II.2 – Ajout d'applications dans l'image

Christophe BLAESS - janvier 2020

Dans la séquence précédente, nous avons modifié **le fichier «`local.conf`»** que nous trouvons dans le sous-répertoire «`conf/`» de notre répertoire de *build*. Les modifications étaient assez simples : ajustement de l'emplacement des dossiers de stockage temporaire, ajout d'utilisateur et configuration de mots de passe, puis modification du nom de machine.

Dans cette séquence nous allons ajouter quelques applications sur notre système. Pour cela, nous allons commencer en modifiant à nouveau ce fichier, puis nous créerons **notre propre recette d'image** personnalisée.

Utilitaires présents dans Poky

Notre première possibilité est d'ajouter des *packages* dont **les recettes sont livrées avec Poky**. Regardons, par exemple les fichiers se trouvant dans les sous-répertoires «`recipes*/`» de l'arborescence de Poky :

```
[build-qemu]$ ls  ../poky/meta/recipes*
```

```
../poky/meta/recipes.txt

../poky/meta/recipes-bsp:
acpid       apmd        efivar      gnu-efi  keymaps  lrzsz
alsa-state  efibootmgr  formfactor  grub     libacpi  opensbi

../poky/meta/recipes-connectivity:
avahi  bluez5   dhcp       iproute2  libnss-mdns  mobile-broad
bind   connman  inetutils  iw        libpcap      neard

../poky/meta/recipes-core:
base-files     coreutils  ell      glib-2.0         images
base-passwd    dbus       expat    glibc            init-ifupd
busybox        dbus-wait  fts      glib-networking  initrdscri
console-tools  dropbear   gettext  ifupdown         initscript

../poky/meta/recipes-devtools:
apt               chrpath             dosfstools     git
autoconf          cmake               dpkg           glide
autoconf-archive  createrepo-c        dwarfsrcfiles  gnu-confi
automake          dejagnu             e2fsprogs      go
binutils          desktop-file-utils  elfutils       help2man
bison             devel-config        expect         i2c-tools
bootchart2        diffstat            fdisk          icecc-cre
btrfs-tools       distcc              file           icecc-too
build-compare     dmidecode           flex           intltool
ccache            dnf                 gcc            json-c
cdrtools          docbook-xml         gdb            libcomps

../poky/meta/recipes-extended:
acpica     cpio          findutils    gzip        libidn
asciidoc   cracklib      foomatic     hdparm      libmnl
at         cronie        gawk         images      libnsl
bash       cups          ghostscript  iptables    libnss-nis
bc         cwautomacros  go-examples  iputils     libpipeline
blktool    diffutils     gperf        less        libsolv
bzip2      ed            grep         libaio      libtirpc
chkconfig  ethtool       groff        libarchive  lighttpd
```

```
../poky/meta/recipes-gnome:
epiphany  gdk-pixbuf  gobject-introspection      gtk+     hicc
gcr       gnome       gsettings-desktop-schemas  gtk-doc  jsor

../poky/meta/recipes-graphics:
builder          fontconfig  kmscube      matchbox-session  pa
cairo            freetype    libepoxy     matchbox-wm       pa
cantarell-fonts  glew        libfakekey   menu-cache        pi
clutter          harfbuzz    libmatchbox  mesa              pc
cogl             images      libsdl2      mini-x-session    st
drm              jpeg        libva        mx                tt

../poky/meta/recipes-kernel:
blktrace   dtc         kexec  linux           linux-libc-heade
cryptodev  kern-tools  kmod   linux-firmware  lttng

../poky/meta/recipes-multimedia:
alsa    flac       lame    libid3tag  libomxil  libsamplerate
ffmpeg  gstreamer  liba52  libogg     libpng    libsndfile

../poky/meta/recipes-rt:
images  README  rt-tests

../poky/meta/recipes-sato:
images               matchbox-desktop   matchbox-sato        p
l3afpad              matchbox-keyboard  matchbox-terminal    p
matchbox-config-gtk  matchbox-panel-2   matchbox-theme-sato  p

../poky/meta/recipes-support:
apr              consolekit   gnome-desktop-testing  libcap
argp-standalone  curl         gnupg                  libcap-ng
aspell           db           gnutls                 libcheck
atk              debianutils  gpgme                  libcroco
attr             dos2unix     icu                    libdaemor
bash-completion  enchant      iso-codes              libevdev
bmap-tools       fribidi      libassuan              libevent
boost            gdbm         libatomic-ops          libexif
```

```
ca-certificates  gmp          libbsd                  libffi
```

La liste n’est pas évidente à lire, mais un comptage rapide avec «`find ../poky/meta/recipes* -name '*bb'| wc -l`» nous indique la présence de près de 800 recettes livrées dans Poky. La plupart d’entre-elles sont des services ou des utilitaires bas-niveaux assez peu visibles au premier abord. Nous pouvons toutefois ajouter sur notre image un petit outil que j’aime bien : **`mc` le *Midnight Commander***. Pour cela nous ajoutons simplement la ligne suivante dans `local.conf` :

```
IMAGE_INSTALL_append = " mc"
```

La variable «`IMAGE_INSTALL`» contient la liste des noms de *packages* à installer sur le système cible.

Le suffixe «`_append`» indique que la chaîne fournie en partie droite de l'affectation doit être ajoutée **à la fin de la variable** concernée. Nous ajoutons donc le nom du *package* `mc` précédé d'une espace qui sert de séparateur entre les éléments de la liste de *packages*.

Nous pourrions préférer utiliser :

```
IMAGE_INSTALL_prepend = "mc "
```

pour ajouter la chaîne **en début de variable** (et en la faisant suivre de l'espace de séparation).

Plusieurs lignes de ce type peuvent se suivre pour allonger d’autant le contenu de l’image.

Après avoir regénéré «`core-image-base`» et nous être connectés, nous pouvons lancer la commande «`mc`» et retrouver l'atmosphère un peu surannée des années 1990 sur Ms-Dos (figure II.2-1). Malgré son aspect désuet aujourd'hui, cet outil peut encore s'avérer très pratique lors de la mise au point d’un système embarqué pour le confort du développement en ligne de commande.

Il existe d'autre outils dont les recettes se trouvent dans Poky que j'aime bien intégrer dans mes images embarquées afin de permettre la mise au point du code métier : `strace`, `valgrind`, `powertop`, `gdbserver`… Nous en reparlerons dans la troisième partie, pour l'instant contentons-nous d'ajouter les lignes suivantes :

```
IMAGE_INSTALL_append = " mc"
IMAGE_INSTALL_append = " strace"
IMAGE_INSTALL_append = " gdbserver powertop"
```

Comme on le voit, on peut aussi bien enchaîner des lignes d'ajout que regrouper les différents packages désirés dans une seule ligne. Je n'ai pas intégré l'utilitaire `valgrind` car il n'est pas supporté par la génération de processeur émulé par `qemuarm`, mais si vous faites un *build* pour Raspberry Pi par exemple, il vous est tout à fait possible de l'ajouter.

Après recompilation de l'image, je vérifie la présence des outils ajoutés. Je ne

m’intéresse pas ici à leur fonctionnement ou leur résultat, juste à leur présence sur ma cible :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox /dev/ttyAMA(

mybox login: root
Password: (linux)

root@mybox:~# strace -V
strace -- version 5.3
Copyright (c) 1991-2019 The strace developers <https://strace.
This is free software; see the source for copying conditions.
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTIC

Optional features enabled: (none)
root@mybox:~# gdbserver --version
GNU gdbserver (GDB) 8.3.1
Copyright (C) 2019 Free Software Foundation, Inc.
gdbserver is free software, covered by the GNU General Public
This gdbserver was configured as "arm-poky-linux-gnueabi"
root@mybox:~# powertop --version
PowerTOP version v2.10
root@mybox:~#
```

Ajout de packages hors Poky

Pendant la mise au point d’un système embarqué, on a souvent besoin d’éditer des fichiers directement sur la cible. Sur une installation «`core-image-base`», nous disposons bien de **l’éditeur «`vi`»**, intégré dans le package `busybox`. Néanmoins tout le monde n’est pas familier des commandes de `vi`, et l’utilisation d’un outil un peu plus intuitif s’avère généralement plus reposante. Supposons que nous souhaitions installer **l'éditeur «nano»** par exemple, qui est plus simple d’emploi que `vi`.

Si nous essayons simplement d’ajouter la ligne :

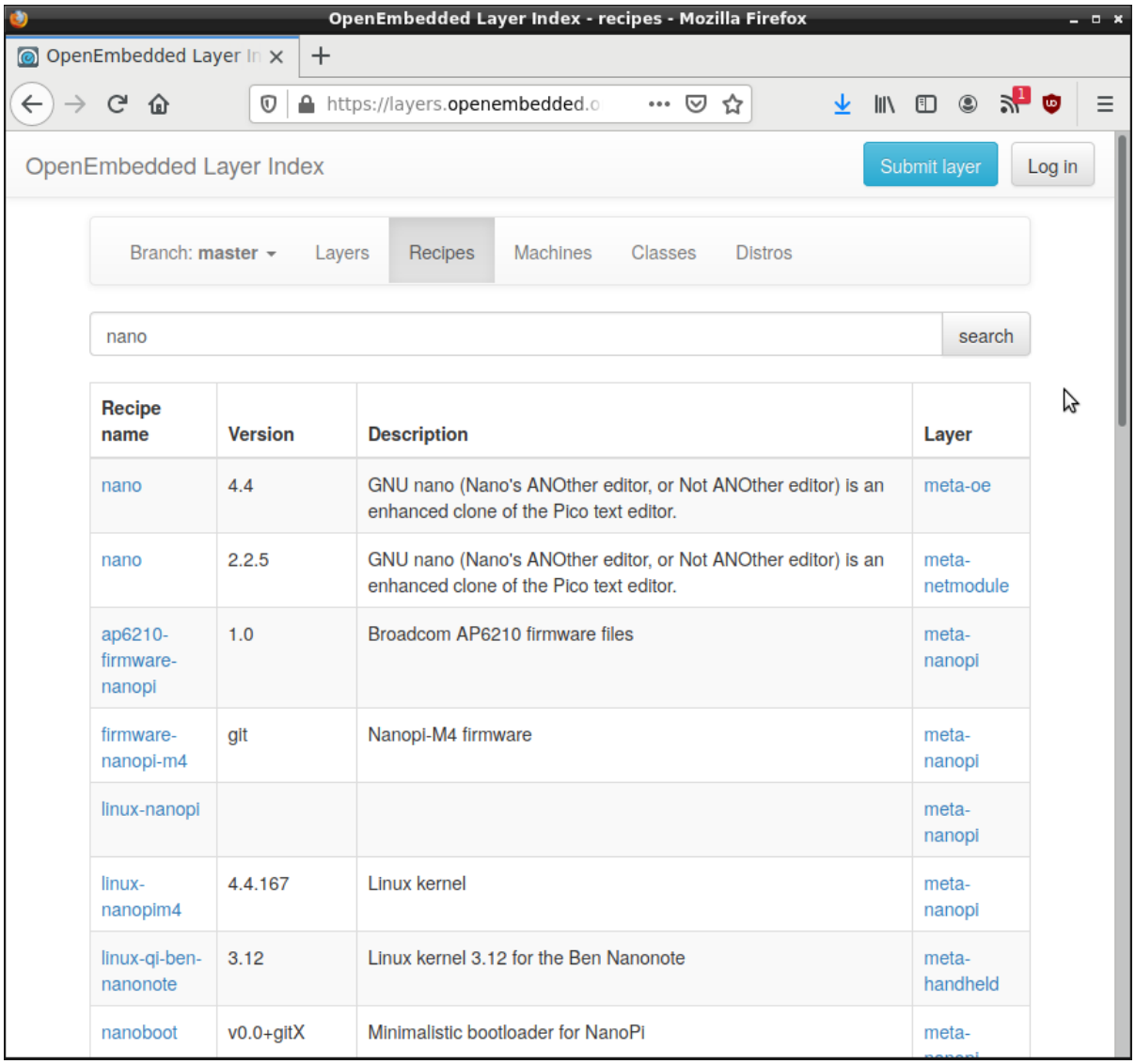
```
IMAGE_INSTALL_append = " nano"
```

dans notre fichier «`local.conf`», `bitbake` va se plaindre, nous afficher une série de lignes en rouge et terminer en indiquant une erreur :

```
ERROR: Nothing RPROVIDES 'nano' (but /home/testing/Build/Lab/Y
NOTE: Runtime target 'nano' is unbuildable, removing...
Missing or unbuildable dependency chain was: ['nano']
ERROR: Required build target 'core-image-base' has no buildabl
Missing or unbuildable dependency chain was: ['core-image-base
```

Clairement, `bitbake` n’a pas trouvé dans les recettes fournies par Poky celle permettant de construire `nano`.

Il va probablement nous falloir **trouver un *layer*** (un ensemble de recettes) à installer sur notre système. Pour cela, l'habitude est de se rendre 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. Nous trouvons alors une page de résultat comme celle de la figure II.2-2.



Clairement la recette qui m'intéresse est la première de la liste. Elle se trouve dans le *layer* indiqué dans la colonne de droite : «`meta-oe`». En cliquant sur ce lien je vois qu'il s'agit d'un *layer* appartenant à un ensemble plus grand : «`meta-openembedded`», et je trouve l'adresse du téléchargement à effectuer.

```
[build-qemu]$ cd ..
[Yocto-lab]$ git clone  git://git.openembedded.org/meta-openem
Clonage dans 'meta-openembedded'…
remote: Counting objects: 121436, done.
remote: Compressing objects: 100% (41174/41174), done.
[...]
Extraction des fichiers: 100% (5553/5553), fait.
```

```
[Yocto-lab]$ ls  meta-openembedded/
contrib           meta-gnome       meta-networking  meta-pytho
COPYING.MIT       meta-initramfs   meta-oe          meta-webse
meta-filesystems  meta-multimedia  meta-perl        meta-xfce
[Yocto-lab]$
```

L’ensemble «`meta-openembedded`» est assez incontournable dès lors que l’on prépare un système avec Yocto. Il contient plus d’un millier de recettes pour de nombreux *packages* très utiles pour Linux embarqué.

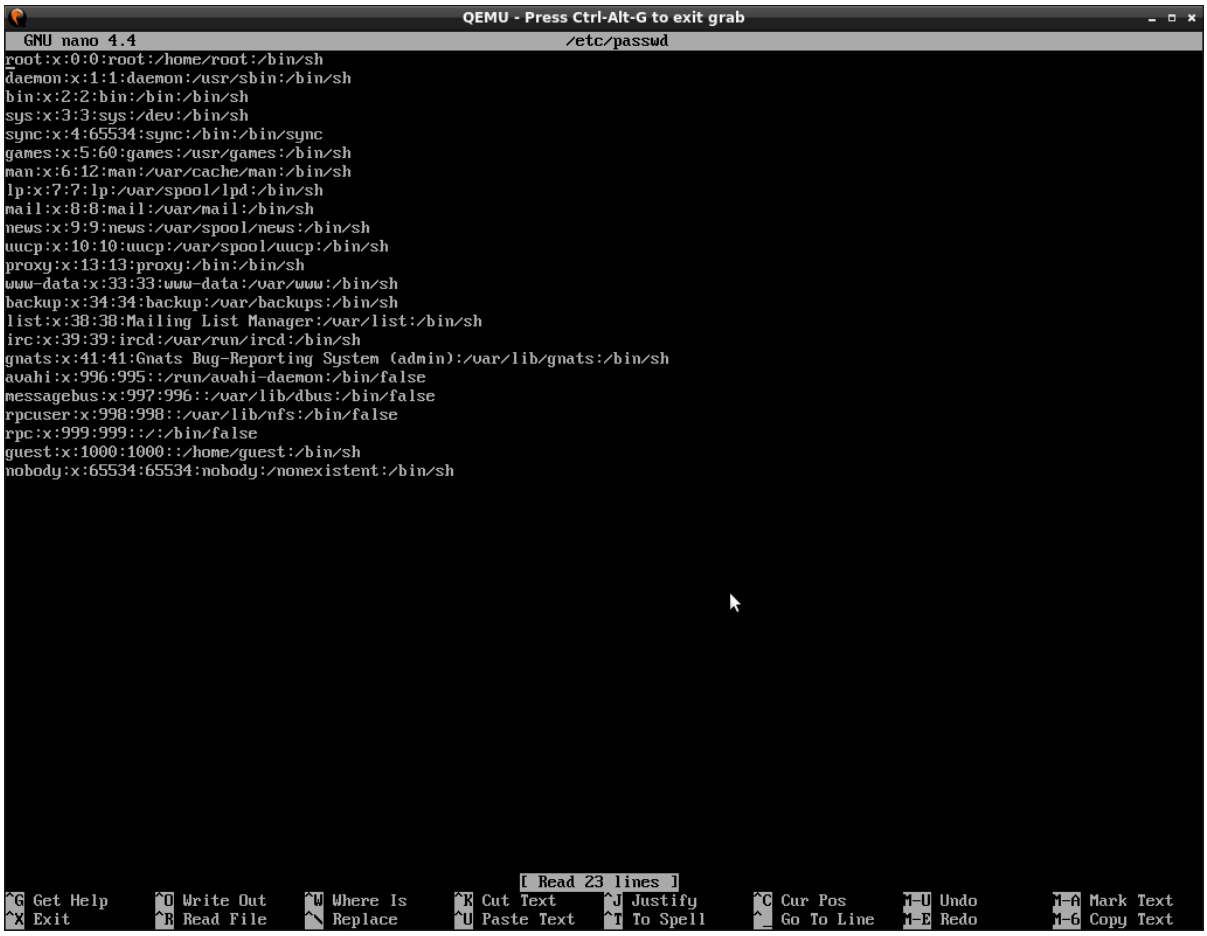
Nous devons ajouter le *layer* «`meta-oe`» à la liste de ceux pris en compte dans notre *build*.

```
[Yocto-lab]$ cd build-qemu/
[build-qemu]$ bitbake-layers add-layer ../meta-openembedded/me
NOTE: Starting bitbake server...
[build-qemu]$
```

Nous pouvons à présent relancer notre compilation, elle se déroule normalement et `nano` est intégré dans notre image.

```
[build-qemu]$ bitbake  core-image-base
  [...]
```

Par exemple, sur la figure II.2-3, nous l'employons pour consulter le fichier `/etc/passwd`.



Utilisation des fonctionnalités d’image

Nous avons utilisé la compilation avec «`core-image-base`» un peu aveuglément, sans comprendre véritablement ce que cela signifie. Nous allons examiner son contenu plus en détail. Pour cela nous recherchons le fichier qui décrit «`core-image-base`» dans les répertoires de Poky. Il s’agit d’une recette avec l’extension «`.bb`».

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

L’emplacement du fichier est logique. Voyons son contenu.

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

Hormis la description et la licence, deux lignes nous 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 notre 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.

Nous voyons que Yocto nous 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-`	Système de fichiers en lecture-seule.

`rootfs`	
`splash`	Écran d’accueil pendant le *boot*

Nous pouvons, par exemple, ajouter dans notre fichier `local.conf` la ligne :

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

pour tester la présence des outils de compilation sur la cible après avoir regénéré l'image. La compilation ci-dessous a bien lieu sur ma cible ARM émulée par Qemu :

```
mybox login: root
Password: (linux)
root@mybox:~# vi my-hello.c
   #include <stdio.h>
   #include <unistd.h>

   int main(void)
   {
     char host[512];
     gethostname(host, 512);
     printf("Hello from %s\n", host);
     return 0;
   }
root@mybox:~# gcc my-hello.c -o my-hello -Wall
root@mybox:~# ./my-hello
Hello from mybox
root@mybox:~# g++ --version
g++ (GCC) 9.2.0
[...]
root@mybox:~#
```

Il n’est pas fréquent d’installer une chaîne de compilation native directement sur la plateforme cible, mais cela peut s’avérer intéressant pendant la phase de

prototypage et de mise au point. Il s'agit d'un point sur lequel Yocto est plus puissant que son confrère Buildroot qui a renoncé à cette possibilité il y a quelques années.

Création d'une image spécifique

Jusqu'à présent nous avons 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 tout 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 nous allons créer **notre propre fichier de description d'image**, et nous n'invoquerons plus «`bitbake core-image-base`» mais «`bitbake my-image`» par exemple (en situation réelle, je nomme plutôt l'image en fonction du projet de mon client).

Pour cela, nous devons commencer par **créer notre propre *layer***. Rien de compliqué, l'outil «`bitbake-layers`» est là pour nous 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-l
[build-qemu]$ ls  ..
build-bbb  build-qemu  build-rpi  downloads  meta-my-layer  me
[build-qemu]$
```

Le *layer* est bien créé mais, comme «`bitbake-layers`» nous l'affiche de manière un peu ambiguë, il n'est pas encore intégré dans la liste des *layers* que nous utilisons pour nos *builds*. Nous devons l'y ajouter :

```
[build-qemu]$ bitbake-layers  show-layers
```

```
NOTE: Starting bitbake server...
layer                 path
=============================================================
meta                  /media/cpb/Yocto-lab/poky/meta  5
meta-poky             /media/cpb/Yocto-lab/poky/meta-poky  5
meta-yocto-bsp        /media/cpb/Yocto-lab/poky/meta-yocto-bsp
meta-oe               /media/cpb/Yocto-lab/meta-openembedded/m
[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
=============================================================
meta                  /media/cpb/Yocto-lab/poky/meta  5
meta-poky             /media/cpb/Yocto-lab/poky/meta-poky  5
meta-yocto-bsp        /media/cpb/Yocto-lab/poky/meta-yocto-bsp
meta-oe               /media/cpb/Yocto-lab/meta-openembedded/m
meta-my-own-layer     /media/cpb/Yocto-lab/meta-my-layer  6
[build-qemu]$
```

Nous créons dans notre *layer* un début d'arborescence pour héberger nos recettes spécifiques. Son nom doit commencer par «`recipes-`». Je choisis arbitrairement «`recipes-custom`».

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

Il faut ensuite y créer un sous-répertoire nommé «`images`» pour stocker nos recettes décrivant des images. Puis nous y copions 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-
```

En le copiant, j'ai renommé le fichier «`core-image-base.bb`» en «`my-image.bb`». Éditons-le, pour obtenir par exemple le contenu 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"
```

Nous éditons `local.conf` pour supprimer toutes les lignes «`IMAGE_INSTALL_append`» et «`IMAGE_FEATURES`» que nous avions ajoutées auparavant et lançons le nouveau *build* avec :

```
[build-qemu]$ bitbake  my-image
```

Après compilation, nos fichiers seront disponibles dans `tmp/deploy/images/qemuarm/` avec le préfixe «`my-image-qemuarm-`». Au lancement, `runqemu` utilise la configuration la plus récente. Néanmoins, il est possible de préciser le nom de l'image pour éviter toute confusion ainsi :

```
[build-qemu]$ runqemu  my-image  qemuarm
```

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 nous utiliserions toujours «+=» qui semble plus simple.

- Avec «+=» l'ajout se fait **immédiatement** dès la lecture du fichier de recette. 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 notre recette d'image, le contenu de `IMAGE_FEATURES` est donc exactement "`splash tools-debug tools-profile tools-sdk ssh-server-dropbear`".
- Avec «`_append`», l'ajout de la chaîne 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 nous avions 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 notre 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.

Conclusion

Nous avons vu dans cette séquence – assez longue – comment personnaliser notre image en ajoutant des applications dont les recettes sont livrées avec Poky ou référencées sur le site *Open Embedded Layers Index*. Nous avons également réussi à créer notre propre image, plutôt que de se limiter à enrichir `core-image-base` en écrivant dans `local.conf`.

Dans la prochaine séquence, nous verrons comment modifier le comportement d’une application existante, sans toucher aux recettes téléchargées…

«« **sommaire** »»