III.1 – Ajouter des scripts ou des fichiers de données

Christophe BLAESS - janvier 2020

Le premier type de fichiers que nous allons ajouter au BSP que nous avons produit jusqu'à présent va être assez simple : il s'agira de **scripts**. L'avantage de ces fichiers est qu'ils ne nécessitent pas de compilation, il suffit de les copier au bon endroit, et avec les bonnes permissions, pour qu'ils soient utilisables sur la cible.

Bien sûr il faut qu'un **interpréteur** adéquat soit disponible pour les interpréter. Pour les scripts ***shell*** cela fait partie de l'environnement de base installé par Yocto, et pour les scripts **Python**, nous devront nous assurer de la présence de l'interpréteur et de certains modules supplémentaires.

Système de fichiers en lecture seule

Avant cela, nous allons commencer par sélectionner une *feature* d'image importante : **la mise en lecture seule du système de fichiers**. Dans les environnements embarqués, une préoccupation constante est en effet la corruption des données lors d'une coupure impromptue de l'alimentation électrique du système. Lorsqu'elle se produit, cette coupure provoque bien évidemment la perte des données qui étaient en cours d'enregistrement, mais peut engendrer une corruption du système de fichiers lui-même. Cela se traduit par une vérification longue au *boot* suivant, voire une impossibilité de monter le système de fichiers et un blocage du système.

Une manière simple de corriger ce défaut est de demander à Yocto de monter le

système de fichiers contenu sur la partition principale en lecture-seulement. Bien sûr on pourra ajouter des partitions supplémentaires en lecture-écriture pour y stocker des données si besoin, en s'arrangeant pour qu'une détection d'incohérence ne soit pas bloquante, nous en reparlerons. L'important est que la partition principale du système soit protégée.

J'édite donc mon fichier d'image pour ajouter la *feature* adéquate :

```
[build-qemu]$ nano  ../meta-my-layer/recipes-custom/images/my-
   [...]
   IMAGE_FEATURES += "read-only-rootfs"
```

Après re-compilation et démarrage, vérifions le comportement :

```
mybox login: root
Password: (linux)
root@mybox:~# pwd
/home/root
root@mybox:~# echo  HELLO  > my-file.txt
-sh: my-file.txt: Read-only file system
```

Pendant la phase de mise au point du système embarqué, il est possible de remonter temporairement la partition principale en lecture-écriture ainsi :

```
root@mybox:~# mount  /  -o  remount,rw
 [188.917856] EXT4-fs (vda): re-mounted. Opts: (null)
root@mybox:~# echo  HELLO  > my-file.txt
root@mybox:~# ls
my-file.txt
```

Puis de revenir en lecture-seule :

```
root@mybox:~# mount  /  -o  remount,ro
 [260.995398] EXT4-fs (vda): re-mounted. Opts: (null)
root@mybox:~# rm  my-file.txt
rm: cannot remove 'my-file.txt': Read-only file system
```

Néanmoins ces lignes de commandes sont un peu fastidieuses, aussi je vous propose de réaliser deux petits scripts, que j'appelle traditionnellement «`rw`» et «`ro`» pour réaliser ces tâches. Ceci nous donnera l'occasion d'écrire une recette pour les intégrer dans notre image.

Intégration de scripts shells

Commençons par écrire les deux **scripts *shell*** dans un répertoire dédié de notre *layer*. Vous pouvez les télécharger ici : `rw` et `ro`.

```
[build-qemu]$ mkdir  -p  ../meta-my-layer/recipes-custom/my-sc
[build-qemu]$ nano  ../meta-my-layer/recipes-custom/my-scripts
  #! /bin/sh
  mount / -o remount,rw
[build-qemu]$ nano  ../meta-my-own-layer/recipes-custom/my-scr
  #! /bin/sh
  mount / -o remount,ro
```

Inutile de les rendre exécutables, nous ne les lancerons pas sur le système de développement, et leurs permissions sur la cible seront indiquées directement dans notre recette. Nous allons créer manuellement le fichier de recette (sans passer par `recipetool` par exemple) afin d'en analyser la structure. Créons le fichier de suivant :

```
[build-qemu]$ nano  ../meta-my-layer/recipes-custom/my-scripts
SUMMARY = "Custom scripts for Yocto training"
SECTION = "custom"

LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835a

SRC_URI  = "file://ro"
SRC_URI += "file://rw"

do_install() {
     install -d ${D}${sbindir}
     install -m 0755 ${WORKDIR}/ro ${D}${sbindir}
     install -m 0755 ${WORKDIR}/rw ${D}${sbindir}
}
```

```
FILES_${PN} += "${sbindir}/ro"
FILES_${PN} += "${sbindir}/rw"
```

Quelques explications sur la syntaxe de notre première recette complète :

- Les variables **`SUMMARY`** et **`SECTION`** sont purement informatives et permettent de ranger la recette et la classer lorsqu'elle est présentée dans une liste comme le site *Open Embedded Layer Index*.
- Yocto accorde beaucoup d'importance aux **licences** des *packages* intégrés dans les images qu'il produit. Son orientation industrielle nécessite une attention particulière pour s'assurer de la compatibilité du code métier (le plus souvent sous licence propriétaire) avec les outils et bibliothèques — généralement libres — employés. Il est donc nécessaire de déclarer dans la variable **`LICENSE`** le type de licence, et dans **`LIC_FILES_CHKSUM`** le nom du fichier la décrivant ainsi qu'une somme de contrôle de ce dernier pour s'assurer que la version est bien identique. Le projet Yocto lui-même est distribuée sous licence MIT (libre et sans *copyleft*), aussi de nombreuses recettes emploient-elles la même licence. C'est le choix que j'ai fait ici. Pour éviter de fournir une version du fichier de licence, j'ai fait référence à celui fourni dans le sous-dossier `meta/files/common-licenses/` de Poky (qui contient près de deux cents variantes des licences les plus courantes). La *checksum* est obtenue avec la commande `md5sum ../poky/meta/files/common-licenses/MIT`.
- La variable **`SRC_URI`** contient une liste des fichiers sources utilisés par la recette ainsi que leur origine. Ici les fichiers sont livrés directement avec la recette aussi sont-ils préfixés «`file://`» mais d'autres descripteurs de provenance sont disponibles («`ftp://`», «`git://`», «`svn://`», etc.). Nous pouvons remarquer l'usage de l'opérateur «+=» pour ajouter un élément dans la liste.
- Lorsqu'il traite une recette, Yocto suit un certain nombre d'étapes (`fetch` pour télécharger les sources, `unpack` pour les décompresser, `patch` pour les modifier éventuellement, `compile` pour produire le code exécutable, `install` pour les placer dans l'arborescence, etc.). Ces étapes sont représentées par des fonctions **`do_fetch()`**, **`do_compile()`**, etc. qui ont des implémentations par défaut mais que nous pouvons surcharger. C'est ce que je fais ici en implémentant la fonction **`do_install()`**. Cette dernière fait appel à l'utilitaire «`install`» (une sorte de «`cp`» plus complet) pour placer les scripts aux endroits voulus. On notera que l'on prend soin de créer d'abord les répertoires (même standards) que l'on utilise avec l'option «`-d`» (*directory*) de la commande «`install`».
- La variable **`WORKDIR`** est automatiquement initialisée avec l'emplacement où

Yocto travaille pour produire la recette. C'est l'endroit où il a placé les fichiers sources obtenus dans l'étape `do_fetch()`. Notre travail consiste à aller les installer dans le répertoire désiré de l'arborescence de la cible. Le point de départ de cette arborescence est toujours représenté par la **variable `D`**. Et la variable **`sbindir`** contient l'emplacement recommandé pour les utilitaires d'administration (répertoire `/usr/sbin` généralement).

- Yocto veut s'assurer que chaque fichier placé dans l'arborescence finale de la cible peut être identifié et attribué à un *package*. Pour cela la recette remplit sa **variable `FILES`** avec la liste des fichiers dont elle est responsable. On utilise **l'opérateur `_${PN}`** (*Package Name*) pour remplir la variable correspondant à ce package.

Nous avons utilisé la variable `sbindir`, il existe tout une liste de variables prédéfinies avec des chemins standards. Voici ci-dessous les plus courantes, pour en savoir plus, se reporter au début du fichier `poky/meta/conf/bitbake.conf`.

Variable	Valeur par défaut
`bindir`	`/usr/bin`
`datadir`	`/usr/share`
`docdir`	`/usr/share/doc`
`includedir`	`/usr/include`
`infodir`	`/usr/share/info`
`libdir`	`/usr/lib`
`libexecdir`	`/usr/libexec`
`localedir`	`/usr/lib/locale`
`localstatedir`	`/var`
`sbindir`	`/usr/sbin`

servicedir	/srv
sysconfdir	/etc
systemd_unitdir	/lib/systemd
systemd_system_unitdir	/lib/systemd/system
systemd_user_unitdir	/usr/lib/systemd/user

Avant de recompiler l'image, il convient d'inclure notre *package* dans le fichier d'image :

```
[build-qemu]$ nano  ../meta-my-layer/recipes-custom/images/my-
  [...]
  IMAGE_INSTALL_append = " my-scripts"
```

Après compilation et démarrage de la cible, nous vérifions que nos scripts sont bien accessibles et se comportent comme voulu :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyAMA0

mybox login: root
Password: (linux)
root@mybox:~# ls
root@mybox:~# echo  HELLO  > my-file.txt
-sh: my-file.txt: Read-only file system
root@mybox:~# rw
[   81.725260] EXT4-fs (vda): re-mounted. Opts: (null)
root@mybox:~# echo  HELLO  > my-file.txt
root@mybox:~# ls
my-file.txt
root@mybox:~# ro
[   87.612135] EXT4-fs (vda): re-mounted. Opts: (null)
root@mybox:~# ls
my-file.txt
root@mybox:~# rm my-file.txt
```

```
rm: cannot remove 'my-file.txt': Read-only file system
root@mybox:~#
```

NB: les messages "*EXT4-fs (vda): re-mounted. Opts: (null)*" ne sont pas des erreurs mais des traces du noyau (nous sommes connectés sur la console principale) lorsqu'une partition est montée ou démontée.

Scripts et modules Python

Nous pouvons facilement ajouter des **scripts Python** sur notre cible, un interpréteur Python 3 étant déjà présent dans notre image, mais le nombre de modules installés par défaut est assez limité. Il est donc possible d'ajouter la ligne suivante dans notre fichier `my-image.bb` pour enrichir l'installation.

```
IMAGE_INSTALL_append = " python-modules"
```

Il n'existe pas de *package* `python-modules` à proprement parler, il s'agit d'une liste de modules déclarée dans la recette de `python` fournie par Poky. Après recompilation, nous pouvons observer qu'un interpréteur Python 2.7 est aussi présent.

```
root@mybox:~# python  -V
Python 2.7.17
root@mybox:~# python3  -V
Python 3.7.5
root@mybox:~#
```

Je rajoute donc, dans un répertoire `meta-my-layer/recipes-custom/python-hello/files/` le petit script `python-hello.py` suivant :

```
#! /usr/bin/python
#
# Christophe BLAESS 2020.
#
# Licence MIT.
#
from __future__ import print_function
```

```
import socket
import sys
print("Python", sys.version[0:3], "says 'Hello' from", socket.
```

Ainsi que la recette `python-hello_1.0.bb` ci-dessous, dans le répertoire `meta-my-layer/recipes-custom/python-hello`.

```
SUMMARY = "Python Hello World for Yocto tutotial"
SECTION = "custom"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835a
SRC_URI  = "file://python-hello.py"
do_install() {
    install -d ${D}${bindir}
    install -m 0755 ${WORKDIR}/python-hello.py ${D}${bindir}
}
FILES_${PN} += "${bindir}/python-hello.py"
RDEPENDS_python-hello += "python"
RDEPENDS_python-hello += "python-modules"
```

On peut noter la présence des deux dernières lignes, qui renseignent la **variable `RDEPENDS`** pour cette recette. Cette variable, abréviation de *Runtime Dependencies* contient la liste des *packages* nécessaires pour le fonctionnement de la recette **sur la cible**. Il faut la distinguer de la **variable `DEPENDS`** qui liste les *packages* nécessaires **pour compiler** une recette sur la machine de développement. Le package `python-modules` doit être installé sur la cible pour deux raisons :

- disposer du module `socket` qui nous fournit la fonction `gethostname()`,
- avoir un interpréteur `/usr/bin/python` (lien vers `python2.7`) afin de disposer d'un script avec une ligne *shebang* indépendante de la version de l'interpréteur.

En ajoutant dans notre image la ligne :

```
IMAGE_INSTALL_append = " python-hello"
```

et en recompilant notre système, nous pouvons observer après démarrage :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyAMA0
```

```
mybox login: root
Password: (linux)
root@mybox:~# python-hello.py
Python 2.7 says 'Hello' from mybox
root@mybox:~#
```

Conclusion

Nous avons vu dans cette séquence comment ajouter dans une image des fichiers fournis par une recette. Dans le cas d'un script, il faut simplement penser à lui ajouter les permissions d'exécution (ce qui se fait facilement avec la commande install) et éventuellement l'interpréteur nécessaire. Dans la séquence suivante, nous écrirons une recette nécessitant une compilation de code source.

«« **sommaire** »»