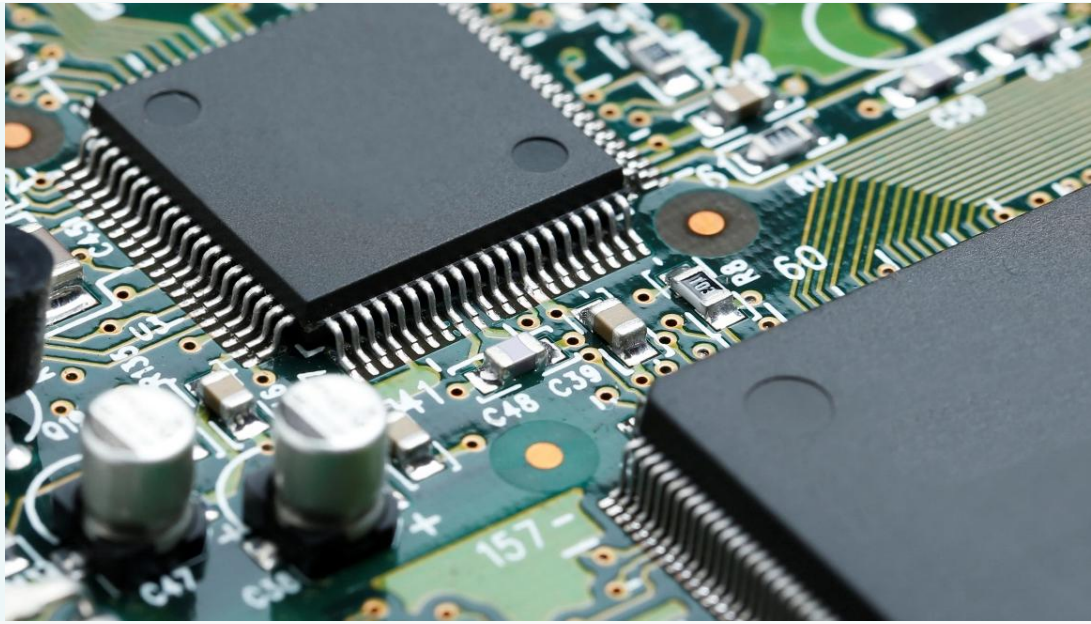
Cursul #5

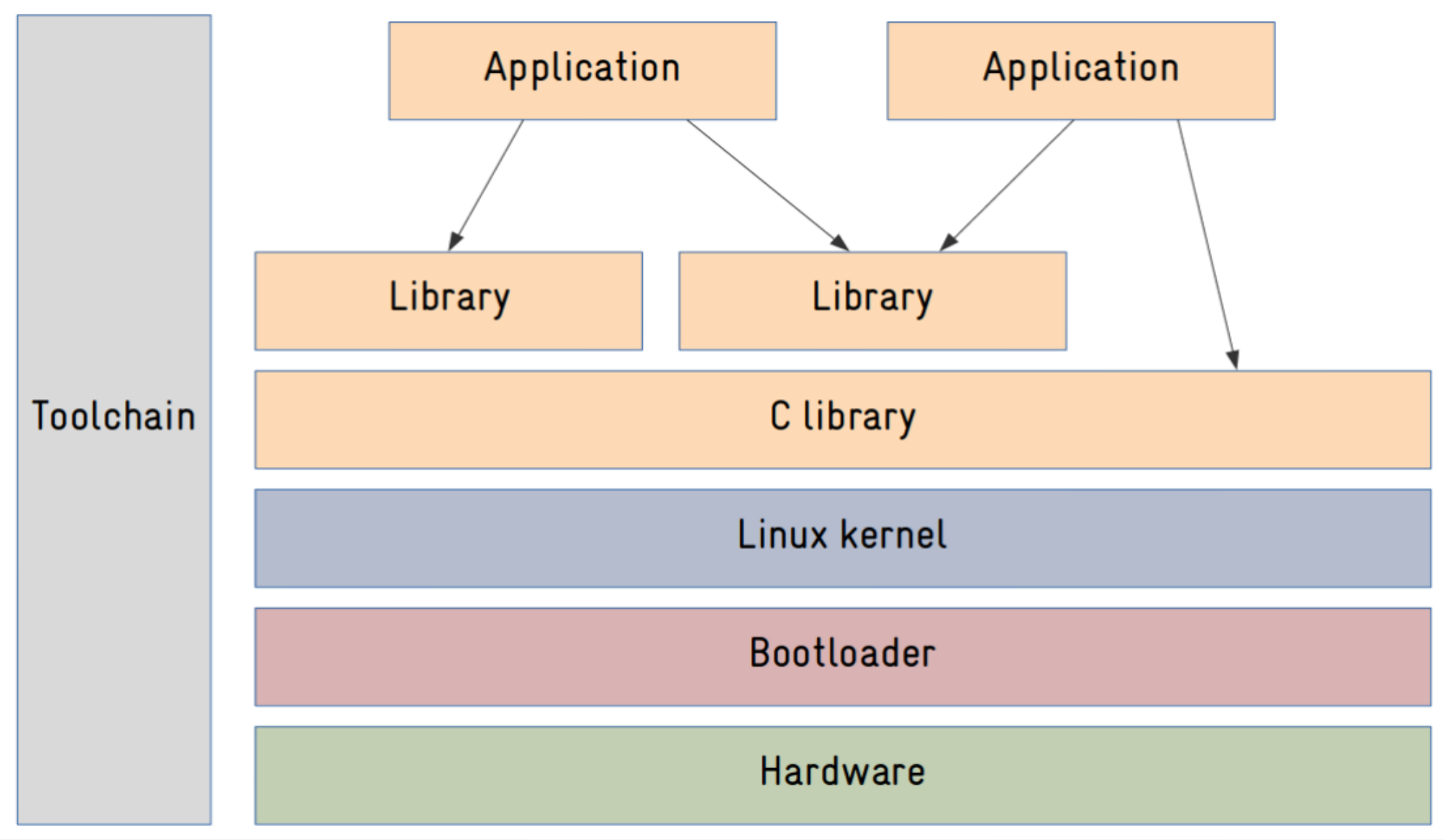
Intro to Yocto Project

3/30/2026



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.

Yocto Project

- Collaborative project that provides a set of tools to help create custom Linux distributions for embedded devices.
- Created in 2010 by several technology companies, hardware manufacturers and embedded Linux so ware solution providers.
- Under the governance of the Linux Foundation, ensuring the project's independence from any member of the organization.
- A new Yocto Project version is expected to be released every 6 months (typically in April and October).

Yocto Project Members

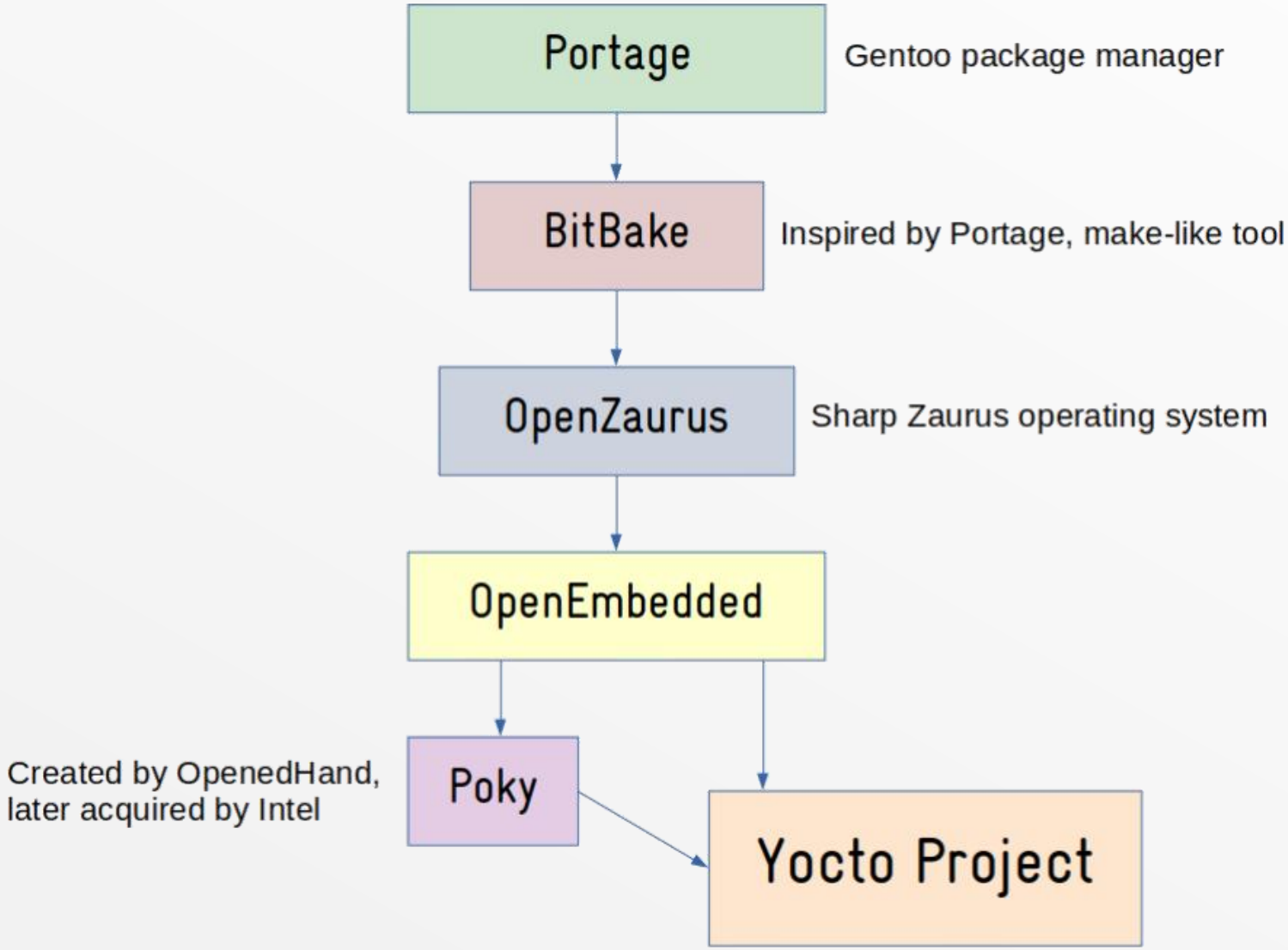
Projects

- 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/so ware-overview/project-components/

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!

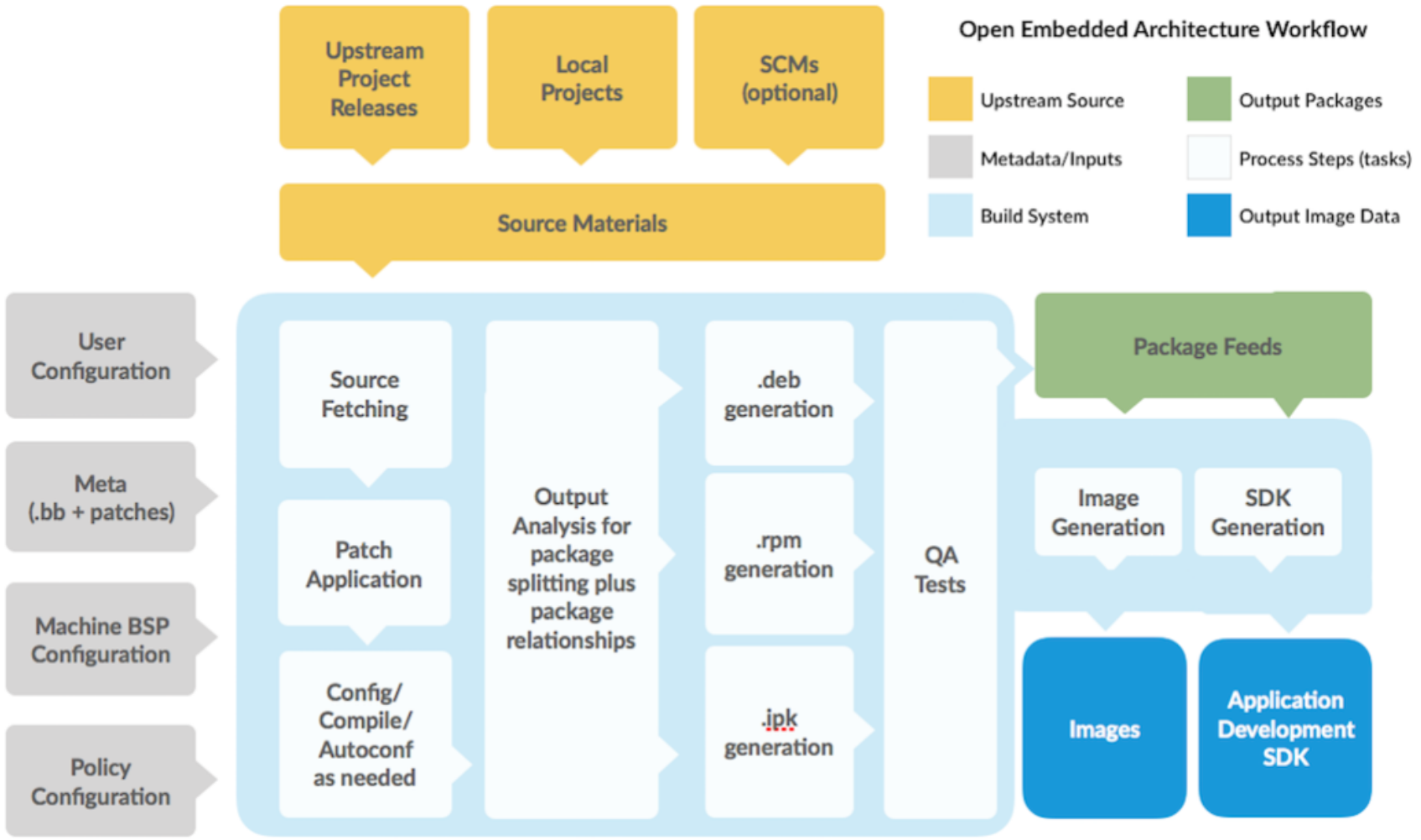
History lesson



Features and advantages

- Extremely configurable and flexible.
- Thousands of pre-configured so ware 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 so ware licenses (e.g. removing GPLv3).
- 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.

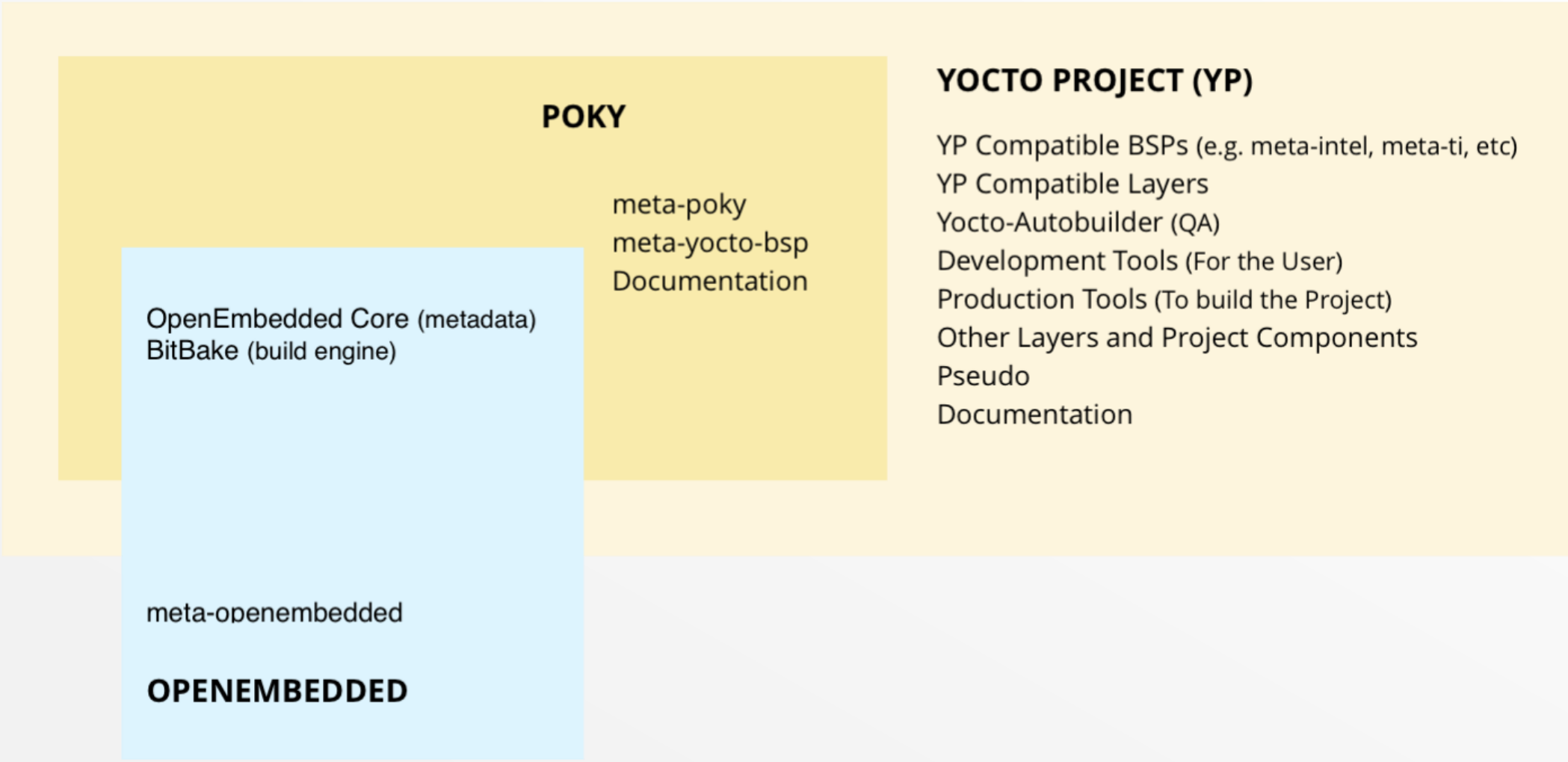
Basic architecture

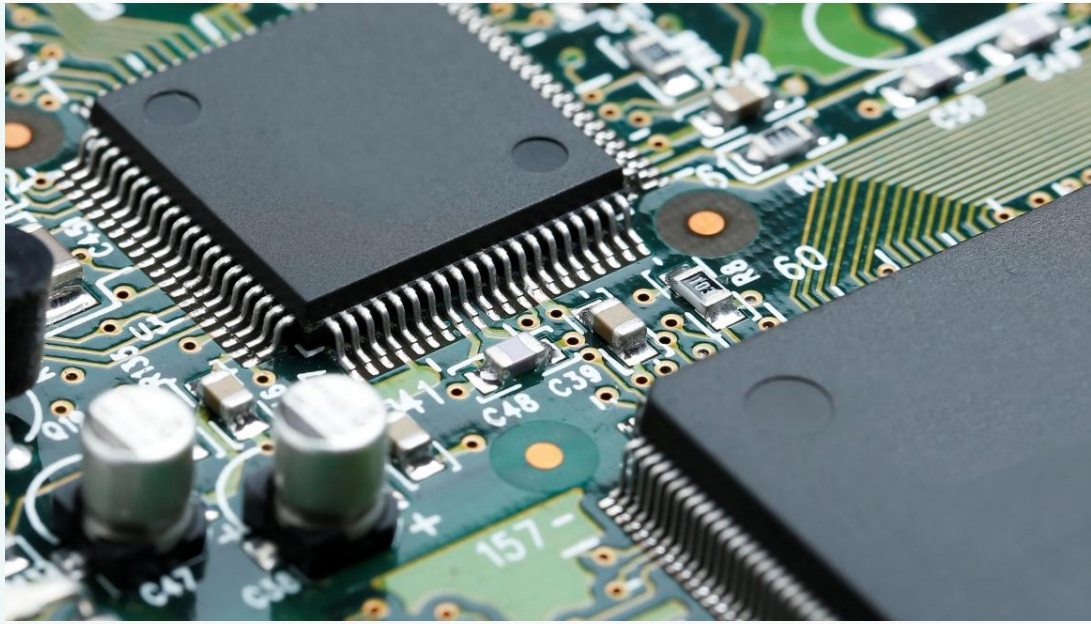


Layers and metadata

- Layers contain metadata for compiling the so ware components that will be part of the Linux distribution.
- There are 3 types of metadata:
  - **Recipes**: set of tasks to compile certain so ware (.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





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 so ware 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

Poky directories

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

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 & local.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 \
  "
```

```
# 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 (rpm, deb, ipk)
CORE_IMAGE_EXTRA_INSTALL	Additional packages to install
EXTRA_IMAGE_FEATURES	Image features to enable

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
```

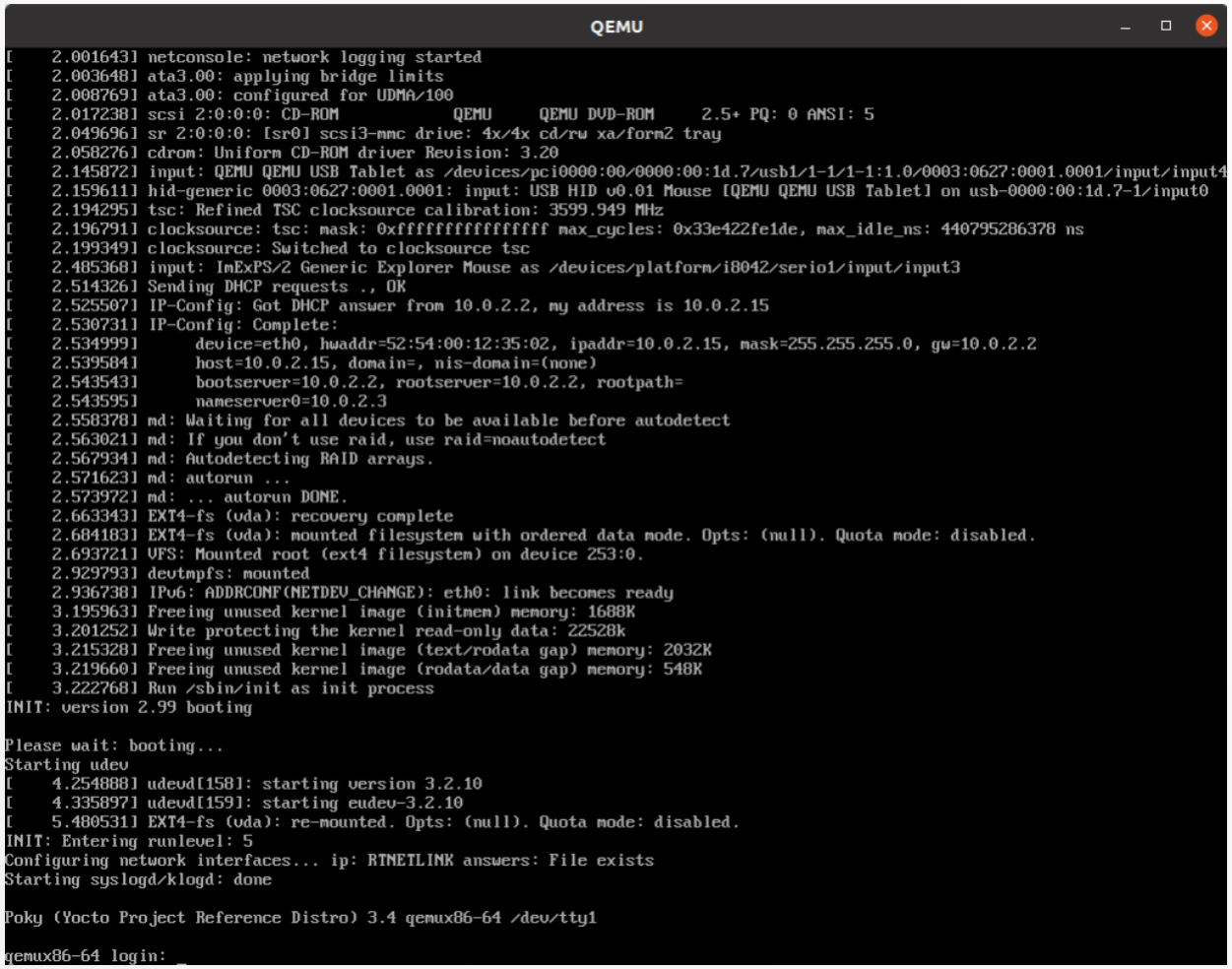
Qemu

- QEMU is an open source hardware emulator that supports multiple architectures, including x86, ARM, MIPS, and PowerPC.
- In OpenEmbedded-Core, there are several pre-configured machines for QEMU:
  - $ ls poky/meta/conf/machine/
  - qemuarmv5.conf qemuppc64.conf qemuriscv64.conf
  - qemuarm64.conf qemumips64.conf qemuppc.conf
  - qemux86-64.conf qemuarm.conf qemumips.conf
  - qemuriscv32.conf qemux86.conf

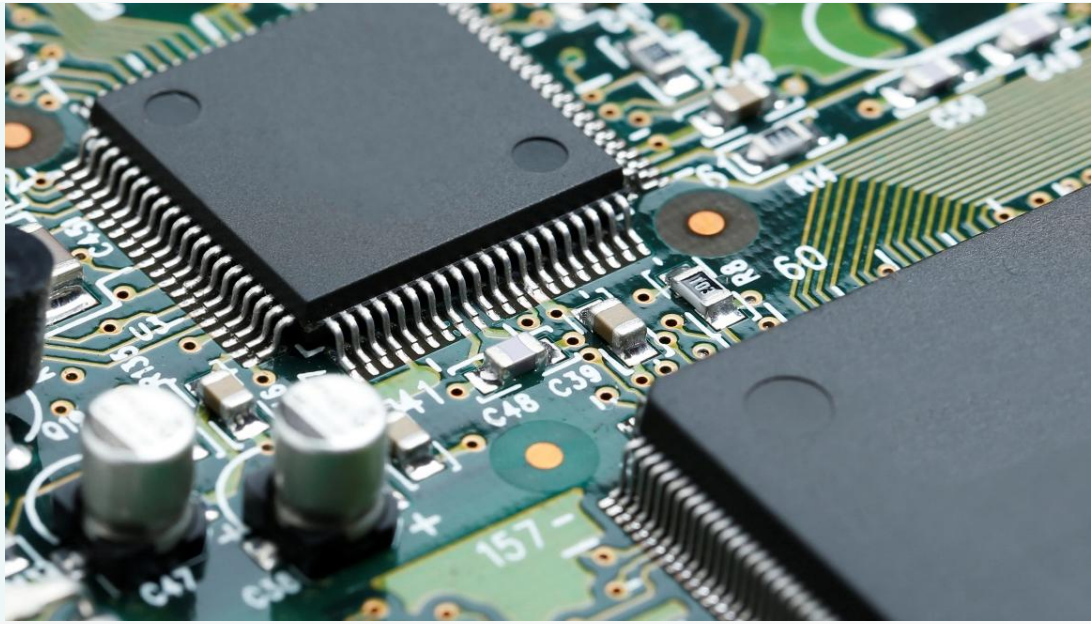
Qemu

- 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 runqemu script to run the generated image: runqemu qemuarm

Qemu


QEMU
[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: 0xffffffffffffffff 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] VFS: 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
[5.480531] 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:

Keywords



Keywords list

- **Append files**
- **BitBake**
- **Build directory**
- **Build system**
- **Classes**
- **Configuration file**
- **Cross-development toolchain**
- **Image**
- **Layer**
- **Metadata**
- **OE-core**
- **Package**
- **Poky**
- **Recipe**
- **Source directory**
- **Tasks**
- **Upstream**

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 so ware 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 so ware based on Autotools.
  - cmake.bbclass: tasks to build so ware 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

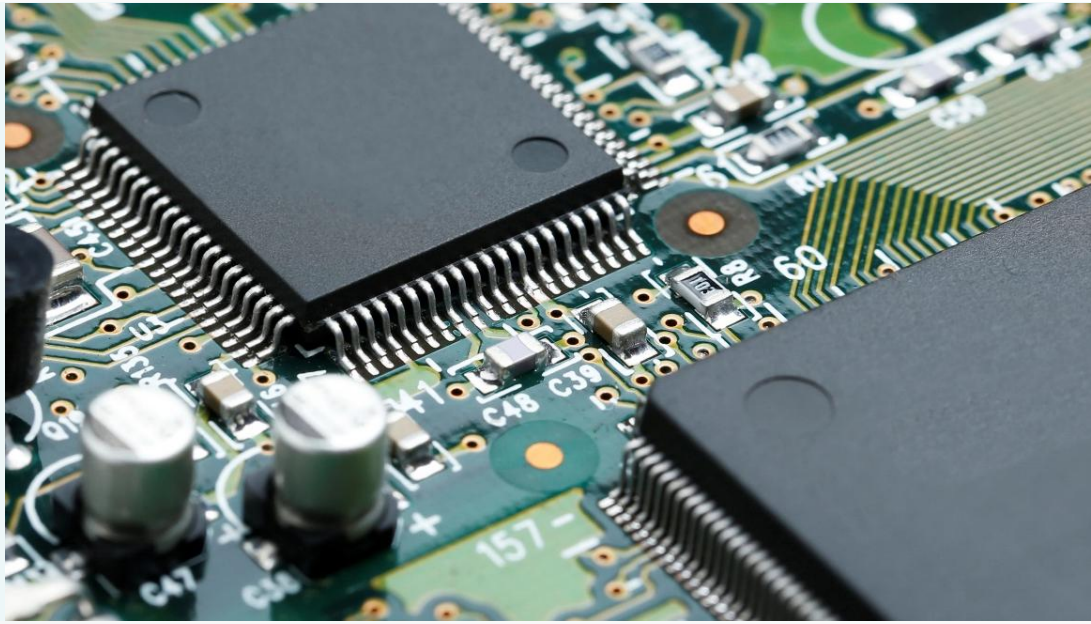
- Configuration files have the .conf extension and contain the definition of variables that will parameterize the build system.
- Among other things, it allows to define information about the hardware architecture and platform, image organization and content, distribution policies, etc.
- Only variable 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.

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).
- The Yocto Project maintains a selected list of layers officially supported by the project.
- OpenEmbedded also maintains a more complete (but less validated) layers database.
- Be aware that hardware manufacturers, vendors and software providers might not list their layers in these databases.



Bitbake

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
- 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

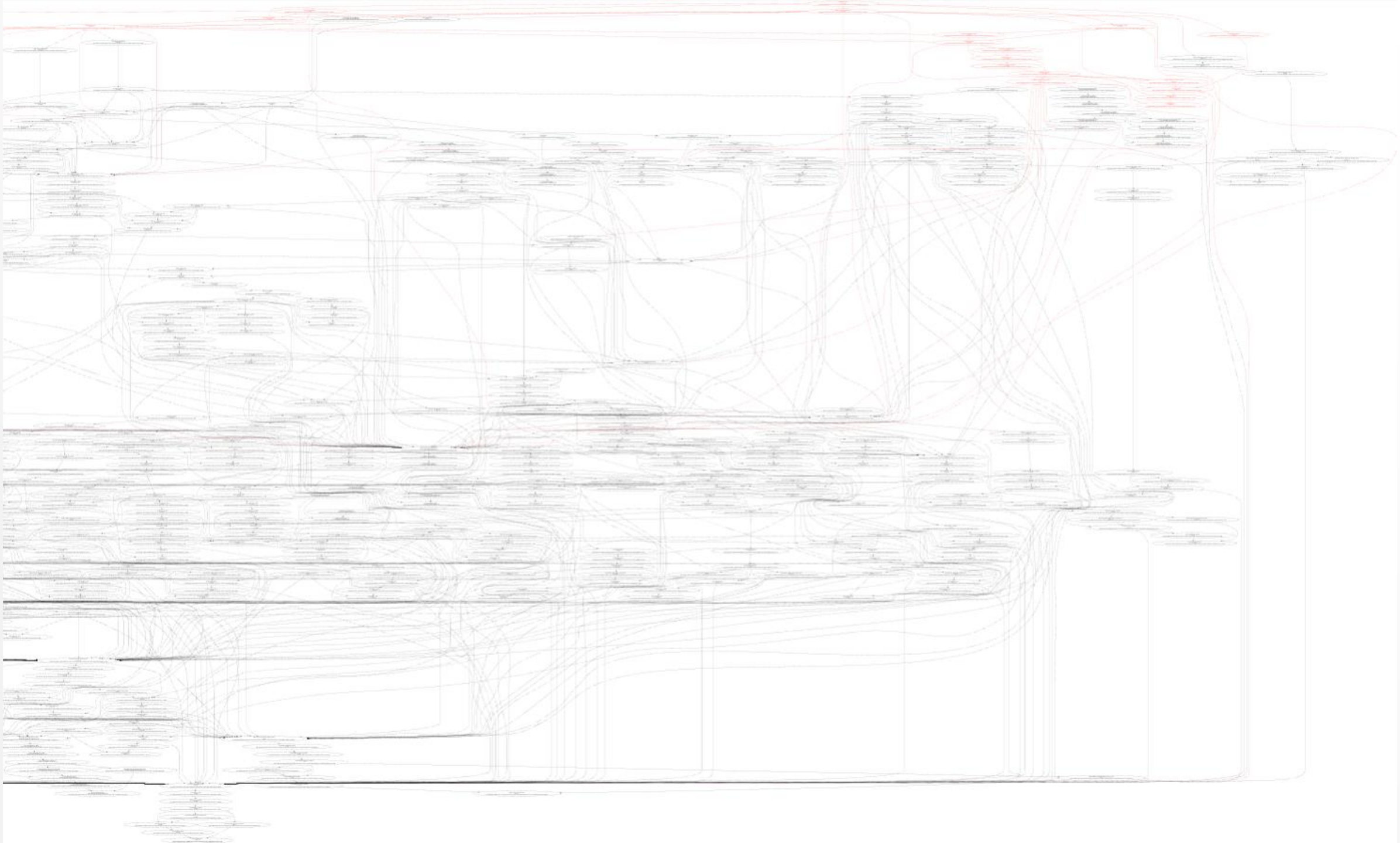
- To clean the build output (does not clean the caches), execute the clean task:
  - $ bitbake -c clean busybox
- To clean the build output and the build caches, execute the cleansstate task:
  - $ bitbake -c cleansstate busybox
- To clean the build output, the build caches, and the downloaded source code, execute the cleanall task:
  - $ bitbake -c cleanall busybox

Parsing configuration files

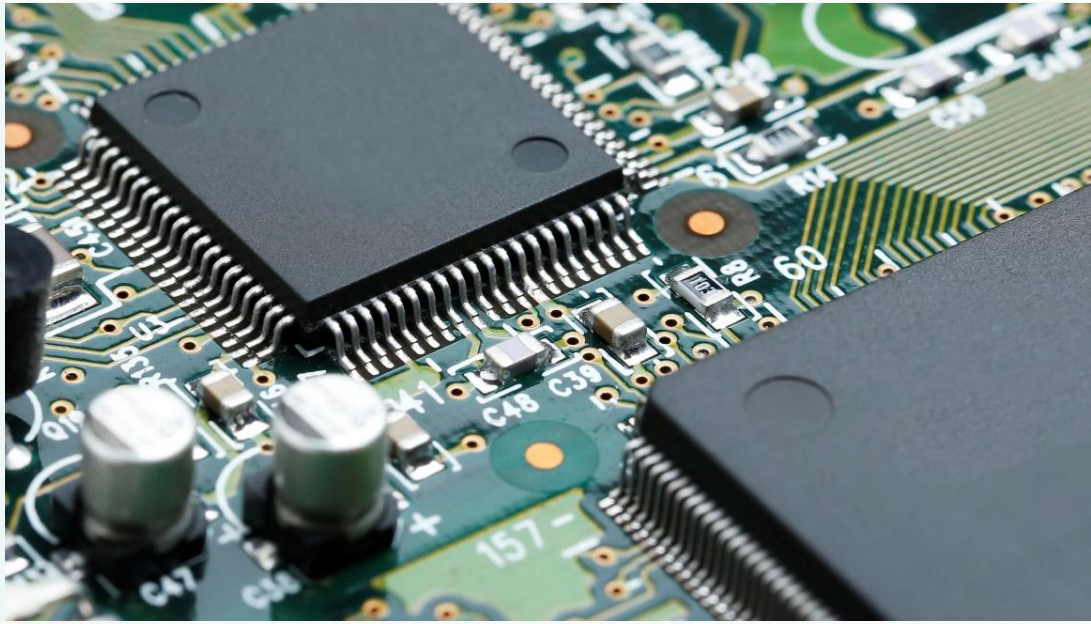
- The first step performed by BitBake is reading the configuration files, in the following order:
  - bblayers.conf, which contains the definition of the layers in the BBLAYERS variable.
  - Configuration files (layer.conf) for each of the layers included in BBLAYERS.
  - bitbake.conf file, which contains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

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



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:
  - 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"
```

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.

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 \
"
```

- Use bitbake-layers command as alternative

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.

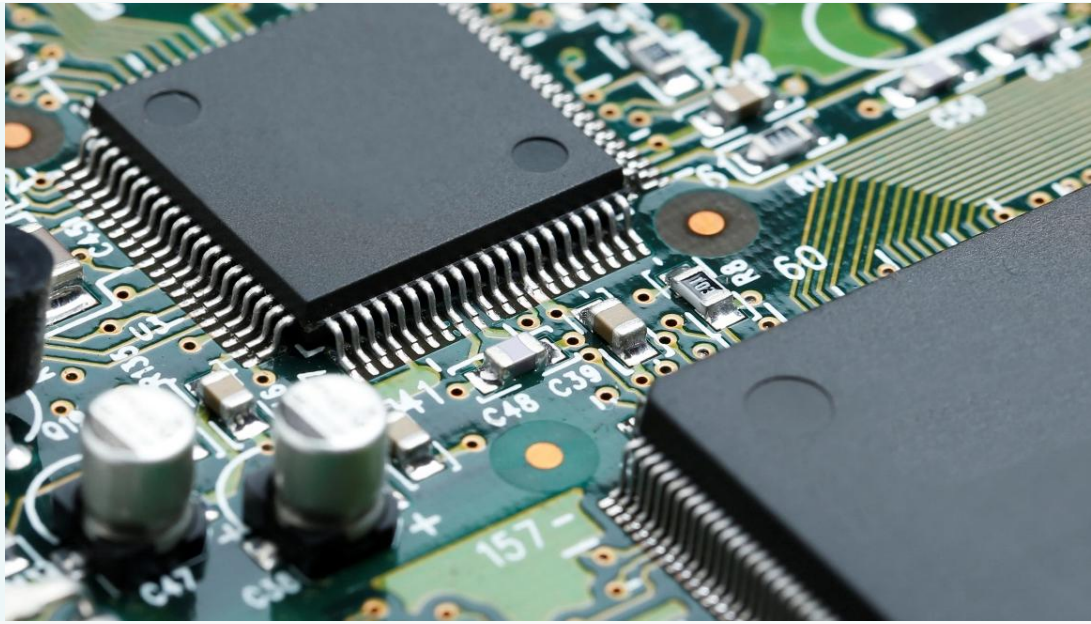
Compatibility program

- 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.

Managing layers

- Try to organize the metadata in layers to make it easier to maintain:
  - Separate BSP, distro and so ware 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.

Recipes



Recipes

- Recipes are files with the .bb extension, processed by the BitBake tool to generate the various so ware 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.
- 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 ass
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 list inside a recipe

```
$ 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 so
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
do_deploy_source_date_epoch_setscene   (setscene version)
do_devshell                  Starts a shell with the environment set up for development/debu
do_fetch                    Fetches the source code
do_install                  Copies files from the compilation directory to a holding area
do_listtasks                 Lists all defined tasks for a target
do_package                   Analyzes the content of the holding area and splits it into sub
do_package_qa                Runs QA checks on packaged files
do_package_qa_setscene           Runs QA checks on packaged files (setscene version)
do_package_setscene              Analyzes the content of the holding area and splits it into sub
do_package_write_rpm             Creates the actual RPM packages and places them in the Package
do package write rpm setscene 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

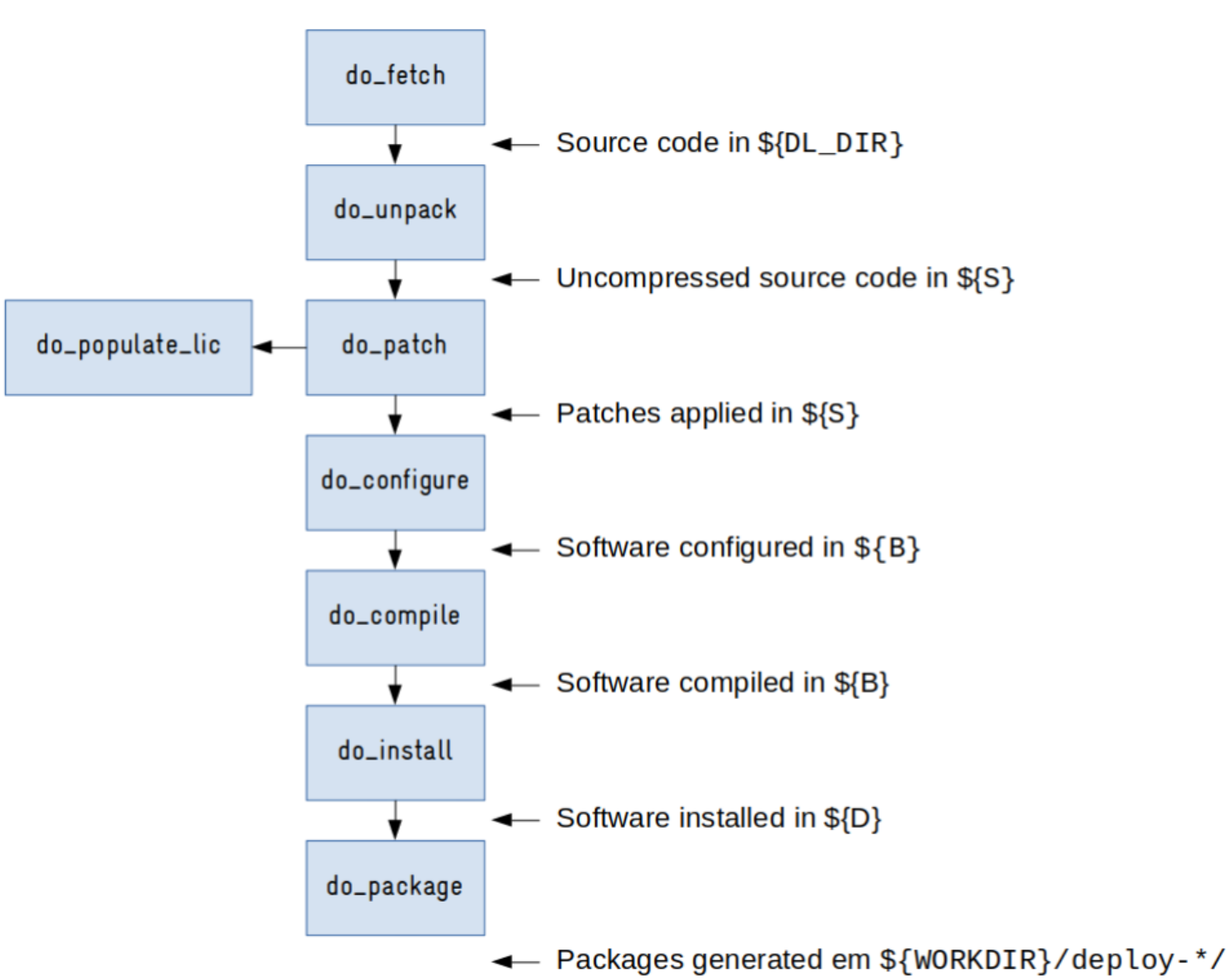
Variables

- 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 so ware is compiled)
D	Destination directory (where the so ware is installed)

Processing a recipe



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

Recipe template

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

LICENSE = ""
LIC_FILES_CHKSUM = ""

SRC_URI = ""
SRC_URI[sha256sum] = ""

DEPENDS = ""

inherit <some_class>
```

```
DESCRIPTION = "Hello World application"

HOMEPAGE = "git://mygitserver.com/hello.git"

SECTION = "tests"


LICENSE = "MIT"

LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=73636211010193737465
65758497289305"


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}

}
```

Q&A