Linux embarqué avec Yocto

Production d'un BSP personnalisé

Christophe BLAESS

christophe.blaess@logilin.fr

https://www.blaess.fr/christophe/
twitter: @chrisblaess

Ingénierie et formations sur Linux et les logiciels libres
https://www.logilin.fr

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap
- *Excalidraw* (en ligne) pour certains schémas

ILY 7.6

https://www.blaess.fr/christophe/

https://www.logilin.fr

Découverte du système Linux embarqué

Première connexion

```
Poky (Yocto Project Reference Distro) 4.0.17 qemuarm /dev/tty1

qemuarm login: root
```

Essayez quelques commandes standards.

```
root@qemuarm:~# cd /
root@qemuarm:/# ls
bin         etc         lost+found  proc        sys         var
boot        home        media       run         tmp
dev         lib         mnt         sbin        usr
root@qemuarm:/# uname -a
Linux qemuarm 5.15.150-yocto-standard #1 SMP PREEMPT Mon Mar 11 14:54:40
UTC 2024 armv7l GNU/Linux
root@qemuarm:/#
```

Le message « *random: crng init done* » qui peut apparaître au bout de quelques dizaines de secondes indique simplement que Linux a terminé l'initialisation de son générateur de nombres aléatoires.

Système de fichiers Linux

Le noyau Linux doit obligatoirement disposer d'un système de fichiers pour fonctionner. Celui-ci peut se trouver sur un disque dur, une partition de mémoire *flash*, voire un *ramdisk*.

Le type de système de fichiers – son format – doit être choisi avec soin en prenant en considération les contraintes opérationnelles (économie d'espace, résistance aux arrêts intempestifs, support spécifique, rapidité...)

Formats classiques

Ext2 : standard Linux, rapide, robuste, **Ext4** journalisation (reprise sur arrêt).

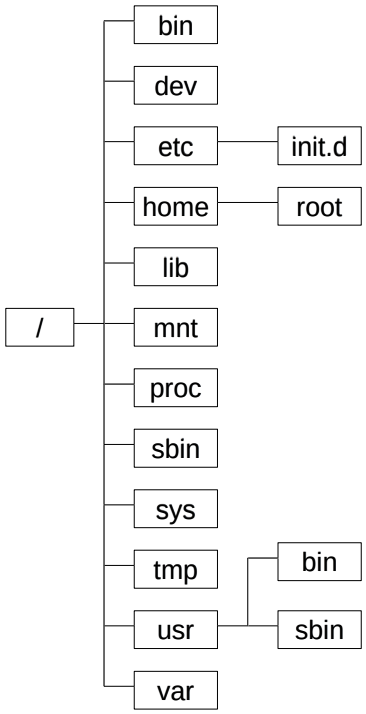
Vfat : Ms-Dos avec noms longs. Standard sur clés USB et cartes flash SD.

F2FS : spécifique pour mémoire Flash avec contrôleur (optimisation *wear leveling*).

Pour mémoire Flash sans contrôleur

JFFS2 : spécifique pour flash NAND/NOR, optimise les accès en écriture, journalisation et compression. Au delà de quelques centaines de Mo on préfère **UBIFS**.

Arborescence typique de Linux



Le pseudo-système de fichiers `devtmpfs` permet de remplir automatiquement `/dev` (sur un disque RAM). Le noyau peut monter automatiquement ce système de fichiers dès le démarrage.

Ces options se trouvent dans la compilation *kernel* :

> Menu « *Device Drivers* », sous menu « *Generic Driver Options* », options « *Maintain a devtmpfs filesystem to mount at /dev* » et « *Automount devtmpfs at /dev, after the kernel mounted the rootfs* ».

Personnalisation de l'image du système

Plusieurs points de personnalisation du système peuvent être nécessaires, nous les étudierons successivement.

- **Configurer le système** : ajouter des utilisateurs, configurer le réseau, etc.
- **Ajuster Busybox :** pour ajouter des applets (commandes Linux standards).
- **Intégrer des *packages* des *layers* de base** de Poky dans l'image produite.
- **Ajouter des *layers*** contenant des packages supplémentaires.
- **Affiner le noyau Linux** : pour ajouter ou supprimer des drivers spécifiques, le support de protocoles réseau, des systèmes de fichiers, etc. Sur architecture ARM la configuration matérielle se fait de plus en plus par le biais du *device tree*.

La personnalisation du système doit se faire au maximum avant la compilation et la production de l'image.

On ne modifie **jamais** les fichiers des *layers* de base (*Poky*, *Open Embedded*...) mais on ajoute plutôt des *layers* personnels contenant des surcharges des recettes originales.

Création d'une image personnelle

Pour personnaliser le contenu du système, on crée une recette d'image dont on peut facilement ajuster le contenu.

On peut s'inspirer par exemple de la recette suivante.

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
```

L'image doit être placée dans un *layer* personnalisé.

Création d'un layer

Le fichier `conf/bblayers.conf` contient la liste des layers à intégrer dans notre compilation :

```
[build-qemu]$ cat  conf/bblayers.conf
 [...]
BBLAYERS ?= " \
  /home/USER/training/layers/poky/meta \
  /home/USER/training/layers/poky/meta-poky \
  /home/USER/training/layers/poky/meta-yocto-bsp \
  "
```

*L'outil **`bitbake-layers`** permet de consulter, ajouter, ou supprimer des layers dans la liste. Rien n'empêche d'éditer `conf/bblayers.conf` manuellement mais `bitbake-layers` vérifie la cohérence des répertoires.*

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

layer             path                                      priority
==========================================================================
meta              /home/USER/training/layers/poky/meta        5
meta-poky         /home/USER/training/layers/poky/meta-poky     5
meta-yocto-bsp    /home/USER/training/layers/poky/meta-yocto-bsp  5
```

Pour ajouter un layer spécifique pour la cible :

```
[build-qemu]$ bitbake-layers  add-layer  <chemin du layer>
```

Travaux pratiques : création d'une image personnalisée

Création d'un layer

Créons un layer personnel et ajoutons-le à la liste à parcourir pour le build.

```
[build-qemu]$ bitbake-layers  create-layer  ../../layers/meta-my-layer


[build-qemu]$ bitbake-layers  add-layer  ../../layers/meta-my-layer


[build-qemu]$ bitbake-layers  show-layers
layer            path                                      priority
==========================================================================
meta             /home/USER/training/layers/poky/meta       5
meta-poky        /home/USER/training/layers/poky/meta-poky  5
meta-yocto-bsp   /home/USER/training/layers/poky/meta-yocto-bsp  5
meta-my-layer    /home/USER/training/layers/meta-my-layer   6
```

Création d'une recette d'image

Créons notre recette d'image

```
[build-qemu]$ mkdir  -p ../../layers/meta-my-layer/recipes-core/images/
```

```
[build-qemu]$ nano  ../../layers/meta-my-layer/recipes-core/images/my-image.bb
```

```
SUMMARY = "My own Yocto Project image."
LICENSE = "MIT"
inherit core-image

IMAGE_INSTALL:append = " os-release"
```

La dernière ligne permet d'ajouter une recette qui intègre dans « /etc » de la cible un fichier d'information sur l'OS.

Générons notre image personnalisée

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

Les fichiers produits sont à présent :

```
[build-qemu]$ ls  tmp/deploy/images/qemuarm/my-*
[...]
tmp/deploy/images/qemuarm/my-image-qemuarm.rootfs.ext4
tmp/deploy/images/qemuarm/my-image-qemuarm.rootfs.manifest
tmp/deploy/images/qemuarm/my-image-qemuarm.rootfs.tar.bz2
tmp/deploy/images/qemuarm/my-image-qemuarm.tar.bz2
tmp/deploy/images/qemuarm/my-image-qemuarm.testdata.json
[...]
[build-qemu]$
```

Administration du système

Utilisateurs et mots de passe

Il existe une classe nommée **`extrausers`**, permettant de configurer les utilisateurs du système et leurs mots de passe avec une variable d'environnement **`EXTRA_USERS_PARAMS`** renseignée dans le fichier d'image.

```
inherit extrausers
```

Pour ajouter un utilisateur :

```
EXTRA_USERS_PARAMS  +=  "useradd -p  <hashed-password>  <user>;"
```

Pour hacher le mot de passe on peut utiliser l'utilitaire `openssl` pour obtenir la chaîne cryptée. Il faut ensuite précéder les trois caractères **`$`** par des *backslashes* **`\`**.

```
$ openssl passwd -5 <password>
```

Pour modifier le mot de passe d'un utilisateur existant (par exemple `root`) :

```
EXTRA_USERS_PARAMS  +=  "usermod -p  <hashed-password>  <user>;"
```

Il existe aussi les commandes `userdel`, `groupadd`, `groupmod`, `groupdel`.

```
$ openssl passwd -5 linux
$5$0HdoiyJ4wW67wvZu$FMys9UvVH/EVMSLp3L7X2Iwf7FTgx7OcU1q4sEYqQz1

EXTRA_USERS_PARAMS += "usermod -p '\$5\$0Hdoy...\$FMys9Uv...z1' root;"
```

Travaux pratiques : configuration des utilisateurs et mots de passe

Par défaut, l'administrateur `root` n'a pas de mot de passe.

Créez-en un de votre choix.

Créez un utilisateur « `guest` » avec le mot de passe « `welcome` ».

Solutions

```
[build-qemu]$ openssl passwd -5 linux
\$5\$GXWuuigS\$fcRncVPUn5uT1dVYdKCWruHHnuf7z4Jpdjbo9kw6r0A
[build-qemu]$ openssl passwd -5 welcome
\$5\$6ftHFftd\$RcFgQMmf.DD1l0aJ24nYMrzSEuer1eU7k8przp/Ob56
```

Ajouts dans **`../../layers/meta-my-layer/recipes-core/images/my-image.bb`** :

```
inherit extrausers

EXTRA_USERS_PARAMS += "usermod -p '\$5\$GXWuuigS\$fcRncVPUn5
uT1dVYdKCWruHHnuf7z4Jpdjbo9kw6r0A' root;"

EXTRA_USERS_PARAMS += "useradd -p '\$5\$6ftHFftd\$RcFgQMmf.D
D1l0aJ24nYMrzSEuer1eU7k8przp/Ob56' guest;"
```

Attention à bien encadrer le mot de passe par des apostrophes et l'ensemble de la ligne de commande par des guillemets.

Pensez à précéder les $ par des \ (3 fois par mot de passe)

Syntaxe des recettes

Toute la configuration nécessaire pour la compilation se trouve dans des **variables** renseignées dans les recettes. Il est possible de supprimer, compléter, surcharger les variables à partir de la recette d'image.

La commande « **`bitbake -e`** » permet de voir tout le contenu de l'environnement.

Affectation et évaluation de variables

Pour définir (ou écraser) le contenu d'une variable `VAR`, on peut écrire :

`VAR := "value"` (évaluation immédiate du contenu)

`VAR  = "value"` (évaluation au moment de l'utilisation).

On lit le contenu d'une variable avec **`${VAR}`**

Ajouter dans `my-image.bb` les lignes :

```
MY_A  = "111"
MY_B := "${MY_A} ${MY_A}"
MY_C  = "${MY_A} ${MY_A}"
MY_A  = "222"
```

Vérifiez le résultat avec :

```
$ bitbake -e my-image | grep '^MY_'
```

Solution :

`MY_A` contient la dernière valeur affectée, soit « `222` »

`MY_B` est remplie avec une chaîne évaluée immédiatement, alors que `MY_A` contient encore « `111` », donc `MY_B` vaut « `111 111` » .

`MY_C` est remplie avec une chaîne évaluée au moment de la lecture du contenu de `MY_C`. La variable `MY_A` ayant été finalement remplie avec « `222` », `MY_C` vaut « `222 222` »

Valeurs par défaut

Pour définir le contenu d'une variable seulement si elle était vide au préalable :

```
VAR ?= "value"
```

Pour fournir un contenu seulement si aucun autre n'est défini, même ultérieurement :

```
VAR ??= "value"
```

Écrivez les lignes suivantes dans `my-image.bb` :

```
MY_A  ?= "111"

MY_B  ?= "111"
MY_B  ?= "222"

MY_C ??= "111"
MY_C ??= "222"
```

Quelles sont les valeurs ?

```
$ bitbake -e my-image | grep '^MY_'
MY_A="      "
MY_B="      "
MY_C="      "
```

Solution :

`MY_A` est assignée une seule fois, elle contient « `111` »

`MY_B` est assignée deux fois avec une priorité à la première affectation, elle vaut donc « `111` » également.

`MY_C` est assignée deux fois, avec une priorité à la dernière affectation. Elle contiendra donc « `222` ».

Concaténations

Ajouts en fin de chaîne :

`VAR **+=** "value"` (insertion d'une espace avant la chaîne insérée)

`VAR:**append** = "value"` (pas d'insertion d'espace)

Ajouts en début de chaîne :

`VAR **=+** "value"` (insertion d'une espace après la chaîne insérée)

`VAR:**prepend** = "value"` (pas d'insertion d'espace)

Note : avant la mi-2021, les suffixes `:prepend` et `:append` s'écrivaient `_prepend` et `_append`.

Vérifiez le comportement avec les séquences suivantes :

```
MY_A  = "111"
MY_A += "222"

MY_B = "111"
MY_B:append = "222"

MY_C =  "111"
MY_C =+ "222"

MY_D = "111"
MY_D:prepend = "222"
```

Les opérateurs `:append` et `:prepend` servent notamment lorsqu'on manipule des chemins de recherche (`PATH`) où le séparateur est un '`:`' et pas une espace.

Solution :

```
$ bitbake -e my-image | grep '^MY_'
MY_A="111 222"
MY_B="111222"
MY_C="222 111"
MY_D="222111"
```

Les concaténations « += » et « =+ » font une affectation immédiate.

Les opérateurs « :append » et « :prepend » réalisent une affectation différée.

Exemples :

```
MY_A += "111"
MY_A ?= "222"

MY_B:append = "111"
MY_B ?= "222"
```

```
$ bitbake -e my-image | grep '^MY_'
MY_A=" 111"
MY_B="222111"
```

Ajouts de packages

Packages standards de Poky

La variable **`IMAGE_INSTALL`** contient la liste des packages à intégrer dans l'image finale.

Pour vérifier le contenu de cette variable :

```
[build-qemu]$ bitbake -e  <nom-de-l'image>  |  grep  ^IMAGE_INSTALL
IMAGE_INSTALL="packagegroup-core-boot   packagegroup-base-extended"
IMAGE_INSTALL_COMPLEMENTARY=""
```

La recette pour *packagegroup-core-boot* se trouve dans

```
poky/meta/recipes-core/packagegroups/packagegroup-core-boot.bb
```

Elle regroupe essentiellement Busybox et des scripts de démarrage

Ajout d'un package standard

La commande **`bitbake -s`** permet de lister les packages disponibles (et leur numéro de version) dans les *layers* inscrits dans `conf/bblayers.conf`.

```
[build-qemu]$ bitbake  -s
Recipe Name                    Latest Version         Preferred Version
===========                    ==============         =================
acl                                :2.3.1-r0
acl-native                         :2.3.1-r0
[...]
dtc                                :1.6.1-r0
dtc-native                         :1.6.1-r0
[...]
vala-native                        :0.56.3-r0
valgrind                           :3.18.1-r0
vim                                :9.0.1592-r0
[...]
```

On peut ajouter un package à l'image en insérant son nom à la fin de `IMAGE_INSTALL`. C'est ce que l'on fait généralement dans le fichier d'image (ou à défaut dans `conf/local.conf`).

Travaux pratiques *:* ajout de packages standards de Poky

Nous souhaitons disposer des outils de débogage **`gdbserver`**, **`dtc`** et **`valgrind`** sur notre cible.

Vérifiez s'ils sont présents dans l'image par défaut.

Vérifiez s'ils sont présents dans les outils standards de Poky (`gdbserver` est inclus dans le *package* `gdb`).

Si c'est bien le cas, ajoutez-les dans l'image désirée.

Solutions

```
[build-qemu]$ bitbake -s | grep ^dtc
dtc                                             :1.6.1-r0

[build-qemu]$ bitbake -s | grep ^gdb
gdb                                             :11.2-r0

[build-qemu]$ bitbake -s | grep ^valgrind
valgrind                                        :3.18.1-r0
```

Ajouts dans `../../layers/meta-my-layer/recipes-core/images/my-image.bb` :

```
IMAGE_INSTALL:append = " dtc"
IMAGE_INSTALL:append = " gdbserver"
IMAGE_INSTALL:append = " valgrind"
```

Ajout d'un layer OpenEmbedded

Les layers de base de Poky ne contiennent qu'un nombre assez limité de packages (quelques centaines).

Si on souhaite ajouter un package supplémentaire, il faut vérifier sa disponibilité dans les recettes de Yocto Project sur :

https://layers.openembedded.org/.

Dans l'onglet *recipes*, chercher l'utilitaire désiré, noter le *layer* concerné et cloner le dépôt correspondant (en prenant la branche **`kirkstone`**)

*Entre autres, le dépôt **`meta-openembedded`** est très riche (des milliers de packages) et contient plusieurs layers :*

```
[build-qemu]$ ls ../../layers/meta-openembedded/
contrib           meta-gnome       meta-networking  meta-ruby       README
COPYING.MIT       meta-gpe         meta-oe          meta-systemd
meta-efl          meta-initramfs   meta-perl        meta-webserver
meta-filesystems  meta-multimedia  meta-python      meta-xfce
```

Travaux pratiques : ajout de packages du dépôt meta-openembedded

Nous souhaitons installer l'éditeur **nano** et le serveur **apache2** sur notre cible.

Mais ils ne sont pas disponibles dans les *layers* de base de Yocto…

Que faire ?

Une fois le serveur Apache installé, vous pouvez vérifier s'il fonctionne en appelant l'adresse IP de la cible (par ex. 192.168.7.2 avec Qemu) depuis le navigateur de votre système hôte.

Solutions

```
[build-qemu]$ cd ../../layers/
[layers]$ git  clone  git://git.openembedded.org/meta-openembedded  -b
kirkstone
[layers]$ cd -

[build-qemu]$ bitbake-layers add-layer ../../layers/meta-openembedded/meta-
oe/
[build-qemu]$ bitbake-layers add-layer ../../layers/meta-openembedded/meta-
webserver/
```

Ajout dans `../../layers/meta-my-layer/recipes-core/images/my-image.bb` :

```
IMAGE_INSTALL:append = " nano apache2"
```

Configuration de Busybox

L'utilitaire **Busybox** – surnommé « le couteau suisse de l'embarqué » – implémente plus de 400 commandes Linux courantes. En voici quelques-unes :

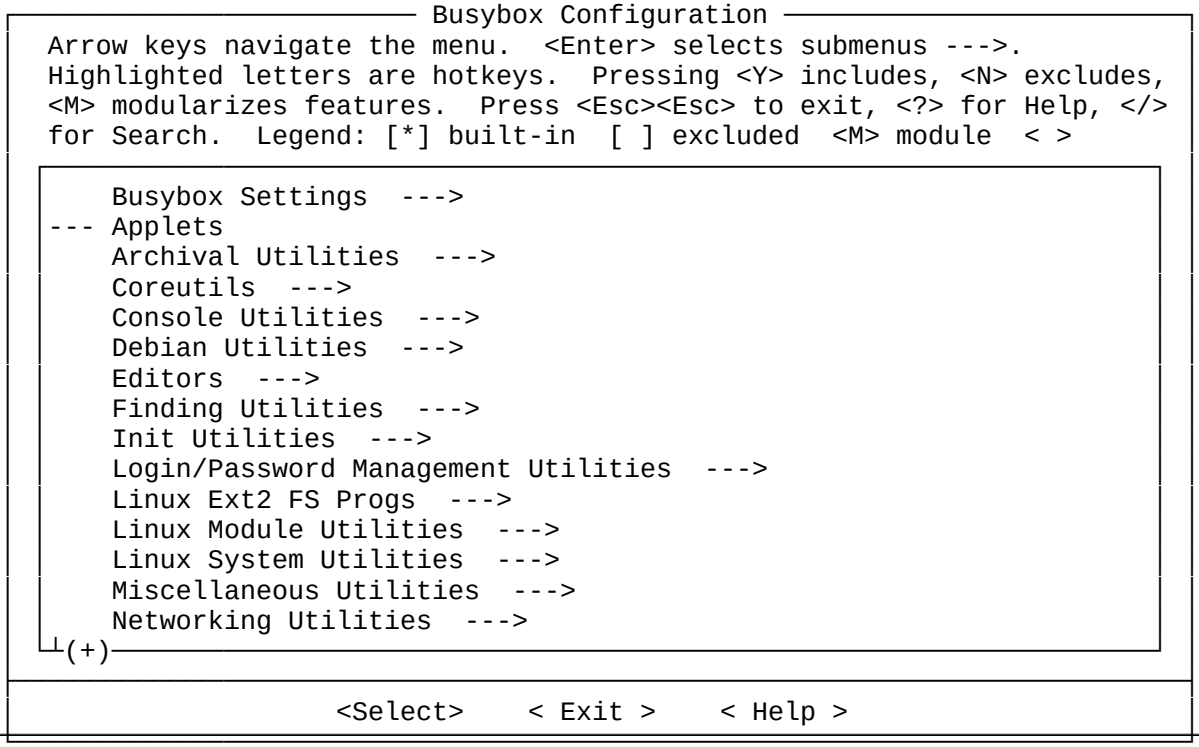
```
[, [[, addgroup, adduser, adjtimex, ar, arp, arping, ash, awk, base64, basename,
beep, blkid, brctl, bunzip2, bzcat, bzip2, cal, cat, catv, chattr, chgrp, chmod,
chown, chroot, chrt, chvt, cksum, clear, cmp, cp, cpio, crond, crontab, cryptpw,
cut, date, dc, dd, deallocvt, delgroup, deluser, depmod, devmem, df, dhcprelay,
diff, dirname, dmesg, dnsd, dnsdomainname, dos2unix, du, dumpkmap, dumpleases,
echo, ed, egrep, eject, env, expr, false, fatattr, fbset, fbsplash, fdflush,
fdformat, fdisk, fgrep, find, flash_eraseall, flashcp, fold, free, freeramdisk,
fsck, fstrim, ftpd, ftpget, ftpput, fuser, getopt, getty, grep, groups, gunzip,
gzip, halt, head, hexdump, hostid, hostname, httpd, hwclock, id, ifconfig, ifdown,
ifup, inetd, init, insmod, iostat, ip, kill, killall, klogd, last, less, ln,
loadfont, loadkmap, logger, login, logname, losetup, ls, lsattr, lsmod, lsof,
lspci, lsusb, lzcat, md5sum, mesg, microcom, mkdir, mkdosfs, mke2fs, mkfifo,
mkfs.ext2, mkfs.vfat, mknod, mktemp, modinfo, modprobe, more, mount, mountpoint,
mpstat, mt, mv, nameif, nanddump, nandwrite, nc, netstat, nice, nmeter, nohup,
nslookup, ntpd, od, openvt, passwd, patch, pgrep, pidof, ping, pipe_progress,
pivot_root, pkill, pmap, poweroff, powertop, printenv, printf, ps, pscan, pstree,
pwd, rdate, rdev, readlink, readprofile, realpath, reboot, renice, reset, resize,
rev, rm, rmdir, rmmod, route, rtcwake, run-parts, runlevel, rx, script,
scriptreplay, sed, seq, setarch, setconsole, setkeycodes, setlogcons, setserial,
setsid, sh, sha1sum, sha256sum, sha3sum, sha512sum, sleep, sort, start-stop-daemon,
stat, strings, stty, su, sulogin, sum, sync, sysctl, syslogd, tac, tail, tar,
taskset, tcpsvd, tee, telnet, telnetd, test, tftp, tftpd, time, top, touch, tr,
traceroute, true, tty, tune2fs, udhcpc, udhcpd, udpsvd, umount, uname, uncompress,
unexpand, uniq, unix2dos, unlink, unzip, uptime, usleep, uudecode, uuencode, vi,
vlock, wall, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat
```

Les *layers* par défaut de Yocto intègrent automatiquement Busybox avec les scripts et fichiers de configuration nécessaires.

Menu de configuration de Busybox

Certaines applets intéressantes de Busybox ne sont pas sélectionnées par défaut.

```
[my-build]$ bitbake -c menuconfig busybox
```



Si l'écran de configuration refuse de s'afficher, on peut configurer la variable `OE_TERMINAL` dans `conf/local.conf` à la valeur « `screen` ».

Intégration de la configuration

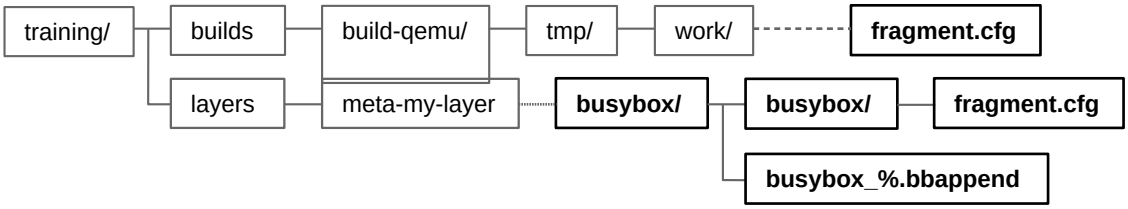
Une fois la configuration de Busybox réalisée, on génère un fichier de différences par rapport à la configuration originale.

```
[build-qemu]$ bitbake  -c  diffconfig  busybox
Config fragment has been dumped into:
 /home/USER/yocto-training/build-qemu/tmp/work/[...]/fragment.cfg
```

Le fichier « fragment » créé est une surcharge de la configuration par défaut. On l'ajoute au layer de configuration pour la cible `busybox` avec l'outil **`recipetool`** :

```
recipetool  appendsrcfile  -w  <layer>  <cible>  <fragment>
```

```
[build-qemu]$ recipetool  appendsrcfile  -w  ../../layers/meta-my-layer/
   busybox   tmp/work/[...]/fragment.cfg
```



Il est conseillé ensuite de « nettoyer » le répertoire de compilation de Busybox avec

```
[build-qemu]$ bitbake  -c clean  busybox
```

Nous retrouverons le même mécanisme de menu et de fichier de différences pour la configuration du noyau Linux.

Travaux pratiques : personnalisation de Busybox

Modifiez la configuration de Busybox pour ajouter les utilitaires suivants :

```
Menu « Linux System Utilities »
    [*] chrt
    [*] taskset


Menu « Networking Utilities »
    [*] ntpd
```

Intégrez les différences de configuration dans votre layer.

Solutions

```
[build-qemu]$ bitbake  -c  menuconfig  busybox
```

(Édition et sauvegarde des modifications)

```
[build-qemu]$ bitbake  -c  diffconfig  busybox

[build-qemu]$ recipetool  appendsrcfile  -w  ../../layers/meta-my-layer/
   busybox  tmp/work/[...]/fragment.cfg

[build-qemu]$ bitbake  -c  clean  busybox

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

Système de fichiers en lecture-seule

Il est préférable, pour des raisons de robustesse, de laisser le système de fichier principal (*rootfs*) en lecture seulement.

Un système de fichiers en lecture-écriture est susceptible de contenir des (petites) incohérences en cas de coupure brutale d'alimentation électrique.

Or le *rootfs* doit être monté par le *kernel* durant le *boot*, et celui-ci ne peut pas gérer ces incohérences. Il n'a pas accès aux outils comme `fsck` (*file system check*) car ils se trouvent dans l'arborescence.

Conserver un *rootfs* en lecture seule est une bonne pratique de l'embarqué.

Il s'agit d'une fonctionnalité proposée par la classe « `core-image` » dont dépend notre image.

Travaux pratiques : configurer le rootfs en lecture-seule

Dans la recette d'image ajouter :

```
IMAGE_FEATURES += "read-only-rootfs"
```

Pour éviter les erreurs sur les architectures détectant dynamiquement les consoles, ajouter dans le fichier `conf/local.conf` :

```
SERIAL_CONSOLES_CHECK:forcevariable = ""
SERIAL_CONSOLES = "115200;ttyAMA0"
```

Note : pour certaines cartes (Beagle Bone Black par exemple), il faut remplacer **`ttyAMA0`** par **`ttyS0`**.