

I.2 – Production d'une image standard

Christophe BLAESS - janvier 2020

- Préparation de l'emplacement de travail
- Poky
- Initialisation de l'environnement de travail
- Production d'une image
- Test de l'image produite
- Conclusion

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	<i>Fido</i>	2015-04
2.0	<i>Jenthro</i>	2015-11
2.1	<i>Krogoth</i>	2016-04
2.2	<i>Morty</i>	2016-11
2.3	<i>Pyro</i>	2017-05
2.4	<i>Rocko</i>	2017-11
2.5	<i>Sumo</i>	2018-04
2.6	<i>Thud</i>	2018-11
2.7	<i>Warrior</i>	2019-05
3.0	<i>Zeus</i>	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 devons 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 répliquables 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-qemu».

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

```
You had no conf/bblayers.conf file. This configuration file has been
created for you with some default values. To add additional meta layers
into your configuration please add entries to conf/bblayers.conf.
```

```
The Yocto Project has extensive documentation about OE including a
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 'runqemu'
```

```
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 effrayante, 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-qemu]$ 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-qemu]$
```

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 :

```
[build-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/gen
MACHINE: [qemux86-64]
FSTYPE: [ext4]
[...]
```

Une fenêtre apparaît avec la console de notre machine virtuelle.

```
QEMU - Press Ctrl-Alt-G to exit grab
[ 9.192498] Key type dns_resolver registered
[ 9.204642] mce: Using 10 MCE banks
[ 9.215281] sched_clock: Marking stable (9239038839, -24325227)->(9880884920, -666171308)
[ 9.226040] Loading compiled-in X.509 certificates
[ 9.242317] Btrfs loaded, crc32c=crc32c-generic
[ 9.278812] Key type encrypted registered
[ 9.296971] printk: console [netcon0] enabled
[ 9.305871] netconsole: network logging started
[ 9.315411] rtc_cmos 00:00: setting system clock to 2019-12-25T21:58:50 UTC (1577311130)
[ 9.373237] IP-Config: Complete:
[ 9.382461]     device=eth0, hwaddr=52:54:00:12:34:02, ipaddr=192.168.7.2, mask=255.255.255.0, gw=192.168.7.1
[ 9.391923]     host=192.168.7.2, domain=, nis-domain=(none)
[ 9.404492]     bootserver=255.255.255.255, rootserver=255.255.255.255, rootpath=
[ 9.444102] input: QEMU QEMU USB Tablet as /devices/pci0000:00/0000:00:01.2/usb1/1-1/1-1.0/0003:0627:0001.0001/input/input4
[ 9.456629] hid-generic 0003:0627:0001.0001: input: USB HID v0.01 Mouse [QEMU QEMU USB Tablet] on usb-0000:00:01.2-l/input0
[ 9.626600] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
[ 9.638010] md: Waiting for all devices to be available before autodetect
[ 9.646076] md: If you don't use raid, use raid=noautodetect
[ 9.664733] md: Autodetecting RAID arrays.
[ 9.673041] md: autorun ...
[ 9.680833] md: ... autorun DONE.
[ 9.812674] EXT4-fs (vda): mounted filesystem with ordered data mode. Opts: (null)
[ 9.830015] VFS: Mounted root (ext4 filesystem) on device 253:0.
[ 9.843149] devtmpfs: mounted
[ 9.895104] Freeing unused kernel image memory: 153K
[ 9.904906] Write protecting the kernel read-only data: 20480k
[ 9.923198] Freeing unused kernel image memory: 2008K
[ 9.935171] Freeing unused kernel image memory: 1200K
[ 9.946296] Run /sbin/init as init process
INIT: version 2.88 booting
Please wait: booting...
Starting udev
[ 12.721127] udevd[117]: starting version 3.2.8
[ 13.003255] udevd[118]: starting udev-3.2.8
[ 18.640725] uvesafb: SeaBIOS Developers, SeaBIOS VBE Adapter, Rev. 1, OEM: SeaBIOS VBE(C) 2011, VBE v3.0
[ 18.871596] uvesafb: no monitor limits have been set, default refresh rate will be used
[ 18.901466] uvesafb: scrolling: redraw
[ 18.910451] uvesafb: cannot reserve video memory at 0xfda000000
[ 18.929214] uvesafb: probe of uvesafb.0 failed with error -5
[ 19.242713] EXT4-fs (vda): re-mounted. Opts: (null)
INIT: Entering runlevel: 5
Configuring network interfaces... ip: RTNETLINK answers: File exists
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 3.0.1 qemu86-64 /dev/tty1
qemu86-64 login:
```

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 «qemu86-64») pour que la connexion se fasse directement dans la console texte, et conserve la même configuration du clavier.

```
QEMU - Press Ctrl-Alt-G to exit grab
[ 18.910451] uvesafb: cannot reserve video memory at 0xf000000
[ 18.929214] uvesafb: probe of uvesafb.0 failed with error -5
[ 19.242713] EXT4-fs (vda): re-mounted. Opts: (null)
INIT: Entering runlevel: 5
Configuring network interfaces... ip: RTNETLINK answers: File exists
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 3.0.1 qemu86-64 /dev/tty1

qemu86-64 login: root
root@qemu86-64:~# uname -a
Linux qemu86-64 5.2.20-yocto-standard #1 SMP PREEMPT Wed Dec 25 16:59:25 UTC 2019 x86_64 GNU/Linux
root@qemu86-64:~# ls /
bin      dev      home     lost+found  mnt      run      sys      usr
boot    etc      lib      media      proc     sbin     tmp      var
root@qemu86-64:~# free
             total        used        free      shared  buff/cache   available
Mem:      236800         29320       201676         196        5804       208660
Swap:            0              0         0
root@qemu86-64:~# cat /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 15
model name     : Intel(R) Core(TM)2 Duo CPU     T7700  @ 2.40GHz
stepping      : 11
cpu MHz        : 2388.316
cache size     : 16384 KB
physical id    : 0
siblings       : 1
core id        : 0
cpu cores      : 1
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 10
wp             : yes
flags           : fpu de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush acpi mmx fxsr sse sse2 ss syscall
nx lm constant_tsc rep_good nopl cpuid pni monitor ssse3 cx16 hypervisor lahf_lm pti
bugs           : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs
bogomips       : 4776.63
clflush size   : 64
cache alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:

root@qemu86-64:~#
```

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.



Ce document est placé sous licence *Creative Common CC-by-nc*. Vous pouvez copier son contenu et le réemployer à votre gré pour une utilisation non-commerciale. Vous devez en outre mentionner sa provenance.



««

sommaire

»»