</td>
<td>2.3.1</td>
<td>Stable</td>
<td>17.0</td>
<td>1.34</td>
</tr>
<tr>
<td>Morty</td>
<td>2.2</td>
<td>Nov 2016</td>
<td>2.2.1</td>
<td>Stable</td>
<td>16.0</td>
<td>1.32</td>
</tr>
<tr>
<td>Krogeth</td>
<td>2.1</td>
<td>Apr 2016</td>
<td>2.1.3</td>
<td>Community</td>
<td>15.0</td>
<td>1.30</td>
</tr>
</tbody>
</table>

**Figure:** https://wiki.yoctoproject.org/wiki/Releases

Support level: Development, Stable, Community
Find which **version** you want to use:

<table>
<thead>
<tr>
<th>Codename</th>
<th>Yocto Project Version</th>
<th>Release Date</th>
<th>Current Version</th>
<th>Support Level</th>
<th>Poky Version</th>
<th>Bitbake branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocko</td>
<td>2.4</td>
<td>Fall 2017</td>
<td></td>
<td>Development</td>
<td>18.0</td>
<td>1.36</td>
</tr>
<tr>
<td>Pyro</td>
<td>2.3</td>
<td>May 2017</td>
<td>2.3.1</td>
<td>Stable</td>
<td>17.0</td>
<td>1.34</td>
</tr>
<tr>
<td>Morty</td>
<td>2.2</td>
<td>Nov 2016</td>
<td>2.2.1</td>
<td>Stable</td>
<td>16.0</td>
<td>1.32</td>
</tr>
<tr>
<td>Krogoth</td>
<td>2.1</td>
<td>Apr 2016</td>
<td>2.1.3</td>
<td>Community</td>
<td>15.0</td>
<td>1.30</td>
</tr>
</tbody>
</table>

**Figure:** https://wiki.yoctoproject.org/wiki/Releases

- Support level: Development, Stable, Community
- A **codename** corresponds to a Poky and Bitbake versions
  
  **Pyro** = Yocto Project v2.3 → Poky v17.0 & Bitbake v1.34
Find which **version** you want to use:

<table>
<thead>
<tr>
<th>Codename</th>
<th>Yocto Project Version</th>
<th>Release Date</th>
<th>Current Version</th>
<th>Support Level</th>
<th>Poky Version</th>
<th>BitBake branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocky</td>
<td>2.4</td>
<td>Fall 2017</td>
<td>Development</td>
<td>18.0</td>
<td>1.36</td>
<td></td>
</tr>
<tr>
<td>Pyro</td>
<td>2.3</td>
<td>May 2017</td>
<td>2.3.1</td>
<td>Stable</td>
<td>17.0</td>
<td>1.34</td>
</tr>
<tr>
<td>Morty</td>
<td>2.2</td>
<td>Nov 2016</td>
<td>2.2.1</td>
<td>Stable</td>
<td>16.0</td>
<td>1.32</td>
</tr>
<tr>
<td>Krogoth</td>
<td>2.1</td>
<td>Apr 2016</td>
<td>2.1.3</td>
<td>Community</td>
<td>15.0</td>
<td>1.30</td>
</tr>
</tbody>
</table>

Support level: Development, Stable, Community

A **codename** corresponds to a Poky and Bitbake versions

Pyro = Yocto Project v2.3 → Poky v17.0 & Bitbake v1.34

How to download:

**Figure:** https://wiki.yoctoproject.org/wiki/Releases
Find which **version** you want to use:

<table>
<thead>
<tr>
<th>Codename</th>
<th>Yocto Project Version</th>
<th>Release Date</th>
<th>Current Version</th>
<th>Support Level</th>
<th>Poky Version</th>
<th>BitBake branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocky</td>
<td>2.4</td>
<td>Fall 2017</td>
<td>Development</td>
<td>18.0</td>
<td>1.36</td>
<td></td>
</tr>
<tr>
<td>Pyro</td>
<td>2.3</td>
<td>May 2017</td>
<td>Stable</td>
<td>17.0</td>
<td>1.34</td>
<td></td>
</tr>
<tr>
<td>Morty</td>
<td>2.2</td>
<td>Nov 2016</td>
<td>Stable</td>
<td>16.0</td>
<td>1.32</td>
<td></td>
</tr>
<tr>
<td>Krogoth</td>
<td>2.1</td>
<td>Apr 2016</td>
<td>Community</td>
<td>15.0</td>
<td>1.30</td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** [https://wiki.yoctoproject.org/wiki/Releases](https://wiki.yoctoproject.org/wiki/Releases)

- Support level: Development, Stable, Community
- A **codename** corresponds to a Poky and Bitbake versions
  - Pyro = Yocto Project v2.3 → Poky v17.0 & Bitbake v1.34
- How to download:

  ```bash
  git clone -b pyro git://git.yoctoproject.org/poky.git
  ```
Layers are sets of recipes, matching a common purpose.
To simplify things, they are just folders.
**Layers** are sets of recipes, matching a common purpose. To simplify things, they are just *folders*.

- **Look at existing layers**

  ![Layer Index](http://layers.openembedded.org/layerindex/)

**Figure:** [http://layers.openembedded.org/layerindex/](http://layers.openembedded.org/layerindex/)
Layers are sets of recipes, matching a common purpose.
To simplify things, they are just folders.

Look at existing layers

Download all other layers on same branch than Poky: Pyro

Figure: http://layers.openembedded.org/layerindex/
Layers are sets of recipes, matching a common purpose. To simplify things, they are just folders.

Look at existing layers:

- Download all other layers on same branch than Poky: Pyro
- Use existing layers before creating a new one ⇒ saves you time
Layers are sets of recipes, matching a common purpose. To simplify things, they are just folders.

Look at existing layers.

Download all other layers on same branch than Poky: Pyro.

✓ Use existing layers before creating a new one \(\Rightarrow\) saves you time.
✓ DO NOT EDIT POKY/UPSTREAM LAYERS \(\Rightarrow\) complicates updates.
Workflow - 2. Configure the build

A script with all variables needed by Bitbake must be sourced:

```bash
source oe-init-build-env
```

Will move you in a build folder

Now, can run any commands

All the local configurations are in the conf folder:

```
build/
|-- conf
  |-- bblayers.conf
  |-- local.conf
```

Edit your `bblayers.conf` with possible additional layers:

```bash
BBLAYERS ?= "
/home/mylene/yocto/poky/meta
/home/mylene/yocto/poky/meta-poky
/home/mylene/yocto/poky/meta-yocto-bsp
/home/mylene/yocto/meta-freescale
/home/mylene/yocto/meta-qt5"
```

 Kernel, drivers and embedded Linux - Development, consulting, training and support -  [https://bootlin.com](https://bootlin.com)
Workflow - 2. Configure the build

- A script with all variables needed by Bitbake must be **sourced**:

```
source oe-init-build-env
```

- Will move you in a build folder
- Now, can run any commands
- All the local configurations are in the `conf` folder

```
build/
|-- conf
   |-- bblayers.conf
   |-- local.conf
```

- Edit your `bblayers.conf` with possible additional layers:

```
BBLAYERS ?= "
/home/mylene/yocto/poky/meta
/home/mylene/yocto/poky/meta-poky
/home/mylene/yocto/poky/meta-yocto-bsp
/home/mylene/yocto/meta-freescale
/home/mylene/yocto/meta-qt5"
```

[13/1]
A script with all variables needed by Bitbake must be **sourced**:

```bash
source oe-init-build-env
```
Workflow - 2. Configure the build

- A script with all variables needed by Bitbake must be **sourced**:

  ```bash
  source oe-init-build-env
  ```

- Will move you in a **build** folder
Workflow - 2. Configure the build

- A script with all variables needed by Bitbake must be **sourced**:

  ```bash
  source oe-init-build-env
  ```

- Will move you in a **build** folder
- Now, can run any commands
Workflow - 2. Configure the build

- A script with all variables needed by Bitbake must be **sourced**:

```bash
source oe-init-build-env
```

- Will move you in a **build** folder
- Now, can run any commands
- All the local configurations are in the **conf** folder
Workflow - 2. Configure the build

- A script with all variables needed by Bitbake must be **sourced**:

```
source oe-init-build-env
```

- Will move you in a **build** folder
- Now, can run any commands
- All the local configurations are in the **conf** folder

```
build/
|-- conf
| `-- bblayers.conf
| `-- local.conf
```
Workflow - 2. Configure the build

- A script with all variables needed by Bitbake must be **sourced**:

```
source oe-init-build-env
```

- Will move you in a **build** folder
- Now, can run any commands
- All the local configurations are in the **conf** folder

```
build/
|-- conf
 | |-- bblayers.conf
 | |-- local.conf
```

- Edit your **bblayers.conf** with possible additional layers:
Workflow - 2. Configure the build

▶ A script with all variables needed by Bitbake must be sourced:

```
source oe-init-build-env
```

▶ Will move you in a build folder
▶ Now, can run any commands
▶ All the local configurations are in the conf folder

```
build/
    |-- conf
        |-- bblayers.conf
        |-- local.conf
```

▶ Edit your bblayers.conf with possible additional layers:

```
BBLAYERS ?= " \\n/home/mylene/yocto/poky/meta \\
/home/mylene/yocto/poky/meta-poky \\
/home/mylene/yocto/poky/meta-yocto-bsp \\
/home/mylene/yocto/meta-freescale \\
/home/mylene/yocto/meta-qt5 \\
" 
```
Workflow - 2. Configure the build

- Edit `local.conf` with your MACHINE and your DISTRO

MACHINE: Describes your hardware. Can find it under specific layers: BSP layers.
- conf/machine/
  - poky: beaglebone, x86, x86-64
  - meta-ti: beagleboard, pandaboard, ...
  - meta-fsl-arm: imx23, imx28, imx6, imx7, ...
  - meta-atmel: at91*, sama5d*, ...

DISTRO: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc. + some specific variables.
- conf/distro/
  - poky: poky, poky-tiny, ...
  - meta-angstrom: angstrom

Noticed that `local.conf` is only for the local workstation.

Avoid changes directly in `local.conf` (or only for test purposes, except for some variables such as MACHINE and DISTRO).

https://bootlin.com
Edit `local.conf` with your **MACHINE** and your **DISTRO**

**MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at `conf/machine/` folders
Workflow - 2. Configure the build

▶ Edit *local.conf* with your **MACHINE** and your **DISTRO**

▶ **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at *conf/machine/* folders

▶ poky: beaglebone, x86, x86-64
Workflow - 2. Configure the build

- Edit *local.conf* with your **MACHINE** and your **DISTRO**
  - **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at *conf/machine/* folders
    - poky: beaglebone, x86, x86-64
    - meta-ti: beagleboard, pandaboard, ...
  - **DISTRO**: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc + some specific variables. Look at *conf/distro/* folders
    - poky: poky, poky-tiny, ...
    - meta-angstrom: angstrom
  - Noticed that *local.conf* is only for the local workstation.
  - Avoid changes directly in *local.conf* (or only for test purposes, except for some variables such as **MACHINE** and **DISTRO**).
Edit `local.conf` with your **MACHINE** and your **DISTRO**

- **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at `conf/machine/` folders
  - poky: beaglebone, x86, x86-64
  - meta-ti: beagleboard, pandaboard, ...
  - meta-fsl-arm: imx23, imx28, imx6, imx7, ...

...
Workflow - 2. Configure the build

- Edit `local.conf` with your **MACHINE** and your **DISTRO**
  - **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at `conf/machine/` folders
    - poky: beaglebone, x86, x86-64
    - meta-ti: beagleboard, pandaboard, ...
    - meta-fsl-arm: imx23, imx28, imx6, imx7, ...
    - meta-atmel: at91*, sama5d*, ...

- **DISTRO**: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc + some specific variables. Look at `conf/distro/` folders
  - poky: poky, poky-tiny, ...
  - meta-angstrom: angstrom

- Noticed that `local.conf` only for the local workstation.
  - Avoid changes directly in `local.conf` (or only for test purposes, except for some variables such as `MACHINE` and `DISTRO`).

---

Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
Workflow - 2. Configure the build

- Edit `local.conf` with your **MACHINE** and your **DISTRO**
  - **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at `conf/machine/` folders
    - poky: beaglebone, x86, x86-64
    - meta-ti: beagleboard, pandaboard, ...
    - meta-fsl-arm: imx23, imx28, imx6, imx7, ...
    - meta-atmel: at91*, sama5d*, ...
  - **DISTRO**: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc + some specific variables. Look at `conf/distro/` folders

- Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
Workflow - 2. Configure the build

- Edit local.conf with your MACHINE and your DISTRO

  - **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at conf/machine/ folders
    - poky: beaglebone, x86, x86-64
    - meta-ti: beagleboard, pandaboard, ...
    - meta-fsl-arm: imx23, imx28, imx6, imx7, ...
    - meta-atmel: at91*, sama5d*, ...

  - **DISTRO**: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc + some specific variables Look at conf/distro/ folders
    - poky: poky, poky-tiny, ...

- Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
Workflow - 2. Configure the build

- Edit *local.conf* with your **MACHINE** and your **DISTRO**
  - **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at *conf/machine/* folders
    - poky: beaglebone, x86, x86-64
    - meta-ti: beagleboard, pandaboard, ...
    - meta-fsl-arm: imx23, imx28, imx6, imx7, ...
    - meta-atmel: at91*, sama5d*, ...
  - **DISTRO**: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc + some specific variables. Look at *conf/distro/* folders
    - poky: poky, poky-tiny, ...
    - meta-angstrom: angstrom

✓ Noticed that *local.conf* only for the local workstation.
Workflow - 2. Configure the build

▶ Edit local.conf with your MACHINE and your DISTRO

▶ MACHINE: Describes your hardware. Can find it under specific layers: BSP layers. Look at conf/machine/ folders
  ▶ poky: beaglebone, x86, x86-64
  ▶ meta-ti: beagleboard, pandaboard, ...
  ▶ meta-fsl-arm: imx23, imx28, imx6, imx7, ...
  ▶ meta-atmel: at91*, sama5d*, ...

▶ DISTRO: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc + some specific variables Look at conf/distro/ folders
  ▶ poky: poky, poky-tiny, ...
  ▶ meta-angstrom: angstrom

▶ Noticed that local.conf ⇒ only for the local workstation.
Workflow - 2. Configure the build

- Edit `local.conf` with your **MACHINE** and your **DISTRO**

  - **MACHINE**: Describes your hardware. Can find it under specific layers: BSP layers. Look at `conf/machine/` folders
  - poky: beaglebone, x86, x86-64
  - meta-ti: beagleboard, pandaboard, ...
  - meta-fsl-arm: imx23, imx28, imx6, imx7, ...
  - meta-atmel: at91*, sama5d*, ...

  - **DISTRO**: Represents the top-level configuration that will apply to every build. It will include tools needed to use your hardware: compiler, libC, etc + some specific variables Look at `conf/distro/` folders
  - poky: poky, poky-tiny, ...
  - meta-angstrom: angstrom

- Noticed that local.conf ⇒ only for the local workstation.

✓ Avoid changes directly in `local.conf` (or only for test purposes, except for some variables such as **MACHINE** and **DISTRO**).
Workflow - 3. Build an image

- What is an IMAGE?
  
  Represents your root filesystem: all your applications, libraries, configuration files, ...
  
  Will find it under `recipes-*/images/`

- Common images already exist in Poky: `core-image-minimal`, `core-image-base`, `core-image-x11`, ...

- Build an existing image:
  ```bash
  bitbake core-image-minimal
  ```
What is an **IMAGE**?

⇒ Represents your root filesystem: all your applications, libraries, configuration files, ...
   Will find it under `recipes-**/images/`
What is an IMAGE?
⇒ Represents your root filesystem: all your applications, libraries, configuration files, ... Will find it under recipes-*/images/

Common images already exist in Poky: core-image-minimal, core-image-base, core-image-x11, ...
What is an IMAGE?

⇒ Represents your root filesystem: all your applications, libraries, configuration files, ... Will find it under `recipes-*/images/`

⇒ Common images already exist in Poky: `core-image-minimal`, `core-image-base`, `core-image-x11`, ...

⇒ Build an existing image:
What is an **IMAGE**?

⇒ Represents your root filesystem: all your applications, libraries, configuration files, ... Will find it under `recipes-*/images/`

► Common images already exist in Poky: `core-image-minimal`, `core-image-base`, `core-image-x11`, ...

► Build an existing image:

```
bitbake core-image-minimal
```
Machine: It represents your hardware

conf/machine/
MACHINE/DISTRO/IMAGE: a little reminder

- **Machine**: It represents your hardware
  
  `conf/machine/`
MACHINE/DISTRO/IMAGE: a little reminder

- **Machine**: It represents your hardware
  
  `conf/machine/`

- **Distro**: Represents the top-level configuration that will apply on every build
  
  `conf/distro/`
MACHINE/DISTRO/IMAGE: a little reminder

- **Machine**: It represents your hardware 
  `conf/machine/`

- **Distro**: Represents the top-level configuration that will apply on every build 
  `conf/distro/`
MACHINE/DISTRO/IMAGE: a little reminder

- **Machine**: It represents your hardware 
  `conf/machine/`

- **Distro**: Represents the top-level configuration that will apply on every build 
  `conf/distro/`

- **Image**: It represents your root filesystem itself: all your applications, libraries, configuration’s files, etc 
  `recipes-core/images`
MACHINE/DISTRO/IMAGE: a little reminder

- **Machine**: It represents your hardware
  `conf/machine/`

- **Distro**: Represents the top-level configuration that will apply on every build
  `conf/distro/`

- **Image**: It represents your root filesystem itself: all your applications, libraries, configuration’s files, etc
  `recipes-core/images`
Workflow - Developer

User/Developer actions

Download ➔ Configure ➔ Compile

Developer's work

Create layer ➔ Create recipe ➔ Create image ➔ Create machine

Create distro ➔ Extend recipe
Workflow - 4. Create a layer

You may have custom hardware, need to update recipes from upstream layers, integrate your own application, etc. Already said before: DO NOT EDIT Poky/Upstream Layers. To be able to do that, we will create our own layer that will host all our modifications/applications. Poky provides a tool to create layers:

```
yocto-layer create <layer_name> -o <dest_dir>
```

✓ The layer's name must be `meta-*` (done automatically using `yocto-layer` tool)

✓ Avoid uppercase and funny/long names

✓ If you have different projects with common parts, try to create two layers. Can re-use some parts.
Workflow - 4. Create a layer

- You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc

Poky provides a tool to create layers:

```
yocto-layer create <layer_name> -o <dest_dir>
```

- The layer's name must be `meta-*` (done automatically using `yocto-layer` tool)
- Avoid uppercase and funny/long names
- If you have different projects with common parts, try to create two layers

Can re-use some parts

- Kernel, drivers and embedded Linux
- Development, consulting, training and support - [https://bootlin.com](https://bootlin.com)
You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc

Already said before: **DO NOT EDIT POKY/UPSTREAM LAYERS**
You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc

Already said before: **DO NOT EDIT POKY/UPSTREAM LAYERS**

To be able to do that, we will create our **own layer** that will host all our modifications/applications
You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc.

Already said before: **DO NOT EDIT POKY/UPSTREAM LAYERS**

To be able to do that, we will create our **own layer** that will host all our modifications/applications.

Poky provides a tool to create layers:
Workflow - 4. Create a layer

▶ You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc

▶ Already said before: **DO NOT EDIT POKY/UPSTREAM LAYERS**

▶ To be able to do that, we will create our **own layer** that will host all our modifications/applications

▶ Poky provides a tool to create layers:

```bash
yocto-layer create <layer_name> -o <dest_dir>
```
You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc

- Already said before: **DO NOT EDIT POKY/UPSTREAM LAYERS**

- To be able to do that, we will create our **own layer** that will host all our modifications/applications

- Poky provides a tool to create layers:

  ```
  yocto-layer create <layer_name> -o <dest_dir>
  ```

- The layer’s name must be **meta-** (done automatically using `yocto-layer` tool)
You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc

Already said before: **DO NOT EDIT POKY/UPSTREAM LAYERS**

To be able to do that, we will create our **own layer** that will host all our modifications/applications

Poky provides a tool to create layers:

```
yocto-layer create <layer_name> -o <dest_dir>
```

- ✓ The layer’s name must be `meta-*` (done automatically using `yocto-layer` tool)
- ✓ Avoid uppercase and funny/long names
Workflow - 4. Create a layer

▶ You may have **custom hardware**, need to update recipes from upstream layers, integrate your **own application**, etc

▶ Already said before: **DO NOT EDIT POKY/UPSTREAM LAYERS**

▶ To be able to do that, we will create our **own layer** that will host all our modifications/applications

▶ Poky provides a tool to create layers:

```
yocto-layer create <layer_name> -o <dest_dir>
```

✓ The layer’s name must be `meta-*` (done automatically using `yocto-layer` tool)

✓ Avoid uppercase and funny/long names

✓ If you have different projects with common parts, try to create two layers

⇒ Can re-use some parts
Workflow - 5. Create a recipe

- A recipe is a file describing tasks for an application to:
  - retrieve its sources
  - configure it
  - compile it
  - install it
- It handles all the dependencies for you.
- Many common tasks are already defined by OpenEmbedded-core
- Organized in folders with the same purpose (recipes-core, recipes-bsp, recipes-kernel, recipes-devtool, recipes-support, ...)

[https://bootlin.com](https://bootlin.com)
A recipe is a file describing **tasks** for an application to:
A recipe is a file describing **tasks** for an application to:
- retrieve its sources
Workflow - 5. Create a recipe

A recipe is a file describing **tasks** for an application to:
- retrieve its sources
- configure it

It handles all the dependencies for you.

Many common tasks are already defined by OpenEmbedded-core

Organized in folders with the same purpose (recipes-core, recipes-bsp, recipes-kernel, recipes-devtool, recipes-support, ...)

https://bootlin.com
A recipe is a file describing **tasks** for an application to:
- retrieve its sources
- configure it
- compile it
A recipe is a file describing **tasks** for an application to:
- retrieve its sources
- configure it
- compile it
- install it
A recipe is a file describing **tasks** for an application to:

- retrieve its sources
- configure it
- compile it
- install it

It handles all the **dependencies** for you.
A recipe is a file describing **tasks** for an application to:
- retrieve its sources
- configure it
- compile it
- install it

- It handles all the **dependencies** for you.
- Many common **tasks** are already defined by OpenEmbedded-core
A recipe is a file describing **tasks** for an application to:
- retrieve its sources
- configure it
- compile it
- install it

It handles all the **dependencies** for you.

Many common **tasks** are already defined by OpenEmbedded-core

Organized in folders with the same purpose (*recipes-core*, *recipes-bsp*, *recipes-kernel*, *recipes-devtool*, *recipes-support*, ...) and a sub-folder with the application’s name.
To create a recipe, you have to create a `.bb file`. It is the format that `bitbake` understands.
To create a recipe, you have to create a `.bb file`. It is the format that `bitbake` understands.

The format of a recipe file name is `<application-name>_<version>.bb`
To create a recipe, you have to create a .bb file. It is the format that bitbake understands.

The format of a recipe file name is `<application-name>_<version>.bb`

A recipe can be divided in three parts:
To create a recipe, you have to create a `.bb` file. It is the format that `bitbake` understands.

The format of a recipe file name is `<application-name>_<version>.bb`

A recipe can be divided in three parts:

- The header: what/who. Description of the application.
To create a recipe, you have to create a `.bb` file. It is the format that `bitbake` understands.

The format of a recipe file name is `<application-name>_<version>.bb`

A recipe can be divided in three parts:
- The header: what/who. Description of the application
- The sources: where. Can be tarballs, remote repository, ...
To create a recipe, you have to create a `.bb file`. It is the format that `bitbake` understands.

The format of a recipe file name is `<application-name>_<version>.bb`.

A recipe can be divided in three parts:

- The header: what/who. Description of the application
- The sources: where. Can be tarballs, remote repository, ...
- The tasks: how. How to proceed with the application’s sources
To create a recipe, you have to create a `.bb` file. It is the format that `bitbake` understands.

The format of a recipe file name is `<application-name>_<version>.bb`.

A recipe can be divided in three parts:
- The header: what/who. Description of the application
- The sources: where. Can be tarballs, remote repository, ...
- The tasks: how. How to proceed with the application’s sources

Classes are available for tasks commonly used: kernel, CMake, autotools, ...
SUMMARY = "nmon performance monitor"
HOMEPAGE = "http://nmon.sf.net"
SECTION = "console/utils"
LICENSE = "GPLv3"
LIC_FILES_CHKSUM = "file://${WORKDIR}/Documentation.txt;md5=dbb13658cf55d687c4f2ff771a696d4a"
DEPENDS = "ncurses"

SRC_URI = "${SOURCEFORGE_MIRROR}/nmon/lmon13g.c;name=lmon \
${SOURCEFORGE_MIRROR}/nmon/Documentation.txt;name=doc \
"
SRC_URI[lmon.md5sum] = "b1b8e6c0123ad232394991f2d4f40494"  
SRC_URI[lmon.sha256sum] = "456ab2a342b31d1a352d0d940af5962fa65a12ae8757ff73e6e73210832ae8b5"  
SRC_URI[doc.md5sum] = "dbb13658cf55d687c4f2ff771a696d4a"  
SRC_URI[doc.sha256sum] = "1f7f83afe62a7210be5e83cd24157adb854c14599efe0b377a7ecca933869278"

CFLAGS += "-D JFS -D GETUSER -Wall -DLARGEMEM"
LDFLAGS += "-ltinfo -lncursesw"

do_compile() {
  ${CC} ${CFLAGS} ${LDFLAGS} ${WORKDIR}/lmon13g.c -o nmon
}

do_install() {
  install -d ${D}${bindir}
  install -m 0755 nmon ${D}${bindir}
}
Workflow - 5. Create a recipe

recipes-support/nmon/nmon_13g.bb

| SUMMARY | = "nmon performance monitor" |
| HOMEPAGE | = "http://nmon.sf.net" |
| SECTION | = "console/utils" |
| LICENSE | = "GPLv3" |
| LIC_FILES_CHKSUM | = "file://${WORKDIR}/Documentation.txt;md5=dbb13658cf55d687c4f2ff771a696d4a" |
| DEPENDS | = "ncurses" |

| SRC_URI | = "${SOURCEFORGE_MIRROR}/nmon/lmon13g.c;name=lmon \ |
| | ${SOURCEFORGE_MIRROR}/nmon/Documentation.txt;name=doc \ |
| | " |
| SRC_URI[lmon.md5sum] | = "b1b8e6c0123ad232394991f244f40494" |
| SRC_URI[lmon.sha256sum] | = "456ab2a342b31d1a352d0d940af5962fa65a12ae8757ff73e6e73210832ae8b5" |
| SRC_URI[doc.md5sum] | = "dbb13658cf55d687c4f2ff771a696d4a" |
| SRC_URI[doc.sha256sum] | = "1f7f83afe62a7210be5e83cd24157adb854c14599e0b377a7ecca933869278" |

| CFLAGS | += "-D JFS -D GETUSER -Wall -DLARGEMEM" |
| LDFLAGS | += "-ltinfo -lncursesw" |

```
do_compile() {
  ${CC} ${CFLAGS} ${LDFLAGS} ${WORKDIR}/lmon13g.c -o nmon
}
```

```
do_install() {
  install -d ${D}${bindir}
  install -m 0755 nmon ${D}${bindir}
}
```
recipies-example/helloworld/helloworld_1.0.bb

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
Workflow - 5. Create a recipe

recipes-example/helloworld/helloworld_1.0.bb

| DESCRIPTION = "Print a friendly, customizable greeting"
| HOMEPAGE = "https://www.gnu.org/software/hello/

"Header"
| PRIORITY = "optional"
| SECTION = "examples"
| LICENSE = "GPLv3"

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

"Tasks"
| inherit autotools
Always use **remote repositories** to host your application sources
⇒ Makes development quicker + keep history
Always use remote repositories to host your application sources
⇒ Makes development quicker + keep history

Do not put application sources in your layer directly!
⇒ Application development ≠ Application Integration
Always use **remote repositories** to host your application sources
⇒ Makes development quicker + keep history

Do not put **application sources** in your layer directly!
⇒ Application development ≠ Application Integration

Keep the same **folder organization**: `recipes-core/recipes-bsp/recipes-devtools/…`
⇒ Find recipes quicker
Always use **remote repositories** to host your application sources
⇒ Makes development quicker + keep history

Do not put **application sources** in your layer directly!
⇒ Application development ≠ Application Integration

Keep the same **folder organization**: `recipes-core/recipes-bsp/recipes-devtools/...`
⇒ Find recipes quicker

Keep the **headers / sources / tasks** organization in the recipe
⇒ All the recipes have the same content organization
✓ Always use **remote repositories** to host your application sources
   ⇒ Makes development quicker + keep history

✓ Do not put **application sources** in your layer directly!
   ⇒ Application development ≠ Application Integration

✓ Keep the same **folder organization**: `recipes-core/recipes-bsp/recipes-devtools/…`
   ⇒ Find recipes quicker

✓ Keep the **headers / sources / tasks** organization in the recipe
   ⇒ All the recipes have the same content organization

✓ Use/Create **include files** when possible
   ⇒ Can extend other versions easily
Workflow - 5. Create a recipe

✓ Always use **remote repositories** to host your application sources
   ⇒ Makes development quicker + keep history

✓ Do not put **application sources** in your layer directly!
   ⇒ Application development ≠ Application Integration

✓ Keep the same **folder organization**: recipes-core/recipes-bsp/recipes-devtools/...
   ⇒ Find recipes quicker

✓ Keep the **headers / sources / tasks** organization in the recipe
   ⇒ All the recipes have the same content organization

✓ Use/Create **include files** when possible
   ⇒ Can extend other versions easily

✓ Know how to **compile** the application **manually** before integrating it in a recipe
   ⇒ Saves you time
It is a good practice not to modify recipes available in Poky.

But it is sometimes useful to modify an existing recipe. The BitBake build engine allows to modify a recipe by extending it. The recipe extensions end in `.bbappend`. Appended files must have the same root name as the recipe they extend, for example `example_0.1.bbappend` applies to `example_0.1.bb`.

If adding new files, you must prepend the `FILESEXTRAPATHS` variable with the path to files' directory.
It is a good practice **not** to modify recipes available in Poky.
It is a good practice **not** to modify recipes available in Poky.

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

But it is sometimes useful to modify an existing recipe

The BitBake *build engine* allows to modify a recipe by **extending** it

>"\textit{Workflow - 6. Extend a recipe}"

\begin{itemize}
  \item It is a good practice \textbf{not} to modify recipes available in Poky.
  \item But it is sometimes useful to modify an existing recipe
  \item The BitBake \textit{build engine} allows to modify a recipe by \textbf{extending} it
\end{itemize}
It is a good practice **not** to modify recipes available in Poky.

But it is sometimes useful to modify an existing recipe

The BitBake *build engine* allows to modify a recipe by **extending** it

The recipe extensions end in `.bbappend`
It is a good practice **not** to modify recipes available in Poky.

But it is sometimes useful to modify an existing recipe

The BitBake *build engine* allows to modify a recipe by **extending** it

The recipe extensions end in `.bbappend`

Appended files must have the **same root name** as the recipe they extend

example_0.1.bbappend applies to example_0.1.bb

⇒ **version specific**
It is a good practice **not** to modify recipes available in Poky.

But it is sometimes useful to modify an existing recipe

The BitBake *build engine* allows to modify a recipe by **extending** it

The recipe extensions end in `.bbappend`

Appended files must have the **same root name** as the recipe they extend

example_0.1.bbappend applies to example_0.1.bb

⇒ **version specific**

If adding new files, you must prepend the `FILESEXTARPATHS` variable with the path to files’ directory.
Workflow - 6. Extend a recipe

recipes-support/nmon/nmon_13g.bbappend

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

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

do_install_append() {
    # Do something
}

---

|--- conf
|   |--- layer.conf
|--- recipes-support
|   |--- nmon
|       |--- files
|           |--- custom-modification-0.patch
|           |--- custom-modification-1.patch
|--- nmon_13g.bbappend
Workflow - 7. Create an image

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
```

An image is no more than a recipe. To create an image, simply create a `.bb` in an `images` folder:

```
mkdir -p recipes-core/images/
touch recipes-core/images/core-image-fe.bb
```
An **image** is the top level recipe and is used alongside the **machine** definition.
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.

To create an image, simply create a `.bb` in an `images` folder:

```
mkdir -p recipes-core/images/
touch recipes-core/images/core-image-fe.bb
```
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:
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
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

An image is no more than a recipe.
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

An image is no more than a recipe. To create an image, simply create a .bb in an images folder.
Workflow - 7. Create an image

- 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`
- An **image** is no more than a recipe.
- To create an image, simply create a `.bb` in an `images` folder.

```
mkdir -p recipes-core/images/
touch recipes-core/images/core-image-fe.bb
```
Some special configuration variables are used to describe an image:

- **IMAGE_INST ALL**
  - List of packages to install in the generated image

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

- **Create a minimal image**
  - Allows to have a minimal rootfs

- **Create different images according to your needs: image-minimal, image-dev, image-x11, image-qt5, etc**
  - Install only what you really need for your board.
Some special configuration variables are used to describe an image:
Some special configuration variables are used to describe an image:

- **IMAGE_INSTALL**  List of packages to install in the generated image
Some special configuration variables are used to describe an image:

- **IMAGE_INSTALL**  List of packages to install in the generated image
- **IMAGE_FSTYPES** List of formats the OpenEmbedded build system will use to create images
Some special configuration variables are used to describe an image:

- **IMAGE_INSTALL**  List of packages to install in the generated image
- **IMAGE_FSTYPES**  List of formats the OpenEmbedded build system will use to create images

✓ Create a **minimal image** to include it in others
   ⇒ Allows to have a minimal rootfs
Some special configuration variables are used to describe an image:

- **IMAGE_INSTALL**  List of packages to install in the generated image
- **IMAGE_FSTYPES**  List of formats the OpenEmbedded build system will use to create images

- Create a **minimal image** to include it in others
  - Allows to have a minimal rootfs
- Create different images according to your needs: image-minimal, image-dev, image-x11, image-qt5, etc
  - Install only what you really need for your board.
recipes-core/images/core-image-fe.bb

inherit core-image

DESCRIPTION = "A small image to boot a device, created for Embedded Recipes"
LICENSE = "MIT"

IMAGE_FSTYPES = "tar.bz2 ext4"
IMAGE_INSTALL = "packagegroup-core-boot \
nmon \
helloworld \
"
Workflow - 8. Create a machine

- Create a machine
  - A machine describes your hardware
    - Stored under `meta-conf/machine/*.conf`
    - The file name corresponds to the value set in the `MACHINE` variable
      - `meta-ti/conf/machine/beaglebone.conf`
        - MACHINE = "beaglebone"
  - Contains configuration variables related to the architecture, to machine’s features and to customize the kernel image or the filesystems used.
    - `TARGET_ARCH`: The architecture of the device being built
    - `PREFERRED_PROVIDER_virtual/kernel`: The kernel recipe to use
    - `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`

- Describe your machine in a README file

- Kernel, drivers and embedded Linux - Development, consulting, training and support - https://bootlin.com
Workflow - 8. Create a machine

- A machine describes your **hardware**
A machine describes your **hardware**

- Stored under `meta-<bsp_name>/conf/machine/*.conf`
A machine describes your **hardware**

- Stored under `meta-<bsp_name>/conf/machine/*.conf`
- The file name corresponds to the value set in the **MACHINE** variable

```bash
meta-ti/conf/machine/beaglebone.conf
MACHINE = "beaglebone"
```
A machine describes your **hardware**

- Stored under `meta-<bsp_name>/conf/machine/*.conf`
- The file name corresponds to the value set in the `MACHINE` variable
  
  ```
  meta-ti/conf/machine/beaglebone.conf
  MACHINE = "beaglebone"
  ```

- Contains configuration variables related to the architecture, to machine’s features and to customize the kernel image or the filesystems used.
A machine describes your **hardware**

Stored under `meta-<bsp_name>/conf/machine/*.conf`

The file name corresponds to the value set in the **MACHINE** variable

```bash
meta-ti/conf/machine/beaglebone.conf
MACHINE = "beaglebone"
```

Contains configuration variables related to the architecture, to machine’s features and to customize the kernel image or the filesystems used.

**TARGET_ARCH** : The architecture of the device being built
A machine describes your **hardware**

- Stored under `meta-<bsp_name>/conf/machine/*`.conf
- The file name corresponds to the value set in the `MACHINE` variable
  `meta-ti/conf/machine/beaglebone.conf`
  `MACHINE = "beaglebone"

- Contains configuration variables related to the architecture, to machine’s features and to customize the kernel image or the filesystems used.
  - `TARGET_ARCH` : The architecture of the device being built
  - `PREFERRED_PROVIDER_virtual/kernel` : The kernel recipe to use
A machine describes your **hardware**

- Stored under `meta-<bsp_name>/conf/machine/*.conf`
- The file name corresponds to the value set in the `MACHINE` variable
  
  ```
  meta-ti/conf/machine/beaglebone.conf
  MACHINE = "beaglebone"
  ```

- Contains configuration variables related to the architecture, to machine’s features and to customize the kernel image or the filesystems used.
  
  - `TARGET_ARCH` : The architecture of the device being built
  - `PREFERRED_PROVIDER_virtual/kernel` : The kernel recipe to use
  - `SERIAL_CONSOLE` : Speed and device for the serial console to attach. Passed to the kernel as the `console` parameter, e.g. `115200 ttyS0`
Workflow - 8. Create a machine

- A machine describes your **hardware**
- Stored under `meta-<bsp_name>/conf/machine/+.conf`
- The file name corresponds to the value set in the `MACHINE` variable
  
  ```
  meta-ti/conf/machine/beaglebone.conf
  MACHINE = "beaglebone"
  ```

  - Contains configuration variables related to the architecture, to machine's features and to customize the kernel image or the filesystems used.

    - `TARGET_ARCH` : The architecture of the device being built
    - `PREFERRED_PROVIDER_virtual/kernel` : The kernel recipe to use
    - `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`
Workflow - 8. Create a machine

- A machine describes your **hardware**
- Stored under **meta-<bsp_name>/conf/machine/*.conf**
- The file name corresponds to the value set in the **MACHINE** variable
  `meta-ti/conf/machine/beaglebone.conf`
  `MACHINE = "beaglebone"`
- Contains configuration variables related to the architecture, to machine’s features and to customize the kernel image or the filesystems used.
  - **TARGET_ARCH**: The architecture of the device being built
  - **PREFERRED_PROVIDER_virtual/kernel**: The kernel recipe to use
  - **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`

✓ Describe your machine in a **README** file
conf/machine/fe-machine.conf

require conf/machine/include/soc-family.inc
require conf/machine/include/tune-cortexa5.inc

TARGET_ARCH = "arm"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-at91"
PREFERRED_PROVIDER_virtual/bootloader ?= "u-boot-at91"

KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "at91-sama5d3_xplained.dtb"

SERIAL_CONSOLE ?= "115200 ttyS0"
Conclusion

User/Developer actions

- Download
- Configure
- Compile

Developer's work

- Create layer
  - Create recipe
  - Create image
  - Create machine
  - Extend recipe
  - Create distro
Thank you for listening!
Questions? Suggestions? Comments?

Mylène Josserand
mylene@bootlin.com

Slides under CC-BY-SA 3.0
http://bootlin.com/pub/conferences/2017/embedded-recipes/josserand-introduction-to-yocto-project/