

I.3 – Production d'images pour des cibles spécifiques

Christophe BLAESS - janvier 2020

- Variable d'environnement MACHINE
- Image pour émulateur ARM
- Et sur une vraie cible embarquée ?
- Un Raspberry Pi, sinon rien !
- Conclusion

Dans la séquence précédente nous avons créé une image standard pour une émulation de processeur x86. Nous allons à présent tester quelques autres cibles possibles : **émulateur Arm**, cartes **Beaglebone Black** et **Raspberry Pi**.

Variable d'environnement MACHINE

Comment Yocto connaît-il la cible pour laquelle nous souhaitons préparer une image ?

Il s'appuie sur la **variable d'environnement «MACHINE»**. Celle-ci doit contenir le nom d'une cible connue par Yocto. Lorsqu'on utilise simplement les *layers* de Poky, comme nous l'avons fait précédemment, la liste est limitée.

Nom	Cible
-----	-------

qemuarm	Émulation de système à processeur ARM 32 bits
qemuarm64	Émulation de système à processeur ARM 64 bits
qemumips	Émulation de système à processeur MIPS 32 bits
qemumips64	Émulation de système à processeur MIPS 64 bits
qemuppc	Émulation de système à processeur PowerPC
qemux86	Émulation de système à processeur x86 32 bits
qemux86-64	Émulation de système à processeur x86 64 bits
beaglebone-yocto	Famille de <i>Single Board Computers</i> ARM 32 bits
genericx86	PC standard à processeur x86 32 bits
genericx86-64	PC standard à processeur x86 64 bits
mpc8315e-rdb	Carte pour processeur PowerQuicc II (PowerPC)
edgerouter	Routeur à processeur MIPS 64 bits

Nous trouvons la liste de ces cibles dans les sous-répertoires «poky/meta/conf/machine/» et «poky/meta-yocto-bsp/conf/machine/».

Nous pouvons également les trouver au début du fichier «conf/local.conf» qui a été automatiquement créé lorsque nous avons appelé le script «poky/oe-init-build-env» pour la première fois.

```
[build-qemu]$ head -40 conf/local.conf
[...]
#
# Machine Selection
#
# You need to select a specific machine to target the build with
# of emulated machines available which can boot and run in the
#
#MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemumips64"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemux86"
#MACHINE ?= "qemux86-64"
#
# There are also the following hardware board target machines
# demonstration purposes:
#
#MACHINE ?= "beaglebone-yocto"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
#MACHINE ?= "mpc8315e-rdb"
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemux86-64 if no other machine
#MACHINE ??= "qemux86-64"
#
```

Dans ce fichier les lignes commençant par un caractère «#» sont considérées comme des commentaires. La seule qui configure la variable «MACHINE» est donc la dernière de cet extrait. C'est ainsi que Yocto a su qu'il devait nous préparer une image pouvant être prise en charge par l'émulateur Qemu-x86.

Mais nous voyons également que l'affectation de la variable est curieuse :

```
MACHINE ??= "qemux86"
```

On peut se demander ce que signifie le symbole «**??=**».

Nous étudierons la syntaxe des fichiers de Yocto ultérieurement, mais précisons simplement pour le moment que cela signifie que la variable n'est affectée que si elle n'a pas de valeur préalable.

Autrement dit, nous pouvons remplir la variable avant de lancer «**bitbake**» et notre affectation aura précedence sur celle par défaut indiqué dans «**conf/local.conf**».

Dans les prochaines étapes nous inscrirons le nom de la machine cible désirée dans ce fichier, mais pour le moment, contentons-nous d'interagir uniquement depuis la ligne de commande.

Image pour émulateur ARM

La syntaxe du *shell* nous permet de précéder une commande (comme «**bitbake**») d'une affectation de variable d'environnement. Essayons cela tout de suite (prévoyez encore un «petit» moment de compilation...).

```
[build-qemu]$ MACHINE=qemuarm bitbake core-image-minimal
Parsing recipes: 100% |#####| Time: 0:01:00
Parsing of 772 .bb files complete (0 cached, 772 parsed). 1298
NOTE: Resolving any missing task queue dependencies
```

Build Configuration:

```
BB_VERSION           = "1.44.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "qemuarm"
DISTRO               = "poky"
DISTRO_VERSION        = "3.0.1"
TUNE_FEATURES        = "arm armv7ve vfp thumb neon callconvent
TARGET_FPU           = "hard"
meta
meta-poky
```

```
meta-yocto-bsp          = "HEAD:12a4c177bb541b3187c7a54d5804f30c3"
```

```
Initialising tasks: 100% |#####| Time: 0:00:05  
Sstate summary: Wanted 557 Found 15 Missed 542 Current 255 (29  
NOTE: Executing Tasks  
NOTE: Setscene tasks completed  
NOTE: Tasks Summary: Attempted 2640 tasks of which 1288 didn't
```

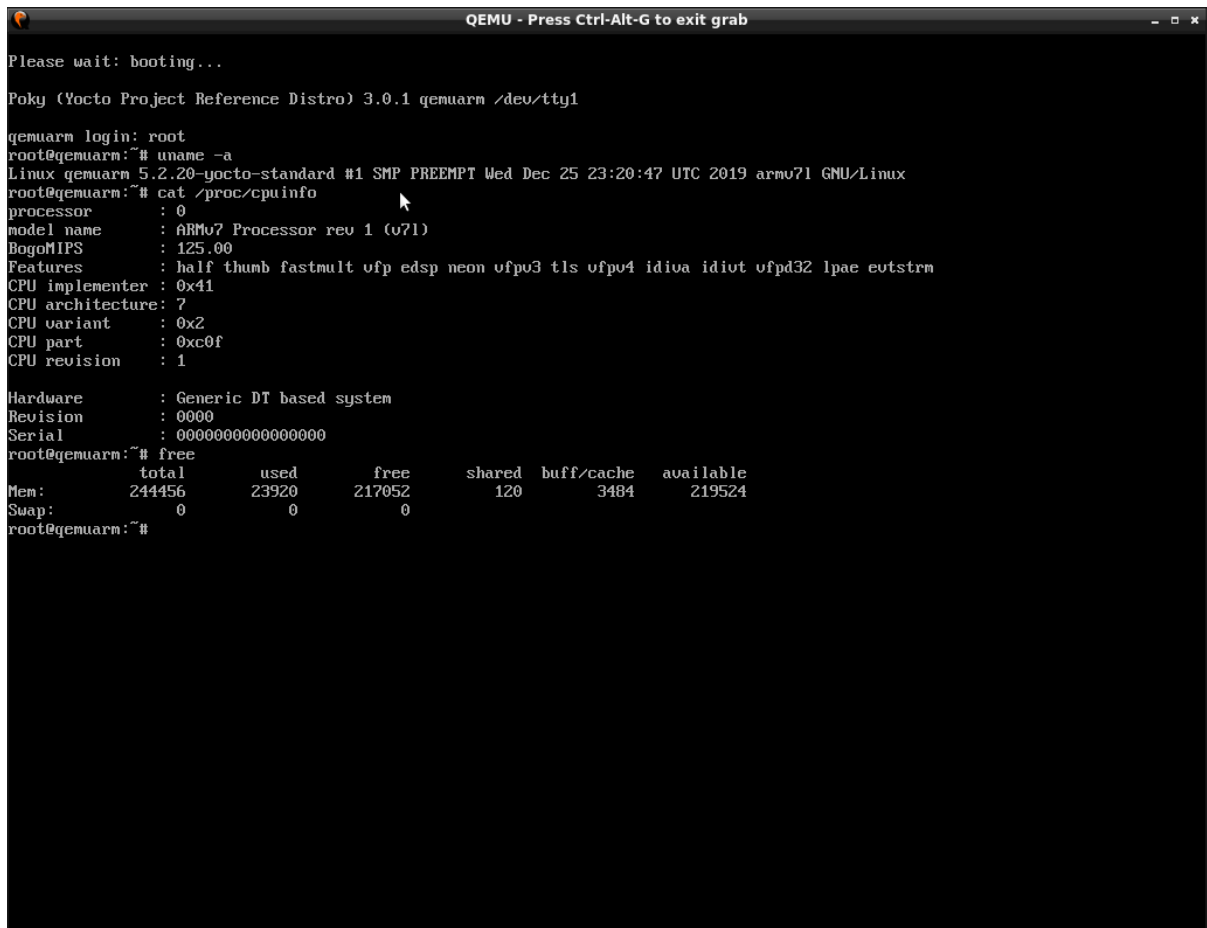
Lorsque bitbake s'arrête de travailler, nous pouvons voir la nouvelle image :

```
[build-qemu]$ ls tmp/deploy/images/  
qemuarm  qemux86-64  
[build-qemu]$ ls tmp/deploy/images/qemuarm/  
core-image-minimal-qemuarm-20191227224904.qemuboot.conf  
core-image-minimal-qemuarm-20191227224904.rootfs.ext4  
core-image-minimal-qemuarm-20191227224904.rootfs.manifest  
core-image-minimal-qemuarm-20191227224904.rootfs.tar.bz2  
core-image-minimal-qemuarm-20191227224904.testdata.json  
core-image-minimal-qemuarm.ext4  
core-image-minimal-qemuarm.manifest  
core-image-minimal-qemuarm.qemuboot.conf  
core-image-minimal-qemuarm.tar.bz2  
core-image-minimal-qemuarm.testdata.json  
modules--5.2.20+git0+bd0762cd13_fcbe51dfa0-r0-qemuarm-20191227  
modules-qemuarm.tgz  
zImage  
zImage--5.2.20+git0+bd0762cd13_fcbe51dfa0-r0-qemuarm-20191227  
zImage-qemuarm.bin  
[build-qemu]$
```

Il nous faut à nouveau installer une version adaptée de **Qemu pour processeur Arm**.

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

Ensuite nous pouvons lancer le script «runqemu» avec le nom de l'architecture «qemuarm» en paramètre.



```
QEMU - Press Ctrl-Alt-G to exit grab

Please wait: booting...

Poky (Yocto Project Reference Distro) 3.0.1 qemuarm /dev/tty1

qemuarm login: root
root@qemuarm:~# uname -a
Linux qemuarm 5.2.20-yocto-standard #1 SMP PREEMPT Wed Dec 25 23:20:47 UTC 2019 armv7l GNU/Linux
root@qemuarm:~# cat /proc/cpuinfo
processor       : 0
model name     : ARMv7 Processor rev 1 (v71)
BogoMIPS      : 125.00
Features      : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae evtstrm
CPU implementer : 0x41
CPU architecture: 7
CPU variant    : 0x2
CPU part       : 0xc0f
CPU revision   : 1

Hardware       : Generic DT based system
Revision      : 0000
Serial        : 0000000000000000
root@qemuarm:~# free
              total        used        free      shared  buff/cache   available
Mem:           244456         23920        217052          120         3484        219524
Swap:              0              0              0
```

La [figure I.3-1](#) nous montre la fenêtre de l'émulateur et la commande «`uname -a`» nous indiquant que l'architecture est bien de type ARM.

Et sur une vraie cible embarquée ?

L'utilisation d'un émulateur comme Qemu présente de nombreux avantages pour la mise au point d'un système embarqué. Toutefois, cela a tendance à dissimuler les problèmes que l'on rencontre lorsque l'on passe à des cibles réelles (configuration du *bootloader*, *flashage* de la mémoire, connexion au système, etc.)

Dans les architectures connues par Yocto, il y a **la famille des BeagleBones**. Nous pouvons relancer un *build* pour cette architecture. Comme notre cible ne sera plus l'émulateur Qemu pour le moment, je préfère recréer un nouveau répertoire de

compilation à côté de «build-qemu».

Disons «build-bbb» comme abréviation de *Beagle Bone Black*, la carte que je vais utiliser.

```
[build-qemu]$ cd ..  
[Yocto-lab]$ source poky/oe-init-build-env build-bbb
```

Je relance une compilation avec une nouvelle architecture cible.

```
[build-bbb]$ MACHINE=beaglebone-yocto bitbake core-image-minimal  
Parsing recipes: 100% |#####  
Parsing of 772 .bb files complete (0 cached, 772 parsed). 1298 nodes  
NOTE: Resolving any missing task queue dependencies
```

Build Configuration:

```
BB_VERSION           = "1.44.0"  
BUILD_SYS            = "x86_64-linux"  
NATIVELSBSTRING     = "ubuntu-18.04"  
TARGET_SYS          = "arm-poky-linux-gnueabi"  
MACHINE              = "beaglebone-yocto"  
DISTRO               = "poky"  
DISTRO_VERSION       = "3.0.1"  
TUNE_FEATURES        = "arm vfp cortexa8 neon callconvention-fp"  
TARGET_FPU           = "hard"  
meta  
meta-poky  
meta-yocto-bsp       = "HEAD:12a4c177bb541b3187c7a54d5804f30c3"
```

Après un nouveau moment de compilation, nous pouvons observer les images produites :

```
[build-bbb]$ ls tmp/deploy/images/beaglebone-yocto/  
[...]  
core-image-minimal-beaglebone-yocto.wic
```

```
[...]  
[build-bbb]$
```

Je n'ai laissé apparaître que le fichier qui va nous servir directement, mais il y en a une trentaine dans ce répertoire (notamment des liens symboliques pointant vers des versions horodatées).

L'installation de l'image sur une *Beaglebone Black* est simple car Yocto a l'amabilité de nous fournir un gros fichier doté de l'**extension «.wic»** qui contient toutes les données à inscrire sur la carte micro-SD, y compris la table des partitions nécessaire. Le script «wic» qui crée ces fichiers est fourni par Poky comme «bitbake» ou «runqemu».

J'insère sur mon PC de travail une carte micro-SD par l'intermédiaire d'un adaptateur USB. J'examine les périphériques blocs disponibles.

```
[build-bbb]$ lsblk
```

NAME	MAJ:MIN	RM	SIZE	RO	TYPE	MOUNTPPOINT
sda	8:0	0	465,8G	0	disk	
├sda1	8:1	0	400G	0	part	/home/testing
└sda2	8:2	0	8G	0	part	[SWAP]
sdb	8:16	1	14,4G	0	disk	
├sdb1	8:17	1	64M	0	part	/media/cpb/BOOT
└sdb2	8:18	1	1G	0	part	/media/cpb/ROOT
nvme0n1	259:0	0	232,9G	0	disk	
├nvme0n1p1	259:1	0	512M	0	part	/boot/efi
├nvme0n1p2	259:2	0	224,6G	0	part	/
└nvme0n1p3	259:3	0	7,8G	0	part	
└─cryptswap1	253:0	0	7,8G	0	crypt	

La carte micro-SD est accessible ici en tant que «/dev/sdb». Comme elle contenait déjà des partitions elle a été auto-montée. Je la démonte pour la détacher du système de fichiers.

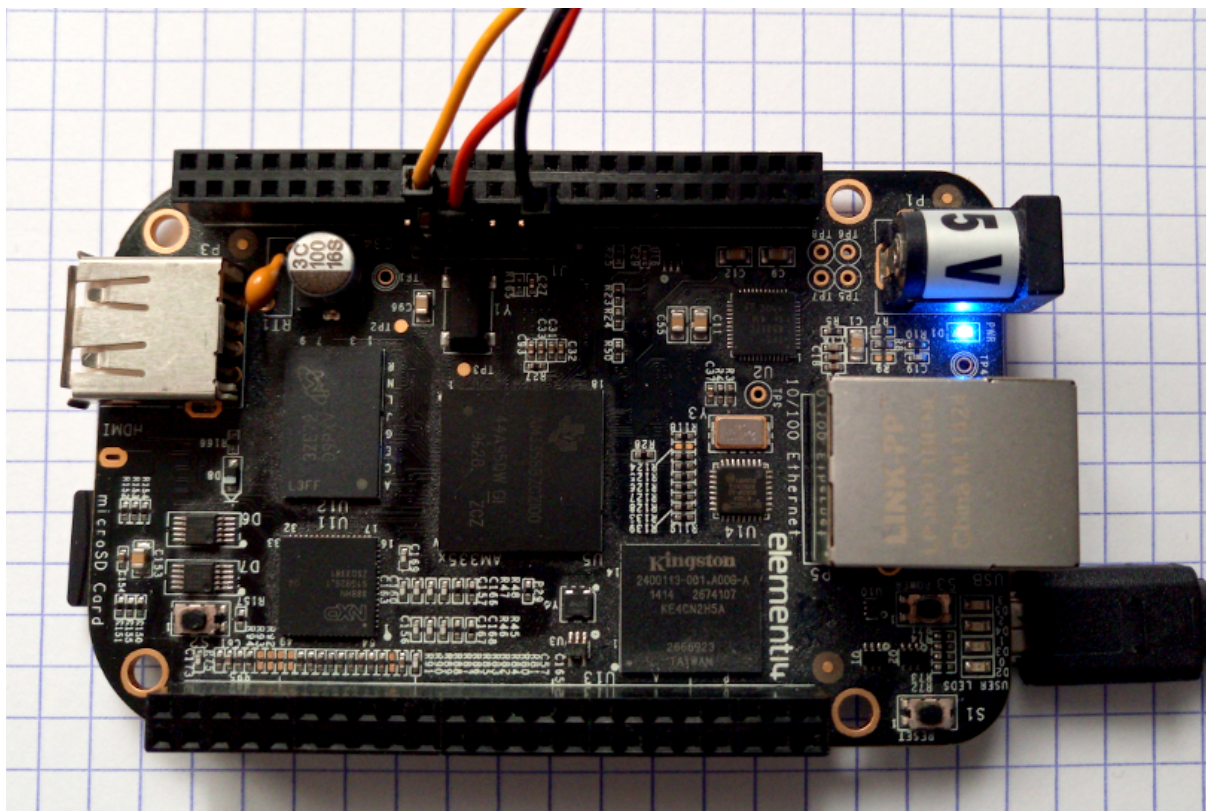
```
[build-bbb]$ umount /dev/sdb?
```


Une fois que je suis sûr que les deux partitions sont bien démontées, je viens écrire le fichier d'extension « .wic » directement sur l'ensemble du périphérique représentant toute la carte micro-SD (« /dev/sdb » dans mon cas).

```
[build-bbb]$ sudo cp tmp/deploy/images/beaglebone-yocto/core  
[build-bbb]$
```

Je peux à présent extraire la carte micro-SD de mon PC, l'insérer dans la *Beaglebone Black* et démarrer celle-ci (suivant les versions de *BeagleBone* il peut être nécessaire de presser un bouton spécifique pour indiquer que l'on souhaite démarrer sur la carte micro-SD et non sur la mémoire eMMC interne).

Je me connecte sur la Beaglebone par l'intermédiaire d'un adaptateur USB-Série (les trois fils jaune, rouge et noir de la [figure I.3-2](#)).



Nous pouvons examiner les traces de *boot* dans la console d'un émulateur de terminal (*minicom*) sur le poste de développement.

U-Boot SPL 2019.07 (Dec 28 2019 - 06:43:28 +0000)
Trying to boot from MMC1

U-Boot 2019.07 (Dec 28 2019 - 06:43:28 +0000)

CPU : AM335X-GP rev 2.1
Model: TI AM335x BeagleBone Black
DRAM: 512 MiB
NAND: 0 MiB
MMC: OMAP SD/MMC: 0, OMAP SD/MMC: 1
Loading Environment from FAT... *** Warning - bad CRC, using c

<ethaddr> not set. Validating first E-fuse MAC

Net: eth0: ethernet@4a100000

Warning: usb_ether MAC addresses don't match:

Address in ROM is de:ad:be:ef:00:01

Address in environment is 84:eb:18:ae:85:d0

, eth1: usb_ether

Hit any key to stop autoboot: 0

switch to partitions #0, OK

mmc0 is current device

SD/MMC found on device 0

switch to partitions #0, OK

mmc0 is current device

Scanning mmc 0:1...

Found /extlinux/extlinux.conf

Retrieving file: /extlinux/extlinux.conf

119 bytes read in 2 ms (57.6 KiB/s)

1: Yocto

Retrieving file: /zImage

6785528 bytes read in 435 ms (14.9 MiB/s)

append: root=PARTUUID=6431d6aa-02 rootwait console=ttyS0,115200

Retrieving file: /am335x-boneblack.dtb

58296 bytes read in 6 ms (9.3 MiB/s)

Flattened Device Tree blob at 88000000

Booting using the fdt blob at 0x88000000

Loading Device Tree to 8ffee000, end 8ffff3b7 ... OK

Starting kernel ...

Le *bootloader* a fini de démarrer, de charger le noyau Linux en mémoire (fichier «/zImage») ainsi que le *device tree* (fichier «/am335x-boneblack.dtb») qui décrit le matériel présent. Le *boot* du noyau est à présent possible.

```
Booting Linux on physical CPU 0x0
Linux version 5.2.17-yocto-standard (oe-user@oe-host) (gcc ver
CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c5387
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruc
OF: fdt: Machine model: TI AM335x BeagleBone Black
Memory policy: Data cache writeback
cma: Reserved 16 MiB at 0x9e800000
CPU: All CPU(s) started in SVC mode.
AM335X ES2.1 (sgx neon)
[...]
VFS: Mounted root (ext4 filesystem) readonly on device 179:2.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
Run /sbin/init as init process
```

Cette dernière ligne indique que l'essentiel du démarrage du noyau est terminé, et qu'il passe le contrôle à l'espace utilisateur.

```
INIT: version 2.88 booting
Starting udev
udev[125]: starting version 3.2.8
udev[126]: starting eudev-3.2.8
EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
Thu Dec 26 06:37:37 UTC 2019
Configuring packages on first boot....
(This may take several minutes. Please do not power off the m
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purg
Removing any system startup links for run-postinsts ...
```

```
/etc/rcS.d/S99run-postinsts
INIT: Entering runlevel: 5
[...]
udhcpd: sending discover
udhcpd: sending discover
udhcpd: no lease, forking to background
done.
Starting syslogd/klogd: done
```

Il n'y a plus qu'à se connecter et tester quelques commandes.

Poky (Yocto Project Reference Distro) 3.0.1 beaglebone-yocto /

```
beaglebone-yocto login: root
root@beaglebone-yocto:~# uname -a
Linux beaglebone-yocto 5.2.17-yocto-standard #1 PREEMPT Thu Dec 10 14:54:11 UTC 2020; root@beaglebone-yocto:~# cat /proc/cpuinfo
processor          : 0
model name        : ARMv7 Processor rev 2 (v7l)
BogoMIPS         : 996.14
Features          : half thumb fastmult vfp edsp thumb2 neon vfpv3
CPU implementer   : 0x41
CPU architecture : 7
CPU variant      : 0x3
CPU part         : 0xc08
CPU revision     : 2

Hardware         : Generic AM33XX (Flattened Device Tree)
Revision        : 0000
Serial          : 2715BBBK2037
root@beaglebone-yocto:~# cat /sys/firmware/devicetree/base/model
TI AM335x BeagleBone Blackroot@beaglebone-yocto
```

Un Raspberry Pi, sinon rien !

La carte fétiche des bidouilleurs Linux de nos jours est l'inévitable **Raspberry Pi**. Bien entendu Yocto permet de générer une image pour cette cible. Toutefois, il nous faut **télécharger un *layer* supplémentaire**, un répertoire contenant les recettes et éléments de configuration propres à cette carte. Les *layers* de Yocto sont faciles à identifier, car leurs noms commencent par le **préfixe «meta-»**.

J'ai pour habitude de télécharger les *layers* que j'ajoute «à côté» du répertoire poky/. Rien ne nous y oblige, il est possible de les regrouper dans un sous-dossier spécifique si on le préfère

Nous verrons dans les prochaines étapes comment rechercher un *layer* adapté à l'architecture ou à la fonctionnalité souhaitées. Dans un premier temps, acceptons simplement l'URL fournie ci-dessous.

```
[Yocto-lab]$ git clone git://git.yoctoproject.org/meta-raspl
Clonage dans 'meta-raspberrypi'...
remote: Counting objects: 6991, done.
remote: Compressing objects: 100% (2962/2962), done.
remote: Total 6991 (delta 3867), reused 6675 (delta 3658)
Réception d'objets: 100% (6991/6991), 1.53 MiB | 764.00 KiB/s,
Résolution des deltas: 100% (3867/3867), fait.
[Yocto-lab]$ ls
build-bbb  build-qemu  meta-raspberrypi  poky
[Yocto-lab]$
```

Nous voyons qu'un répertoire «meta-raspberrypi/» est bien apparu. Préparons à nouveau un répertoire de travail.

```
[Yocto-lab]$ source poky/oe-init-build-env build-rpi
You had no conf/local.conf file. This configuration file has t
[...]
[build-rpi]$
```

Nous devons commencer par **ajouter le *layer*** téléchargé plus haut dans la configuration actuelle. Regardons d'abord la liste des *layers* déjà pris en compte pour le futur *build*.

La commande «**bitbake-layers**» et son option «**show-layers**» vont nous servir.

```
[build-rpi]$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                                path
=====
meta                                /home/testing/Build/Lab/Yocto-lab/poky/r
meta-poky                           /home/testing/Build/Lab/Yocto-lab/poky/r
meta-yocto-bsp                       /home/testing/Build/Lab/Yocto-lab/poky/r
```

Trois *layers* sont déjà préconfigurés dans notre image. Ils se trouvent tous les trois dans le dossier «*poky/*» téléchargés initialement. Nous pouvons également observer une valeur de priorité, qui indique l'ordre de prise en charge des *layers* (nous examinerons cela plus en détail ultérieurement).

Ajoutons le *layer* spécifique pour Raspberry Pi en utilisant l'option «**add-layer**» de l'outil «**bitbake-layers**».

```
[build-rpi]$ bitbake-layers add-layer ../meta-raspberrypi/
NOTE: Starting bitbake server...
[build-rpi]$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                                path
=====
meta                                /home/testing/Build/Lab/Yocto-lab/poky/r
meta-poky                           /home/testing/Build/Lab/Yocto-lab/poky/r
meta-yocto-bsp                       /home/testing/Build/Lab/Yocto-lab/poky/r
meta-raspberrypi                     /home/testing/Build/Lab/Yocto-lab/meta-r
```

Nous voyons bien que le nouveau *layer* a été ajouté à la liste. Sa priorité est plus élevée que celles des autres. Son contenu sera donc analysé *après* celui des autres *layers*. Les fichiers de recette qu'il contient pourront donc surcharger les précédents et ajuster la configuration avant la compilation proprement dite.

Où cette liste de *layers* est-elle stockée ? Nous avons vu que le répertoire de

compilation ne contient pas beaucoup de fichiers de configuration. Pourtant l'un d'eux peut attirer notre attention :

```
[build-rpi]$ ls conf/  
bblayers.conf  local.conf  templateconf.cfg
```

Le fichier «bblayers» (bb représentant bitbake) contient ceci :

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf  
# changes incompatibly  
POKY_BBLAYERS_CONF_VERSION = "2"  
  
BBPATH = "${TOPDIR}"  
BBFILES ?= ""  
  
BBLAYERS ?= " \  
    /home/cpb/Yocto-lab/poky/meta \  
    /home/cpb/Yocto-lab/poky/meta-poky \  
    /home/cpb/Yocto-lab/poky/meta-yocto-bsp \  
    /home/cpb/Yocto-lab/meta-raspberrypi \  
    "
```

Les premières lignes configurent une variable (POKY_BBLAYERS_CONF_VERSION) à usage interne de Poky, qui ne nous concerne pas. Nous voyons que **la variable BBLAYERS**, est renseignée (si ce n'est fait auparavant) avec les chemins vers les répertoires des *layers*.

Nous aurions très bien pu rajouter manuellement la dernière ligne plutôt qu'appeler «bitbake-layers», mais l'avantage d'utiliser ce dernier est qu'il vérifie la cohérence de la configuration.

Voyons les versions de Raspberry Pi connues par le *layer* que nous avons téléchargé :

```
[build-rpi]$ls ../meta-raspberrypi/conf/machine/
```

```
include      raspberrypi2.conf      raspberrypi4-64.
raspberrypi0.conf      raspberrypi3-64.conf      raspberrypi4.cor
raspberrypi0-wifi.conf      raspberrypi3.conf      raspberrypi-cm3.
```

Nom	Cible
raspberrypi	Les premiers modèles de Raspberry Pi B et B+
raspberrypi2	Le Raspberry Pi modèle 2
raspberrypi3	Les Raspberry Pi 3 et 3B+, compilation 32 bits
raspberrypi3-64	Les Raspberry Pi 3 et 3B+, compilation 64 bits
raspberrypi4	Le Raspberry Pi 4, compilation 32 bits
raspberrypi4-64	Le Raspberry Pi 4, compilation 64 bits
raspberrypi3	Les Raspberry Pi 3 et 3B+, compilation 32 bits
raspberrypi-cm	Le premier Raspberry Pi <i>Compute Module</i>
raspberrypi-cm3	Le Raspberry Pi <i>Compute Module 3</i>
raspberrypi0	Le Raspberry Pi Zéro initial
raspberrypi0-wifi	Le second Raspberry Pi Zéro, avec wifi

Je lance une compilation pour le Raspberry Pi 4, à ce jour le petit dernier de la gamme (et le plus puissant !) :


```
[build-rpi]$ MACHINE=raspberrypi4 bitbake core-image-minimal
Parsing recipes: 100% |#####|
Parsing of 801 .bb files complete (0 cached, 801 parsed). 1327
NOTE: Resolving any missing task queue dependencies
```

Build Configuration:

```
BB_VERSION           = "1.44.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-18.04"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "raspberrypi4"
DISTRO               = "poky"
DISTRO_VERSION        = "3.0.1"
TUNE_FEATURES        = "arm vfp cortexa7 neon vfpv4 thumb call
TARGET_FPU           = "hard"
meta
meta-poky
meta-yocto-bsp        = "HEAD:12a4c177bb541b3187c7a54d5804f30c3
meta-raspberrypi     = "zeus:cee2557dc872ddaf721e6badb981c7772
```

Tout comme nous l'avons fait avec la carte *BeagleBone Black* précédemment, l'image produite est copiée sur une carte micro-SD que l'on prend soin de démonter auparavant. Le fichier à copier a une extension «.rpi-sdimg» (et non pas «.wic» comme c'est souvent le cas pour d'autres machines).

```
[build-rpi]$ ls tmp/deploy/images/raspberrypi4/
[...]
core-image-minimal-raspberrypi4.rpi-sdimg
[...]
[build-rpi]$ umount /dev/sdb?
[build-rpi]$ sudo cp tmp/deploy/images/raspberrypi4/core-image-minimal-raspberrypi4.rpi-sdimg /dev/sdb
[build-rpi]$
```

On peut assister au *boot* classique du Raspberry Pi sur l'une des **sorties micro-HDMI**.



```
[ 1.040566] mmc-bcm2835 fe300000.mmcnr: DMA channel allocated
[ 1.068264] sdhci-iproc fe340000.emmc2: Linked as a consumer to regulator.1
[ 1.071561] usb 1-1: new high-speed USB device number 2 using xhci_hcd
[ 1.090686] mmc1: queuing unknown CIS tuple 0x80 (2 bytes)
[ 1.094017] mmc1: queuing unknown CIS tuple 0x80 (3 bytes)
[ 1.097286] mmc1: queuing unknown CIS tuple 0x80 (3 bytes)
[ 1.101821] mmc1: queuing unknown CIS tuple 0x80 (7 bytes)
[ 1.105020] mmc1: queuing unknown CIS tuple 0x80 (3 bytes)
[ 1.106614] mmc0: SDHCI controller on fe340000.emmc2 [fe340000.emmc2] using ADMA
[ 1.110837] of_cfs_init
[ 1.112698] of_cfs_init: OK
[ 1.115005] Waiting for root device /dev/mmcblk0p2...
[ 1.162824] random: fast init done
[ 1.212145] mmc0: new ultra high speed DDR50 SDHC card at address aaaa
[ 1.214877] mmcblk0: mmc0:aaaa SL08G 7.40 GiB
[ 1.218237] mmcblk0: p1 p2
[ 1.223949] mmc1: new high speed SDIO card at address 0001
[ 1.240221] EXT4-fs (mmcblk0p2): mounted filesystem with ordered data mode. Opts: (null)
[ 1.241983] VFS: Mounted root (ext4 filesystem) readonly on device 179:2.
[ 1.244006] devtmpfs: mounted
[ 1.252019] Freeing unused kernel memory: 2048K
[ 1.254810] usb 1-1: New USB device found, idVendor=2109, idProduct=3431, bcdDevice= 4.2
[ 1.256461] usb 1-1: New USB device strings: Mfr=0, Product=1, SerialNumber=0
[ 1.258100] usb 1-1: Product: USB2.0 Hub
[ 1.261140] hub 1-1:1.0: USB hub found
[ 1.263018] hub 1-1:1.0: 4 ports detected
[ 1.271780] Run /sbin/init as init process
INIT: version 2.88 booting

Please wait: booting...
Starting udev
[ 1.523468] udevd[113]: starting version 3.2.8
[ 1.526186] random: udevd: uninitialized urandom read (16 bytes read)
[ 1.528744] random: udevd: uninitialized urandom read (16 bytes read)
[ 1.530415] random: udevd: uninitialized urandom read (16 bytes read)
[ 1.554435] udevd[114]: starting udevd-3.2.8
[ 1.758665] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
Thu Dec 26 12:49:48 UTC 2019
INIT: Entering runlevel: 5
Configuring network interfaces... [ 2.157357] bcmgenet: Skipping UMAC reset
[ 2.252149] bcmgenet fd580000.genet: configuring instance for external RGMII (no delay)
udhcpc: started, v1.31.0
udhcpc: sending discover
[ 3.271762] bcmgenet fd580000.genet eth0: Link is Down
udhcpc: sending discover
[ 7.431765] bcmgenet fd580000.genet eth0: Link is Up - 1Gbps/Full - flow control rx/tx
udhcpc: sending discover
udhcpc: sending select for 192.168.3.11
udhcpc: lease of 192.168.3.11 obtained, lease time 43200
/etc/udhcpc.d/50default: Adding DNS 192.168.3.254
done.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 3.0.1 raspberrypi4 /dev/tty1

raspberrypi4 login: _
```

On peut également se connecter sur son port série en utilisant minicom sur la machine hôte.

Dans le cas des Raspberry Pi 3 ou 4, il est nécessaire pour cela d'éditer le fichier «config.txt» se trouvant sur la première partition de la carte SD pour y ajouter la ligne «enable_uart=1». Ceci peut être réalisé automatiquement par Yocto si on ajoute la ligne «ENABLE_UART="1"» dans le fichier conf/local/conf.

```
Poky (Yocto Project Reference Distro) 3.0.1 raspberrypi4 /dev/
```

```
raspberrypi4 login: root
```

```
root@raspberrypi4:~# uname -a
```

```
Linux raspberrypi4 4.19.75 #1 SMP Thu Dec 26 11:13:52 UTC 2019
```

```
root@raspberrypi4:~# cat /proc/cpuinfo
```

```
processor          : 0
model name        : ARMv7 Processor rev 3 (v7l)
BogoMIPS          : 108.00
Features          : half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer   : 0x41
CPU architecture : 7
CPU variant       : 0x0
CPU part          : 0xd08
CPU revision      : 3
```

```
processor          : 1
model name        : ARMv7 Processor rev 3 (v7l)
BogoMIPS          : 108.00
Features          : half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer   : 0x41
CPU architecture : 7
CPU variant       : 0x0
CPU part          : 0xd08
CPU revision      : 3
```

```
processor          : 2
model name        : ARMv7 Processor rev 3 (v7l)
BogoMIPS          : 108.00
Features          : half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer   : 0x41
CPU architecture : 7
```

```
CPU variant      : 0x0
CPU part         : 0xd08
CPU revision     : 3

processor        : 3
model name       : ARMv7 Processor rev 3 (v7l)
BogoMIPS         : 108.00
Features         : half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer  : 0x41
CPU architecture: 7
CPU variant      : 0x0
CPU part         : 0xd08
CPU revision     : 3

Hardware         : BCM2835
Revision         : b03111
Serial           : 1000000079d9d08b
Model            : Raspberry Pi 4 Model B Rev 1.1
root@raspberrypi4:~# cat /sys/firmware/devicetree/base/model
Raspberry Pi 4 Model B Rev 1.1
```

Conclusion

Cette séquence nous a permis de créer des images standards de Yocto pour différentes cibles et de les tester. Le passage de l'émulateur à une carte réelle est important. Cela permet de s'assurer de la disponibilité des outils nécessaires (notamment lorsqu'il faut employer un utilitaire spécifique pour placer le code en mémoire *Flash Nand*) et de la maîtrise des techniques employées.

Les temps de compilation que nous avons observés sont vraiment très longs (pas loin de dix heures de compilation entre cette séquence et la précédente). Ceci ne concerne que la première compilation pour une plateforme donnée. À partir de maintenant, nous observerons des temps de compilation beaucoup plus raisonnables !

Pour le moment nous n'avons pas du tout personnalisé notre image. Nous allons y remédier dans les séquences suivantes...



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

»»