III.3 – Intégrer notre application dans la production de Yocto

Christophe BLAESS - janvier 2020

Pour le moment, nous avons compilé notre code applicatif métier à l'extérieur de Yocto, en utilisant la *toolchain* extraite (et installée éventuellement sur une autre machine de développement). Ceci est parfaitement acceptable — et même recommandé — pendant les phases initiales de codage, mise au point, et débogage pour simplifier le travail du développeur et réduire les temps de compilation.

Une fois le projet suffisament stable pour les premiers déploiements, il est intéressant d'incorporer la compilation du code métier directement dans la production de l'image embarquée. Généralement cela concerne une branche stable du code source, tandis que des branches de développement, d'expérimentation et de test coexistent dans le système de gestion de versions.

Dans cette séquence nous allons voir comment **écrire une recette** simple afin d'intégrer directement notre code applicatif dans l'image produite par `bitbake`.

Recette utilisant les *Autotools*

Les recettes les plus simples à écrire sont celles qui font appel aux méthodes des outils de construction standards comme les *Autotools*. En effet, Yocto contient déjà des classes (des fichiers «`.bbclass`» situés dans `poky/meta/classes/`) capables de compiler ce type de projet. Il suffit que notre recette demande à hériter des méthodes de la classe.

Je commence par créer un répertoire `hello-autotools` dans mon layer.

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

Puis j'y écris un fichier de recette (certains outils comme `recipetool` peuvent d'ailleurs créer une recette de base) :

```
[build-qemu]$ nano  ../meta-my-own-layer/recipes-custom/hello-
```

Le fichier de recette contient les lignes suivantes :

```
DESCRIPTION = "A simple hello world application built with aut
SECTION = "Custom"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;md5=8

SRC_URI="https://www.blaess.fr/christophe/yocto-lab/files/${BP
SRC_URI[md5sum] = "572c660dcbf1cbd8ac1018baba26f3ea"

S = "${WORKDIR}/${PN}"

inherit autotools
```

Ce fichier mérite quelques explications :

- Nous avons déjà parlé des variables `DESCRIPTION`, `SECTION`, `LICENSE` et `LIC_FILES_CHKSUM` dans la séquence III.1.
- **La variable `SRC_URI`** contient le lien pour télécharger l'archive du projet. On y inscrit ici l'URL et le nom de l'archive. J'ai utilisé ici **la variable `BP`** qui est équivalente à «`${BPN}-${PV}`» soit le nom de base du *package* (`BPN`) suivi du numéro de version (`PV`). Les variables `BPN` et `PV` sont remplies par Yocto à partir du nom du fichier de recette (ici `hello-autotools_1.0.bb`) en le découpant au niveau du caractère souligné (*underscore*) «_». Yocto connaît les types d'URL courant (`http://`, `https://`, `ftp://`, `git://`, `svn://`, etc.) ainsi que les principales extensions de compression (`.tar.gz`, `.zip`, `.tar.bz2`, etc.). Il s'occupe donc de télécharger et d'extraire les fichiers sources.

- Lorsque les fichiers sources ne sont pas fournis avec la recette (dans un répertoire `files/` par exemple), et qu'ils sont téléchargés, il est nécessaire de vérifier l'intégrité du *package*, pour cela **la variable `SRC_URI[md5sum]`** fournit une somme de contrôle *MD5* (obtenue avec l'utilitaire `md5sum`).
- **La variable «S»** (comme *Sources*) indique l'emplacement où `bitbake` doit se placer pour réaliser les étapes de la compilation. Le *package* a été décompressé à l'intérieur du **répertoire «`$WORKDIR`»**, mais suivant les projets l'emplacement des sources peut varier. Parfois il y aura un simple répertoire avec le nom du projet (c'est le cas ici, avec `hello-autotools/`), parfois ce nom sera complété d'un numéro de version (du type `hello-autotools-1.0/`), parfois il y aura un sous-répertoire supplémentaire (par exemple `hello-autotools/src/`) ou encore le package sera directement déployé dans `$WORKDIR`. Quelque soit la situation, on remplit la variable «S» en conséquence.
- **La ligne «`inherit autotools`»** enfin indique que la recette doit hériter des méthodes de la classe `autotools` pour les étapes `do_compile()`, `do_install()`, etc.

Il nous suffit à présent d'ajouter la ligne suivante dans notre fichier image.

```
IMAGE_INSTALL_append = " hello-autotools"
```

Après re-compilation, notre projet est bien installé :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyAMA0
mybox login: root
Password: (linux)
root@mybox:~# hello-autotools
Hello from mybox (built with autools)
root@mybox:~#
```

Recette utilisant *CMake*

Les projets que l'on compile avec **l'environnement *CMake*** peuvent facilement être incorporés à Yocto, car il existe une classe «`cmake`» prenant en charge ce type de production. Je crée le répertoire et la recette suivante :

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

```
[build-qemu]$ nano  ../meta-my-layer/recipes-custom/hello-cmak
DESCRIPTION = "A simple hello world application built with CMa
SECTION = "Custom"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;md5=8

SRC_URI="https://www.blaess.fr/christophe/yocto-lab/files/${BP
SRC_URI[md5sum] = "88c263b9a5d53f3cbfdb24a7f7f4b583"

S = "${WORKDIR}/${PN}"

inherit cmake
```

La principale différence (hormis la description et la somme de contrôle du package) est la dernière ligne. Bien sûr, on ajoute dans le fichier d'image la ligne suivante :

```
IMAGE_INSTALL_append = " hello-cmake"
```

Et après compilation et *boot*, on observe :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyAMA0
mybox login: root
Password: (linux)
root@mybox:~# hello-cmake
Hello from mybox (built with CMake)
root@mybox:~#
```

Recette pour un projet avec `Makefile` personnalisé

Il arrive fréquemment qu'un projet dispose d'**un `Makefile` spécifique**, directement livré avec les sources, sans passer par la phase de configuration des *Autotools* ou de *CMake*.

Supposons que ce `Makefile` soit suffisamment bien conçu pour tirer parti des variables configurées par le script livré avec la *toolchain* (voir ce paragraphe de la séquence précédente pour plus de détails).

Nous pouvons dans ce cas utiliser la commande `oe_runmake()` de Yocto pour

encadrer l'appel à `make` en vérifiant sa réussite. Bien entendu, des paramètres ou des cibles spécifiques de `make` peuvent être spécifiées en argument.

Prenons l'exemple d'un projet avec un `Makefile` très simple, proposant notamment les commandes :

- `make` : pour compiler un programme,
- `make install` : pour l'installer dans le répertoire indiqué dans la variable `DESTDIR`.

Nous pouvons créer une recette qui télécharge, compile et installe l'exécutable en développant les étapes `do_compile()` et `do_install()`. À nouveau je crée un répertoire dans mon *layer* pour accueillir la recette :

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

Puis j'y place le fichier de recette hello-makefile_1.0.bb suivant :

```
DESCRIPTION = "A simple hello world application built with a M
SECTION = "Custom"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;md5=8

SRC_URI="https://www.blaess.fr/christophe/yocto-lab/files/${BP
SRC_URI[md5sum] = "3bd7b83d110df1517f0e7fa8f0dbea80"

S = "${WORKDIR}/${PN}"

do_compile() {
        oe_runmake
}

do_install() {
        oe_runmake DESTDIR="${D}${bindir}" install
}
```

Après ajout de «`IMAGE_INSTALL_append = " hello-makefile"`» dans mon fichier d'image, je relance le *build* et au démarrage de l'émulateur je peux exécuter :

```
mybox login: root
Password: (linux)
root@mybox:~# hello-makefile
Hello from mybox (build with standard Makefile)
root@mybox:~#
```

Recette pour un projet sans `Makefile`

Finalement, les recettes les plus complexes sont celles qui doivent compiler des **packages ne comportant aucun `Makefile`** (ou un `Makefile` mal fichu, qui ne permet pas la *cross-compilation*). Pour nous placer dans la situation la plus défavorable, imaginons un projet ne comportant qu'un fichier source à télécharger depuis un site web, avec le contenu suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/param.h>

#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN  64
#endif

int main(void)
{
        char hostname[MAXHOSTNAMELEN];

        gethostname(hostname, MAXHOSTNAMELEN);
        hostname[MAXHOSTNAMELEN - 1] = '\0';

        fprintf(stdout, "Simple hello from %s (built without M

#ifndef NDEBUG
        fprintf(stdout,"  [DEBUG MODE]\n");
#endif

#ifdef MY_OPTION
        fprintf(stdout, "  [MY OPTION]\n");
#endif
```

```
        return EXIT_SUCCESS;
}
```

Le fichier ci-dessus prend en considération deux constantes symboliques qui peuvent être précisées lors de la compilation :

- Par convention, on active la constante symbolique `NDEBUG` (*No Debug*) pour indiquer que le code n'est plus en mode «débogage» et que certains messages de traces et certaines vérifications peuvent être ignorés.
- J'ai ajouté une constante `MY_OPTION` qui nous permettra de vérifier le comportement en affichant un message spécifique si elle est présente lors de la compilation.

Je crée un répertoire `hello-simple` dans mon *layer*, et j'ajoute la recette `hello-simple.bb` suivante, dans laquelle la compilation est explicitement réalisée par la méthode `do_compile()` et la copie de l'exécutable par `do_install()`. Le fichier source initial n'ayant pas de numéro de version, on voit que le fichier de recette est nommé simplement avec le nom du *package*.

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

Voici le contenu du fichier de recette :

```
DESCRIPTION = "A simple hello world application built without
SECTION = "Custom"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;md5=8

SRC_URI="https://www.blaess.fr/christophe/yocto-lab/files/${BP
SRC_URI[md5sum] = "40ae84f9ff78004561227a42f27c8131"

S = "${WORKDIR}/"

TARGET_CFLAGS += "-DNDEBUG -DMY_OPTION"

do_compile() {
        ${CC} ${CFLAGS} ${LDFLAGS} ${S}/${BPN}.c -o ${WORKDIR}
}
```

```
do_install() {
        install -m 0755 -d ${D}${bindir}
        install -m 0755 ${WORKDIR}/${BPN} ${D}${bindir}
}
```

Comme on le voit, la fonction `do_compile()` fait directement appel au compilateur C, avec les options nécessaires pour la compilation et l'édition des liens. La variable `CFLAGS` est automatiquement renseignée avec le contenu de la variable `TARGET_CFLAGS` pendant la *cross-compilation* pour la cible. Dans cette variable, nous avons défini les deux constantes `NDEBUG` et `MY_OPTION`. Le code source du programme appelle `fprintf()` en fonction de la présence de ces constantes. Bien entendu, pour des projets réels, on est amené à définir des options plus complètes dans `TARGET_CFLAGS`, et éventuellement `TARGET_LDFLAGS`.

La recette ci-dessus ne fait pas référence au nom du fichier source, elle utilise seulement le nom du package à travers la variable «`${BPN}`», elle pourrait donc être adaptée facilement (à la somme de contrôle MD5 près) pour n'importe quel fichier à compiler de cette manière. Après compilation, nous pouvons vérifier que les deux options ont bien été passées au compilateur. Le premier message ne s'affiche que si l'option n'est pas définie, il est donc normal qu'il ne soit pas présent ici.

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

mybox login: root
Password: (linux)
root@mybox:~# hello-simple
Simple hello from mybox (built without Makefile)
  [MY OPTION]
root@mybox:~#
```

Conclusion

Nous avons vu dans cette séquence différentes manières d'intégrer dans le *build* de Yocto des applications externes, qu'il s'agisse de projets développés spécifiquement ou des utilitaires libres glanés sur Internet (dans ce cas, il est toutefois préférable de s'assurer que la recette ne soit pas déjà disponible quelque part). Cette intégration de code personnalisé dans l'image produite par Yocto est généralement réalisée en fin de projet, car il est plus simple pendant la phase de mise au point d'assurer la compilation indépendamment de `bitbake`, comme nous l'avons vu dans la séquence précédente.

Ainsi s’achève la troisième partie de ce cours en ligne, après le développement du code métier, nous allons nous intéresser à l’ajustement de l'image pour une cible spécifique

«« **sommaire** »»