

YOCTO PROJECT



Sergio Prado
sergio.prado@e-labworks.com
<https://www.linkedin.com/in/sprado>

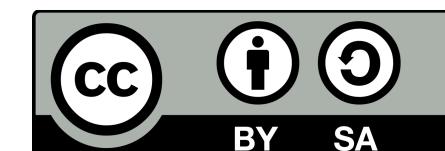
ABOUT THIS DOCUMENT

- Yocto Project v1.2.0 training slides.
- The most current version of the slides is available on the Embedded Labworks website.

<https://e-labworks.com/training/en/ypr/slides>

- This document is made available under the Commons Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0). The full terms of this license are available at the link below:

<https://creativecommons.org/licenses/by-sa/4.0/>



ABOUT THE INSTRUCTOR

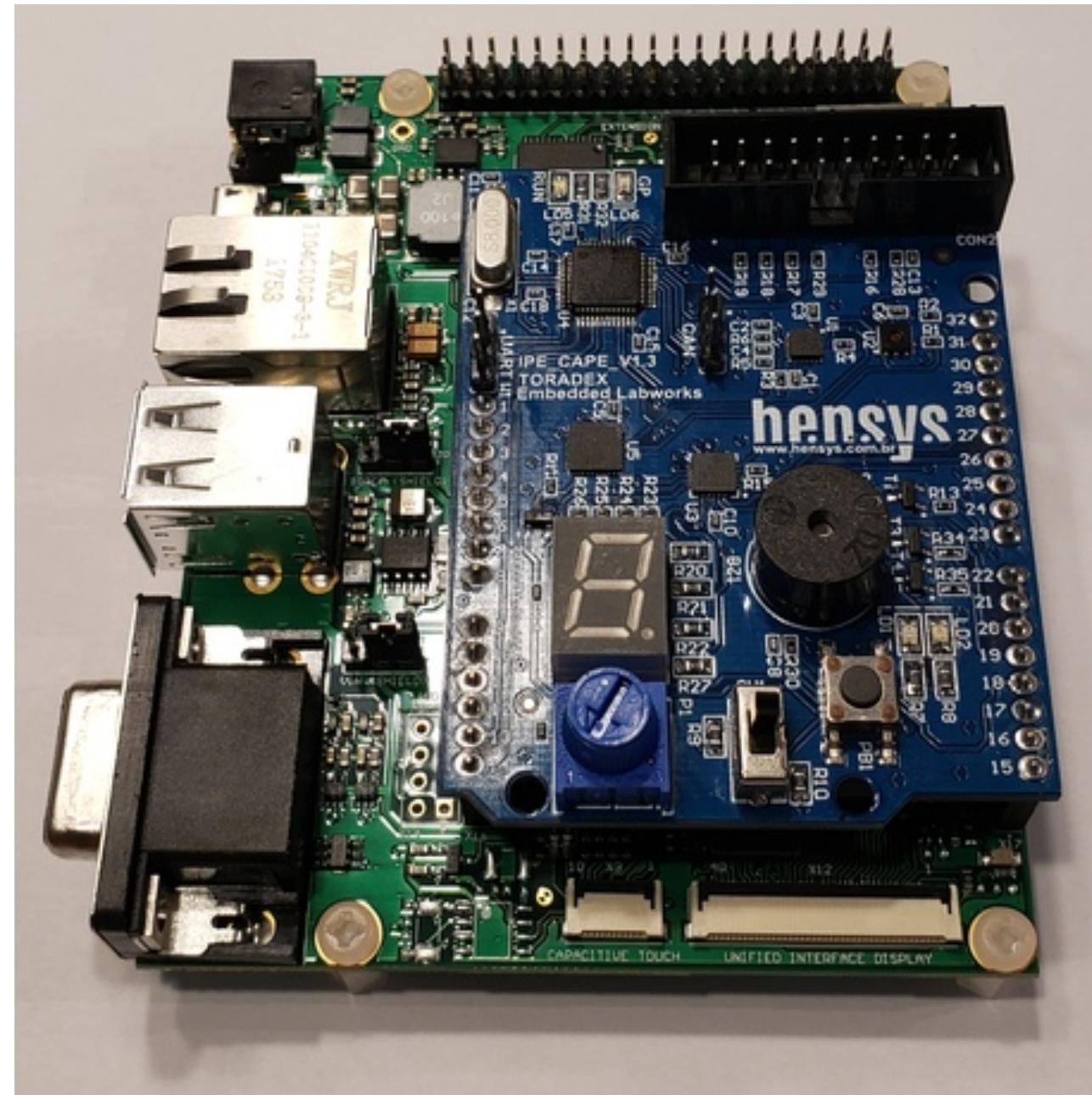
- More than 25 years of experience in software development for embedded systems, working mainly with BSP development in projects with embedded Linux, embedded Android and real-time operating systems.
- Consultant & Trainer at Embedded Labworks, providing consulting, training and software development for embedded systems.
<https://e-labworks.com/en>
- Active in the embedded systems community in Brazil, one of the creators of the Embarcados website and blogger.
<https://sergioprado.blog>
- Free and open source software contributor, including projects like Buildroot, Yocto Project and the Linux kernel.



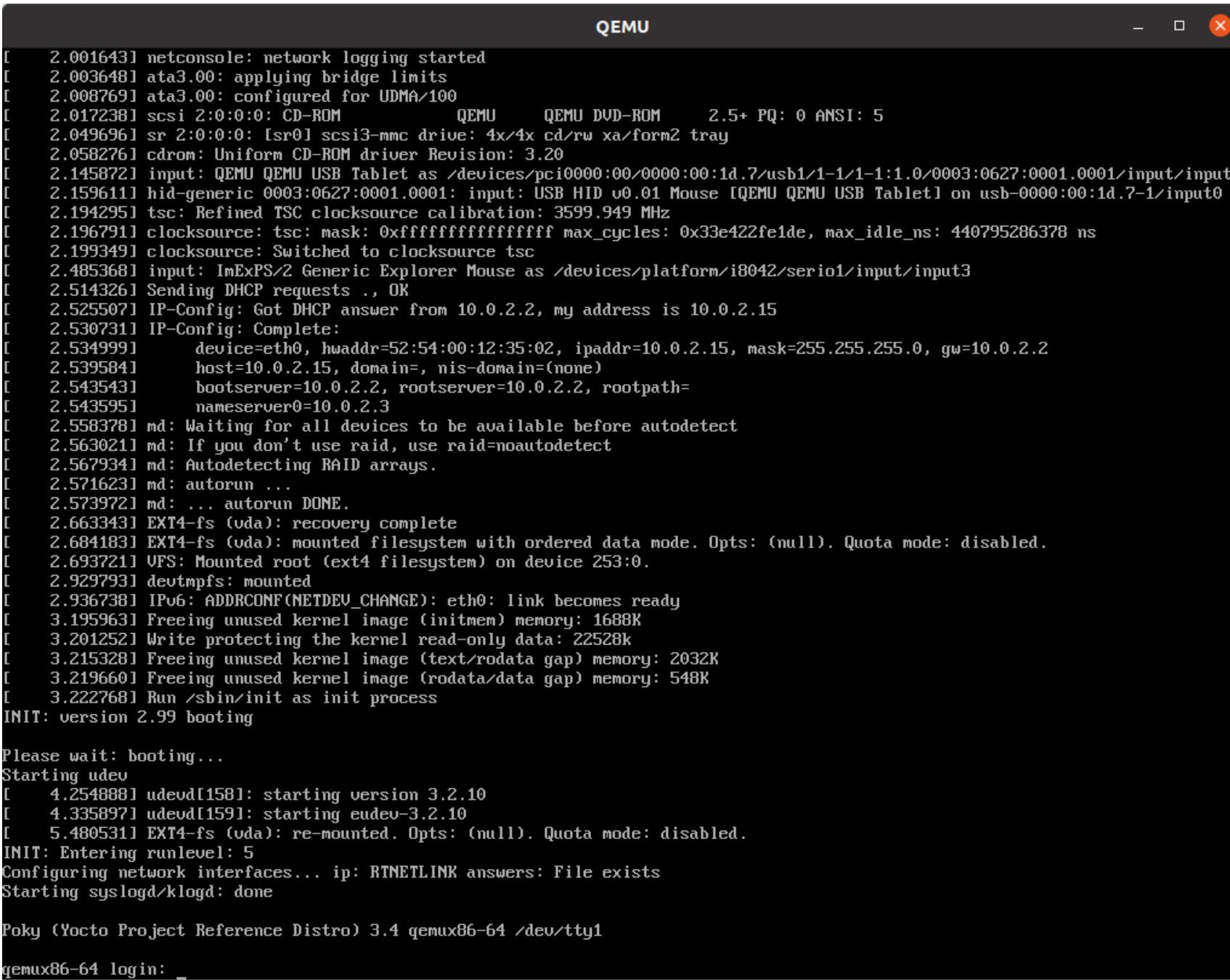
TRAINING SCHEDULE

- **DAY 1:** Introduction to the Yocto Project, first steps with Poky, overview of the OpenEmbedded build system and the BitBake tool, introduction to the concept of layers and recipes.
- **DAY 2:** Advanced recipe concepts, rootfs and image customization.
- **DAY 3:** Development of BSP layers, generation and use of SDKs, additional tools and resources.

TARGET: COLIBRI IMX6



TARGET: QEMU



The screenshot shows a terminal window titled "QEMU". The window contains a detailed log of a Linux kernel boot process. The log includes messages about network logging, device configuration (like CD-ROM and USB drives), network interface setup (DHCP), and file system mounting (EXT4). It also shows the transition to runlevel 5 and the start of networking. The final line of the log is "Poky (Yocto Project Reference Distro) 3.4 qemux86-64 /dev/tty1", followed by a prompt "qemux86-64 login: _".

```
[ 2.001643] netconsole: network logging started
[ 2.003648] ata3.00: applying bridge limits
[ 2.008769] ata3.00: configured for UDMA/100
[ 2.017238] scsi 2:0:0:0: CD-ROM           QEMU   QEMU DVD-ROM    2.5+ PQ: 0 ANSI: 5
[ 2.049696] sr 2:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[ 2.058276] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 2.145872] input: QEMU QEMU USB Tablet as /devices/pci0000:00/0000:00:1d.7/usb1/1-1/1-1:1.0/0003:0627:0001.0001/input/input4
[ 2.159611] hid-generic 0003:0627:0001.0001: input: USB HID v0.01 Mouse [QEMU QEMU USB Tablet] on usb-0000:00:1d.7-1/input0
[ 2.194295] tsc: Refined TSC clocksource calibration: 3599.949 MHz
[ 2.196791] clocksource: tsc: mask: 0xfffffffffffffff max_cycles: 0x33e422fe1de, max_idle_ns: 440795286378 ns
[ 2.199349] clocksource: Switched to clocksource tsc
[ 2.485368] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
[ 2.514326] Sending DHCP requests .., OK
[ 2.525507] IP-Config: Got DHCP answer from 10.0.2.2, my address is 10.0.2.15
[ 2.530731] IP-Config: Complete:
[ 2.534999]   device=eth0, hwaddr=52:54:00:12:35:02, ipaddr=10.0.2.15, mask=255.255.255.0, gw=10.0.2.2
[ 2.539584]   host=10.0.2.15, domain=, nis-domain=(none)
[ 2.543543]   bootserver=10.0.2.2, rootserver=10.0.2.2, rootpath=
[ 2.543595]   nameserver0=10.0.2.3
[ 2.558378] md: Waiting for all devices to be available before autodetect
[ 2.563021] md: If you don't use raid, use raid=noautodetect
[ 2.567934] md: Autodetecting RAID arrays.
[ 2.571623] md: autorun ...
[ 2.573972] md: ... autorun DONE.
[ 2.663343] EXT4-fs (uda): recovery complete
[ 2.684183] EXT4-fs (uda): mounted filesystem with ordered data mode. Opts: (null). Quota mode: disabled.
[ 2.693721] UFS: Mounted root (ext4 filesystem) on device 253:0.
[ 2.929793] devtmpfs: mounted
[ 2.936738] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 3.195963] Freeing unused kernel image (initmem) memory: 1688K
[ 3.201252] Write protecting the kernel read-only data: 22528K
[ 3.215328] Freeing unused kernel image (text/rodata gap) memory: 2032K
[ 3.219660] Freeing unused kernel image (rodata/data gap) memory: 548K
[ 3.222768] Run /sbin/init as init process
INIT: version 2.99 booting

Please wait: booting...
Starting udev
[ 4.254888] udevd[158]: starting version 3.2.10
[ 4.335897] udevd[159]: starting eudev-3.2.10
[ 5.400531] EXT4-fs (uda): re-mounted. Opts: (null). Quota mode: disabled.
INIT: Entering runlevel: 5
Configuring network interfaces... ip: RTNETLINK answers: File exists
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 3.4 qemux86-64 /dev/tty1
qemux86-64 login: _
```

PREREQUISITES

- Basic knowledge of GNU/Linux command line tools (*cat, echo, vim, grep, find*, etc).
- Embedded Linux architecture (toolchain, bootloader, kernel, rootfs).
- Compilers and build tools (GCC, make, cmake, autotools, etc).
- Version control with Git.
- Basic programming in C and C++.

LABS ENVIRONMENT

```
$ tree -L 2 /opt/labs/
/opt/labs/
├── dl
│   ├── appsecret.tar.bz2
│   ├── fping-5.0.tar.gz
│   └── layers.tar.bz2
...
├── docs
│   ├── agenda.html
│   ├── answers.html
│   ├── install.html
│   ├── labs.html
│   ├── slides.pdf
│   └── version.txt
└── ex
    ├── downloads
    └── sstate-cache
└── tools
    ├── build.conf
    ├── build.sh
    ├── layers.conf
    ├── prepare.cfg
    └── prepare.sh
```

DURING THE TRAINING...

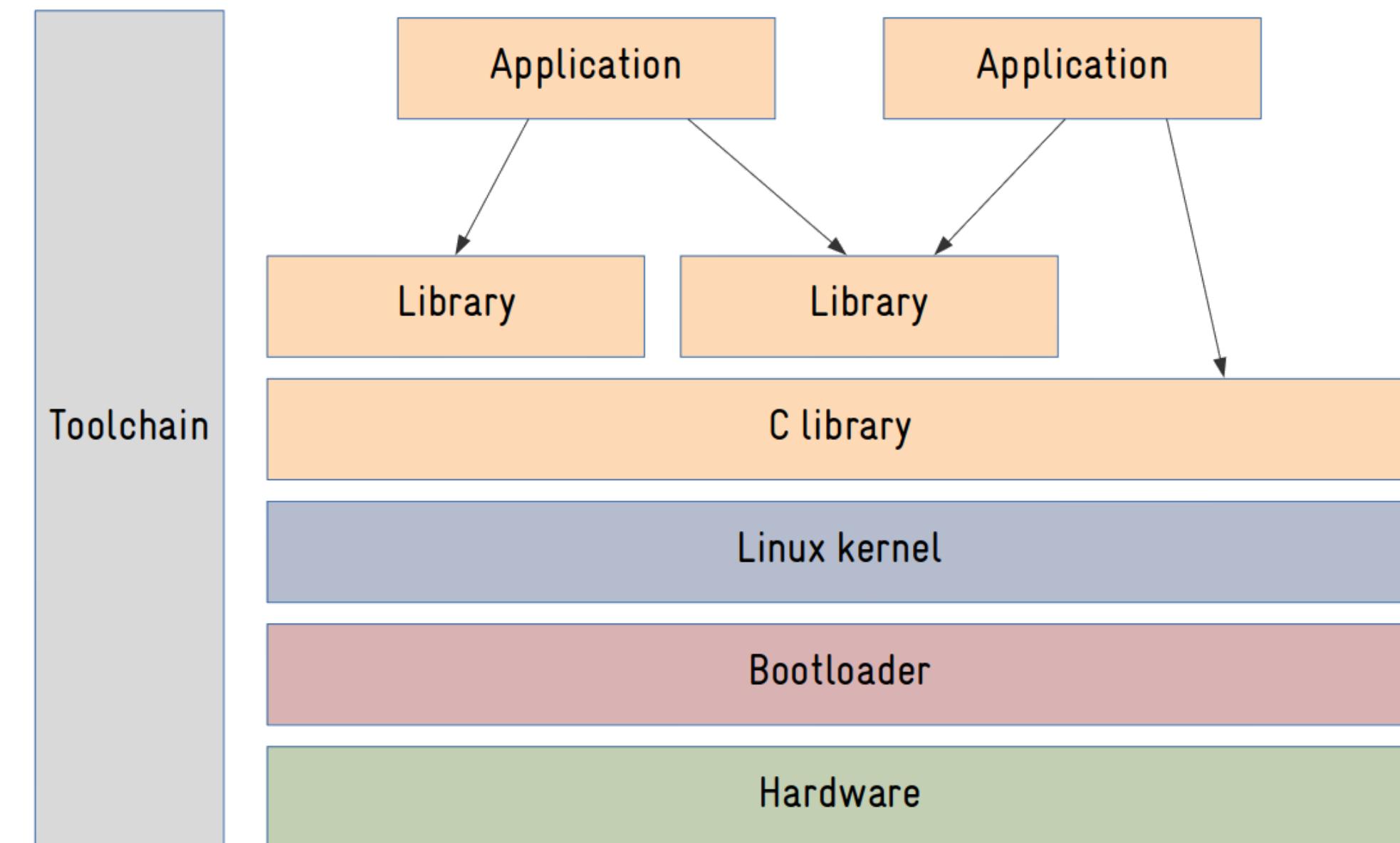
- Ask...
- Express your point of view...
- Share your experience...
- Help...
- Participate!

YOCTO PROJECT

INTRODUCTION TO THE YOCTO PROJECT



EMBEDDED LINUX



SYSTEM COMPONENTS

- **Hardware:** target platform.
- **Bootloader:** basic hardware initialization, loads and runs the Linux kernel.
- **Linux Kernel:** Manages the hardware (CPU, memory, I/O).
- **Rootfs:** Root file system.
 - C library: Operating system's API, kernel and user applications interface.
 - User libraries and applications.
- **Toolchain:** set of tools to build operating system artifacts (bootloader, kernel, rootfs).

WORKING WITH EMBEDDED LINUX

- We can adopt 3 different approaches when developing an embedded Linux system:
 - Use a third-party Linux distribution.
 - Build a Linux system/distribution manually.
 - Build a Linux system/distribution using a build system.

SOLUTION 1: THIRD-PARTY DISTRIBUTION

- There are Linux distributions sold as products by companies such as MontaVista and WindRiver.
<https://www.mvista.com/products>
- There are also several open source options, including Android, Debian, Ubuntu, Fedora IoT, etc.
- Advantages of using a third-party distribution:
 - Simple to use.
 - Easy to extend its functionalities.
 - Working (but limited) development environment.



THIRD-PARTY DISTRIBUTION DRAWBACKS

- Not optimized for the target platform, impacting resource consumption (CPU, memory, storage, power, etc).
- High boot time.
- Little flexibility for customization.
 - The software components (init system, daemons, etc) are very coupled and dependent on each other.
- Possible impact on security.



THIRD-PARTY DISTRIBUTION DRAWBACKS

- License compliance is a challenge, especially when using copyleft code (GPL and similar).
- Some features are difficult to implement (remote update, secure boot, etc).
- Customizations are not easily reproducible.
- Extra maintenance work on future updates.

SOLUTION 2: MANUAL PROCESS

- Generating a Linux system manually allows full control over the tools and components of the system.
- However, generating a complete Linux system manually is an extremely labor-intensive, time-consuming, difficult to reproduce and error-prone activity.
- For the more adventurous:
<https://www.linuxfromscratch.org/>

SOLUTION 3: BUILD SYSTEM

- A build system allows generating a complete and customized Linux system (or distribution) for the target platform.
- Provides a set of tools that automate the build of all operating system components (toolchain, bootloader, kernel, rootfs).
- Usually contains a large set of pre-configured packages to be enabled and used by the system.
- Facilitates configuring, extending and maintaining the Linux system over time.

ADVANTAGES OF A BUILD SYSTEM

- Flexibility, providing full control over all operating system components.
- Allows the generation of an image optimized for the target platform.
- Greater control over the use of hardware resources (CPU, memory, storage, power, etc).
- Make it possible to have shorter boot times.

ADVANTAGES OF A BUILD SYSTEM

- Facilitates the implementation of features such as secure boot and remote update.
- Helps to manage software license compliance.
- Builds are controlled and reproducible.
- Configuration files and other metadata used to create the Linux distribution can be versioned, making it easier to control and maintain the operating system over time.

OPEN SOURCE BUILD SYSTEMS

- Buildroot, developed by the community:
<https://www.buildroot.org>
- PTXdist, maintained by Pengutronix:
<https://www.ptxdist.org>
- OpenWRT, build system (and distribution) focused on network devices:
<https://openwrt.org>
- OpenEmbedded (used by the Yocto Project), more flexible and also more complex:
<https://www.openembedded.org>

WHAT IS THE YOCTO PROJECT?

- Collaborative project that provides a set of tools to help create custom Linux distributions for embedded devices.
<https://www.yoctoproject.org>
- Created in 2010 by several technology companies, hardware manufacturers and embedded Linux software solution providers.
- Under the governance of the Linux Foundation, ensuring the project's independence from any member of the organization.

YOCTO PROJECT MEMBERS



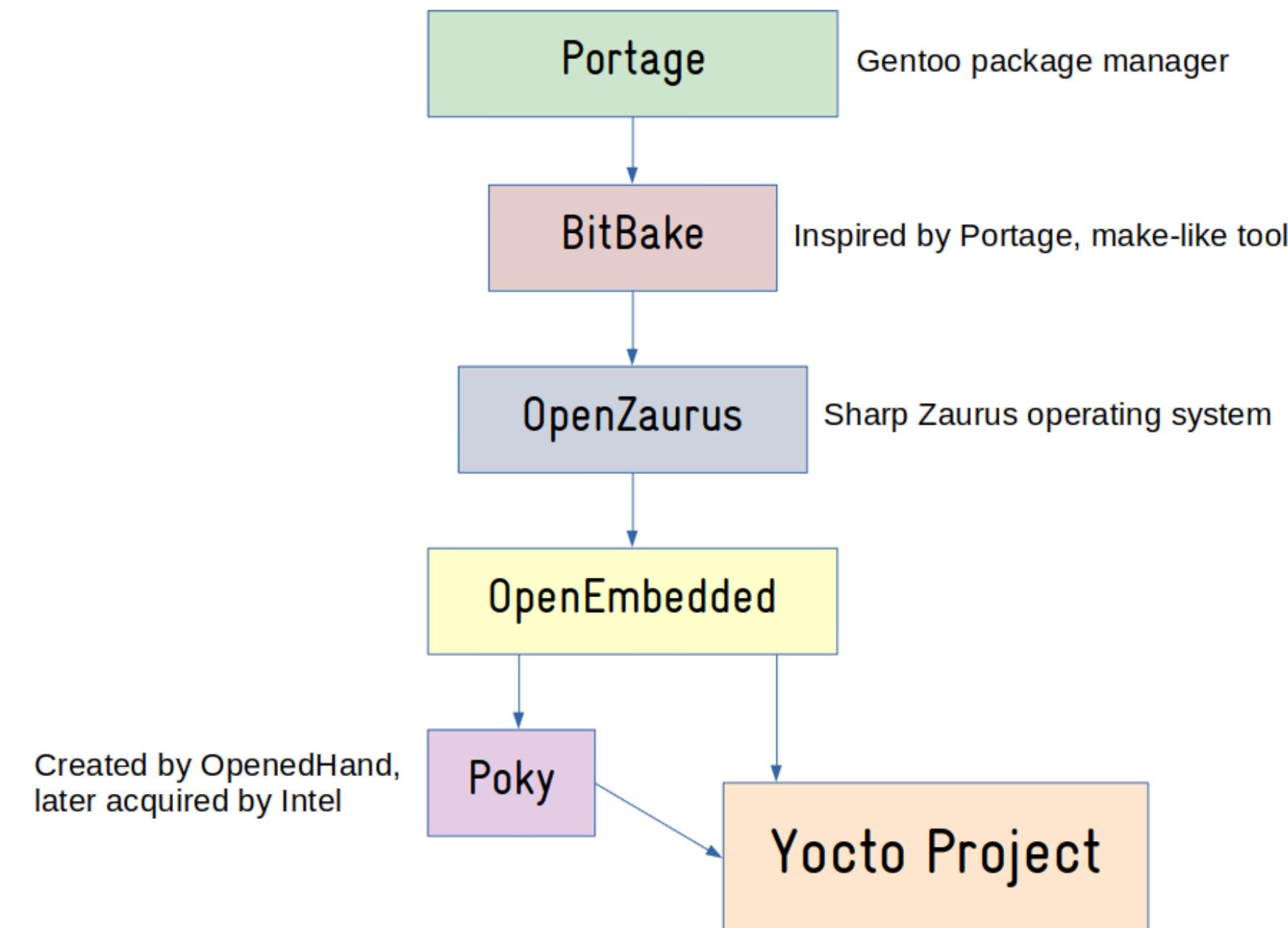
PROJECTS

- These are some of the projects that are part of the Yocto Project:
 - Openembedded-Core
 - BitBake
 - Poky
 - Matchbox
 - AutoBuilder
 - Toaster
- A more complete list of projects is available at the link below:
<https://www.yoctoproject.org/software-overview/project-components/>

THE YOCTO PROJECT IS NOT...

- The Yocto Project is not a tool or build system!
 - It is an "umbrella" project composed of several open-source tools and projects to build Linux distributions for embedded devices.
 - Under the hood, it uses OpenEmbedded (BitBake, OpenEmbedded-Core) as its build system.
- Yocto Project is not a Linux distribution.
 - But can create a Linux distribution for you!

A LITTLE BIT OF HISTORY



RELEASE CYCLE

- Details on the schedule, features, development and release process:
<https://wiki.yoctoproject.org/wiki/Planning#Yocto>
- A new Yocto Project version is expected to be released every 6 months (typically in April and October).
<https://wiki.yoctoproject.org/wiki/Releases>
- Patches to fix critical bugs and security flaws are applied also in a previous version and LTS versions.
- In this training we will use Yocto Project 4.0.1 (Kirkstone).

FEATURES AND ADVANTAGES

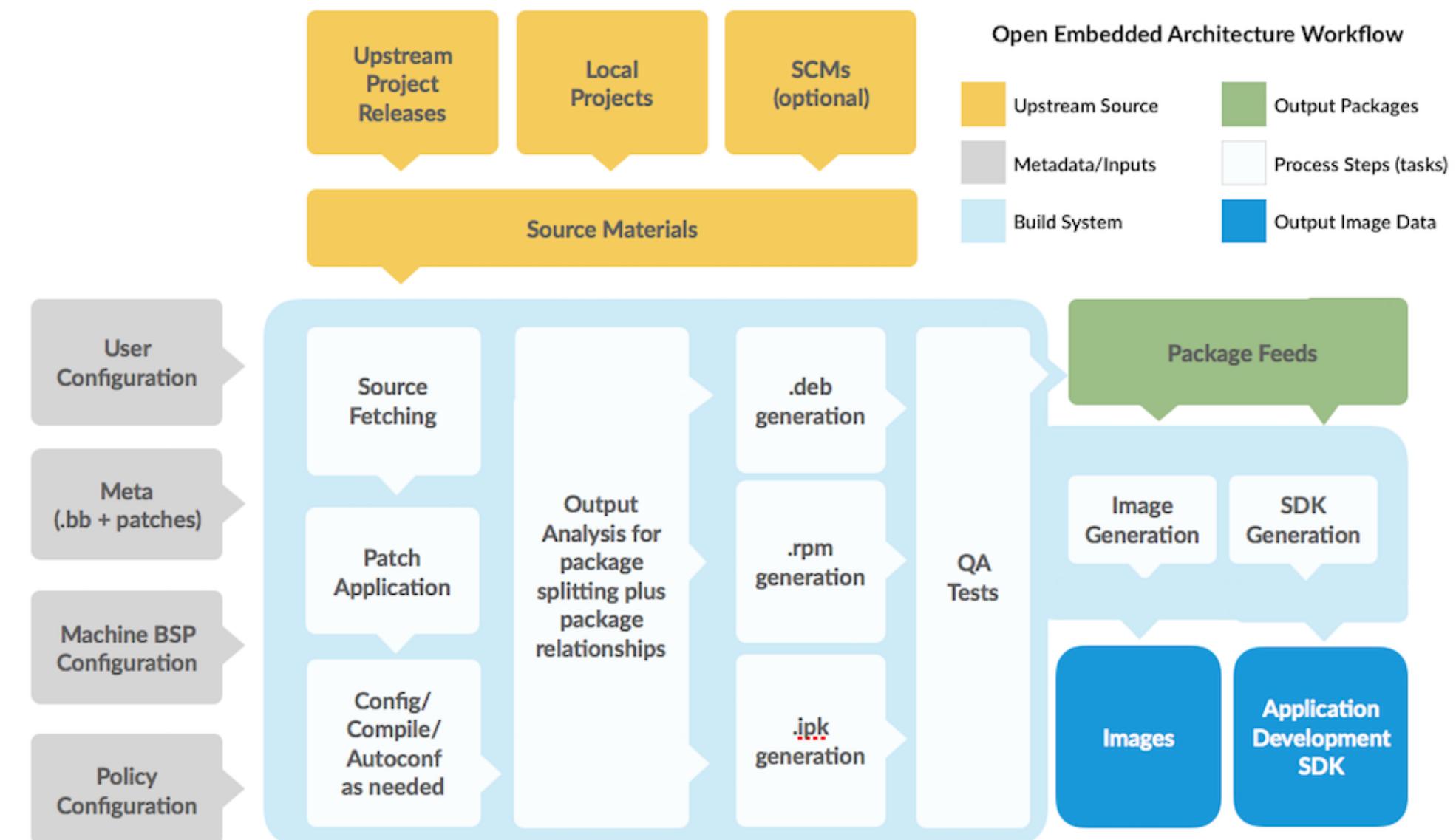
- Extremely configurable and flexible.
- Thousands of pre-configured software packages available for cross-compilation.
- Provides facilities to maintain and extend the system via the layers mechanism.
- Supported by major hardware architectures (ARM, MIPS, x86, PPC) and SoC/SoM manufacturers.
 - It is a *de facto standard* for BSP development.
- Facilitates the management of software licenses (e.g. removing GPLv3).



FEATURES AND ADVANTAGES

- Total support for generating cross-platform Linux systems.
 - Trivial to change the generation of an entire image to a different hardware platform.
- Allows the generation of development tools such as SDKs and emulators.
- Software package management support (*rpm*, *deb*, *ipk*).
 - Enables the development of Linux distributions that requires a package management system.
- Very active community.

BASIC ARCHITECTURE



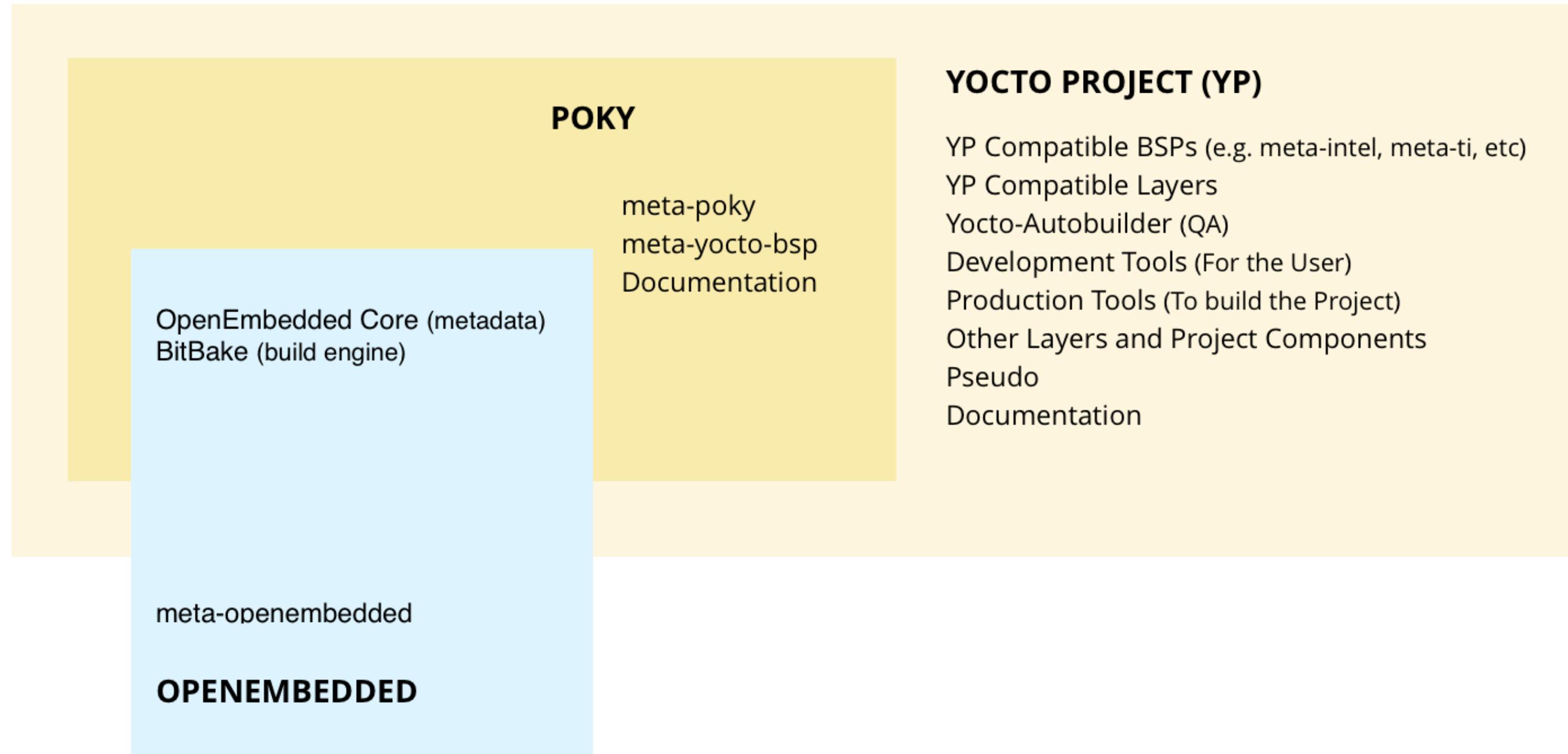
CONCEPTS: POKY

- Poky is the reference distribution of the Yocto Project, available through a Git repository.
<https://git.yoctoproject.org/poky>
- Contains the OpenEmbedded build system (BitBake and OpenEmbedded-Core) and an additional set of metadata.
- Can generate Linux distributions for QEMU and some reference platforms (Beaglebone, EdgeRouter, x86-64, etc).
- It is easily extended through the concept of layers.

CONCEPTS: LAYERS AND METADATA

- Layers contain metadata for compiling the software components that will be part of the Linux distribution.
- There are 3 types of metadata:
 - **Recipes:** set of tasks to compile certain software (*.bb*, *.bbappend*).
 - **Classes:** definition of common tasks that can be reused in recipes (*.bbclass*).
 - **Configuration files:** definition of variables that dictate and influence the generation of the Linux distribution (*.conf*).

YOCTO PROJECT COMPONENTS



DOCUMENTATION

- The Yocto Project is a very well-documented project:
<https://docs.yoctoproject.org/>
- Project introduction and overview:
<https://docs.yoctoproject.org/overview-manual/index.html>
- Quick start guide:
<https://docs.yoctoproject.org/brief-yoctoprojectqs/index.html>
- Reference Manual:
<https://docs.yoctoproject.org/ref-manual/index.html>

DOCUMENTATION

- Yocto Project Development Manual:
<https://docs.yoctoproject.org/dev-manual/index.html>
- BSP (Board Support Package) development guide:
<https://docs.yoctoproject.org/bsp-guide/index.html>
- Linux kernel development manual:
<https://docs.yoctoproject.org/kernel-dev/index.html>

DOCUMENTATION

- Application development manual (SDK):
<https://docs.yoctoproject.org/sdk-manual/index.html>
- BitBake Documentation:
<https://docs.yoctoproject.org/bitbake.html>
- Feel free to refer to this documentation during the training, particularly the Yocto Project's Reference Manual.
<https://docs.yoctoproject.org/ref-manual/index.html>

YOCTO PROJECT

INTRODUCTION TO POKY



POKY

- The Yocto Project community is responsible for maintaining several projects, including Poky.
- Poky is the reference distribution for the Yocto Project.
- With Poky, it's possible to build a custom Linux distribution for hardware platforms officially supported by the project (QEMU, Beaglebone, EdgeRouter, etc).
- In this section, we will have a brief introduction on how to use Poky to build a custom Linux distribution.



THE BUILD MACHINE

- Good processing capacity (e.g. Intel Core i7, 4+ CPUs) is a requirement, with lots of memory (16Gb+) and disk space (100Gb+, SSD).
- It's recommended to use a machine with an officially supported Linux distribution (Fedora, openSUSE, CentOS, Debian or Ubuntu).
- There are some software prerequisites, including *gcc*, *git*, *python*, *tar* and *make*.
- For more detailed information, see the project's website.

<https://docs.yoctoproject.org/dev-manual/start.html#preparing-the-build-host>



DOWNLOADING POKY

- The Poky repository is versioned with Git and can be cloned as below:

```
$ git clone git://git.yoctoproject.org/poky.git
```

- This command will clone the *master* branch of Poky, not recommended as it is a development branch.
- The recommendation is to use the *-b* parameter to clone a stable branch of the project. Example for the Kirkstone release:

```
$ git clone git://git.yoctoproject.org/poky.git -b kirkstone
```



POKY SOURCE CODE

```
$ ls poky/
bitbake                         MAINTAINERS.md    meta-skeleton      README.poky.md
contrib                          Makefile        meta-yocto-bsp    README.qemu.md
documentation                    MEMORIAM       oe-init-build-env scripts
LICENSE                           meta           README.hardware.md
LICENSE.GPL-2.0-only             meta-poky       README.md
LICENSE.MIT                      meta-selftest   README.OE-Core.md
```



POKY DIRECTORIES

Directory	Description
bitbake	BitBake tool
documentation	Project's documentation source code
meta	OpenEmbedded-Core metadata
meta-poky	Poky distribution metadata
meta-selftest	Metadata to test the build system
meta-skeleton	Example metadata
meta-yocto-bsp	Metadata of reference platforms (Beaglebone, etc)
scripts	General purpose scripts (<i>runqemu</i> , <i>devtool</i> , <i>oe-pkgdata-util</i> , etc)

PREPARING THE BUILD DIRECTORY

- After cloning Poky, the next step is to source the build environment configuration script (*oe-init-build-env*), available at the root directory of Poky's source code:

```
$ source poky/oe-init-build-env
```

- This script will configure the build environment in the current terminal, creating by default a directory called *build* with the configuration files needed to build an image.
- Optionally, you can pass the name of the build directory when sourcing the configuration script:

```
$ source poky/oe-init-build-env build-test
```



THE BUILD DIRECTORY

```
$ tree build/
build/
└── conf
    ├── bblayers.conf
    ├── local.conf
    └── templateconf.cfg
```



THE BUILD DIRECTORY

- By default, the entire build process will happen inside the build directory.
- The build directory is created with some configuration files, including *bblayers.conf* and *local.conf*.
- The *bblayers.conf* file allows to configure the layers that will be used to build the image.
- The *local.conf* file allows to define global configuration variables that will be used to customize the build.

BBLAYERS.CONF

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/opt/labs/ex/layers/poky/meta \
/opt/labs/ex/layers/poky/meta-poky \
/opt/labs/ex/layers/poky/meta-yocto-bsp \
"
```



LOCAL.CONF

```
# Machine Selection
MACHINE ??= "qemuarm"

# Where to place downloads
DL_DIR ?= "${TOPDIR}/downloads"

# Where to place shared-state files
SSTATE_DIR ?= "${TOPDIR}/sstate-cache"

# Where to place the build output
TMPDIR = "${TOPDIR}/tmp"

# Default policy config
DISTRO ?= "poky"

# Package Management configuration
PACKAGE_CLASSES ?= "package_rpm"

...
```



VARIABLES IN LOCAL.CONF

Variable	Description
MACHINE	Target platform
DISTRO	Distribution name
DL_DIR	Download directory
SSTATE_DIR	Shared-state caches directory
TMPDIR	Build directory
PACKAGE_CLASSES	Packaging Format (<i>rpm, deb, ipk</i>)
CORE_IMAGE_EXTRA_INSTALL	Additional packages to install
EXTRA_IMAGE_FEATURES	Image features to enable

BUILDING THE IMAGE

- With *bblayers.conf* and *local.conf* configured, just start the build process:

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

- The command above will process the recipe *core-image-minimal.bb*, generating a minimum image to be executed on the target (selected by the *MACHINE* variable).
- Other image recipes are available for processing:

```
$ ls poky/meta/recipes-core/images/
build-appliance-image           core-image-minimal-initramfs.bb
build-appliance-image_15.0.0.bb   core-image-minimal-mtdutils.bb
core-image-base.bb               core-image-ptest-all.bb
core-image-minimal.bb            core-image-ptest-fast.bb
core-image-minimal-dev.bb        core-image-tiny-initramfs.bb
```



OUTPUT DIRECTORY

- The entire build process will take place in the directory pointed to by the *TMPDIR* variable (*tmp/* by default):

```
$ tree -L 1 tmp/
tmp/
├── abi_version
├── buildstats
├── cache
├── deploy
├── hosttools
├── log
├── pkgdata
├── saved_tmpdir
├── sstate-control
├── stamps
├── sysroots-components
├── sysroots-uninative
└── work
    └── work-shared
```

GENERATED IMAGES

- The generated images will be available in *tmp/deploy/images/<machine>/*:

```
$ ls tmp/deploy/images/qemuarm/
core-image-minimal-qemuarm-20220627125439.qemuboot.conf
core-image-minimal-qemuarm-20220627125439.rootfs.ext4
core-image-minimal-qemuarm-20220627125439.rootfs.manifest
core-image-minimal-qemuarm-20220627125439.rootfs.tar.bz2
core-image-minimal-qemuarm-20220627125439testdata.json
core-image-minimal-qemuarm.ext4
core-image-minimal-qemuarm.manifest
core-image-minimal-qemuarm.qemuboot.conf
core-image-minimal-qemuarm.tar.bz2
core-image-minimal-qemuarmtestdata.json
modules--5.15.44+git0+947149960e_1585df4dbb-r0-qemuarm-20220627103615.tgz
modules-qemuarm.tgz
zImage
zImage--5.15.44+git0+947149960e_1585df4dbb-r0-qemuarm-20220627103615.bin
zImage-qemuarm.bin
```

QEMU

- QEMU is an open source hardware emulator that supports multiple architectures, including x86, ARM, MIPS, and PowerPC.
<https://www.qemu.org/>
- In OpenEmbedded-Core, there are several pre-configured machines for QEMU:

```
$ ls poky/meta/conf/machine/
include          qemuarmv5.conf    qemuppc64.conf    qemuriscv64.conf
qemuarm64.conf   qemumips64.conf   qemuppc.conf      qemux86-64.conf
qemuarm.conf     qemumips.conf     qemuriscv32.conf  qemux86.conf
```

QEMU

- If any of these platforms are selected via the *MACHINE* variable:

```
MACHINE ??= "qemuarm"
```

- You can build an image for it:

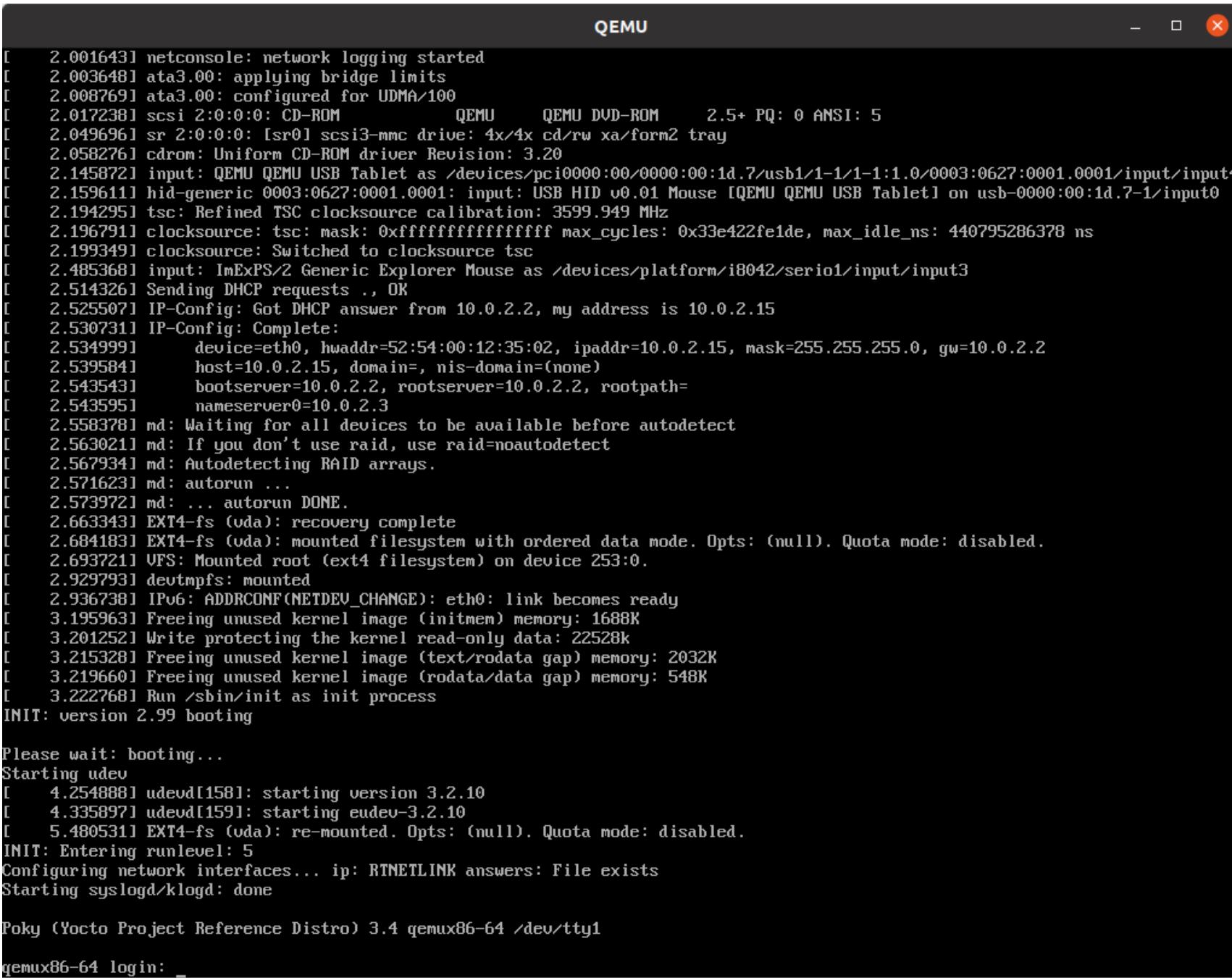
```
$ bitbake core-image-minimal
```

- And then use the *rungemu* script to run the generated image:

```
$ runqemu qemuarm
```



QEMU



The screenshot shows a terminal window titled "QEMU" displaying a Linux kernel boot log. The log includes messages such as network logging starting, device configuration, USB device detection, DHCP requests, IP configuration, and file system mounting. It also shows the transition to runlevel 5 and the start of the Poky (Yocto Project Reference Distro) 3.4 environment.

```
[ 2.001643] netconsole: network logging started
[ 2.003648] ata3.00: applying bridge limits
[ 2.008769] ata3.00: configured for UDMA/100
[ 2.017238] scsi 2:0:0:0: CD-ROM           QEMU   QEMU DVD-ROM    2.5+ PQ: 0 ANSI: 5
[ 2.049696] sr 2:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[ 2.058276] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 2.145872] input: QEMU QEMU USB Tablet as /devices/pci0000:00/0000:00:1d.7/usb1/1-1/1-1:1.0/0003:0627:0001.0001/input/input4
[ 2.159611] hid-generic 0003:0627:0001.0001: input: USB HID v0.01 Mouse [QEMU QEMU USB Tablet] on usb-0000:00:1d.7-1/input0
[ 2.194295] tsc: Refined TSC clocksource calibration: 3599.949 MHz
[ 2.196791] clocksource: tsc: mask: 0xfffffffffffffff max_cycles: 0x33e422fe1de, max_idle_ns: 440795286378 ns
[ 2.199349] clocksource: Switched to clocksource tsc
[ 2.485368] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
[ 2.514326] Sending DHCP requests .., OK
[ 2.525507] IP-Config: Got DHCP answer from 10.0.2.2, my address is 10.0.2.15
[ 2.530731] IP-Config: Complete:
[ 2.534999]   device=eth0, hwaddr=52:54:00:12:35:02, ipaddr=10.0.2.15, mask=255.255.255.0, gw=10.0.2.2
[ 2.539584]   host=10.0.2.15, domain=, nis-domain=(none)
[ 2.543543]   bootserver=10.0.2.2, rootserver=10.0.2.2, rootpath=
[ 2.543595]   nameserver0=10.0.2.3
[ 2.558378] md: Waiting for all devices to be available before autodetect
[ 2.563021] md: If you don't use raid, use raid=noautodetect
[ 2.567934] md: Autodetecting RAID arrays.
[ 2.571623] md: autorun ...
[ 2.573972] md: ... autorun DONE.
[ 2.663343] EXT4-fs (vda): recovery complete
[ 2.684183] EXT4-fs (vda): mounted filesystem with ordered data mode. Opts: (null). Quota mode: disabled.
[ 2.693721] UFS: Mounted root (ext4 filesystem) on device 253:0.
[ 2.929793] devtmpfs: mounted
[ 2.936738] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 3.195963] Freeing unused kernel image (initmem) memory: 1688K
[ 3.201252] Write protecting the kernel read-only data: 22528K
[ 3.215328] Freeing unused kernel image (text/rodata gap) memory: 2032K
[ 3.219660] Freeing unused kernel image (rodata/data gap) memory: 548K
[ 3.222768] Run /sbin/init as init process
INIT: version 2.99 booting

Please wait: booting...
Starting udev
[ 4.254888] udevd[158]: starting version 3.2.10
[ 4.335897] udevd[159]: starting eudev-3.2.10
[ 5.400531] EXT4-fs (vda): re-mounted. Opts: (null). Quota mode: disabled.
INIT: Entering runlevel: 5
Configuring network interfaces... ip: RTNETLINK answers: File exists
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 3.4 qemux86-64 /dev/tty1
qemux86-64 login:
```

LAB 1

GETTING STARTED WITH POKY



YOCTO PROJECT

INTRODUCTION TO THE BUILD SYSTEM

POKY

- Poky is the Yocto Project's reference Linux distribution, and it has basically two major components:
 - **BitBake**: task processor and executor.
 - **Metadata**: definition of tasks and parameters.
- Poky's metadata includes:
 - OpenEmbedded-Core (*meta*): metadata shared with the OpenEmbedded community, which serves as the basis for creating the Linux distribution.
 - Other metadata (*meta-poky*, *meta-yocto-bsp*, etc): metadata maintained by the Yocto Project's community.



METADATA

- Metadata contains the definition of what must be done during the generation of the Linux distribution.
- There are three types of metadata:
 - Recipes (*.bb*, *.bbappend*).
 - Classes (*.bbclass*).
 - Configuration files (*.conf*).

RECIPES

- Recipes are files with a *.bb* or *.bbappend* extension.
- They have this name because they contain the "recipe" to process a certain software component.
- A recipe includes information such as the software component description and version, source code location, dependencies, compilation and installation procedures, etc.
- A list of the main variables that can be used to write a recipe is available in the Yocto Project's reference guide.
<https://docs.yoctoproject.org/ref-manual/varlocality.html#recipes>



APPEND FILES

- Append files have the `.bbappend` extension and make it possible to modify the behavior of a recipe without changing its implementation.
- The contents of an append file are concatenated at the end of the corresponding recipe, making it possible to redefine tasks and variables and change the behavior of the original recipe.
- For each append file, there must be a corresponding recipe.
- Append files encourage reuse and make it easier to maintain metadata in the Yocto Project.

CLASSES

- Classes are files with the *.bbclass* extension, used to abstract common functionalities shared by recipes and other metadata.
- Examples of classes:
 - *autotools.bbclass*: tasks to build software based on Autotools.
 - *cmake.bbclass*: tasks to build software based on CMake.
 - *kernel.bbclass*: tasks to compile the Linux kernel.
- See the Yocto Project reference guide for a list of existing classes.
<https://docs.yoctoproject.org/ref-manual/classes.html>

CONFIGURATION FILES

- Configuration files have the `.conf` extension and contain the definition of variables that will parameterize the build system.
- Among other things, it allows to define information about the hardware architecture and platform, image organization and content, distribution policies, etc.
- Only variable definitions and include directives are allowed in configuration files.
- Common variables that can be used in configuration files are documented in the Yocto Project's reference guide.

<https://docs.yoctoproject.org/ref-manual/varlocality.html#configuration>



LAYERS

- Metadata (recipes, classes and configuration files) are organized into layers in the build system.
- Each layer is composed of a set of metadata grouped in a directory (*meta-*).
- Layers facilitate maintenance and encourage the reuse of metadata.
- In the Yocto Project, a Linux system is generated by combining two or more layers with the required metadata.

POKY LAYERS

```
$ tree -L 1 poky/
poky/
├── bitbake
├── contrib
├── documentation
├── LICENSE
├── LICENSE.GPL-2.0-only
├── LICENSE.MIT
├── MAINTAINERS.md
├── Makefile
├── MEMORIAM
└── meta
    ├── meta-poky
    ├── meta-selftest
    ├── meta-skeleton
    ├── meta-yocto-bsp
    ├── oe-init-build-env
    ├── README.hardware.md -> meta-yocto-bsp/README.hardware.md
    ├── README.md -> README.poky.md
    ├── README.OE-Core.md
    ├── README.poky.md -> meta-poky/README.poky.md
    ├── README.qemu.md
    └── scripts
```



TYPES OF LAYERS

- We can classify layers into three types, which can be combined to generate the image of the operating system:
 - BSP (Board Support Package) layers.
 - Distro (distribution) layers.
 - Software layers (additional software components).

BSP LAYERS

- A BSP layer provides metadata related to the hardware platform:
 - Machine configuration file with the definition of the CPU architecture, configuration and functionality provided by the hardware, etc.
 - Recipes for compiling the bootloader (e.g. U-Boot) and the Linux kernel.
 - Recipes for compiling software for specific hardware features (GPU, security, etc).
- Detailed information on creating and maintaining BSP layers is documented on the project's website.
<https://docs.yoctoproject.org/bsp-guide/bsp.html>

META-YOCTO-BSP

```
$ tree meta-yocto-bsp/
meta-yocto-bsp/
├── conf
│   └── layer.conf
└── machine
    ├── beaglebone-yocto.conf
    ├── edgerouter.conf
    ├── genericx86-64.conf
    ├── genericx86.conf
    └── include
...
├── recipes-bsp
│   └── formfactor
│       └── gma500-gfx-check
└── recipes-graphics
    └── xorg-xserver
├── recipes-kernel
    └── linux
└── wic
    ├── beaglebone-yocto.wks
    ├── edgerouter.wks
    └── genericx86.wks.in
```



META-RASPBERRYPI

```
$ tree meta-raspberrypi/
meta-raspberrypi/
├── classes
└── conf
    └── layer.conf
    └── machine
        ...
        ├── raspberrypi3-64.conf
        ├── raspberrypi3.conf
        ├── raspberrypi4-64.conf
        ├── raspberrypi4.conf
        ├── raspberrypi-cm3.conf
        ├── raspberrypi-cm.conf
        └── raspberrypi.conf
...
    ├── recipes-connectivity
    ├── recipes-core
    ├── recipes-devtools
    ├── recipes-graphics
    ├── recipes-kernel
    ├── recipes-multimedia
    ├── recipes-sato
    └── wic
```



DISTRO LAYERS

- The distribution layer defines some rules and policies for generating the operating system image, including:
 - Toolchain and C library (*glibc*, *musl*, *newlib*, etc).
 - Basic software components (init manager, graphics server, etc).
 - Packaging type (*ipk*, *deb*, *rpm*).
 - Software version and provider.
- The main variables that can be used in the distribution context are documented in the Yocto Project's reference guide.
<https://docs.yoctoproject.org/ref-manual/varlocality.html#distribution-distro>

META-POKY

```
$ tree meta-poky/
meta-poky/
├── classes
│   └── poky-sanity.bbclass
└── conf
    ├── bblayers.conf.sample
    ├── conf-notes.txt
    ├── distro
    │   ├── include
    │   ├── poky-altcfg.conf
    │   ├── poky-bleeding.conf
    │   ├── poky.conf
    │   └── poky-tiny.conf
    ├── layer.conf
    ├── local.conf.sample
    ├── local.conf.sample.extended
    └── site.conf.sample
├── README.poky.md
└── recipes-core
    ├── busybox
    │   └── busybox
    └── busybox_%.bbappend
...
```

META-LUNEOS

```
$ tree meta-luneos/
meta-luneos/
├── classes
└── conf
    ├── distro
    │   └── include
    │       └── luneos.conf
    └── layer.conf
├── COPYING.MIT
├── lib
├── licenses
├── README
├── recipes-android
├── recipes-benchmark
├── recipes-connectivity
├── recipes-containers
├── recipes-core
├── recipes-devtools
├── recipes-extended
├── recipes-graphics
├── recipes-kernel
└── recipes-luneos
...
...
```



SOFTWARE LAYERS

- Software layers provide additional recipes that might compose the final image.
- These layers typically group a set of recipes with similar characteristics:
 - Network applications (e.g. *meta-networking*).
 - Graphics/development framework (e.g. *meta-qt5*).
 - Multimedia applications (e.g. *meta-multimedia*).
 - Python language support (*meta-python*).
 - Product-specific applications.



OPENEMBEDDED-CORE

```
$ tree -L 1 meta
meta
├── classes
├── conf
├── COPYING.MIT
├── files
├── lib
├── recipes-bsp
├── recipes-connectivity
├── recipes-core
├── recipes-devtools
├── recipes-example
├── recipes-extended
├── recipes-gnome
├── recipes-graphics
├── recipes-kernel
├── recipes-multimedia
├── recipes-rt
├── recipes-sato
├── recipes-support
└── recipes.txt
    └── site
```



META-QT5

```
$ tree -L 2 meta-qt5/
meta-qt5/
├── classes
│   ├── cmake_qt5.bbclass
│   ├── populate_sdk_qt5_base.bbclass
│   ├── populate_sdk_qt5.bbclass
│   ├── qmake5_base.bbclass
│   ├── qmake5.bbclass
│   └── qmake5_paths.bbclass
├── conf
└── layer.conf
...
└── recipes-qt
    ├── demo-extrafiles
    ├── examples
    ├── malikit
    ├── meta
    ├── packagegroups
    ├── qmllive
    ├── qsiv
    ├── qt5
    └── qtchooser
...
```



LAYER INDEX

- The Yocto Project maintains a selected list of layers officially supported by the project.

<https://www.yoctoproject.org/software-overview/layers/>

- OpenEmbedded also maintains a more complete (but less validated) layers database.

<https://layers.openembedded.org/layerindex/branch/master/layers/>

- Be aware that hardware manufacturers, vendors and software providers might not list their layers in these databases.

BITBAKE

- The BitBake tool has the following syntax:

```
$ bitbake [options] [recipename/target recipe:do_task ...]
```

- For example, the command below will process the BusyBox recipe:

```
$ bitbake busybox
```

- By default, it will run the *build* task, which depends on all other "normal" tasks needed to process the recipe.

```
$ bitbake -c build busybox
```



RECIPE TASKS

- The `-c` option can be used to execute a specific task:

```
$ bitbake -c fetch busybox
```

- Optionally, the following syntax can also be used:

```
$ bitbake busybox:do_fetch
```

- To list all of the tasks of a recipe, execute the *listtasks* task:

```
$ bitbake -c listtasks busybox
```



RUNNING TASKS

- Use the `-f` option to force a task to run:

```
$ bitbake -f -c compile busybox
```

- The `-k` option causes `bitbake` to continue executing tasks (as far as it can go), even in case of errors:

```
$ bitbake -k busybox
```

- The `--no-setscene` option causes BitBake to ignore build caches:

```
$ bitbake --no-setscene -c compile busybox
```

CLEAN TASKS

- To clean the build output (does not clean the caches), execute the *clean* task:

```
$ bitbake -c clean busybox
```

- To clean the build output and the build caches, execute the *cleansstate* task:

```
$ bitbake -c cleansstate busybox
```

- To clean the build output, the build caches, and the downloaded source code, execute the *cleanall* task:

```
$ bitbake -c cleanall busybox
```

BITBAKE INTERNALS

- Bitbake's objective is to execute tasks described in recipes.
- But how do all metadata are identified and parsed, so tasks are executed until the final operating system image is generated?
- In other words, what happens in the command below?

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



PARSING CONFIGURATION FILES

- The first step performed by BitBake is reading the configuration files, in the following order:
 - *bblayers.conf*, which contains the definition of the layers in the *BBLAYERS* variable.
 - Configuration files (*layer.conf*) for each of the layers included in *BBLAYERS*.
 - *bitbake.conf* file, which contains the basic and global settings that affect all recipes and tasks that will be executed.



BITBAKE.CONF

- The *bitbake.conf* file will include several other configuration files, including:
 - User local files (*site.conf*, *local.conf*, etc).
 - Machine configuration file (*MACHINE*).
 - Distribution configuration file (*DISTRO*).
- After reading the configuration files, BitBake will inherit several classes, including the *base.bbclass*, which defines a set of tasks that will be inherited by all recipes.
- The entire process of reading the metadata can be visualized with the *-e* parameter.



INCLUDE HISTORY

```
$ bitbake -e
#
# INCLUDE HISTORY:
#
# /opt/labs/ex/build-qemu/conf/bblayers.conf
# /opt/labs/ex/layers/poky/meta/conf/layer.conf
# /opt/labs/ex/layers/poky/meta-poky/conf/layer.conf
# /opt/labs/ex/layers/poky/meta-yocto-bsp/conf/layer.conf
# conf/bitbake.conf includes:
#   /opt/labs/ex/layers/poky/meta/conf/abi_version.conf
#   conf/site.conf
#   conf/auto.conf
#   /opt/labs/ex/build-qemu/conf/local.conf
#   /opt/labs/ex/layers/poky/meta/conf/multiconfig/default.conf
#   /opt/labs/ex/layers/poky/meta/conf/machine/qemuarm.conf includes:
#     /opt/labs/ex/layers/poky/meta/conf/machine/include/arm/armv7a/tune-cortexa15.inc includes:
#       /opt/labs/ex/layers/poky/meta/conf/machine/include/arm/arch-armv7ve.inc includes:
#         /opt/labs/ex/layers/poky/meta/conf/machine/include/arm/arch-armv7a.inc includes:
#           /opt/labs/ex/layers/poky/meta/conf/machine/include/arm/arch-armv6.inc includes:
#             /opt/labs/ex/layers/poky/meta/conf/machine/include/arm/arch-armv5-dsp.inc includes:
#               /opt/labs/ex/layers/poky/meta/conf/machine/include/arm/arch-armv5.inc includes:
#                 /opt/labs/ex/layers/poky/meta/conf/machine/include/arm/arch-armv4.inc includes:
...
...
```

TASKS DEPENDENCIES

- When reading the configuration files, the *BBFILES* variable will be populated with a list of available recipes and append files.
- Recipes and append files will be parsed, one by one, and a list of tasks will be defined for each recipe, including dependencies between them.
- Finally, BitBake executes the *build* task of the recipe, which causes all its dependencies to run, until the final operating system image is generated.
- The complete process is described in detail in the BitBake's manual.
<https://docs.yoctoproject.org/bitbake/index.html>

DEPENDENCY CHART

- BitBake can generate a dependency graph in the *dot* format with the *-g* parameter.

```
$ bitbake -g core-image-minimal
$ ls *.dot
task-depends.dot
```

- The generated file can be converted to an image or opened with a *dot* file reader:

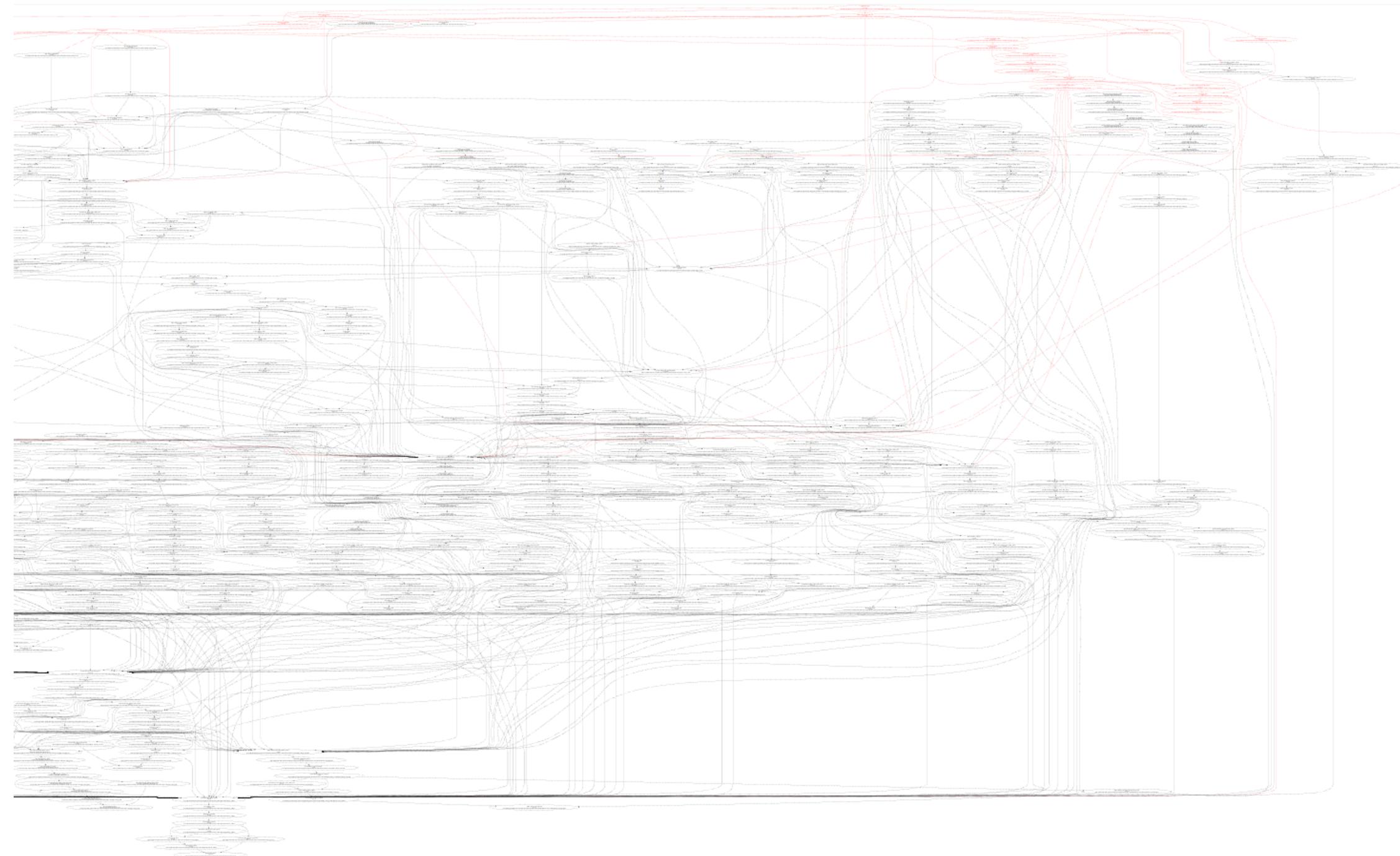
```
$ dot -Tps task-depends.dot -o task-depends.ps
$ xdot task-depends.dot
```

- BitBake is also able to display the dependency graph with a graphical tool (Task Dependency Explorer):

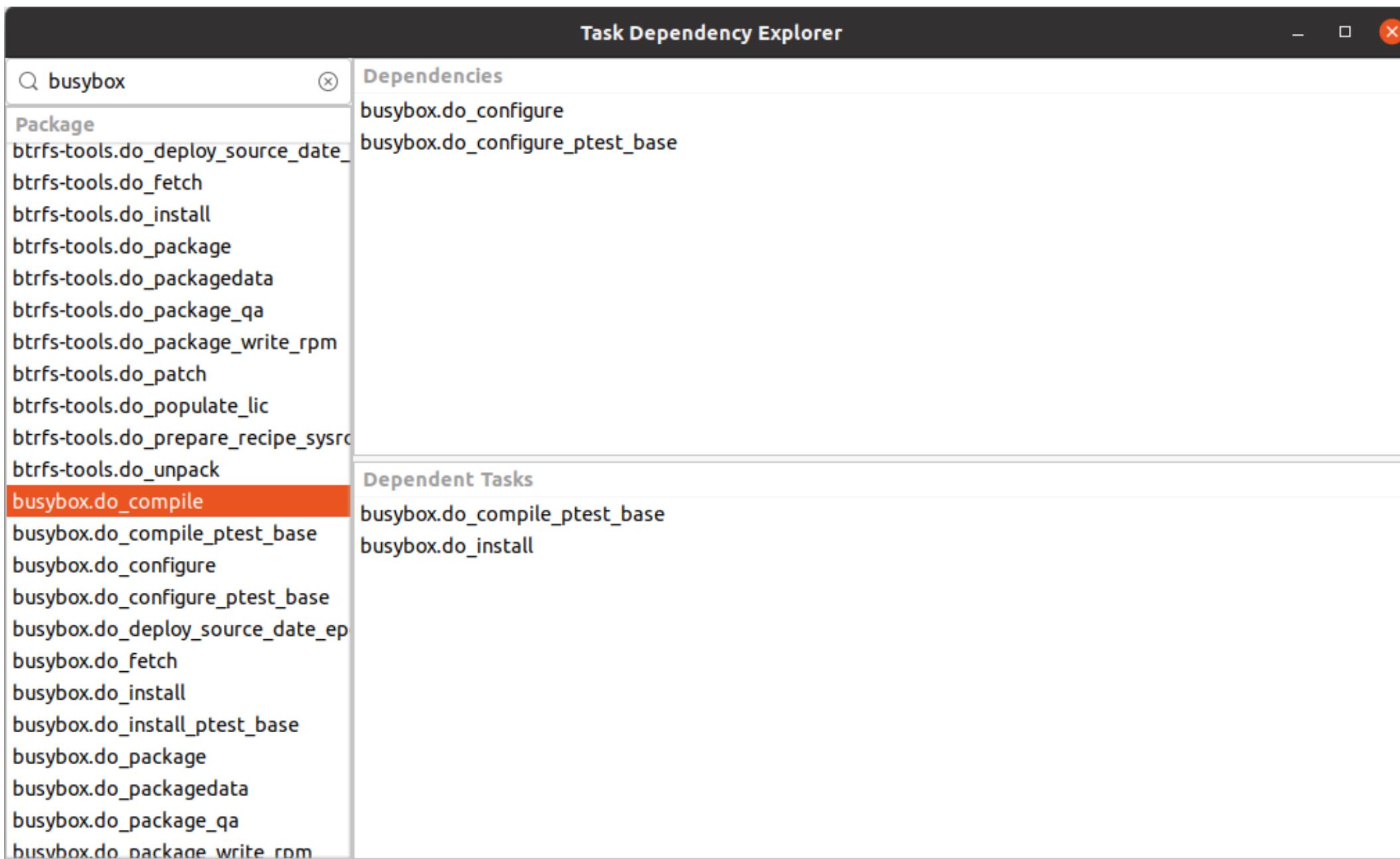
```
$ bitbake -g core-image-minimal -u taskexp
```



DEPENDENCY GRAPH



TASK DEPENDENCY EXPLORER



LAB 2

EXPLORING THE BUILD SYSTEM



YOCTO PROJECT

LAYERS



LAYERS

- OpenEmbedded/Yocto Project supports organizing build system metadata into layers.
- Layers allow isolating operating system customizations, making the development much more modular and easier to maintain.
- The OpenEmbedded community maintains a very complete database of existing layers at the Layer Index website:
<https://layers.openembedded.org/layerindex/branch/master/layers/>

LAYER ORGANIZATION

- Layers are organized into directories in the build system.
- The layer directory can have any name, but the default is to start with *meta-*.
- Poky already comes with some layers:

```
$ ls -d meta*
meta  meta-poky  meta-selftest  meta-skeleton  meta-yocto-bsp
```

- A distribution built with the Yocto Project will use these and other community-provided and project-specific layers.



CREATING A LAYER

- First create a directory for the layer:

```
$ mkdir meta-labworks
```

- Inside the layer directory, create the layer configuration file (*conf/layer.conf*):

```
$ cd meta-labworks
$ mkdir conf
$ vim conf/layer.conf
```

- The content of this file is similar to other layer configuration files, so we can just copy/paste and change it as needed.

LAYER.CONF

```
BBPATH .= ":${LAYERDIR}"

BBFILES += " \
    ${LAYERDIR}/recipes-*//*/*.bb \
    ${LAYERDIR}/recipes-*//*/*.bbappend \
"

BBFILE_COLLECTIONS += "meta-labworks"
BBFILE_PATTERN_meta-labworks = "^${LAYERDIR}/*"
BBFILE_PRIORITY_meta-labworks = "6"

LAYERVERSION_meta-labworks = "1"
LAYERDEPENDS_meta-labworks = "core"

LAYERSERIES_COMPAT_meta-labworks = "kirkstone"
```



LAYER.CONF VARIABLES

- The *BBPATH* variable must be set so that BitBake can find the files provided by the layer (configuration files, classes, etc).

```
BBPATH .= ":${LAYERDIR}"
```

- The *BBFILES* variable must include the paths of recipes provided by the layer:

```
BBFILES += " \
${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend \
"
```

- The layer name must be added to the *BBFILE_COLLECTIONS* variable.

```
BBFILE_COLLECTIONS += "meta-labworks"
```

LAYER.CONF VARIABLES

- The *BBFILE_PATTERN* variable must contain a regular expression, which will be used to match files from BBFILES in the layer.

```
BBFILE_PATTERN_meta-labworks = "^${LAYERDIR}/*"
```

- The *BBFILE_PRIORITY* variable defines the layer priority, useful for example in situations where the same recipe appears in more than one layer.

```
BBFILE_PRIORITY_meta-labworks = "6"
```

LAYER.CONF VARIABLES

- The *LAYERVERSION* variable defines the layer version.

```
LAYERVERSION_meta-labworks = "2"
```

- The *LAYERDEPENDS* variable defines the layer's dependencies on other layers.

```
LAYERDEPENDS_meta-labworks = "core"
```

- The *LAYERSERIES_COMPAT* variable allows to set the layer's compatibility with Yocto Project releases.

```
LAYERSERIES_COMPAT_meta-labworks = "kirkstone"
```

ADDING METADATA TO LAYERS

- After creating the configuration file, the next step is to add metadata to the layer:
 - If the layer has recipes, these are usually added in subdirectories starting with *recipes*-.
 - If it's a distribution layer, then distro configuration files are defined in *conf/distro/*.
 - If it's a BSP layer, then machine configuration files are defined in *conf/machine/*.
- It's common to have a license file (eg *COPYING.MIT*), in addition to a *README* file describing the layer contents, dependencies, maintainer, how to contribute, usage instructions, etc.



META-LABWORKS

```
$ tree meta-labworks/
meta-labworks/
├── conf
│   └── layer.conf
├── COPYING.MIT
└── README
```



ENABLING THE LAYER

- To enable the layer, just add it to the *BBLAYERS* variable in *conf/bblayers.conf*:

```
BBLAYERS ?= " \
/home/sprado/yocto/poky/meta \
/home/sprado/yocto/poky/meta-yocto \
/home/sprado/yocto/poky/meta-yocto-bsp \
/home/sprado/yocto/poky/meta-labworks \
"
```

BITBAKE-LAYERS

```
$ bitbake-layers -h
usage: bitbake-layers [-d] [-q] [-F] [--color COLOR] [-h] <subcommand> ...

...
subcommands:
<subcommand>
    add-layer          Add one or more layers to bblayers.conf.
    remove-layer       Remove one or more layers from bblayers.conf.
    flatten            flatten layer configuration into a separate output directory.
    show-layers         show current configured layers.
    show-overlaid      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.
    layerindex-fetch   Fetches a layer from a layer index along with its dependent layers
    layerindex-show-depends
                        Find layer dependencies from layer index.
    create-layer        Create a basic layer

Use bitbake-layers <subcommand> --help to get help on a specific command
```

BITBAKE-LAYERS: LISTING THE LAYERS

```
$ bitbake-layers show-layers
layer          path                  priority
-----
meta           /opt/labs/ex/layers/poky/meta  5
meta-poky      /opt/labs/ex/layers/poky/meta-poky  5
meta-yocto-bsp /opt/labs/ex/layers/poky/meta-yocto-bsp  5
```

BITBAKE-LAYERS: CREATING LAYERS

```
$ bitbake-layers create-layer --priority 10 ../layers/meta-labworks
Add your new layer with 'bitbake-layers add-layer ../layers/meta-labworks'

$ tree ../layers/meta-labworks/
..../layers/meta-labworks/
    ├── conf
    │   └── layer.conf
    ├── COPYING.MIT
    ├── README
    └── recipes-example
        └── example
            └── example_0.1.bb

$ bitbake-layers add-layer ..../layers/meta-labworks

$ bitbake-layers show-layers
layer                  path                      priority
=====
meta                  /opt/labs/ex/layers/poky/meta  5
meta-poky             /opt/labs/ex/layers/poky/meta-poky  5
meta-yocto-bsp        /opt/labs/ex/layers/poky/meta-yocto-bsp  5
meta-labworks          /opt/labs/ex/layers/meta-labworks  10
```

DYNAMIC LAYERS

- To use a layer with metadata that depends on another layer (e.g. append files), it is required to download all dependencies for the layer to work.
 - For example, a user of a BSP layer that contains append files for graphical applications will need to download the layer that contains the graphical application recipes (even if it's not required), or BitBake will complain.
- In this case, the dynamic layers functionality can be used to enable conditional processing of metadata.
- A dynamic layer can be enabled via the `BBFILES_DYNAMIC` variable in `layer.conf`.

EXAMPLE: DYNAMIC LAYERS

```
$ tree -L 2 meta-oe/dynamic-layers/meta-python/recipes-core
meta-oe/dynamic-layers/meta-python/recipes-core
└── packagegroups
    └── packagegroup-meta-oe.bbappend

$ tree -L 2 meta-oe/dynamic-layers/perl-layer/recipes-core/
meta-oe/dynamic-layers/perl-layer/recipes-core/
└── packagegroups
    └── packagegroup-meta-oe.bbappend
```

```
# meta-oe/conf/layer.conf

...
BBFILES_DYNAMIC += " \
    meta-python:${LAYERDIR}/dynamic-layers/meta-python/recipes-*/*/*.bb \
    perl-layer:${LAYERDIR}/dynamic-layers/perl-layer/recipes-*/*/*.bb \
"
...
"
```

COMPATIBILITY PROGRAM

- When creating a layer to use with the Yocto Project, it's beneficial to ensure that the layer is compliant with the project's standards:
 - Ensures a minimum quality standard defined by the community, including interoperability with other layers.
 - Allows the usage of the *Yocto Project Compatible Logo*.
- The layer certification process involves running a script ([yocto-check-layer](#)) and registering the layer on the Yocto Project's website.
- More information about this process is available at [Making Sure Your Layer is Compatible With Yocto Project](#).



YOCTO-CHECK-LAYER

```
$ yocto-check-layer --without-software-layer-signature-check meta-labworks/
INFO: Detected layers:
INFO: meta-training: LayerType.SOFTWARE, /opt/labs/ex/layers/meta-training
INFO:
INFO: Setting up for meta-training(LayerType.SOFTWARE), /opt/labs/ex/layers/meta-training
INFO: Getting initial bitbake variables ...
INFO: Getting initial signatures ...
INFO: Adding layer meta-training
INFO: Starting to analyze: meta-training
INFO: -----
INFO: skipped "BSPCheckLayer: Layer meta-training isn't BSP one."
INFO: test_layerseries_compat (common.CommonCheckLayer)
INFO: ... ok
INFO: test_parse (common.CommonCheckLayer)
INFO: ... ok
...
INFO: Ran 7 tests in 435.104s
INFO: OK
INFO: (skipped=3)
INFO:
INFO: Summary of results:
INFO:
INFO: meta-training ... PASS
```

LAB 3

CREATING LAYERS

YOCTO PROJECT

INTRODUCTION TO RECIPES

RECIPES

- Recipes are files with the *.bb* extension, processed by the BitBake tool to generate the various software components of the operating system.
- The name of a recipe file is normally in the form *<name>_<version>.bb*. Examples:
 - libdrm_2.4.110.bb
 - libpng_1.6.37.bb
 - mmc-utils_git.bb
- To process a recipe, simply use the first part of its name:

```
$ bitbake libdrm
```



PACKAGES

- The end result of processing a recipe is packages.
 - Currently, the Yocto Project supports three types of packaging: *ipk*, *deb* and *rpm*.
- Packages contain files generated during recipe processing, aggregated by type (-*dbg* for debug files, -*doc* for documentation, etc).
- A single recipe will generate multiple packages!

```
$ bitbake -e unzip | grep ^PACKAGES=
PACKAGES="unzip-src unzip-dbg unzip-staticdev unzip-dev unzip-doc unzip-locale unzip"
```



CREATING RECIPES

- Before creating a new recipe, check if there isn't already something available in the OpenEmbedded Layer Index database.
<https://layers.openembedded.org/layerindex/branch/kirkstone/recipes/>
- The *bitbake-layers* command can help search for existing recipes in the layers enabled in *bblayers.conf*:

```
$ bitbake-layers show-recipes | grep "busybox:" -A 1
busybox:
meta          1.35.0
```

- If you need to create a new recipe, other recipes can be used as a starting point.
- Also, some tools can help creating recipes, like *recipetool* and *devtool*. We will study these tools later in the training.

STRUCTURE OF A RECIPE

- A recipe basically contains the definition of **tasks** and **variables**.
- Tasks contain instructions to process a recipe (download the source code, unpack, apply patches, configure, compile, install, package, etc).
- Variables allow to parameterize and configure the behavior of tasks (source code location, patches to be applied, dependencies, compilation flags, etc).

SETSERIAL_2.17.BB

```
SUMMARY = "Controls the configuration of serial ports"
DESCRIPTION = "setserial is a program designed to set and/or report the configuration information as
HOMEPAGE = "http://setserial.sourceforge.net"
AUTHOR = "Theodore Ts'o <tytso@mit.edu>"
SECTION = "console/utils"

LICENSE = "GPLv2.0"
LIC_FILES_CHKSUM = "file://version.h;beginline=1;endline=6;md5=2e7c59cb9e57e356ae81f50f4e4dfd99"
PR = "r3"

DEPENDS += "groff-native"

inherit autotools-brokensep

SRC_URI = "${SOURCEFORGE_MIRROR}/setserial/${BPN}-${PV}.tar.gz \
          file://add_stdlib.patch \
          file://ldflags.patch \
          "
SRC_URI[md5sum] = "c4867d72c41564318e0107745eb7a0f2"
SRC_URI[sha256sum] = "7e4487d320ac31558563424189435d396ddf77953bb23111a17a3d1487b5794a"

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

TASKS

- Task is a step in processing a recipe (downloading source code, applying patches, configuring, compiling, etc).
- Tasks can be defined within the recipe file or inheriting a class.
- By default, all recipes automatically inherit some classes when BitBake is parsing the metadata, including the *base.bbclass* class.
- These classes define some common tasks in all recipes (*do_fetch*, *do_unpack*, *do_patch*, *do_configure*, *do_compile*, etc).

TASKS IN AN EMPTY RECIPE

```
$ cat ./layers/meta-labworks/recipes-test/empty-recipe/empty-recipe_1.0.bb
LICENSE = "CLOSED"

$ bitbake empty-recipe -c listtasks
...
do_build                         Default task for a recipe - depends on all other normal tasks
do_checkuri                       Validates the SRC_URI value
do_clean                           Removes all output files for a target
do_cleanall                        Removes all output files, shared state cache, and downloaded s
do_cleansstate                     Removes all output files and shared state cache for a target
do_compile                          Compiles the source in the compilation directory
do_configure                        Configures the source by enabling and disabling any build-time
do_deploy_source_date_epoch        (setscene version)
do_deploy_source_date_epoch_setscene   Starts a shell with the environment set up for development/deb
do_devshell                         Fetches the source code
do_fetch                            Copies files from the compilation directory to a holding area
do_install                           Lists all defined tasks for a target
do_listtasks                        Analyzes the content of the holding area and splits it into su
do_package                           Runs QA checks on packaged files
do_package_qa                        Runs QA checks on packaged files (setscene version)
do_package_qa_setscene              Analyzes the content of the holding area and splits it into su
do_package_setscene                 Creates the actual RPM packages and places them in the Package
do_package_write_rpm                Creates the actual RPM packages and places them in the Package
```



INHERITING CLASSES

- Tasks in a recipe can be changed, replaced or extended through classes (files with a `.bbclass` extension).
- Classes define common tasks that can be reused by recipes.
- A class can be used in a recipe via the `inherit` keyword.

```
inherit autotools
```

- The most commonly used classes are documented in the Reference Manual:
<https://docs.yoctoproject.org/ref-manual/classes.html>

IMPLEMENTING TASKS

- A recipe can change existing tasks or even implement custom ones.
- Tasks can be implemented in Shell Script or Python.
- Before implementing a task, make sure that a class with the necessary logic for processing the recipe doesn't already exist.

SHELL SCRIPT VS PYTHON

- Tasks are typically implemented in Shell Script:

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

- Tasks can also be implemented in Python:

```
python do_showdate() {  
    import time  
    print time.strftime('%d/%m/%Y', time.gmtime())  
}
```

CHANGING TASKS

- It's possible to reimplement an existing task:

```
do_compile() {  
    # instructions to compile here  
}
```

- Or add instructions to an existing task with *append* or *prepend*:

```
do_configure:prepend() {  
    # extra configure commands  
}  
do_install:append() {  
    # extra install commands  
}
```

CREATING A NEW TASK

- Tasks can be added with the keyword *addtask*:

```
do_mkimage() {  
    # create image here  
}  
addtask mkimage
```

- New tasks don't run by default, but can be executed manually with the *-c* parameter:

```
$ bitbake myrecipe -c mkimage
```

- For a new task to run automatically while processing a recipe, it must be added to the task dependency chain:

```
addtask mkimage after do_compile before do_install
```



VARIABLES

- Variables are used to parameterize and define the behavior of tasks during recipe processing. Examples:
 - The *do_fetch* task uses the *SRC_URI* variable to identify the location of the source code.
 - The *do_patch* task uses the *SRC_URI* variable to identify the patches that must be applied to the source code.
 - The *do_populate_lic* task uses the *LIC_FILES_CHKSUM* variable to validate the checksum of license files.
- Some variables are automatically defined by BitBake when a recipe is processed.

AUTO-GENERATED VARIABLES

Variable	Description
PN	Package name
PV	Package version
PR	Package revision
WORKDIR	Working directory (where the recipe is processed)
S	Source directory (where the source code is unpacked)
B	Build directory (where the software is compiled)
D	Destination directory (where the software is installed)

AUTO-GENERATED VARIABLES

```
$ cat ./layers/meta-labworks/recipes-test/empty-recipe/empty-recipe_1.0.bb
LICENSE = "CLOSED"

$ bitbake empty-recipe -e | grep ^PN=
PN="empty-recipe"

$ bitbake empty-recipe -e | grep ^PV=
PV="1.0"

$ bitbake empty-recipe -e | grep ^PR=
PR="r0"

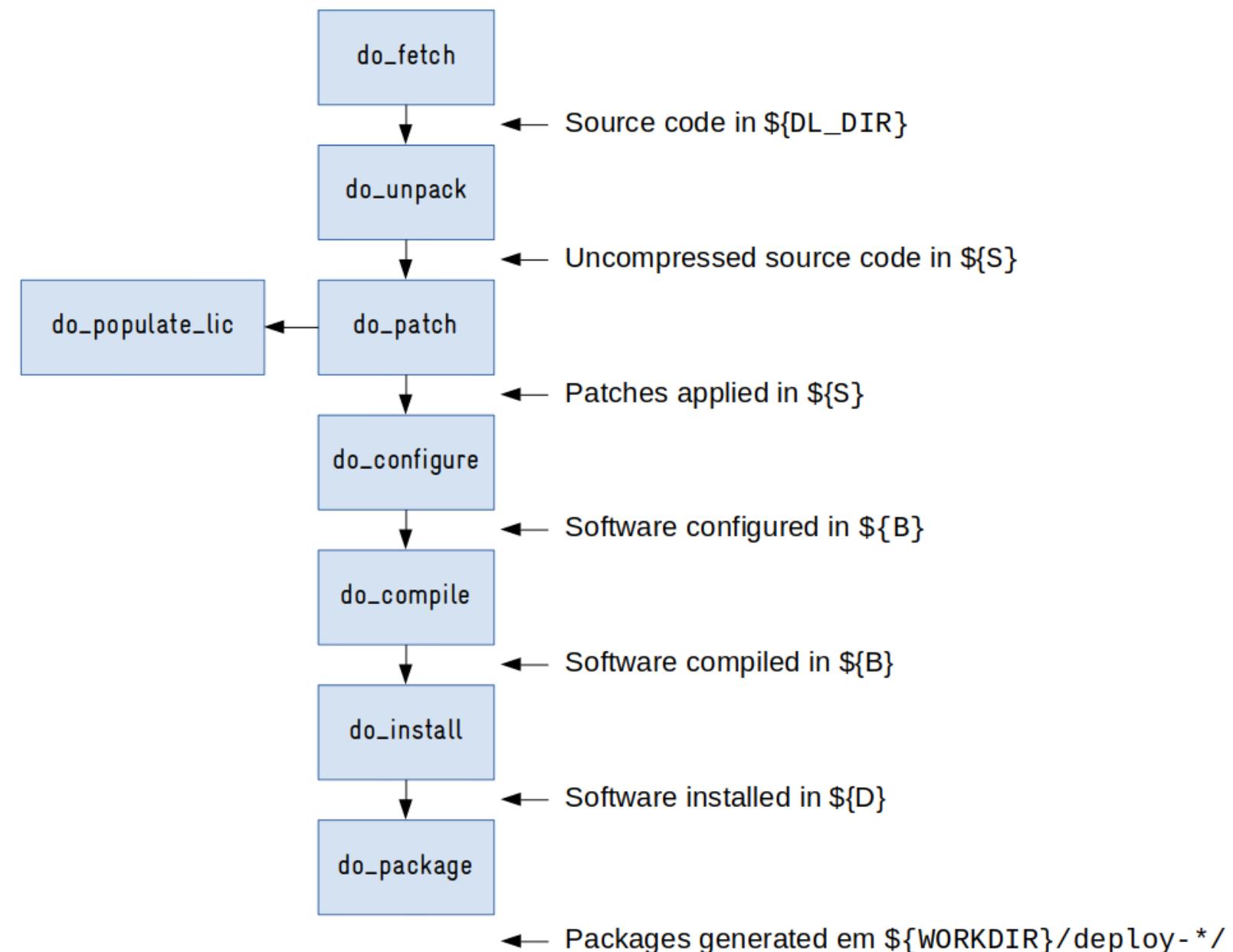
$ bitbake empty-recipe -e | grep ^WORKDIR=
WORKDIR="/opt/labs/ex/build/tmp/work/core2-64-poky-linux/empty-recipe/1.0-r0"

$ bitbake empty-recipe -e | grep ^S=
S="/opt/labs/ex/build/tmp/work/core2-64-poky-linux/empty-recipe/1.0-r0/empty-recipe-1.0"

$ bitbake empty-recipe -e | grep ^B=
B="/opt/labs/ex/build/tmp/work/core2-64-poky-linux/empty-recipe/1.0-r0/empty-recipe-1.0"

$ bitbake empty-recipe -e | grep ^D=
D="/opt/labs/ex/build/tmp/work/core2-64-poky-linux/empty-recipe/1.0-r0/image"
```

PROCESSING A RECIPE



DO_FETCH

- The *SRC_URI* variable is used to define the source code location.

```
SRC_URI = "https://www.mpfr.org/mpfr-$PV/mpfr-$PV.tar.xz"
```

- The source code can be obtained from a variety of sources, including HTTP, FTP, GIT, SVN, etc.
 - See the [BitBake documentation](#) for a complete list of supported protocols.
- The source code is downloaded to the directory pointed to by the *DL_DIR* variable.

DO_FETCH

- If the source code is downloaded from a remote server without using a version control system (VCS), it's mandatory to provide the hash of the file being downloaded.

```
SRC_URI = "http://www.libsdl.org/release/SDL2-${PV}.tar.gz"  
SRC_URI[sha256sum] = "65be9ff6004034b5b2ce9927b5a4db1814930f169c4b2dae0a1e4697075f287b"
```

- If the source code is available in a VCS repository, it's mandatory to define the variable *SRCREV* and optionally *PV*.

```
SRC_URI = "git://git.yoctoproject.org/${BPN};branch=master"  
SRCREV = "7ad885912efb2131e80914e964d5e635b0d07b40"  
PV = "0.3+git${SRCPV}"
```

DO_UNPACK

- The source code will be unpacked into the directory defined by the *S* variable.
- If the source code is downloaded from a VCS repository, it's necessary to set the *S* variable.

```
SRC_URI = "git://anongit.freedesktop.org/git/xorg/lib/${XORG_PN};protocol=https"
S = "${WORKDIR}/git"
```

- A recipe can have local files (e.g. patches), provided in the *SCR_URI* variable through the *file* protocol.

```
SRC_URI = "http://www.mirrorservice.org/sites/lsof.itap.purdue.edu/pub/tools/unix/lsof/lsof_${PV} .
file://lsof-remove-host-information.patch"
```

DO_UNPACK

- These files are stored along with the metadata in the recipe directory, and BitBake uses the *FILESPATH* variable to search for them.

```
$ tree poky/meta/recipes-extended/lsof/
poky/meta/recipes-extended/lsof/
├── files
│   └── lsof-remove-host-information.patch
└── lsof_4.91.bb
$ bitbake lsof -e | grep ^FILESPATH=
```

- During the execution of the *do_unpack* task, these files are copied to the recipe's working directory (*WORKDIR*).

DO_PATCH

- All files defined in the *SRC_URI* variable with *.patch* or *.diff* extensions are treated as patches.
- By default, the *-p1* option will be used to apply patches (this behavior can be changed with the *striplevel* option).

```
SRC_URI += "file://0002-llvm-allow-env-override-of-exe-path.patch;striplevel=2"
```

- The *patchdir* option allows applying the patch from a specific directory.

```
SRC_URI += "file://0001-Workaround-for-GCC-11-uninit-variable-warnings-946.patch;patchdir=src"
```

DO_POPULATE_LIC

- The *do_populate_lic* task is responsible for managing the license of the software.
- The LICENSE variable must be used to specify the license:

```
LICENSE = "GPL-2.0-only"
```

- A list of common license names is available in OpenEmbedded-Core at *files/common-licenses/*.
- The recipe must also contain the *LIC_FILES_CHKSUM* variable to ensure that the license file has not been changed.

```
LIC_FILES_CHKSUM = "file://LICENSE;md5=44bc22578be94b6536c8bdc3a01e5db9"
```

DO_CONFIGURE

- It's common for applications to have a software configuration mechanism before starting the build (e.g. Autotools and CMake).
- Usually, the implementation of this task is inherited from a class. For example, to compile software based on Autotools:

```
inherit autotools
```

- In the case of Autotools, the *EXTRA_OECONF* variable can be used to pass extra options to the configuration script.

```
EXTRA_OECONF = "--enable-debug"
```

- Check the documentation of the class being used for more information.



DO_COMPILE

- Similar to the configure task, the implementation of the compile task is usually inherited from a class.
- For example, to compile a CMake based application:

```
inherit cmake
```

- If necessary, a custom build task can be defined:

```
do_compile() {
    ${CC} test.c -o test
}
```

- The *EXTRA_OEMAKE* and *EXTRA_OECMAKE* variables can be used to pass extra commands to the *make* and *cmake* tools, respectively.



DO_INSTALL

- Similar to the configure and compile tasks, the implementation of the *do_install* task is usually inherited from a class.
- If necessary, a custom installation task can be defined.
- It's also possible to customize the install task by appending extra commands to it:

```
do_install:append() {  
    install -d ${D}/${libdir}/pkgconfig  
    install -m 0644 ${B}/src/libgcrypt.pc ${D}/${libdir}/pkgconfig/  
}
```

- The installation is executed in the directory pointed to by the *D* variable, which by default contains *\${WORKDIR}/image*.

DO_PACKAGE

- The purpose of this task is to package the various software artifacts (executables, libraries, documentation, etc.) generated when building the recipe.

```
$ bitbake libpcap -e | grep ^PACKAGES=
PACKAGES="libpcap-src libpcap-dbg libpcap-staticdev libpcap-dev libpcap-doc libpcap-locale libpcap
```

- For this, the software artifacts generated during the build are divided into directories in `${WORKDIR}/packages-split`.

```
$ ls -1 tmp/work/cortexa9t2hf-neon-poky-linux-gnueabi/libpcap/1.10.1-r0/packages-split/
libpcap
libpcap-dbg
libpcap-dev
libpcap-doc
libpcap-locale
libpcap-src
libpcap-staticdev
```

DO_PACKAGE

- The *FILES* variable is used to define where each software artifact should go:

```
$ bitbake libpng -e | grep ^FILES\:  
FILES:libpng="/usr/bin/* /usr/sbin/* /usr/libexec/* /usr/lib/lib*.so.* /etc /com /var /bin/* /sbin/* /  
FILES:libpng-dev="/usr/include /lib/lib*.so /usr/lib/lib*.so /usr/lib/*.la /usr/lib/*.o"  
FILES:libpng-doc="/usr/share/doc /usr/share/man /usr/share/info /usr/share/gtk-doc /usr/share/gnome/he  
FILES:libpng-locale="/usr/share/locale"  
FILES:libpng-src=""  
FILES:libpng-staticdev="/usr/lib/*.a /lib/*.a /usr/lib/libpng/*.a"  
FILES:libpng-tools="/usr/bin/png-fix-itxt /usr/bin/pngfix /usr/bin/pngcp"
```

- After splitting software artifacts into directories, final installation packages are generated in \${WORKDIR}/deploy-<package-format>.

```
$ ls tmp/work/cortexa9t2hf-neon-poky-linux-gnueabi/libpcap/1.10.1-r0/deploy-rpms/cortexa9t2hf_neon/  
libpcap1-1.10.1-r0.cortexa9t2hf_neon.rpm libpcap-dev-1.10.1-r0.cortexa9t2hf_neon.rpm  
libpcap-src-1.10.1-r0.cortexa9t2hf_neon.rpm libpcap-dbg-1.10.1-r0.cortexa9t2hf_neon.rpm  
libpcap-doc-1.10.1-r0.cortexa9t2hf_neon.rpm libpcap-staticdev-1.10.1-r0.cortexa9t2hf_neon.rpm
```

WORKING DIRECTORY

```
$ ls -1 tmp/work/core2-64-poky-linux/libpcap/1.10.1-r0/
build
configure.sstate
debugsources.list
deploy-rpms
deploy-source-date-epoch
image
libpcap-1.10.1
libpcap.spec
license-destdir
package
packages-split
pkgdata
pkgdata-pdata-input
pkgdata-sysroot
pseudo
recipe-sysroot
recipe-sysroot-native
source-date-epoch
sysroot-destdir
temp
```

LOGS

```
$ ls tmp/work/core2-64-poky-linux/libpcap/1.10.1-r0/temp/
depsig.do_deploy_source_date_epoch
depsig.do_deploy_source_date_epoch.2949190
depsig.do_package
depsig.do_package.3006016
depsig.do_packagedata
depsig.do_packagedata.3006753
depsig.do_package_qa
depsig.do_package_qa.3006796
depsig.do_package_write_rpm
depsig.do_package_write_rpm.3006795
depsig.do_populate_lic
depsig.do_populate_lic.2949191
depsig.do_populate_sysroot
depsig.do_populate_sysroot.3006017
log.do_compile
log.do_compile.3005561
log.do_configure
log.do_configure.3001312
log.do_deploy_source_date_epoch
log.do_deploy_source_date_epoch.2949190
log.do_fetch
log.do_fetch.2947221
log.do_install
run.do_unpack
run.do_unpack.2949177
run.emit_pkgdata.3006016
run.extend_recipe_sysroot.2947221
run.extend_recipe_sysroot.2949182
run.extend_recipe_sysroot.3001129
run.extend_recipe_sysroot.3001312
run.extend_recipe_sysroot.3005808
run.extend_recipe_sysroot.3006016
run.extend_recipe_sysroot.3006017
run.extend_recipe_sysroot.3006795
run.extend_recipe_sysroot.3006796
run.fixup_perms.3006016
run.package_convert_pr_autoinc.3006016
run.packagedata_translate_pr_autoinc.3006753
run.package_depchains.3006016
run.package_do_filedeps.3006016
run.package_do_pkgconfig.3006016
run.package_do_shlibs.3006016
run.package_do_split_locales.3006016
run.package_fixsymlinks.3006016
run.package_get_auto_pr.3006753
run.package_name_hook.3006016
```



RECIPE TEMPLATE

```
DESCRIPTION = ""
HOMEPAGE = ""
SECTION = ""

LICENSE = ""
LIC_FILES_CHKSUM = ""

SRC_URI = ""
SRC_URI[sha256sum] = ""

DEPENDS = ""

inherit <some_class>
```



EXAMPLE 1: HELLO.C

```
DESCRIPTION = "Hello World application"
HOMEPAGE = "git://mygitserver.com/hello.git"
SECTION = "tests"

LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=7363621101019373746565758497289305"

SRC_URI = "git://mygitserver.com/hello.git;protocol=https;branch=main"
SRCREV = "6a4be9e9946df310d9402f995f371c7deb8c27ba"

S = "${WORKDIR}/git"

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

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

EXAMPLE 2: AUTOTOOLS

```
SUMMARY = "GNU Hello World application"
HOMEPAGE = "https://hello.gnu.org"
SECTION = "tests"

LICENSE = "GPL-2.0-or-later"
LIC_FILES_CHKSUM = "file://COPYING;md5=751419260aa954499f7abaabaa882bbe"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"
SRC_URI[sha256sum] = "d21b2d1fd78c1efbe1f2c16dae1cb23f8fd231dcf891465b8debe636a9054b0c"

DEPENDS = "zlib"

inherit autotools

EXTRA_OECONF = "--enable-debug"
```



EXAMPLE 3: AUTOTOOLS

```
SUMMARY = "X11 Code Viewer"
DESCRIPTION = "Allow viewing of X11 code in a fancy way which allows easier \
              and more productive X11 programming"
AUTHOR = "John Bazz <john.bazz@example.org>"
HOMEPAGE = "http://www.example.org/xcv/"
SECTION = "x11/applications"
LICENSE = "GPL-2.0"
DEPENDS = "libsm libx11 libxext libxaw"
PV = "0.9+git${SRCPV}"

# upstream does not yet publish any release so we have to fetch last working version from GIT
SRCREV = "6a5e79ae8a0f4a273a603b2df1742972510d3d8f"
SRC_URI = "git://xcv.example.org/xcv;protocol=http \
           file://toolbar-resize-fix.patch"

S = "${WORKDIR}/xcv/"

inherit autotools

do_configure:prepend() {
    rm ${S}/aclocal.m4
}

do_install() {
```



OPENEMBEDDED STYLE GUIDE

- There is an official OpenEmbedded style guide for metadata development.
<https://www.openembedded.org/wiki/Styleguide>
- The style guide standardizes code formatting, the organization of variables, task declaration order, among other things.
- The idea is to follow the same coding standard, making it easier to read and maintain BitBake metadata.

LAB 4

DEVELOPING RECIPES



YOCTO PROJECT

ADVANCED RECIPE CONCEPTS



BITBAKE AND VARIABLES

- When processing a recipe, BitBake will parse all variables defined in configuration files, class files, recipes and append files.
- During parsing, assignments to the same variable might happen in different places in the parsed metadata.
- For this reason, it is important to know the BitBake syntax when assigning a value to a variable.

VARIABLE ASSIGNMENT

- The = operator performs the assignment at the moment it finds the assignment (hard assignment).

```
VAR = "value"
```

- The ?= operator performs the assignment at the moment it finds the assignment, in case the variable is not yet defined (soft assignment).

```
VAR ?= "value"
```

- The ??= operator performs the assignment at the end of the parsing, in case the variable is not yet defined (weak assignment).

```
VAR ??= "value"
```

VARIABLE EXPANSION

- The \${} operator makes it possible to expand a variable inside another variable.
- Using the = operator, the expansion only happens when the variable is used:

```
VAR1 = "A"  
VAR2 = "${VAR1} B"  
VAR1 = "C"
```

- Using the := operator the expansion happens at assignment time:

```
VAR1 = "A"  
VAR2 := "${VAR1} B"  
VAR1 = "C"
```

CONCATENATION

- The `+=` operator can be used to concatenate to the end of a variable (append), and the `=+` operator can be used to concatenate to the beginning of a variable (prepend).

```
VAR = "1"
VAR =+ "0"
VAR += "2"
```

- The `"+="` and `"=+"` operators add a space when concatenating.
- Use the operators `".=`" and `"=.`" to concatenate without adding spaces:

```
VAR = "1"
VAR =. "0 "
VAR .= " 2"
```

APPEND AND PREPEND

- Concatenation can also be done by adding *:append* and *:prepend* to variables.

```
VAR      = "1"  
VAR:prepend = "0 "  
VAR:append  = " 2"
```

- Like the `.=` and `=.` operators, spaces are not automatically added.
- But unlike the `.=` and `=.` operators, the assignment happens only at the end of the parsing process.

REMOVING

- To remove a value from a variable, simply add `:remove` to the variable name:

```
VAR      = "1 2 1 3"  
VAR:remove = "1"
```

- The final result will be:

```
VAR = "2 3"
```

- All occurrences of the string are removed.
- Extra spaces around the string are also removed.

OVERRIDES

- BitBake provides a mechanism called overrides to conditionally set the value of variables.
- A variable called *OVERRIDES* contains values separated by colons:

```
OVERRIDES = "linux:arm:colibri-imx6"
```

- These values can be used to conditionally set values to variables:

```
VAR:arm = "value"
```

OVERRIDES

- In the example below, what will be the value of the variable *VAR* after the parsing?

```
OVERIDES = "linux-gnueabi:arm:armv7a:mx6:mx6dl:colibri-imx6:poky"
VAR      = "1"
VAR:mx6 = "2"
VAR:ppc = "3"
```

- You can also conditionally concatenate using *append* or *prepend*.

```
VAR:append:mx6 = " 4"
```

OLD OVERRIDES SYNTAX

- The usage of the : character to do conditional assignment via *OVERRIDES* is recent, adopted by default as of release 3.4 (honister).
- Before this release, the _ character was used for conditional assignments.

```
VAR_mx6 = "1"  
VAR_append_mx6 = " 2"
```

- More information about this change is available in the Yocto Project's [Release Migration Guide](#).

APPEND FILES

- Append files have the `.bbappend` extension and make it possible to modify the behavior of a recipe without changing its original implementation.
- It can be used to apply patches, customize build flags, install additional files, and so on, making the maintenance of the metadata quite flexible.
- The way it works is very simple: the content of an append file is concatenated at the end of the recipe, making it possible to redefine tasks and variables, and consequently change the behavior of the original recipe.

CREATING APPEND FILES

- Suppose you want to change the behavior of the *json-c* recipe available in the OpenEmbedded-Core layer:

```
$ ls poky/meta/recipes-devtools/json-c/json-c_0.15.bb
```

- The first step is to create the same directory structure in your layer:

```
$ mkdir -p meta-labworks/recipes-devtools/json-c/
```

- And implement the append file inside of it (the % character indicates that the append file is valid for any version of the recipe):

```
$ vim meta-labworks/recipes-devtools/json-c/json-c_%.bbappend
```



CONTENT OF AN APPEND FILE

- In an append file it's possible to:
 - Define new tasks, redefine existing tasks, concatenate commands to tasks.
 - Assign values to variables, define new variables, redefine existing variables.
- It's common to use the *FILESEXTRAPATHS* variable in an append file.
 - This variable allows adding the append recipe directory to BitBake's search paths (*FILESPATH*).

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

EXAMPLE 1: ALSA-LIB_%._BBAPPEND

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

IMX_PATCH = " \
    file://0001-add-conf-for-multichannel-support-in-imx.patch \
    file://0005-add-ak4458-conf-for-multichannel-support.patch \
    file://0006-add-conf-for-iMX-XCVR-sound-card.patch \
"
SRC_URI:append:imx-nxp-bsp = "${IMX_PATCH}"

PACKAGE_ARCH:imx-nxp-bsp = "${MACHINE_SOCARCH}"
```



EXAMPLE 2: IPROUTE2_%BBAPPEND

```
do_install:append () {  
    install -d ${D}/usr/include/tc  
    cp -a ${B}/include ${D}/usr/include  
    cp -a ${B}/tc/*.{h,c} ${D}/usr/include/tc  
}
```



EXAMPLE 3: MTD-UTILS_%._BBAPPEND

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"
SRC_URI:append = " file://ubihealthd.service"

inherit systemd

SYSTEMD_PACKAGES = "${PN}"
SYSTEMD_SERVICE:${PN} = "ubihealthd.service"
SYSTEMD_AUTO_ENABLE = "disable"

do_install:append() {
    install -d ${D}${systemd_unitdir}/system
    install -m 0644 ${WORKDIR}/ubihealthd.service ${D}${systemd_unitdir}/system/
    sed -i -e 's,@SBINDIR@,${sbindir},g' \
        -e 's,@SYSCONFDIR@,${sysconfdir},g' \
        ${D}${systemd_unitdir}/system/*.service
}
```



LISTING APPEND FILES

- The *bitbake-layers* command can be used to list existing append files:

```
$ bitbake-layers show-append
alsa-lib_1.2.1.2.bb:
    /opt/labs/ex/layers/meta-freescale/recipes-multimedia/alsa/alsa-lib_%.bbappend
alsa-state.bb:
    /opt/labs/ex/layers/meta-freescale-3rdparty/recipes-bsp/alsa-state/alsa-state.bbappend
    /opt/labs/ex/layers/meta-freescale/recipes-bsp/alsa-state/alsa-state.bbappend
    /opt/labs/ex/layers/meta-toradex-nxp/recipes-bsp/alsa-state/alsa-state.bbappend
base-files_3.0.14.bb:
    /opt/labs/ex/layers/meta-lmp/meta-lmp-base/recipes-core/base-files/base-files_%.bbappend
    /opt/labs/ex/layers/meta-toradex-torizon/recipes-core/base-files/base-files_%.bbappend
base-passwd_3.5.29.bb:
    /opt/labs/ex/layers/meta-toradex-torizon/recipes-core/base-passwd/base-passwd_3.5.29.bbappend
bind_9.11.32.bb:
    /opt/labs/ex/layers/meta-virtualization/recipes-core/bind/bind_%.bbappend
busybox_1.31.1.bb:
    /opt/labs/ex/layers/meta-yocto/meta-poky/recipes-core/busybox/busybox_%.bbappend
    /opt/labs/ex/layers/meta-virtualization/recipes-core/busybox/busybox_%.bbappend
    /opt/labs/ex/layers/meta-security/recipes-core/busybox/busybox_%.bbappend
    /opt/labs/ex/layers/meta-lmp/meta-lmp-base/recipes-core/busybox/busybox_%.bbappend
...
...
```

INCLUDE AND REQUIRE

- BitBake allows recipes to include files with the *include* and *require* directives.
 - *include*: if the file is not found, recipe processing continues normally.
 - *require*: if the file is not found, an error will occur while processing the recipe.
- A common use case of these directives is to split the recipe implementation into two files, to make maintenance easier or support multiple versions.
 - Version-dependent metadata (ex: *libmodbus_3.0.6.bb*)
 - Version-independent metadata (eg *libmodbus.inc*).

INCLUDE AND REQUIRE

```
$ tree meta-openembedded/meta-oe/recipes-extended/libmodbus
meta-openembedded/meta-oe/recipes-extended/libmodbus
├── libmodbus
│   └── Fix-float-endianness-issue-on-big-endian-arch.patch
├── libmodbus_3.0.6.bb
└── libmodbus_3.1.7.bb
    └── libmodbus.inc
```

```
$ cat meta-openembedded/meta-oe/recipes-extended/libmodbus/libmodbus_3.0.6.bb
require libmodbus.inc

SRC_URI[md5sum] = "c80f88b6ca19cab4cefffc195ca07771"
SRC_URI[sha256sum] = "046d63f10f755e2160dc56ef681e5f5ad3862a57c1955fd82e0ce036b69471b6"
```

VARIABLE FLAGS

- Variable flags (**varflags**) is a BitBake feature for associating extra information with variables and tasks.
- In the example below, the variable *FOO* has two flags, *a* and *b*, with values *123* and *456* respectively.

```
FOO[a] = "123"
FOO[b] = "456"
```

- Some interesting varflags use cases include enabling software functionality (via *PACKAGECONFIG*) and controlling the execution of tasks.

```
do_configure[noexec] = "1"
```

- More information is available in the [BitBake's User Manual](#).



PACKAGECONFIG

- The `PACKAGECONFIG` variable is used in recipes as a mechanism to enable functionality when building software.
- The recipe defines the available features that could be enabled:

```
PACKAGECONFIG ??= ""
PACKAGECONFIG[f1] = "\n    --with-f1, \
    --without-f1, \
    build-deps-for-f1, \
    runtime-deps-for-f1, \
    runtime-recommends-for-f1, \
    packageconfig-conflicts-for-f1"
PACKAGECONFIG[f2] = "..."
```

- An append file can be used to enable the desired functionality:

```
PACKAGECONFIG:append = " f1"
```

PACKAGECONFIG

- For example, the recipe *apache2_2.4.53.bb* sets the following *PACKAGECONFIG* options:

```
$ cat meta-webserver/recipes-httpd/apache2/apache2_2.4.53.bb | grep PACKAGECONFIG
PACKAGECONFIG ?= "${@bb.utils.filter('DISTRO_FEATURES', 'selinux', d)}"
PACKAGECONFIG[selinux] = "--enable-selinux,--disable-selinux,libsopenssl,libsopenssl"
PACKAGECONFIG[openldap] = "--enable-ldap --enable-authnz-ldap,--disable-ldap --disable-authnz-ldap
PACKAGECONFIG[zlib] = "--enable-deflate,,zlib,zlib"
```

- An append file can be created to compile Apache2 with *zlib* support:

```
# cat meta-labworks/recipes-httpd/apache2/apache2_%.bbappend
PACKAGECONFIG:append = " zlib"
```



PACKAGE VERSIONING

- The variables *PV*, *PR* and *PE* are part of the package versioning scheme when a recipe is built.
 - *PV* (Package Version): represents the package version, usually the software version, read from the recipe filename (e.g. *pciutils_3.7.0.bb*).
 - *PR* (Package Revision): contains *r0* by default and should be incremented when a change in the recipe impacts the way the software is built.
 - *PE* (Package Epoch): is by default empty and should be defined if the package versioning scheme is changed.



BITBAKE -S

```
$ bitbake -s
Recipe Name
=====
...
liburi-perl-native :5.08-r0
libusb1           :1.0.24-r0
libusb1-native    :1.0.24-r0
libuv             :1.42.0-r0
libuv-native      :1.42.0-r0
libva             :2.12.0-r0
libva-initial     :2.12.0-r0
libva-utils       :2.12.0-r0
libvorbis         :1.3.7-r0
libwebp           :1.2.1-r0
libwpe             :1.10.1-r0
libx11             1:1.7.2-r0
libx11-native     1:1.7.2-r0
libxau             1:1.0.9-r0
libxau-native     1:1.0.9-r0
libxcb             :1.14-r0
libxcb-native     :1.14-r0
libcomposite       1:0.4.5-r0
libcomposite-native 1:0.4.5-r0
```

VERSIONS AND PRIORITIES

- If there is more than one version of the same recipe, by default BitBake selects the most recent recipe.
- If recipes are in layers with different priorities (*BBFILE_PRIORITY*), the recipe in the highest priority layer is selected.
- The *PREFERRED_VERSION* variable can be used in a configuration file to define the desired version.

```
PREFERRED_VERSION_gtk+ ?= "2.13.3"
```

VERSIONS AND PRIORITIES

- A recipe can use the *DEFAULT_PREFERENCE* variable to increase or decrease its priority (0 by default).

```
DEFAULT_PREFERENCE = "-1"
```

- When in doubt about which version is being selected, check *PV* with *bitbake -e*:

```
$ bitbake busybox -e | grep ^PV=
PV="1.34.1"
```



INSPECTING VERSIONS

- The *bitbake-layers* command can be used to show recipes available in multiple layers:

```
$ bitbake-layers show-overlaid
podman:
meta-toradex-torizon 3.4.1+gitAUTOINC+a6493ae690
meta-virtualization 2.0.1+gitAUTOINC+a11c4ead10
python3-colorama:
meta-virtualization 0.3.9
meta-python 0.4.3
python3-docker:
meta-lmp-base 4.2.1
meta-virtualization 4.2.0
python3-docker-compose:
meta-lmp-base 1.26.0
meta-virtualization 1.25.4
python3-scapy:
meta-security 2.4.3
meta-python 0.25
python3-websocket-client:
meta-virtualization 0.57.0
meta-python 0.56.0
...
```

DEBUGGING RECIPES

- Several techniques can be used to debug problems in recipes:
 - Recipe processing logs analysis.
 - BitBake logs analysis.
 - Variable inspection.
 - Cleanup of build caches.
 - The devshell task.
 - Inspecting the content of generated packages.
- The [Debugging Tools and Techniques](#) section of the Yocto Project's development manual has detailed information on debugging recipes.

RECIPE PROCESSING LOGS

- In case of errors processing a recipe, check the logs generated in `${WORKDIR}/temp`:

```
$ ls -1 temp/log.*  
temp/log.do_cleansstate  
temp/log.do_cleansstate.2772038  
temp/log.do_compile  
temp/log.do_compile.2779663  
temp/log.do_configure  
temp/log.do_configure.2778094  
...  
  
$ ls -1 temp/run.*  
temp/run.do_configure  
temp/run.do_configure.2778094  
temp/run.do_fetch  
temp/run.do_fetch.2776775  
temp/run.do_patch  
temp/run.do_patch.2776970  
...
```

BITBAKE LOGS

- General BitBake issues can be viewed with the `-D` parameter, which can be set multiple times to increase the debug level.

```
$ bitbake -DDD virtual/kernel
DEBUG: Using cache in '/home/sprado/workspace/yocto/training/build/cache/bb_unihashes.dat'
DEBUG: Processing core in collection list
DEBUG: Processing yocto in collection list
DEBUG: Processing yoctobsp in collection list
DEBUG: Processing meta-labworks in collection list
DEBUG: Sanity-checking tuning 'core2-64' (default) features:
DEBUG: m64: IA32e (x86_64) ELF64 standard ABI
DEBUG: core2: Enable core2 specific processor optimizations
Loading cache...DEBUG: Cache: default: Cache dir: /home/sprado/workspace/yocto/training/build/tmp/cache
Loaded 1471 entries from dependency cache.
DEBUG: Target list: ['virtual/kernel']
DEBUG: providers for virtual/kernel are: ['linux-yocto']
DEBUG: selecting /home/sprado/workspace/yocto/training/layers/poky/meta/recipes-kernel/linux/linux-yocto
NOTE: selecting linux-yocto to satisfy virtual/kernel due to PREFERRED_PROVIDERS
DEBUG: sorted providers for virtual/kernel are: ['/home/sprado/workspace/yocto/training/layers/poky/meta/recipes-kernel/linux/linux-yocto']
DEBUG: adding /home/sprado/workspace/yocto/training/layers/poky/meta/recipes-kernel/linux/linux-yocto
DEBUG: Added dependencies ['xz-native', 'bc-native', 'virtual/x86_64-poky-linux-binutils', 'virtual/x86_64-poky-linux-kernel']
DEBUG: Added runtime dependencies ['kernel-base', 'kernel-image-bzimage'] for /home/sprado/workspace/yocto/training/build/tmp/work/core2-64-poky-linux/virtual/kernel/1.0-r0
NOTE: Resolving any missing task queue dependencies
```

VARIABLE INSPECTION

- Sometimes it is necessary to inspect the values of variables while processing a recipe (perhaps because the assignment being done is not working as expected).
- This can be done with BitBake -e parameter.

```
$ bitbake busybox -e | grep ^PV=
PV="1.35.0"
```

- The same result can be obtained with the *bitbake-getvar* command:

```
$ bitbake-getvar -r busybox --value PV
1.35.0
```

CACHES AND TASK EXECUTION

- Use BitBake `-c` option to execute a specific task in the recipe:

```
$ bitbake unzip -c compile
```

- If the task has already executed, running again won't have any effect. To force the task to run, its build cache should be removed.

```
$ bitbake unzip -c cleansstate && bitbake unzip
```

- Alternatively, the `-f` parameter can be used to force a task to run:

```
$ bitbake -f -c compile unzip && bitbake unzip
```

DEBUGGING WITH DEVSHELL

- The *do_devshell* task allows opening a new terminal to debug the recipe.

```
$ bitbake unzip -c devshell
```

- In the devshell terminal, all environment variables needed for cross-compilation are defined, making it possible to directly use compilation commands (make, cmake, autotools, etc).

```
$ echo $CC
x86_64-poky-linux-gcc -m64 -march=core2 -mtune=core2 -msse3 -mfpmath=sse -fstack-protector-strong
```

- Within the devshell, it is possible to directly execute recipe tasks:

```
$ ./temp/run.do_compile
```

INSPECTING PACKAGES

- Sometimes processing a recipe might return success, but the end result is not what is expected.
- For example, the recipe is processed, but the generated packages do not contain the expected artifacts.
- To analyze these types of problems, the *oe-pkgdata-util* tool can be used.

OE-PKGDATA-UTIL

```
$ oe-pkgdata-util
oe-pkgdata-util: error: the following arguments are required: <subcommand>
usage: oe-pkgdata-util [-h] [-d] [-p PKGDATA_DIR] <subcommand> ...
```

OpenEmbedded pkgdata tool - queries the pkgdata files written out during do_package

options:

-h, --help	show this help message and exit
-d, --debug	Enable debug output
-p PKGDATA_DIR, --pkgdata-dir PKGDATA_DIR	Path to pkgdata directory (determined automatically if not specified)

subcommands:

lookup-pkg	Translate between recipe-space package names and runtime package names
list-pkgs	List packages
list-pkg-files	List files within a package
lookup-recipe	Find recipe producing one or more packages
package-info	Show version, recipe and size information for one or more packages
find-path	Find package providing a target path
read-value	Read any pkgdata value for one or more packages
glob	Expand package name glob expression

Use oe-pkgdata-util <subcommand> --help to get help on a specific command

OE-PKGDATA-UTIL

```
$ oe-pkgdata-util list-pkgs lsof*
lsof
lsof-dbg
lsof-dev
lsof-doc
lsof-src

$ oe-pkgdata-util list-pkg-files lsof
lsof:
    /usr/sbin/lsof

$ oe-pkgdata-util find-path /lib/libblkid.so.1
util-linux-libblkid: /lib/libblkid.so.1

$ oe-pkgdata-util lookup-recipe util-linux-libblkid
util-linux

$ oe-pkgdata-util package-info util-linux-libblkid
util-linux-libblkid 2.37.2-r0 util-linux 2.37.2-r0 util-linux-libblkid: 347209
```

QA CHECKS

- One of the possible reasons for an error when processing a recipe is the quality checks (QA checks) of the build system.
- When processing a recipe, the build system will perform several checks to avoid common problems and report them to the user.
- Some checks just generate warning messages, but the execution completes successfully.
- Other checks generate error messages, failing to build the recipe.

EXAMPLE: QA CHECK

```
$ bitbake busybox
...
ERROR: busybox-1.34.1-r0 do_package: QA Issue: busybox: Files/directories were installed but not shi
  /usr/share
  /usr/share/udhcpc
  /usr/share/udhcpc/default.script
Please set FILES such that these items are packaged. Alternatively if they are unneeded, avoid insta
busybox: 3 installed and not shipped files. [installed-vs-shipped]
ERROR: busybox-1.34.1-r0 do_package: Fatal QA errors found, failing task.
ERROR: Logfile of failure stored in: /home/sprado/workspace/yocto/training/build/tmp/work/core2-64-p
ERROR: Task (/home/sprado/workspace/yocto/training/layers/poky/meta/recipes-core/busybox/busybox_1.3
INFO: Tasks Summary: Attempted 736 tasks of which 735 didn't need to be rerun and 1 failed.

Summary: 1 task failed:
  /home/sprado/workspace/yocto/training/layers/poky/meta/recipes-core/busybox/busybox_1.34.1.bb:do_p
Summary: There were 2 ERROR messages shown, returning a non-zero exit code.
```

HANDLING QA ERRORS

- While it's possible to bypass QA checks, it is recommended to try to understand the reason and fix them.
- A list of all existing QA checks is documented in the Yocto Project's Reference Manual:
<https://docs.yoctoproject.org/ref-manual/qa-checks.html>
- If it's not possible or practical to solve a QA problem, the *INSANE_SKIP* variable can be used to disable the check.

```
INSANE_SKIP:${PN} = "installed-vs-shipped"
```



LAB 5

CUSTOMIZING RECIPES



YOCTO PROJECT

IMAGE CUSTOMIZATION



CUSTOMIZING IMAGES

- During the development of a Linux distribution with the Yocto Project, it will be necessary to customize the final rootfs image, for several reasons:
 - Add or remove software components (packages).
 - Add or remove files (configuration files, shell scripts, blobs, etc).
 - Add or remove users and configure passwords.
 - Configure file and directory permissions.
 - Modify the size, format and type of the final image (ext4, f2fs, btrfs, etc).
 - And so on!



WHERE TO CUSTOMIZE?

- An image can be customized in different places:
 - Changes can be made in the *local.conf* file.
 - Customizations can be done on an image recipe.
 - Customizations can be done in a distro configuration file.

LOCAL.CONF

- The easiest way to customize an image is to change the *local.conf* file.
- However, using *local.conf* to customize an image has some disadvantages, including:
 - It doesn't allow all types of customization.
 - The change affects all builds that use *local.conf*.
 - It's harder to maintain and doesn't scale.
- For this reason, the *local.conf* file can be used for simple modifications and prototyping, but changing it is not a recommended approach for production images customization.

IMAGE RECIPES

- Image recipes allow for greater flexibility and control in customizing the final operating system image.
- Image recipes are "special recipes" that describe:
 - Packages that must be installed on the image.
 - Customizations that must be done in the generated rootfs.
 - Other parameters for generating the final image (format, type, size, etc.).



IMAGE RECIPES

- Usually, image recipes are stored in a directory called *images*. Example:

```
$ ls poky/meta/recipes-core/images/
build-appliance-image           core-image-minimal-initramfs.bb
build-appliance-image_15.0.0.bb   core-image-minimal-mtdutils.bb
core-image-base.bb               core-image-ptest-all.bb
core-image-minimal.bb            core-image-ptest-fast.bb
core-image-minimal-dev.bb       core-image-tiny-initramfs.bb
```

- There are several image recipes available in the Yocto Project:
<https://docs.yoctoproject.org/ref-manual/images.html>
- It is very common to create new image recipes when working with the Yocto Project.

CORE-IMAGE-MINIMAL.BB

```
SUMMARY = "A small image just capable of allowing a device to boot."  
  
IMAGE_INSTALL = "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"  
  
IMAGE_LINGUAS = ""  
  
LICENSE = "MIT"  
  
inherit core-image  
  
IMAGE_ROOTFS_SIZE ?= "8192"  
IMAGE_ROOTFS_EXTRA_SPACE:append = "${@bb.utils.contains("DISTRO_FEATURES", "systemd", " + 4096", "",
```

CREATING IMAGE RECIPES

- When creating an image recipe, we can build on top of existing recipes.
 - We can simply copy an existing image recipe to our layer and customize it.
 - We can build on top of an existing recipe using the *require* directive:

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

- After creating the image recipe, image-specific variables can be used to customize the image, like *IMAGE_INSTALL*, *CORE_IMAGE_EXTRA_INSTALL* and *IMAGE_FEATURES*.

CORE-IMAGE-MINIMAL-MTDUTILS.BB

```
require core-image-minimal.bb

DESCRIPTION = "Small image capable of booting a device with support for the \
Minimal MTD Utilities, which let the user interact with the MTD subsystem in \
the kernel to perform operations on flash devices."

IMAGE_INSTALL += "mtd-utils"
```



ADDING PACKAGES

- The `IMAGE_INSTALL` and `CORE_IMAGE_EXTRA_INSTALL` variables can be used in an image recipe to add packages to the image.

```
IMAGE_INSTALL += "mtd-utils"
```

- These variables must be set to the package name, not the recipe name!
 - Remember that one recipe can generate multiple packages, and the generated packages will not necessarily have the same name used in the recipe!



RECIPES VS PACKAGES

```
$ bitbake pcre
...
NOTE: Tasks Summary: Attempted 717 tasks of which 717 didn't need to be rerun and all succeeded.

$ cat conf/local.conf | grep CORE_IMAGE_EXTRA_INSTALL
CORE_IMAGE_EXTRA_INSTALL += "pcre"

$ bitbake core-image-minimal
...
ERROR: Nothing RPROVIDES 'pcre' (but /opt/labs/ex/layers/poky/meta/recipes-core/images/core-image-mi
NOTE: Runtime target 'pcre' is unbuildable, removing...
Missing or unbuildable dependency chain was: ['pcre']
ERROR: Required build target 'core-image-minimal' has no buildable providers.
Missing or unbuildable dependency chain was: ['core-image-minimal', 'pcre']

$ bitbake pcre -e | grep ^PACKAGES=
PACKAGES="libpcrecpp libpcreposix pcregrep pcregrep-doc pcretest pcretest-doc libpcre-ptest
libpcre-src libpcre-dbg libpcre-staticdev libpcre-dev libpcre-doc libpcre-locale libpcre"
```

MANIFEST FILE

- Packages installed in an image can be displayed via the image's manifest file:

```
$ cat tmp/deploy/images/qemux86-64/core-image-minimal-qemux86-64.manifest
base-files qemux86_64 3.0.14
base-passwd core2_64 3.5.29
busybox core2_64 1.34.1
busybox-hwclock core2_64 1.34.1
busybox-syslog core2_64 1.34.1
busybox-udhcpc core2_64 1.34.1
eudev core2_64 3.2.10
init-ifupdown qemux86_64 1.0
init-system Helpers-service core2_64 1.60
initscripts core2_64 1.0
initscripts-functions core2_64 1.0
kernel-5.14.15-yocto-standard qemux86_64 5.14.15+git0+edb4dc09c5_f04b30fc15
kernel-module-uvesafb-5.14.15-yocto-standard qemux86_64 5.14.15+git0+edb4dc09c5_f04b30fc
ldconfig core2_64 2.34
libblkid1 core2_64 2.37.2
libc6 core2_64 2.34
libkmod2 core2_64 29
...
```

PACKAGEGROUPS

- Packagegroups allow grouping software packages with a common goal. Examples:
 - Software packages to enable a graphical stack.
 - Software packages with debugging tools.
 - Software packages with project-specific applications.
- A packagegroup is nothing more than a recipe that inherits the *packagegroup* class to group packages with common functionality.

PACKAGEGROUPS RECIPES

- Typically, packagegroups are located in a directory called *packagegroups*:

```
$ ls poky/meta/recipes-core/packagegroups/
nativesdk-packagegroup-sdk-host.bb
packagegroup-base.bb
packagegroup-core-boot.bb
packagegroup-core-buildessential.bb
packagegroup-core-eclipse-debug.bb
packagegroup-core-nfs.bb
packagegroup-core-sdk.bb
packagegroup-core-ssh-dropbear.bb
packagegroup-core-ssh.openssh.bb
packagegroup-core-standalone-sdk-target.bb
packagegroup-core-tools-debug.bb
packagegroup-core-tools-profile.bb
packagegroup-core-tools-testapps.bb
packagegroup-cross-canadian.bb
packagegroup-go-cross-canadian.bb
packagegroup-go-sdk-target.bb
packagegroup-rust-cross-canadian.bb
packagegroup-self-hosted.bb
```

PACKAGEGROUP-CORE-ECLIPSE-DEBUG.BB

```
SUMMARY = "Remote debugging tools for Eclipse integration"

inherit packagegroup

RDEPENDS:${PN} = "\\
    gdbserver \
    tcf-agent \
    openssh-sftp-server \
    "
```



USING PACKAGEGROUPS

- A packagegroup can be added to an image using the *IMAGE_INSTALL* and *CORE_IMAGE_EXTRA_INSTALL* variables.
- For example, there is a packagegroup to add Weston support in the image:

```
$ ls poky/meta/recipes-graphics/packagegroups/packagegroup-core-weston.bb
```

- To install this packagegroup on the image, just add it to the *IMAGE_INSTALL* or *CORE_IMAGE_EXTRA_INSTALL* variables.

```
IMAGE_INSTALL += "packagegroup-core-weston"
```

IMAGE FEATURES

- Image features allow enabling some "functionality" in the image via the *IMAGE_FEATURES* and *EXTRA_IMAGE_FEATURES* variables.
- Some image features are directly related to installing additional packages:

```
IMAGE_FEATURES = "tools-debug"
```

- While others will directly change the content of the generated image.

```
IMAGE_FEATURES = "read-only-rootfs"
```

- Documentation of existing image features:
<https://docs.yoctoproject.org/ref-manual/features.html#image-features>



EXAMPLE: IMAGE FEATURES

```
$ cat poky/meta/classes/core-image.bbclass | grep FEATURE_PACKAGES
FEATURE_PACKAGES_weston = "packagegroup-core-weston"
FEATURE_PACKAGES_x11 = "packagegroup-core-x11"
FEATURE_PACKAGES_x11-base = "packagegroup-core-x11-base"
FEATURE_PACKAGES_x11-sato = "packagegroup-core-x11-sato"
FEATURE_PACKAGES_tools-debug = "packagegroup-core-tools-debug"
FEATURE_PACKAGES_eclipse-debug = "packagegroup-core-eclipse-debug"
FEATURE_PACKAGES_tools-profile = "packagegroup-core-tools-profile"
FEATURE_PACKAGES_tools-testapps = "packagegroup-core-tools-testapps"
FEATURE_PACKAGES_tools-sdk = "packagegroup-core-sdk packagegroup-core-standalone-sdk-target"
FEATURE_PACKAGES_nfs-server = "packagegroup-core-nfs-server"
FEATURE_PACKAGES_nfs-client = "packagegroup-core-nfs-client"
FEATURE_PACKAGES_ssh-server-dropbear = "packagegroup-core-ssh-dropbear"
FEATURE_PACKAGES_ssh-server.openssh = "packagegroup-core-ssh.openssh"
FEATURE_PACKAGES_hwcodecs = "${MACHINE_HWCODECS}"

$ cat poky/meta/classes/rootfs-postcommands.bbclass | grep IMAGE_FEATURES
ROOTFS_POSTPROCESS_COMMAND += '${@bb.utils.contains_any("IMAGE_FEATURES", [ 'debug-tweaks', 'empty-r
ROOTFS_POSTPROCESS_COMMAND += '${@bb.utils.contains_any("IMAGE_FEATURES", [ 'debug-tweaks', 'allow-e
ROOTFS_POSTPROCESS_COMMAND += '${@bb.utils.contains_any("IMAGE_FEATURES", [ 'debug-tweaks', 'allow-r
ROOTFS_POSTPROCESS_COMMAND += '${@bb.utils.contains_any("IMAGE_FEATURES", [ 'debug-tweaks', 'post-in
ROOTFS_POSTPROCESS_COMMAND += '${@bb.utils.contains("IMAGE_FEATURES", "read-only-rootfs", "read_only,
APPEND:append = '${@bb.utils.contains("IMAGE_FEATURES", "read-only-rootfs", " ro", "", d)}'
SYSTEMD_DEFAULT_TARGET ?= '${@bb.utils.contains_any("IMAGE_FEATURES", [ "x11-base", "weston" ], "gra
```



REMOVING PACKAGES

- The *PACKAGE_EXCLUDE* variable allows to exclude a package from the image.
 - In this case, the build process may fail if some other package depends on it.
- The *BAD_RECOMMENDATIONS* variable allows removing from the final image a package that was installed through the *RRECOMMENDS* variable.
- The *NO_RECOMMENDATIONS* variable allows removing from the image all packages that were installed through the *RRECOMMENDS* variable.

ADDING USERS AND GROUPS

- The *useradd* and *extrausers* classes can be used to add users and groups to the generated image.
- It's recommended to use the *useradd* class when the user/group to be created is associated with a package installed in the image.
<https://docs.yoctoproject.org/ref-manual/classes.html#useradd-bbclass>
- It's recommended to use the *extrausers* class when the user/group is global to the image, and not associated with any specific program or package.
<https://docs.yoctoproject.org/ref-manual/classes.html#extrausers-bbclass>

EXAMPLE: USERADD

```
inherit useradd

USERADD_PACKAGES = "${PN}"

USERADD_PARAM_${PN} = "-d /home/user1 -r -s /bin/bash user1; -d /home/user2 -r -s /bin/bash user2"

GROUPADD_PARAM_${PN} = "group1; group2"

do_install:append () {
    install -d -m 755 ${D}${datadir}/user1
    install -d -m 755 ${D}${datadir}/user2

    chown -R user1 ${D}${datadir}/user1
    chown -R user2 ${D}${datadir}/user2

    chgrp -R group1 ${D}${datadir}/user1
    chgrp -R group2 ${D}${datadir}/user2
}

FILES_${PN} = "${datadir}/user1/* ${datadir}/user2/*"
```

EXAMPLE: EXTRAUSERS

```
inherit extrausers
EXTRA_USERS_PARAMS = "\
    useradd -p '' tester; \
    groupadd developers; \
    userdel nobody; \
    groupdel -g video; \
    groupmod -g 1020 developers; \
    usermod -s /bin/sh tester; \
    "
```

FILE AND DIRECTORY PERMISSIONS

- By default, the build system uses the *fs-perms.txt* file (located in OpenEmbedded-Core) to set the file and directory permissions of the generated rootfs.
- It's possible to change this configuration file via the *FILESYSTEM_PERMS_TABLES* variable.
- This may be necessary if you want to apply global permission settings on the image (settings associated with an application should be made in the corresponding recipe).
- This configuration must be performed in a separate layer, possibly in a distro configuration file.

FS-PERMS.TXT

```
# This file contains a list of files and directories with known permissions.  
# It is used by the packaging class to ensure that the permissions, owners  
# and group of listed files and directories are in sync across the system.  
#  
# The format of this file  
#  
#<path>          <mode>  <uid>   <gid>   <walk>  <fmode> <fuid>  <fgid>  
  
...  
  
# Documentation should always be corrected  
${mandir}          0755    root    root    true    0644    root    root  
${infodir}          0755    root    root    true    0644    root    root  
${docdir}           0755    root    root    true    0644    root    root  
${datadir}/gtk-doc  0755    root    root    true    0644    root    root  
  
# Fixup locales  
${datadir}/locale   0755    root    root    true    0644    root    root  
  
...
```

ADDING FILES

- A common need when customizing images is the addition of files (blobs, shell scripts, configuration files, etc).
- If the file is related to a certain software, we can change the corresponding recipe to add it to the image.
- In case the file is not associated with any software, we can create a specific recipe to add it to the image.

EXAMPLE: ADDING FILES

```
DESCRIPTION = "Some closed source library from hell"  
  
LICENSE = "CLOSED"  
  
SRC_URI = "http://hell.com/mylib.so;unpack=false"  
SRC_URI[sha256sum] = "2ea483c3c4ce87f4a3c851077c3b8ea8e7d5539 8bf856fa3b0765e65085617bd"  
  
do_install() {  
    install -d ${D}${libdir}  
    install -m 0755 ${WORKDIR}/mylib.so ${D}${libdir}  
}  
  
FILES_${PN} = "${libdir}/mylib.so"
```



POST-INSTALL SCRIPTS

- Packages may contain post-install scripts, which allow the execution of commands during their installation (during rootfs generation or at runtime).
- They are useful to prepare the root filesystem for a certain package (create directories and temporary files, set permissions, change configuration files, etc).
- A post-installation script can be defined via the *pkg_postinst* function:

```
pkg_postinst:<PACKAGENAME>() {  
    # commands to execute  
}
```

- There is also support for pre-install, pre-uninstall, and post-uninstall scripts through the *pkg_preinst*, *pkg_prerm* and *pkg_postrm* functions, respectively.



EXAMPLE: POST-INSTALL SCRIPTS

```
$ cat meta-openembedded/meta-oe/recipes-shells/zsh/zsh_5.8.bb
...
pkg_postinst:${PN} () {
    touch ${sysconfdir}/shells
    grep -q "bin/zsh" ${sysconfdir}/shells || echo /bin/zsh >> ${sysconfdir}/shells
    grep -q "bin/sh" ${sysconfdir}/shells || echo /bin/sh >> ${sysconfdir}/shells
}

$ cat poky/meta/recipes-connectivity/avahi/avahi_0.8.bb
...
pkg_postinst:avahi-daemon () {
    if [ -z "$D" ]; then
        killall -q -HUP dbus-daemon || true
    fi
}
```



POST-INSTALL ON BOOT

- Sometimes it is necessary to delay running a post-installation script until the device first boots.
- It might be useful when it's required to run the post-installation script directly on the target.
- This can be done via the *pkg_postinst_ontarget()* function:

```
$ cat meta-openembedded/meta-oe/recipes-bsp/nvme-cli/nvme-cli_1.13.bb
...
pkg_postinst_ontarget:${PN}() {
    ${sbindir}/nvme gen-hostnqn > ${sysconfdir}/nvme/hostnqn
    ${bindir}/uuidgen > ${sysconfdir}/nvme/hostid
}
```

ROOTFS_POSTPROCESS_COMMAND

- An easy way to execute a command after the rootfs is created is via the `ROOTFS_POSTPROCESS_COMMAND` variable.
- The functions added to this variable are executed at the end of the rootfs generation process.
- For example, the code below is intended to change the password of the *admin* user:

```
run_set_admin_pass () {
    sed 's%^admin:[^:]*%admin:$6$3WwbKfr1$4vb1knvGr6FcDe:\
        %' ${IMAGE_ROOTFS}/etc/shadow \
        ${IMAGE_ROOTFS}/etc/shadow.new;
    mv ${IMAGE_ROOTFS}/etc/shadow.new \
        ${IMAGE_ROOTFS}/etc/shadow ;"
}
ROOTFS_POSTPROCESS_COMMAND += " run_set_admin_pass ; "
```

CHANGING THE IMAGE FORMAT

- The *IMAGE_FSTYPES* variable can be used to change the rootfs image format.

```
IMAGE_FSTYPES = "ext4"
```

- A list of existing image types is available in the Yocto Project reference guide.
https://docs.yoctoproject.org/ref-manual/variables.html#term-IMAGE_TYPES
- The *IMAGE_CMD* variable can be used to define custom image formats (see *poky/meta/classes/image_types.bbclass* for more examples):

```
IMAGE_CMD:jffs2 = "mkfs.jffs2 --root=${IMAGE_ROOTFS} --faketime \  
--output=${IMGDEPLOYDIR}/${IMAGE_NAME}${IMAGE_SUFFIX}.jffs2 \  
${EXTRA_IMAGECMD}"
```

CHANGING THE IMAGE SIZE

- The *IMAGE_ROOTFS_SIZE* variable can be used to set the final rootfs size (in KBs).

```
IMAGE_ROOTFS_SIZE = "1048576"
```

- If the *IMAGE_ROOTFS_SIZE* variable is not defined, or if its value is not enough to support the generated system, the image will be generated by multiplying the rootfs size by the *IMAGE_OVERHEAD_FACTOR* variable, which by default has the value 1.3, generating an image 30% larger than the minimum required value.
- The *IMAGE_ROOTFS_EXTRA_SPACE* variable can be used to force an extra space when creating the image (in KBs).

```
IMAGE_ROOTFS_EXTRA_SPACE = "524288"
```



CREATING DISTRIBUTIONS

- Poky is the reference distribution used by default in builds with the Yocto Project.
- To have more control over package selection, build options and other low-level settings, a custom distribution can be created.
- Distributions are defined in distro configuration files, stored in layers in the *conf/distro/* directory.

EXAMPLE: POKY DISTRIBUTION

```
$ tree poky/meta-poky/conf/distro/
poky/meta-poky/conf/distro/
├── include
│   ├── gcsections.inc
│   ├── poky-distro-alt-test-config.inc
│   ├── poky-floating-revisions.inc
│   └── poky-world-exclude.inc
├── poky-altcfg.conf
├── poky-bleeding.conf
└── poky.conf
    └── poky-tiny.conf
```



CREATING A DISTRIBUTION

- We can use Poky or OpenEmbedded-Core as a reference to create a distribution configuration file:
 - *poky/meta-poky/conf/distro/poky.conf*
 - *poky/meta/conf/distro/defaultsetup.conf*
- In the configuration file, distro-related variables can be used to customize the distribution (*TCMODE*, *DISTRO_NAME*, *DISTRO_VERSION*, *TCLIBC*, etc).
- To use the distribution, the *DISTRO* variable needs to be defined in *local.conf* with the same name as the distro configuration file.



DISTRO VARIABLES

Variable	Description
DISTRO	Distribution name
DISTRO_NAME	Long distribution name
DISTRO_VERSION	Distribution Version
TCMODE	Toolchain
TCLIBC	C library (glibc, musl, newlib, baremetal)
INIT_MANAGER	Init system (sysvinit, systemd, mdev-busybox)

DISTRO VARIABLES

Variable	Description
DISTRO_EXTRA_RDEPENDS	Add packages to the image
DISTRO_EXTRA_RRECOMMENDS	Add packages to the image (if they exist)
PREFERRED_VERSION	Set/force a Package Version
PACKAGE_CLASSES	Package type to be used
DISTROVERRIDES	Distribution-specific overrides
MIRRORS/PREMIRRORS	Alternative sources for source code download

DISTRO FEATURES

- Distro features allow to control the software installed in the image:
 - Enable the installation of additional packages.
 - Change the behavior of configuration scripts.
- Distro features are defined in distro configuration files via the *DISTRO_FEATURES* variable:

```
DISTRO_FEATURES = "opengl ppp ipv6 usrmerge"
```

- The complete list of distro features is documented in the Yocto Project's Reference Manual.

<https://docs.yoctoproject.org/ref-manual/features.html#distro-features>



PACKAGEGROUP-CORE-FULL-CMDLINE.BB

```
SUMMARY = "Standard full-featured Linux system"
DESCRIPTION = "Package group bringing in packages needed for a more traditional full-featured Linux
PR = "r6"

inherit packagegroup

...

RDEPENDS:packagegroup-core-full-cmdline-sys-services = "\\\n    at \
    cronie \
    logrotate \
    ${@bb.utils.contains('DISTRO_FEATURES', 'nfs', 'nfs-utils rpcbind', '', d)} \
    "
```

LIBPCAP_1.10.1.BB

```
SUMMARY = "Interface for user-level network packet capture"
DESCRIPTION = "Libpcap provides a portable framework for low-level network \
monitoring. Libpcap can provide network statistics collection, \
security monitoring and network debugging."
...
PACKAGECONFIG ??= "${@bb.utils.contains('DISTRO_FEATURES', 'bluetooth', 'bluez5', '', d)} \
${@bb.utils.filter('DISTRO_FEATURES', 'ipv6', d)} \
"
PACKAGECONFIG[bluez5] = "--enable-bluetooth,--disable-bluetooth,bluez5"
PACKAGECONFIG[dbus] = "--enable-dbus,--disable-dbus,dbus"
PACKAGECONFIG[ipv6] = "--enable-ipv6,--disable-ipv6,"
PACKAGECONFIG[libnl] = "--with-libnl,--without-libnl,libnl"
...
```



RUNTIME PACKAGE MANAGEMENT

- With the Yocto Project, it's possible to build an image with package management support.
- On the development machine, a package repository can be made available remotely through a WEB server to the target.
- On the target, package management tools are installed to make it possible to:
 - Install and remove packages from the image at runtime.
 - Maintain a database of installed packages.

ENABLING PACKAGE MANAGEMENT

- To prepare the target to support package management, just enable the *package-management* image feature.

```
IMAGE_FEATURES += "package-management"
```

- Some additional variables can be defined to configure the package server location (*PACKAGE_FEED_URIS*, *PACKAGE_FEED_ARCHS*, etc).
- On the host, each time packages are changed, it is necessary to regenerate the package index with the command below:

```
$ bitbake package-index
```

- More detailed information in the [Yocto Project's Development Manual](#).



TEMPLATE FILES

- It's common to store in a distro layer the template files used to initialize the build directory (*local.conf*, *bblayers.conf*).

```
$ ls meta-labworks/conf/  
bblayers.conf.sample  layer.conf  local.conf.sample
```

- When initializing the build directory, the environment variable *TEMPLATECONF* can be used to define the location of the template files:

```
$ export TEMPLATECONF=$PWD/layers/meta-labworks/conf  
$ source layers/poky/oe-init-build-env build-test
```

TIPS FOR CUSTOMIZATION

- Use *local.conf* for testing and prototyping, but concentrate changes in a separate layer, using image recipes and distro configuration files for production images customizations.
- Create image recipes for each product to be developed.
 - It is common to create image recipe variants for different build types (debug, production, etc).
- It is recommended to create a distribution with the settings and policies that will serve as the basis for your products.

MANAGING LAYERS

- Try to organize the metadata in layers to make it easier to maintain:
 - Separate BSP, distro and software metadata into different layers.
 - Different products might require different layers.
 - Metadata maintained by different teams might require different layers.
- Never mix application code with metadata.
- Always version control layers in Git repositories.
 - Tools like [repo](#) or [kas](#) can help manage the layers that will make up the project.

LAB 6

CUSTOMIZING IMAGES



YOCTO PROJECT

BSP DEVELOPMENT



BSP LAYER

- BSP (Board Support Package) is a term used to refer to all the source code necessary for a hardware platform to support a certain operating system (in our case, the Linux kernel).
- To support a new hardware platform in the Yocto Project, a BSP layer is required.
- A BSP layer contains hardware-related metadata, including:
 - Machine configuration files.
 - Bootloader and kernel recipes.
 - Recipes for hardware-specific libraries and applications.
 - Classes for the target platform (e.g. logic to generate a custom image format).

META-YOCTO-BSP

```
$ tree -L 3 poky/meta-yocto-bsp/
poky/meta-yocto-bsp/
├── conf
│   └── layer.conf
└── machine
    ├── beaglebone-yocto.conf
    ├── edgerouter.conf
    ├── genericx86-64.conf
    └── genericx86.conf
        └── include
└── lib
    └── oeqa
        ├── controllers
        │   └── selftest
        └── README.hardware.md
└── recipes-bsp
    ├── formfactor
    │   └── formfactor
    │       └── formfactor_0.0.bbappend
    └── gma500-gfx-check
        ├── gma500-gfx-check
        └── gma500-gfx-check_1.0.bb
└── recipes-graphics
    └── xorg-xserver
```

CREATING A BSP LAYER

- Before creating a BSP layer, check if there isn't already an implementation available for the target platform.
- It's very common for vendors and hardware manufacturers to support the Yocto Project by providing a BSP layer for their products.
- A good reference of existing machines and BSP layers is available in the OpenEmbedded Layers Index database:
<https://layers.openembedded.org/layerindex/branch/kirkstone/machines/>

CREATING A BSP LAYER

- A BSP layer has the same structure as other layers and can be created as follow:
 - Create a layer with the command *bitbake-layers create-layer*.
 - Add a machine configuration file to *conf/machine*.
- Optionally, other metadata can be created, including:
 - Bootloader recipes in *recipes-bsp/*.
 - Kernel recipes in *recipes-kernel/*.
- The BSP Development Guide has a lot of information on how to create and maintain BSP layers for the Yocto Project.
<https://docs.yoctoproject.org/bsp-guide/index.html>



STRUCTURE OF A BSP LAYER

```
$ tree meta-labworks-bsp
meta-labworks-bsp
├── classes
└── conf
    └── layer.conf
        └── machine
            └── my-machine.conf
├── recipes-bsp
└── recipes-core
    ├── recipes-graphics
    └── recipes-kernel
        └── linux
```



MACHINE CONFIGURATION FILE

- One of the main tasks when developing a BSP layer is implementing the machine configuration file.
- In this file, we must define all the necessary information about the hardware, including:
 - Hardware architecture and compiler optimization options.
 - Bootloader and kernel configuration.
 - Specific packages to support the hardware platform.
- Most of the time, we can create a machine configuration file based on other implementations for the same SoC.

CPU ARCHITECTURE

- In the machine configuration file, the CPU architecture is usually defined by including a file that contains the settings for a particular hardware platform.

```
require conf/machine/include/tune-cortexa9.inc
```

- Several architecture configuration files are available in OpenEmbedded-Core at *conf/machine/include/<arch>/*.
- In these files, some variables like *TARGET_ARCH*, *DEFAULTTUNE* and *TUNE_ARCH* are set so that the build system knows the hardware architecture. This way:
 - A toolchain optimized for the target platform is generated.
 - Optimized build flags are used for cross-compilation.



SOC_FAMILY AND MACHINEOVERIDES

- In the machine configuration file, it's possible to define specific overrides for the target platform.
- For this, we can use the variables *SOC_FAMILY* or *MACHINEOVERIDES*:

```
SOC_FAMILY =. "mx6:mx6dl:"  
include conf/machine/include/soc-family.inc
```

```
MACHINEOVERIDES =. "qemuall:"
```

- This feature allows the use of conditional assignment per architecture or machine in other metadata.

```
SRC_URI:append:mx6 = " file://fix-imx6-bug.patch"
```



BOOTLOADER AND KERNEL

- In the machine configuration file, we also need to define bootloader and kernel recipes that will be used to generate the Linux distribution for the target platform.
- This is done using the variables *PREFERRED_PROVIDER* and *PREFERRED_VERSION*.
- To use these variables, we need to understand how the *PROVIDES* mechanism and BitBake virtual packages work.



PROVIDES

- An specific software artifact (e.g. a kernel image) might be provided by more than one recipe.
- For a recipe to indicate that it provides a certain software artifact, there is the concept of virtual package.
- For example, a kernel recipe will use the *PROVIDES* variable to indicate that it provides the Linux kernel virtual package:

```
PROVIDES += "virtual/kernel"
```

- A bootloader recipe might do the same:

```
PROVIDES += "virtual/bootloader"
```

SELECTING RECIPES

- In the machine configuration file, the *PREFERRED_PROVIDER* and (optionally) *PREFERRED_VERSION* variables can be used to select the kernel recipe:

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ??= "5.15%"
```

- The same can be done for the bootloader:

```
PREFERRED_PROVIDER_virtual/bootloader = "u-boot-toradex"
PREFERRED_VERSION_u-boot-toradex = "2020.04%"
```

U-BOOT VARIABLES

Variable	Description
UBOOT_MACHINE	U-Boot board configuration file name
UBOOT_SUFFIX	U-Boot image extension (e.g. <i>bin</i>)
SPL_BINARY	SPL filename (e.g. <i>MLO</i>)
UBOOT_ENTRYPOINT	U-Boot Image Entry Point
UBOOT_LOADADDRESS	U-Boot Load Address
UBOOT_LOCALVERSION	String to concatenate in the U-Boot version

KERNEL VARIABLES

Variable	Description
KERNEL_IMAGETYPE	Kernel image name (e.g. <i>zImage</i>)
KERNEL_DEVICETREE	Name of device tree files (<i>.dtb</i>)
KERNEL_EXTRA_ARGS	Make extra arguments
KERNEL_MODULE_AUTOLOAD	Modules to automatically load at boot time
SERIAL_CONSOLES	Serial port (TTY) to start getty application

ADDING PACKAGES

Variable	Description
MACHINE_ESSENTIAL_EXTRA_RDEPENDS	Required machine-specific packages to install in the image (affects images that include <i>packagegroup-core-boot</i> , including <i>core-image-minimal</i>)
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS	Recommended machine-specific packages to install in the image (affects images that include <i>packagegroup-core-boot</i> , including <i>core-image-minimal</i>)



MACHINE FEATURES

- The `MACHINE_FEATURES` variable allows to specify the features supported by a hardware platform. Examples:
 - `touchscreen`: the hardware has a touchscreen interface.
 - `wifi`: the hardware has a WiFi interface.
 - `bluetooth`: the hardware has a Bluetooth interface.
- Enabling a machine feature might result in adding additional packages to the image and/or changing the way some software is compiled.
- Machine features are documented in the Yocto Project's Reference Manual.
<https://docs.yoctoproject.org/ref-manual/features.html#ref-features-machine>

EXAMPLE: PACKAGEGROUP-BASIC.BB

```
SUMMARY = "Basic task to get a device online"

PR = "r13"

...

RRECOMMENDS:${PN} = "\\
    ${MACHINE_EXTRA_RRECOMMENDS} \
    ${@bb.utils.contains("MACHINE_FEATURES", "usbhost", "usbutils", "", d)} \
    ${@bb.utils.contains("MACHINE_FEATURES", "alsa", "alsa-utils-alsamixer", "", d)} \
    ${@bb.utils.contains("MACHINE_FEATURES", "usbgadget", "kernel-module-g-ether kernel-module-g-ser\
\
    ${@bb.utils.contains("DISTRO_FEATURES", "bluetooth", "bluez5", "", d)} \
    ${@bb.utils.contains("DISTRO_FEATURES", "wifi", "iw wpa-supplicant", "", d)} \
\
    tzdata \
\
    cpufrequtils \
    htop \
"
\
..."
```

EXAMPLE: MATCHBOX-PANEL-2_2.11.BB

```
SUMMARY = "Simple GTK+ based panel for handheld devices"
DESCRIPTION = "A flexible always present 'window bar' for holding application \
    launchers and small 'applet' style applications"
HOMEPAGE = "http://matchbox-project.org"
BUGTRACKER = "http://bugzilla.yoctoproject.org/"

...
DEPENDS = "gtk+3 startup-notification dbus dbus-glib dbus-glib-native"
DEPENDS += " ${@bb.utils.contains("MACHINE_FEATURES", "acpi", "libacpi", "", d)}"
DEPENDS += " ${@bb.utils.contains("MACHINE_FEATURES", "apm", "apmd", "", d)}"

...
EXTRA_OECONF = "--enable-startup-notification --enable-dbus"
EXTRA_OECONF += " ${@bb.utils.contains("MACHINE_FEATURES", "acpi", "--with-battery=acpi", "", d)}"
EXTRA_OECONF += " ${@bb.utils.contains("MACHINE_FEATURES", "apm", "--with-battery=apm", "", d)}"

..."
```

MACHINE FEATURES VS DISTRO FEATURES

- Machine features and distro features work together to add support for certain functionality in the Linux distribution.
- Some features will only be enabled if they are in both variables (*DISTRO_FEATURES* and *MACHINE_FEATURES*).
- During metadata processing, BitBake combines the common values of *DISTRO_FEATURES* and *MACHINE_FEATURES* into a variable called *COMBINED_FEATURES*.

COMBINED_FEATURES

```
$ bitbake core-image-minimal -e | grep ^DISTRO_FEATURES=
DISTRO_FEATURES="acl alsalibc argp bluetooth debuginfod ext2 ipv4 ipv6 largefile pcmcia usbgadget
usbhost wifi xattr nfs zeroconf pci 3g nfc x11 vfat seccomp largefile opengl ptest multiarch
wayland vulkan pulseaudio sysvinit gobject-introspection-data ldconfig"

$ bitbake core-image-minimal -e | grep ^MACHINE_FEATURES=
MACHINE_FEATURES="alsa bluetooth usbgadget screen vfat x86 pci rtc qemu-usermode"

$ bitbake core-image-minimal -e | grep ^COMBINED_FEATURES=
COMBINED_FEATURES="usbgadget bluetooth alsalibc pci vfat"
```



EXAMPLE: IRDA-UTILS_0.9.18.BB

```
SUMMARY = "Common files for IrDA"
DESCRIPTION = "Provides common files needed to use IrDA. \
IrDA allows communication over Infrared with other devices \
such as phones and laptops."
HOMEPAGE = "http://irda.sourceforge.net/"
BUGTRACKER = "http://sourceforge.net/p/irda/bugs/"

...
RRECOMMENDS:${PN} = "\n    kernel-module-pxaficp-ir \
    kernel-module-irda \
    kernel-module-ircomm \
    kernel-module-ircomm-tty \
    kernel-module-irlan \
    ${@bb.utils.contains('DISTRO_FEATURES', 'ppp', 'kernel-module-irnet', '', d)} \
    kernel-module-irport \
    kernel-module-irtty \
    kernel-module-irtty-sir \
    kernel-module-sir-dev \
    ${@bb.utils.contains('COMBINED_FEATURES', 'usbhost', 'kernel-module-ir-usb', '', d)} "
...
"
```

MACHINE: BEAGLEBONE-YOCTO.CONF

```
#@TYPE: Machine
#@NAME: Beaglebone-yocto machine
#@DESCRIPTION: Reference machine configuration for http://beagleboard.org/bone and http://beagleboar

PREFERRED_PROVIDER_virtual/xserver ?= "xserver-xorg"
XSERVER ?= "xserver-xorg \
            xf86-video-modesetting \
            "
MACHINE_EXTRA_RRECOMMENDS = "kernel-modules kernel-devicetree"
EXTRA_IMAGEDEPENDS += "virtual/bootloader"
DEFAULTTUNE ?= "cortexa8hf-neon"
include conf/machine/include/arm/armv7a/tune-cortexa8.inc

IMAGE_FSTYPES += "tar.bz2 jffs2 wic wic.bmap"
EXTRA_IMAGECMD:jffs2 = "-lnp "
WKS_FILE ?= "beaglebone-yocto.wks"
MACHINE_ESSENTIAL_EXTRA_RDEPENDS += "kernel-image kernel-devicetree"
do_image_wic[depends] += "mtools-native:do_populate_sysroot dosfstools-native:do_populate_sysroot vi
SERIAL_CONSOLES ?= "115200;ttyS0 115200;tty00 115200;ttyAMA0"
SERIAL_CONSOLES_CHECK = "${SERIAL_CONSOLES}"
```



MACHINE: COLIBRI-IMX6.CONF

```
MACHINEOVERRIDES =. "mx6dl:"\n\ninclude conf/machine/include/imx-base.inc\ninclude conf/machine/include/arm/armv7a/tune-cortexa9.inc\n\nPREFERRED_PROVIDER_virtual/kernel:use-nxp-bsp ??= "linux-toradex"\nKBUILD_DEFCONFIG:use-nxp-bsp ?= "colibri_imx6_defconfig"\nKERNEL_DEVICETREE += "imx6dl-colibri-eval-v3.dtb imx6dl-colibri-cam-eval-v3.dtb \\ \n\timx6dl-colibri-aster.dtb imx6dl-colibri-iris.dtb \\ \n\timx6dl-colibri-iris-v2.dtb"\nKERNEL_DEVICETREE:use-mainline-bsp = "imx6dl-colibri-eval-v3.dtb"\nKERNEL_IMAGETYPE = "zImage"\n# The kernel lives in a separate FAT partition, don't deploy it in /boot/\nRRECOMMENDS:${KERNEL_PACKAGE_NAME}-base = ""\n\nIMX_DEFAULT_BOOTLOADER = "u-boot-toradex"\nPREFERRED_PROVIDER_u-boot-default-script = "u-boot-script-toradex"\n\nUBOOT_SUFFIX = "img"\nSPL_BINARY = "SPL"\nUBOOT_CONFIG ??= "spl"\nUBOOT_CONFIG[spl] = "colibri_imx6_defconfig,,u-boot.img"\nUBOOT_MAKE_TARGET = ""\nUBOOT_ENTRYPOINT:use-mainline-bsp = "0x10008000"
```

ENABLING THE MACHINE

- To enable the machine, first add its layer to the *bblayers.conf* file:

```
BBLAYERS ?= "\\\n    /opt/labs/ex/layers/poky/meta \\\n    /opt/labs/ex/layers/poky/meta-poky \\\n    /opt/labs/ex/layers/poky/meta-yocto-bsp \\\n    /opt/labs/ex/layers/meta-labworks \\\n    /opt/labs/ex/layers/meta-labworks-bsp \\\n"
```

- And set the *MACHINE* variable in *local.conf* with the name used in the machine configuration file.

```
MACHINE ??= "colibri-imx6"
```

CREATING BSP RECIPES

- A BSP layer will probably need some recipes, at least for compiling the bootloader and the Linux kernel.
- Creating a recipe for a BSP layer is no different from other layers (the only detail is that there are some specific BSP-related classes that we can use).
 - To compile U-Boot, it's necessary to include the file *recipes-bsp/u-boot/u-boot.inc* in the recipe (currently, there is no class to compile U-Boot).
 - To compile the Linux kernel, the *kernel* class can be inherited. Optionally, the *kernel-yocto* class can also be inherited to benefit from additional features such as the Kernel Advanced Metadata.
 - To compile a kernel module, just inherit the *module* class.

EXAMPLE: U-BOOT-TORADEX_2020.07.BB

```
SUMMARY = "U-Boot bootloader with support for Toradex Computer on Modules"
HOMEPAGE = "http://www.denx.de/wiki/U-Boot/WebHome"
SECTION = "bootloaders"
LICENSE = "GPL-2.0-or-later"
LIC_FILES_CHKSUM = "file://Licensing/README;md5=30503fd321432fc713238f582193b78e"

require recipes-bsp/u-boot/u-boot.inc

DEPENDS += "bc-native dtc-native bison-native"

PV = "2020.07+git${SRCPV}"
SRC_URI = " \
    git://git.toradex.com/u-boot-toradex.git;branch=${SRCBRANCH} \
    file://fw_env.config \
"
SRCBRANCH = "toradex_2020.07"
SRCREV = "ab862daf5d5a2eebf305c5c125f0463b0ff34161"

UBOOT_INITIAL_ENV = "u-boot-initial-env"

PROVIDES += "u-boot"

B = "${WORKDIR}/build"
S = "${WORKDIR}/git"
```

EXAMPLE: LINUX-YOCTO-CUSTOM.BB

```
inherit kernel
require recipes-kernel/linux/linux-yocto.inc

# Override SRC_URI in a copy of this recipe to point at a different source
# tree if you do not want to build from Linus' tree.
SRC_URI = "git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git;protocol=git;nocheckout=1

LINUX_VERSION ?= "4.2"
LINUX_VERSION_EXTENSION:append = "-custom"

# Modify SRCREV to a different commit hash in a copy of this recipe to
# build a different release of the Linux kernel.
# tag: v4.2 64291f7db5bd8150a74ad2036f1037e6a0428df2
SRCREV_machine="64291f7db5bd8150a74ad2036f1037e6a0428df2"

PV = "${LINUX_VERSION}+git${SRCPV}"

# Override COMPATIBLE_MACHINE to include your machine in a copy of this recipe
# file. Leaving it empty here ensures an early explicit build failure.
COMPATIBLE_MACHINE = "(^$)"
```



EXAMPLE: HELLO-MOD_0.1.BB

```
SUMMARY = "Example of how to build an external Linux kernel module"
DESCRIPTION = "${SUMMARY}"
LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

SRC_URI = "file://Makefile \
           file://hello.c \
           file://COPYING \
           "

S = "${WORKDIR}"

# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.

RPROVIDES:${PN} += "kernel-module-hello"
```

EXTENDING A BSP

- Many vendors provide Yocto Project based BSPs ready to use with their hardware platforms.
- In this case, instead of creating a BSP layer from scratch, we can simply extend the vendor's BSP.
- For this, it's recommended to create an additional layer to implement the customizations.
 - Use append files to customize the recipes.
 - Use patch files to apply changes software components (bootloader, kernel, etc).

CONFIGURING THE KERNEL

- A common need for BSP customization is changing the kernel configuration.
- The kernel configuration menu can be opened directly with BitBake executing the *menuconfig* task.

```
$ bitbake virtual/kernel -c menuconfig
```

- The configuration is done directly in the kernel build directory, so you can just recompile the kernel to test the changes.
- However, the configuration will not be persistent, and may be lost in an eventual cleanup of the kernel build directory.



CONFIGURING THE KERNEL

- For more control, customizations in the kernel configuration can be saved along with the metadata in a BSP layer.
- This can be done in two different ways:
 - Override the default configuration provided by the BSP.
 - Create configuration fragment files.

OVERRIDING THE CONFIGURATION

- To override the kernel configuration, we can create a custom configuration file called *defconfig*.
- This custom configuration file can be created manually or through BitBake:

```
$ bitbake virtual/kernel -c menuconfig  
$ bitbake virtual/kernel -c savedefconfig
```

- Then an append file for the kernel recipe can be created to add the *defconfig* file to the *SRC_URI* variable.



OVERRIDING THE CONFIGURATION

```
$ tree recipes-kernel/
recipes-kernel
└── linux
    ├── files
    │   └── defconfig
    └── linux-toradex_%.bbappend

$ cat recipes-kernel/linux/linux-toradex_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI += "file://defconfig"
```

CONFIGURATION FRAGMENTS

- Configuration fragments allow to define parts of the kernel configuration in files with the *.cfg* extension.
- Kernel configuration fragments can be created manually or via BitBake:

```
$ bitbake virtual/kernel -c kernel_configme
$ bitbake virtual/kernel -c menuconfig
$ bitbake virtual/kernel -c diffconfig
```

- An append file for the kernel recipe can be created to add the configuration fragments to the *SRC_URI* variable.



CONFIGURATION FRAGMENTS

```
$ tree recipes-kernel/
recipes-kernel
└── linux
    ├── files
    │   └── ntfs.cfg
    └── linux-yocto_%.bbappend

$ cat recipes-kernel/linux/files/ntfs.cfg
CONFIG_NTFS_FS=y

$ cat recipes-kernel/linux/linux-yocto_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI += "file://ntfs.cfg"
```

PATCHING THE KERNEL

- Keeping some kernel patches in the BSP layer might be necessary when we choose to not maintain a custom kernel repository.
- To apply kernel patches we can follow the same process that we have already studied in the training.
- Important: if the amount of kernel patches in the BSP layer starts to grow (5+), it's time to think about forking the kernel or even upstreaming the work!
 - This also applies to U-Boot patches.

PATCHING THE KERNEL

```
$ tree recipes-kernel/
recipes-kernel/
└── linux
    ├── files
    │   └── 0001-fixbug.patch
    └── linux-toradex_%.bbappend

$ cat recipes-kernel/linux/linux-toradex_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI += "file://0001-fixbug.patch"
```

ADVANCED METADATA

- Another way to configure the kernel is through a feature called Advanced Metadata.
- The idea is to organize patches and kernel configuration fragments into features, which can be enabled as needed.
- A kernel feature is described in a file with the .scc extension and can be enabled through the *SRC_URI* or *KERNEL_FEATURES* variables.
- Complete documentation about this feature is available in the Yocto Project's Kernel Development Manual.

<https://docs.yoctoproject.org/kernel-dev/advanced.html>



EXAMPLE: ADVANCED METADATA

```
$ tree meta-labworks/recipes-kernel/linux/
meta-labworks/recipes-kernel/linux/
    └── files
        ├── 0001-fix-tpm2-bug.patch
        ├── tpm2.cfg
        └── tpm2.scc
    └── linux-yocto_%.bbappend

$ cat meta-labworks/recipes-kernel/linux/files/tpm2.scc
define KFEATURE_DESCRIPTION "Enable TPM 2.0"
kconf hardware tpm2.cfg
patch 0001-fix-tpm2-bug.patch

$ cat meta-labworks/recipes-kernel/linux/files/tpm2.cfg
CONFIG_HW_RANDOM TPM=y
CONFIG_TCG TPM=y
CONFIG_TCG_TIS_CORE=y
CONFIG_TCG_TIS=y

$ cat meta-labworks/recipes-kernel/linux/linux-yocto_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI:append = " file://tpm2.scc"
```

LAB 7

DEVELOPING AND CUSTOMIZING BSPS



YOCTO PROJECT

SOFTWARE DEVELOPMENT KIT (SDK)



WORKING WITH BUILD SYSTEMS

- Although possible, using the build system for software development is usually not effective.
- The build system is an integration tool focused on helping with building and maintaining the software artifacts of the operating system.
 - It might not be very effective as an application development tool (write/compile/deploy/test workflow).
- For this reason, build systems usually provide developers with a set of development tools, which are generically called SDK.

SDK

- An SDK (Software Development Kit) provides an infrastructure of tools, libraries and utilities that enable the development of applications for a specific target platform.
- Contains a set of components, including:
 - Toolchain (compiler, linker, debugger, etc).
 - Sysroot (libraries, header files, debug symbols, etc).
 - Setup scripts and additional tools.

TOOLCHAIN

- There are basically four types of toolchains (*build* is the machine that generates the toolchain, *host* is the machine that executes the toolchain, *target* is the machine that executes the binary generated by the toolchain):
 - **Native toolchain:** build A, host A, target A.
 - **Cross-native toolchain:** build A, host B, target B.
 - **Cross-compiling toolchain:** build A, host A, target B.
 - **Canadian toolchain:** build A, host B, target C.
- Although *cross-native* is an option for embedded Linux development, it's more common to use a *cross-compiling* toolchain.

SYSROOT

- The sysroot contains several software artifacts needed to compile and link binaries for the target platform:
 - Header files (*.h*, *.hpp*, etc).
 - Static and dynamic libraries (*.so*, *.a*).
 - Binaries with debugging symbols (useful for debugging and resolving symbols).
- The sysroot content is usually based on the rootfs of an image generated for the target.



SCRIPTS AND OTHER TOOLS

- An SDK may contain configuration scripts and other additional tools.
- The configuration script helps to configure the development environment:
 - Set the *PATH* environment variable with the directory of the SDK tools.
 - Configure other useful environment variables for cross-compilation such as *CC*, *CFLAGS*, *LDFLAGS*, etc.
- Other tools might be available in the SDK.
 - Yocto Project's eSDK (Extensible SDK) provides a very interesting tool called *devtool*.

GENERATING SDKS

- The Yocto Project is able to generate SDKs in a few different ways:
 - *meta-toolchain*: Creates a generic SDK with a basic sysroot.
 - *populate_sdk*: Creates a complete SDK with a sysroot based on an image.
 - *populate_sdk_ext*: Creates a complete SDK with a sysroot based on an image, plus additional tools to manipulate the image and the metadata.
- The generated SDK will be a self-contained installation script that can be executed on development machines.



META-TOOLCHAIN

- A generic SDK with a basic sysroot can be generated by processing the *meta-toolchain* recipe.

```
$ bitbake meta-toolchain
```

- At the end of the process, an installation script will be available in *tmp/deploy/sdk*.

```
$ ls -1 tmp/deploy/sdk/
poky-glibc-x86_64-meta-toolchain-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.host.manifest
poky-glibc-x86_64-meta-toolchain-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.sh
poky-glibc-x86_64-meta-toolchain-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.target.manifest
poky-glibc-x86_64-meta-toolchain-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.testdata
```

- This SDK can be used for bare-metal development (bootloader, kernel) or to compile simple applications.

META-TOOLCHAIN: INSTALLING

```
$ tmp/deploy/sdk/poky-glibc-x86_64-meta-toolchain-cortexa9t2hf-neon-colibri-imx6-training-toolchain--  
Poky (Yocto Project Reference Distro) SDK installer version 4.0.1  
=====  
Enter target directory for SDK (default: /opt/poky/4.0.1):  
You are about to install the SDK to "/opt/poky/4.0.1". Proceed [Y/n]? y  
[sudo] password for sprado:  
Extracting SDK.....done  
Setting it up...done  
SDK has been successfully set up and is ready to be used.  
Each time you wish to use the SDK in a new shell session, you need to source the environment setup s  
$ . /opt/poky/4.0.1/environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi  
  
$ ls -l /opt/poky/4.0.1/  
total 36  
-rw-r--r-- 1 root root 4084 jul 13 11:21 environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi  
-rw-r--r-- 1 root root 22387 jul 13 11:21 site-config-cortexa9t2hf-neon-poky-linux-gnueabi  
drwxr-xr-x 4 root root 4096 jul 13 11:20 sysroots  
-rw-r--r-- 1 root root 90 jul 13 11:21 version-cortexa9t2hf-neon-poky-linux-gnueabi
```

META-TOOLCHAIN: USING

```
$ source /opt/poky/4.0.1/environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi

$ echo $CC
arm-poky-linux-gnueabi-gcc -mthumb -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a9 -fstack-protector-strong -fno-strict-aliasing -fno-PIE -fno-�

$ $CC --version
arm-poky-linux-gnueabi-gcc (GCC) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ cat main.c
int main(void){return 0;}

$ $CC main.c -o main

$ file main
main: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /l
```



POPULATE_SDK

- A complete SDK with a sysroot based on an image can be generated by processing the *populate_sdk* task:

```
$ bitbake core-image-minimal -c populate_sdk
```

- At the end of the process, an installation script will be available in *tmp/deploy/sdk*.

```
$ ls tmp/deploy/sdk/
poky-glibc-x86_64-core-image-minimal-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.host.
poky-glibc-x86_64-core-image-minimal-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.sh
poky-glibc-x86_64-core-image-minimal-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.target
poky-glibc-x86_64-core-image-minimal-cortexa9t2hf-neon-colibri-imx6-training-toolchain-4.0.1.test
```

- This SDK can be used for the development of bare-metal code and Linux applications, making it possible to link with any library installed in the system image.

POPULATE_SDK: INSTALLING

```
$ tmp/deploy/sdk/poky-glibc-x86_64-core-image-minimal-cortexa9t2hf-neon-colibri-imx6-training-toolch
Poky (Yocto Project Reference Distro) SDK installer version 4.0.1
=====
Enter target directory for SDK (default: /opt/poky/4.0.1): y
You are about to install the SDK to "/opt/labs/ex/build/y". Proceed [Y/n]? y
Extracting SDK.....done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup s
$ . /opt/labs/ex/build/y/environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi

$ ls -l /opt/poky/4.0.1/
total 36
-rw-r--r-- 1 root root 4084 jul 13 11:21 environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi
-rw-r--r-- 1 root root 22387 jul 13 11:21 site-config-cortexa9t2hf-neon-poky-linux-gnueabi
drwxr-xr-x 4 root root 4096 jul 13 11:20 sysroots
-rw-r--r-- 1 root root 90 jul 13 11:21 version-cortexa9t2hf-neon-poky-linux-gnueabi
```

POPULATE_SDK: USING

```
$ source /opt/poky/4.0.1/environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi

$ echo $CC
arm-poky-linux-gnueabi-gcc -mthumb -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a9 -fstack-protector-strong -fno-strict-aliasing -fno-PIE -fno-�

$ $CC --version
arm-poky-linux-gnueabi-gcc (GCC) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ cat main.c
int main(void){return 0;}

$ $CC main.c -o main

$ file main
main: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /l
```



POPULATE_SDK: AUTOTOOLS

```
$ ls
configure.ac  hello.c  Makefile.am

$ cat hello.c
#include <stdio.h>
int main(void) { printf("Hello World!\n"); }

$ cat configure.ac
AC_INIT(hello,0.1)
AM_INIT_AUTOMAKE([foreign])
AC_PROG_CC
AC_CONFIG_FILES(Makefile)
AC_OUTPUT

$ cat Makefile.am
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```



POPULATE_SDK: AUTOTOOLS

```
$ source /opt/poky/4.0.1/environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi
$ autoreconf
$ ls
aclocal.m4 autom4te.cache compile configure configure.ac depcomp
hello.c install-sh Makefile.am Makefile.in missing

$ ./configure ${CONFIGURE_FLAGS}

$ echo ${CONFIGURE_FLAGS}
--target=arm-poky-linux-gnueabi --host=arm-poky-linux-gnueabi --build=x86_64-linux
--with-libtool-sysroot=/opt/poky/4.0.1/sysroots/cortexa9t2hf-neon-poky-linux-gnueabi

$ make

$ make install DESTDIR=install/
$ tree install/
install/
└── usr
    └── local
        └── bin
            └── hello
```

POPULATE_SDK_EXT

- A complete SDK with a sysroot based on an image and additional tools can be generated by processing the *populate_sdk_ext* task:

```
$ bitbake core-image-minimal -c populate_sdk_ext
```

- At the end of the process, an installation script will be available in *tmp/deploy/sdk*.

```
$ ls -1 tmp/deploy/sdk/  
poky-glibc-x86_64-core-image-training-cortexa9t2hf-neon-colibri-imx6-training-toolchain-ext-4.0.1.host  
poky-glibc-x86_64-core-image-training-cortexa9t2hf-neon-colibri-imx6-training-toolchain-ext-4.0.1.sh  
poky-glibc-x86_64-core-image-training-cortexa9t2hf-neon-colibri-imx6-training-toolchain-ext-4.0.1.target  
poky-glibc-x86_64-core-image-training-cortexa9t2hf-neon-colibri-imx6-training-toolchain-ext-4.0.1.test
```

- This SDK is called Extensible SDK (eSDK) and extends the functionality of a normal SDK, allowing to customize and manipulate the operating system image through the *devtool* tool.



POPULATE_SDK_EXT: INSTALLING

```
$ tmp/deploy/sdk/poky-glibc-x86_64-core-image-training-cortexa9t2hf-neon-colibri-imx6-training-toolc
Poky (Yocto Project Reference Distro) Extensible SDK installer version 4.0.1
=====
Enter target directory for SDK (default: ~/poky_sdk):
You are about to install the SDK to "/home/sprado/poky_sdk". Proceed [Y/n]?
Extracting SDK.....done
Setting it up...
Extracting buildtools...
Preparing build system...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup s
$ . /home/sprado/poky_sdk/environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi

$ ls /home/sprado/poky_sdk/
bitbake-cookerdaemon.log
buildtools
cache
conf
downloads
environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi
layers
                                         preparing_build_system.log
                                         site-config-cortexa9t2hf-neon-poky-linux-gnu
                                         sstate-cache
                                         sysroots
                                         tmp
                                         version-cortexa9t2hf-neon-poky-linux-gnueabi
                                         workspace
```

POPULATE_SDK_EXT: USING

```
$ source ~/poky_sdk/environment-setup-cortexa9t2hf-neon-poky-linux-gnueabi
SDK environment now set up; additionally you may now run devtool to perform development tasks.
Run devtool --help for further details.

$ echo $CC
arm-poky-linux-gnueabi-gcc -mthumb -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a9 -fstack-protector-strong

$ $CC --version
arm-poky-linux-gnueabi-gcc (GCC) 11.3.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ cat main.c
int main(void){return 0;}

$ $CC main.c -o main

$ file main
main: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-armhf.so.3, buildtime RPATH '/usr/lib', not stripped
```

DEVTOOL

- The big difference between the SDK generated with the *populate_sdk* task and the SDK generated with the *populate_sdk_ext* task is the support for the *devtool* tool.
- This tool provides a set of functionality to help develop and integrate software into an image generated with OpenEmbedded/Yocto Project.
- It has a Git-like design, with sub-commands for each of the supported features.
- *devtool* is not only limited to the SDK and can be used in the Yocto Project build environment.

INTERESTING COMMANDS

- *devtool add*: creates a recipe from an application's source code.
- *devtool edit-recipe*: opens the recipe code for editing.
- *devtool build*: processes a recipe.
- *devtool deploy-target*: installs generated packages on the target (via SSH).
- *devtool finish*: integrates the generated metadata (recipes, appends, etc) into a layer.
- *devtool upgrade*: upgrades a recipe/application version.
- *devtool modify*: modify an existing recipe/application.



DEVTOOL ADD

```
$ devtool add --version 4.4 fping http://www.fping.org/dist/fping-4.4.tar.gz
INFO: Fetching http://www.fping.org/dist/fping-4.4.tar.gz...
...
INFO: Recipe /home/sprado/poky_sdk/workspace/recipes/fping/fping_4.4.bb has been automatically created

$ tree -L 3 workspace/
workspace/
├── appends
│   └── fping_4.4.bbappend
├── conf
│   └── layer.conf
├── README
└── recipes
    └── fping
        └── fping_4.4.bb
└── sources
    └── fping
        ├── aclocal.m4
        ├── CHANGELOG.md
        ├── ci
        ├── compile
        ├── ...
        └── src
```



GENERATED REVENUE: FPING_4.4.BB

```
# Recipe created by recipetool
# This is the basis of a recipe and may need further editing in order to be fully functional.
# (Feel free to remove these comments when editing.)

# WARNING: the following LICENSE and LIC_FILES_CHKSUM values are best guesses - it is
# your responsibility to verify that the values are complete and correct.
#
# The following license files were not able to be identified and are
# represented as "Unknown" below, you will need to check them yourself:
#   COPYING
LICENSE = "Unknown"
LIC_FILES_CHKSUM = "file://COPYING;md5=c6170fbaddfc74f011515291d96901"

SRC_URI = "http://www.fping.org/dist/fping-${PV}.tar.gz"
SRC_URI[md5sum] = "10e50c164ddff6941eff5fef41c21e3d"
SRC_URI[sha1sum] = "8be8be5b8dc5c0dd3505db79b8fe1438fe4ef590"
SRC_URI[sha256sum] = "9f854b65a52dc7b1749d6743e35d0a6268179d1a724267339fc9a066b2b72d11"
SRC_URI[sha384sum] = "1647b4cf07d751961fbe37bde12292dbc3820e396fe37271d088168a605588a7b4def9f14ac697"
SRC_URI[sha512sum] = "8c9ff78edee10ce8e02a0d6189d4a2d91cc60954740c15730f8f1a17d037ee9f565828fa0dc47

# NOTE: if this software is not capable of being built in a separate build directory
# from the source, you should replace autotools with autotools-brokensep in the
# inherit line
inherit autotools
```



DEVTOOL EDIT-RECIPE

```
$ devtool edit-recipe fping
```

```
diff --git a/fping_4.4.bb b/fping_4.4.bb
index 175c8cd0187e..8f8542dad2c9 100644
--- a/fping_4.4.bb
+++ b/fping_4.4.bb
@@ -8,15 +8,14 @@
 # The following license files were not able to be identified and are
 # represented as "Unknown" below, you will need to check them yourself:
 # COPYING
-LICENSE = "Unknown"
+SUMMARY = "fping sends ICMP ECHO_REQUEST packets to network hosts"
+HOMEPAGE = "http://www.fping.org/"
+
+LICENSE = "BSD"
 LIC_FILES_CHKSUM = "file://COPYING;md5=c6170fbaddfc74f011515291d96901"

 SRC_URI = "http://www.fping.org/dist/fping-${PV}.tar.gz"
-SRC_URI[md5sum] = "10e50c164ddff6941eff5fef41c21e3d"
-SRC_URI[sha1sum] = "8be8be5b8dc5c0dd3505db79b8fe1438fe4ef590"
 SRC_URI[sha256sum] = "9f854b65a52dc7b1749d6743e35d0a6268179d1a724267339fc9a066b2b72d11"
-SRC_URI[sha384sum] = "1647b4cf07d751961fbe37bde12292dbc3820e396fe37271d088168a605588a7b4def9f14ac69
-SRC_URI[sha512sum] = "8c9ff78edee10ce8e02a0d6189d4a2d91cc60954740c15730f8f1a17d037ee9f565828fa0dc4

 # NOTE: if this software is not capable of being built in a separate build directory
 # from the source, you should replace autotools with autotools-brokense in the
```



DEVTOOL BUILD/DEPLOY-TARGET

```
$ devtool build fping
...
NOTE: fping: compiling from external source tree /home/sprado/poky_sdk/workspace/sources/fping
NOTE: Tasks Summary: Attempted 621 tasks of which 595 didn't need to be rerun and all succeeded.

$ devtool deploy-target fping root@192.168.7.2
...
INFO: Successfully deployed /home/sprado/poky_sdk/tmp/work/core2-64-poky-linux/fping/4.4-r0/image

# fping --version
fping: Version 4.4
fping: comments to david@schweikert.ch
```

DEVTOOL FINISH

```
$ devtool finish --force --remove-work fping meta-labworks
...
INFO: No patches or files need updating
INFO: Moving recipe file to /home/sprado/poky_sdk/layers/meta-labworks/recipes-fping/fping
INFO: -r argument used on fping, removing source tree. You will lose any unsaved work

$ tree layers/meta-labworks/recipes-fping/
layers/meta-labworks/recipes-fping/
└── fping
    └── fping_4.4.bb
```

DEVTOOL UPGRADE

```
$ devtool upgrade --version 5.0 fping
...
INFO: Rebasing devtool onto d6ab87c90bc0b5b542db1c302edf2f15cc6df490
INFO: Upgraded source extracted to /home/sprado/poky_sdk/workspace/sources/fping
INFO: New recipe is /home/sprado/poky_sdk/workspace/recipes/fping/fping_5.0.bb

$ devtool build fping

$ devtool deploy-target fping root@192.168.7.2

# fping --version
fping: Version 5.0

$ devtool finish --force --remove-work fping meta-labworks
...
INFO: Removing file /home/sprado/poky_sdk/layers/meta-labworks/recipes-fping/fping/fping_4.4.bb
INFO: Moving recipe file to /home/sprado/poky_sdk/layers/meta-labworks/recipes-fping/fping

$ ls layers/meta-labworks/recipes-fping/fping/
fping_5.0.bb
```

DEVTOOL MODIFY

```
$ devtool modify fping
INFO: Source tree extracted to /home/sprado/poky_sdk/workspace/sources/fping
INFO: Recipe fping now set up to build from /home/sprado/poky_sdk/workspace/sources/fping

$ cd workspace/sources/fping
$ vim src/fping.c && git commit -s -m "change version message"

$ devtool build fping
$ devtool deploy-target fping root@192.168.7.2

# fping --version
fping: Version 5.0 (test)

$ devtool finish --force --remove-work fping meta-labworks
INFO: Adding new patch 0001-change-version-message.patch
INFO: Updating recipe fping_5.0.bb

$ tree layers/meta-labworks/recipes-fping/
layers/meta-labworks/recipes-fping/
└── fping
    ├── fping
    │   └── 0001-change-version-message.patch
    └── fping_5.0.bb
```

DEVTOOL --HELP

```
$ devtool --help
usage: devtool [--basepath BASEPATH] [--bbpath BBPATH] [-d] [-q] [--color COLOR] [-h] <subcommand> .

OpenEmbedded development tool

options:
  --basepath BASEPATH      Base directory of SDK / build directory
  --bbpath BBPATH          Explicitly specify the BBPATH, rather than getting it from the metadata
  -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:
  Beginning work on a recipe:
    add                      Add a new recipe
    modify                   Modify the source for an existing recipe
    upgrade                  Upgrade an existing recipe
  Getting information:
    status                   Show workspace status
    search                   Search available recipes
    latest-version           Report the latest version of an existing recipe
    check-upgrade-status     Report upgradability for multiple (or all) recipes
  Working on a recipe in the workspace:
```



SDK VS eSDK

Feature	SDK (populate_sdk)	eSDK (populate_sdk_ext)
Toolchain	Yes	Yes
Debugger	Yes	Yes
Size	100+ MBytes	1+ GBytes
devtool	No	Yes
Compile Images	No	Yes
Upgradable	No	Yes
Manageable sysroot	No	Yes
Construction	Packages	Shared State

LAB 8

GENERATING AND USING SDKS



YOCTO PROJECT

ADDITIONAL TOOLS AND RESOURCES

INTRODUCTION

- There are several projects and resources provided by the Yocto Project that can benefit users during the development and maintenance of the distribution.
<https://www.yoctoproject.org/software-item/>
- The list of projects and additional resources includes:
 - License management.
 - Build caches (shared state cache).
 - Build history.
 - Toaster.
 - Build tools (build appliance, CROPS, autobuilder).

LICENSE MANAGEMENT

- One of the concerns when developing products that uses free software is license compliance.
- OpenEmbedded/Yocto Project solves this problem by tracking the license of each software component through the *LICENSE* and *LIC_FILES_CHKSUM* variables.
- The *LICENSE* variable accepts the following values:
 - The name of the files in *files/common-licenses/* from OpenEmbedded-Core (each file in this directory describes a license in either SPDX or OSI formats).
 - *SPDXLICENSEMAP* varflags defined in *conf/licenses.conf* from OpenEmbedded-Core.
- A warning message is displayed if the license declared in the *LICENSE* variable is not found.



COMMON-LICENSES

```
$ ls poky/meta/files/common-licenses/
0BSD                                DRL-1.0          NL0D-2.0
AAL                                 DSDP             NLPL
Abstyles                            DSSL             Nokia
Adobe                               dvipdfm          NOSL
Adobe-2006                           ECL-1.0         Noweb
Adobe-Glyph                          ECL-2.0         NPL-1.0
ADSL                                eCos-2.0        NPL-1.1
AFL-1.1                             EDL-1.0         NPOSL-3.0
AFL-1.2                             EFL-1.0         NRL
AFL-2.0                             EFL-2.0         NTP
AFL-2.1                             eGenix          NTP-0
AFL-3.0                             Entessa          OASIS
Afmparse                            EPICS            OCCT-PL
AGPL-1.0-only                       EPL-1.0         OCLC-2.0
AGPL-1.0-or-later                   EPL-2.0         ODbL-1.0
AGPL-3.0-only                       ErlPL-1.1       ODC-By-1.0
AGPL-3.0-or-later                   etalab-2.0      OFL-1.0
Aladdin                            EUDatagrid      OFL-1.0-no-RFN
AMDPLPA                            EUPL-1.0        OFL-1.0-RFN
AML                                EUPL-1.1        OFL-1.1
```

LICENSES.CONF

```
$ cat meta/conf/licenses.conf
# Standards are great! Everyone has their own. In an effort to standardize licensing
# names, common-licenses will use the SPDX standard license names. In order to not
# break the non-standardized license names that we find in LICENSE, we'll set
# up a bunch of VarFlags to accommodate non-SPDX license names.
#
# We should really discuss standardizing this field, but that's a longer term goal.
# For now, we can do this and it should grab the most common LICENSE naming variations.
#
# We should NEVER have a GPL/LGPL without a version!!!!
# Any mapping to MPL/LGPL/GPL should be fixed

# AGPL variations
SPDXLICENSEMAP[AGPL-3] = "AGPL-3.0-only"
SPDXLICENSEMAP[AGPL-3+] = "AGPL-3.0-or-later"
SPDXLICENSEMAP[AGPLv3] = "AGPL-3.0-only"
SPDXLICENSEMAP[AGPLv3+] = "AGPL-3.0-or-later"
SPDXLICENSEMAP[AGPLv3.0] = "AGPL-3.0-only"
SPDXLICENSEMAP[AGPLv3.0+] = "AGPL-3.0-or-later"
SPDXLICENSEMAP[AGPL-3.0] = "AGPL-3.0-only"
SPDXLICENSEMAP[AGPL-3.0+] = "AGPL-3.0-or-later"

# BSD variations
SPDXLICENSEMAP[BSD-0-Clause] = "0BSD"
```



DISABLING A LICENSE

- During the development of a product, it may be necessary to avoid using software under a certain license.
- For example, the GPLv3 license has a clause to prevent what is commonly called Tivoization (hardware restrictions that prevent the user from updating the software on the device).
 - This license requirement may be prohibitive for some commercial products.
- For these cases, the *INCOMPATIBLE_LICENSE* variable can be used to prevent the usage of software with certain licenses.

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



PROPRIETARY AND COMMERCIAL SOFTWARE

- Recipes can use the *CLOSED* value to define that a software is proprietary.

```
LICENSE = "CLOSED"
```

- Commercial or special software licenses can be specified in the recipe using the *LICENSE_FLAGS* variable.

```
LICENSE_FLAGS = "commercial"
```

- For BitBake to process recipes with such special licenses, it's necessary to set the *LICENSE_FLAGS_ACCEPTED* variable.

```
LICENSE_FLAGS_ACCEPTED = "commercial"
```



LICENSE COMPLIANCE

- One of the concerns in a project that uses free software is compliance with software licenses.
- While each license has its own "requirements", there are typically three things a developer should be concerned about:
 - Availability of the original source code and any changes made to it.
 - Releasing a copy of the license text.
 - Availability of build scripts.
- The Yocto Project is able to help with these requirements to ensure compliance with software licenses.

DISTRIBUTING THE SOURCE CODE

- The *archiver* class and the *ARCHIVER_MODE* variable can be used to generate a directory with the source code of the software components used in the distribution:

```
INHERIT += "archiver"
ARCHIVER_MODE[src] = "original"
```

- At the end of the image generation, the sources in tarball format and their respective patches will be available in *tmp/deploy/sources*.
- If necessary, the *COPYLEFT_LICENSE_EXCLUDE* variable can be used to exclude software components of a certain license.



DISTRIBUTING THE SOURCE CODE

```
$ ls tmp/deploy/sources/x86_64-poky-linux/busybox-1.34.1-r0/
0001-du-l-works-fix-to-use-145-instead-of-144.patch
0001-mktemp-add-tmpdir-option.patch
0001-sysctl-ignore-EIO-of-stable_secret-below-proc-sys-ne.patch
0001-testsuite-check-uudecode-before-using-it.patch
0001-testsuite-use-www.example.org-for-wget-test-cases.patch
0001-Use-CC-when-linking-instead-of-LD-and-use-CFLAGS-and.patch
busybox-1.34.1.tar.bz2
busybox-cron
busybox-cross-menuconfig.patch
busybox-httdp
busybox-klogd.service.in
busybox-syslog.default
busybox-syslog.service.in
busybox-udhcpc-no_deconfig.patch
busybox-udhcpd
default.script
defconfig
fail_on_no_media.patch
find-touchscreen.sh
getopts.cfg
hwclock.sh
inetd
inetd.conf
login-utilities.cfg
longopts.cfg
makefile-libbb-race.patch
mdev
mdev.conf
mdev-mount.sh
mount-via-label.cfg
pgrep.cfg
rck
rcS
recognize_command.patch
resize.cfg
rev.cfg
run-ptest
series
sha1sum.cfg
sha256sum.cfg
simple.script
syslog
syslog.cfg
syslog.conf
syslog-startup.conf
unicode.cfg
```

DISTRIBUTING THE LICENSE TEXT

- By default, licenses for all software components processed by BitBake are kept in the build directory at *tmp/deploy/licenses*.
- However, some licenses require the license text to be distributed in the image along with the generated artifacts (library, executable, etc).
- For this, the following variables can be used in a configuration file:

```
COPY_LIC_MANIFEST = "1"  
COPY_LIC_DIRS = "1"  
LICENSE_CREATE_PACKAGE = "1"
```

- The licenses will be added to the image in the */usr/share/common-licenses* directory.

DISTRIBUTING THE LICENSE TEXT

```
# cat /usr/share/common-licenses/license.manifest
PACKAGE NAME: base-files
PACKAGE VERSION: 3.0.14
RECIPE NAME: base-files
LICENSE: GPLv2

PACKAGE NAME: base-passwd
PACKAGE VERSION: 3.5.29
RECIPE NAME: base-passwd
LICENSE: GPLv2

PACKAGE NAME: busybox
PACKAGE VERSION: 1.34.1
RECIPE NAME: busybox
LICENSE: GPLv2 & bzip2-1.0.4
...
.

# ls /usr/share/common-licenses/busybox/
LICENSE.0           LICENSE.1           generic_GPLv2       generic_bzip2-1.0.4   recipeinfo
```



DISTRIBUTING THE BUILD SCRIPTS

- To provide the build scripts, just give the user the necessary commands to download the correct versions of the layers and configure the build directory:

```
$ git clone -b kirkstone git://git.yoctoproject.org/poky  
  
$ git clone -b release_branch git://git.mycompany.com/meta-my-bsp-layer  
$ git clone -b release_branch git://git.mycompany.com/meta-my-software-layer  
  
$ find . -name ".git" -type d -exec rm -rf {} \\;  
  
$ source poky/oe-init-build-env build
```



SHARED STATE CACHE

- The Yocto Project implements a build cache mechanism called shared state cache.
<https://docs.yoctoproject.org/singleindex.html#shared-state-cache>
- This feature provides a very efficient and incremental build mechanism.
- Build caches are stored by default in the build directory at `sstate-cache/`.
 - This value can be changed via the `SSTATE_DIR` variable.
- To share the sstate cache directory between multiple developers or machines, the NFS protocol is recommended.



BUILD HISTORY

- During the development of a Linux distribution with the Yocto Project, any change in the metadata can impact the generated image, causing possible regressions.
- The build history is a tool that makes it possible to analyze and track changes in the build output, identifying possible unwanted changes.
- It's implemented by the *buildhistory* class and makes it possible to identify:
 - Inclusion or removal of files and packages.
 - Changes in files and image size.
 - Changes in package versions.

ENABLING BUILD HISTORY

- Build history can be enabled by inheriting the *buildhistory* class in a configuration file.

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

- The *BUILDHISTORY_COMMIT* variable enables commits in a Git repository to make it easier to track the changes.
- Keep in mind that the build history may slightly increase the build time and disk usage.

BUILD HISTORY DIRECTORY

- The build history is stored by default in *buildhistory/*.
 - Information about generated images is stored *buildhistory/images/*.
 - Information about installed packages is stored in *buildhistory/packages/*.
- The build history directory can be changed via the *BUILDHISTORY_DIR* variable.

```
BUILDHISTORY_DIR ?= "${TOPDIR}/buildhistory"
```

BUILDHISTORY DIRECTORY

```
$ tree buildhistory/
buildhistory/
├── images
│   └── qemux86_64
│       └── glibc
│           └── core-image-minimal
│               ├── build-id.txt
│               ├── depends.dot
│               ├── depends-nokernel.dot
│               ├── depends-nokernel-nolibc.dot
│               ├── depends-nokernel-nolibc-noupdate.dot
│               ├── depends-nokernel-nolibc-noupdate-nomodules.dot
│               ├── files-in-image.txt
│               ├── image-files
│               │   └── etc
│               │       └── group
│               │           └── passwd
│               ├── image-info.txt
│               ├── installed-package-info.txt
│               ├── installed-package-names.txt
│               ├── installed-package-sizes.txt
│               └── installed-packages.txt
└── metadata-revs
    └── packages
```

ANALYZING THE BUILD HISTORY

- The build history can be analyzed directly with Git.

```
$ git log  
$ git show HEAD
```

- Alternatively, the *buildhistory-diff* tool can be used, which has a simpler and easy to understand output:

```
$ buildhistory-diff -a HEAD^
```

- The build history can also be accessed through a web interface:

<https://git.yoctoproject.org/buildhistory-web/about/>



BUILD HISTORY WITH GIT

```
$ git log --oneline
9e0bb5bf49e5 (HEAD -> master) Build 20211222192916 of poky 3.4 for machine qemux86-64 on sprado-offi...
ca3d6aa3424a (tag: build-minus-1) Build 20211222185937 of poky 3.4 for machine qemux86-64 on sprado-...
9117fee4f44a (tag: build-minus-2) Build 20211222185640 of poky 3.4 for machine qemux86-64 on sprado-...

$ git show HEAD
...
diff --git a/images/qemux86_64/glibc/core-image-minimal/files-in-image.txt b/images/qemux86_64/glibc/
index fa46c3592795..ac4ca6548976 100644
--- a/images/qemux86_64/glibc/core-image-minimal/files-in-image.txt
+++ b/images/qemux86_64/glibc/core-image-minimal/files-in-image.txt
@@ -826,6 +826,7 @@ lrwxrwxrwx root      root          19 ./usr/sbin/chroot -> /bin/busybox.no
    lrwxrwxrwx root      root          19 ./usr/sbin/delgroup -> /bin/busybox.nosuid
    lrwxrwxrwx root      root          19 ./usr/sbin/deluser -> /bin/busybox.nosuid
    lrwxrwxrwx root      root          19 ./usr/sbin/fbset -> /bin/busybox.nosuid
+-rwxr-xr-x root      root        48112 ./usr/sbin/fping
 -rwxr-xr-x root      root       92800 ./usr/sbin/groupadd
 -rwxr-xr-x root      root       88512 ./usr/sbin/groupdel
 -rwxr-xr-x root      root      59400 ./usr/sbin/groupmems
diff --git a/images/qemux86_64/glibc/core-image-minimal/image-info.txt b/images/qemux86_64/glibc/cor...
index bea8e7d265cd..8c3d9004d80e 100644
--- a/images/qemux86_64/glibc/core-image-minimal/image-info.txt
+++ b/images/qemux86_64/glibc/core-image-minimal/image-info.txt
@@ -4,10 +4,10 @@ USER_CLASSES = buildstats
```

BUILDHISTORY-DIFF

```
$ buildhistory-diff -a HEAD^
Changes to images/qemux86_64/glibc/core-image-minimal (files-in-image.txt):
    /usr/sbin/fping was added
images/qemux86_64/glibc/core-image-minimal: IMAGE_INSTALL: added "fping"
images/qemux86_64/glibc/core-image-minimal: IMAGESIZE changed from 38844 to 38892 (+0%)
Changes to images/qemux86_64/glibc/core-image-minimal (installed-package-names.txt):
    fping was added
```

RECIPETOOL

- The *recipetool* tool automates Yocto Project's recipe management.
- It has features similar to the *devtool* tool, but is more focused on creating and changing recipes.
- Some interesting features available:
 - Create a new recipe.
 - Create an append file.
 - Edit existing recipes.
 - Set variables in recipes.



RECIPETOOL

```
$ recipetool -h
usage: recipetool [-d] [-q] [--color COLOR] [-h] <subcommand> ...

OpenEmbedded recipe tool

options:
  -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:
  create          Create a new recipe
  appendfile      Create/update a bbappend to replace a target file
  appendsrcfiles  Create/update a bbappend to add or replace source files
  appendsrcfile   Create/update a bbappend to add or replace a source file
  edit            Edit the recipe and appends for the specified target. This obeys $VISUAL if set, o
  setvar          Set a variable within a recipe
  newappend       Create a bbappend for the specified target in the specified layer
Use recipetool <subcommand> --help to get help on a specific command
```

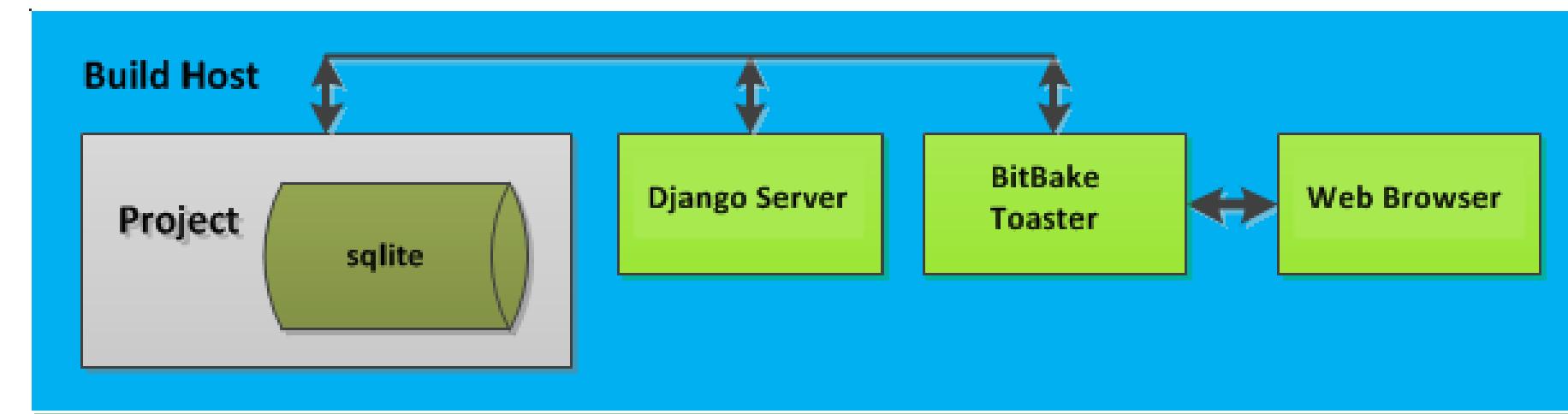


TOASTER

- Toaster is a web interface to the Yocto Project's OpenEmbedded Build System.
<https://docs.yoctoproject.org/toaster-manual/intro.html>
- With a simple and easy-to-use interface, Toaster provides two main functionalities:
 - Collect and display information about the build process.
 - Configure and build images.
- It is developed in Python (Django) and contains 3 components:
 - Web-based frontend.
 - Backend that communicates with the BitBake server.
 - SQLite database to store the build information.



TOASTER: ARCHITECTURE



RUNNING TOASTER

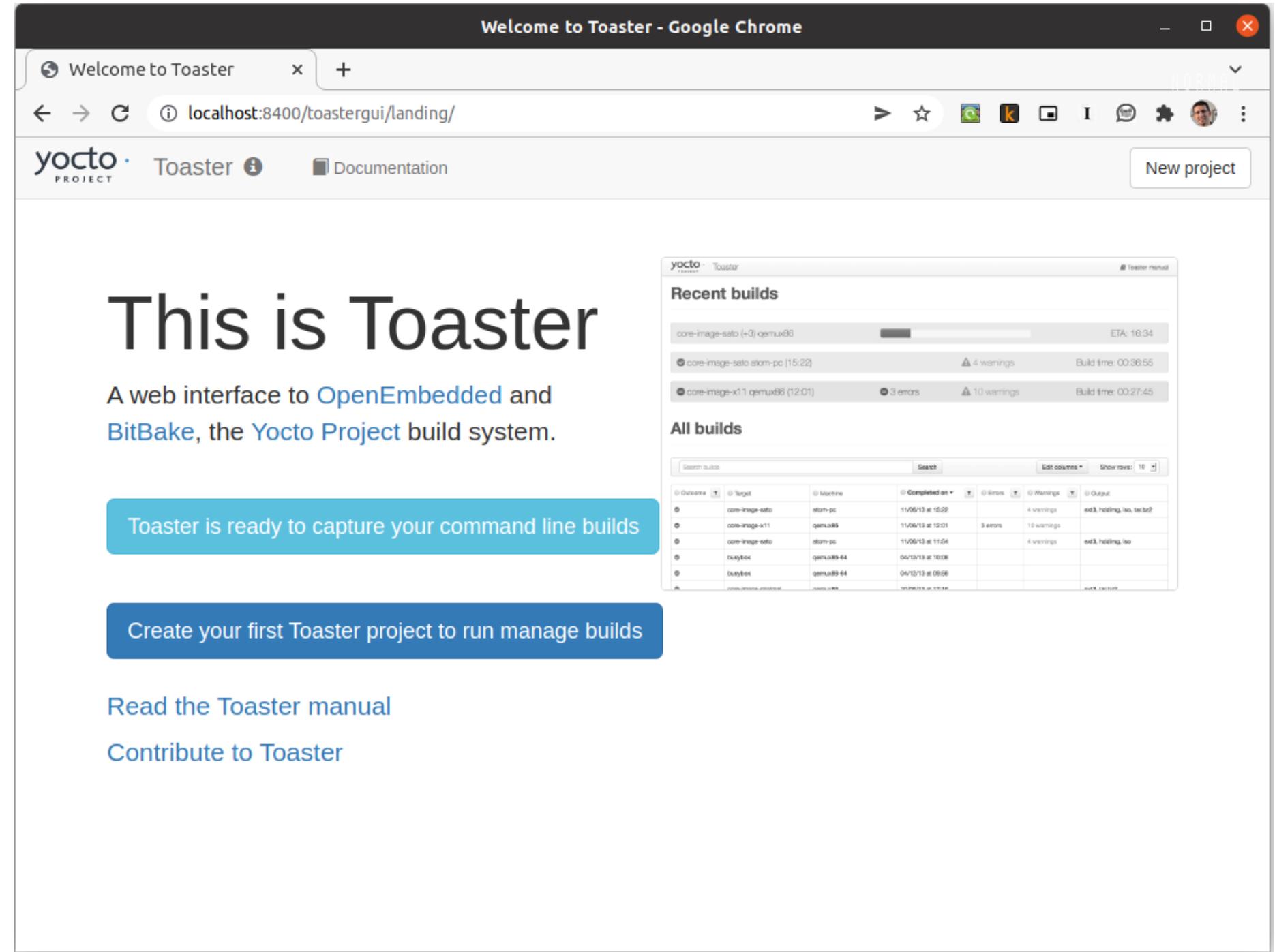
- Toaster can be run on a local instance in the developer's machine or a remote server shared with multiple developers.
- Instructions on how to prepare the machine are available in the documentation:
<https://docs.yoctoproject.org/toaster-manual/start.html>
- Toaster can be started with the command below.

```
$ source toaster start
```

- Its web interface is available by default at <http://127.0.0.1:8000>.



TOASTER: HOMEPAGE



The screenshot shows a web browser window titled "Welcome to Toaster - Google Chrome". The address bar displays "localhost:8400/toastergui/landing/". The main content area is titled "yocto PROJECT Toaster" and features a "New project" button. Below this, a large heading "This is Toaster" is displayed, followed by the text "A web interface to OpenEmbedded and BitBake, the Yocto Project build system." A blue call-to-action button says "Toaster is ready to capture your command line builds". Another blue button below it says "Create your first Toaster project to run manage builds". At the bottom, there are links to "Read the Toaster manual" and "Contribute to Toaster". On the right side of the screen, a sidebar titled "yocto - Toaster" shows "Recent builds" and "All builds". The "Recent builds" section lists three entries: "core-image-salo (=3) gemux86" (ETA: 16:34), "core-image-salo atom-pc [15:22]" (4 warnings, Build time: 00:38:55), and "core-image-x11 gemux86 (12:01)" (3 errors, 10 warnings, Build time: 00:27:45). The "All builds" section is a table with columns: Outcome, Target, Machine, Completed on, Errors, Warnings, and Output. It lists several completed builds, such as "core-image-salo atom-pc 11/08/13 at 15:29", "core-image-x11 gemux86 11/08/13 at 12:01", "core-image-salo atom-pc 11/08/13 at 11:54", "busybox gemux86-64 05/12/13 at 10:08", and "busybox gemux86-64 04/10/13 at 09:58".

TOASTER: CONFIGURING A BUILD

Configuration - Teste - Toaster - Google Chrome

localhost:8400/toastergui/project/2/

yocto PROJECT Toaster All builds All projects Documentation New project

Test Project

Configuration Builds (0) Import layer New custom image Type the recipe you want to build Build

Configuration

- COMPATIBLE METADATA
- Custom images
- Image recipes
- Software recipes
- Machines
- Layers
- Distros

EXTRA CONFIGURATION

- BitBake variables

ACTIONS

- Delete project

Machine
qemux86-64

Distro
poky

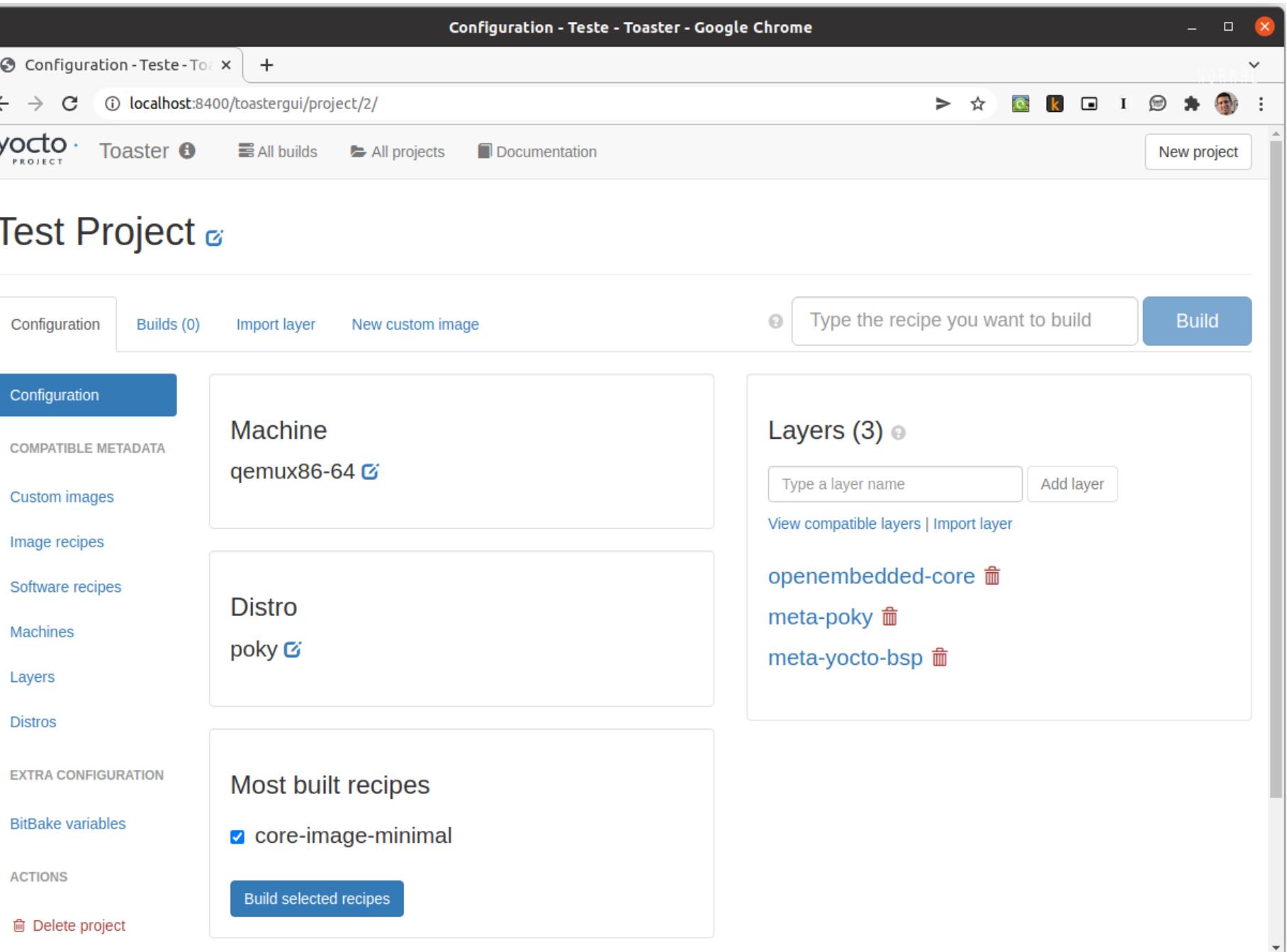
Layers (3)

- Type a layer name
- Add layer
- [View compatible layers](#) | [Import layer](#)
- openembedded-core
- meta-poky
- meta-yocto-bsp

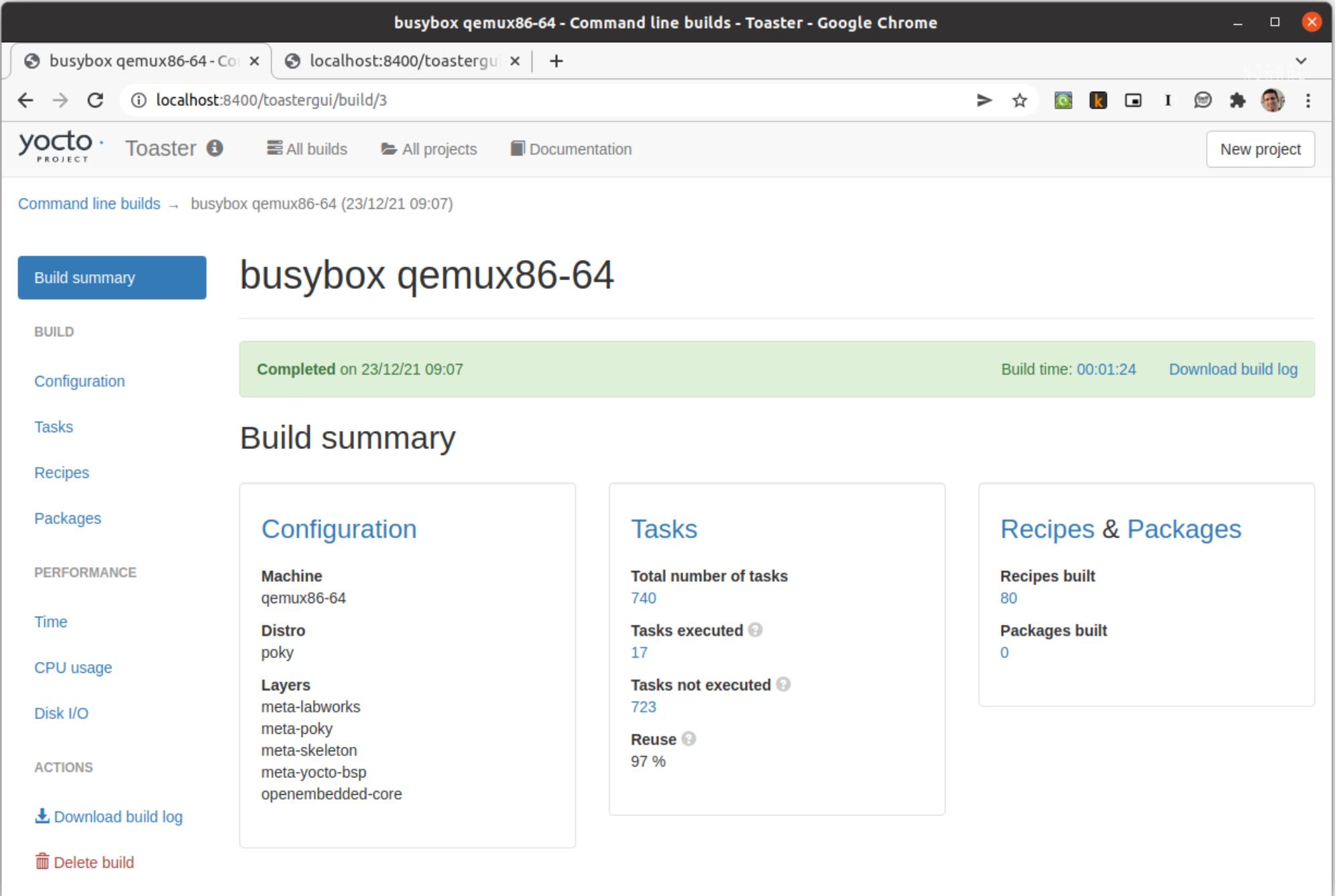
Most built recipes

- core-image-minimal

[Build selected recipes](#)



TOASTER: INSPECTING A BUILD



The screenshot shows a web browser window titled "busybox qemux86-64 - Command line builds - Toaster - Google Chrome". The URL is "localhost:8400/toastergui/build/3". The page displays a build summary for the "busybox qemux86-64" project, which was completed on 23/12/21 09:07. The build time was 00:01:24, and a download link for the build log is available.

Build summary

busybox qemux86-64

BUILD

Completed on 23/12/21 09:07 Build time: 00:01:24 Download build log

Tasks

Build summary

Configuration

- Machine: qemux86-64
- Distro: poky
- Layers:
 - meta-labworks
 - meta-poky
 - meta-skeleton
 - meta-yocto-bsp
 - openembedded-core

Recipes & Packages

- Recipes built: 80
- Packages built: 0

ACTIONS

- [Download build log](#)
- [Delete build](#)

BUILD APPLIANCE

- Build Appliance is a virtual machine image that allows anyone to try and use the Yocto Project on machines running operating systems other than Linux distributions (Windows, MacOS, etc).

<https://www.yoctoproject.org/software-item/build-appliance/>

- It's not recommended for development and production environments.
 - It is focused on people that want to try and experiment with the Yocto Project.

- Available for download from the Yocto Project tools page.

<https://www.yoctoproject.org/software-overview/downloads/>



CROPS

- CROPS (CROss PlatformS) is a project that provides a Docker container with a preconfigured Yocto Project development environment.
<https://github.com/crops/poky-container>
- Makes it possible to maintain a build environment on any operating system compatible with Docker (Windows, Linux, MacOS, etc).
- Example command to start CROPS:

```
$ docker run --rm -it -v /home/myuser/mystuff:/workdir crops/poky --workdir=/workdir
```

- Instructions on how to configure and use CROPS are [documented in the Yocto Project development manual](#).



AUTOBUILDER

- Autobuilder is an automation tool maintained by the Yocto Project.
<https://autobuilder.yoctoproject.org/>
- It's a continuous integration (CI) tool:
 - Allows you to identify build issues when new commits are introduced.
 - Assists in integration tests with external repositories.
 - Helps to feed the build cache (sstate cache).
- Internally, Autobuilder uses another project called [Buildbot](#).

LAB 9

EXPLORING ADDITIONAL TOOLS AND RESOURCES



YOCTO PROJECT

FINAL CONSIDERATIONS

LINKS

- Yocto Project website:
<https://www.yoctoproject.org/>
- OpenEmbedded website:
http://www.openembedded.org/wiki/Main_Page
- Yocto Project documentation:
<https://docs.yoctoproject.org/>

BOOKS

- [Embedded Linux Development with Yocto Project](#) - Otavio Salvador/Daiane Angolini
- [Embedded Linux Development using Yocto Projects](#) - Otavio Salvador/Daiane Angolini
- [Embedded Linux Projects Using Yocto Project Cookbook](#) - Alex González
- [Embedded Linux Systems with the Yocto Project](#) - Rudolf Streif

YOCTO PROJECT

THANKS!

