Introduction to yocto PROJECT / OpenEmbedded

Pierre Ficheux (pierre.ficheux@smile.fr)

December 2023

History and basics

“*Yocto* (symbol **y**) is a unit prefix in the metric system denoting a factor 10^{-24}

It was adopted in 1991 by the General Conference on Weights and Measures. It comes from the Latin/Greek octo (ὀκτώ), meaning ‘eight’ $(10^{-3})^{8}$. Yocto is the smallest official SI prefix.”

Also sprach Wikipedia !

OpenEmbedded, introduction

- OE is a “cross-compilation framework”
- Started in 2003 by Chris Larson, Michael Lauer and Holger Schuring for OpenZaurus to replace Buildroot
- The Zaurus (SHARP) is the “first” PDA running Linux OS in 2001

OE (Yocto) principles

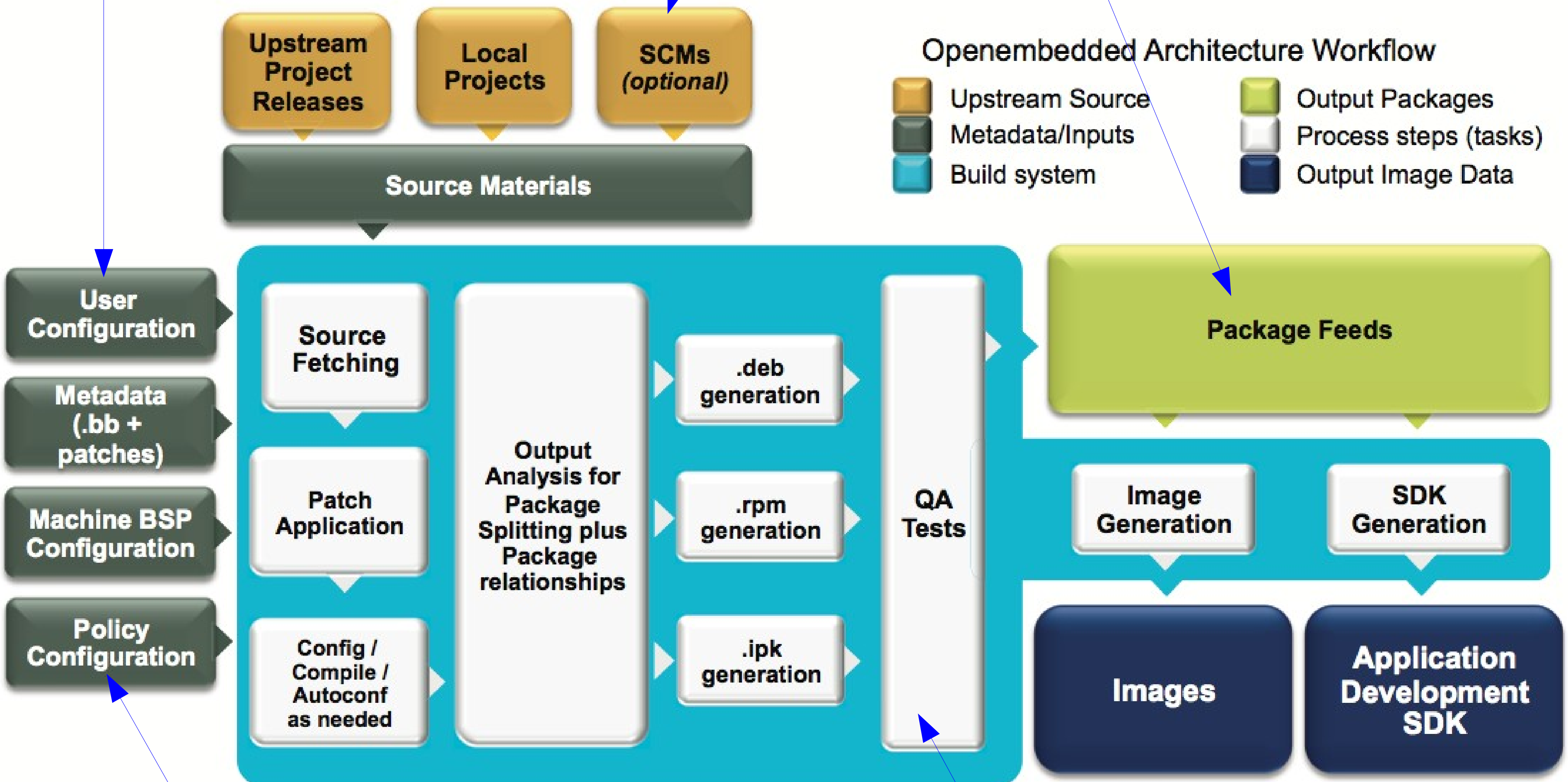
- OE uses an “engine” named *BitBake* (written in Python and inspired by *Portage - Gentoo*) and a set of “recipes”
- A recipe contains a `.bb` file (**B**it**B**ake)
- OE uses “inheritance” (in a recipe or in global configuration)
- Recipes use “classes” (`.bbclass`)
- Many “external” recipes (and so external projects)
- Every component, including the kernel, the bootloader and the Linux image uses a recipe
- OE provides package management (IPK, then RPM and DEB with Yocto)
- OE can produce a “standalone” SDK (a shell script)

Yocto/OE “workflow”

Using Source Control Management

`build/conf/*.conf`

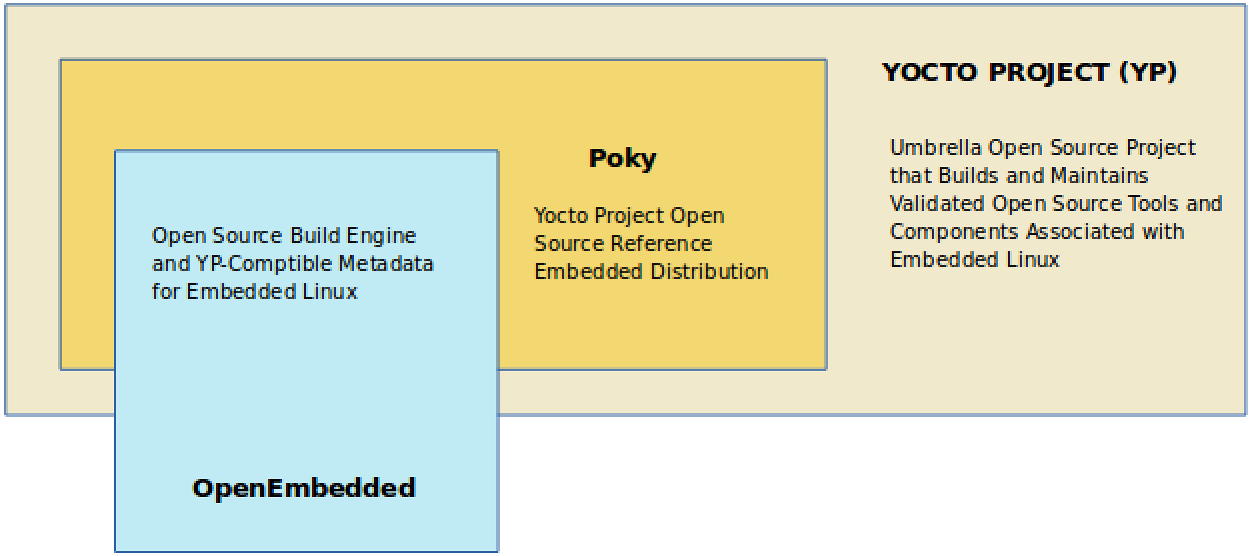
Validated packages (QA)



Quality Assurance (“insane” class)

Depends on DISTRO (default is “poky”)

- Yocto is an umbrella open source project started in 2010
- It gathers numerous upstream projects like OE, BitBake, Poky, EGlibc, etc.
- Many members including Intel, Linaro, NXP, Huawei, TI, Juniper, Wind River, Mentor Graphics, etc.
- Project architect is Richard Purdie, who joins the Linux foundation as a “fellow” (just like Linus) in December 2010



The Yocto Project :-)

- Organization similar to Linux kernel one

  “meritocracy presided over by a benevolent dictator”
- A real collaborative project
- A new release every six months
- Increased popularity in the industry since the beginning of the Yocto project which integrates OE
  - Support from the Linux Foundation, Intel & friends
  - Good documentation → big improvement compared to the initial OE project
  - Possible to create images from a few MB to several hundred MB
  - Standalone SDK and eSDK (with Devtool)

- Used by HW makers (NXP, TI, Xilinx, etc.) for BSPs
- Used by editors for their products → Wind River
- Yocto is *not* an embedded Linux distribution, but provides “templates” and a set of tools to create a custom one → “metadata” organized as “layers”
  - HW support (meta-intel, meta-raspberrypi)
  - Distributions (meta-poky, meta-angstrom)
  - Miscellaneous components (meta-agl) → AGL
- Available layers listed in:

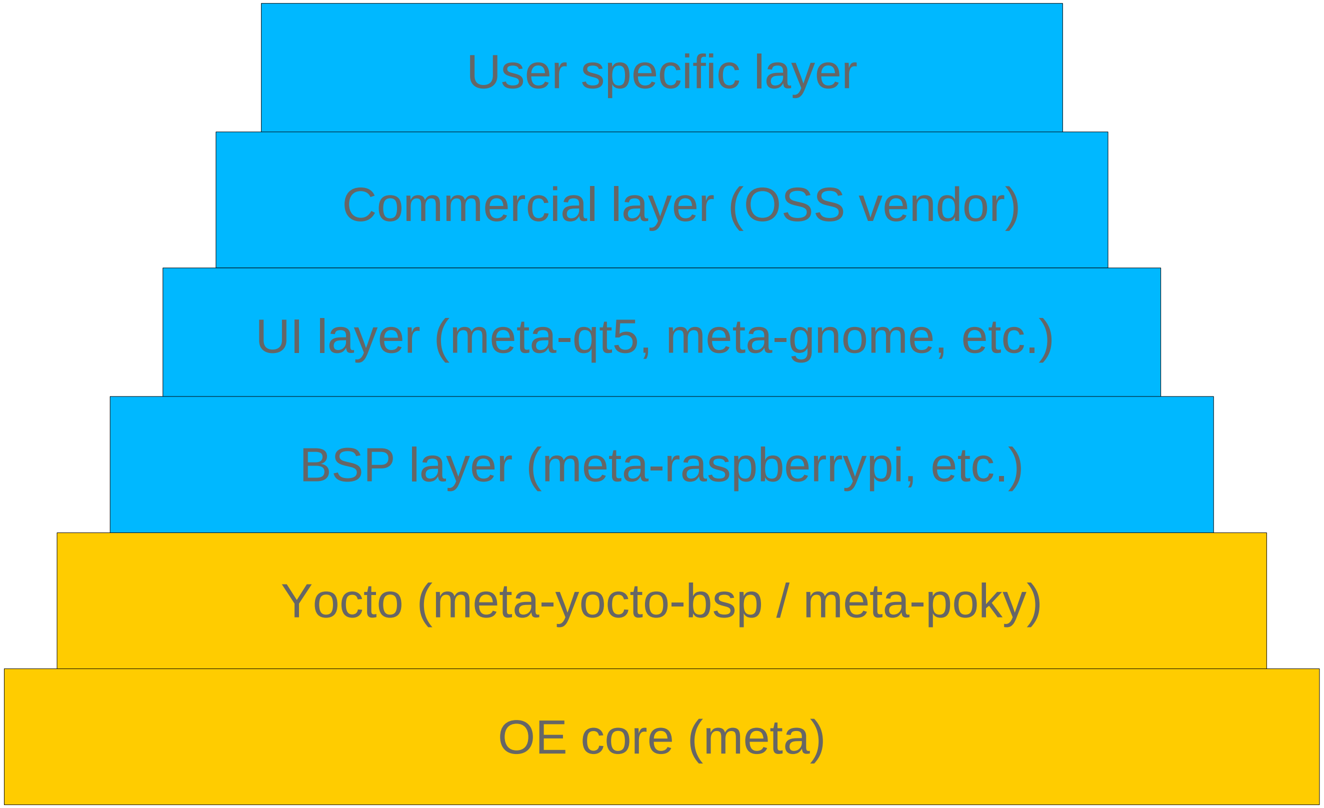
  http://layers.openembedded.org/layerindex

It's not an embedded Linux distribution

– it creates a custom one for you (Yocto Project website)

- Building a first Linux image takes lots of time (more than Buildroot !)
- Text mode only (Toaster GUI is not fair enough)
- It’s difficult to learn due to the large number of environment variables

Yocto layers principle



External project

Yocto project

- Poky is the Yocto Project reference distribution

  “Poky is a reference system of the Yocto Project - a collection of Yocto Project tools and metadata that serves as a set of working examples”
- Replaced *Angstrom,* which was the OE project reference distribution before Yocto
- The last version is 4.3 (Nanbield), 3.1.18 (Dunfell) is used for the training
- The version can be seen in `poky.conf`

```
$ grep VERSION meta-poky/conf/distro/poky.conf
```

- Boot on a QEMU/ARM

```
Poky (Yocto Project Reference Distro) 3.1.18 qemuarm /dev/ttyAMA0


qemuarm login: root
root@qemuarm:~# uname -a
Linux qemuarm 5.4.205-yocto-standard #1 SMP PREEMPT Thu Jul 14 13:03:13 UTC
2022 armv7l GNU/Linux
```

Some external “distros”

- Angstrom (meta-angstrom)
  - Reference distribution for OE
  - Back to Yocto Project in September 2012
- Arago (meta-arago-distro)
  - Provided by TI
- Yogurt (meta-yogurt)
  - Provided by Phytec

“distro” vs “image”

- The “distro” defines the build configuration policy in configuration files
  - features (such as “systemd”, “opengl”, “wayland”, etc.)
  - QA check settings
  - and more !
- The “image” defines the components you flash on the board
  - Bootloader (U-Boot)
  - Linux kernel
  - Root-filesystem (ext4, tar, etc.)
  - Additional “image features” such a SSH, X11, etc.

How it works !

- OE uses different kinds of “metadata”
  - Recipe (`.bb`) and extended one (`.bbappend`)
  - Class (`.bbclass`)
  - Include (`.inc`)
  - Configuration (`.conf`)
- A recipe describes how to build one (or several) packages for the target :

  `$ bitbake my-recipe`
- A recipe can provide build information from a single piece of software to a full Linux image

  `my-recipe.bb`

  `busybox_`**`1.31.0`**`.bb` ◄—— version defined in the recipe name

  `core-image-minimal.bb`

Class

- A class file (`.bbclass`) contains data/functions to be shared between recipes
- Some famous class files
  - Autotools → `autotools.bbclass`
  - CMake → `cmake.bbclass`
  - Linux module → `module.bbclass`
- Using a class (in a recipe or a configuration file)

```
inherit autotools # in a recipe
INHERIT += "rm_work" # in local.conf
```

- Class files are usually located in `<layer-name>/classes`

Includes

- Enable recipes to include common data using the `require` or `include` directives
- The `include` directive does not throw any error even if the file can't be found

```
require linux.inc
include ../common/firmware.inc
include conf/distro/include/security_flags.conf
```

Configuration files

- Configuration files (`.conf`) define configuration variables for the target (`poky.conf`, `bitbake.conf`, etc.)
- `local.conf` enables to define parameters for the current build directory (**target device**, binary package format, build options, etc.)
- `bblayers.conf` list the used layers for the current build directory
  - supplied by Yocto (mandatory)
  - added layers (at least from the BSP)
- Those two files are dynamically generated when creating build environment with `oe-init-build-env`

- The “chef” using recipes !
- BitBake processes configuration files to get packages to build
- Each recipe must be provided by a “provider”
- Similar to GNU-Make in the Buildroot world
- Perform the build steps of recipes (*fetch*, *unpack*, *patch*, *license*, *configure*, *compile*, *install*, *package*, etc.)
- Each step is defined as a `do_<step-name>()` function → `do_fetch()`, `do_compile()`, `do_install()`, etc.

  → see `bitbake -c listtasks` command
- The default configuration is in `meta/conf/bitbake.conf`

Some BitBake options

- BitBake provides a few useful options:

  `-D` show debug information → `-DDD`

  `-v` verbose output

  `-n` “dry run” mode → nothing is done

  `-e` show environment variables

  `-s` show available recipes

  `-c` perform a SINGLE step

  `-C` invalidate time stamp and execute task

  `-f` force execution of the operation (even if not required)

  `-g` output dependency tree

  `-u` specify the user interface to use

  → `$ bitbake -g -u depexp <recipe>`

BitBake “single steps” examples

- We can perform a *single* step (*fetch*, etc.)

```
$ bitbake -c <step> <recipe>
$ bitbake -c fetch hello          Download the source code
$ bitbake -c listtasks hello      List tasks to run
$ bitbake -c clean hello          Clean binaries (package, etc.)
$ bitbake -c cleansstate hello    Clean package + shared state cache
$ bitbake -c cleanall hello       Clean the source code too !
```

- Downloading sources (with dependencies)

```
$ bitbake <recipe> --runall=fetch
```

- To rebuild packages from a recipe

```
$ bitbake -c cleansstate <recipe>
$ bitbake <recipe>
```

Building and testing an image

Yocto “in a nutshell” (QEMU/ARM)

- Download Poky 3.1.x (Yocto reference distro)

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

- Create the build directory

```
$ cd poky
$ source oe-init-build-env qemuarm-build
```

- Set the target device in `conf/local.conf`

```
# This sets the default machine to be qemux86-64 if no other
machine is selected:
MACHINE ??= "qemux86-64"
...
MACHINE = "qemuarm"
```

- Build a basic image

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

- Test the image (exit QEMU with Ctrl-A X if text mode !)

```
$ runqemu [qemuarm] [image-name] [nographic]
```

- Download the Pi BSP layer (meta-raspberrypi)

```
$ cd poky
$ git clone -b dunfell git://git.yoctoproject.org/meta-raspberrypi
```

- Create the build directory

```
$ source oe-init-build-env rpi3-build
```

- Add the Pi 3 BSP layer directory to `conf/bblayers.conf`

```
$ bitbake-layers add-layer ../meta-raspberrypi
```

- Set the target device in `conf/local.conf`

```
MACHINE = "raspberrypi3"
```

- Create the image

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

- Copy the image to an SD card (depending on card reader)

```
$ umount /dev/mmcblk0p* # unmount the SD, VERY IMPORTANT
$ sudo dd if=<path>/core-image-minimal-raspberrypi3.wic of=/dev/mmcblk0 bs=1M
# Faster with bmaptool !
$ sudo bmaptool copy <path>/core-image-minimal-raspberrypi3.wic.bz2 /dev/mmcblk0
```

`bitbake-layers` options

- Display layers list
  `show-layers`
- Create a new layer
  `create-layer <layer-path>`
- Add an existing layer to the list
  `add-layer <layer-path>`
- Remove a layer from the list
  `remove-layer <layer-name>`
- Display available recipes
  `show-recipes`
- Display ‘appended’ files (`.bbappend`)
  `show-appends`

- Yocto BSP is based on the NXP community project
- Machine definition in board dependent layers
- Use the “repo“ tool (Google)

```
$ repo init -u https://github.com/boundarydevices/boundary-bsp-platform -b dunfell
$ repo sync
$ MACHINE=<board-name> DISTRO=boundary-wayland source setup-environment <build-dir>
$ bitbake <image-name> # such as boundary-image-multimedia-full
```

- Create the Micro-SD

```
$ cd tmp/deploy/images/<board-name>
$ umount /dev/mmcblk0p* # unmount the SD, VERY IMPORTANT
$ sudo dd if=boundary-image-multimedia-full-<MACHINE>.wic of=/dev/mmcblk0 bs=1M
```

Or use `bmaptool` !

- You can reload the build environment with:

```
$ source setup-environment <build-dir>
```

Watch out !

- The “build” directory includes :
  - build outputs (packages, etc. in `tmp` directory)
  - layers “cache” directory (`cache`)
  - build “cache” directory (`sstate-cache`)
- Several build directories can live together in the same Yocto source tree
- A layer should not be in the build directory !

Generated directories

- The `build/conf` directory is dynamically generated and contains `local.conf` and `bblayers.conf`
- Source code is stored in the `downloads` directory during the “fetch” step (can be shared between build directories)
- All output files are in the `tmp` directory (!)
- The `tmp/deploy` directory contains the final build artifacts

```
$ ls -1 deploy
images
ipk
licenses
rpm
deb
```

images ← Kernel and generated archives (root-fs)

ipk ← IPK packages

licenses ←

```
$ ls -1 linux-yocto
COPYING
generic_GPLv2
```

rpm ← RPM packages

deb ← DEB packages

Generated directories

- Files created in `tmp/deploy/images/<machine-name>`:
  - OS image (`.sdcard`, `.wic`, etc.)
  - Linux kernel
  - root-filesystem image(s) (`tar.bz2`, `ext4`, etc.)
  - Bootloader image (if required)
  - DT files
  - etc.
- The files are timestamped → 20200124131018
- A symbolic link points to the current file

- The tmp/work directory provides other pieces of information:
  - Root-filesystem content
  - Build directory of each recipe
  - Logs for every build steps → log.do_X.<pid>
- Eats x 10 Gb against 500 Mb for deploy !

```
$ ls -1 work/raspberrypi-poky-linux-gnueabi/core-image-minimal/1.0-r0/
installed_pkgs.txt
...
rootfs
temp
```

installed packages (see log.do_rootfs)

```
bin
boot
dev
etc
...
```

```
log.do_populate_lic
log.do_populate_lic.13363
log.do_rootfs
log.do_rootfs.13927
log.do_rootfs.16374
...
```

Configuration

Setting BitBake variables

- Yocto variables use *only* “strings” !
- Variables are evaluated *when used* (referencing)

  `A = "`**`aval`**`"`

  `B = "pre`**`${A}`**`post" → pre`**`aval`**`post`

  `A = "`**`change`**`"`

  → B is now set to `pre`**`change`**`post`
- Environment variables operators

```
VARIABLE = "value"
VARIABLE += "after_with_space"
VARIABLE =+ "before_with_space"
VARIABLE .= "after_without_space"
VARIABLE =. "before_without_space"
```

- Listed operators takes *immediate effect* during parsing

Using append, prepend, remove

- You can also “append” and “prepend” a variable's value
- It’s called “override-style syntax”

```
VARIABLE:append = "after_without_space"
VARIABLE:prepend = "before_without_space"
```

“Effects are applied at variable expansion time rather than being immediately applied (provide “guaranteed” operations)” (Yocto documentation)

- You can also “remove” a part of the variable’s value

```
VARIABLE = "goodbye cruel world"
VARIABLE:remove = "cruel"
```

Override-style syntax update

- The overrides syntax has evolved over Yocto versions
- The evolution concerns the separator character of the variable and the directive
- Until versions 3.0 (Zeus) only the character '_' is used as separator

```
VARIABLE_append = "after_without_space"
```

- For versions 3.1 (Dunfell) 3.2 (Gatesgarth) 3.3 (Hardknott) it is possible to use either the character '_' or the character ':' as separator

```
VARIABLE_append = "after_without_space"
VARIABLE:append = "after_without_space"
```

- From version 3.4 (Honister) only the character ':' is used as separator

```
VARIABLE:append = "after_without_space"
```

- Setting a default value with `?=` or `??=`
  - `?=` set a variable if it is undefined *when statement is parsed* → “softer assignment” (immediate assignment → only *first* assignment is used)

    ```
    A = "before"
    A ?= "change"
    ```

    → A is set to `before`
  - `??=` assignment is done at the end of the parsing process → “weaker assignment” (*last* assignment used)

    ```
    A ??= "somevalue"
    A ??= "someothervalue"
    ```

    → A is set to `someothervalue` if not set before

    → When multiple `??=` assignments exist, the *last one* is used

Immediate variable expansion with :=

- Variable's content is expanded *immediately*, rather than *when used* (no referencing)

```
T = "123"
A := "${B} ${A} test ${T}"
B = "${T} bval"
T = "456"
C = "cval"
C := "${C}append"
```

B is undefined !

- Finally
  - `A` contains `test 123` because B and A at that time of parsing are undefined !
  - `B` contains `456 bval` (referencing)
  - `C` contains `cvalappend`
- Not commonly used (except for a `.bbappend`)

- Yocto supports RPM, IPK or DEB formats
  - RPM = used by Red Hat and its derivatives
  - DEB = Debian / Ubuntu
  - IPK = optimized for embedded software (simplified DEB, OE)
- Yocto defaults to RPM

```
# We default to rpm:
PACKAGE_CLASSES ?= "package_rpm"
```

- IPK is a “lightweight” format !
  - 268 Mo for the RPM image (Raspberry Pi)
  - 68 Mo for the IPK image
- RPM offers more possibilities for developers (dependencies)
- Most of the time, packages database is not included in the final product !

Adding packages

- We can add packages in an image recipe file

```
IMAGE_INSTALL += "pkg_1 pkg_2"
```

- We can use `conf/local.conf` with a different syntax

```
IMAGE_INSTALL:append = " pkg_1 pkg_2"
```

Don’t forget the “space” !

- Don’t NOT use += in `conf/local.conf`

“Using IMAGE_INSTALL with the += BitBake operator within the /conf/local.conf file or from within an image recipe is not recommended. Use of this operator in these ways can cause ordering issues.” (Yocto documentation)

- A “feature” depends on a set of packages
- The list of features is described in `core-image.bbclass`
  - package-management
  - tools-debug
  - debug-tweaks
  - nfs-server
  - empty-root-password
  - ...
- Use the following syntax in `local.conf`

```
EXTRA_IMAGE_FEATURES = "package-management"
```

- Use the following syntax in a recipe file

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

Some useful options (`local.conf`)

- Deletion of temporary workspace (`work`)

  → To be used during first build of an image

  ```
  INHERIT += "rm_work"
  ```

- Don’t remove temporary workspace for recipes

  ```
  RM_WORK_EXCLUDE += "busybox glibc"
  ```

- Add free space to the root-filesystem (in KB)

  ```
  IMAGE_ROOTFS_EXTRA_SPACE = "50000"
  ```

- Enable virtual UART (specific option for Pi 3)

  ```
  ENABLE_UART = "1"
  ```

- Add a root-filesystem format

  ```
  IMAGE_FSTYPES += "cpio.gz"
  ```

- Build a read-only root-filesystem

  ```
  EXTRA_IMAGE_FEATURES += "read-only-rootfs"
  ```

About `local.conf`

- The `local.conf` file defines:
  - The target device with `MACHINE` variable
  - Compilation or build options (`BB_NUMBER_THREADS`)
  - Target specific options
- It can be used to temporarily add:
  - packages
  - features

    → to be defined later in a dedicated image recipe
- It should not be managed by Git because it’s mostly generated !

Creating recipes

What is a recipe ?

- A directory containing – at least - a recipe file (`.bb` or `.bbappend`)

```
mypack-auto
└── mypack-auto_1.0.bb
```

- The directory could contain a sub-directory with additional files

```
mypack-msg
├── files
│   └── message.h
└── mypack-msg_1.0.bb
```

- The sub-directory name could be:
  - The recipe name
  - The name `files`

- The recipe file defines:
  - BitBake variables (license, sources URI, etc.)
  - BitBake build steps (`do_compile()`, `do_install()`, etc.)
- Source format
  - Use a `Makefile` (BusyBox, etc.)
  - Use a build system such as Autotools or CMake (much easier thanks to the “class” feature)
  - Kernel module sources (use the “module” class)
- Sources URI (`SRC_URI`)
  - remote archive files (http://, etc.)
  - Git repository (git://, https://)
  - “local” source files (rarely)

    `meta/recipes-devtools/makedevs`

- `SRC_URI` is the list of needed files to build the recipe

```
SRC_URI = "http://mysource.com/mypack-1.0.tar.gz;name=<name> \
          file://my_file.h ... \
          "
```

"local" file

- You must provide a checksum for the external archive(s)

```
SRC_URI[<name>.md5sum] = "md5-value"
SRC_URI[<name>.sha256sum] = "sha256-value"
```

- The name is not needed in case of a single archive

```
SRC_URI[md5sum] = "md5-value"
SRC_URI[sha256sum] = "sha256-value"
```

- Defining the license (GPL ?) is mandatory !

```
LICENSE = "<license-name>"
LIC_FILES_CHKSUM = "<license-checksum>"
```

Some additional variables

- PN = Package Name
- PV = Package Version, default to “1.0”
- PR = Package Revision, default to “r0”
- WORKDIR = package's build directory

  tmp/work/*/${PN}/${PV}-${PR}
- S = extracted source code directory
- D = install directory before packaging
- SUMMARY = short recipe description ≤ 72 characters
- Linux variables
  - base_bindir = /bin
  - base_libdir = /lib
  - etc.

Adding new functions (advanced)

- You can use the “:append” or “:prepend” operators

```
do_deploy:append()
do_deploy:prepend()
```

- You can also define shell or Python functions

```
do_sometask() {
   <shell code>
}
python do_sometask() {
   <Python code>
}
```

- In the last case you should define the execution order

```
addtask <sometask> (before|after) <other-task>

do_copy_modules() {...}
addtask copy_modules before do_configure
```

don’t use the “do_” prefix !

- The recipes must be in a layer !
- We create the `meta-training` directory which includes our training recipes

```
meta-training
├── conf
│   └── layer.conf
└── recipes-core
    ├── mypack-auto
    │   └── mypack-auto_1.0.bb
    └── mypack-gen
        └── mypack-gen_1.0.bb
```

- The `conf/layer.conf` file must define the layer with `BBPATH`, `BBFILES`, etc. variables in order to make it visible to BitBake
- Create/add/remove layer with `bitbake-layers`

- The recipes files (`.bb`, `.bbappend`) must be in a sub-directory

  `BBFILES += "${LAYERDIR}/`**`recipes-*/*`**`/*.bb`
- The `recipes-*` directory should match recipes categories
  - `core` for user space
  - `kernel` for kernel space
  - `graphics` for graphical utilities
  - `bsp` for bootloader and firmware
  - etc.

PW 1

First example (local sources)

```
DESCRIPTION = "HelloWorld"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=8ca43cbc842c2336e835926c2166c28b"
SRC_URI = "file://helloworld.c file://COPYING"

# No archive to uncompress, copy the source files in the working directory →
tmp/work/*/${PN}/${PV}-${PR}
S = "${WORKDIR}"

# No Makefile → you should define the compile + install build steps
do_compile() {
     ${CC} ${CFLAGS} ${LDFLAGS} helloworld.c -o helloworld
}

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

- Building

```
$ bitbake mypack-hello
```

- IPK package content

```
$ dpkg -c tmp/deploy/ipk/armv7vet2hf-neon/mypack-hello_1.0-r0_armv7vet2hf-neon.ipk
drwxrwxrwx root/root         0 2018-07-11 08:32 ./
drwxr-xr-x root/root         0 2018-07-11 08:32 ./usr/
drwxr-xr-x root/root         0 2018-07-11 08:32 ./usr/bin/
-rwxr-xr-x root/root      5532 2018-07-11 08:32 ./usr/bin/helloworld
```

→ other packages (`-dev`, `-dbg`, `-src`) are generated

- Installation :

```
# opkg install <path>/mypack-hello_1.0-r0_armv7vet2hf-neon.ipk
# opkg install <URL>
```

- RPM package content

```
$ rpm -qpl <path>/mypack-hello_1.0-r0_armv7vet2hf-neon.rpm
```

- Installation

```
# rpm -ivh <path>/pack-hello-1.0-r0.armv7vet2hf-neon.rpm
```

IPK format based server

- Embedded alternative to RPM or DEB format
- Footprint is highly reduced
- Use `bitbake package-index` to create `Packages.gz`
- Give access to `tmp/deploy/ipk` (HTTP)
- Configuration in `local.conf`

```
PACKAGE_CLASSES = "package_ipk"
IMAGE_ROOTFS_EXTRA_SPACE = "10000"
```

- Configure `opkg.conf` on the target device

```
src/gz   all <URL>/all
src/gz   armv7vet2hf-neon <URL>/armv7vet2hf-neon
src/gz   qemuarm <URL>/qemuarm
```

- Update the target database

```
# opkg update
# opkg install <package>
```

Generic recipe example

- The source archive includes the following files:

```
COPYING
hello_gen.c
Makefile
```

- BitBake automatically executes the `make` command
- The `Makefile` must contain an “install” target

```
install:
   mkdir -p $(DESTDIR)/usr/bin
   cp hello_gen $(DESTDIR)/usr/bin
```

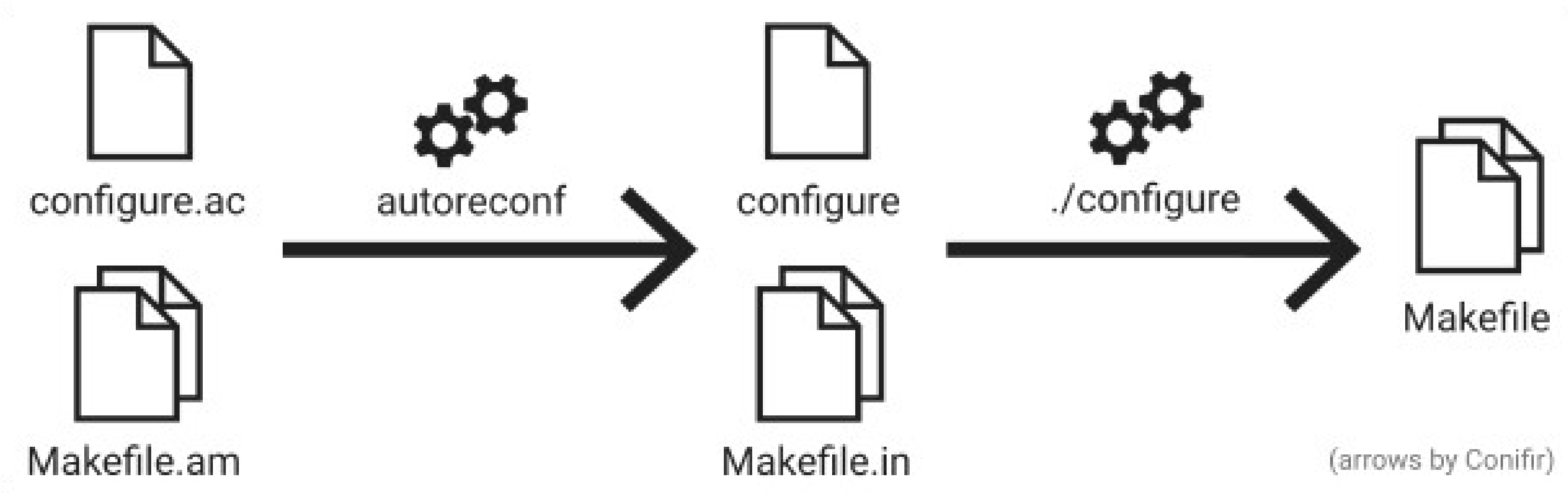
- Using Autotools or CMake is recommended for portability
- Build the package

```
$ bitbake mypack-gen
```

Generic recipe example

```
DESCRIPTION = "Helloworld software (generic)"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://COPYING;md5=8ca43cbc842c2336e835926c2166c28b"

SRC_URI = "http://pficheux.free.fr/pub/tmp/mypack-gen-1.0.tar.gz"

do_install() {
        oe_runmake install DESTDIR=${D}
}

SRC_URI[md5sum] = "2421f06a3ea5c9c35ac1a833f4587499"
```

PW 3


configure.ac
Makefile.am
autoreconf
configure
Makefile.in
./configure
Makefile
(arrows by Conifir)

- GNU project standard (1991)
- Autotools simplifies the compilation since the `Makefile` is generated by the `configure` script
- Based on several commands (`aclocal`, `autoheader`, `autoconf`, `automake`, `libtoolize`, etc.)
- The `configure` script could be provided with sources
- It can be generated from `Makefile.am` and `configure.ac` files

```
$ autoreconf -f -i -v
$ mkdir build && cd build
$ ../configure [--host=<target-sdk>] [--prefix=<install-dir>]
$ make
$ make install [DESTDIR=<root-path>]
```

Cross compiler

/usr/local → /usr

Autotools standard!

Target root-fs

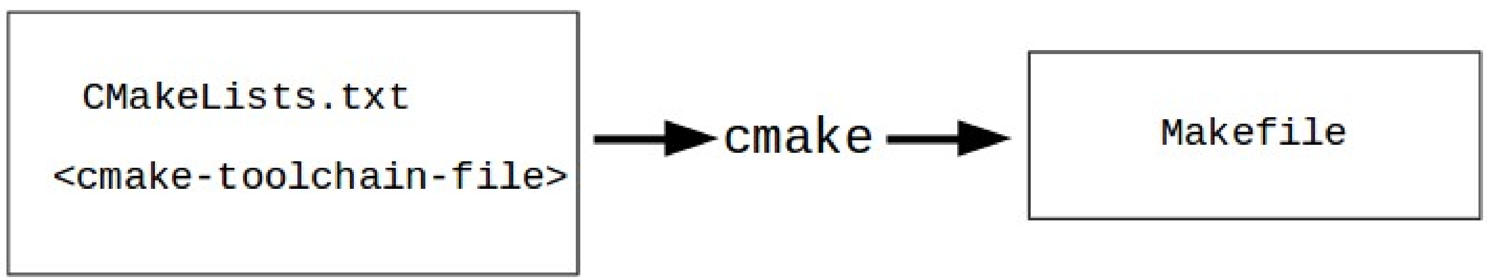
- `EXTRA_OECONF` can be used to pass options to `configure`

PW 3

```
DESCRIPTION = "Helloworld software (autotools)"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://COPYING;md5=8ca43cbc842c2336e835926c2166c28b"

SRC_URI = "http://pficheux.free.fr/pub/tmp/mypack-auto-1.0.tar.gz"

inherit autotools

SRC_URI[md5sum] = "b282082e4e5cc8634b7c6caa822ce440"
```

PW 3

CMake



- CMake is an alternative to Autotools (2000)
- Based on the `cmake` command and 2 configuration files
  - `CMakeLists.txt` → project files
  - `toolchain.cmake` → toolchain definition
- Native compilation (x86)

```
$ mkdir b_x86 && cd b_x86
$ cmake ..
$ make
$ sudo make install
```

- Cross compilation

You can also define the CC variable

```
$ mkdir b_arm && cd b_arm
$ cmake ..
[-DCMAKE_TOOLCHAIN_FILE=<cmake-toolchain-file-path>]
-DCMAKE_INSTALL_PREFIX=<install-dir>
$ make
$ make install [DESTDIR=<root-path>]
```

PW 3

```
DESCRIPTION = "Helloworld software (cmake)"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://COPYING;md5=8ca43cbc842c2336e835926c2166c28b"

SRC_URI = "http://pficheux.free.fr/pub/tmp/mypack-cmake-1.0.tar.gz"

inherit cmake

SRC_URI[md5sum] = "70e89c6e3bff196b4634aeb5870ddb61"
```

PW 3

- Use SRCREV to define the “commit” to use

```
SRCREV = "09906a2b36bd9a12292b23c07cee5741f9c3af86"
```

- SRCPV returns the version string of the current package

```
PV = "0.1+git${SRCPV}"
```

- Define the SRC_URI variable

```
SRC_URI = "git://github.com/pficheux/yocto-test-apps.git;protocol=https"
S = "${WORKDIR}/git"
```

- `DEPENDS` et `RDEPENDS` keywords are used to define dependencies
- The `DEPENDS` keyword indicates a *build-time* dependency
- The following line means that the *recipes* rec_1 and rec_2 are needed at build time

```
DEPENDS = "rec_1 rec_2"
```

- A common case is a `.so` library required to build an executable
- The `RDEPENDS` keyword indicates a *runtime* dependency between *packages* (not *recipes*)
- The following line means that the *package* “pkg_1” is needed at runtime

```
RDEPENDS:${PN} += "pkg_1"
```

→ see “flex” and “m4” recipes

- Basic examples based on mypack-auto recipe
  - bbexample-lib recipe uses Autotools
  - mypack-auto-lib recipe uses Autotools
- The executable requires `libbbexample.so.1`
- Thanks to Autotools, recipe syntax is similar to an executable one (use `inherit`)
- Please note the produced package is `libbbexample*.ipk`

- You can have several versions of a recipe in several layers
- The layer priority defines the version to build
- In case of same priority, the higher version of the recipe is built
- The layer priority is defined in `layer.conf`

```
BBFILE_PRIORITY_training = "6"
```

- We can force a specific version in `local.conf`

```
PREFERRED_VERSION_mypack-hello = "1.0"
```

”Appending” a recipe :-)

- Updating an existing recipe (without updating the recipe file) → use a `.bbappend` file *in another layer*
- Updating the logo used in the `meta/recipes-core/psplash` recipe

```
meta/recipes-core/psplash/
├── files
│   ├── psplash-init
│   └── psplash-poky-img.h
└── psplash_git.bb
```

OE logo

- In `meta-poky/recipes-core/psplash` we have:

```
meta-poky/recipes-core/psplash
├── files
│   └── psplash-poky-img.h
└── psplash_git.bbappend
```

Yocto logo

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

Use cases for appending

- Customizing `/etc/network/interfaces` → `init-ifupdown`
- Enabling I2C in `config.txt` → `rpi_config_git`

```
do_deploy:append() {
    # Enable i2c by default
    echo "dtparam=i2c_arm=on" >> ${DEPLOYDIR}/bootfiles/config.txt
}
```

- Integrating patches
- Automatic loading of kernel modules (create a `.bbappend` for the kernel or the module recipe)
- Customizing the kernel, BusyBox or U-Boot configuration
- Appended recipes can be listed with the `show-appends` option
- The layer priority is used for the appending order !
- Update the PR variable to “r1” (for PW convenience)

PW 7 & 8

Using a patch

- Patches are located in the local directory
- `.patch` files are applied following the “ASCII” order
- Patches must be added to the `SRC_URI` variable

```
SRC_URI += "file://my.patch"
SRC_URI:append = " file://my.patch"
```

Configuration “fragment”

- A configuration “snippet” (`.config`) in a fragment file (`.cfg`)

```
# Use /proc/config.gz
CONFIG_IKCONFIG=y
CONFIG_IKCONFIG_PROC=y
```

- The fragment is “merged” with the configuration file
- Works with the Linux kernel, BusyBox, U-Boot, SWUpdate
- May not work given the recipe syntax
  - OK for “standard” kernel (using `defconfig`)
  - KO for custom Raspberry Pi kernel
- Usable in a `.bbappend`

```
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI += "file://my_fragment.cfg"
```

Linux kernel and modules

- The Linux kernel is defined as a standard recipe !
- Commonly defined in `<bsp-layer>/recipes-kernel/linux`
- The kernel recipe name is `virtual/kernel`
- The kernel can be provided by several “providers” and/or use several versions → use of `PREFERRED` directives

```
PREFERRED_VERSION_linux-raspberrypi ??= "5.4.%"
PREFERRED_PROVIDER_virtual/kernel ?= "linux-raspberrypi"
```

- Kernel provider/version can be redefined in `local.conf`

```
PREFERRED_PROVIDER_virtual/kernel = "linux-raspberrypi-rt"
```

- Not a kernel specific option

```
PREFERRED_PROVIDER_u-boot = "u-boot_rpi"
```

- Use `COMPATIBLE_MACHINE` variable to define the target list

```
COMPATIBLE_MACHINE = "^rpi$"
```

- Building kernel packages

```
$ bitbake virtual/kernel
```

- Modifying the kernel configuration (`.config`)

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

→ updated `.config` is not saved in the recipe !

- To validate the modifications, save it as the `defconfig` file in the recipe (or use a fragment)
- Creating the `defconfig` file from `.config`

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

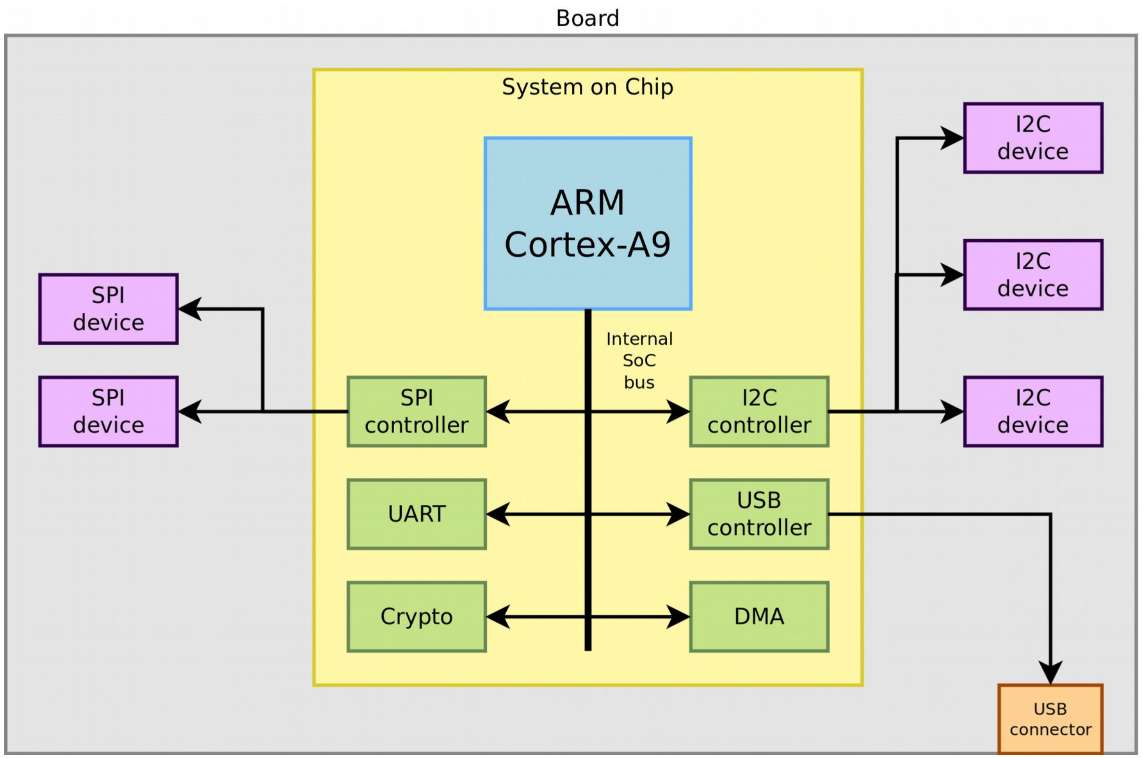
- Creating the `.config` file (without compiling)

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

- Compiling the kernel

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

- ARM platform is difficult to maintain
  - CPU by ARM (ARMvx)
  - SoC (CPU + controllers – SPI, I2C, UART, etc.)
  - Board/module itself (with additional devices)
- Before the DT, the kernel contained the HW description



Open Firmware / DT history

- Sun Microsystems - Open Boot / Open Firmware (1988)
- Used on SPARC for the hardware description
- IEEE-1275 standard
- Apple used Open Firmware on Power Mac 7200 (1995)
- Common Hardware Reference Platform (CHRP)
- Device tree specifications 0.4 released in June 2022
- OS independent

- What Linus Torvalds thought about ARM Linux

  *“Gaah. Guys, this whole ARM thing is a f*cking pain in the ass.”*
- Since 4.x, the kernel needs hardware description in DT
- Device tree binary (dtb) is compiled from the “dts(i)” sources with the “dtc” utility
- Support for “overlays” (dtbo) to update / defines hardware resources
- A new (not that simple) syntax
- Linux bindings in

  `Documentation/devicetree/bindings`
- DT sources (dts or dtsi) in `arch/arm/boot/dts`
- Includes with `#include` directive

  `#include "am33xx.dtsi"`

- The DT files are compiled with the kernel
- Compile DT files with:

```
$ make dtbs
```

- Compile a single DT file (dts):

```
dtc -I dts -O dtb -o <dtb-file> <dts-file>
```

- Un-compile a binary (dtb)

```
dtc -I dtb -O dts <dtb-file>
```

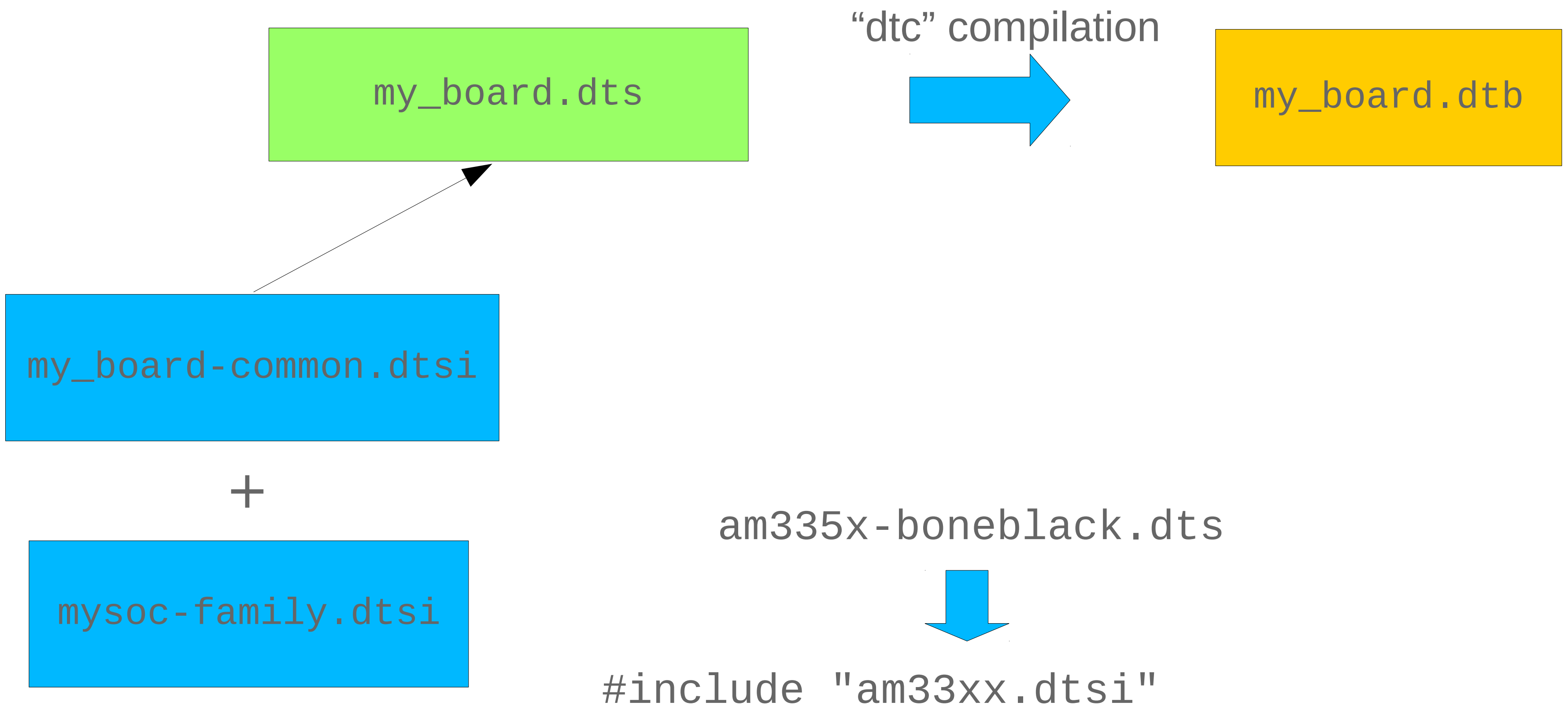
- Bootloader (U-Boot) should support DT

```
# bootz <kernel-addr> - <dtb-addr>
```

- Add the DT part to static kernel with `CONFIG_ARM_APPENDED_DTB` option

```
$ cat zImage my.dtb > zImage_dtb
```

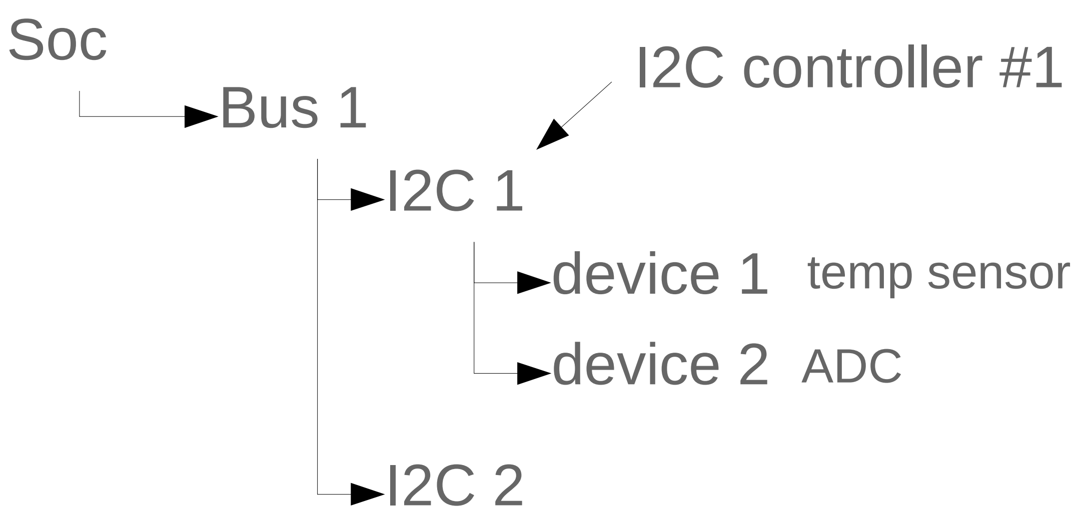
Producing the DT




```
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
#include "am335x-boneblack-common.dtsi"
```

- Raspberry Pi / Raspbian uses `/boot` and `/boot/overlays`
- BBB / Debian uses `/boot/dts` and `/lib/firmware`
- Yocto uses `/boot/devicetree` for added dtb / dtbo
- Several ways to load a DT
  - “configfs” → `/sys/kernel/config`
  - U-Boot overlays (`uEnv.txt`)
  - Raspberry Pi (`dtoverlay` + `config.txt`)
  - BBB (CAPE Manager then U-Boot overlays)

- DT is a node tree (close to the HW structure)
- Each node describes a part of hardware
- Each node includes “properties”
- The syntax is close to JSON



Syntax summary

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        // hex is implied in byte arrays. no '0x' prefix is required
        a-byte-data-property = [01 23 34 56];

        child-node@0 {
            second-child-property = <1>;
            a-string-property = "Hello, world";
            // A "phandle" to the label "node2"
            a-reference = <&node2>
        };

     child-node@1 {
        };
    };

    // Label
    node2: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>; /* each number (cell) is a uint32 */
        child-node@ {
        };
    };
};
```

- A node includes
  - One or more properties
  - Zero or more children
- The “unit address” is useful to define two identical components with different addresses
- A “label” is a reference (or “phandle”) to another node
- Device tree structure is available from:

```
/proc/device-tree
/sys/firmware/devicetree/base
```

Some useful properties

- “compatible” defines the association with the driver

```
compatible = "my-driver"; /* Use "my-driver" driver */
```

- “status” defines the hardware status (enabled or disabled)

```
status = "okay";
```

- “exclusive-use” is useful for resource reservation

```
exclusive-use = "P9.24", "P9.26"; //UART tx/rx for BBB
```

Reserved nodes

- “cpus” defines a CPU list

```
cpus {
  cpu@0 {
    compatible = "arm,cortex-a9";
  };
};
```

- “memory” for mapped/non-mapped memory

```
#address-cells = <0x2>,
#size-cells = <0x1>;
memory {
   reg = <0x90000000 00000000 0x800000>; // 2x u32 for addr, 1 u32 for size
};
```

- “aliases” defines a link to an alias node

```
aliases { eth0 = &fec; } ;
```

- “chosen” extends kernel boot parameters (bootargs)

```
chosen {
   bootargs = "console=ttyAMA0,115200";
};
```

- Using overlay(s) is another way to configure DT
- A `.dtbo` file is loaded to “update” some parts of DT
- As an example, we can enable/disable I2C with the following code:

```
/dts-v1/;
/plugin/;
/ {
  compatible = "arm,versatile-pb";
  fragment@0 {
    target-path = "/i2c@10002000";
    __overlay__ {
      status = "disabled"; // use "okay" for enabling
    };
  };
};
```

- Use cases of DT update/creation
  - Creating a custom board from evaluation board
  - Adding new HW (sensor ?)
- Most of the time updating a `.dts` needs a kernel patch (`.bbappend`) as `.dts` files are provided with kernel sources
- The `KERNEL_DEVICETREE` variable is used to define the dtb (in the kernel recipe)

```
KERNEL_DEVICETREE = "<dtb-name>"
```

- We can use the “devicetree” class for a device tree overlay recipe (see the PW)

- The kernel source tree is necessary to build a module !
- It is located in `tmp/work-shared/qemuarm`

```
$ ls -1 tmp/work-shared/qemuarm/
kernel-build-artifacts
kernel-source
```

used for `KERNEL_SRC` variable in `Makefile`

- Build a module with

```
$ bitbake make-mod-scripts
$ make KERNEL_SRC=<path>/kernel-build-artifacts
```

build the “kernel artifacts”

- This method needs Yocto to be installed on the machine
- Two other ways:
  - Integrate the kernel sources in the Yocto SDK (see further)
  - Use the kernel sources from the kernel Git repository

Kernel module recipe

- Use the “module” class (`module.bbclass`)
- The “hello-mod” recipe generates the following packages

```
hello-mod-*.ipk
kernel-module-hello-*.ipk
```

- To auto-load a module during boot

```
KERNEL_MODULE_AUTOLOAD += "hello"
→ # cat /etc/modules-load.d/hello.conf
  hello
```

- To pass parameters to the module

```
KERNEL_MODULE_PROBECONF = "hello"
module_conf_hello = "options hello param=42"
→ # cat /etc/modprobe.d/hello.conf
  options hello param=42
```

→ add the lines to the recipe file (or a kernel `.bbappend`)

Images

Special image

- Images inherit the "core-image" class `meta/classes/core-image.bbclass`
- "core-image" inherits the "image" class `meta/classes/image.bbclass`
- The `core-image.bbclass` file lists all of the available "features" (`IMAGE_FEATURES` variable)
- A few image examples are provided in `meta/recipes-core/images`
  - core-image-minimal = the simpler one (no PM / kernel modules)
  - core-image-base = basic image (from core-image class)
  - etc.
- We use a "light" image and we add packages and features
- Image will be tested with NFS-Root !

Testing NFS-Root (server)

- The root-filesystem is installed on the PC
- NFS configuration in `/etc/exports` file

```
/home/stage/rootfs_yocto *(rw,no_root_squash,sync)
```

Keep “root” access rights

- The “no_root_squash” option MUST NOT be used except for NFS-Root !
- Restarting NFS server (or loading the new configuration)

```
$ sudo service nfs-kernel-server restart
```

OR

```
$ sudo exportfs -a
```

Testing NFS-Root (client)

- We should update the kernel boot parameters (Pi example)

```
console=ttyAMA0,115200 root=/dev/nfs
rootfstype=nfs nfsroot=192.168.2.1:/home/stage/rootfs_yocto,tcp,v3
ip=dhcp rw
```

- Same configuration for QEMU/ARM (static address)

```
console=ttyAMA0 mem=256M
root=/dev/nfs
nfsroot=192.168.7.1:/home/stage/rootfs_yocto,nfsvers=3,port=3049,udp,mount
port=3048 rw  ip=192.168.7.2::192.168.7.1:255.255.255.0
```

- Most of the time, the options are defined in the bootloader configuration (U-Boot)
- Please read the kernel documentation

`Documentation/filesystems/nfs/nfsroot.txt`

(Simple) image customization

- The is to create a final image with some additional packages
- Add development options (features ?) to `local.conf`
- Based on `core-image-minimal`:
  - Required packages (applications, drivers, etc.) using `IMAGE_INSTALL`
  - Required “features” using `IMAGE_FEATURES`
- You should add an image recipe to `recipe-core/images`

```
# Base this image on core-image-minimal
include recipes-core/images/core-image-minimal.bb
# Add example packages to the rootfs
IMAGE_INSTALL += "mypack-gen mypack-auto ..."
# Add PM ? (not needed for production image)
# IMAGE_FEATURES += "package-management"
```

PW 13

Using a “package group”

- A list of packages that defines the image dependencies
- Groups are defined in
  `meta/recipes-core/packagegroups`
- “core-image-minimal” dependency example
  `IMAGE_INSTALL = "packagegroup-core-boot ... "`
- You can create your own package groups in external layers !

Creating a custom “package group”

- Define a group (a recipe) that lists the required packages

```
inherit packagegroup
RDEPENDS:${PN} = "<list-of-pkgs>"
```

- To build an image using this package group, you need to add it with `IMAGE_INSTALL`

```
IMAGE_INSTALL += "<packagegroup-name>"
```

Customizing the distribution (aka "distro")

- The “distro” is defined by the `DISTRO` variable in `conf/local.conf`

```
DISTRO = "my-distro"
```

- Defined by the `conf/distro/my-distro.conf` file in an external layer

```
require conf/distro/poky.conf
DISTRO = "my-distro"
DISTRO_NAME = "My Yocto distribution"
DISTRO_VERSION = "1.0"
...
```

- Most of the time, the distro is based on “poky” (but not mandatory)

- Add a “distro” feature with:

```
DISTRO_FEATURES:append = " systemd"
```

- Remove a “distro” feature with:

```
DISTRO_FEATURES:remove = "ptest"
```

- Usable in `my-distro.conf` or `local.conf`

Adding a SysvInit service

- Add a SysvInit script started at boot time
- Use the “update-rc.d” class and define the “runlevel”

```
INITSCRIPT_NAME = "my-service-name"
INITSCRIPT_PARAMS = "defaults 99"
```

- Much simpler than systemd but less powerful
- Recent projects use systemd !
- The PW is included in the final project (Yocto / IoT)

Adding a systemd service

- Systemd is a “distro feature”
- One need to include systemd with the following options in `local.conf`:

```
DISTRO_FEATURES:append = " systemd"
DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit"
VIRTUAL-RUNTIME_init_manager = "systemd"
VIRTUAL-RUNTIME_initscripts = "systemd-compat-units"
```

- Use the “systemd” class in the service recipe

Yocto recipe tools

- We can modify source code with Devshell

```
$ bitbake -c devshell <recipe>
```

- Open an new terminal where you can use standard development tools (`cmake`, `make`), instead of `bitbake`
- Not SCM available (Git)
- May be useful for a quick (and dirty) modification

- Devtool is useful to add / modify / upgrade recipes
- Three main functionalities:
  - Creating a recipe from source code (add + edit-recipe)
  - Modifying an existing recipe (modify)
  - Upgrading the source code version for an existing recipe (upgrade)
- Typical syntax

```
$ devtool <command> <recipe> <parameters>
```

- Very useful to create several patches in a `.bbappend` (for the Linux kernel ?)
- Based on the “externalsrc” class

- Devtool uses a temporary “workspace”
- The default is `workspace` in the “build” directory
- You can create an external one with the “create-workspace” command
- The workspace path is added to `bblayers.conf`
- The sources in `workspace/sources` are managed by Git !
- Created/modified recipe is copied to a layer (with “finish” or “update-recipe” command)
- Devtool creates a patch (and a `.bbappend`) or an updated recipe
- Check the PW for a simple patch (mypack-cmake)

Typical Devtool session

- Start modifying existing recipe

```
$ devtool modify <recipe>
```

- Update source code

```
$ cd <workspace>/sources/<recipe>
$ vi file.c
$ git commit -a -m "updated code"
```

- Build the recipe package (optional)

```
$ devtool build <recipe>
```

- Create a `.bbappend` and finish working on the recipe

```
$ devtool finish <recipe> <layer-path>
```

- Update the original recipe (not recommended ?)

```
$ devtool update-recipe <recipe>
```

- Add a updated recipe (.bbappend) to an alternate layer

```
$ devtool update-recipe -a <layer-path> <recipe>
```

Continuous Integration (CI)

- Modification (upgrade) should not add “regression”
  - Standard components (OS)
  - Added applications (developed with SDK)
- Methods and tools
  - Unit / functional test (per package)
  - Global test (Yocto image)
  - Emulation + test automation
  - Jenkins, LAVA, SonarQube, QEMU, Gitlab, etc.
- Yocto provides “ptest” (unit test) and “testimage” (global test)

Testing packages (ptest)

- The recipe must inherit from the “ptest” class

```
inherit ptest
```

- The recipe must include a `run-ptest` script
- The “distro” must include the “ptest” feature
- Add the following option to image (`local.conf`)

```
EXTRA_IMAGE_FEATURES += "ptest-pkgs"
```

- The image will now include `/usr/lib/<pkg>/ptest`
- List the available tests + start a test with:

```
# ptest-runner -l
# ptest-runner <pkg-name>
```

- Use SSH for automatic testing

```
$ ssh root@<target-IP> ptest-runner
```

- Several recipes use ptest (BusyBox, BlueZ, etc.)

Global testing (testimage)

- Image configuration (`local.conf`)

```
INHERIT += "testimage"
TEST_SUITES = "ping"
# For a real board (not QEMU)
TEST_TARGET = "simpleremote"
TEST_SERVER_IP = "192.168.2.1"
TEST_TARGET_IP = "192.168.2.<board-IP>"
```

- Build, install and boot the new image for the real target
- Test from the PC

```
$ bitbake -c testimage core-image-minimal
RESULTS:
RESULTS - ping.PingTest.test_ping - Testcase 964: PASSED
SUMMARY:
core-image-minimal () - Ran 1 test in 0.032s
core-image-minimal - OK - All required tests passed
```

- In case of QEMU, the target must NOT be started !

Customizing testimage

- Tests located in `meta/lib/oeqa/runtime/cases`
- Add new tests to `<layer-name>/lib/oeqa/runtime/cases`

Software Development Kit (SDK/eSDK)

- The (extensible) SDK generated by Yocto
- The “internal” Yocto compiler is not usable without BitBake
- The SDK is a classical cross toolchain
  - Cross compiler
  - Cross debugger (GDB / GDBSERVER)
  - QEMU emulator (x86, ARM)
  - etc.
- The eSDK adds Devtool
- The reference documentation is “Application Development and the Extensible Software Development Kit (eSDK)”

- Most of the time, the toolchain is produced by Yocto
- The following command creates a “generic” toolchain as an installation script

```
$ bitbake meta-toolchain
```

- Install the toolchain by the following command:

```
$ sudo tmp/deploy/sdk/poky-glibc-x86_64-meta-toolchain-armv7vet2hf-neon-
qemuarm-toolchain-3.1.18.sh
```

- Use of the toolchain as follows:

```
$ . /opt/poky/3.1.18/environment-setup-armv7vet2hf-neon-poky-linux-gnueabi
$ $CC -o hello hello.c
$ file hello
hello: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-,
BuildID[sha1]=46cbeddae55f86b421accc359b6d9aea71c0037f, for GNU/Linux
3.2.0, with debug_info, not stripped
```

Building SDK or eSDK

- Some images include components required at build time (libraries, tools, etc.)
- The “populate_sdk” Bitbake task creates a toolchain including all specific libraries

```
$ bitbake -c populate_sdk training-image
```

- The “populate_sdk_ext” creates eSDK
- Add the following line to `local.conf` to include kernel headers (for kernel development)

```
TOOLCHAIN_TARGET_TASK:append = " kernel-devsrc"
```

- The kernel sources must be configured (as root !)

```
# . /opt/poky/3.1.18/environment-setup-<arch>
# cd /opt/poky/3.1.18/sysroots/<arch>/usr/src/kernel
# make oldconfig && make prepare && make scripts
```

- Building a module

```
$ . /opt/poky/3.1.18/environment-setup-<arch>
$ make KERNEL_SRC=/opt/poky/3.1.18/sysroots/<arch>/usr/src/kernel
```

Using an external toolchain

- Not the common way but used by some BSP (TI)
- We should define some variables in `local.conf`

```
TCMODE = "<toolchain-name>"
EXTERNAL_TOOLCHAIN = "<toolchain-path>"
```

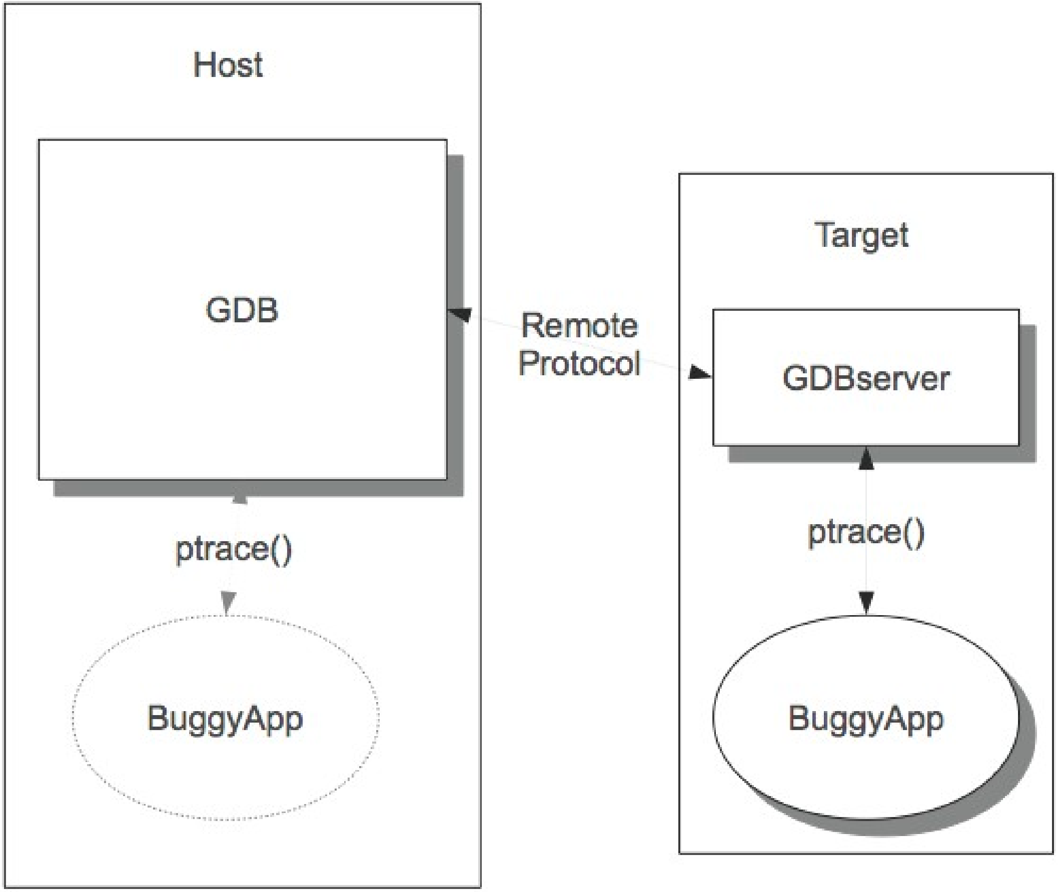
- Toolchain parameters (`TCMODE`) defined in external layer such as meta-arm
- Default value in `meta/conf/distro/defaultsetup.conf`

```
TCMODE ?= "default"
require conf/distro/include/tcmode-${TCMODE}.inc
```

- The toolchain must be supported by the layer !
- We can add toolchains definition in a custom layer

Remote debugging

- Not the favorite task for Yocto but could be useful !
- You must add the “gdbserver” package provided by the “gdb” recipe



- Add `gdbserver` to the image

```
# opkg install gdbserver
```

- Principle
  - Start the application on the target with `gdbserver` and select a free TCP port
  - Start the same application on the host with the cross debugger + use `target-remote`
- Example :

```
root@qemuarm:~# gdbserver :9999 <program-name>
Process myprog created; pid = 12810
```

target

```
$ arm-poky-linux-gnueabi-gdb <program-name>
GNU gdb (GDB) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
...
(gdb) target remote 192.168.7.2:9999
(gdb) continue
```

host (PC/x86)

don't use "run" !!!

PW 21

Adding debugging symbols

- Most of the time, a production image does not include debug symbols
- Yocto provides a way to use a debug version when necessary
- Add the following options to `local.conf`

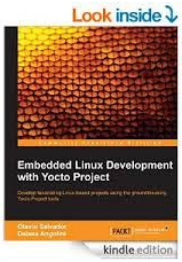
```
IMAGE_GEN_DEBUGFS = "1"
IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
```

- Yocto will create a new image (debug version) such as `core-image-minimal-qemuarm-dbg.tar.bz2`
- Just extract standard + debug images in the same root-fs directory
- Use the Yocto documentation !

- Yocto/OE is a powerful tool but no that simple to handle
- Significant initial investment but smart management of multiple configurations (projects, HW)
- Runtime package management available (not available with Buildroot)
- Yocto can easily generate a SDK, easy to install
- Mostly used in industry
- A quick test is simpler with Buildroot !
- No efficient GUI (but is that possible?)

Bibliography

- http://www.openembedded.org
- https://www.yoctoproject.org
- https://wiki.yoctoproject.org/wiki/Releases
- https://www.yoctoproject.org/documentation ***
- http://www.yoctoproject.org/docs/current/mega-manual/mega-manual.html ***
- http://www.yoctoproject.org/docs/current/bitbake-user-manual/bitbake-user-manual.html ***
- https://wiki.yoctoproject.org/wiki/Technical_FAQ ***
- https://wiki.yoctoproject.org/wiki/Minimal_Image
- https://wiki.yoctoproject.org/wiki/How_do_I
- http://elinux.org/Yocto_Project_Introduction
- http://elinux.org/Bitbake_Cheat_Sheet
- https://www.packtpub.com/application-development/embedded-linux-development-yocto-project
- http://events.linuxfoundation.org/sites/events/files/slides/belloni-yocto-for-manufacturers_0.pdf
- http://www.codeproject.com/Articles/774826/Adding-rd-party-components-to-Yocto-OpenEmbedded-L
- http://www.jumpnowtek.com/yocto/Using-your-build-workstation-as-a-remote-package-repository.html
- https://wiki.openwrt.org/doc/techref/opkg
- “Linux embarqué, mise en place et développement” (french book)
  https://www.eyrolles.com/Informatique/Livre/linux-embarque-9782212674842
- https://wiki.koansoftware.com/index.php/Building_Software_from_an_External_Source



Bibliography

- https://lists.yoctoproject.org/pipermail/yocto/2014-July/020412.html
- White paper “Linux pour l'embarqué” http://www.smile.fr/Ressources/Livres-blancs/Ingenierie
- https://software.intel.com/sites/default/files/m/e/e/8/b/7/42871-10_Develop_2BApplications_2Bon_2BYocto.pdf
- http://www.yoctoproject.org/docs/latest/bsp-guide/bsp-guide.html
- https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/package-manager-white-paper.pdf
- https://www.pearsonhighered.com/program/Streif-Embedded-Linux-Systems-with-the-Yocto-Project/PGM275649.html
- http://www.linuxembedded.fr/2016/05/la-mise-au-point-des-recettes-yocto ***
- https://elinux.org/images/a/a6/2018-ELC-YP%2BKernel-Hudson-reduced.pdf ***
- https://www.packtpub.com/virtualization-and-cloud/embedded-linux-development-using-yocto-project-cookbook-second-edition
- Using Devtool https://www.youtube.com/watch?v=CiD7rB35CRE ***
- A Devtool article by Smile https://linuxembedded.fr/2021/07/methodes-dintegration-de-paquets-yocto
- Arago http://arago-project.org/wiki/index.php/Setting_Up_Build_Environment
- DT introduction https://elinux.org/images/f/f9/Petazzoni-device-tree-dummies_0.pdf
- https://www.blaess.fr/christophe/yocto-lab/sequence-IV-3/index.html#modification-du-device-tree
- Using “testimage” https://wiki.yoctoproject.org/wiki/Image_tests ***
- Using “ptest” https://wiki.yoctoproject.org/wiki/Ptest ***
- Using “externalsrc” https://wiki.koansoftware.com/index.php/Building_Software_from_an_External_Source