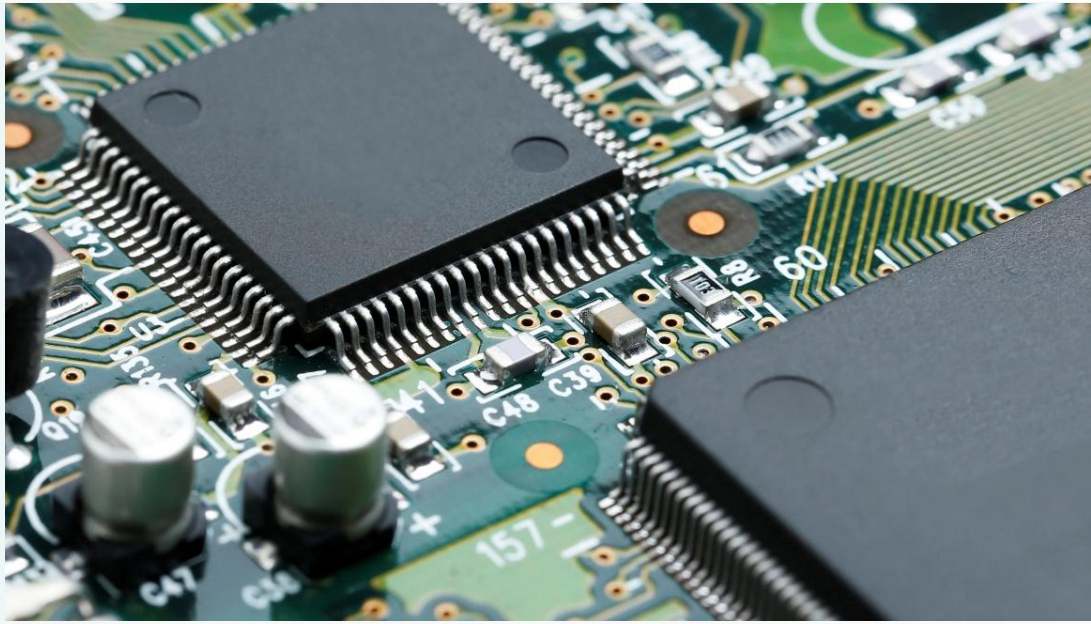
Cursul #6

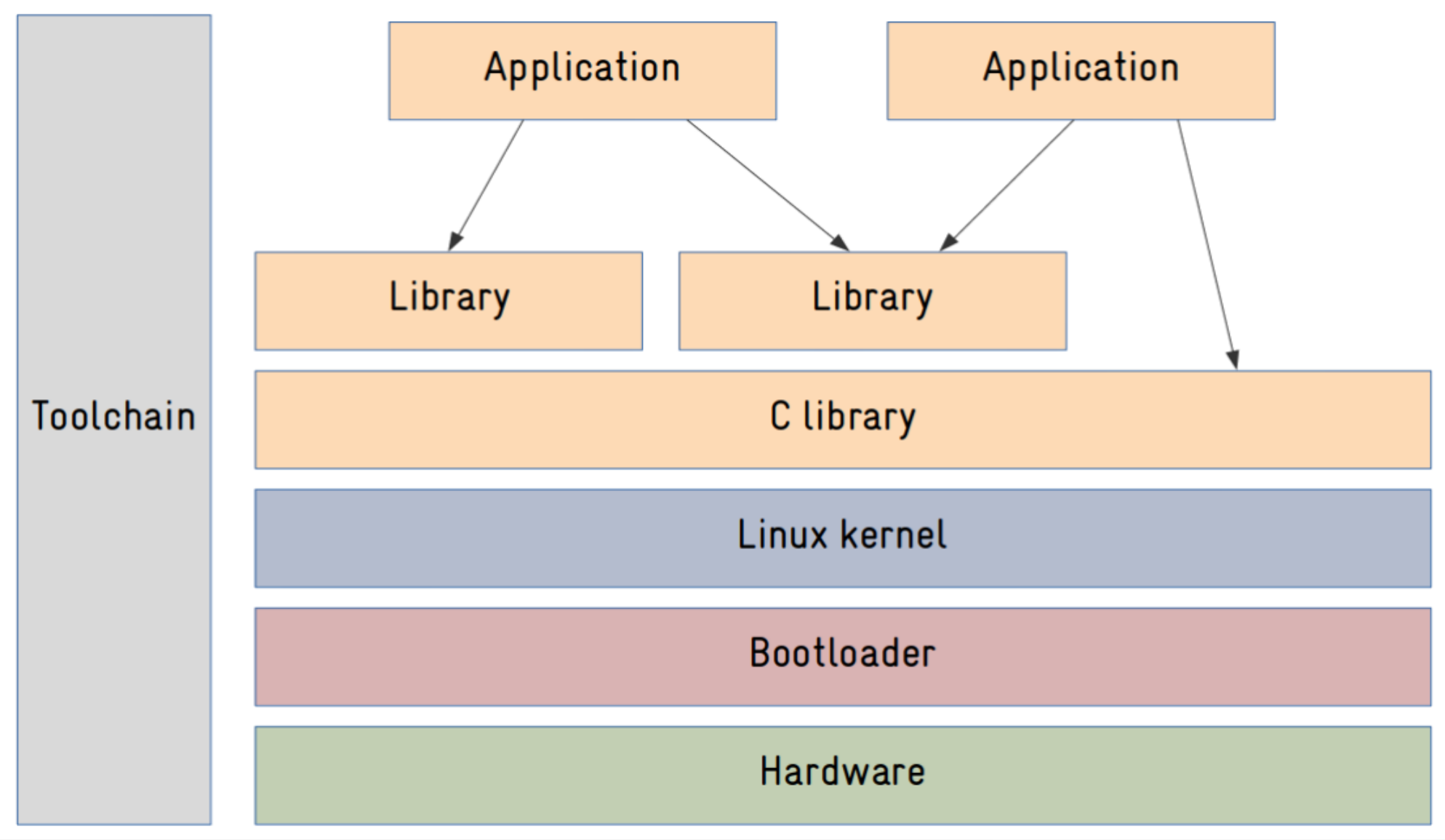
Intro to Yocto Project

4/20/2026

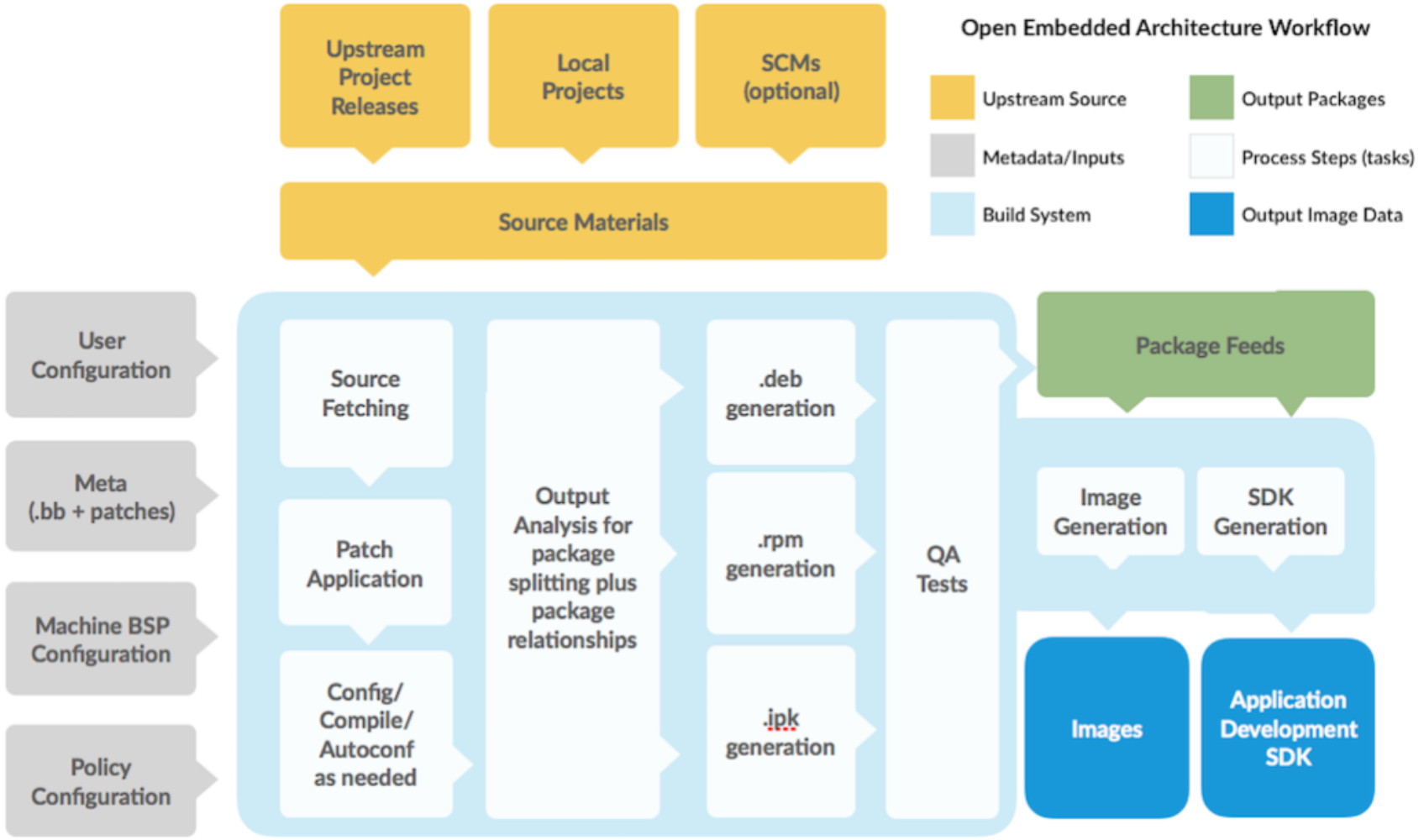


Yocto Project

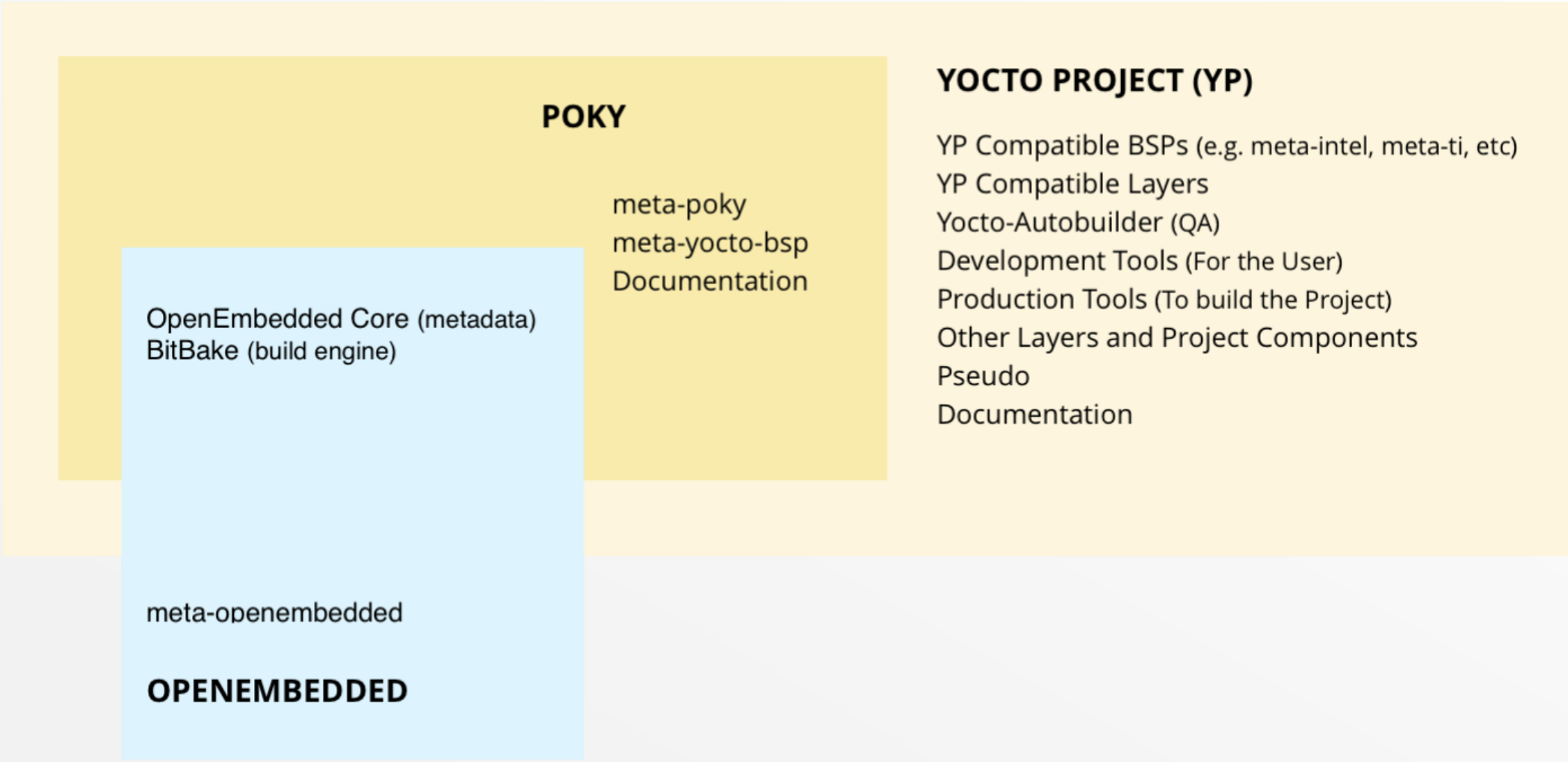
Embedded Linux



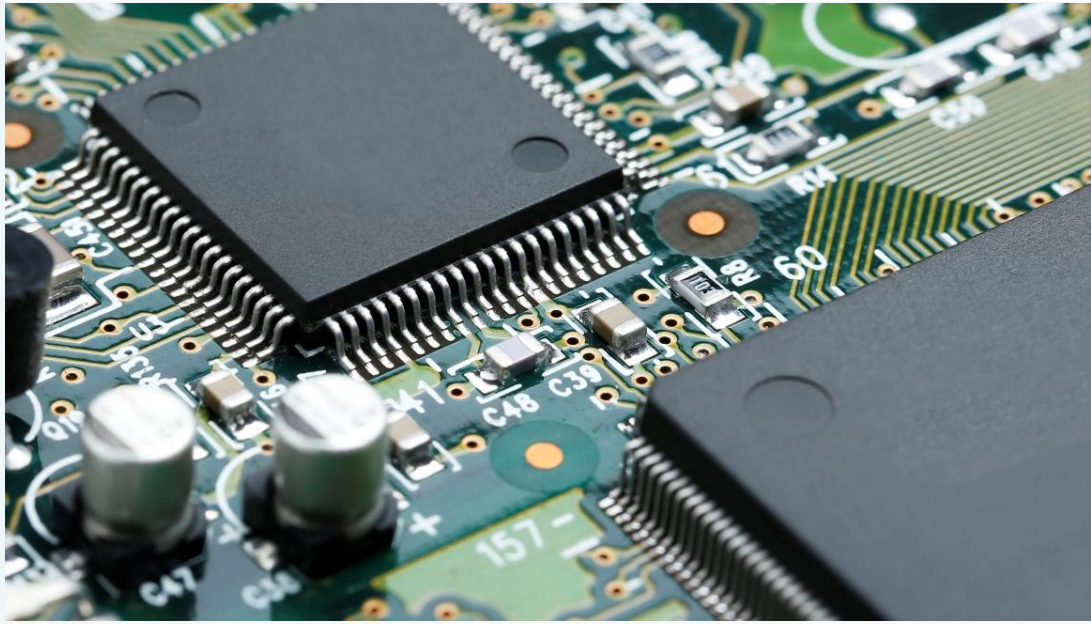
Basic architecture



Yocto Project components



Development



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).

Packageconfig

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

```
PACKAGECONFIG ??= ""
PACKAGECONFIG[f1] = "\
  --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"
```

Package versioning

- Bitbake –s, in order to identify the latest version for the available recipes.
- 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.

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.
- 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"
- The PREFERRED_VERSION variable can be used in a configuration file to define the desired version.
  - PREFERRED_VERSION_gtk+ ?= "2.13.3“

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.

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

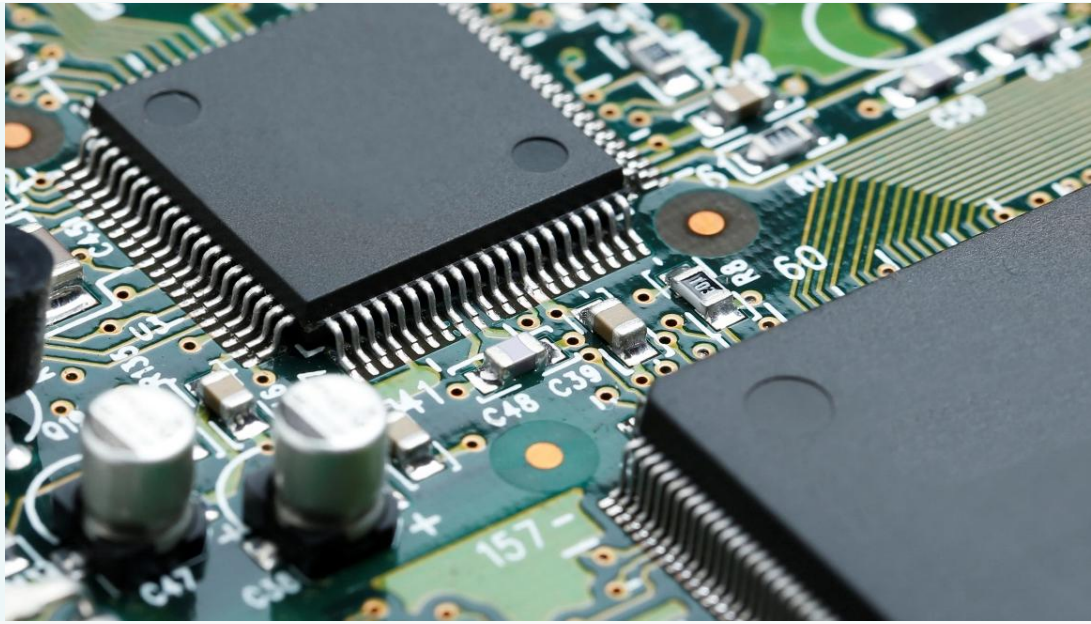
Inspecting packages

- 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.

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.

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 so ware 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

- Usually, image recipes are stored in a directory called images.
- It is very common to create new image recipes when working with the Yocto Project.
- 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.

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.

Adding image 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!

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
kernel-5.14.15-yocto-standard qemux86_64 5.14.15+git0+edb4dc09c5_f04b30fc15
...
```

Packagegroups

- Packagegroups allow grouping so ware 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.
- Typically, packagegroups are located in a directory called packagegroups

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

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

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.

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 so ware, we can change the corresponding recipe to add it to the image.
- In case the file is not associated with any so ware, we can create a specific recipe to add it to the image.

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.

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
... s
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 a er 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$4vblknvGr6FcDe:\
      %' ${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

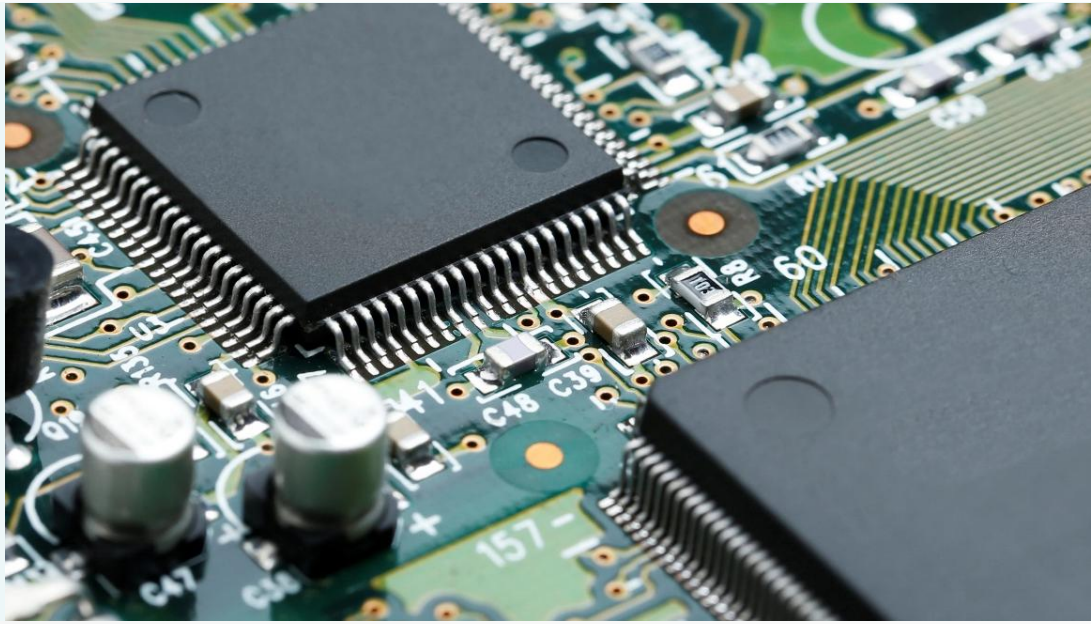
- The IMAGE_CMD variable can be used to define custom image formats (see poky/meta/classes/image_types.bbclass for more examples):

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

Distro



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.
- 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_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
DISTROOVERRIDES	Distribution-specific overrides
MIRRORS/PREMIRRORS	Alternative sources for source code download

Distro features

- Distro features allow to control the so ware 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

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.

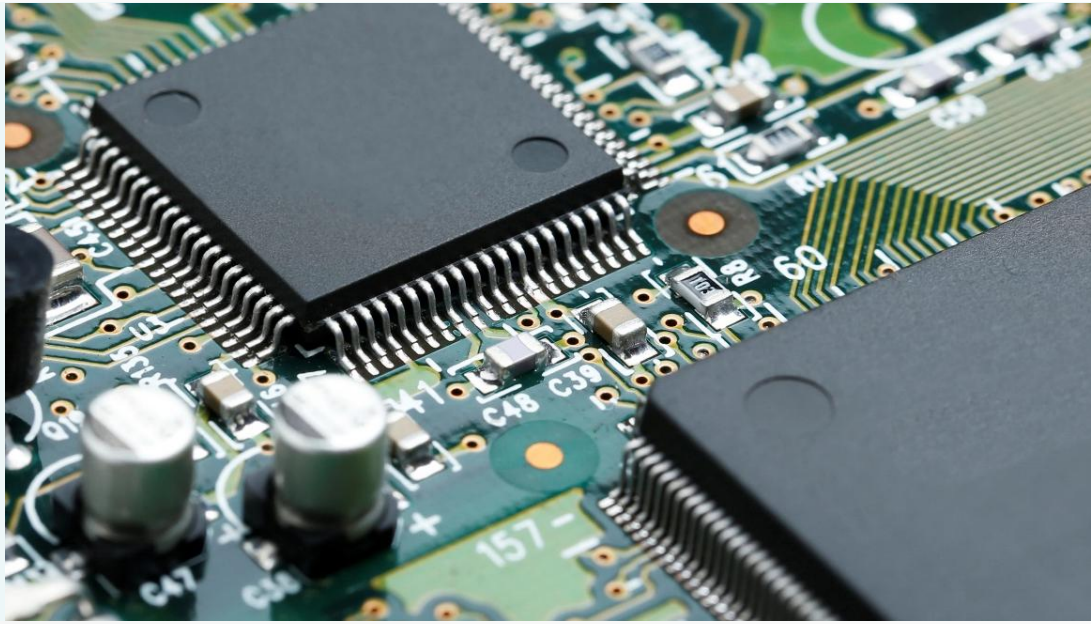
Enabling package management

- 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

BSP layer



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
│   └── u-boot
├── 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 MACHINEOVERRIDES

- 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 MACHINEOVERRIDES:

```
SOC_FAMILY =. "mx6:mx6dl:"
include conf/machine/include/soc-family.inc
MACHINEOVERRIDES =. "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 so ware artifact (e.g. a kernel image) might be provided by more than one recipe.
- For a recipe to indicate that it provides a certain so ware 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%"
```

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 so ware is compiled.
- Machine features are documented in the Yocto Project's Reference Manual. https://docs.yoctoproject.org/ref-manual/features.html#ref-features-machine

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.

Enabling the machine

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

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

- 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.

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

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.

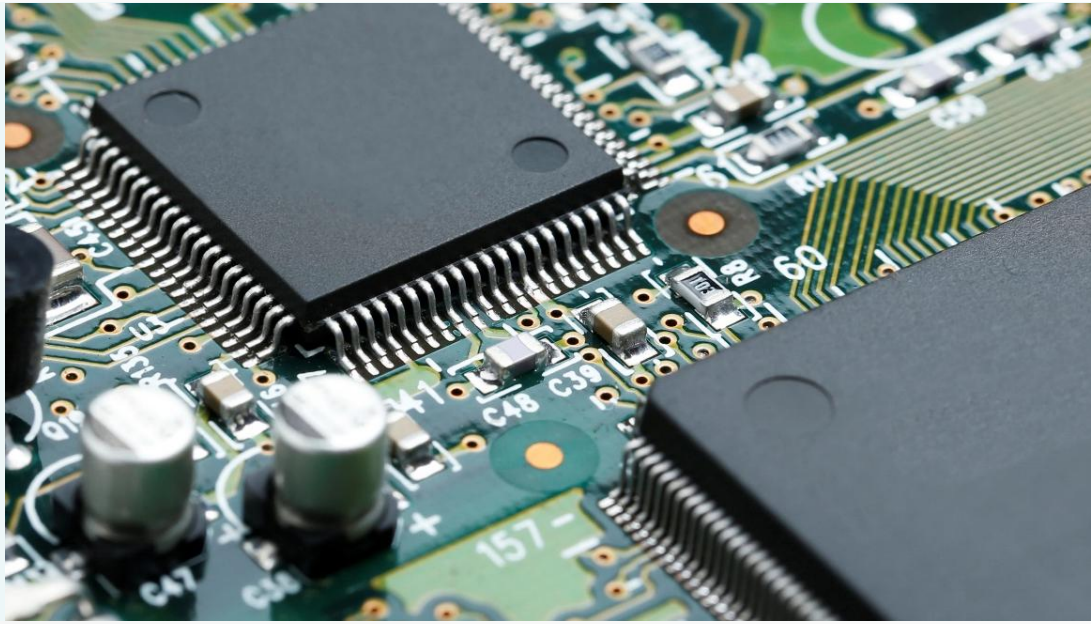
Patching the kernel

- 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.

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

SDK



Working with build system

- Although possible, using the build system for so ware development is usually not effective.
- The build system is an integration tool focused on helping with building and maintaining the so ware 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 (So ware 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 so ware 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 the 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.mani

  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.ma

  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.

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.testd
  ```

- 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_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.targe
  poky-glibc-x86_64-core-image-training-cortexa9t2hf-neon-colibri-imx6-training-toolchain-ext-4.0.1.testd
  ```

- 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.

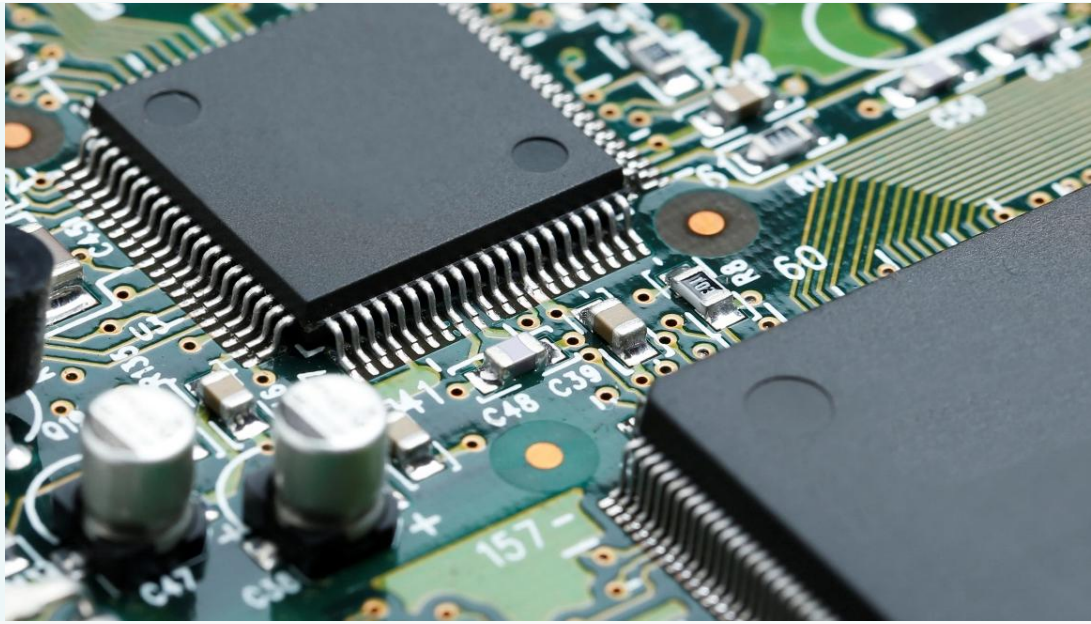
Devtool

- 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.

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

Additional tools



License management

- One of the concerns when developing products that uses free so ware 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-license

$ ls poky/meta/files/common-licenses/

0BSD	DRL-1.0	NLOD-2.0
AAL	DSDP	NLPL
Abstyles	DSSSL	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

...

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 so ware 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 so ware is proprietary.
  - LICENSE = "CLOSED"
- Commercial or special so ware 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 so ware 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 so ware 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 so ware components of a certain license.

Distributing the license text

- By default, licenses for all so ware 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.

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.

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.

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.

Q&A