III.2 – Compiler une application métier en dehors de Yocto

Christophe BLAESS - janvier 2020

Nous avons vu comment produire une image dont le contenu est personnalisé, et nous savons y ajouter automatiquement des fichiers existants (**des scripts** par exemple) sur notre machine de production.

Il est rare qu'un système embarqué ne contienne que du code métier écrit sous forme de scripts. Ceci est possible (notamment si les applications sont développées en Python, NodeJS, etc.) mais la plupart du temps, il sera nécessaire de compiler au moins une partie de l'applicatif métier.

Yocto, comme tout environnement Linux, propose une chaîne de compilation Gnu complète, contenant des **compilateurs C, C++, Java**, et même Ada ou Fortran si on le souhaite. Dans cette séquence nous allons extraire cette toolchain afin de pouvoir compiler notre code métier indépendamment de Yocto, puis installer manuellement sur la cible les fichiers exécutables produits.

Cette approche est couramment employée lors de la phase de mise au point et débogage du code métier. Une fois que l'applicatif est bien finalisé, il sera intéressant de l'intégrer directement dans le *build* de Yocto, ce sera le sujet de la prochaine séquence.

Extraction de la *toolchain*

La première étape consiste à demander à Yocto de nous fournir une chaîne de compilation que nous pourrons utiliser indépendamment de `bitbake`, des recettes et des *layers* que nous avons vus jusqu'à présent.

La plupart du temps notre système cible sera construit autour d'un processeur (par exemple un cœur ARM) différent de celui équipant notre système de production (généralement dans la gamme x86-64). Il faut donc disposer d'une **chaîne de *cross-compilation***, c'est-à-dire un ensemble comprenant le compilateur, l'assembleur, l'éditeur de liens, l'archiveur, le débogueur, etc. fonctionnant sur notre PC mais produisant du code binaire pour une architecture différente.

Il est possible de trouver sur le web des chaînes de *cross-compilation* toutes prêtes, mais il est largement préférable de prendre celle qui a été produite par Yocto lors de sa première compilation.

Vous vous souvenez, du premier appel à `bitbake` qui durait plusieurs heures ?

S'il était si long, c'est justement parce qu'il préparait ses propres outils de compilation. Demandons-lui de les partager avec nous…

Pour cela nous appelons `bitbake` en lui ajoutant une option supplémentaire «**`-c populate_sdk`**». Celle-ci lui demande de nous fournir le SDK (*Software Development Kit*), l'ensemble des outils nécessaires pour produire du code métier pour notre image. Cet ensemble est un peu plus riche que la simple chaîne de compilation, puisqu'il contient également les fichiers *headers* pour les bibliothèques supplémentaires ajoutées dans l'image.

```
[build-qemu]$ bitbake  -c populate_sdk  my-image
[...]
[build-qemu]$ ls  tmp/deploy/
 images  licenses  rpm  sdk
```

La durée de cette préparation est très variable en fonction de ce qui a été compilé au préalable. Ne soyez pas surpris si elle atteint une heure, voire plus. Nous voyons qu'un nouveau sous-dossier «`sdk/`» est apparu dans le répertoire `tmp/deploy/`. Examinons son contenu :

```
[build-qemu]$ ls tmp/deploy/sdk/
poky-glibc-x86_64-my-image-armv7vet2hf-neon-qemuarm-toolchain-
poky-glibc-x86_64-my-image-armv7vet2hf-neon-qemuarm-toolchain-
poky-glibc-x86_64-my-image-armv7vet2hf-neon-qemuarm-toolchain-
poky-glibc-x86_64-my-image-armv7vet2hf-neon-qemuarm-toolchain-
```

```
[build-qemu]$
```

Quatre fichiers sont donc présents. Les deux fichiers aux suffixes «`.manifest`» sont des listes des *packages* et bibliothèques installés sur la cible et sur le poste de développement. Le fichier «`.testdat.json`» contient la configuration (variables d'environnement) utilisée pour produire la *toolchain*. Le fichier qui nous intéresse le plus est le script d'extension «`.sh`». Il occupe quand même plus de 600 Mo, car il encapsule toute la chaîne de compilation sous forme binaire.

L'intérêt d'**extraire cette *toolchain*** est que nous pouvons l'installer sur une machine différente de celle que nous avons employée pour faire fonctionner `bitbake`. Dans le cadre d'un développement embarqué industriel c'est très utile car cela permet à l'équipe système de se concentrer sur l'infrastructure bas-niveau (utilitaires, bibliothèques, noyau, support matériel, etc.) tout en fournissant aux développeurs de l'équipe applicative l'environnement pour compiler et tester leur code métier.

Le SDK peut être installé sur toute machine dont l'architecture est compatible avec celle qui a produit le script (ici `x86_64` avec la bibliothèque `glibc`). On peut parfois rencontrer des soucis de compatibilité des versions de la bibliothèque C, que l'on contourne souvent en fournissant aux développeurs applicatifs une machine virtuelle contenant la bonne version de la *libC* et la *toolchain* préinstallée (machine virtuelle qu'ils peuvent d'ailleurs employer sur un autre système d'exploitation que Linux).

Je vais installer ici la *toolchain* sur la même machine que celle qui m'a servi à préparer le SDK. Je copie néanmoins le script dans un autre emplacement avant de l'exécuter.

```
[build-qemu]$ cp  tmp/deploy/sdk/poky-glibc-x86_64-my-image-ar
[build-qemu]$ cd
[~]$ ./poky-glibc-x86_64-my-image-armv7vet2hf-neon-qemuarm-toc
Poky (Yocto Project Reference Distro) SDK installer version 3.
Enter target directory for SDK (default: /opt/poky/3.0.1):
```

Le script nous demande où installer la *toolchain*. Il nous propose par défaut une installation dans «`/opt`». Ceci est parfaitement adapté lorsque plusieurs développeurs travaillent sur la même machine et sont tous amenés à employer cette chaîne de compilation. L'installation nécessite alors les droits *root*, ce que le script obtient en invoquant «`sudo`».

Dans le cadre de ce cours, je serai le seul à appeler la *toolchain*, aussi préféré-je l'installer directement dans mon répertoire personnel. Je fournis donc un chemin personnalisé. J'incorpore dans le chemin le nom du projet (`yocto-lab`) car mon

répertoire `~/sdk` contient déjà plusieurs *toolchains* appartenant à différents projets de mes clients.

```
Enter target directory for SDK (default: /opt/poky/3.0.1): /ho
You are about to install the SDK to "/home/cpb/sdk/yocto-lab".
```

Après confirmation, le script passe à l'installation de la *toolchain* pendant quelques instants.

```
Extracting SDK.....................................done
Setting it up......done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you
 $ . /home/cpb/sdk/yocto-lab/environment-setup-armv7vet2hf-neo
[~]$
```

À la fin de l'installation, un commentaire nous indique qu'un script est installé dans le répertoire indiqué, et qu'il devra être appelé avant chaque session de travail employant notre *toolchain*. Ce script configure les variables d'environnement nécessaires à l'invocation des différents outils de la chaîne de compilation. Voici l'état de quelques unes de ces variables avant l'appel du script.

```
[~]$ echo $PATH
/home/cpb/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bi
[~]$ echo $ARCH

[~]$ echo $CROSS_COMPILE

[~]$ echo $CC

[~]$ echo $CFLAGS

[~]$ echo $CXX

[~]$ echo $CXXFLAGS

[~]$ echo $LD

[~]$ echo $LDFLAGS
```

```
[~]$ echo $AS

[~]$ echo $GDB

[~]$
```

La plupart des variables concernées sont vides. Le script doit être "sourcé" pour modifier l'environnement du *shell* appelant. Voici les mêmes variables après son exécution.

```
[~]$ source  ~/sdk/yocto-lab/environment-setup-armv7vet2hf-nec
[~]$ echo $PATH
/home/cpb/sdk/yocto-lab/sysroots/x86_64-pokysdk-linux/usr/bin:
[~]$ echo $ARCH
arm
[~]$ echo $CROSS_COMPILE
arm-poky-linux-gnueabi-
[~]$ echo $CC
arm-poky-linux-gnueabi-gcc -march=armv7ve -mthumb -mfpu=neon -
[~]$ echo $CFLAGS
-O2 -pipe -g -feliminate-unused-debug-types
[~]$ echo $CXX
arm-poky-linux-gnueabi-g++ -march=armv7ve -mthumb -mfpu=neon -
[~]$ echo $CXXFLAGS
-O2 -pipe -g -feliminate-unused-debug-types
[~]$ echo $LD
arm-poky-linux-gnueabi-ld --sysroot=/home/cpb/sdk/yocto-lab/sy
[~]$ echo $LDFLAGS
-Wl,-O1 -Wl,--hash-style=gnu -Wl,--as-needed
[~]$ echo $AS
arm-poky-linux-gnueabi-as
[~]$ echo $GDB
arm-poky-linux-gnueabi-gdb
[~]$
```

Toutes ces variables sont assez standards dans les `Makefile`. Voici leurs significations :

Nom	Signification
`ARCH`	Architecture de la cible. Surtout utilisé pour la compilation du noyau.
`CROSS_COMPILE`	Préfixe à ajouter aux noms des outils standards. Employé par certains `Makefile` dont celui du noyau.
`PATH`	Liste des répertoires à parcourir pour rechercher les exécutables.
`C`	Compilateur C.
`CFLAGS`	Options du compilateur C.
`CXX`	Compilateur C++.
`CXXFLAGS`	Options du compilateur C++.
`CPP`	Préprocesseur C/C++.
`CPPFLAGS`	Options du préprocesseur C/C++.
`AS`	Assembleur.
`LD`	Éditeur de liens (*linker*).
`LDFLAGS`	Options du *linker*.
`AR`	Archiveur.
`STRIP`	Suppresseur de symboles.
`RANLIB`	Indexeur de fichiers d’archives (bibliothèques).

`NM`	Visualisation des symboles d'un fichier objet.
`M4`	Macro-processeur employé par les *autotools* et pour la documentation.

Vérifions la présence et le fonctionnement du compilateur C par exemple :

```
[~]$ $CC  --version
arm-poky-linux-gnueabi-gcc (GCC) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTIC
[~]$
```

Compilation manuelle d'un fichier source C

Nous pouvons prendre le petit fichier source suivant nommé «`my-hello.c`»:

```
#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, "Hello from %s\n", hostname);

        return EXIT_SUCCESS;
```

```
}
```

Tout d'abord **nous le compilons pour la machine de développement** afin de vérifier son bon fonctionnement. C'est un avantage qu'offre l'utilisation de Linux sur la machine hôte comme sur la cible : pouvoir tester facilement — au moins en partie — le code métier sur notre poste de développement.

```
[~]$ cc  my-hello.c  -Wall  -o my-hello
[~]$ ./my-hello
Hello from TR-B-01
[~]$
```

Parfait ! Maintenant **nous le *cross-compilons* pour la cible** :

```
[~]$ $CC  $CFLAGS  my-hello.c  -o  my-hello
[~]$ ./my-hello
-bash: ./my-hello : impossible d'exécuter le fichier binaire :
[~]$ file my-hello
my-hello: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (
[~]$
```

Après compilation, il est évidemment impossible d’exécuter le programme puisqu’il est prévu pour un processeur d’une architecture différente. C’est bien ce que nous indique la commande `file`.

Lorsque Qemu démarre, il crée une interface Ethernet pour la cible reliée à une interface «*Tap*» sur la machine hôte (c’est pour cela qu’il demande les droits *root* au démarrage). Les deux interfaces se trouvent dans le sous-réseau `192.168.7.0/24`, la machine hôte étant en `192.168.7.1` et la cible en `192.168.7.2`. Nous pouvons **utiliser la commande «scp» pour transférer l’exécutable** produit sur la cible. Comme le système de fichiers principal est en lecture-seule, j’envoie mon exécutable dans le répertoire «`/tmp`» sur lequel est monté un système de fichier virtuel `tmpfs` résidant en mémoire.

```
[~]$ scp  my-hello  root@192.168.7.2:/tmp/
The authenticity of host '192.168.7.2 (192.168.7.2)' can't be
RSA key fingerprint is SHA256:quorH2IdLg92wW2kLI9KwAC9FBo5m3a›
Are you sure you want to continue connecting (yes/no)? yes
```

```
Warning: Permanently added '192.168.7.2' (RSA) to the list of
root@192.168.7.2's password: (linux)
my-hello                                      100%   14KB  45.0
[~]$
```

Je peux alors me connecter sur la cible pour lancer mon exécutable :

```
Poky (Yocto Project Reference Distro) 3.0.1 mybox ttyAMA0
mybox login: root
Password: (linux)
root@mybox:~# /tmp/my-hello
Hello from mybox
root@mybox:~#
```

Compilation avec un Makefile

La compilation manuelle directe en appelant le compilateur n'est utilisable que dans des cas très simples, pour un projet avec un seul fichier source, et ne nécessitant pas d'options spécifiques de compilation. Dès que le nombre de fichiers sources impliqués augmente, **on fait appel à un `Makefile`**.

Voici un exemple simple de Makefile qui fait référence aux variables configurées par le script de la *toolchain*.

```
CC ?= gcc
CFLAGS += -Wall
EXE = my-hello
OBJS = my-hello.o

.PHONY: all

all: $(EXE)

$(EXE): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $<

my-hello.o: my-hello.c
    $(CC) $(CFLAGS) -c $^
```

```
.PHONY: clean

clean:
    rm -f *.o $(EXE)
```

On peut l'utiliser pour une compilation native (dont le code résultat sera exécuté sur la même machine que celle de compilation).

Pour pouvoir compiler le code sur PC, il ne faut pas que le script `environment-setup-armv7vet2hf-neon-poky-linux-gnueabi` ait été «sourcé» auparavant, sinon l'environnement contiendra les variables correspondant au *cross-compiler*. Il faut donc travailler dans un nouveau shell, par exemple en ouvrant un nouvel onglet ou un nouveau terminal.

```
[~]$ echo $CC
[~]$ make clean
rm -f *.o my-hello
[~]$ make
cc -Wall -c my-hello.c
cc  -o my-hello my-hello.o
[~]$ ./my-hello
Hello from TR-B-01
[~]$
```

Puis nous pouvons recommencer, après avoir sourcé le script, pour faire une *cross-compilation* dont le résultat devra être exécuté sur la cible.

```
[~]$ source  sdk/yocto-lab/environment-setup-armv7vet2hf-neon-
[~]$ make  clean
rm -f *.o my-hello
[~]$ make
arm-poky-linux-gnueabi-gcc  -march=armv7ve -mthumb -mfpu=neon
arm-poky-linux-gnueabi-gcc  -march=armv7ve -mthumb -mfpu=neon
[~]$ ./my-hello
-bash: ./my-hello : impossible d'exécuter le fichier binaire :
[~]$
```

Bien sûr le fichier exécutable ne peut être lancé sur la machine de compilation et comme précédemment il faudrait le copier sur la cible pour pouvoir le faire fonctionner.

Compilation avec les *Autotools*

Le mécanisme des *Autotools* permet d'assurer une bonne portabilité d'un code source sur différentes plateformes supportées par le projet Gnu. Ils sont utilisé dans de nombreux projets libres. Voici une version de notre *Hello World* utilisant ces outils :

```
[~]$ wget  https://www.logilin.fr/files/yocto-lab/hello-autotc
  [...]
hello-autotools-1.0.tar.bz2     100%[=========================
[~]$ tar  xf  hello-autotools-1.0.tar.bz2
[~]$ cd  hello-autotools/
[hello-autotools]$ ls
aclocal.m4      compile      configure     depcomp
autom4te.cache  config.h.in  configure.ac  hello-autotools.c
[hello-autotools]$
```

Réalisons une première compilation native pour le PC de développement. Pour cela, on appelle le script «`configure`» fourni dans le package. Comme précédemment, il ne faut pas que le script `environment-setup-`… ait été chargé par le shell. Il convient donc de basculer temporairement dans un nouveau shell.

```
[hello-autotools]$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking whether gcc understands -c and -o together... yes
checking for style of include used by make... GNU
```

```
checking dependency style of gcc... gcc3
checking how to run the C preprocessor... gcc -E
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking for stdlib.h... (cached) yes
checking sys/param.h usability... yes
checking sys/param.h presence... yes
checking for sys/param.h... yes
checking for unistd.h... (cached) yes
checking for gethostname... yes
checking that generated files are newer than configure... done
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
config.status: executing depfiles commands
[hello-autotools]$
```

Après toutes ses vérifications, le script «`configure`» a préparé un `Makefile` que nous pouvons utiliser pour produire l'exécutable.

```
[hello-autotools]$ make
make  all-am
make[1] : on entre dans le répertoire « /home/cpb/hello-autotc
gcc -DHAVE_CONFIG_H -I.     -g -O2 -MT hello-autotools.o -MD -
mv -f .deps/hello-autotools.Tpo .deps/hello-autotools.Po
gcc  -g -O2   -o hello-autotools hello-autotools.o
make[1] : on quitte le répertoire « /home/cpb/hello-autotools
[hello-autotools]$ ./hello-autotools
Hello from TR-B-01 (built with autotools)
[hello-autotools]$
```

Nous pouvons également appeler «`configure`» en lui indiquant avec son option «`--host`» le préfixe à ajouter devant les noms des outils de la chaîne de compilation. L'option «`--host`» n'est pas très judicieusement choisie, elle prête à confusion car dans l'univers des systèmes embarqués, on appelle généralement «hôte» le poste de développement (souvent un PC). Avec la commande `configure`, il faut comprendre «hôte» comme «machine qui hébergera l'exécutable final».

Il faut que `configure` puisse appeler les commandes de *cross-compilation*, aussi devons nous d'abord appeler le script d'initialisation de l'environnement.

```
[hello-autotools]$ make  clean
test -z "hello-autotools" || rm -f hello-autotools
rm -f *.o
[hello-autotools]$ source  ~/sdk/yocto-lab/environment-setup-a
```

Puis nous appelons «`configure`» Pour connaître le préfixe à transmettre, on peut utiliser le contenu de la variable `CROSS_COMPILE`.

```
[hello-autotools]$ ./configure  --host=$CROSS_COMPILE
configure: loading site script /home/cpb/sdk/yocto-lab/site-co
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for arm-poky-linux-gnueabi--strip... arm-poky-linux-g
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether make supports nested variables... yes
checking for arm-poky-linux-gnueabi--gcc... arm-poky-linux-gnu
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... yes
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether arm-poky-linux-gnueabi-gcc  -march=armv7ve -m
checking for arm-poky-linux-gnueabi-gcc  -march=armv7ve -mthum
checking whether arm-poky-linux-gnueabi-gcc  -march=armv7ve -m
checking for style of include used by make... GNU
checking dependency style of arm-poky-linux-gnueabi-gcc  -marc
```

```
checking how to run the C preprocessor... arm-poky-linux-gnuea
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for sys/stat.h... yes
checking for stdlib.h... yes
  [...]
```

Le reste de la compilation est identique à la précédente :

```
[hello-autotools]$ make
make  all-am
make[1] : on entre dans le répertoire « /home/cpb/hello-autotc
arm-poky-linux-gnueabi-gcc  -march=armv7ve -mthumb -mfpu=neon
mv -f .deps/hello-autotools.Tpo .deps/hello-autotools.Po
arm-poky-linux-gnueabi-gcc  -march=armv7ve -mthumb -mfpu=neon
make[1] : on quitte le répertoire « /home/cpb/hello-autotools
[hello-autotools]$
```

Compilation avec *CMake*

Il est possible également de *cross-compiler* automatiquement un projet utilisant **l'outil CMake**. Celui est également employé pour la compilation de nombreux projets libres. Voici un exemple de projet avec CMake :

```
[~]$ wget  https://www.logilin.fr/files/yocto-lab/hello-cmake-
[...]
hello-cmake-1.0.tar.bz2 100%[===================>]     642  --
[~]$ tar  xf  hello-cmake-1.0.tar.bz2
[~]$ cd  hello-cmake/
[hello-cmake]$ ls
CMakeLists.txt  hello-cmake.c
[hello-cmake]$
```

Il nous suffit de «sourcer» le script d'initialisation de l'environnement et d'appeler `cmake` puis `make` pour que la compilation soit automatiquement produite pour la cible embarquée.

```
[hello-cmake]$ source  ~/sdk/yocto-lab/environment-setup-armv7
[hello-cmake]$ cmake  .
-- Toolchain file defaulted to '/home/cpb/sdk/yocto-lab/sysroo
-- The C compiler identification is GNU 9.2.0
-- The CXX compiler identification is GNU 9.2.0
-- Check for working C compiler: /home/cpb/sdk/yocto-lab/sysro
-- Check for working C compiler: /home/cpb/sdk/yocto-lab/sysro
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /home/cpb/sdk/yocto-lab/sys
-- Check for working CXX compiler: /home/cpb/sdk/yocto-lab/sys
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/cpb/hello-cmake
[hello-cmake]$ make
Scanning dependencies of target hello-cmake
[ 50%] Building C object CMakeFiles/hello-cmake.dir/hello-cmak
[100%] Linking C executable hello-cmake
[100%] Built target hello-cmake
[hello-cmake]$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.tx
[hello-cmake]$ file  hello-cmake
hello-cmake: ELF 32-bit LSB shared object, ARM, EABI5 version
[hello-cmake]$
```

Conclusion

Nous avons vu dans cette séquence comment récupérer la *toolchain* de Yocto et compiler une application de manière indépendante de `bitbake`. Nous avons compilé du code en appelant directement le *cross-compiler*, en utilisant un `Makefile` écrit spécifiquement, en employant les *Autotools* et *CMake*.

Naturellement, dans un cadre de développement applicatif industriel on fait rarement appel directement au compilateur ou à `make`, on passe plutôt par l'intermédiaire d'un **Environnement de Développement Intégré** comme Eclipse, Code::Blocks,

Netbeans, Geany, Visual Studio, etc. Le principe est néanmoins identique à ce qu'on a vu ici, il suffira de configurer l'environnement de développement pour qu'il accède à la *toolchain* exportée par Yocto.

L’étape suivante dans cette progression va être d’incorporer directement notre code métier dans la production de l’image Yocto en écrivant notre propre recette.

«« **sommaire** »»