Using Yocto Project to build rich and reliable embedded Linux distributions

Christian Charreyre

▶ **To cite this version:**

Christian Charreyre. Using Yocto Project to build rich and reliable embedded Linux distributions. Embedded Real Time Software (ERTS'14), Feb 2014, TOULOUSE, France. ⟨hal-02271300⟩

HAL Id: hal-02271300

https://hal.science/hal-02271300v1

Submitted on 26 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Using Yocto Project to build rich and reliable embedded Linux distributions

Author : Christian Charreyre – CIO Informatique Industrielle – christian.charreyre@cioinfoindus.fr

Abstract

Linux is often used as an embedded Operating System for smart devices with rich features (along with Android). Using such an OS with its quickly evolving ecosystem needs strong tools to insure the embedded distribution will remain reliable in an industrial context. This paper aims to quickly review tools developed by the embedded community to address this challenge, then focusing on Yocto who tends to become the standard, due to its support by many major actors.

Keywords : embedded linux distribution, embedded linux tools, Yocto project, reliability, reproducibility.

Introduction

During the last 15 years, Linux rank in Operating Systems used in embedded market has constantly grown, and it is now a major solution to build embedded devices that are not subject to security constraints or certification process.

10 years ago, Linux could be seen as an alternative to legacy RTOS, like pSOS, VxWorks etc... (as long as hard real time was not mandatory). At that time only the very basic components were used, such at the kernel itself, the C library (glibc) and a few basic tools provided by Busybox, the Swiss Army Knife of Embedded Linux.

Building the complete software of the device under development was mainly developing from scratch the specific application, built over these basic components. The model was very similar to the one with legacy RTOS, with the substitution of legacy OS by Linux. Developers used another operating system but there was no real rupture in the development paradigm.

Nowadays, there is an endless pressure to build devices with much more features than before, including high connectivity through a lot of channels (Wifi, 3G, Bluetooth etc...), rich Man Machine Interfaces, high level graphic features, audio and video performances etc....

Along with this feature level pressure, there is also a need for reduced Time To Market, as product development cycle becomes shorter and shorter, due to competition on innovation.

These evolutions in embedded devices requirements are driven by the generalization of smart devices like smartphones and tablets, and the leadership of iOS and Android in this kind of smart devices. People want to have the same features level they find on their smartphone, on embedded devices in their car, on the machine they use or in the product they interact with.

It is the reason why there are emerging projects for the embedded market built on top of Android, which becomes a challenger for Linux as embedded OS. Nevertheless Android is a good solution for telephony and tablets (it was designed for this segment), but it is not necessary an easy solution when used for other purposes it was not originally designed for.

Thanks to the great Linux ecosystem, it is possible to get high feature level and reduced TTM, and go on building smart devices on a Linux base, but this means that developing on Embedded Linux today is completely different that 10 years ago.

Software teams will have to collect very various components from the Linux ecosystem (libraries, codecs, stacks, frameworks ...), use them as “modular bricks” they will assemble unchanged or locally adapted, and build the final software image of the device. In such a scheme, the number and even the volume of code of components assembled from the ecosystem can be much more important than the software developed from

scratch.

All these software components are driven by distinct organizations, communities, or companies, without anyone that could centrally manage the evolutions, interactions of the components : Linux ecosystem is very dynamic, very innovative, but also very fragmented, and this fragmentation has to be addressed in order to build reliable products.

Managing this complexity is the daily work of Linux distribution suppliers for the desktop. But using a Linux distribution as is for an embedded device is not the solution, unless the embedded device is very similar to a PC.

So the Linux embedded community need tools to help engineers build attractive products that can integrate many components from the Linux ecosystem, without giving up reliability and long term maintenability.

Embedded distribution tools overview

As a very first solution, it is possible to use a Do It Yourself strategy, doing things manually :

- select the software components you need and download them
- configure them
- compile them
- install them in an embedded file system image on the development PC
- condition this embedded file system image in the final deployable format.

The problem quickly tend to be insoluble due to dependencies between software components : the high level components generally rely on a lot of shared libraries, basic services etc …. Finally all needed components build a dependency graph that must be treated in the right order, from basic elements to final ones. The coherence of versions between all the software components adds difficulty to this Do It Yourself process, as each dependency not only is a relationship between 2 components, but between precise version of these 2 components (not necessary the last one).

So Do It Yourself manually, without automation tools, can only be envisaged on a very small number of pieces, i.e. on simple applications.

Over the years, a few dedicated tools to automate the build of an embedded Linux distribution have emerged. All the major solutions share the same philosophy as Do It Yourself strategy : download software components from the Linux ecosystem (upstream projects), if necessary apply local patches, then configure and cross compile them, and finally build the deployable software image. But they use additional informations to solve dependencies problem on a large base.

Such tools are built around an engine for tasks execution, meta data that describe dependencies and actions on components and optional patches. With the exception of patches, these tools don't include source code, but the rules to efficiently use source code from upstream projects.

The main solutions in this area are Buildroot [1], Scratchbox [2], LTIB [3] and Open Embedded [4] / Angstrom [5] / Yocto [6].

Scratchbox seems not being active (last commit in April 2012).

LTIB is the build tool used up to now by Freescale, but Freescale is currently swapping to Yocto, so LTIB will be soon deprecated.

Buildroot is still active, but focused on simpler images than Open Embedded / Angstrom / Yocto (less available components).

Before a closer look on Open Embedded / Angstrom / Yocto, the following table summarizes strong and weak points of the other solutions :

Tool	Pros	Cons
Do It Yourself	Complete control on the build process	Need to solve all dependencies manually Limited to simple configurations (few components)
Buildroot [1]	Simple to use (Makefile based) Active project and community	Intermediate set of available software components (much more possibilities than Do It Yourself, but too limited for smart devices) No partial build nor packages
Scratchbox [2]		Deprecated
LTIB [3]	De facto standard up to now for Freescale hardware Well supported by Freescale regarding BSPs Large set of components available	Old solution, Freescale swaps to Yocto, will be soon deprecated Need old Linux distributions on developer's PC Not used by many hardware providers

Table 1 : other tools comparison

Yocto development environment

These 3 tools are close cousins, Angstrom and Yocto inherit from Open Embedded.

Open Embedded was originally developed to build the Linux Distribution Open Zaurus for Sharp Zaurus (an ARM based PDA). It is now used for all kind of hardware platforms.

Angstrom and Yocto share with Open Embedded the same task engine (Bitbake) and the same kind of meta data, allowing constant exchanges between the 3 environments.

Today Yocto (the real name is Yocto Project, but for convenience we will use Yocto in this document) is hosted by the Linux Foundation, and is supported by major actors like silicon vendors (Texas Instruments, Intel, Freescale)[1], major tools vendors (Wind River, MontaVista, Mentor Graphics)[2], embedded consultants etc.... So the document focuses on Yocto, but the majority of concepts remains applicable for Angstrom and Open Embedded.

To describe Yocto features, we will use the paradigm of a black box, with inputs and outputs (see Figure 1).

Yocto's inputs are :

- Upstream projects source code (projects picked from the Linux ecosystem).
- Local source code (code components developed internally).
- Meta data : recipes + local patches to apply to source code.
- Configuration files : hardware platform, toolchain tunings, preferred components versions, misc parameters.

These input are used by the bitbake task engine to produce a set of main outputs :

- a final deployable image in various formats (tar.bz2, UBI, ext3 / ext4 ….).
- an application development SDK, to allow developments from outside the framework.
- Packages for each built software component (RPM, DEB or IPK format available).

1 This list does not pretend to be exhaustive
2 This list does not pretend to be exhaustive

Other interesting output are the packages and licenses manifest of the final image, as correct license management is an important point when developing a Linux based embedded device. After the image is built, the list of packages used in the image, and the associated licenses, is available.

Inside the black box, the bitbake engine builds a dependencies tree of all individual tasks necessary to perform the asked job, then automate their execution in the right order (don't put the roof if you haven't yet build the walls).

Individual tasks performed by bitbake on a component are :

- fetch the source[3].
- unpack it.
- patch it with local patches.
- configure the software component (autotools appreciated).
- compile it.
- stage it (install headers and libraries in the right location, so that others can use them).
- package it (build RPM or DEB or IPK package).
- deploy it (copy the package in the package feed).

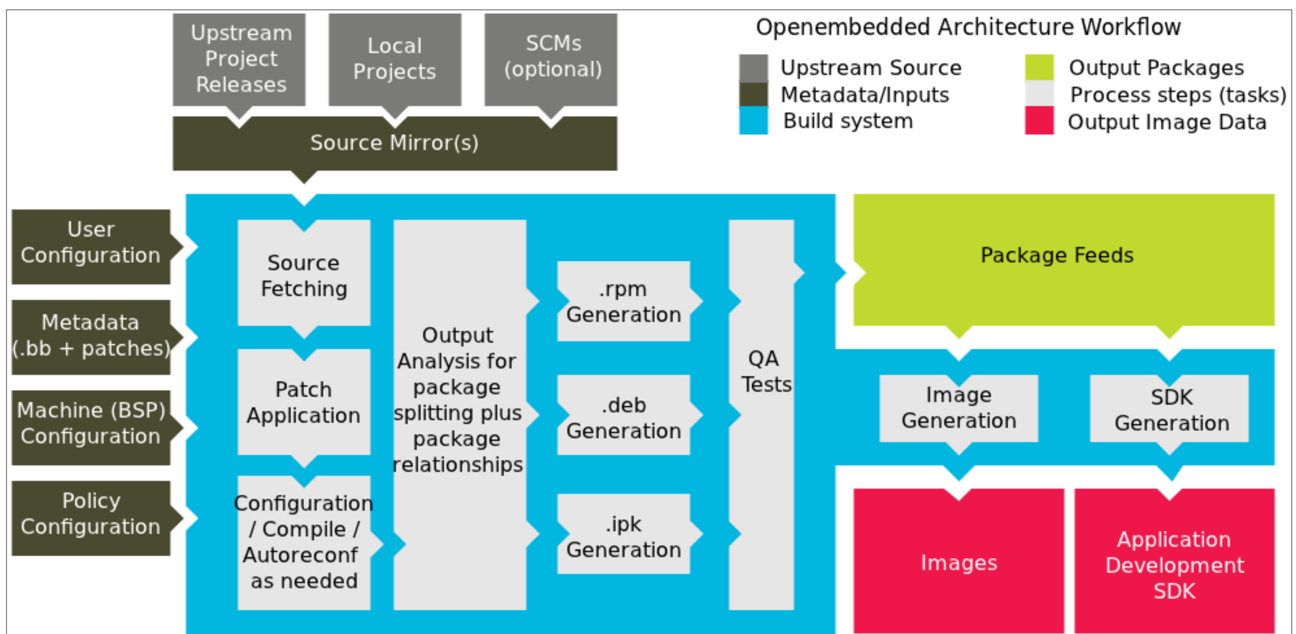

Figure 1 : The Yocto project development environment - Credit : Yocto project

For a final image, the tool will also build the complete root file system in a temporary directory, then build the final deployable image in the chosen format : compressed tar archive, ext3/ext4 image, UBIFS, live image based on initramfs, image for Qemu or VirtualBox, etc....

Thanks to bitbake, meta data and the dependencies tree, it is possible to build a complete QT Embedded demo image with the single command "`bitbake qt4e-demo-image`" (this single command will perform a few thousands of elementary tasks, and require large disk space, and hours of machine computing).

The magic of dependencies tree resides in the recipes that describe the actions to build a software component, but also that manage the complexity of components interactions through dependencies description and components version management (see Figures 2 and 3).

3 Fetched source is stored locally, so that reproducibility is ensured if source disappeared from then Net.

```
DESCRIPTION = "MoNav is a fast navigation system featuring exact routing with
OpenStreetMap data."
HOMEPAGE = "http://code.google.com/p/monav"
SECTION = "x11/applications"
LICENSE = "GPLv3+"
LIC_FILES_CHKSUM = "file://misc/license_template_christian;
md5=d99c9b3bafdde80adee296762376348d"

DEPENDS = "qt-mobility-x11"
PR = "r1"

SRC_URI = "http://monav.googlecode.com/files/${BPN}-${PV}.tar.gz \
           file://monav.png \
           file://monav.desktop"

SRC_URI[md5sum] = "d048ccef8c6a21e8656aa4af3fcb8329"
SRC_URI[sha256sum] =
"5a3bf9e9f7368b81ba8e2f755960082fc42a2e2c78f9de645f99ba293c77ee7f"

inherit qmake2 qt4x11

EXTRA_QMAKEVARS_PRE="CONFIG+="release""
QMAKE_PROFILES="monavclient.pro"

do_install_append() {
    install -d ${D}${datadir}/monav
    install -d ${D}${datadir}/monav/images
    cp -a ${S}/images/* ${D}${datadir}/monav/images

    install -d ${D}${datadir}/icons
    install -m 0644 ${WORKDIR}/monav.png ${D}${datadir}/icons

    install -d ${D}${bindir}
    install -m 0755 ${S}/bin/monav ${D}${bindir}/monav

    install -d ${D}/${datadir}/applications
    install -m 0644 ${WORKDIR}/monav.desktop ${D}/${datadir}/applications
}

FILES_${PN} += "${bindir}/monav ${datadir}/icons/monav.png"
```

Package informations

Explicit dependency

Download adress + local files

Installation directives

Figure 2 : Yocto recipe for a software package

```
DESCRIPTION = "A foundational basic image without support for X that can be \
reasonably used for customization and is suitable for implementations that \
conform to Linux Standard Base (LSB)."

IMAGE_FEATURES += "splash ssh-server-openssh"

IMAGE_INSTALL = "\
    ${CORE_IMAGE_BASE_INSTALL} \
    packagegroup-core-basic \
    packagegroup-core-lsb \
    "

inherit core-image
```

Image informations

Features wanted in the image

Software packages to install

Figure 3 : Yocto recipe for a deployable image

With this kind of distributed database of components versions and interactions (the complete set of recipes), Yocto solves the complexity of assembling a rich embedded distribution in a constantly moving upstream projects ecosystem.

The set of meta data is maintained by the Yocto user's community, who works in order to deliver recipes of components who will correctly assemble as the versions in the meta data set are coherent. The recipes developer's community does the same job as standard PC distributions do at binary packages level.

Modularization of recipes

Yocto provides mechanism to modularize such an amount of software components. The modularization is based on two main ideas : a prioritized layers structure and the possibility to modify recipes without duplication.

Recipes are grouped in layers that overlap each other, depending on their priority. The more general layer is meta (Open Embedded Core meta data), and then layers are stacked. For instance Yocto specific's recipes are in the meta-yocto layer, then machine specific recipes (BSP) are in a meta-xxx-bsp layer etc...

If a recipe is present in 2 layers, the version from the highest priority layer is used.

Additional layers can be defined for local in house developments, for instance one layer for recipes global to the company, one layer per software product, one layer per specific hardware BSP ...

Through configuration files, layers necessary to build the final image are selected. From a product to the other, active layers may differ, sharing a common basic set of layers, and then enabling those layers that make sense for the product. By this way reusability between projects without duplication is easier, reducing maintenance problems.

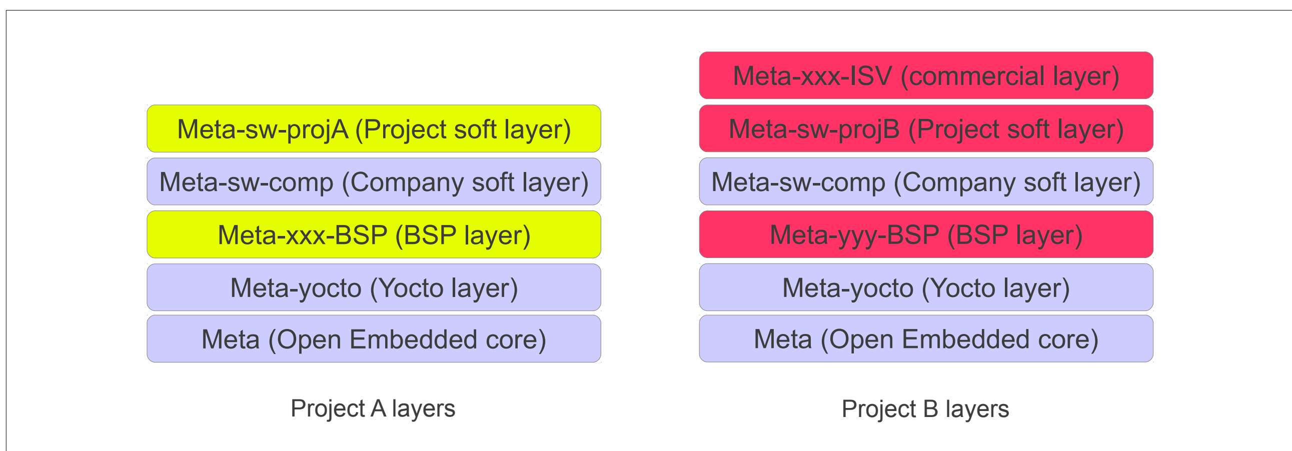

Figure 4 : assembling layers in projects

It is possible to customize a recipe located in an underlying layer, without recipe duplication : the creation of an append recipe file modifies the underlying recipe differentially (same idea than a patch modifies a source code differentially). For instance this mechanism allows to modify password files located in Open Embedded core from an append recipe located in the company software layer.

Using append recipes (bbappend files), developers can customize recipes coming from each underlying layer, avoiding recipes duplication so that new versions of the recipe will be easier to adopt (see Figure 5 and 6).

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${P}:"

SRC_URI += "file://root-passwd.patch \
"
```

Add a patch to underlying recipe

Figure 5 : Append recipe base-passwd-3.5.26.bbpappend

```
We set a password to root

--- base-passwd/passwd.master.orig  2005-07-08 06:31:58.000000000 +0200
+++ base-passwd/passwd.master 2013-02-11 14:31:58.000000000 +0200
@@ -1,4 +1,4 @@
-root::0:0:root:/home/root:/bin/sh
+root:SLvEGe9/3fJBQ:0:0:root:/home/root:/bin/sh
 daemon:*:1:1:daemon:/usr/sbin:/bin/sh
 bin:*:2:2:bin:/bin:/bin/sh
 sys:*:3:3:sys:/dev:/bin/sh
```

Modify root password

Figure 6 : Associated patch file root-passwd.patch

Reproducible images

When developing software for an industrial product, the same image must be generated during years, whatever the circumstances. Simply cloning a master to ensure reproducibility is not a good solution as sometimes bugs must be corrected or new features must be added, implying image regeneration. So it is recommended to suppress any manual task during image generation or deployment (automatic tasks are much more reliable than manual ones). When building a software image, the problem often comes when parameterizing the image, with things like network addresses, user accounts and password (if used), or every parameter that is normally defined interactively on a standard desktop distribution, by installer or by administrator.

In Yocto, all these parameters are defined prior installation, during the building phase of the image. By defining all these parameters set in the recipe of the ad hoc component (for instance IP addresses in the netbase component), the built image is completely defined prior to installation, removing manual configuration tasks, and increasing reliability. If the image has to be rebuilt for any reason, the definition of all parameters in the recipes set guarantees that the final result will be what was expected when designing the embedded distribution, without risk of error due to manual actions.

During the image build, many tools are used on the developer's PC : autotools, make, cross toolchain, potentially perl, python, bison, yacc, etc ….

Normally all these tools would come from the developer's PC distribution : they are subject to change each time the distribution is updated (distribution update is quite current in Linux's world).

To avoid any risk of change in the generated elements, Yocto first compiles the complete set of tools necessary for packages and images builds. It then systematically uses these specific native tools instead of their counterparts coming from the PC distribution (see Figure 7). This isolation from the developer's machine guarantees that updates on this machine won't affect the embedded distribution generated with Yocto. It also guarantees that results will remain exactly the same on different machines in a multi developers team.

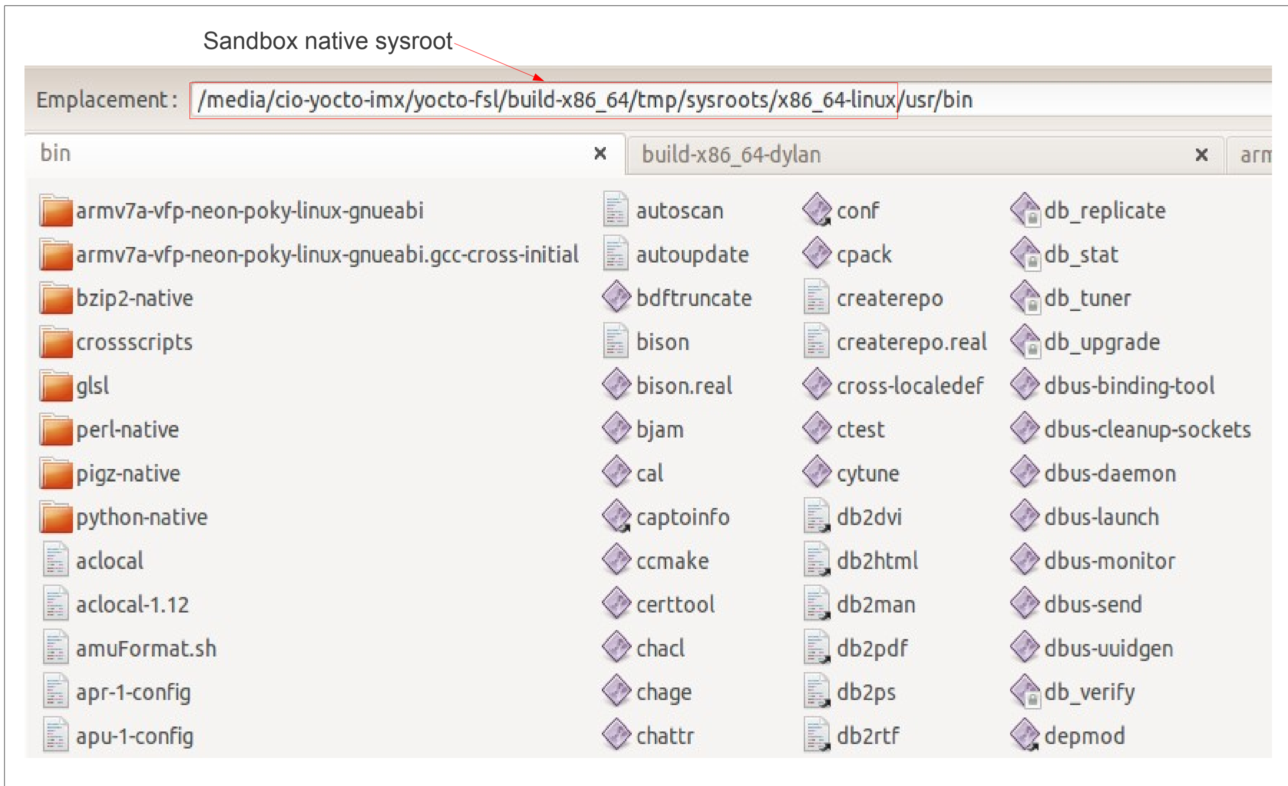

Figure 7 : Location of native tools used during recipe execution

Sandboxing

The role of the sandbox is to completely isolate things managed by Yocto, from the native distribution.

Native tools generated by Yocto and used during image generation are the first half of Yocto's sandbox : the native part.

The second (and main) part of the sandbox gathers components that are related to the target. It generally covers items that are cross-compiled[4].

Generally a software component needs precise versions of used libraries, or at least a precise range of accepted versions.

Software versions used on the target part have no reason to be identical as their counterpart in the native distribution, so particular caution is necessary for includes and libraries versions used during configuration and compilation phases.

The most critical part comes from headers inclusion : by default the toolchain will try to use headers from /usr/include (native headers). If a native header corresponding to a library used during compilation is found in /usr/include and used, and if the native library version is not correct, then it can lead to compilation errors or execution bugs. Compilation errors will be relatively easily detected but execution bugs due to wrong interface definitions can be quite difficult to detect (for instance a structure passed as a parameter to a function of the library with different definition in the caller and in the library).

4 Even if target and host development machine architectures are the same, a “cross-compiler” generated toolchain is used, with same host and target platforms selected during configuration.

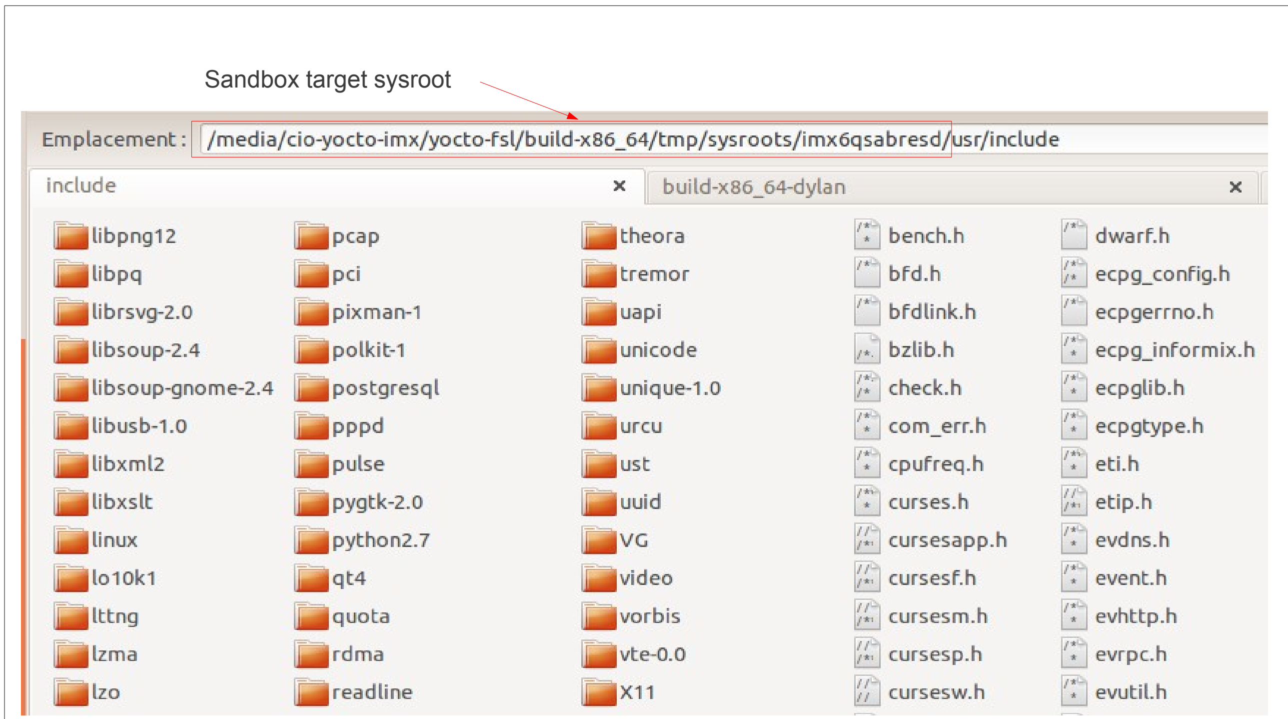

Figure 8 : Location of headers in the target part of the sandbox

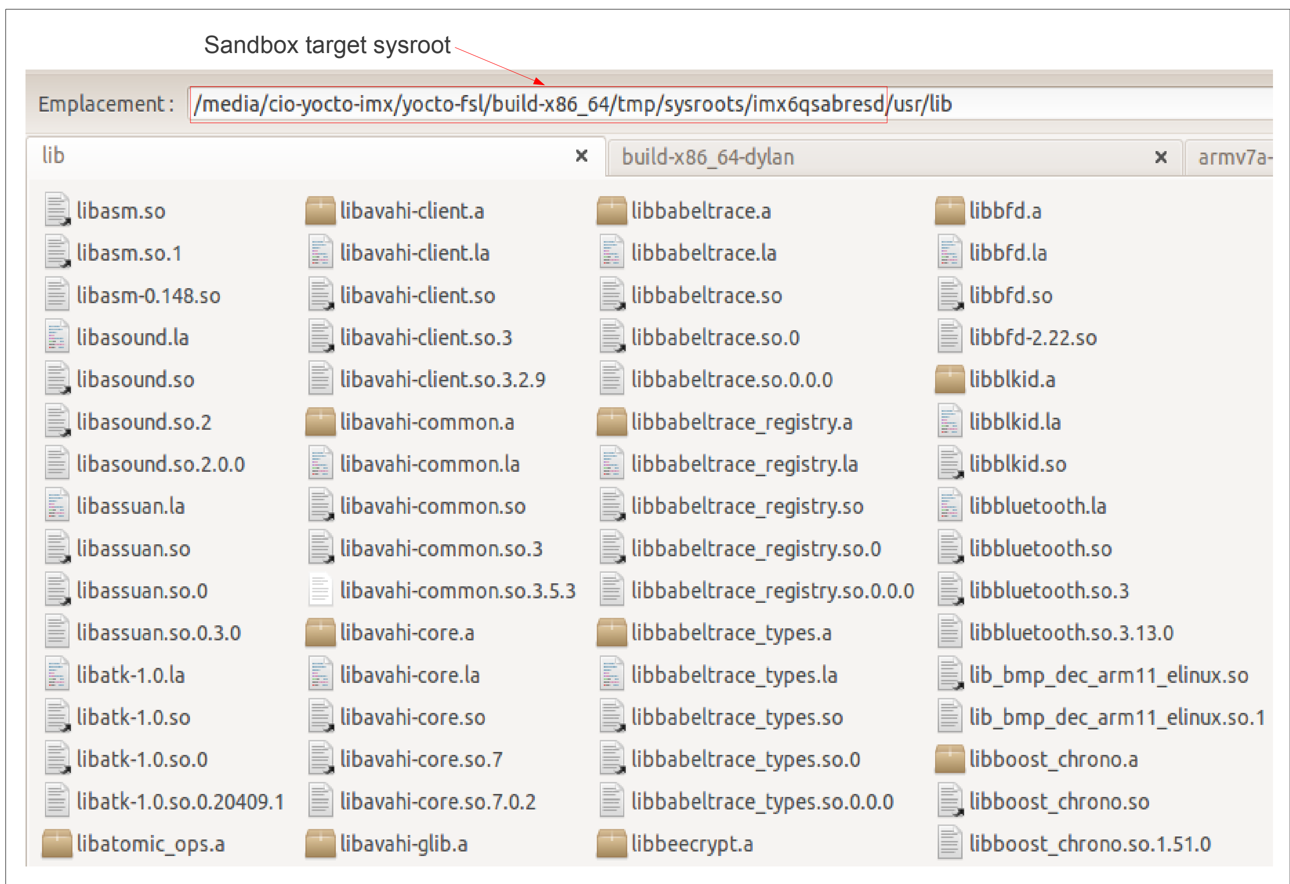

Figure 9 : Location of libraries in the target part of the sandbox

To avoid such mistakes, Yocto deploys all headers and libraries inside the target part of the sandbox, and uses the --sysroot option of gcc to forbid any access to native elements.

With such protection, the success of an application compilation strictly depends on the presence of needed includes and libraries in the sandbox's sysroot. It is the reason why each recipe execution needs a dependency tree resolution and dependent tasks execution in the appropriate order, so that used libraries can be found.

Conclusion

Thanks to its large set of recipes, the investment of major actors and a large users community, Yocto helps embedded developers for complexity management, reproducibility and reliability of the build of a complex embedded image. It offers embedded developers a full feature framework to build embedded Linux distributions that are targeted to the hardware resources of the device (because they are cross compiled for this specific device), and that use all available upstream projects of the Linux ecosystem to fulfill rich feature requirements.

The framework and the community that manage meta data address the complexity and fragmentation of Linux world to allow to offer users high level experience without compromise on quality and reliability of the built images. Nevertheless the tool is mainly a packaging and build tool : the global reliability depends not only on packaging, but also on the intrinsic quality of software components developed in upstream projects. So Yocto contributes to the reliability of embedded developments, but the project is not the only nor ultimate actor to meet this goal.

The framework is used today by major founders to build their Linux Board Support Packages, and by software companies to develop new solutions. Yocto is the foundation of commercial products like Wind River Linux. The release deliveries are predictable, with 2 release per year, one around April and the other around October.

Nevertheless it is not the ultimate tool, and a few drawbacks can be pointed :

- the framework needs a lot of disk space and machine resources.
- the documentation is good for current usages, but advanced usage may require to look under the hood, and a lot of study of internal mechanisms may be necessary to do advanced things the right way (Python knowledge welcome).
- the work is generally done through command line interface. This kind of interface can trouble developers who use to work with Eclipse. A dedicated Eclipse plugin for Yocto is available, but at the moment it can not be used in all use cases.

As a final conclusion Yocto is certainly the framework to use when developing products based on embedded Linux, as long as they need to leverage on the Linux ecosystem and rely on many software components. For simpler embedded distributions, Buildroot can be seen as an alternative, except if the product starts simple but becomes more complex in future versions. In this latter case, it would be better to start directly with Yocto to avoid a package build system change during the lifetime of the product.

References

[1] Buildroot : http://buildroot.uclibc.org/

[2] Scratchbox : https://maemo.gitorious.org/scratchbox2/pages/Home

[3] LTIB : http://ltib.org/

[4] Open Embedded : http://www.openembedded.org/wiki/Main_Page

[5] Ångström : http://www.angstrom-distribution.org/

[6] Yocto Project : https://www.yoctoproject.org/