yocto PROJECT

Workshop
Yocto Project, an automatic generator of embedded linux distributions

Marco Cavallini, KOAN

LINUXLAB
BUILDING SMARTER DEVICES

Rights to copy

ftp://ftp.koansoftware.com/public/talks/LinuxLAB-2018/LinuxLAB-2018-Yocto-Koan.pdf

Agenda

- Workshop details
  - How we did it, setup and boot QEMU
- Yocto introduction details
  - General concepts
- Layers
  - Needed to start doing something useful
- Recipes
  - Extending layers with a new recipe
- Debugging
  - Typical debugging tecniques
- Images
  - Create a new image
- Devtool
  - Create and modify a recipe using devtool

Lecture details

During 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 or Yocto Project with other systems used in your company.

Workshop activities

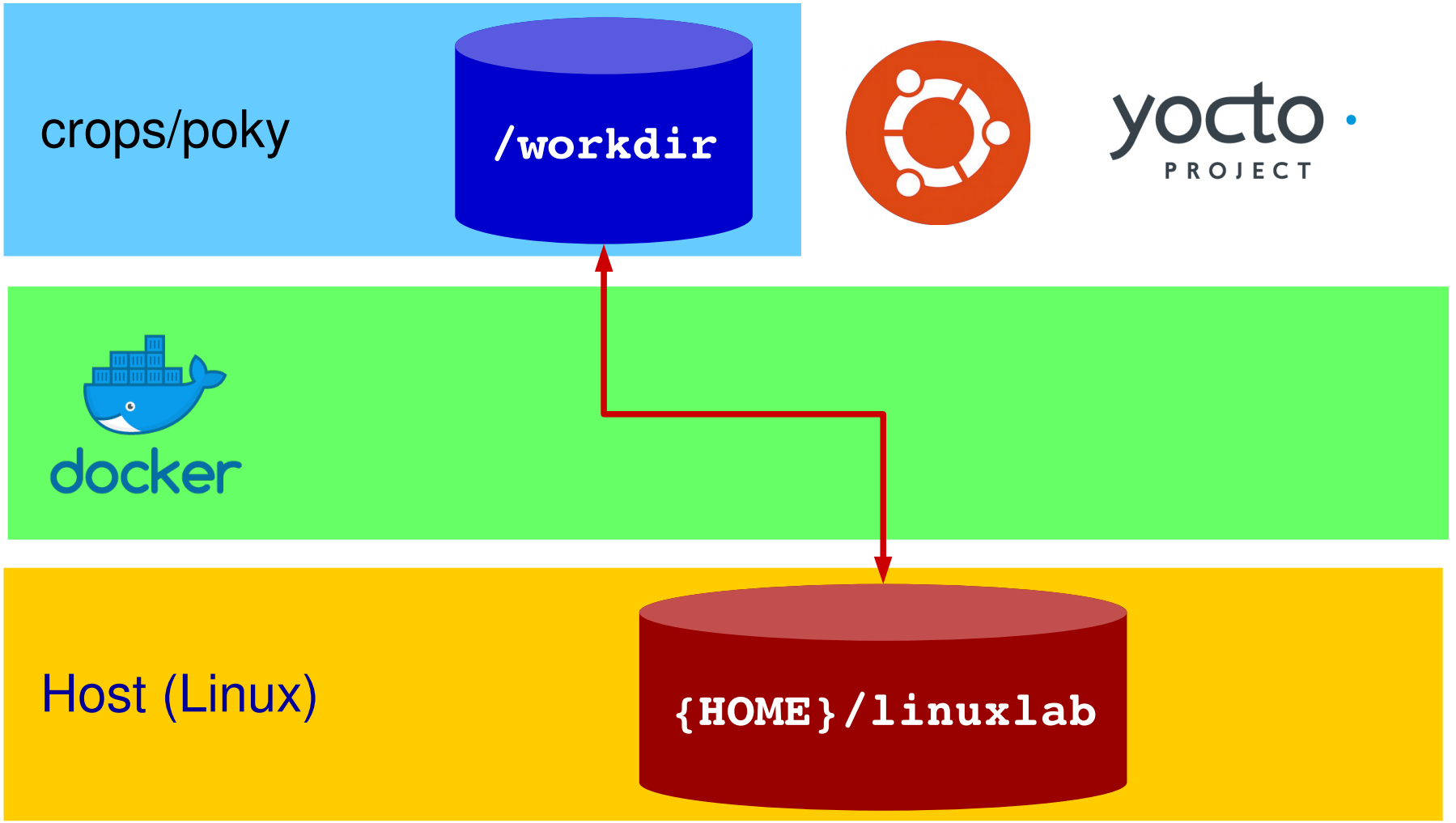
This is a workshop, so you are free to study and analyze the internals of the system during lectures.
An icon saying **“Try This!”** in a slide will point you out when you can dig in the code:

Workshop details

- The setup of this workshop is described in the project website: https://github.com/koansoftware/linuxlab-yocto
- Based on CROPS/poky-container (CROss PlatformS)
- Start the docker system

```
docker run --rm -it -v \
      ${HOME}/linuxlab:/workdir crops/poky \
         --workdir=/workdir
```


Workshop details

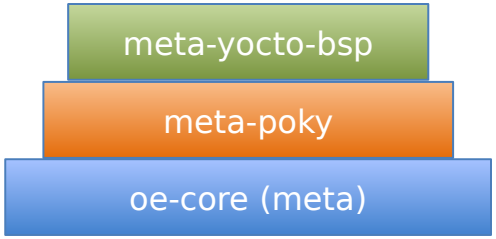


Poky system layout

```
/workdir/poky/
 |---build      (or whatever name you choose)
       Project build directory
 |---downloads (DL_DIR)
     Downloaded source cache
 |---poky       (Do Not Modify anything in here*)
  Poky, bitbake, scripts, oe-core, metadata
 |---sstate-cache (SSTATE_DIR)
     Binary build cache
```

Poky directory layout

```
/workdir/poky/
|---LICENSE
|---README
|---README.hardware
|---bitbake/                (The build tool)
|---documentation/
|---meta/                   (oe-core)
|---meta-poky/              (Yocto distro metadata)
|---meta-yocto-bsp/         (Yocto Reference BSPs)
|---oe-init-build-env       (Project setup script)
|---scripts/                (Scripts and utilities)
```



Note: A few files have been items omitted to facility the presentation on this slide

Setting up a build directory

- **Start by setting up a build directory**
  - Local configuration
  - Temporary build artifacts


```
$ cd /workdir/poky/
$ source ./oe-init-build-env build
```

- It is possible to replace *build* with whatever directory name you want to use for your project

- **IMPORTANT: You need to re-run this script in any new terminal you start (and don't forget the project directory)**

Build directory layout

```
/workdir/poky/build/
|---bitbake.lock
|---cache/                  (bitbake cache files)
|---conf/
|    |--bblayers.conf       (bitbake layers)
|    |--local.conf          (local configuration)
|    `--site.conf           (optional site conf)
`---tmp/                    (Build artifacts)
```

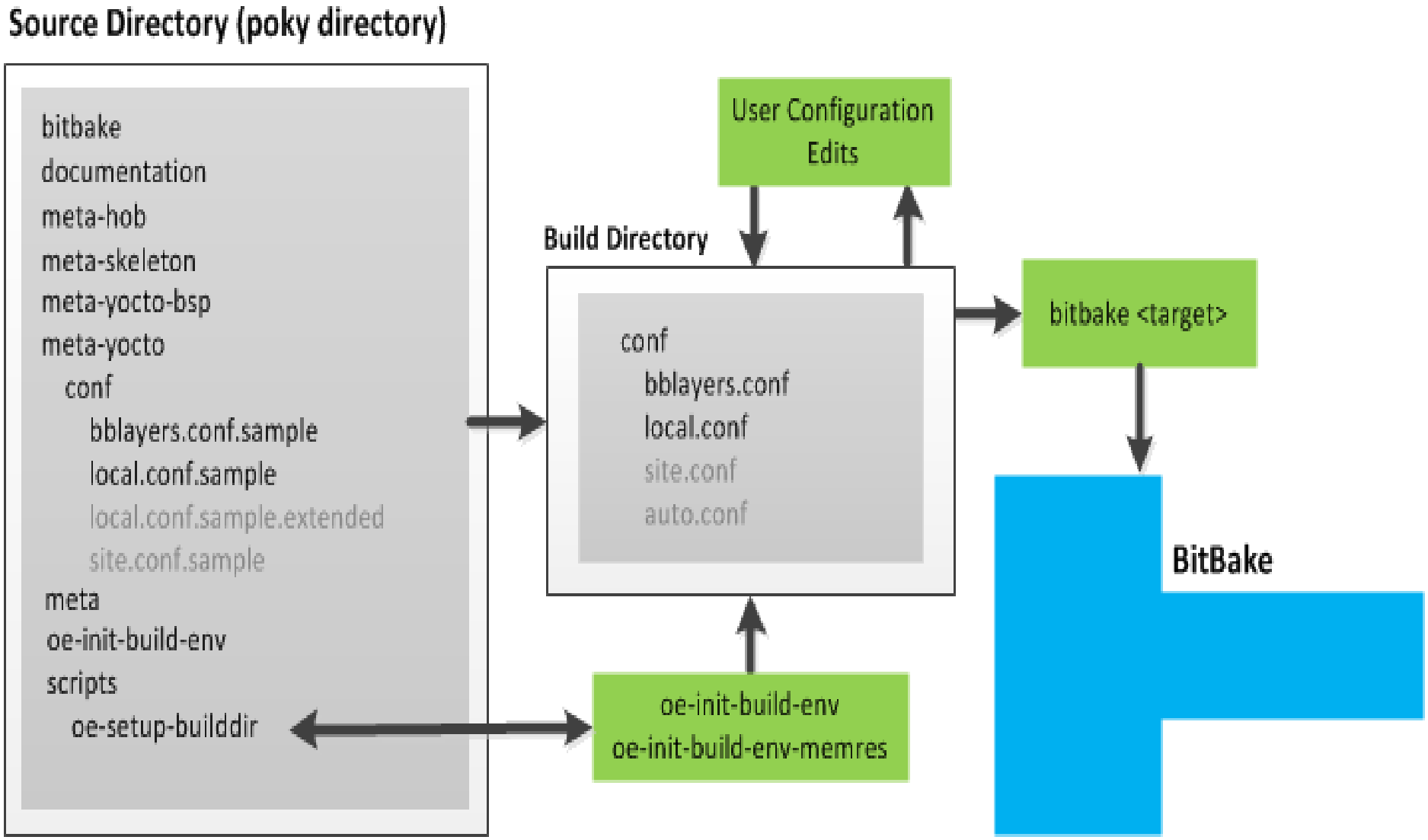
Building a linux image

- **General Procedure:**
  - Create a project directory using

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

  - Configure build by editing local.conf
  - `/workdir/poky/build/conf/local.conf`
    - Select appropriate `MACHINE` type
    - Set shared downloads directory (`DL_DIR`)
    - Set shared state directory (`SSTATE_DIR`)
  - Build your selected Image
  - $ bitbake -k core-image-minimal
  - (Detailed steps follow…)

User configuration



Typical build configuration

- You usually need to configure build by editing local.conf

  `/workdir/poky/build/conf/local.conf`

  - Set appropriate `MACHINE`, `DL_DIR` and `SSTATE_DIR`
  - Add the following to the bottom of local.conf

```
MACHINE = "qemuarm"
DL_DIR = "${SOMEWHERE}/downloads"
SSTATE_DIR = "${SOMEWHERE}/sstate-cache/${MACHINE}"
```

Building the final image

- This builds an entire embedded Linux distribution
- Choose from one of the available Images
- The following builds a minimal embedded target

```
$ bitbake -k core-image-minimal
```

- On a fast computer the first build may take the better part of an hour on a slow machine multiple ...
- The next time you build it (with no changes) it may take as little as 5 mins (due to the shared state cache)

Generated artefacts

- All the artefacts genarated are stored in the deploy directory
- `/workdir/poky/build/tmp/deploy/image/qemuarm`
- Look inside this directory!

Booting your image with QEMU

- The runqemu script is used to boot the image with QEMU
- It auto-detects settings as much as possible, enabling the following command to boot our reference images:

  $ runqemu qemuarm slirp nographic

  - Use slirp when using docker
  - Use nographic if using a non-graphical session (ssh)

This is to run your QEMU target system

- Replace *qemuarm* with your value of `MACHINE`
- Your QEMU instance should boot
- Quit by closing the qemu window
- If using “nographic”, kill it from another terminal:

  $ `killall qemu-system-arm`

- or pressing these keys from the QEMU terminal

  `[ Ctrl-A X ]`

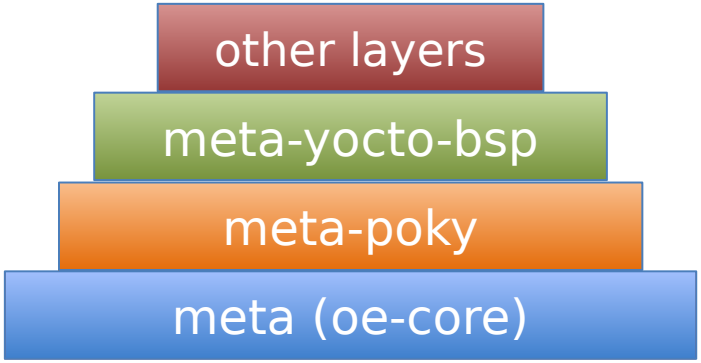
This is to kill the QEMU target

YOCTO PROJECT

This section will introduce the basic concepts of the Yocto Project

Yocto Project overview

- **Collection of tools and methods enabling**
  - Rapid evaluation of embedded Linux on many popular off-the-shelf boards
  - Easy customization of distribution characteristics
- **Supports x86, ARM, MIPS, PowerPC**
- **Based on technology from the OpenEmbedded Project**
- **Layer architecture allows for easy re-use of code**



What is the Yocto Project

- Umbrella organization under Linux Foundation
- Backed by many companies interested in making Embedded Linux easier for the industry
- Co-maintains OpenEmbedded Core and other tools (including opkg)

Yocto Project governance

- Organized under the Linux Foundation
- Technical Leadership Team
- Advisory Board made up of participating organizations

Project member organizations

Platinum members

Gold members

Silver members

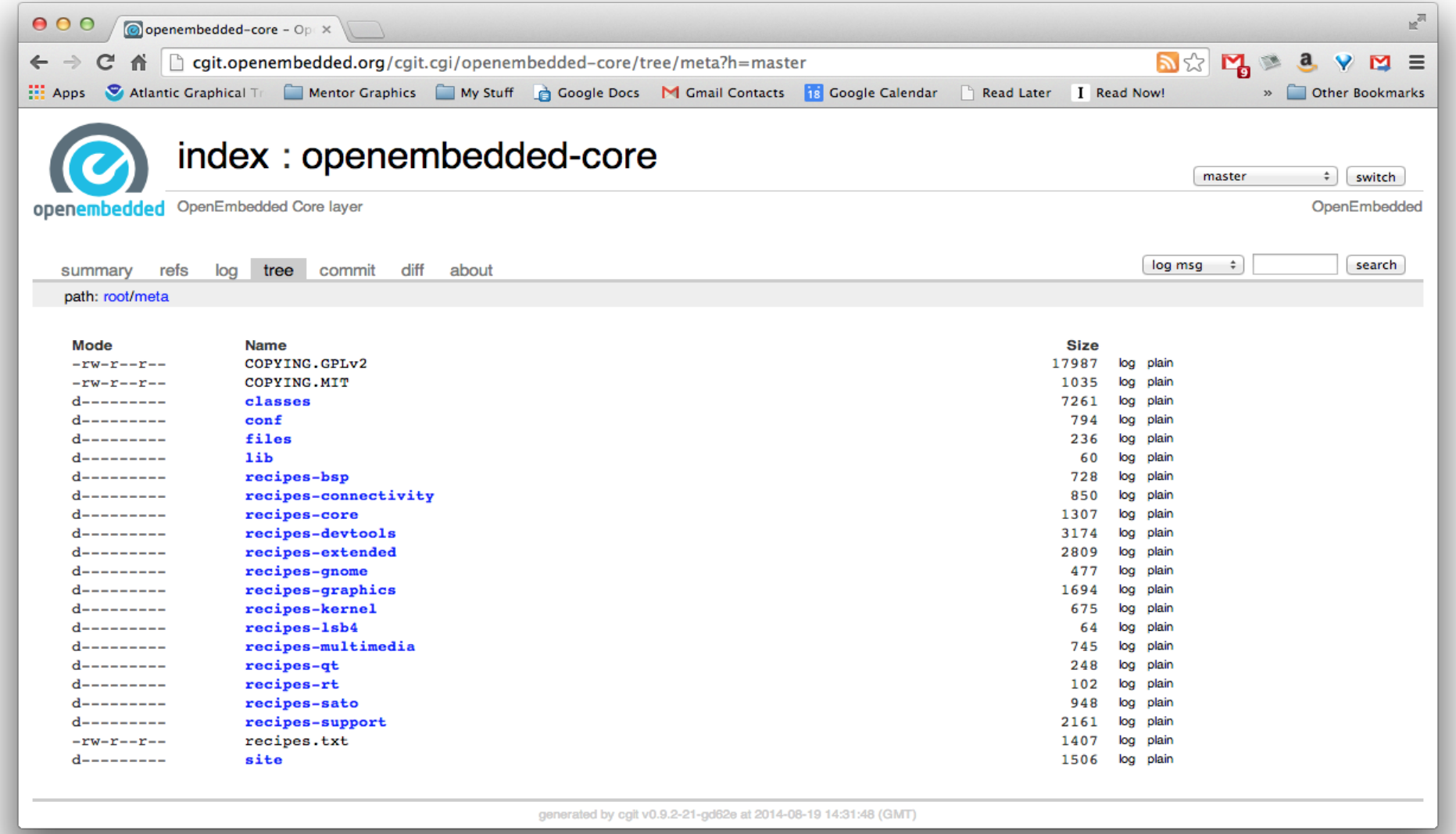
Yocto Project overview

- **YP builds packages - then uses these packages to build bootable images**
- Supports use of popular package formats including:
  - rpm, deb, ipk
- Releases on a 6-month cadence
- Latest (stable) kernel, toolchain and packages, documentation
- App Development Tools including Eclipse plugin, SDK, toaster

Yocto release versions

Name	Revision	Poky	Release Date
Bernard	1.0	5.0	Apr 5, 2011
Edison	1.1	6.0	Oct 17, 2011
Denzil	1.2	7.0	Apr 30, 2012
Danny	1.3	8.0	Oct 24, 2012
Dylan	1.4	9.0	Apr 26, 2013
Dora	1.5	10.0	Oct 19, 2013
Daisy	1.6	11.0	Apr 24, 2014
Dizzy	1.7	12.0	Oct 31, 2014
Fido	1.8	13.0	Apr 22, 2015
Jethro	2.0	14.0	Oct 31, 2015
Krogoth	2.1	15.0	Apr 29, 2016
Morty	2.2	16.0	Oct 28, 2016

Name	Revision	Poky	Release Date
Pyro	2.3	17.0	Apr, 2017
Rocko	2.4	18.0	Oct, 2017
Sumo	2.5	19.0	Apr, 2018
Thud	2.6	20.0	Oct, 2018
???	2.7	21.0	Apr, 2019
			Oct, 2019
			Apr, 2020
			Oct, 2020
			...



Introduction to OpenEmbedded

- **The OpenEmbedded Project co-maintains OE-core build system:**
  - bitbake build tool and scripts
  - Metadata and configuration
- **Provides a central point for new metadata**
  - (see the OE Layer index)

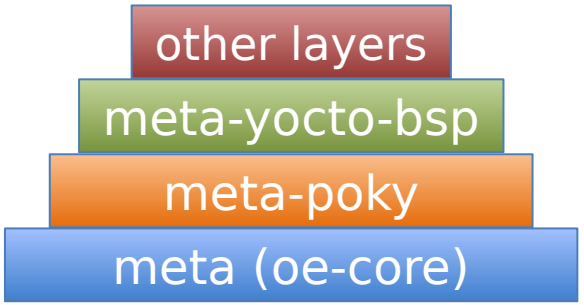
What is Bitbake

- **Bitbake**
  - Powerful and flexible build engine (Python)
  - Reads metadata
  - Determines dependencies
  - Schedules tasks

Metadata – a structured collection of "recipes" which tell BitBake what to build, organized in layers

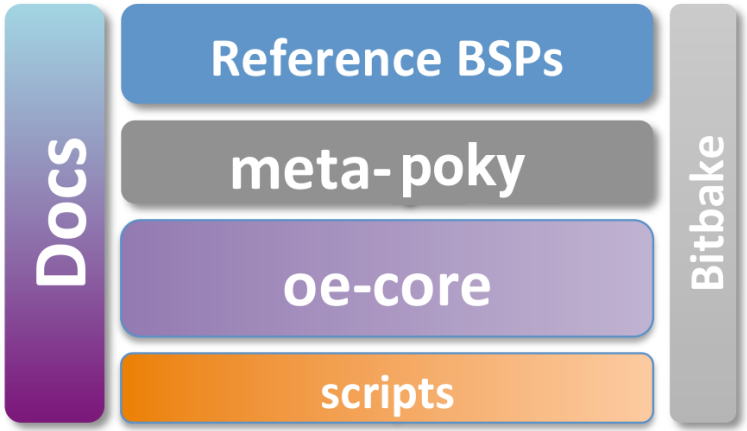
What is Poky

- **Poky is a reference distribution**
- **Poky has its own git repo**
  - `git clone git://git.yoctoproject.org/poky`
- **Primary Poky layers**
  - oe-core (poky/meta)
  - meta-poky (poky/meta-poky)
  - meta-yocto-bsp
- **Poky is the starting point for building things with the Yocto Project**



Poky in detail

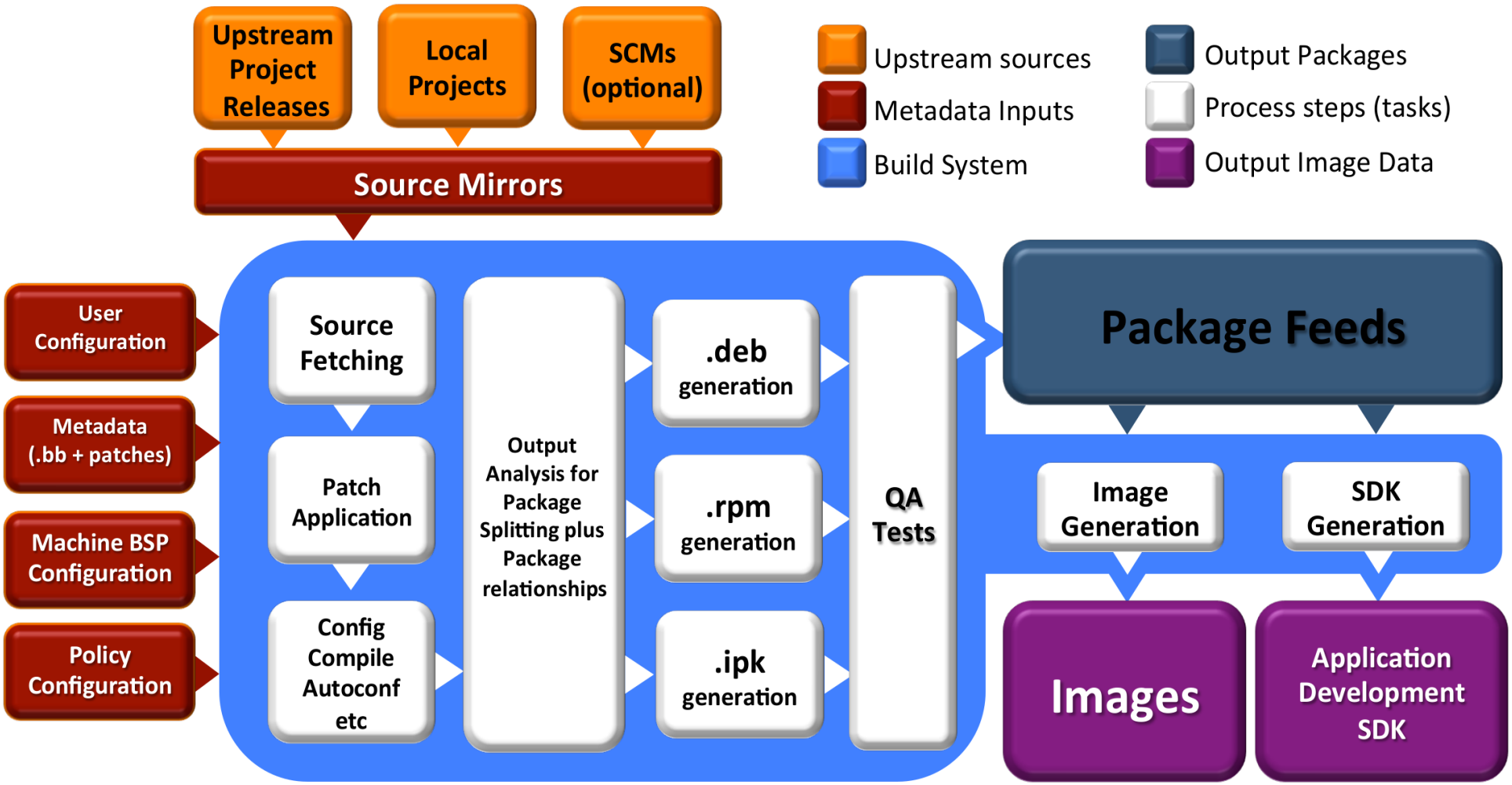
- **Contains core components**
  - Bitbake tool: A python-based build engine
  - Build scripts (infrastructure)
  - Foundation package recipes (**oe-core**)
  - meta-poky (Contains distribution policy)
  - Reference BSPs
  - Yocto Project documentation



Putting it all together

- **Yocto Project** is a large collaboration project
- **OpenEmbedded** is providing most metadata
- **Bitbake** is the build tool
- **Poky** is the Yocto Project's reference distribution
  - Poky contains a version of bitbake and oe-core from which you can start your project

Build system workflow

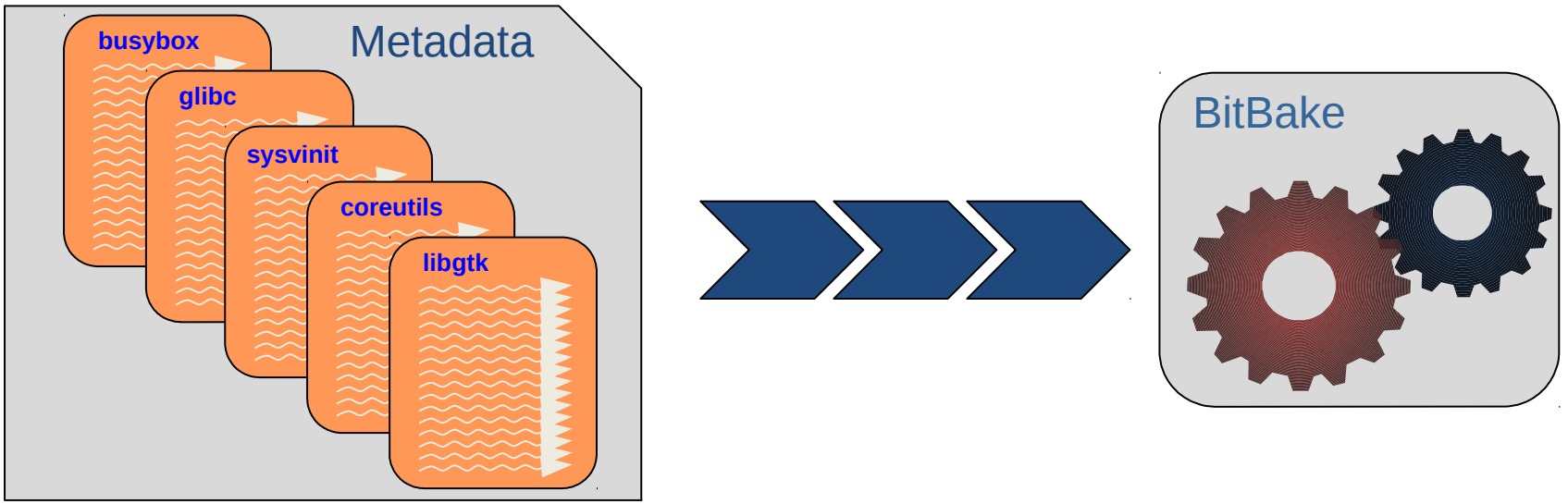


BITBAKE

This section will introduce the concept of the bitbake build tool and how it can be used to build recipes

Metadata and bitbake

- **Most common form of metadata: The Recipe**
- **A *Recipe* provides a “list of ingredients” and “cooking instructions”**
- **Defines settings and a set of tasks used by bitbake to build binary packages**



What is metadata

- **Metadata exists in four general categories:**
- **Recipes (*.bb)**
  - Usually describe build instructions for a single package
- **PackageGroups (special *.bb)**
  - Often used to group packages together for a FS image
- **Classes (*.bbclass)**
  - Inheritance mechanism for common functionality
- **Configuration (*.conf)**
  - Drives the overall behavior of the build process

Other metadata

- **Append files (*.bbappend)**
  - Define additional metadata for a similarly named .bb file
  - Can add or override previously set values
- **Include files (*.inc)**
  - Files which are used with the *`include`* directive
  - Also can be included with *`require`* (mandatory include)
  - Include files are typical found via the BBPATH variable

Introduction to bitbake

- ***Bitbake* is a task executor and scheduler**
- **By default the *build* task for the specified recipe is executed**

  $ bitbake myrecipe

- **You can indicate which task you want run**

  $ bitbake -c clean myrecipe

  $ bitbake -c cleanall myrecipe

- **You can get a list of tasks with**

  $ bitbake -c listtasks myrecipe

Building recipes

- **By default the highest version of a recipe is built (can be overridden with `DEFAULT_PREFERENCE` or `PREFERRED_VERSION` metadata)**
  $ bitbake myrecipe
- **You can specify the version of the package you want built (version of upstream source)**
  $ bitbake myrecipe-1.0
- **You can also build a particular revision of the package metadata**
  $ bitbake myrecipe-1.0-r0
- **Or you can provide a recipe file to build**
  $ bitbake -b mydir/myrecipe.bb

Running bitbake for the first time

- **When you do a really big build, running with *--continue* (*-k*) means bitbake will proceed as far as possible after finding an error**
  - $ bitbake -k core-image-minimal
  - **When running a long build (e.g. overnight) you want as much of the build done as possible before debugging issues**
- **Running bitbake normally will stop on the first error found**
  - $ bitbake core-image-minimal
- ***We'll look at debugging recipe issue later...***

Bitbake is a task scheduler

- **Bitbake builds recipes by scheduling build tasks in parallel**
  $ bitbake recipe
- **This looks for `recipe.bb` in `BBFILES`**
- **Each recipe defines build tasks, each which can depend on other tasks**
- **Recipes can also depend on other recipes, meaning more than one recipe may be built**
- **Tasks from more than one recipe are often executed in parallel at once on multi-cpu build machines**

Bitbake default tasks*

`do_fetch`	**Locate and download source code**
`do_unpack`	**Unpack source into working directory**
`do_patch`	**Apply any patches**
`do_configure`	**Perform any necessary pre-build configuration**
`do_compile`	**Compile the source code**
`do_install`	**Installation of resulting build artifacts in WORKDIR**
`do_populate_sysroot`	**Copy artifacts to sysroot**
`do_package_*`	**Create binary package(s)**

Note: to see the list of all possible tasks for a recipe, do this:

```
$ bitbake -c listtasks <recipe_name>
```

*Simplified for illustration

Simple recipe task list*

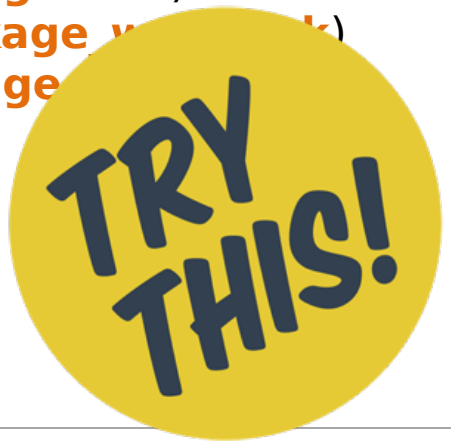
chris — sleep — 117×32 — ⌘5

$ **bitbake hello**

NOTE: Running task 337 of 379 (ID: 4, hello_1.0.0.bb, **do_fetch**)
NOTE: Running task 368 of 379 (ID: 0, hello_1.0.0.bb, **do_unpack**)
NOTE: Running task 369 of 379 (ID: 1, hello_1.0.0.bb, **do_patch**)
NOTE: Running task 370 of 379 (ID: 5, hello_1.0.0.bb, **do_configure**)
NOTE: Running task 371 of 379 (ID: 7, hello_1.0.0.bb, **do_populate_lic**)
NOTE: Running task 372 of 379 (ID: 6, hello_1.0.0.bb, **do_compile**)
NOTE: Running task 373 of 379 (ID: 2, hello_1.0.0.bb, **do_install**)
NOTE: Running task 374 of 379 (ID: 11, hello_1.0.0.bb, **do_package**)
NOTE: Running task 375 of 379 (ID: 3, hello_1.0.0.bb, **do_populate_sysroot**)
NOTE: Running task 376 of 379 (ID: 8, hello_1.0.0.bb, **do_packagedata**)
NOTE: Running task 377 of 379 (ID: 12, hello_1.0.0.bb, **do_package_w...k**)
NOTE: Running task 378 of 379 (ID: 9, hello_1.0.0.bb, **do_package**

Tip: call this example

bitbake psplash | cat



*Output has been formatted to fit this slide.

*Simplified for illustration

sstate cache

- **Several bitbake tasks can use past versions of build artefacts if there have been no changes since the last time you built them**

do_packagedata	Creates package metadata used by the build system to generate the final packages
do_package	Analyzes the content of the holding area and splits it into subsets based on available packages and files
do_package_write_ipk	Creates the actual **.ipk** packages and places them in the Package Feed area
do_populate_lic	Writes license information for the recipe that is collected later when the image is constructed
do_populate_sysroot	Copies a subset of files installed by do_install into the sysroot in order to make them available to other recipes

Simple recipe build from sstate*

chris — sleep — 117×32 — ⌘5

```
$ bitbake -c clean hello

$ bitbake hello

NOTE: Running setscene task 69 of 74 (hello_1.0.0.bb, do_populate_sysroot_setscene)
NOTE: Running setscene task 70 of 74 (hello_1.0.0.bb, do_populate_lic_setscene)
NOTE: Running setscene task 71 of 74 (hello_1.0.0.bb, do_package_qa_setscene)
NOTE: Running setscene task 72 of 74 (hello_1.0.0.bb, do_package_write_ipk_setscene)
NOTE: Running setscene task 73 of 74 (hello_1.0.0.bb, do_packagedata_setscene)
```

*Output has been formatted to fit this slide.

*Simplified for illustration

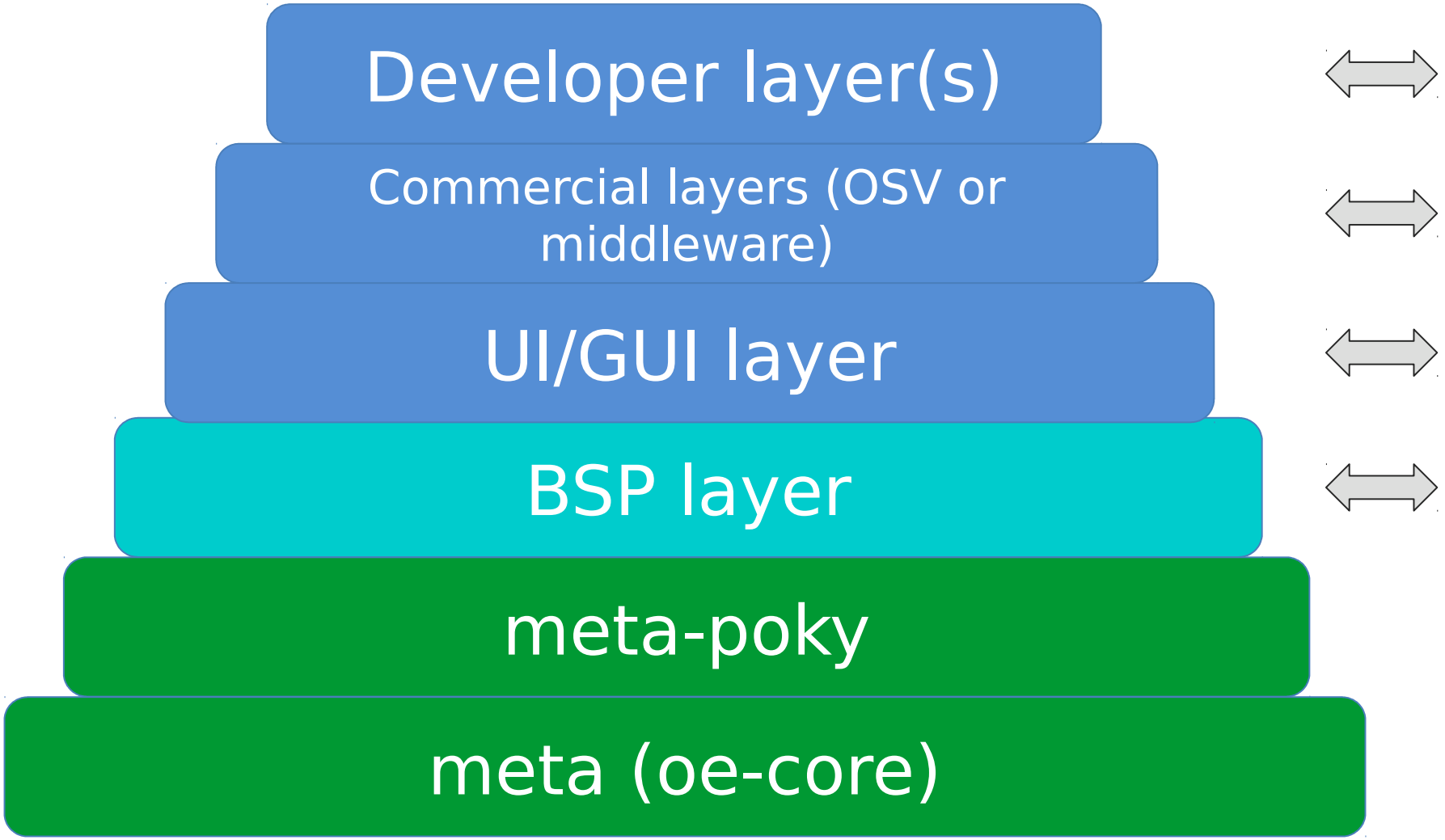
LAYERS

This section will introduce the concept of layers and how important they are in the overall build architecture

Layers

- Metadata is provided in a series of layers which allow you to override any value without editing the originally provided files
- A layer is a logical collection of metadata in the form of recipes
- A layer is used to represent oe-core, a Board Support Package (BSP), an application stack, and your new code
- All layers have a priority and can override policy, metadata and config settings of layers with a lesser priority

Layer hierarchy



Board Support Packages

- **BSPs are layers to enable support for specific hardware platforms**
- **Defines machine configuration variables for the board (`MACHINE`)**
- **Adds machine-specific recipes and customizations**
  - Boot loader
  - Kernel config
  - Graphics drivers (e.g, Xorg)
  - Additional recipes to support hardware features

Using layers

- **Layers are added to your build by inserting them into the BBLAYERS variable within your bblayers file**
  /workdir/poky/build/conf/bblayers.conf

```
BBLAYERS ?= "                                   \
    ${HOME}/yocto/poky/meta                     \
    ${HOME}/yocto/poky/meta-poky                \
    ${HOME}/yocto/poky/meta-yocto-bsp           \
    "
```

Note on using layers

- When doing development with Yocto, do not edit files within the Poky source tree
- Use a new custom layer for modularity and maintainability
- Layers also allow you to easily port from one version of Yocto/Poky to the next version

Creating a custom layer

- They all start with “*meta-*” by convention
- They all stored in the same directory by convention

- There are three ways to create a new layer:

1. Manually *(maybe copying from an existing one)*
2. Using the *yocto-layer* tool *(covered in the next slide)*
3. Using *bitbake-layer**s*** tool *(covered later)*

- **NOTE: The *yocto-layer* tool is deprecated and no longer available starting from version *sumo*.**

yocto-layer to create custom layers

- Using the *yocto-layer* tool is possible to create a layer
  - $ **yocto-layer create** meta-training
  - This will create *meta-training* in the current dir


```
yocto-layer create meta-training \
                  -o /workdir/yocto/poky/meta-training
```

NOTE: Instead of this command we are going to use a pre-configured layer from GitHub *(covered later)*
https://github.com/koansoftware/meta-training

bitbake-layers to create custom layers

- Using *bitbake-layers* tool is possible to create a layer
  - $ bitbake-layers create-layer ...
    - The default priority of the layer is: 6

```
bitbake-layers create-layer \
                    --example-recipe-name dummy \
                              /workdir/poky/meta-linuxlab
```

New custom layer

$ tree /workdir/poky/meta-linuxlab

```
meta-linuxlab/
|--COPYING.MIT              (The license file)
|--README                   (Starting point for README)
|--conf
|   `--layer.conf           (Layer configuration file)
`--recipes-dummy            (A grouping of recipies)
    `--dummy                 (The example package)
        `--dummy_0.1.bb      (The example recipe)
```

layer.conf

```
# We have a conf and classes directory, add to
BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "linuxlab"
BBFILE_PATTERN_linuxlab = "^${LAYERDIR}/"
BBFILE_PRIORITY_linuxlab = "6"
```


Recipe in the new custom layer (1)

$ cat /workdir/poky/meta-linuxlab/dummy_0.1.bb

```
SUMMARY = "bitbake-layers recipe"
DESCRIPTION = "Recipe created by bitbake-layers"
LICENSE = "MIT"

python do_build() {
    bb.plain("**********************************");
    bb.plain("*                                *");
    bb.plain("*  Example recipe created by bitbake-layers *");
    bb.plain("*                                *");
    bb.plain("**********************************");
}
```

Recipe in the new custom layer (2)


```
$ cat /workdir/poky/meta-linuxlab/dummy_0.1.bb

SUMMARY = "bitbake-layers recipe"
DESCRIPTION = "Recipe created by bitbake-layers"
LICENSE = "MIT"

inherit logging

do_compile() {
    bbplain "**********************************************"
    bbplain "*                                            *"
    bbplain "*  Example recipe created by bitbake-layers  *"
    bbplain "*                                            *"
    bbplain "**********************************************"
}
```

Edit the file and modify this part

Add layers to your build

- **Add your layer to *bblayers.conf***
- /workdir/poky/build/conf/bblayers.conf

```
BBLAYERS ?= "                                \
    ${HOME}/yocto/poky/meta                  \
    ${HOME}/yocto/poky/meta-poky             \
    ${HOME}/yocto/poky/meta-yocto-bsp        \
    ${HOME}/yocto/build/meta-linuxlab        \
    "
```

- **Manually, or using a dedicated command…**
  (next slide)

Adding layers to your build

- Add your layer to *bblayers.conf*
- Using this command is possible to avoid a manual edit of the file bblayers.conf

```
bitbake-layers add-layer \
              /workdir/poky/build/meta-linuxlab
```

bitbake-layers

- The command *bitbake-layers* allows to invesigate the layers in the system

```
bitbake-layers -h
bitbake-layers show-layers
bitbake-layers show-recipes flex
bitbake-layers show-overlayed
bitbake-layers show-appends
```

bitbake-layers

```
$ bitbake-layers –help

usage: bitbake-layers [-d] [-q] [--color COLOR] [-h] <subcommand>

BitBake layers utility
optional arguments:
  -d, --debug           Enable debug output
  -q, --quiet           Print only errors
  --color COLOR         Colorize output (where COLOR is auto, always, never)
  -h, --help            show this help message and exit
subcommands:
  <subcommand>
    layerindex-fetch    Fetches a layer from a layer index along with its
                        dependent layers, and adds them to conf/bblayers.conf.
    layerindex-show-depends
                        Find layer dependencies from layer index.
    add-layer           Add a layer to bblayers.conf.
    remove-layer        Remove a layer from bblayers.conf.
    flatten             flatten layer configuration into a separate output
                        directory.
    show-layers         show current configured layers.
    show-overlayed      list overlayed recipes (where the same recipe exists
                        in another layer)
    show-recipes        list available recipes, showing the layer they are
                        provided by
    show-appends        list bbappend files and recipe files they apply to
    show-cross-depends  Show dependencies between recipes that cross layer
                        boundaries.
```

RECIPES

This section will introduce the concept of metadata and recipes and how they can be used to automate the building of packages

What is a recipe

- **A recipe is a set of instructions for building packages, including:**
  - Where to obtain the upstream sources and which patches to apply (this is called "*fetching*")
    - `SRC_URI`
  - Dependencies (on libraries or other recipes)
    - `DEPENDS, RDEPENDS`
  - Configuration/compilation options
    - `EXTRA_OECONF, EXTRA_OEMAKE`
  - Define which files go into what output packages
    - `FILES_*`

Recipe filename rules

- **The recipe filename have some important mandatory rules:**
  - The format of a recipe file name is
    - **<package-name>_<package-version>.bb**
  - There is only one single character '_' to separate PN from PV
  - The recipe name has to be all lowercase
    - example_1.0.bb
    - my-recipe-long-name_1.2.0.bb
  - The higher PV value is taken into account (if not overridden)
    - example_1.0.bb
    - example_2.0.bb
    - example_git.bb
  - It is possible to define a particular version with override
    - PREFERRED_VERSION_example = "1.0"

Example recipe : ethtool_3.15.bb

chris — ssh — 80×24

```
SUMMARY = "Display or change ethernet card settings"
DESCRIPTION = "A small utility for examining and tuning the settings of your eth
ernet-based network interfaces."
HOMEPAGE = "http://www.kernel.org/pub/software/network/ethtool/"
SECTION = "console/network"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=b234ee4d69f5fce4486a80fdaf4a4263 \
                    file://ethtool.c;beginline=4;endline=17;md5=c19b30548c582577
fc6b443626fc1216"

SRC_URI = "${KERNELORG_MIRROR}/software/network/ethtool/ethtool-${PV}.tar.gz \
           file://run-ptest \
           file://avoid_parallel_tests.patch \
           file://ethtool-uint.patch \
           "

SRC_URI[md5sum] = "7e94dd958bcd639aad2e5a752e108b24"
SRC_URI[sha256sum] = "562e3cc675cf5b1ac655cd060f032943a2502d4d59e5f278f02aae9256
2ba261"

inherit autotools ptest
RDEPENDS_${PN}-ptest += "make"

                                                    1,1           Top
```

What a recipe can do

- **Build one or more packages from source code**
  - Host tools, compiler, utilities
  - Bootloader, Kernel, etc
  - Libraries, interpretors, etc
  - Userspace applications
- Package Groups
- Full System Images

Recipe operators (Python)

```
A = “foo”                    (late assignment)
B ?= “0t”                    (default value)
C ??= “abc”                  (late default)
D := “xyz”                   (Immediate assignment)*

A .= “bar”          “foobar”     (append)
B =. “WO”           “W00t”       (prepend)
C += “def”          “abc def”    (append)
D =+ “uvw”              “uvw xyz”   (prepend)
```

**It is possible to use a "weaker" assignment than in the previous section by using the "??=" operator. This assignment behaves identical to "?=" except that the assignment is made at the end of the parsing process rather than immediately.*

More recipe operators

A = “foo”
A_append = “bar” “foobar”
B = “0t”
B_prepend = “WO” “W00t”

OVERRIDES = “os:arch:machine”

A = “abc”
A_os = “ABC” (Override)
A_append_arch = “def” (Conditional append)
A_prepend_os = “XYZ” (Conditional prepend)

Bitbake variables metadata

- **These are set automatically by bitbake**
  - **`TOPDIR`** – The build directory
  - **`LAYERDIR`** – Current layer directory
  - **`FILE`** – Path and filename of file being processed
- **Policy variables control the build**
  - **`BUILD_ARCH`** – Host machine architecture
  - **`TARGET_ARCH`** – Target architecture
  - And many others...

Build time metadata (1)

- **PN – Pakage name (“myrecipe”)**
- **PV – Package version (1.0)**
- **PR – Package Release (r0)**
- **P** = “${PN}-${PV}”
- **PF** = “${PN}-${PV}-${PR}”
- **FILE_DIRNAME** – Directory for FILE
- **FILESPATH** = "${FILE_DIRNAME}/${PF}:\
- ${FILE_DIRNAME}/${P}:\
- ${FILE_DIRNAME}/${PN}:\
- ${FILE_DIRNAME}/files:${FILE_DIRNAME}

Build time metadata (2)

- **TOPDIR – The build directory**
- **TMPDIR** = “${TOPDIR}/tmp”
- **WORKDIR** = ${TMPDIR}/work/${PF}”
- **S** = “${WORKDIR}/${P}” (Source dir)
- **B** = “${S}” (Build dir)
- **D** = “${WORKDIR}/${image}” (Destination dir)
- **DEPLOY_DIR** = “${TMPDIR}/deploy”
- **DEPLOY_DIR_IMAGE** = “${DEPLOY_DIR}/images”

Dependency metadata

- **Build time package variables**
  - **DEPENDS** – Build time package dependencies
  - **PROVIDES** = “${P} ${PF} ${PN}”
- **Runtime package variables**
  - **RDEPENDS** – Runtime package dependencies
  - **RRECOMMENDS** – Runtime recommended packages
  - Others…

Common metadata

- **Variables you commonly set**
  - **`SUMMARY`** – Short description of package/recipe
  - **`HOMEPAGE`** – Upstream web page
  - **`LICENSE`** – Licenses of included source code
  - **`LIC_FILES_CHKSUM`** – Checksums of license files at time of packaging (checked for change by build)
  - **`SRC_URI`** – URI of source code, patches and extra files to be used to build packages. Uses different fetchers based on the URI.
  - **`FILES`** – Files to be included in binary packages

Examining recipes : bc

- **Look at 'bc' recipe:**
- **Found in**

  poky/meta/recipes-extended/bc/bc_1.06.bb
  - Uses LIC_FILES_CHKSUM and SRC_URI checksums
  - Note the DEPENDS build dependency declaration indicating that this package depends on `flex` to build

Examining recipe : bc_1.06.bb

```
SUMMARY = "Arbitrary precision calculator language"
HOMEPAGE = "http://www.gnu.org/software/bc/bc.html"

LICENSE = "GPLv2+ & LGPLv2.1"
LIC_FILES_CHKSUM = "file://COPYING;md5=94d55d512a9ba36caa9b7df079bae19f \
                    file://COPYING.LIB;md5=d8045f3b8f929c1cb29a1e3fd737b499 \
file://bc/bcdefs.h;endline=31;md5=46dffdaf10a99728dd8ce358e45d46d8 \
file://dc/dc.h;endline=25;md5=2f9c558cdd80e31b4d904e48c2374328 \
file://lib/number.c;endline=31;md5=99434a0898abca7784acfd36b8191199"

SECTION = "base"
DEPENDS = "flex"

SRC_URI = " ${GNU_MIRROR}/bc/bc-${PV}.tar.gz \
            file://fix-segment-fault.patch "
SRC_URI[md5sum] = "d44b5dddebd8a7a7309aea6c36fda117"
SRC_URI[sha256sum] =
"4ef6d9f17c3c0d92d8798e35666175ecd3d8efac4009d6457b5c99cea72c0e33"

inherit autotools texinfo update-alternatives

ALTERNATIVE_${PN} = "dc"
ALTERNATIVE_PRIORITY = "100"
BBCLASSEXTEND = "native"
```

Building upon bbclass

- **Use inheritance for common design patterns**
- **Provide a class file (.bbclass) which is then inherited by other recipes (.bb files)**

```
inherit autotools
```

  - Bitbake will include the *autotools.bbclass* file
  - Found in a *classes* directory via the BBPATH

Examining recipes : Flac

- **Look at 'flac' recipe**
- **Found in**

  `poky/meta/recipes-multimedia/flac/flac_1.3.2.bb`

  - Inherits from both *autotools* and *gettext*
  - Customizes autoconf configure options (EXTRA_OECONF) based on "TUNE" features
  - Breaks up output into multiple binary packages
    - See PACKAGES var. This recipe produces additional packages with those names, while the FILES_* vars specify which files go into these additional packages

Examining recipe : flac_1.3.2.bb

```
SUMMARY = "Free Lossless Audio Codec"
DESCRIPTION = "FLAC stands for Free Lossless Audio Codec, a lossless audio
compression format."
HOMEPAGE = "https://xiph.org/flac/"
BUGTRACKER = "http://sourceforge.net/p/flac/bugs/"
SECTION = "libs"

LICENSE = "GFDL-1.2 & GPLv2+ & LGPLv2.1+ & BSD"
LIC_FILES_CHKSUM = "file://COPYING.FDL;md5=ad1419ecc56e060eccf8184a87c4285f \
 file://src/Makefile.am;beginline=1;endline=17;md5=09501c864f...65553129817ca \
 file://COPYING.GPL;md5=b234ee4d69f5f...80fdaf4a4263 \
 file://src/flac/main.c;beginline=1;endline=18;md5=09777e2...f13568d0beb81199 \
 file://COPYING.LGPL;md5=fbc093901857fcd118f065f900982c24 \
 file://src/plugin_common/all.h;beginline=1;endline=18;md5=f56cb4ba9a..3215271 \
 file://COPYING.Xiph;md5=b59c1b6d7fc0fb7965f821a3d36505e3 \
 file://include/FLAC/all.h;beginline=65;endline=70;md5=64474f2...28d8b8b25c983a48"

DEPENDS = "libogg"
SRC_URI = "http://downloads.xiph.org/releases/flac/${BP}.tar.xz"
SRC_URI[md5sum] = "454f1bfa3f93cc708098d7890d0499bd"
SRC_URI[sha256sum] =
"91cfc3ed61dc40f47f050a109b08610667d73477af6ef36dcad31c31a4a8d53f"
```

(con't next page)

... flac_1.3.2.bb

(con't from previous page)

```
CVE_PRODUCT = "libflac"
inherit autotools gettext
EXTRA_OECONF = "--disable-oggtest \
                --with-ogg-libraries=${STAGING_LIBDIR} \
                --with-ogg-includes=${STAGING_INCDIR} \
                --disable-xmms-plugin \
                --without-libiconv-prefix \
                ac_cv_prog_NASM="" \
                "
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "altivec", " --enable-altivec", "
--disable-altivec", d)}"
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "core2", " --enable-sse", "", d)}"
EXTRA_OECONF += "${@bb.utils.contains("TUNE_FEATURES", "corei7", " --enable-sse", "", d)}"

PACKAGES += "libflac libflac++ liboggflac liboggflac++"
FILES_${PN} = "${bindir}/*"
FILES_libflac = "${libdir}/libFLAC.so.*"
FILES_libflac++ = "${libdir}/libFLAC++.so.*"
FILES_liboggflac = "${libdir}/libOggFLAC.so.*"
FILES_liboggflac++ = "${libdir}/libOggFLAC++.so.*"
```

Grouping local metadata

- **Sometimes sharing metadata between recipes is easier via an *include file***

`include file.inc`

- Will include `.inc` file if found via BBPATH
- Can also specify an absolute path
- If not found, will continue without an error

`require file.inc`

- Same as an include
- Fails with an error if not found

Examining recipes : ofono

- **Look at 'ofono' recipe(s):**
- **Found in**

  `poky/meta/recipes-connectivity/ofono/ofono_1.19.bb`

  - Splits recipe into common `.inc` file to share common metadata between multiple recipes
  - Sets a conditional build configuration options through the `PACKAGECONFIG` var based on a `DISTRO_FEATURE` (in the `.inc` file)
  - Sets up an init service via do_install_append()
  - Has a `_git` version of the recipe (not shown)

Examining recipe : ofono_1.19.bb

```
require ofono.inc

SRC_URI  = "\
  ${KERNELORG_MIRROR}/linux/network/${BPN}/${BP}.tar.xz \
  file://ofono \
"
SRC_URI[md5sum] = "a5f8803ace110511b6ff5a2b39782e8b"
SRC_URI[sha256sum] =
"a0e09bdd8b53b8d2e4b54f1863ecd9aebe4786477a6cbf8f655496e8e
db31c81"

CFLAGS_append_libc-uclibc = " -D_GNU_SOURCE"
```

Examining recipe : ofono.inc

```
HOMEPAGE = "http://www.ofono.org"
SUMMARY  = "open source telephony"
DESCRIPTION = "oFono is a stack for mobile telephony devices on Linux.
oFono supports speaking to telephony devices through specific drivers, or
with generic AT commands."
LICENSE  = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=eb723b61539feef013de476e68b5c50a \

file://src/ofono.h;beginline=1;endline=20;md5=3ce17d5978ef3445def265b98899c
2ee"

inherit autotools pkgconfig update-rc.d systemd bluetooth

DEPENDS  = "dbus glib-2.0 udev mobile-broadband-provider-info"

INITSCRIPT_NAME = "ofono"
INITSCRIPT_PARAMS = "defaults 22"

PACKAGECONFIG ??= "\
    ${@bb.utils.filter('DISTRO_FEATURES', 'systemd', d)} \
    ${@bb.utils.contains('DISTRO_FEATURES', 'bluetooth', 'bluez', '', d)} \
    "
PACKAGECONFIG[systemd] = "--with-systemdunitdir=$
{systemd_unitdir}/system/,--with-systemdunitdir="
PACKAGECONFIG[bluez] = "--enable-bluetooth, --disable-bluetooth, ${BLUEZ}"
```

(con't next page)

ofono.inc

```
(con't from previous page)
EXTRA_OECONF += "--enable-test"

SYSTEMD_SERVICE_${PN} = "ofono.service"

do_install_append() {
  install -d ${D}${sysconfdir}/init.d/
  install -m 0755 ${WORKDIR}/ofono ${D}${sysconfdir}/init.d/ofono

  # Ofono still has one test tool that refers to Python 2 in the shebang
  sed -i -e '1s,#!.*python.*,#!${bindir}/python3,' ${D}$
{libdir}/ofono/test/set-ddr

}

PACKAGES =+ "${PN}-tests"
RDEPENDS_${PN} += "dbus"
RRECOMMENDS_${PN} += "kernel-module-tun mobile-broadband-provider-info"

FILES_${PN} += "${systemd_unitdir}"
FILES_${PN}-tests = "${libdir}/${BPN}/test"
RDEPENDS_${PN}-tests = "python3 python3-pygobject python3-dbus"
```


Add the Koan training layers

- Let’s add another layer to the system
- This is pre-defined *meta-training* layer

```
cd /workdir/poky
git clone https://github.com/koansoftware/meta-training.git
```

- Add your layer to *bblayers.conf*

```
bitbake-layers add-layer \
            /workdir/poky/build/meta-training
```

meta-training layer

```
yocto/build$ tree meta-training
meta-training/
|--COPYING.MIT              (The license file)
|--README                   (Starting point for README)
|--conf
|   `--layer.conf           (Layer configuration file)
`--recipes-example          (A grouping of recipies)
    `--example              (The example package)
        |--example-0.1      (files for v. 0.1 of example)
        |   `--helloworld.c
        `--example_0.1.bb   (The example recipe)
```

example_1.0.bb

```
# example_1.0.bb

SUMMARY = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://$
{COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://helloworld.c"

# Avoid a compilation error: No GNU_HASH in the elf binary
TARGET_CC_ARCH += "${LDFLAGS}"

S = "${WORKDIR}"

do_compile() {
        ${CC} helloworld.c -o helloworld
}

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

helloworld.c

```
# helloworld.c

#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");

    return 0;
}
```


Build the new recipe

- **You can now build the new recipe**
  $ bitbake example

- **This will now build the *example_0.1.bb* recipe which is found in**
  meta-training/recipes-example/example/example_0.1.bb

Note: Build fails w/o ${CFLAGS} and ${LDFLAGS} meanwhile (QA-error) in the recipe.

Add example recipe to the image

- **Add the new recipe example to the final image**
  - Configure build by editing local.conf
  - `/workdir/poky/build/conf/local.conf`
  - Add the name of your recipe
  - **IMAGE_INSTALL_append = " example"**
- **Then rebuild the final image**
  - bitbake core-image-minimal
- **And test it running QEMU**
  - runqemu qemuarm slirp nographic

yocto PROJECT

WHEN THINGS GO WRONG

Some useful tools to help guide you when something goes wrong

Bitbake environment

- **Each recipe has its own environment which contains all the variables and methods required to build that recipe**
- **You've seen some of the variables already**
  - DESCRIPTION, SRC_URI, LICENSE, S, LIC_FILES_CHKSUM, do_compile(), do_install()
- **Example**
  - S = "${WORKDIR}"
  - What does this mean?

Examine a recipe's environment (1)

➢ **To view a recipe's environment**

$ bitbake -e myrecipe

➢ **Where is the source code for this recipe"**

$ bitbake -e virtual/kernel | grep "^S="

S="${HOME}/yocto/build/tmp/work-shared/qemuarm/kernel-source"

➢ **What file was used in building this recipe?**

$ bitbake -e netbase | grep "^FILE="

FILE="${HOME}/yocto/poky/meta/recipes-core/netbase/netbase_5.3.bb"

Examine a recipe's environment (2)

➢ What is this recipe's full version string?

$ bitbake -e netbase | grep "^PF="
PF="netbase-1_5.3-r0"

➢ Where is this recipe's BUILD directory?

$ bitbake -e virtual/kernel | grep "^B="
B="${HOME}/yocto/build/tmp/work/qemuarm-poky-linux-\
gnueabi/linux-yocto/3.19.2+gitAUTOINC+9e70b482d3\
_473e2f3788-r0/linux-qemuarm-standard-build"

➢ What packages were produced by this recipe?

$ bitbake -e virtual/kernel | grep "^PACKAGES="
PACKAGES="kernel kernel-base kernel-vmlinux kernel-image \ kernel-dev
kernel-modules kernel-devicetree"

Bitbake log files

- **Every build produces lots of log output for diagnostics and error chasing**
  - Verbose log of bitbake console output:
    - Look in …/tmp/log/cooker/<machine>

```
$ cat tmp/log/cooker/qemuarm/20160119073325.log  | grep 'NOTE:.*task.*Started'

NOTE: recipe hello-1.0.0-r0: task do_fetch: Started
NOTE: recipe hello-1.0.0-r0: task do_unpack: Started
NOTE: recipe hello-1.0.0-r0: task do_patch: Started
NOTE: recipe hello-1.0.0-r0: task do_configure: Started
NOTE: recipe hello-1.0.0-r0: task do_populate_lic: Started
NOTE: recipe hello-1.0.0-r0: task do_compile: Started
NOTE: recipe hello-1.0.0-r0: task do_install: Started
NOTE: recipe hello-1.0.0-r0: task do_populate_sysroot: Started
NOTE: recipe hello-1.0.0-r0: task do_package: Started
NOTE: recipe hello-1.0.0-r0: task do_packagedata: Started
NOTE: recipe hello-1.0.0-r0: task do_package_write_rpm: Started
NOTE: recipe hello-1.0.0-r0: task do_package_qa: Started
NOTE: recipe ypdd-image-1.0-r0: task do_rootfs: Started
```


Bitbake per-recipe log files (1)

- **Every recipe produces lots of log output for diagnostics and debugging**
- **Use the Environment to find the log files for a given recipe:**

```
$ bitbake -e hello | grep “^T=”
T="${HOME}yocto/build/tmp/work/armv5e-poky-linux-gnueabi/hello/1.0.0-r0/temp"
```

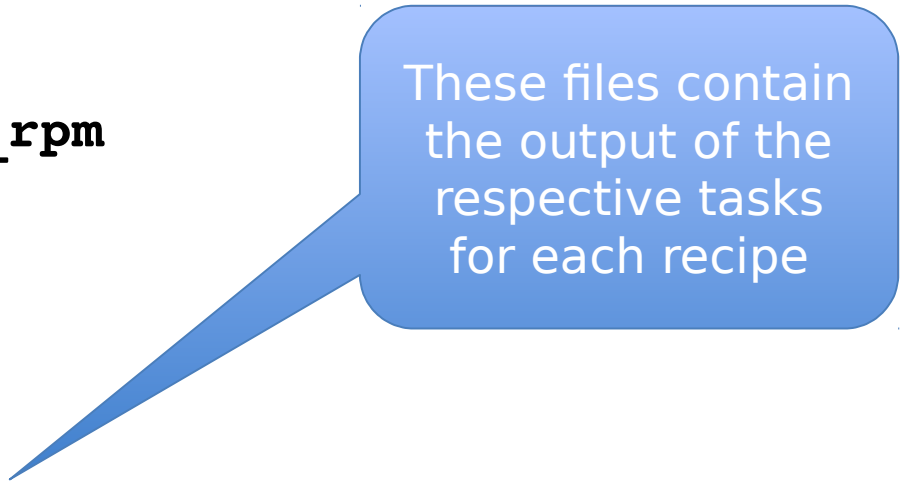
- **Each task that runs for a recipe produces "log" and "run" files in ${WORKDIR}/temp**

Bitbake per-recipe log files (2)

```
$ cd ${T} (This means T as printed out in the previous slide)

$ find . -type l -name 'log.*'

./log.do_package_qa
./log.do_package_write_rpm
./log.do_package
./log.do_fetch
./log.do_populate_lic
./log.do_install
./log.do_configure
./log.do_unpack
./log.do_populate_sysroot
./log.do_compile
./log.do_packagedata
./log.do_patch
```



Bitbake per-recipe log files (3)

```
$ cd ${T} (This means T as printed out in the previous slide)

$ find . -type l -name 'run.*'

./run.do_fetch
./run.do_patch
./run.do_configure
./run.do_populate_sysroot
./run.do_package_qa
./run.do_unpack
./run.do_compile
./run.do_install
./run.do_packagedata
./run.do_populate_lic
./run.do_package
./run.do_package_write_rpm
```

These files contain the commands executed which produce the build results

Debugging recipes

➢ **Enable bitbake debug messages [D, DD, DDD]**

$ bitbake -DDD <packagename>

➢ **Use the power of devshell**

$ bitbake -c devshell <packagename>

➢ **Add messages in the recipe**

```
inherit logging
do_install() {
    bbwarn "----This is a debug message-----"
    }
```

Look at classes/logging.bbclass

IMAGES

This section will introduce the concept of images; recipes which build embedded system images

What is an image

- Building an image creates an entire Linux distribution from source
  - Compiler, tools, libraries
  - BSP: Bootloader, Kernel
  - Root filesystem:
    - Base OS
    - services
    - Applications
    - etc

Extending an image

- You often need to create your own Image recipe in order to add new packages or functionality
- With Yocto/OpenEmbedded it is always preferable to extend an existing recipe or inherit a class
- The simplest way is to inherit the core-image bbclass
- You add packages to the image by adding them to `IMAGE_INSTALL`

A simple image recipe

- **Create an `images` directory**

  $ mkdir -p /workdir/yocto/build/meta-linuxlab/recipes-images/images

- **Create the image recipe**

  $ vi /workdir/yocto/build/meta-linuxlab/recipes-images/images/linuxlab-image.bb

```
DESCRIPTION = "A core image for LINUXLAB"
LICENSE = "MIT"

# Core files for basic console boot
IMAGE_INSTALL = "packagegroup-core-boot"

# Add our desired packages
IMAGE_INSTALL += "example psplash"

inherit core-image

IMAGE_ROOTFS_SIZE ?= "8192"
```

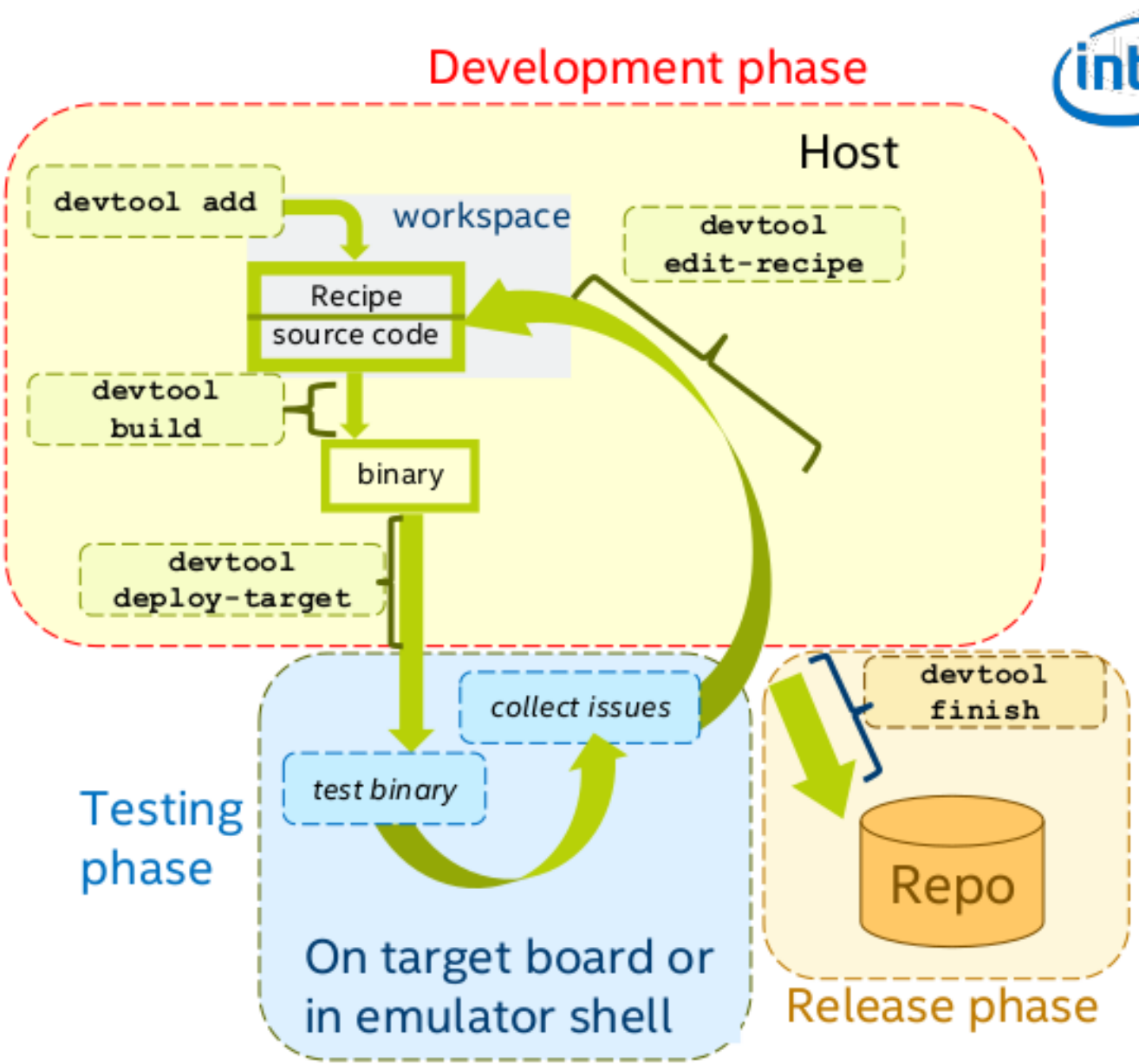
DEVTOOL

This section will introduce the devtool

Devtool overview

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

devtool workflow



Devtool usage (1)

- There are three ways of creating a new **devtool** project:
- To create a new recipe:

```
devtool add <recipe> <fetchURI>
```

  - Where recipe is the recipe's name
  - fetchuri can be a local path or a remote URI
- To modify **an existing** recipe:

```
devtool modify <recipe>
```

- To upgrade a given recipe:

```
devtool upgrade -V <version><recipe>
```

  - Where version is the new version of the upstream package

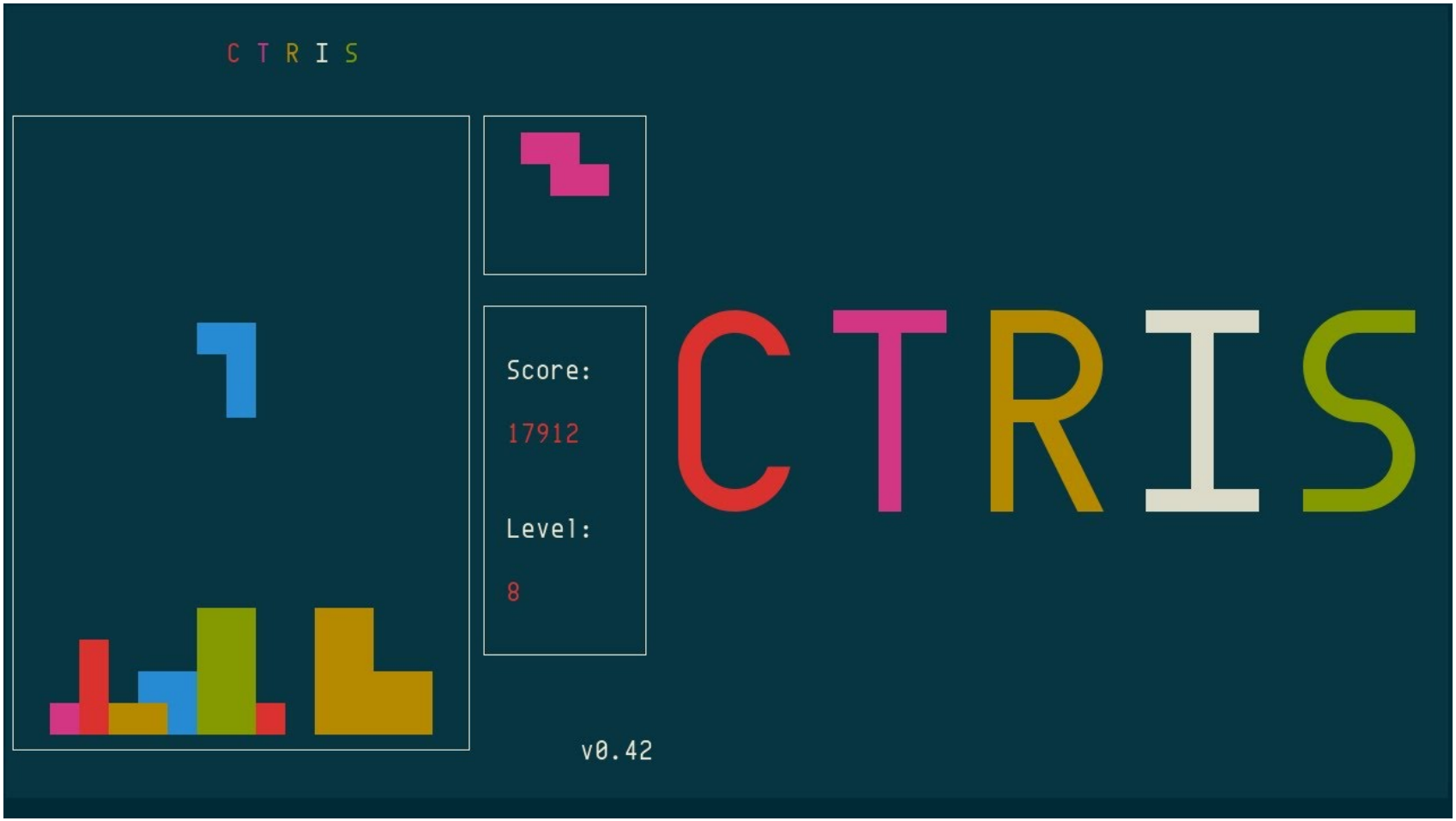
Devtool usage (2)

- Once a **devtool** project is started, commands can be issued:
- Edit recipe in a text editor (as defined by the EDITOR environment variable):
  `devtool edit-recipe <recipe>`
- Build the given recipe:
  `devtool build <recipe>`
- Build an image with the additional **devtool** recipes' packages:
  `devtool build-image <imagename>`

Devtool usage (3)

- Upload the recipe’s package on target, which is a live running target with an SSH server running (user@address):
  `devtool deploy-target <recipe> <target>`
- Generate patches from git commits made locally:
  `devtool update-recipe <recipe>`
- Remove recipe from the control of **devtool**:
  `devtool reset <recipe>`
  - Standard layers and remote sources are used again as usual

Let’s have fun



NOTE: enable the extended characters in the QEMU target with

```
export TERM=xterm
```

Devtool in action

- Create a new recipe using **devtool**

```
devtool add ctris <fetchURI>
```

```
devtool add ctris https://github.com/koansoftware/ctris
```

- Modify the recipe with **devtool**

```
devtool modify ctris
```

- Rebuild the final imagewith **devtool**

```
devtool build-image core-image-minimal
```

- Finalize the recipe in an existing layer *(and delete it from the devtool workspace)*

```
devtool finish -f ctris /workdir/poky/meta-linuxlab
```

Use the editor of the host Instead of this

- *Continued next page...*

Last steps

- After the devtool finish (from now) ctris can be built only using **bitbake** as usual

```
bitbake ctris
```

- Remember to add the recipe name in the **local.conf** (or in the image recipe) before rebuilding the image

```
IMAGE_INSTALL_append = “ ctris”
```

- Rebuild the image

```
bitbake core-image-minimal
```

ctris_git.bb

```
# ctris.git.bb - Recipe created by recipetool
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://LICENSE;md5=2c1c00f9d3ed9e24fa69b932b7e7aff2 \
file://COPYING;md5=0636e73ff0215e8d672dc4c32c317bb3"

SRC_URI = "git://github.com/koansoftware/ctris;protocol=https"

# Modify these as desired
PV = "1.0+git${SRCPV}"
SRCREV = "3e0b8bc914cf47c1885e6e168106579a96f16e9b"

S = "${WORKDIR}/git"

# Avoid a compilation error: No GNU_HASH in the elf binary
TARGET_CC_ARCH += "${LDFLAGS}"

DEPENDS = "ncurses"
EXTRA_OEMAKE += "-e"

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


APPLICATION DEVELOPMENT

This section will introduce the cross compiler generated by Yocto

Cross compiler

- Yocto can create a re-distributable cross-compiler

```
bitbake meta-toolchain
```

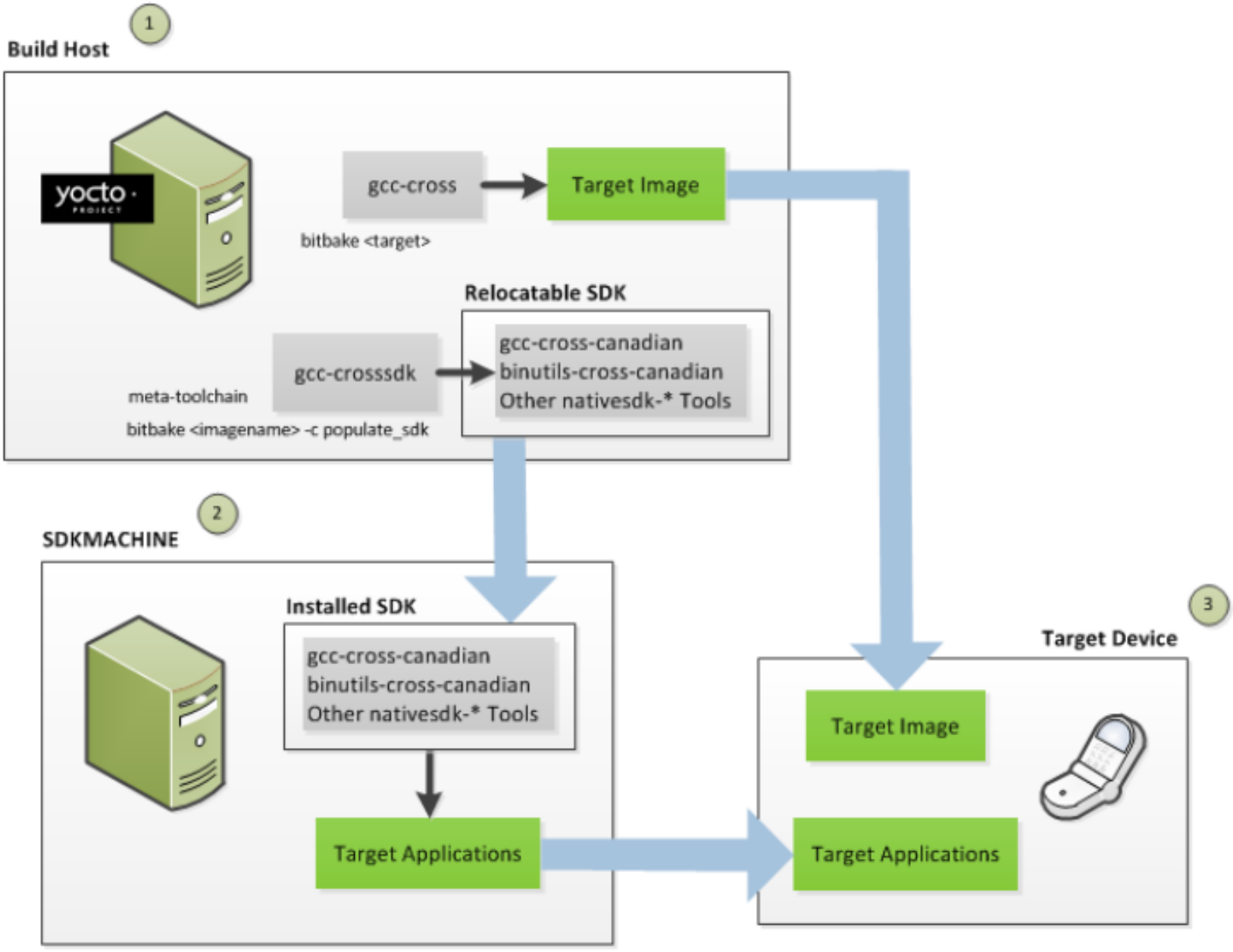
- Or a complete SDK for your target

```
bitbake -c populate_sdk <image-name>
```

- Or even an SDK for Qt5

```
bitbake meta-toolchain-qt5
```

Cross-Development Toolchain



Cross compiler

- Install it on any linux distribution

```
$ cd $HOME/poky/build/tmp/deploy/sdk

$ ./poky-glibc-x86_64-meta-toolchain-
cortexa8hf-vfp-neon-toolchain-2.4.sh
```

- Once installed you can use it setting the build environment

```
$ source /opt/poky/2.4/environment-setup-
cortexa8hf-vfp-neon-poky-linux-gnueabi
```

Thank you!

http://yoctoproject.org

Questions?

Work with us

LAB
BUILDING SMARTER DEVICES

KOAN services

Embedded Linux Training

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux
Yocto Project
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
BSP creation fir new boards
System optimization
Linux kernel drivers
Application and interface development

Consulting

Help in decision making
System architecture
Identification of suitable technologies
Managing licensing requirements
System design and performance review

Technical Support

Development tool and application support
Issue investigation and solution follow-up with mainstream developers
Help getting started

Follow us on LinkedIn

http://koansoftware.com