II.3 – Personnalisation des recettes

Christophe BLAESS - janvier 2020

Nous avons vu comment ajouter des applications dont les recettes sont livrées avec Poky ou référencées sur *Open Embedded Layers Index*. Supposons que nous devions adapter de manière plus ou moins importante le contenu d’une de ces recettes.

Comment procéder sans toucher aux fichiers originaux ?

Nous allons le voir dans cette séquence...

Adaptation d’une recette de Poky

Il y a plusieurs manières d’adapter un élément d’une image, qui dépendent du type de modification à effectuer. Notre première approche va consister à **remplacer un fichier fourni par une recette**.

Pour ce faire, on va commencer par rechercher dans l’arborescence de Poky le répertoire où se trouve la recette, et examiner les fichiers présents pour comprendre le mécanisme de construction. Prenons par exemple l’application `psplash` qui affiche une image lors du *boot* du système.

```
[build-qemu]$ cd  ../poky/
[poky]$ find  .  -name psplash
./meta/recipes-core/psplash
./meta-poky/recipes-core/psplash
[poky]$
```

Surprise, deux répertoires sont consacrés à cette application. Le premier concerne la recette d'origine de `psplash`, et le second traite de la personnalisation de l'application pour l'image Yocto. Examinons leurs contenus :

```
[poky]$ ls  meta/recipes-core/psplash/
files  psplash_git.bb
[poky]$ ls  meta/recipes-core/psplash/files/
psplash-init  psplash-poky-img.h
[poky]$
```

Le répertoire de base contient un fichier recette au format `.bb` qui indique comment télécharger et compiler l'application et un sous-répertoire `files/`. Dans ce dernier se trouvent un script de lancement `psplash-init` et un fichier *header* d'extension `.h` qui contient l'image sous forme de tableau en C :

```
[poky]$ cat  meta/recipes-core/psplash/files/psplash-poky-img.
/* GdkPixbuf RGB C-Source image dump 1-byte-run-length-encoded

#define POKY_IMG_ROWSTRIDE (1920)
#define POKY_IMG_WIDTH (640)
#define POKY_IMG_HEIGHT (480)
#define POKY_IMG_BYTES_PER_PIXEL (3) /* 3:RGB, 4:RGBA */
#define POKY_IMG_RLE_PIXEL_DATA ((uint8*) \
  "\377\377\377\377\377\377\377\377\377\377\377\377\377\377\37
  "\377\377\377\377\377\377\377\377\377\377\377\377\377\377\37
  "\377\377\377\377\377\377\377\377\377\377\377\377\377\377\37
  [...]
  "\377\377\377\377\377\223\377\377\377\322\376\377\374\3\372\
  "\276\306fx\203\306]oz\3as~\250\260\270\362\367\371\322\376\
```

```
"\377\377\377\377\377\377\377\377\377\377\223\377\377\37
"\377\374\2\320\326\330p\203\216\312]oz\2hz\205\302\307\312\
[...]
"\377\377\377\377\377\377\377\377\377\377\377\377\377\37
"\377\377\377\377\377\377\377\377\377\377\377\377\377\37
"\377\377\377\377\232\377\377\377")
```

Ce charabia apparent est bien la représentation d'une image graphique pixel par pixel. Voyons l'autre répertoire — celui plus spécifique Yocto — intitulé `pspslash` :

```
[poky]$ ls  meta-poky/recipes-core/psplash/
 files  psplash_git.bbappend
```

Il contient un fichier **`.bbappend`** avec le même nom que la recette `.bb` du répertoire précédent. Il s'agit d'une **extension de recette**, qui est prise en compte **après** la recette principale et peut donc venir modifier son contenu avant de l'interpréter.

Il est important de bien comprendre que `bitbake` charge d'abord en mémoire toutes les recettes et extensions en analysant leurs contenus (étape «`Parsing recipes`») avant d'organiser le travail (étape «`Initialising tasks`») puis de réaliser les opérations nécessaires (étape «`Executing RunQueue Tasks`»). Il est donc possible pour une extension de recette `.bbappend` de surcharger le contenu précédent d'une recette `.bb` et de modifier son comportement.

Voyons le contenu de cette extension :

```
[poky]$ cat  meta-poky/recipes-core/psplash/psplash_git.bbappe
FILESEXTRAPATHS_prepend_poky := "${THISDIR}/files:"
```

Il s'agit d'un contenu que l'on retrouve très souvent dans les fichiers `.bbappend` avec de légères variations. Nous l'avons vu dans la séquence précédente, contrairement aux langages de programmation habituels, il y a une véritable interprétation de la partie gauche d'une affectation. Le caractère souligné (*underscore*) «_» sert à préfixer un opérateur qui précise ou limite la portée de cette affectation. La partie gauche signifie ici «*Lors de la compilation d'une image Poky*»

(_poky) «*ajouter au début*» (_prepend) de la variable FILESEXTRAPATHS.

L'affectation «**:=**» indique que l'interprétation de la partie droite doit se faire dès la lecture du fichier. Avec une affectation «=», elle serait différée au moment de l'analyse de toutes les variables lues. Ceci nous garantit que l'on prend en compte immédiatement le contenu de la variable THISDIR. Comme son nom l'indique, cette dernière représente le répertoire courant, celui de l'extension de recette.

La partie droite de l'affectation permet de préciser le sous-dossier files/ qui est juste à côté de la recette. On notera qu'il est suivi d'un caractère deux-points «:».

La variable FILESEXTRAPATHS contient la liste des chemins dans lesquels on recherche les fichiers nécessaires pour la réalisation d'une recette. Les chemins de la liste sont séparés par des deux-points «:» et ils sont parcourus dans l'ordre de la liste.

En ajoutant le répertoire files/ accompagnant cette extension de recette au début de la liste, on s'assure que les fichiers qu'il contient auront précédence sur ceux de la recette initiale. Ici, le nom du répertoire (files/) est le même que le répertoire de la recette initiale, mais cela n'est pas obligatoire.

Voyons ce que contient ce nouveau répertoire files/ :

```
[poky]$ ls  meta-poky/recipes-core/psplash/files/
psplash-poky-img.h
[poky]$
```

Le fichier présent dans l'arborescence meta-poky/ indiqué vient donc remplacer celui de l'arborescence meta/. Ceci permet d'afficher une image personnalisée au démarrage. Nous pouvons réaliser le même type de modification pour afficher notre propre *splashscreen*.

Créons un répertoire de travail dans notre *layer*, avec le même nom que la recette initiale :

```
[poky]$ cd  ../meta-my-layer
[meta-my-layer]$ mkdir  -p  recipes-core/psplash/files
```

```
[meta-my-layer]$ cd  recipes-core/psplash/files/
[files]$
```

Après avoir consulté la documentation de `psplash`, je crée avec Gimp une image de dimension 640×400 pixels que j'exporte au format PNG. J'ai fait plusieurs essais pour obtenir la dimension qui me convient (et qui est légèrement différente de l'originale). Vous pouvez créer votre propre image ou télécharger celle-ci.

```
[files]$ ls
my-splash.png
```

J'utilise l'outil `gdk-pixbuf-csource` (fourni sur ma machine par le *package* `libgdk-pixbuf2.0-dev`) pour lui demander de convertir mon image PNG en fichier *header* pour le langage C.

```
[files]$ gdk-pixbuf-csource  --macros  my-splash.png  > psplas
```

Il est nécessaire de modifier quelques éléments de l'image, que l'on peut automatiser avec les lignes `sed` suivantes :

```
[files]$ sed  -i  -e  "s/MY_PIXBUF/POKY_IMG/g"  psplash-poky-i
[files]$ sed  -i  -e  "s/guint8/uint8/g"  psplash-poky-img.h
```

Nous avons obtenu le fichier *header* représentant l'image. Il nous faut créer le fichier d'extension de recette, de la même manière que celui de `meta-poky`.

```
[files]$ cd  ..
[psplash]$ nano  psplash_git.bbappend
```

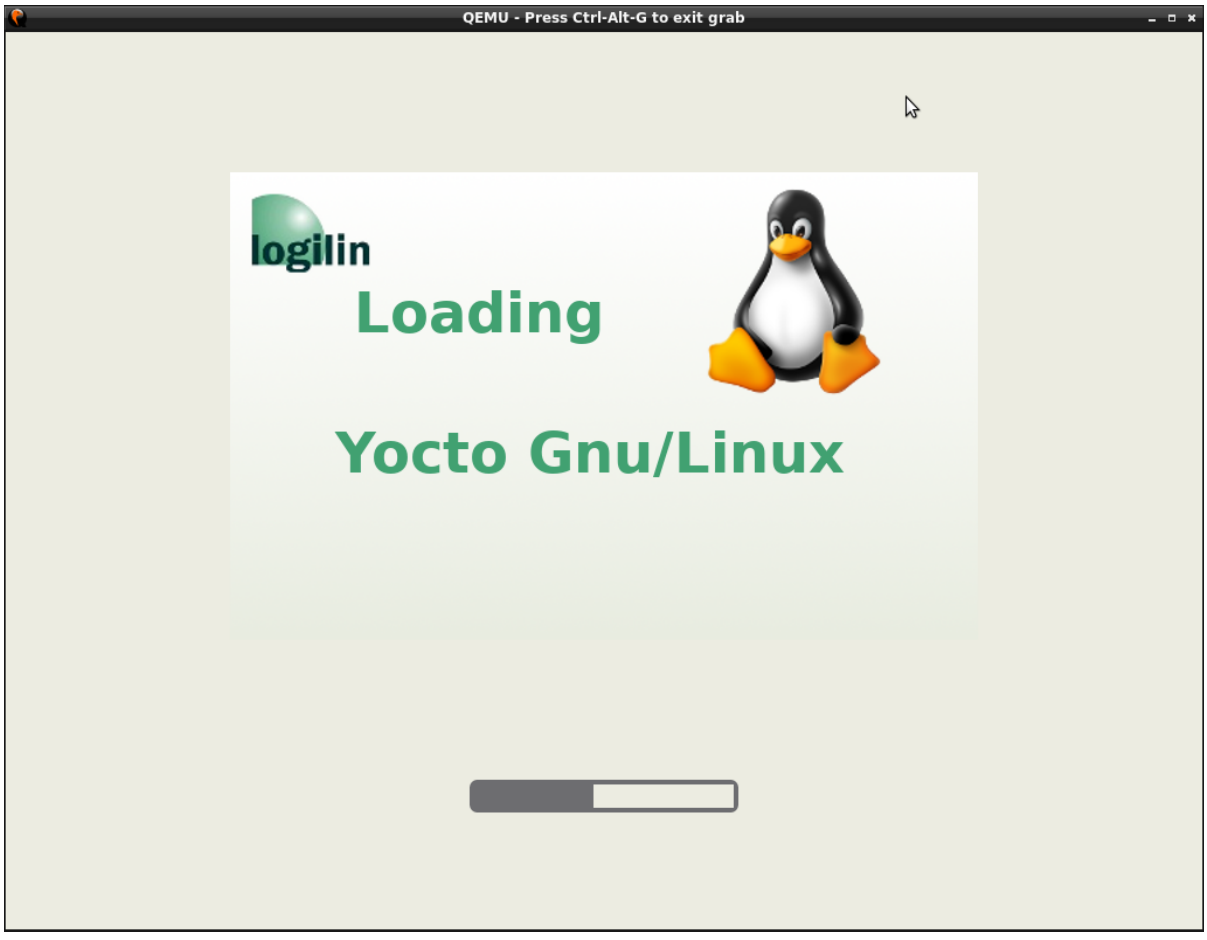
Le fichier ne contient que la ligne indiquant que le contenu du répertoire `files/` doit être plus prioritaire lors de la rechercher d'un fichier.

```
FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

Nous pouvons alors regénérer et tester notre image.

```
[psplash]$ cd  ../../../build-qemu/
[build-qemu]$ bitbake  my-image
```

Au démarrage, nous avons le plaisir de voir notre *splashscreen* personnalisé apparaître comme sur la figure II.3-1.



Nous voyons ainsi comment **remplacer un fichier complet proposé par une recette**. Cela peut être utile dans de nombreux cas, principalement pour des éléments de configuration système (nous le retrouverons par exemple pour ajuster la configuration du réseau).

Néanmoins d’autres modifications peuvent être nécessaires, celles qui consistent à modifier une petite partie d’un fichier. Par exemple quelques lignes d’un fichier source avant compilation. Pour cela on préfère la méthode du *patch*.

Ajout d'un patch dans une recette

Nous allons commencer par **produire et faire appliquer un *patch*** sur un fichier fourni directement par une recette, sans qu'il y ait de compilation. Je vous propose par exemple de prendre la recette «`base-files`» qui se trouve dans le répertoire `meta/recipes-core/` de Poky. Comme son nom l’indique il s’agit d’une recette qui fournit directement des fichiers de base situés dans son sous-répertoire `base-files/`. L’un d’eux est «`profile`», qui est copié dans le répertoire `etc/` de la cible. Il configure des variables d’environnement du shell comme `PATH`, `PS1`, `TERM`, `EDITOR`… avec des valeurs par défaut que l’utilisateur pourra surcharger s’il le souhaite.

La variable `EDITOR` justement, qui indique l’éditeur préféré de l’utilisateur, est initialisée à la valeur «`vi`». Mais nous avons installé sur notre image l'éditeur `nano`, il serait dommage de ne pas en profiter. Créons donc un *patch* pour modifier cette ligne du fichier.

Lorsque le *patch* à créer concerne les fichiers fournis par une recette, comme c'est le cas ici, le plus simple est d'appeler manuellement `diff`. Lorsqu'il s'agira de modification des fichiers d'un projet téléchargé avec `git` par exemple, on préférera faire appel au système de gestion de version pour produire le *patch*. Nous en verrons un exemple dans la séquence IV.3.

Pour commencer je crée une copie temporaire du répertoire `base-files/` qui contient tous les fichiers. Je l’effacerai quand j’aurai fini de préparer le *patch*, je reste donc dans mon répertoire de travail initial.

```
[build-qemu]$ cp  -R  ../poky/meta/recipes-core/base-files/bas
```

J’en crée une deuxième copie où je ferai la modification.

```
[build-qemu]$ cp  -R  ../poky/meta/recipes-core/base-files/bas
```

Puis j'édite le fichier `profile` du répertoire `base-files-modified/` pour remplacer la ligne :

```
EDITOR="vi"            # needed for packages like cron, git-cc
```

par :

```
EDITOR="nano"          # needed for packages like cron, git-cc
```

Je vérifie que le *patch* puisse être créé correctement :

```
[build-qemu]$ diff  -ruN  base-files-origin/  base-files-modif
diff -ruN base-files-origin/profile base-files-modified/profil
--- base-files-origin/profile   2019-12-28 14:08:38.755388472
+++ base-files-modified/profile 2019-12-28 14:11:21.555616677
@@ -2,7 +2,7 @@
 # and Bourne compatible shells (bash(1), ksh(1), ash(1), ...)

 PATH="/usr/local/bin:/usr/bin:/bin"
-EDITOR="vi"                  # needed for packages like cron
+EDITOR="nano"                # needed for packages like cron
 [ "$TERM" ] || TERM="vt100"   # Basic terminal capab. For scr

 # Add /sbin & co to $PATH for the root user
```

Le *patch* est correct, je l'enregistre en créant un répertoire `base-files/` dans notre *layer* personnalisé, avant de supprimer les deux répertoires temporaires.

```
[build-qemu]$ mkdir  -p  ../meta-my-layer/recipes-core/base-fi
[build-qemu]$ diff  -ruN  base-files-origin/  base-files-modif
```

```
[build-qemu]$ rm  -rf  base-files-origin/  base-files-modified
```

Nous avons obtenu ainsi **notre fichier de patch**. Il doit nécessairement avoir une extension «`.patch`». Il est recommandé de lui donner un nom significatif (par exemple `prefer-nano-to-vi-in-profile`) et l'usage veut que le nom du fichier commence par un nombre qui permettra d'ordonner les *patches* dans le cas où plusieurs sont fournis (les *patches* pouvant modifier successivement les mêmes fichiers, l'ordre d'application est important).

Je crée ensuite une extension de recette dans le répertoire `base-files/` de notre *layer*, en vérifiant au préalable le nom de la recette à surcharger :

```
[build-qemu]$ ls  ../poky/meta/recipes-core/base-files/
base-files  base-files_3.0.14.bb
```

Lorsque le fichier d'extension s'applique à une version spécifique d'une recette (par exemple `3.0.14` uniquement), on le nomme `base-files_3.0.14.bbappend`. Attention, le **caractère souligné (*underscore*) «_»** dans le nom de recette a une véritable signification : il permet de distinguer le nom du *package* (qui ne peut donc pas contenir de souligné, uniquement des tirets «`-`» pour séparer les mots) du numéro de version.

Si l'extension s'applique plusieurs versions de la recette, on utilise le **caractère générique pourcent «%»** dont le rôle rapelle celui de l'astérique «`*`» dans les motifs du shell :

- «`base-files_3.%.bbappend`» s'appliquerait à toutes les recettes de `base-files` dont le numéro majeur de version est 3.
- «`base-files_%.bbappend`» s'applique à toutes les versions.

En toute rigueur un *patch* s'applique à une version spécifique d'un fichier, même s'il est peu probable que le fichier `profile` change beaucoup d'une version à l'autre de la recette `base-files`. Je crée donc la recette suivante :

```
[build-qemu]$ nano ../meta-my-layer/recipes-core/base-files/ba
    FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

```
SRC_URI += "file://001-prefer-nano-to-vi-in-profile.patch'
```

Comme auparavant nous ajoutons le chemin du sous-répertoire contenant notre patch dans `FILESEXTRAPATHS`. Comme notre fichier est le seul de ce nom, il ne surcharge rien, on pourrait donc ajouter notre répertoire à la fin de la variable en utilisant le suffixe «`_append`» au lieu de «`_prepend`». Notons que l'utilisation de «+=» ne fonctionnerait pas, car il utilise toujours une espace comme caractère de séparation et non un deux-points.

On ajoute notre *patch* à la liste des fichiers appartenant à la recette (variable `SRC_URI`. Son extension «`.patch`» suffit à ce qu'il soit pris en compte correctement. On peut relancer la génération de l'image.

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

Et nous testons notre résultat :

```
mybox login: root
Password: (linux)
root@mybox:~# echo  $EDITOR
nano
root@mybox:~#
```

La variable d'environnement est bien initialisée avec la modification apportée par notre *patch*.

Patch sur un fichier source de package

Le *patch* que nous avons développé dans l'exemple précédent était produit et appliqué sur un fichier présent dans une recette. Nous allons à présent voir comment produire **un *patch* s'appliquant sur un fichier source d'un *package*** téléchargé et compilé par une recette.

Pour continuer avec les *packages* que nous avons déjà installés, je vous propose de

travailler sur `nano`. La première chose à faire est de regarder le fichier de recette pour voir comment `bitbake` obtient les sources du package.

```
[build-qemu]$ cat  ../meta-openembedded/meta-oe/recipes-suppor
DESCRIPTION = "GNU nano (Nano's ANOther editor, or \
Not ANOther editor) is an enhanced clone of the \
Pico text editor."
HOMEPAGE = "http://www.nano-editor.org/"
SECTION = "console/utils"
LICENSE = "GPLv3"
LIC_FILES_CHKSUM = "file://COPYING;md5=f27defe1e96c2e1ecd4e0c9

DEPENDS = "ncurses file"
RDEPENDS_${PN} = "ncurses-terminfo-base"

PV_MAJOR = "${@d.getVar('PV').split('.')[0]}"

SRC_URI = "https://nano-editor.org/dist/v${PV_MAJOR}/nano-${PV
SRC_URI[md5sum] = "9650dd3eb0adbab6aaa748a6f1398ccb"
SRC_URI[sha256sum] = "2af222e0354848ffaa3af31b5cd0a77917e9cb77

inherit autotools gettext pkgconfig

PACKAGECONFIG[tiny] = "--enable-tiny,"
```

Nous pouvons remarquer que la variable `SRC_URI` qui décrit la provenance des fichiers source indique l'URL suivante pour le téléchargement.

```
SRC_URI = "https://nano-editor.org/dist/v${PV_MAJOR}/nano-${PV
```

Deux variables sont mises à contribution dans ce chemin : `PV` et `PV_MAJOR`. La **variable PV (*Package Version*)** est omniprésente dans les recettes de Yocto, elle est automatiquement remplie avec le numéro de version du package concerné. Comment ce numéro est-il obtenu ? Simplement grâce au nom du fichier de recette. Il s'agit ici de «`nano_4.4.bb`», donc `PV` est remplie avec la chaîne «`4.4`».

La variable `PV_MAJOR` est définie juste au-dessus de `SRC_URI` grâce à un petit morceau de script Python en-ligne qui extrait la première portion de `PV` en se basant le caractère point «`.`» comme séparateur. Ici, il s'agit donc de «4». Autrement dit, l'URL devient : `https://nano-editor.org/dist/v4/nano-4.4.tar.xz`.

Téléchargeons ce *package* dans notre répertoire de travail :

```
[build-qemu]$ wget https://nano-editor.org/dist/v4/nano-4.4.ta
  [...]
2019-12-29 05:57:57 (5,49 MB/s) - «nano-4.4.tar.xz» enregistre
```

Comme nous l'avions fait précédemment avec le *package* `base-files`, nous extrayons l'archive en deux versions pour utiliser `diff` après modification.

```
[build-qemu]$ tar  xf  nano-4.4.tar.xz
[build-qemu]$ mv  nano-4.4  nano-4.4-origin
[build-qemu]$ tar  xf  nano-4.4.tar.xz
[build-qemu]$ mv  nano-4.4  nano-4.4-modified
[build-qemu]$
```

Pour cet exemple, je cherche à faire une modification simple mais assez facilement visible. Par exemple, je propose de **modifier le texte de bienvenue** qui apparaît lorsqu'on lance «`nano`» sans argument juste au-dessus des deux lignes rappelant les raccourcis clavier. On voit ce texte sur la figure II.3-2.

Nous pouvons trouver le message de bienvenue dans le fichier `nano-4.4/src/nano.c` à la ligne 2688 :

```
statusbar(_("Welcome to nano.  For basic help, type Ctrl+G."));
```

Je le modifie pour le remplacer par :

```
statusbar(_("Welcome to my patched version of nano."));
```

Puis je produis un *patch* entre les deux versions :

```
[build-qemu]$ diff -ruN nano-4.4-origin/ nano-4.4-modified/ >
```

Utilisation de `recipetool`

Maintenant que notre fichier de *patch* est prêt, il nous reste à le stocker dans un répertoire de notre *layer*, puis écrire une extension de recette au format «`.bbappend`» pour `nano` comme nous l'avons déjà fait pour `base-files`.

Un outil fourni par Yocto peut nous aider à réaliser cette tâche un peu répétitive : **`recipetool`**.

Cet utilitaire est conçu pour créer ou modifier des recettes, permettant ainsi d'ajouter assez facilement des *patches*, des fragments de configuration, etc.
Personnellement, je l'utilise surtout pour modifier la configuration du *kernel* Linux ou de `busybox`.

Nous allons demander à `recipetool` de créer une extension pour la recette de `nano` dans le *layer* `meta-my-layer` et d'ajouter le *patch* que nous venons de produire.

```
[build-qemu]$ recipetool  appendsrcfile  ../meta-my-layer/  na
NOTE: Starting bitbake server...
  [...]
NOTE: Writing append file /home/cpb/Yocto-lab/meta-my-layer/re
NOTE: Copying 001-modified-welcome-message.patch to /home/cpb/
```

Nous voyons que `recipetool` a créé un sous-répertoire de notre *layer* pour `nano`. Il l'a placé dans l'arborescence `recipes-support/`, respectant ce qui existait dans le *layer* `meta-openembedded` original.

```
[build-qemu]$ ls  ../meta-my-layer/
conf         README        recipes-custom   recipes-support
COPYING.MIT  recipes-core  recipes-example
[build-qemu]$ ls  ../meta-my-layer/recipes-support/
nano
[build-qemu]$ ls  ../meta-my-layer/recipes-support/nano/
nano  nano_4.4.bbappend
```

Je n'ai pas utilisé l'option «`-w`» (comme *wildcard*, caractère générique) de `recipetool`, aussi celui-ci crée une extension de recette pour la version exacte de

`nano` présente dans le *layer* original (4.4).

Le sous-répertoire `nano/` du répertoire d'extension joue le même rôle que le sous-répertoire `files/` que nous avions créé avec `base-files` : il abrite le *patch* à appliquer.

```
[build-qemu]$ ls  ../meta-my-layer/recipes-support/nano/nano
001-modified-welcome-message.patch
```

L'extension de recette est simple et rappelle celle que nous avions écrite précédemment :

```
[build-qemu]$ cat  ../meta-my-layer/recipes-support/nano/nano_
SRC_URI += "file://001-modified-welcome-message.patch;subdir=r

FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

Nous pouvons relancer le *build* de notre système :

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

Puis sur notre cible virtuelle, nous pouvons admirer le résultat de notre *patch* en lançant la commande `nano`, comme on le voit sur la figure II.3-3.

Conclusion

Nous avons vu dans cette séquence comment modifier le comportement de recettes existantes sans toucher aux fichiers originaux, avec différentes approches selon le type de modification à apporter.

Le principe des extensions grâce aux fichiers `.bbappend` est très puissant et permet de garantir la pérennité des modifications que l'on apporte même si les versions des *packages* d'origine évoluent.

Les deux premières parties de ce cours en ligne nous ont permis de voir comment créer une image de Linux embarqué pour une architecture cible de notre choix, et d'ajuster son contenu. Il est temps à présent de s'intéresser à l'ajout de notre propre code, ce qui sera l'objet de la troisième partie.

provenance.