I.2 – Production d'une image standard

Christophe BLAESS - janvier 2020

Cette étape va nous permettre de produire avec Yocto une image de Linux embarqué prête à être utilisée sur l’émulateur Qemu. Nous allons ainsi vérifier le fonctionnement sur notre système de la configuration la plus simple de Yocto.

Préparation de l'emplacement de travail

Pour mettre en œuvre un système Yocto, il est indispensable de télécharger un minimum de recettes accompagnées des outils de construction nécessaire. Ceci est regroupé dans ce que l’on nomme la **distribution de référence** de Yocto, appelée «**Poky**».

Commençons notre projet en préparant un répertoire de travail dans lequel nous stockerons tous nos fichiers nécessaires à la compilation.

```
[~]$ mkdir  Yocto-lab
[~]$ cd  Yocto-lab/
[Yocto-lab]$
```

Poky

Nous téléchargeons alors la distribution de référence Poky en prenant sa dernière version stable sur le dépôt `git` de Yocto.

Deux versions stables sont proposées par an : la première vers le mois d'avril, la seconde vers le mois de novembre. Les versions sont numérotées et disposent également d'**un nom de code**.

La branche la plus récente au moment de l'écriture de ce texte est la version 3.0, sortie fin octobre 2019. Son nom de code est «**Zeus**». Voici quelques versions précédentes :

Version	Nom de code	Date
1.8	*Fido*	2015-04
2.0	*Jenthro*	2015-11
2.1	*Krogoth*	2016-04
2.2	*Morty*	2016-11
2.3	*Pyro*	2017-05
2.4	*Rocko*	2017-11
2.5	*Sumo*	2018-04
2.6	*Thud*	2018-11
2.7	*Warrior*	2019-05
3.0	*Zeus*	2019-10

D'où viennent ces étranges noms de code ? Il s'agit des noms de robots de combat peuplant un jeu de stratégie datant de la fin des années 90 : «*Total Annihilation*». Voici par exemple une page qui décrit Zeus.

Téléchargeons cette branche :

```
[Yocto-lab]$ git  clone  git://git.yoctoproject.org/poky  -b z
Clonage dans 'poky'...
remote: Counting objects: 461483,, done.
remote: Compressing objects: 100% (108897/108897), done.
remote: Total 461483 (delta 345399), reused 461264 (delta 3451
Réception d'objets: 100% (461483/461483), 160.18 MiB | 4.97 Mi
Résolution des deltas: 100% (345399/345399), fait.
Extraction des fichiers: 100% (5004/5004), fait.
[Yocto-lab]$ ls
poky
[Yocto-lab]$
```

Le répertoire `poky/` est le point central de Yocto, nous y trouvons par exemple le script `bitbake/bin/bitbake` que nous utiliserons pour produire nos images.

À aucun moment nous ne modifierons le contenu de ce répertoire (ni ceux des *layers* que nous serons amenés à télécharger ultérieurement) ; c'est une règle importante dans l'utilisation de Yocto.

Pour assurer la reproductibilité d'un système construit avec Yocto et sa persistance, nous devrons nous discipliner à ne jamais altérer les données téléchargées.

Si une modification d'une recette est nécessaire (que ce soit au niveau du code source ou des données de paramétrage) nous utiliserons un système très efficace de surcharge des recettes.

Sur une branche donnée, il peut y avoir plusieurs versions disponibles (mises à jour, correctifs…). Elles sont identifiées par des ***tags*** pour **`git`**. Pour m'assurer que les opérations à venir soient réplicables par le lecteur intéressé, je bascule sur un *tag* précis. Pour cela je vérifie la liste des *tags* disponibles.

```
[Yocto-lab]$ cd  poky/
[poky]$ git  tag
  [...]
yocto-2.6
yocto-2.6.1
yocto-2.6.2
yocto-2.7
yocto-2.7.1
yocto-3.0
yocto-3.0.1
 [...]
```

```
[poky]$
```

La version `3.0.1` est la plus récente. Je l'extrais :

```
[poky]$ git  checkout  yocto-3.0.1
  [...]
[poky]$ cd  ..
[Yocto-lab]$
```

Attention à bien penser à «remonter» d'un dossier après la commande `git checkout`, nous ne travaillerons jamais dans le répertoire `poky/`.

Initialisation de l'environnement de travail

Nous avons mentionné par exemple que le script `bitbake` se trouve dans le sous-répertoire `poky/bitbake/bin`. Il n'est pour le moment pas accessible par mon shell. Si j'essaye de taper la commande `bitbake`, l'exécution échoue :

```
[Yocto-lab]$ bitbake
bitbake: commande introuvable
[Yocto-lab]$
```

Pour y accéder, il faudrait configurer correctement la variable «PATH» de mon shell. Ainsi que d'autres variables d'environnement nécessaires à la complexité de Yocto. Pour simplifier cette tâche, Poky nous fournit un **script d'initialisation de l'environnement**. On lui passe en argument le nom d'un répertoire de travail à utiliser pour la suite de notre projet. Si le répertoire n'existe pas il le crée. Puis il nous place dans ce répertoire.

Le script s'appelle **`oe-init-build-env`** (oe pour *Open Embedded* l'un des projets initiateurs de Yocto) et se trouve au sommet de l'arborescence de Poky.

Une remarque : pour que le script puisse modifier les variables d'environnement de notre shell, il faut qu'il soit invoqué avec la commande «`source`» ainsi :

```
$ source  nom-du-script
```

ou précédé d'un simple point :

```
$ .  nom-du-script
```

Je vais faire un premier essai avec une cible virtuelle fonctionnant dans **l'émulateur Qemu**. Je choisis donc de nommer mon répertoire de compilation «`build-quemu`».

```
[Yocto-lab]$ source  poky/oe-init-build-env  build-qemu
You had no conf/local.conf file. This configuration file has t
created for you with some default values. You may wish to edit
example, select a different MACHINE (target hardware). See cor
for more information as common configuration options are comme

You had no conf/bblayers.conf file. This configuration file ha
created for you with some default values. To add additional me
into your configuration please add entries to conf/bblayers.co

The Yocto Project has extensive documentation about OE includi
manual which can be found at:
    http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/


### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-ide-support

You can also run generated qemu images with a command like 'ru

Other commonly useful commands are:
 - 'devtool' and 'recipetool' handle common recipe tasks
 - 'bitbake-layers' handles common layer tasks
 - 'oe-pkgdata-util' handles common target package tasks
```

```
[build-qemu]$
```

Lorsqu'on invoque le script «`oe-init-build-env`» et que le répertoire de travail n'existe pas encore, il est plutôt bavard, car il nous décrit tout ce qu'il crée… Il nous indique ensuite quelques cibles de compilation que nous pouvons employer avec la commande `bitbake`. Le *prompt* du shell nous montre que nous sommes dans un nouveau répertoire, vérifions son emplacement et son contenu.

```
[build-quemu]$ pwd
~/Yocto-lab/build-quemu
[build-quemu]$ ls
conf
[build-quemu]$ ls  conf/
bblayers.conf  local.conf  templateconf.cfg
[build-quemu]$
```

Le répertoire de travail a été créé à côté de «`poky/`» et contient pour le moment un unique sous-répertoire avec trois fichiers. Nous serons amenés par la suite à éditer le fichier de configuration «`conf/local.conf`», mais dans un premier temps, satisfaisons-nous de son contenu original afin de nous familiariser avec le fonctionnement de Yocto.

Il est important de comprendre que l'appel du script «`poky/oe-init-build-env`» devra représenter la première tâche de **toutes nos sessions de travail**, même pour retourner dans un répertoire dans lequel on a déjà travaillé au préalable. Ceci est indispensable pour avoir une initialisation correcte des variables d'environnement nécessaires à la compilation.

Production d'une image

Cette configuration minimale est déjà suffisante pour produire une première image. En utilisant les propositions affichées dans les dernières lignes de sortie du script, nous allons lancer la production d'une image basique.

```
[build-quemu]$ bitbake  core-image-minimal
```

Armons-nous de patience, sur un PC moyen cette étape dure environ **deux heures !**

Elle dépend énormément de la puissance du poste de travail et du débit de la connexion Internet. Cette durée peut sembler effayante, mais elle ne concerne que **la première compilation** pour une cible donnée, car Yocto prépare un nombre considérable d'outils et de *packages* qu'il n'a plus besoin de recréer par la suite. Lorsqu'on fait une modification d'une configuration existante (ajout d'un *package*, modification d'un fichier de configuration, etc.) la compilation ne dure que quelques secondes à quelques minutes.

Qu'avons-nous obtenu au bout de ces deux heures de compilation ?

Tout d'abord notre répertoire de travail s'est peuplé de nouveaux sous-répertoires :

```
[build-quemu]$ ls
bitbake-cookerdaemon.log  cache  conf  downloads  sstate-cache
[build-quemu]$
```

Le résultat utile de la compilation se trouve dans **le sous-dossier «`tmp/deploy/`»**. Oui, moi aussi je trouve curieux de placer les fichiers de résultats dans un répertoire nommé «`tmp`», j'aurais préféré un nom comme «`output`» à la manière de Buildroot. Ceci dit, nous verrons qu'il est possible de modifier ce comportement dans le fichier «`local.conf`».

```
[build-quemu]$ ls  tmp/deploy/
images  licenses  rpm
```

On y trouve un répertoire qui contient **les images à installer** sur notre cible, un qui regroupe **les textes des licences** logicielles de toutes les applications, bibliothèques, utilitaires, etc. employés pour produire l'image. Enfin, le répertoire «`rpm/`» contient **un ensemble de *packages*** qui ont été compilés pendant la production et qui pourront être ajoutés par la suite à notre image.

Voyons le contenu du sous-dossier «`images/`». Il ne contient qu'un sous-répertoire représentant l'architecture de la cible :

```
[build-quemu]$ ls  tmp/deploy/images/
qemux86-64
```

Nous voyons que l'architecture choisie par défaut est celle d'un PC (x86, 64 bits) virtuel émulé par l'outil Qemu. Dans ce sous-répertoire, nous trouvons de multiples

fichiers.

```
[build-quemu]$ ls  tmp/deploy/images/qemux86-64/
bzImage
bzImage--5.2.20+git0+bd0762cd13_dd25a04fc5-r0-qemux86-64-20191
bzImage-qemux86-64.bin
core-image-minimal-qemux86-64-20191227155843.qemuboot.conf
core-image-minimal-qemux86-64-20191227155843.rootfs.ext4
core-image-minimal-qemux86-64-20191227155843.rootfs.manifest
core-image-minimal-qemux86-64-20191227155843.rootfs.tar.bz2
core-image-minimal-qemux86-64-20191227155843.testdata.json
core-image-minimal-qemux86-64.ext4
core-image-minimal-qemux86-64.manifest
core-image-minimal-qemux86-64.qemuboot.conf
core-image-minimal-qemux86-64.tar.bz2
core-image-minimal-qemux86-64.testdata.json
modules--5.2.20+git0+bd0762cd13_dd25a04fc5-r0-qemux86-64-20191
modules-qemux86-64.tgz

[build-quemu]$
```

Dans la liste des fichiers ci-dessus, plusieurs sont des liens symboliques (par exemple «`core-image-minimal-qemux86-64.ext4`») pointant vers un fichier avec un nom plus complexe incluant l'horodatage de la production du fichier (comme «`core-image-minimal-qemux86-64-20191227155843.rootfs.ext4`»).

Test de l'image produite

Le fichier principal est «**`core-image-minimal-qemux86-64.ext4`**» qui contient l'image du système de fichiers à fournir à notre PC émulé.

Mais cela ne suffit pas, il faut également une image du noyau Linux, c'est le fichier «`bzImage`».

L'émulateur Qemu est extrêmement souple et configurable, et pour le lancer correctement il faudrait passer une dizaine d'options sur la ligne de commande.

Le fichier «`core-image-minimal-qemux86-64.qemuboot.conf`» regroupe les paramètres de Qemu afin de simplifier son lancement. Il est prévu pour être analysé par **le script «`runqemu`»** fourni par Poky au même titre que «`bitbake`» par

exemple.

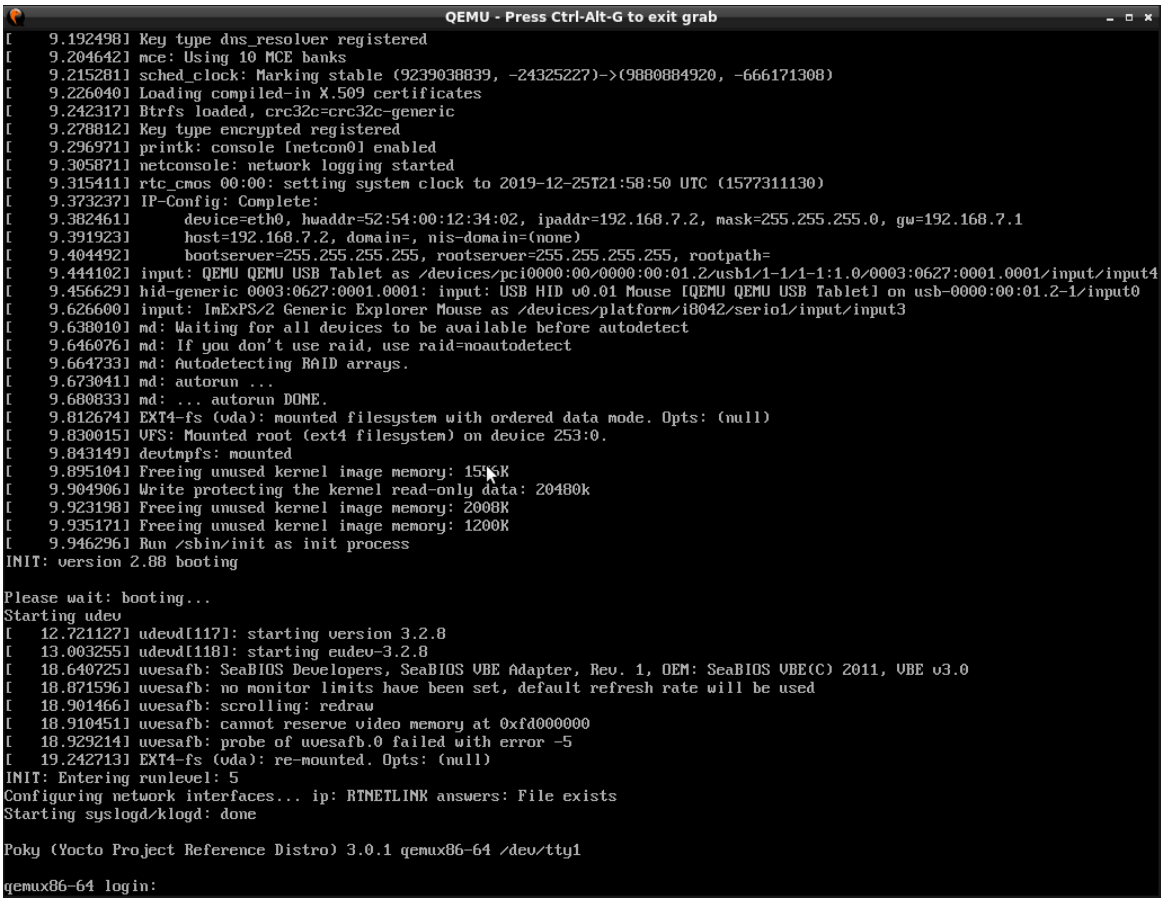
Il présuppose néanmoins que l'utilitaire «`qemu-system-x86_64`» soit présent sur notre PC. Il faut l'installer en utilisant le gestionnaire de logiciels de notre distribution. Sur mon PC Ubuntu je saisis :

```
[buid-quemu]$ sudo  apt  install  qemu-system-x86
```

Pour **lancer Qemu** et tester notre image, il nous suffit donc d'exécuter la commande :

```
[build-quemu]$ runqemu  qemux86-64
runqemu - INFO - Running MACHINE=qemux86-64 bitbake -e...
runqemu - INFO - Continuing with the following parameters:
KERNEL: [/home/cpb/Yocto-lab/build-quemu/tmp/deploy/images/qem
MACHINE: [qemux86-64]
FSTYPE: [ext4]
  [...]
```

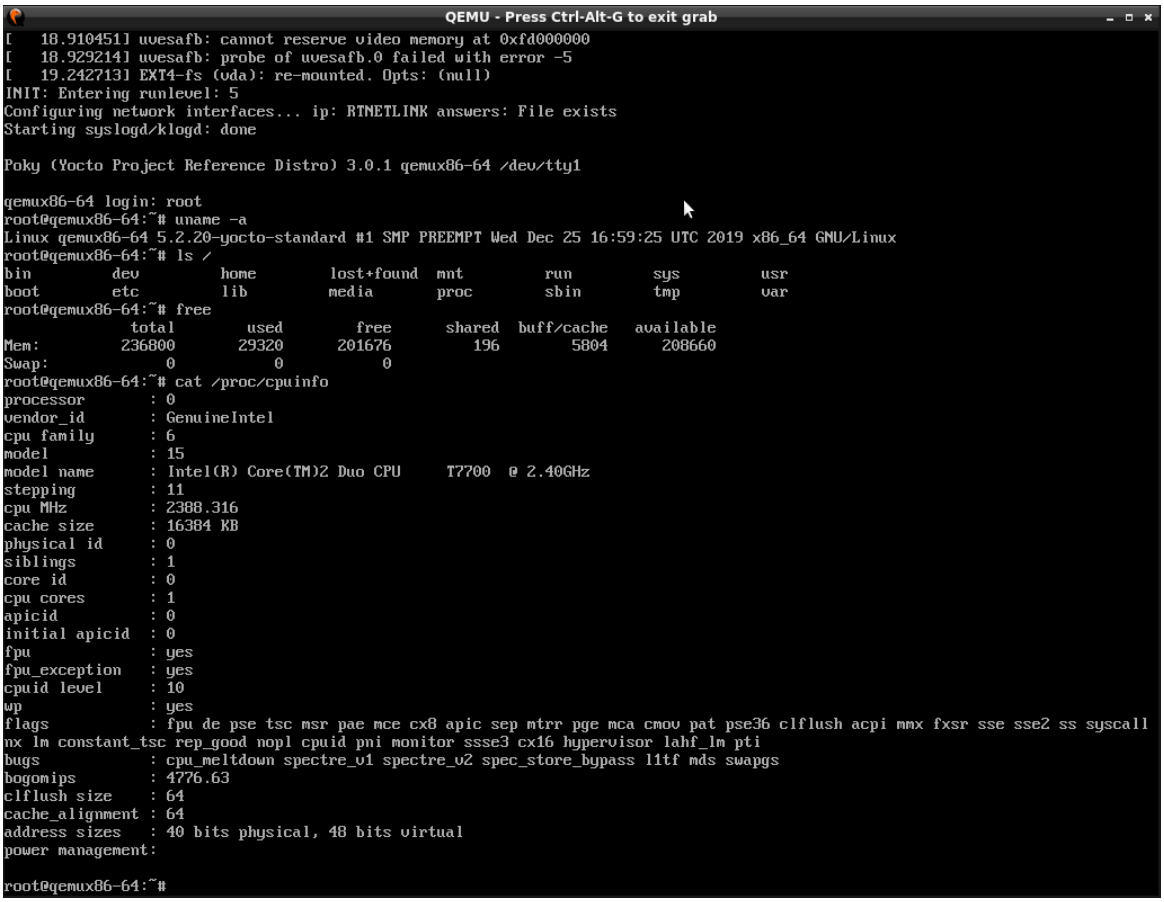
Une fenêtre apparaît avec la console de notre machine virtuelle.



La figure I.2-1 nous montre les traces de la fin du boot et l'invitation à la connexion. Nous pouvons nous connecter sous l'identité `root` — par défaut il n'y a pas de mot de passe.

Si vous cliquez dans la fenêtre de Qemu, il est possible que celle-ci vous capture le pointeur de la souris et que vous ne puissiez pas en sortir. La solution est inscrite dans la barre de titre de la fenêtre : Pressez simultanément les touches «CTRL», «ALT» et «G».

On peut voir sur la figure I.2-2 la saisie de quelques commandes dans cette machine virtuelle. Notez que le clavier est en mode *Qwerty*, ce qui peut poser des soucis de saisie dans les environnements francophones. Il est possible de passer l'argument «`nographic`» sur la ligne de commande de «`runqemu`» (avant l'argument «`qemux86-64`») pour que la connexion se fasse directement dans la console texte, et conserve la même configuration du clavier.



Conclusion

Je vous encourage à réaliser cette première expérience car elle vous permettra de valider votre plateforme de compilation. Il faut en effet disposer d'un certain nombre d'outils («`make`», «`gcc`», «`git`», etc.) qu'il vous faudra éventuellement installer au préalable.

Cette première compilation sans autre intervention manuelle que le téléchargement de Poky, l'invocation d'un script et le lancement de la commande «`bitbake`» permet de vérifier que tout est en place pour le fonctionnement de Yocto.

En outre l'installation de «`qemu-system-x86`» ne prend pas beaucoup de temps et permet de tester une image Yocto, vérifier son contenu, se familiariser avec l'arborescence, sans avoir besoin de transférer l'image sur une carte cible, ni de se soucier de la connectique à utiliser. Bien sûr nous construirons par la suite des images pour de véritables systèmes embarqués.

Un mot concernant la **plateforme de compilation** : l'idéal est de disposer d'un bon PC avec au minimum quatre cœurs, 8 Go de RAM et une centaine de Go de disque dur disponible. Yocto est un outil assez gourmand, tant en espace disque qu'en puissance CPU. S'il est possible de le faire fonctionner sur une machine virtuelle, je

le déconseille pendant la phase d’expérimentation et de prototypage car les temps de compilation deviennent très longs et vite démotivants.

«« **sommaire** »»