Yocto pour Raspberry pi 4 B 64 bits

Table of Contents

Yocto est un projet open source collaboratif de la Linux Foundation dont l'objectif est de produire des outils et des processus permettant la création de distributions Linux pour les logiciels embarqués et IoT qui sont indépendantes de l'architecture sous-jacente du matériel embarqué. L'objectif du projet est d'améliorer leyo processus de développement logiciel pour les distributions Linux embarquées.

Yocto fournit des outils, des métadonnées et des processus interopérables qui permettent le développement rapide et reproductible de systèmes embarqués basés sur Linux dans lesquels chaque aspect du processus de développement peut être personnalisé.

En utilisant **Yocto**, on peut créer des images Linux personnalisées pour les appareils embarqués. Ce tutoriel présente la création d'une image minimale de base pour raspberry pi 4b. Avec la petite modification dans le fichier de configuration, on peut créer les images pour différentes versions de Raspberry pi.

Le projet Yocto fournit également 2 images OS Linux basées sur Yocto pour différentes études de cas :
* **OS Linux basé sur Yocto: version Lite** - une image Yocto légère qui ne pèse que 60 Mo et comprend les packages de base nécessaires dans un produit IoT « à mission unique », y compris : les éditeurs Vim et Nano, lsmode, ping, ifconfig, systemd gestionnaire de service et busybox réduit.
* **Système d'exploitation Linux basé sur Yocto: version complète** - une image Yocto légère qui ne pèse que 75 Mo et comprend des packages de base ainsi que des packages communs supplémentaires utilisés dans de nombreux projets IoT, notamment : les éditeurs Vim et Nano, systemd service manager, lsmode, ping, ifconfig, busybox ET Python3, Cron, iptables et SSH.

Au niveau matériel, compter minimum un processeur 3 cores (cas des machines virtuelles par exemple) avec au moins 4 Go de RAM (les compilations sont gourmandes en ressources). Enfin prévoir un espace disque minimum de 150 Go. Des modifications sont possibles pour réduire l'empattement en mémoire.

Préparation de l'hôte Linux

La commande suivante permet d'installer le nécessaire à l'utilisation de yocto :

```
sudo apt-get install gawk wget git-core diffstat unzip \
                     texinfo gcc-multilib build-essential chrpath socat \
                     libsdl1.2-dev xterm  python
```

En complément, les librairies et outils suivants peuvent être installés :

```
sudo apt-get install  cpio python3 python3-pip python3-pexpect \
xz-utils debianutils iputils-ping gitk libhighgui-dev
```

Installation de yocto

L'installation de yocto est relativement simple, il suffit de télécharger le dépôt et d'aligner les branches sur le bon SHA1 dans git.

Dans un premier temps, se placer dans un dossier pour le téléchargement (par exemple /home/Views/):

```
git clone git://git.yoctoproject.org/poky
cd poky
git checkout master
cd ..
git clone https://github.com/openembedded/meta-openembedded.git
cd meta-openembedded
git checkout master
cd ..
```

Il est possible de télécharger une version spécifique, par exemple la version zeus de Yocto.

```
git clone -b zeus git://git.yoctoproject.org/poky.git poky-zeus
cd poky-zeus
git clone -b zeus git://git.openembedded.org/meta-openembedded
```

Configuration et compilation d'une image de base

Ajout d'un nouveau meta, exemple du meta-raspberry

Afin de pouvoir compiler sur la raspberry, il faut ajouter le layer correspondant. Ce layer contient tout le nécessaire pour réaliser des cross compilation vers la raspberry pi, quelque soit la version. Il suffit de spécifier la machine cible, comme on le verra par la suite.

Ce dossier s'ajoute très simplement comme lors du téléchargement de yocto. (On le placera dans le dossier /home/Views/poky)

```
git clone git://git.yoctoproject.org/meta-raspberrypi
cd meta-raspberry
git checkout 79ea44b997a6eb6e2f0d36b1d583c85fa9e37f36
cd ..
```

Il est possible de télécharger une version spécifique, par exemple la version zeus de Yocto.

`git clone -b zeus git://git.yoctoproject.org/meta-raspberrypi`

Pour avoir le support **qt**[1)] télécharger meta-qt5

```
git clone -b zeus https://github.com/meta-qt5/meta-qt5
```

On a maintenant toutes les couches nécessaires pour construire le Yocto. Il faut à présent le configurer pour compiler une image utilisable par la raspberry.

Configuration de yocto

Pour ce faire, on va utiliser le script `oe-init-build-env` présent dans la racine du dossier poky pour initialiser les variables du système de build **Poky**, et créer une structure de répertoire adaptée:

1. **build** – répertoire de construction
2. **source** – code source des recettes d'assemblage
3. **download** – répertoire pour télécharger le code du programme (bases de données git, archives tar.gz)

```
source oe-init-build-env
```

On peut éventuellement donner le dossier de construction comme argument, `source`

```
oe-init-build-env <build_folder_name>
```

Dans le dossier de construction, on peut maintenant voir un dossier appelé `conf` qui contiendra les fichiers de configuration

```
conf/
 ├── bblayers.conf
 ├── local.conf
 └── templateconf.cfg
```

Ce script va créer un dossier build et amènera dedans par la même occasion.

C'est dans le dossier conf de ce dossier build que l'on va travailler.

Plus spécifiquement, on va modifier les fichiers `bblayers.conf` et `local.conf`.

Ainsi le fichier `bblayer.conf` est écrit ainsi (à adapter par rapport à l'install) :

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"
BBPATH = "${TOPDIR}"
BBFILES ?= ""
BBLAYERS ?= " \
/home/macross/Views/poky/meta \
/home/macross/Views/poky/meta-poky \
/home/macross/Views/poky/meta-yocto-bsp \
/home/macross/Views/poky/meta-raspberrypi \
/home/macross/Views/poky/meta-openembedded/meta-oe \
/home/macross/Views/poky/meta-openembedded/meta-python \
/home/macross/Views/poky/meta-openembedded/meta-multimedia \
/home/macross/Views/poky/meta-openembedded/meta-networking \
/home/macross/Views/poky/meta-openembedded/meta-filesystems \
"
```

Pour la version zeus `conf/bblayers.conf` ressemblera à ceci:

```
POKY_BBLAYERS_CONF_VERSION="2"
BBPATH="${TOPDIR}"
BBFILES?=""
BBLAYERS?=" \
/home/eclabs/Yocto/poky-zeus/meta \
/home/eclabs/Yocto/poky-zeus/meta-poky \
/home/eclabs/Yocto/poky-zeus/meta-yocto-bsp \
/home/eclabs/Yocto/poky-zeus/meta-raspberrypi \
"
```

Concernant le fichier `local.conf`, les modifications sont plus importantes.

Tout d'abord, il faut rechercher la ligne où est indiqué la machine à compiler. Par défaut, elle est du type X86. Il faut la remplacer par la version de la raspberry pi qu'on souhaite compiler (voir documentations dans le meta associé). Dans le cas traité c'est soit la V1 B ou la V2 que l'on va utiliser.

```
MACHINE ??= "raspberrypi2"
```

On peut construire une image Yocto pour les différentes versions matérielles de Raspberry pi en changeant simplement cette variable **MACHINE**.

1. raspberrypi0
2. raspberrypi0w
3. raspberrypi3
4. raspberrypi3-64
5. raspberrypi4
6. raspberrypi4-64 etc

Pour une image minimale, on peut utiliser la commande suivante:

`bitbake core-image-minimal`

Il est préférable de regrouper les modifications apportées à «local.conf» vers le début de ce fichier en les encadrant clairement pour pouvoir identifier facilement les actions. En voici un exemple :

```
[...]
# <LOGILIN>

MACHINE = "qemuarm"
DL_DIR = "${TOPDIR}/../downloads"
SSTATE_DIR = "${TOPDIR}/../sstate-cache"

# </LOGILIN>
[...]
```

Bien sûr dès que la complexité des modifications dépasse quelques lignes, il devient préférable de les suivre par l'intermédiaire d'un système de gestion de versions comme git.

Les variables renseignées ici sont les suivantes :

- **MACHINE**: comme on l'a vu dans la séquence précédente, ceci permet de sélectionner la plateforme cible. Ici, une émulation de carte ARM par Qemu.
- **DL_DIR**: lorsque Yocto télécharge des packages de fichiers sources pour les compiler, il les stocke dans ce répertoire en vérifiant au préalable s'ils n'ont pas déjà été chargés. Par défaut il s'agit du sous-dossier «downloads/» dans le répertoire de compilation (variable TOPDIR). On a inséré ici un «../» pour remonter le répertoire de stockage des fichiers téléchargés un cran au-

dessus, à côté de poky/ et de build-qemu/. L'intérêt sera de conserver ce répertoire pour le mettre en commun avec d'autres builds pour d'autres cibles.
- **SSTATE_DIR**: comme **DL_DIR**, il s'agit d'un répertoire de stockage. Lorsque Yocto progresse dans ses compilations il stocke des fichiers temporaires, des fichiers objets, etc. dans ce dossier. Il peut ensuite les retrouver pour accélérer (très sensiblement) les builds suivants. Comme précédemment, on a remonté le répertoire d'un «../» par rapport à sa situation par défaut pour le partager avec d'autres builds éventuels.
- On note également la présence d'une autre variable qui peut s'avérer utile dans certaines situations : **BB_NUMBER_THREADS**. Yocto est très gourmand. Autant en place mémoire ou disque qu'en ressource CPU. Lorsqu'un build s'exécute, tous les coeurs de processeur sont sollicités à 100%. Si on doit utiliser la machine de compilation pour un autre travail pendant la compilation, le système va manquer sérieusement de fluidité. On peut donc restreindre le nombre de jobs que bitbake lance simultanément pour le rendre moins agressif, en indiquant par exemple **«BB_NUMBER_THREADS="2"»** pour se limiter à deux tâches en parallèle au prix d'un allongement inévitable du temps de compilation.

Les modifications plus importantes vont se faire par l'ajout de ligne à la fin du fichier `local.conf`.

```
LICENSE_FLAGS_WHITELIST = "commercial"
ENABLE_UART = "1"
INHERIT += "rm_work"
EXTRA_IMAGE_FEATURES += " package-management"
RM_OLD_IMAGE = "1"
IMAGE_FSTYPES = "tar.bz2 rpi-sdimg"
RM_WORK_EXCLUDE += " opencv"
```

Une autree étape de personnalisation concerne le nom de la machine. Visible dans le prompt il vaut par défaut le contenu de la variable **MACHINE**. On peut le personnaliser en ajoutant la ligne suivante dans `local.conf` :

```
hostname_pn-base-files = "mybox"
```

La syntaxe de cette ligne est plus complexe qu'elle en a l'air. Lorsque bitake analyse les recettes et les fichiers de configuration, il lit cette ligne ainsi : «Lors du traitement de la recette dont le nom de package (**pn** pour Package Name) est **base-files**, configurer la variable hostname avec la valeur mybox».

En règle générale, dans une affectation de variable, le caractère «souligné» (underscore) «_» permet de préciser ou restreindre la portée de l'opération. La chaîne de caractères **_pn** peut être imaginée comme une sorte d'opérateur pour restreindre la modification de l'affectation de la variable à la recette dont le nom est «base-files».

Comme son nom l'indique, la recette **base-files**, livrée avec Poky, fournit un ensemble de fichiers de configuration standards comme `/etc/fstab`, `/etc/host.conf`, `/etc/shells`, etc. En outre elle fournit des méthodes pour personnaliser **hostname**, et proposer un message de bienvenue «Poky (Yocto Project Reference Distro) 3.3.1 qemuarm /dev/tty1» avant l'invite de connexion.

Ajout d'un nouveau programme à l'image

Afin d'ajouter un programme à l'image qu'on va générer, on peut se rendre dans `meta-`

`raspberry/recipes-core/image` afin d'éditer le fichier de l'image que l'on souhaite compiler. Dans le cas, il s'agira de l'image **rpi-basic-image** qui se trouve dans le fichier `rpi-test-image.bb`.

On va ajouter les lignes suivantes à la fin du fichier :

```
IMAGE_INSTALL_append += " packagegroup-core-ssh-openssh openssh-sftp-server"
IMAGE_INSTALL_append += " opencv opencv-samples"
```

Ces différents éléments permettent d'ajouter **ssh**, **opencv** et **nodejs** respectivement dans l'image qu'on va compiler.

On peut trouver un layer (un ensemble de recettes) à installer sur le site https://layers.openembedded.org, de cliquer sur l'onglet «recipes» et de saisir le nom du package désiré dans le moteur de recherche.

L'ensemble «meta-openembedded» est assez incontournable dès lors que l'on prépare un système avec Yocto. Il contient près de deux mille recettes pour de nombreux packages très utiles pour Linux embarqué.

Compilation et copie du résultat sur carte SD

Afin de compiler l'image il faut se trouver dans le dossier build via le script `oe-init-build-env`.

à présent on peut lancer la commande suivante et compter plusieurs heures de compilation même avec une bonne machine. Ceci est dû au temps de récupération des sources très long.

```
bitbake rpi-test-image -k
```

Le **-k** permet de compiler un maximum de choses même en cas d'erreur.

Il créera une image du noyau, RFS (Root File System), dtbs (device tree binary) et les codes de démarrage. Pour la première fois, la construction prendra un certain temps (on construit un Linux à partir des sources).

Une fois la compilation terminé. L'image de la carte SD se trouve dans le dossier `build/tmp/deploy/images//*.rpi-sdimg`

Graver l'image sur la carte SD en utilisant une des méthodes décrites ici et tenter de booter sur la raspberry.

Via ssh, le nom de connexion est root. pas de mot passe. De même en connexion directe.

Production d'images pour des cibles spécifiques

Dans la séquence précédente nous avons créé une image standard pour un Raspberry Pi. On va aller à présent sur quelques autres cibles possibles : émulateur Arm, cartes Beaglebone Black et Raspberry Pi. 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 qu'on a 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, on observera des temps de compilation beaucoup plus raisonnables !

Variable d'environnement MACHINE

Pour connaitre la cible pour laquelle on souhaite préparer une image **yocto** s'appuie sur la variable d'environnement «MACHINE». Celle-ci doit contenir le nom d'une cible connue. Lorsqu'on utilise simplement les layers de Poky, comme nous l'a 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 Single Board Computers 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

On peut trouver la liste de ces cibles dans les sous-répertoires `poky/meta/conf/machine/` et `poky/meta-yocto-bsp/conf/machine/`.

On peut également les trouver au début du fichier `conf/local.conf` qui a été automatiquement créé lorsqu'on a 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. There are
a selection
# of emulated machines available which can boot and run in the QEMU
emulator:
#
#MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemumips64"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemux86"
#MACHINE ?= "qemux86-64"
#
# There are also the following hardware board target machines included for
# 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 is
selected:
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 préparer une image pouvant être prise en charge par l'émulateur **Qemu-x86**.

Le symbole «??=» dans l'affectation de la variable (`MACHINE ??= "qemux86"`) signifie que la variable n'est affectée que si elle n'a pas de valeur préalable.

Autrement dit, on peut remplir la variable avant de lancer «bitbake» et cette affectation aura précédence sur celle par défaut indiqué dans «conf/local.conf».

Dans les prochaines étapes nous inscrira le nom de la machine cible désirée dans ce fichier, mais pour le moment, on se contentera d'interagir uniquement depuis la ligne de commande.

Image pour émulateur ARM

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

compilation...).

```
[build-qemu]$ MACHINE=qemuarm  bitbake  core-image-minimal
Loading cache: 100% |
| ETA:  --:--:--
Loaded 0 entries from dependency cache.
Parsing recipes: 100%
|##################################################################################
#################################################################################
#| Time: 0:00:43
Parsing of 814 .bb files complete (0 cached, 814 parsed). 1438 targets, 59
skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.50.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "qemuarm"
DISTRO               = "poky"
DISTRO_VERSION       = "3.3.1"
TUNE_FEATURES        = "arm armv7ve vfp thumb neon callconvention-hard"
TARGET_FPU           = "hard"
meta
meta-poky
meta-yocto-bsp       = "HEAD:05a8aad57ce250b124db16705acec557819905ae"

Initialising tasks: 100%
|##################################################################################
#################################################################################|
Time: 0:00:01
Sstate summary: Wanted 663 Local 18 Network 0 Missed 645 Current 429 (2%
match, 40% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3091 tasks of which 1596 didn't need to be
rerun and all succeeded.
```

Lorsque bitbake s'arrête de travailler, on peut voir la nouvelle image :

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

[build-qemu]$ ls tmp/deploy/images/qemuarm/
core-image-minimal-qemuarm-20210708174346.qemuboot.conf
core-image-minimal-qemuarm-20210708174346.rootfs.ext4
core-image-minimal-qemuarm-20210708174346.rootfs.manifest
core-image-minimal-qemuarm-20210708174346.rootfs.tar.bz2
core-image-minimal-qemuarm-20210708174346.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--5.10.34+git0+38eb7ca3f4_78e8e722ee-r0-qemuarm-20210708174346.tgz
modules-qemuarm.tgz
zImage
zImage--5.10.34+git0+38eb7ca3f4_78e8e722ee-r0-qemuarm-20210708174346.bin
zImage-qemuarm.bin

[build-qemu]$
```

Il faut donc installer une version adaptée de Qemu pour processeur Arm.

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

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

La commande «uname -a» dans la fenêtre de l'émulateur doit indiquer que l'architecture est bien de type ARM.

Image sur beaglebone

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. On peut relancer un build pour cette architecture. Comme la cible ne sera plus l'émulateur Qemu pour le moment, il est préfèrable de recréer un nouveau répertoire de compilation à côté de «build-quemu».

Par exemple **build-bbb** comme abréviation de Beagle Bone Black, la carte qu'on veut utiliser.

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

On relance une compilation avec une nouvelle architecture cible.

```
[build-bbb]$ MACHINE=beaglebone-yocto  bitbake  core-image-minimal
Loading cache: 100% |
| ETA:  --:--:--
Loaded 0 entries from dependency cache.
```

```
Parsing recipes: 100%
|#####################################################################
#####################################################################
#| Time: 0:00:50
Parsing of 814 .bb files complete (0 cached, 814 parsed). 1438 targets, 59
skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.50.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-20.04"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "beaglebone-yocto"
DISTRO               = "poky"
DISTRO_VERSION       = "3.3.1"
TUNE_FEATURES        = "arm vfp cortexa8 neon callconvention-hard"
TARGET_FPU           = "hard"
meta
meta-poky
meta-yocto-bsp       = "HEAD:05a8aad57ce250b124db16705acec557819905ae"
```

Après un nouveau moment de compilation, on peut observer les images produites :

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

Ici n'apparaissent que le fichier qui vont 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 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».

Insèrer sur le PC hôte une carte micro-SD par l'intermédiaire d'un adaptateur USB et examiner les périphériques blocs disponibles.

```
[build-bbb]$ lsblk
 NAME           MAJ:MIN RM   SIZE RO TYPE  MOUNTPOINT
 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 il faut la démonter pour la détacher du système de fichiers.

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

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

```
[build-bbb]$ sudo  cp  tmp/deploy/images/beaglebone-yocto/core-image-
minimal-beaglebone-yocto.wic  /dev/sdb
[build-bbb]$
```

On peut à présent extraire la carte micro-SD du PC hôte, 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).

On peut alors se connecter sur la Beaglebone par l'intermédiaire d'un adaptateur USB-Série (les trois fils jaune, rouge et noir).

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

```
U-Boot SPL 2021.01 (Jan 11 2021 - 18:11:43 +0000)
Trying to boot from MMC1

U-Boot 2021.01 (Jan 11 2021 - 18:11:43 +0000)

CPU  : AM335X-GP rev 2.1
Model: TI AM335x BeagleBone Black
DRAM:  512 MiB
WDT:   Started with servicing (60s timeout)
NAND:  0 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
Loading Environment from FAT... *** Warning - bad CRC, using default
environment

 not set. Validating first E-fuse MAC
Net:   Could not get PHY for ethernet@4a100000: addr 0
eth2: ethernet@4a100000, eth3: usb_ether
Hit any key to stop autoboot:  0
switch to partitions #0, OK
mmc0 is current device
```

```
SD/MMC found on device 0
Failed to load 'boot.scr'
Failed to load 'uEnv.txt'
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 3 ms (38.1 KiB/s)
1:      Yocto
Retrieving file: /zImage
7568576 bytes read in 484 ms (14.9 MiB/s)
append: root=PARTUUID=a2e742e0-02 rootwait console=ttyS0,115200
Retrieving file: /am335x-boneblack.dtb
62633 bytes read in 6 ms (10 MiB/s)
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Device Tree to 8ffed000, end 8ffff4a8 ... OK

Starting kernel ...
```

Le bootloader U-boot 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 5.10.21-yocto-standard (oe-user@oe-host) (arm-poky-linux-
gnueabi-gcc (GCC) 10.21
CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: TI AM335x BeagleBone Black
Memory policy: Data cache writeback
cma: Reserved 16 MiB at 0x9e800000
Zone ranges:
  Normal   [mem 0x0000000080000000-0x000000009fefffff]
  HighMem  empty
Movable zone start for each node
Early memory node ranges
  node   0: [mem 0x0000000080000000-0x000000009fefffff]
Initmem setup node 0 [mem 0x0000000080000000-0x000000009fefffff]
CPU: All CPU(s) started in SVC mode.
  [...]

  [...]
VFS: Mounted root (ext4 filesystem) readonly on device 179:26.
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 2.99 booting
Starting udev
udevd[121]: starting version 3.2.10
udevd[122]: starting eudev-3.2.10
EXT4-fs (mmcblk1p2): re-mounted. Opts: (null)
Fri Mar  9 12:34:56 UTC 2018
Configuring packages on first boot....
 (This may take several minutes. Please do not power off the machine.)
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
 Removing any system startup links for run-postinsts ...
  /etc/rcS.d/S99run-postinsts
INIT: Entering runlevel: 5
  [...]
udhcpc: sending discover
udhcpc: sending discover
udhcpc: no lease, forking to background
done.
Starting syslogd/klogd: done
```

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

```
beaglebone-yocto login: root
root@beaglebone-yocto:~# uname -a
Linux beaglebone-yocto 5.10.21-yocto-standard #1 PREEMPT Tue Mar 9 04:04:38
UTC 2021 armv7l Gnu Linuxx
root@beaglebone-yocto:~# cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 2 (v7l)
BogoMIPS        : 996.14
Features        : half thumb fastmult vfp edsp thumbee neon vfpv3 tls vfpd32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x3
CPU part        : 0xc08
CPU revision    : 2

Hardware        : Generic AM33XX (Flattened Device Tree)
Revision        : 0000
Serial          : 6000BBBK7210
root@beaglebone-yocto:~# cat /sys/firmware/devicetree/base/model
TI AM335x BeagleBone Black root@beaglebone-yocto:~#
```

Image pour Raspberry Pi

Bien entendu Yocto permet de générer une image pour Raspberry Pi. Toutefois, il 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-».

On peut télécharger les layers «à côté» du répertoire poky/. Mais rien n'y oblige, et il est possible de les regrouper dans un sous-dossier spécifique si on le préfère.

Nous verrq 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-raspberrypi  -b
hardknott
Clonage dans 'meta-raspberrypi'...
remote: Enumerating objects: 8942, done.
remote: Counting objects: 100% (8942/8942), done.
remote: Compressing objects: 100% (4389/4389), done.
remote: Total 8942 (delta 5172), reused 7300 (delta 4144)
Réception d'objets: 100% (8942/8942), 1.80 Mio | 2.08 Mio/s, fait.
Résolution des deltas: 100% (5172/5172), fait.

[Yocto-lab]$ ls
build-bbb  build-qemu  meta-raspberrypi  poky

[Yocto-lab]$
```

On voit qu'un répertoire `meta-raspberrypi/` est bien apparu. On va préparer 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 therefore been
  [...]
[build-rpi]$
```

On doit commencer par ajouter le layer téléchargé plus haut dans la configuration actuelle. Il faut d'abord regarder la liste des layers déjà pris en compte pour le futur build avec la commande **bitbake-layers** et son option **show-layers**:

```
[build-rpi]$ bitbake-layers  show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/testing/Build/Lab/Yocto-lab/poky/meta  5
meta-poky             /home/testing/Build/Lab/Yocto-lab/poky/meta-poky  5
meta-yocto-bsp        /home/testing/Build/Lab/Yocto-lab/poky/meta-yocto-bsp
5
```

Trois layers sont déjà préconfigurés dans l'image. Ils se trouvent tous les trois dans le dossier «poky/» téléchargés initialement. On peut également observer une valeur de priorité, qui indique l'ordre de prise en charge des layers).

Ajouter 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                                      priority
==========================================================================
meta                  /home/testing/Build/Lab/Yocto-lab/poky/meta  5
meta-poky             /home/testing/Build/Lab/Yocto-lab/poky/meta-poky  5
meta-yocto-bsp        /home/testing/Build/Lab/Yocto-lab/poky/meta-yocto-bsp
5
meta-raspberrypi      /home/testing/Build/Lab/Yocto-lab/meta-raspberrypi  9
```

On peut voir 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 ? On a vu que le répertoire de compilation ne contient pas beaucoup de fichiers de configuration. Pourtant l'un d'eux peut attirer l'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/bblayers.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_CONV_VERSION** à usage interne de Poky, qui ne nous concerne pas. on voit que la variables **BBLAYERS**, est renseignée (si ce n'est fait auparavant) avec les chemins vers les répertoires des layers.

On aurait 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.

On peut voir les versions de Raspberry Pi connues par le layer que l'on a téléchargé :

```
[build-rpi]$ls  ../meta-raspberrypi/conf/machine/
include                 raspberrypi3-64.conf  raspberrypi-cm3.conf
raspberrypi0.conf       raspberrypi3.conf     raspberrypi-cm.conf
raspberrypi0-wifi.conf  raspberrypi4-64.conf  raspberrypi.conf
raspberrypi2.conf       raspberrypi4.conf
```

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
raspberrypi-cm	Le premier Raspberry Pi Compute Module
raspberrypi-cm3	Le Raspberry Pi Compute Module 3
raspberrypi0	Le Raspberry Pi Zéro initial
raspberrypi0-wifi	Le second Raspberry Pi Zéro, avec wifi

Lancer 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
Loading cache: 100% |
| ETA:  --:--:--
Loaded 0 entries from dependency cache.
Parsing recipes: 100%
|##################################################################################
##################################################################################
#| Time: 0:00:57
Parsing of 850 .bb files complete (0 cached, 850 parsed). 1474 targets, 66
skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.50.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-20.04"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "raspberrypi4"
DISTRO               = "poky"
DISTRO_VERSION       = "3.3.1"
TUNE_FEATURES        = "arm vfp cortexa7 neon vfpv4 thumb callconvention-
hard"
TARGET_FPU           = "hard"
meta
meta-poky
```

```
meta-yocto-bsp       = "HEAD:05a8aad57ce250b124db16705acec557819905ae"
meta-raspberrypi     = "hardknott:064f5404ea90f02bd15088de6317692098d9f770"
```

Tout comme on l'av 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 «.wic.bz2», on peut le décompresser au préalable puis le copier sur la carte SD, ou utiliser bzcat pour le décompresser à la volée :

```
[build-rpi]$ ls  tmp/deploy/images/raspberrypi4/
  [...]
core-image-minimal-raspberrypi4.rpi-sdimg
  [...]
[build-rpi]$ umount  /dev/sdb?
[build-rpi]$ sudo  sh
# bzcat tmp/deploy/images/raspberrypi4/core-image-minimal-
raspberrypi4-20210712055807.rootfs.wic.bz2 > /dev/sdc
# exit
[build-rpi]$
```

On peut assister au boot classique du Raspberry Pi sur l'une des sorties micro-HDMI (pour des raisons pratiques la photo ci-dessous date de la version précédente de Poky, mais le boot est identique avec la dernière version).

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`.

```
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 5.10.31-v7l (oe-user@oe-host) (arm-poky-linux-
gnueabi-gcc (GCC) 1
0.2.0, GNU ld (GNU Binutils) 2.36.1.20210209) #1 SMP Fri Apr 23 15:16:49 UTC
2021
[    0.000000] CPU: ARMv7 Processor [410fd083] revision 3 (ARMv7),
cr=30c5383d
[    0.000000] CPU: div instructions available: patching division code
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, PIPT instruction
cache
[    0.000000] OF: fdt: Machine model: Raspberry Pi 4 Model B Rev 1.1
   [...]
Starting syslogd/klogd: done
```

```
raspberrypi4 login: root
root@raspberrypi4:~# uname -a
Linux raspberrypi4 5.10.31-v7l #1 SMP Fri Apr 23 15:16:49 UTC 2021 armv7l
GNU/Linux
```

```
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 vfpv4 idiva
idivt vfpd32 lpae ev
tstrm crc32
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 vfpv4 idiva
idivt vfpd32 lpae ev
tstrm crc32
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 vfpv4 idiva
idivt vfpd32 lpae ev
tstrm crc32
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 vfpv4 idiva
idivt vfpd32 lpae ev
tstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

Hardware        : BCM2711
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
```

Utilisation des fonctionnalités d'image

Lorsqu'on demande à bitbake de produire une image, il recherche un fichier d'extension «.bb» avec le nom de l'image dans les sous-répertoires de Poky. Par exemple le fichier **«core-image-base»** est une recette avec l'extension «.bb».

```
[build-qemu]$ find  ../poky/  -name  core-image-base.bb
../poky/meta/recipes-core/images/core-image-base.bb
```

Ce fichier doit se trouver dans un sous-répertoire «images/» se trouvant dans un répertoire dont le nom commence par «recipes-»

```
[build-qemu]$ cat  ../poky/meta/recipes-core/images/core-image-base.bb
SUMMARY = "A console-only image that fully supports the target device \
 hardware."
IMAGE_FEATURES += "splash"
LICENSE = "MIT"
inherit core-image
```

Hormis la description et la licence, deux lignes intéressent :

- **«inherit core-image»**: la recette hérite d'une classe prédéfinie qui décrit le contenu d'une image en proposant des fonctionnalités optionnelles.
- **«IMAGE_FEATURES += "splash"»** ajoute la fonctionnalité splashscreen à l'image. Dans la définition de la classe core-image, une ligne («FEATURE_PACKAGES_splash = "psplash"») indique quel package doit être ajouté pour répondre à cette fonctionnalité.

On voit que Yocto propose ainsi ce mécanisme de features, des fonctionnalités de haut-niveau que l'on peut sélectionner sans se soucier du détail du package correspondant.

Voici les fonctionnalités proposées par la classe «core-image», qui est implémentée dans le fichier `poky/meta/classes/core-image.bbclass`.

x11	Serveur X-window
x11-base	Serveur X-window et environnement minimal (Matchbox)
x11-sato	Serveur X-window et environnement Sato pour mobile.
tools-debug	Outils de débogage (gdb, gdbserver, strace, gcore…)
eclipse-debug	Outils de débogage distant avec Eclipse.
tools-profile	Outils de profiling (perf, lttng, gprof…)
tools-testapps	Outils de tests du matériel.
tools-sdk	Installation des outils de développement natifs (gcc, make…) sur la cible.

nsf-server	Serveur NFS
nfs-client	Client NFS
ssh-server-dropbear	Serveur SSH basé sur Dropbear.
ssh-server-openssh	Serveur SSH basé sur Openssh
hwcodecs	Support des codecs pour l'accélération matérielle
package-management	Support des packages sur la cible (installation des outils et base de données).
dev-pkgs	Installation des fichiers headers nécessaire pour le développement des packages présents.
dbg-pkgs	Tous les packages sont compilés avec les informations de débogage
doc-pkgs	Installation de la documentation associée à tous les packages
read-only-rootfs	Système de fichiers en lecture-seule.
splash	Écran d'accueil pendant le boot

On peut, par exemple, ajouter dans le fichier local.conf la ligne :

```
IMAGE_FEATURES += "tools-sdk"
```

pour intégrer dans l'image les outils nécessaires à la compilation de code directement sur la cible.

Création d'une image spécifique

Jusqu'à présent on a utilisé l'image core-image-base en ajoutant dans local.conf des lignes «IMAGES_INSTALL_append». Cette approche est parfaitement adaptée pour les premières phases de configuration du système, mais trouve rapidement ses limites. Pour les systèmes industriels en effet, il est souvent nécessaire de gérer toute une gamme de produits différents. On est donc amenés à réaliser régulièrement des séries de builds avec peu de différences entre-eux. On préfère généralement factoriser toute la configuration commune aux différents produits dans une image particulière et ne laisser dans les fichiers local.conf que les spécificités propres à chaque build.

Autrement dit on va créer un fichier de description d'image, et on n'invoquera plus `bitbake core-image-base` mais `bitbake my-image` par exemple.

Pour cela, on doit commencer par créer un layer personalisé. Rien de compliqué, l'outil **bitbake-layers** est là pour aider.

```
[build-qemu]$ bitbake-layers  create-layer  ../meta-my-layer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer ../meta-my-layer'
[build-qemu]$ ls  ..
build-bbb  build-qemu  build-rpi  downloads  meta-my-layer  meta-
openembedded  meta-raspberrypi  poky  sstate-cache
[build-qemu]$
```

Le layer est bien créé mais, comme **bitbake-layers** l'affiche de manière un peu ambiguë, il n'est pas encore intégré dans la liste des layers que bitbake parcourera lors des builds. On doit l'y ajouter :

```
[build-qemu]$ bitbake-layers  show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/testing/Build/Lab/poky/meta   5
meta-poky             /home/testing/Build/Lab/poky/meta-poky  5
meta-yocto-bsp        /home/testing/Build/Lab/poky/meta-yocto-bsp  5
meta-oe               /home/testing/Build/Lab/meta-openembedded/meta-oe  6

[build-qemu]$ bitbake-layers  add-layer  ../meta-my-layer/
NOTE: Starting bitbake server...
[build-qemu]$ bitbake-layers  show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/testing/Build/Lab/poky/meta   5
meta-poky             /home/testing/Build/Lab/poky/meta-poky  5
meta-yocto-bsp        /home/testing/Build/Lab/poky/meta-yocto-bsp  5
meta-oe               /home/testing/Build/Lab/meta-openembedded/meta-oe  6
meta-my-layer         /home/testing/Build/Lab/meta-my-layer  6
[build-qemu]$
```

Créer dans le layer un début d'arborescence pour héberger nos recettes spécifiques. Son nom doit commencer par `recipes-` par exemple `recipes-custom`.

```
[build-qemu]$ mkdir  ../meta-my-layer/recipes-custom/
```

Il faut ensuite y créer un sous-répertoire nommé «images» pour stocker les recettes décrivant des images. Puis y copier le fichier `core-image-base.bb` pour avoir un point de départ.

```
[build-qemu]$ mkdir  ../meta-my-layer/recipes-custom/images/
[build-qemu]$ cp  ../poky/meta/recipes-core/images/core-image-base.bb
../meta-my-layer/recipes-custom/images/my-image.bb
```

En le copiant, on a renommé le fichier `core-image-base.bb` en `my-image.bb` dont le contenu est le suivant :

```
SUMMARY = "A customized image for development purposes."
LICENSE = "MIT"
inherit core-image
IMAGE_FEATURES += "splash"
IMAGE_FEATURES += "tools-debug"
IMAGE_FEATURES += "tools-profile"
IMAGE_FEATURES += "tools-sdk"
IMAGE_FEATURES += "ssh-server-dropbear"
IMAGE_INSTALL_append = " mc"
IMAGE_INSTALL_append = " nano"
```

Editer local.conf pour supprimer toutes les lignes «IMAGE_INSTALL_append» et «IMAGE_FEATURES» qu'on a ajouté auparavant.

On peut noter, dans la recette d'image ci-dessus la présence de deux syntaxes différentes pour ajouter une chaîne de caractères à la fin du contenu d'une variable :

- **«IMAGE_FEATURES += “splash”»**: ajoute la chaîne splash en la précédant automatiquement d'une espace
- **«IMAGE*INSTALL*append = “ mc”»**: ajoute la chaîne mc et l'espace que nous avons explicitement indiquée.

Mais la différence entre ces deux syntaxes ne se limite pas uniquement à cette histoire d'espace. Sinon on utiliserait toujours «+=» qui semble plus simple.

- Avec **+=** l'évaluation du contenu précédent, auquel ajouter la chaîne se fait immédiatement à l'analyse de la ligne. La variable **IMAGE_FEATURES** est initialisée avec une chaîne vide dans le fichier `poky/meta/classes/image.bbclass` chargé par héritage depuis le fichier `poky/meta/classes/core-image.bbclass`. Après lecture de la recette d'image, le contenu de **IMAGE_FEATURES** est donc exactement `splash tools-debug tools-profile tools-sdk ssh-server-dropbear`.
- Avec **_append**, l'évaluation du contenu de la chaîne avant ajout est différé et n'a lieu qu'une fois que bitbake a lu et initialisé toutes les variables d'environnement. **IMAGE_INSTALL** est initialisée dans `poky/meta/classes/core-image.bbclass` avec `"IMAGE_INSTALL ?= "${CORE_IMAGE_BASE_INSTALL}"`
- (la variable **CORE_IMAGE_BASE_INSTALL** étant remplie quelques lignes auparavant). L'affectation **?=** indique que la variable n'est initialisée que si elle n'existe pas au préalable. Si on avait utilisé **+=** pour ajouter mc et nano, l'initialisation de **IMAGE_INSTALL** dans core-image-base n'aurait pas eu lieu, celle-ci n'aurait contenu que “ mc nano” et le système aurait été inutilisable.

Savoir quand utiliser **+=** et quand préférer **_append** n'est pas simple quand il s'agit de compléter des variables initialisées dans des recettes fournies par Poky (ou par des layers supplémentaires, notamment pour le support du matériel). Il est souvent nécessaire d'aller jeter un œil sur l'implémentation des recettes, et il est bon de se familiariser avec l'arborescence de `poky/` et celle de `meta-openembedded/` par exemple.

[1)]
Qt est une API orientée objet et développée en C++, qui offre des composants d'interface graphique, d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, d'analyse XML, etc.