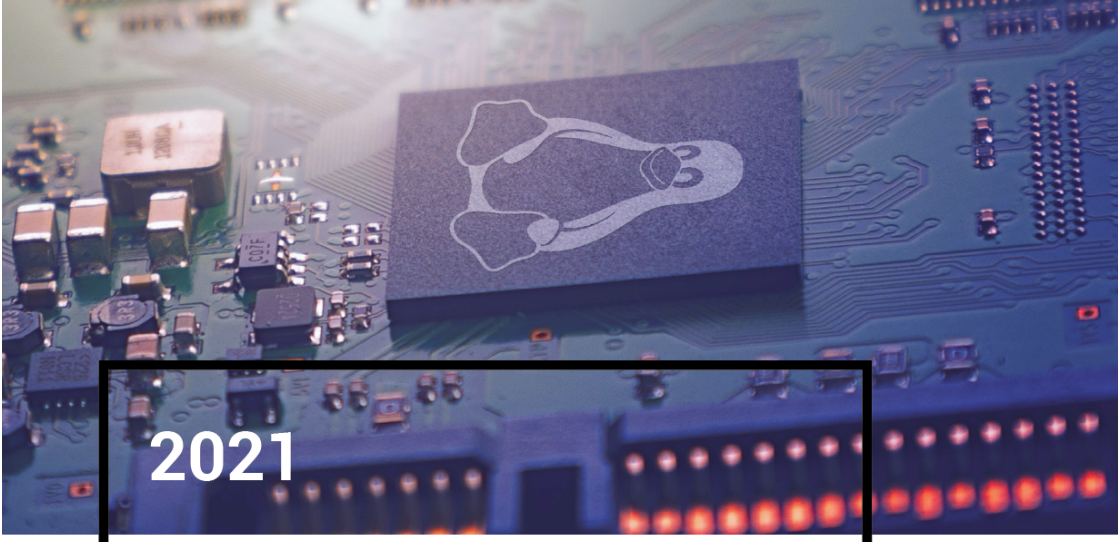


2021

Linux embarqué

Le meilleur des solutions
open source

SMILE
I.T IS OPEN

Pierre Ficheux
Directeur technique Smile ECS

Table des matières 4

A propos de Smile

Nous sommes plus de +1800 passionnés travaillant sur des solutions digitales créatives et innovantes, dans plus de 7 pays.
Notre approche est basée sur la compréhension des spécificités de votre marché, de votre business, de vos enjeux financiers et de vos défis présents et à venir.
Notre ADN open source est la garantie de notre proposition de valeur. Dans un souci constant de neutralité vis-à-vis de nos partenaires, nous proposons toujours la solution la plus adaptée à vos objectifs business et votre organisation.

Nous vous accompagnons dans les prises de décisions technologiques qui amélioreront votre business et la digitalisation de votre entreprise, depuis la phase de conception jusqu'au lancement et développement de la solution mise en place : e-commerce, applications métiers, data, objets connectés, solutions embarquées, infrastructure.

Nous sommes funs et innovants, mais aussi pragmatiques et orientés résultat.
Nous pensons que notre contribution est étroitement liée à votre succès car c'est ainsi que nous donnons du sens à notre travail.
Ouverts, nous favorisons le partage de connaissances avec nos livres blancs thématiques, et contribuons de façon importante à l'écriture de code dans de nombreux modules et cœur de solutions open source comme Drupal ou Magento.
Notre force repose sur l'association d'une vaste communauté d'experts aux parcours et horizons très différents.

Visionnaires, formés et certifiés, les Smiliens s'engagent pour faire grandir les projets que vous nous confiez, avec toute l'audace, la technicité et l'innovation qui nous caractérisent.

Nous sommes SMILE, leader européen du digital et de l'open source.

A propos de ce livre blanc

Le livre blanc consacré à l'utilisation de Linux dans les systèmes industriels et embarqués fut le premier de la série en 2016. Ce livre constitue une mise à jour importante et se focalise sur Yocto et non plus sur Buildroot.
Après un bref historique et quelques rappels sur les principes d'un système embarqué (matériel et logiciel) nous décrirons les éléments fondamentaux constituant une distribution Linux.
Nous continuerons par la définition d'un système « Linux embarqué » puis l'introduction à la notion de « build system » (outil de construction de distribution). La suite du livre sera consacrée à l'étude de l'outil Yocto, qui est désormais le plus répandu dans le monde industriel pour la mise en place de systèmes embarqués basés sur Linux.

L'auteur de ce livre (Pierre FICHEUX) est directeur technique et expert Linux pour la division ECS de Smile (Embedded and Connected Systems). Il est également l'auteur de nombreux articles et ouvrages sur le sujet traité dans ce livre.

Introduction aux systèmes embarqués

Par définition, un système embarqué est l'association de matériel (un calculateur) et de logiciel. Contrairement à l'informatique classique (poste de travail ou serveur), le système est dédié à un ensemble fini de fonctionnalités et il en est de même pour le logiciel.
Les premiers domaines d'application étaient limités à l'armement et au spatial pour lesquels les coûts des programmes étaient très élevés. La principale difficulté à l'époque était l'absence de processeur car le 4004 - premier microprocesseur disponible commercialement - fut créé par Intel (seulement) en 1971. Dans le domaine spatial, on s'accorde à dire que le premier système embarqué fut l'« Apollo Guidance Computer » [1] créé en 1967 par le MIT pour le programme spatial Apollo. Ce dernier disposait de 36 Kmots de ROM, 2 Kmots de RAM et fonctionnait à la fréquence de 2 MHz. De nos jours, on peut simuler le comportement de ce calculateur grâce à une page web animée par du code Javascript ! Dans le cas des applications militaires, on peut citer le calculateur D-17B, système de guidage pour le missile LGM-30 datant du début des années 60 [2].
Il n'était pas question à l'époque d'évoquer la notion de *système d'exploitation embarqué* et on parlait simplement de logiciel embarqué souvent écrit en langage machine. Par contre, le programme Apollo utilisait de nombreux calculateurs IBM/360 au sol et le logiciel le plus complexe de la mission occupait 6 Mo. Sur cet IBM/360, la NASA utilisait une version modifiée de l'OS/360 nommée RTOS/360 [3], afin de disposer d'un comportement temps réel.

REMARQUE
Même si on utilise désormais des langages évolués comme C ou Ada, certains systèmes embarqués sont toujours dépourvus d'OS. On parle alors de système « bare metal ».

A partir des années 80, l'évolution technologique permet d'intégrer de nombreuses fonctions dans un processeur. De ce fait, de nombreux systèmes d'exploitation temps réel (RTOS pour *Real Time Operating System*)

fonctionnent sur des processeurs du commerce, la plupart étant commercialisés par des éditeurs spécialisés. Nous pouvons citer VRTX (1981) édité par Mentor Graphics (désormais SIEMENS) et célèbre pour être utilisé dans le télescope Hubble, VxWorks (1987) édité par Wind River et LynxOS (1986) édité par Lynx Software. VxWorks est toujours très utilisé dans les industries sensibles comme le spatial. Il fut entre autre choisi pour la sonde spatiale Pathfinder (1996) ainsi que pour la mission Curiosity lancée en 2011 et toujours active. Dans le domaine du logiciel libre, le système d'exploitation RTEMS est également fréquemment choisi par la NASA ou l'agence spatiale européenne (ESA). Ce système diffusé sous licence GPL modifiée est utilisé pour le système de communication du rover Curiosity et de nombreuses autres missions. Il est également utilisé dans les satellites développés par Airbus Defense and Space.
A partir des années 2000, les systèmes embarqués ne se limitent plus aux applications industrielles ni donc aux RTOS ce qui permet de choisir un système comme Linux (qui par défaut n'a pas de capacités temps réel). Linux est désormais présent dans 95 % des boîtiers d'accès à Internet, décodeur TV, sans compter le fameux système Android qui intègre un noyau Linux. Plus récemment, des alliances d'acteurs de l'industrie automobile ont permis la mise en place de projets comme GENIVI ou AGL (basés sur Yocto) dont le but est de créer des environnements utilisables pour les applications IVI (pour *In Vehicle Infotainment*).

Linux comme système embarqué

Dans cette partie nous allons décrire les principales différences entre un système Linux classique et un système « Linux embarqué ».

From UNIX to Linux

Le système GNU/Linux fait partie de la famille UNIX qui est un standard industriel créé à la fin des années 60 (1969) par la société américaine AT&T, initialement pour ses besoins propres de recherche et développement. Il existait à l'époque un système d'exploitation multi-tâches et multi-utilisateurs

nommé MULTICS [5]. Ce dernier - créé en 1964 au MIT - était surtout adapté aux gros systèmes. La programmation de MULTICS utilisait un langage proche du langage C créé par IBM sous le nom de PL/1 (pour *Programming Language number 1*). L'équipe de recherche d'AT&T avait pour but de créer un nouveau langage (le langage C) ainsi qu'un système d'exploitation reprenant les concepts de MULTICS mais pouvant fonctionner sur des mini-ordinateurs de l'époque, soit des machines très peu puissantes. En hommage à son prédécesseur, le système fut nommé UNICS puis UNIX. En France, l'un des premiers calculateur pouvant fonctionner sous UNIX fut le SM-90 [6] (équipé d'un CPU Motorola 68000 et de 512 Ko de RAM). De conception française, la machine fut rapidement évincée par les stations de travail d'origine américaine (en premier lieu SUN puis IBM, HP, Silicon Graphics, etc.) utilisant leurs propres versions d'UNIX (SunOS puis Solaris, AIX, HP/UX, ou IRIX).
De par la faible puissance des machines à l'époque de sa conception, l'architecture UNIX est assez simple.

- Un noyau assurant les principaux services (mémoire, scheduling, etc.) et contenant des pilotes (drivers) pour la gestion des périphériques. Si l'architecture du processeur le permet (présence d'une MMU pour *Memory Management Unit*), ce noyau fonctionne en mode dit protégé ;
- Un ensemble d'utilitaires, bibliothèques et fichiers système fonctionnant dans l'espace dit utilisateur, soit le niveau de privilège le plus faible. L'utilitaire le plus célèbre est le « shell » permettant d'exécuter d'autres commandes en tapant leur nom dans un terminal. On parle souvent de Bourne Shell (du nom de son auteur Stephen Bourne). Lorsqu'une commande est exécutée, elle est chargée en RAM et devient un processus. Cet ensemble est généralement nommé « root-filesystem » (ou plus simplement « root-fs »). UNIX a de plus introduit la notion de fichier généralisé car – presque – tout sous UNIX est vu comme un fichier (répertoire, périphérique, etc.).

REMARQUE
Le proverbe UNIX le plus célèbre est certainement :
« Sous UNIX tout est fichier sauf les processus ».

Cette architecture créée au début des années 70 n'a pas fondamentalement évolué tant les principes adoptés sont à la fois intelligents, simples et efficaces. D'autres systèmes d'exploitation utilisent ces concepts, ou du moins s'en approchent et on parle alors de système « UNIX like » (citons OS/9, Xenix voire VxWorks ou même Android).

Même si le noyau Linux fut créé au début des années 90 (soit 20 ans plus tard), l'architecture choisie est quasiment identique ce qui valut à Linus Torvalds les reproches d'Andrew Tannenbaum, grand spécialiste du système UNIX et auteur de MINIX, une version d'UNIX utilisée par Linus Torvalds pour construire son premier système. Selon le vénérable Professeur Tannenbaum, Linus aurait manqué d'audace en créant un nième système UNIX monolithique (fonctionnant avec un seul noyau fournissant l'ensemble des services).

« To me, writing a monolithic system in 1991 is a truly poor idea. »

Distribution Linux

La plupart des utilisateurs s'orientent vers l'utilisation d'une distribution qui correspond à un ensemble de composants logiciels que l'on peut installer sur un calculateur grâce à une procédure la plus simple possible. Les plus célèbres distributions sont Debian, Ubuntu, Fedora ou bien RHEL (dans le monde professionnel). Au fur et à mesure de l'histoire de Linux (projet démarré, rappelons le en 1991), la procédure d'installation s'est considérablement simplifiée passant de la compilation manuelle de dizaines de composants (le noyau Linux et les divers utilitaires du projet GNU qui constituent la partie visible du système) jusqu'à une procédure quasiment automatique qui n'a rien à envier aux systèmes d'exploitation commerciaux tels que Windows ou MacOS X.

Cette simplicité idyllique concerne cependant les serveurs et PC de bureau basés sur l'architecture x86. Dans ce cas, l'espace disque est quasiment illimité (difficile de nos jours de trouver un disque dur de capacité inférieure à 500 Go) par rapport à l'espace occupé par la distribution (quelques Go voire plus). De ce fait, la procédure prend l'initiative d'installer de nombreux composants parfois inutiles qui eux-même dépendent d'autres composants. Si on prend l'exemple d'une distribution Ubuntu récente, on note assez rapidement qu'un système minimal est composé au bas mot de près de 2000 composants (ou « paquets »).

Le cas d'un système embarqué est assez différent. Même si la puissance CPU et la capacité mémoire ont fortement augmenté ces derniers temps, on ne peut comparer un smartphone récent (parfois plus puissant qu'un PC/x86 vieux de quelques années) avec un système embarqué de puissance moyenne qui sera soumis à de nombreuses contraintes (coût, consommation, sécurité, longévité, etc.). En plus de cela, si on évoque le désormais fameux « internet des objets » (qui utilise de nombreux systèmes embarqués), on ne peut bien entendu pas envisager que le moindre capteur autonome un peu évolué soit rechargé tous les jours ou presque (comme c'est le cas pour un smartphone). Ce cas est certes un peu extrême car il n'est pas forcément judicieux d'utiliser un système Linux dans tous les cas de figure. Cependant nous verrons que l'utilisation d'un tel système a parfois quelques avantages.

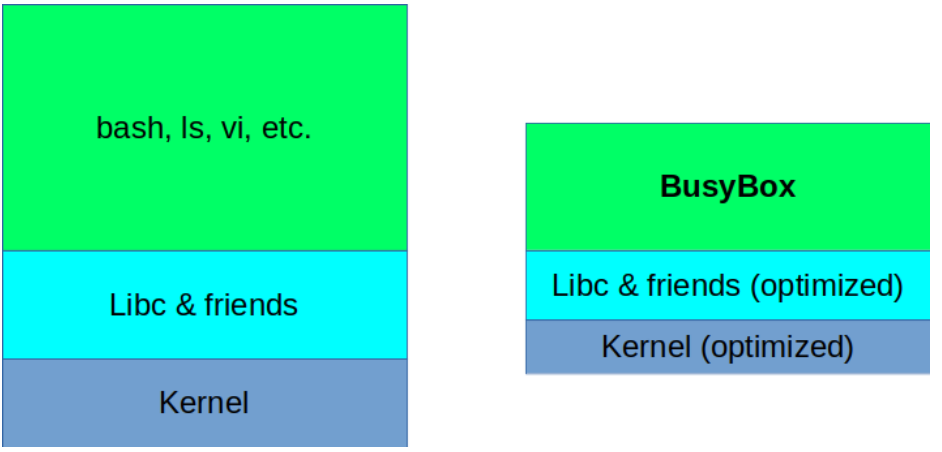
En effet, qui aurait cru il y a plus de 20 ans – à l'époque du Nokia 3210 - que la quasi-totalité des smartphones seraient finalement basés sur un noyau Linux (cas d'Android) ou d'une architecture proche d'UNIX (cas des systèmes Android et iOS).

Architecture « Linux embarqué »

Comme dans le cas d'une distribution Linux classique, une distribution "Linux embarqué" est composée de deux éléments principaux :

1. Le noyau Linux
2. Le root-filesystem

La principale différence concerne l'optimisation de ces deux composants comme le montre le schéma suivant. Un système Linux est composé du noyau (kernel space) et du root-filesystem (user space), ce dernier contenant les commandes Linux, les fichiers système et les bibliothèques (Glibc, etc.).



GNU/Linux vs Embedded Linux

On peut optimiser le noyau Linux en modifiant sa configuration afin de limiter les fonctionnalités et donc le nombre de pilotes de périphériques activés (dans la partie statique du noyau ou bien dans la liste des pilotes dynamiques). Si on considère l'espace utilisé (empreinte mémoire), l'optimisation de la partie user space est largement plus significative. A titre

d'exemple, la commande bash présente dans tous les systèmes Linux occupe le même espace (environ 1 Mo) que l'exécutable busybox qui remplace environ 300 commandes Linux (bien entendu dans des versions allégées en fonctionnalités). Nous décrirons plus en détail BusyBox un peu plus loin dans ce livre.
Le compilateur est bien entendu un élément fondamental pour la création du système. En effet, la cible choisie utilise le plus souvent une architecture différente de celle du poste de développement ce qui nécessite l'installation d'un compilateur dit « croisé » (exemple, x86 vers ARM). Certaines distributions comme Ubuntu intègrent directement les paquets nécessaires. Cependant il est également possible (et recommandé) d'obtenir des chaînes de compilation auprès de Linaro, ARM ou bien SIEMENS (Sourcery CodeBench). De nos jours, les outils de construction ou « build system » (Yocto, Buildroot) que nous verrons plus tard peuvent également produire un compilateur croisé à partir des sources.

L'outil BusyBox

Le projet BusyBox a démarré en 1995 sous l'impulsion de Bruce Perens, membre du projet Debian. Le but était de produire une disquette de réparation pour la distribution Debian et rappelons que l'espace disponible sur un tel support n'était que de 1,44 Mo. BusyBox rassemble donc les outils courants d'une distribution Linux (shell, éditeurs, commandes de manipulation de fichiers, etc.) dans un exécutable unique nommé busybox. Bien entendu les fonctionnalités de chaque commande sont largement inférieures à celles des versions originales du projet GNU mais comme nous l'avons dit ce point n'est pas un problème pour un système embarqué. Le principe de BusyBox étant d'intégrer toutes les commandes dans un seul exécutable, on place un lien symbolique correspondant au nom de la commande (sh, ls, vi, etc.) ce qui permet d'exécuter la bonne portion de code (et d'optimiser l'empreinte mémoire utilisée).

```
$ ls -l bin/sh bin/busybox
-rwxr-xr-x 1 pierre pierre 968032 oct.  13 16:03
bin/busybox
```

```
lrwxrwxrwx 1 pierre pierre      7 oct.  13 16:03 bin/sh ->
busybox
```

On note la taille très raisonnable de l'exécutable busybox légèrement inférieure à la seule taille de la commande bash d'une distribution Linux classique (environ 1 Mo).

```
$ ls -l /bin/bash
-rwxr-xr-x 1 root root 1113504 juin   7  2019 /bin/bash
```

Pour que le système soit exploitable, il faut également installer les bibliothèques partagées nécessaires à l'exécution de BusyBox.

```
$ LANG=C aarch64-linux-gnu-readelf -a bin/busybox | grep
"Shared library:"
 0x0000000000000001 (NEEDED)             Shared library:
[libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library:
[libresolv.so.2]
 0x0000000000000001 (NEEDED)             Shared library:
[libc.so.6]
```

Notons cependant qu'il est possible de compiler BusyBox de manière statique afin qu'il intègre le code des bibliothèques nécessaires. Cette option est cependant assez éloignée du cas réel car en général BusyBox n'est pas la seule application d'un système embarqué. Il est également possible d'ajouter ou retirer des commandes afin d'ajuster la taille de l'exécutable aux besoins du projet en passant par une phase de configuration via la commande `make menuconfig`.

Mise en place des bibliothèques système

Les bibliothèques système sont fournies par la chaîne de compilation croisée. Même si la Glibc (GNU C Library) est la plus répandue de nos jours, il est également possible d'utiliser une autre bibliothèque *libc* plus légères comme uClibc ou Musl. Le projet Android AOSP utilise sa propre bibliothèque libc

Bionic (spécifique à Google). Dans le cas de Yocto ou Buildroot, on produit en général la chaîne de compilation à partir des sources (donc en précisant le type de libc). Ces outils permettent également d'utiliser une chaîne binaire et dans ce cas, le type de libc est bien entendu imposé par la chaîne.

Créer une image « Linux embarqué »

Lors de la précédente version de ce livre, nous avions décrit précisément comment construire une image Linux à partir des sources du noyau et de BusyBox, l'ensemble étant intégré dans un squelette de root-filesystem très simple et testé sur une Raspberry Pi 2. Nous considérons que l'exercice n'est plus nécessaire et nous allons nous limiter à la description des différentes solutions.

Les différentes solutions envisageables

Nous avons déjà évoqué la construction manuelle d'une distribution. Cette solution reste un exercice de style (ou un cas d'école) mais n'est désormais plus utilisable dans un cadre industriel. En effet, la maintenance d'une telle configuration représente une lourde charge très difficile à valoriser car il faut dédier une équipe (ou au moins une personne) à celà.

Actuellement, les solutions envisageables sont donc les suivantes :

1. Adapter une distribution Linux classique
2. Utiliser un « build system » (Yocto, Buildroot, etc.)
3. Utiliser ELBE ou DEBOS (Debian)

La première solution est uniquement utilisable dans le cadre d'un POC (preuve de concept) car ce type de distribution n'est pas adapté à une cible. Par contre, la mise en place est simple car on travaille directement sur la cible avec un système de paquets de type APT. De plus, les distributions classiques sont le plus souvent maintenues en majorité pour l'architecture

x86 (même s'il y a des exceptions comme la Raspbian, désormais Raspberry Pi OS).
La solution du « build system » est la principale évoquée dans ce livre mais elle n'a bien entendu pas que des avantages (bien qu'étant la plus répandue). En effet, des solutions basées sur l'écosystème Debian (soit ELBE [7] et DEBOS [8]) sont également possibles et ont l'avantage de s'appuyer sur un projet célèbre pour la qualité de sa production.

Notion de « build system »

Dans la plupart des cas, une carte industrielle est fournie avec un BSP (pour *Board Support Package*), ce dernier étant lié à « build system » .
Le « build system » - que l'on peut à peu près traduire en français par « système de construction » - permet de créer une distribution Linux à partir des sources des composants. Dans ce cas, on parle plus d'*image* que de distribution car le résultat - le fichier image binaire - est écrit sur une mémoire flash. Toutes les distributions classiques déjà citées (Debian, Ubuntu, etc.) utilisent leur propre système de construction mais ce dernier est rarement visible car l'utilisateur installe la distribution à partir des paquets binaires. D'autres outils de type « build system » sont également utilisés par des projets dont le plus connu est AOSP (Android Open Source Project).
Dans ce livre, nous allons mettre en avant l'outil plus que la distribution pour la bonne raison que l'on n'a pas réellement affaire à une distribution, comme l'indique la devise de Yocto.

« It's not an embedded Linux distribution – it creates a custom one for you. »

Un build system offre les fonctionnalités suivantes :

- Un ensemble de règles de production (une par composant), une règle est appelée « recette » (ou *recipe* en anglais). Les recettes peuvent définir des dépendances entre les composants (pour construire A il faut B).

- Un « moteur de construction » permettant de réaliser les différentes étapes de chaque recette (télécharger, extraire, configurer, compiler, installer, etc.). On parle souvent du « cuisinier qui utilise des recettes » et cette analogie est très réaliste. Le moteur se charge également d'intégrer les différents composants binaires dans l'image finale à installer sur la cible.

REMARQUE
Un outil de build system n'est pas destiné au développement mais à l'intégration des composants (standards ou ajoutés). Il permet à l'utilisateur de se focaliser sur la valeur ajoutée de son produit (i.e. ses applications), la gestion de composants génériques étant prise en compte par l'outil. Par contre, il permet de créer les composants nécessaires à un environnement de développement croisé (dont le compilateur croisé inclus dans le SDK).

Dans la suite de la présentation nous allons décrire le principal outil de build system utilisé dans l'industrie, soit Yocto. Il existe cependant d'autres outils comme Buildroot, OpenWrt ou PTXdist. Buildroot étant également très connu, nous l'évoquerons brièvement à la fin de ce livre.

Les principaux outils disponibles

Comme nous l'avons dit précédemment, ces outils sont fréquemment utilisés et existent depuis longtemps. Yocto est certainement l'outil le plus puissant (mais également le plus complexe). Il est soutenu par la Fondation Linux, Intel et de nombreux acteurs importants du monde de l'embarqué. Le projet date de 2010 mais il s'appuie sur deux composants plus anciens, soit l'architecture de recettes OpenEmbedded (2003) et le moteur BitBake (2004), les deux étant utilisés pour le projet OpenZaurus.
Buildroot fut initialement créé par les développeurs de uClibc (Micro-C-libC) afin de créer des images de test (« build root-fs » d'où le nom de l'outil). Il utilise GNU-Make (commande make) comme moteur et donc des fichiers à ce format pour définir les recettes. Il est d'un abord beaucoup plus simple que Yocto mais sa communauté est beaucoup plus réduite et il ne bénéficie

donc pas comme Yocto d'un financement de la part de la fondation Linux.
OpenWrt est dérivé de Buildroot. Il fut initialement développé pour le routeur Linksys WRT54GL (d'où le nom) mais fonctionne désormais sur près de 1800 modèles de routeurs. Contrairement à Buildroot, il inclut la notion de paquet binaire. OpenWrt a été officiellement utilisé pour la box de SFR ainsi que le routeur Fonera de la communauté FON.
D'autres outils similaires sont également diffusés sous licence libre (PTXdist, LTIB, etc.) mais leur utilisation est plus confidentielle. Les principes utilisés sont cependant les mêmes.

Utilisation de Yocto

Yocto est basé sur deux composants, soit OpenEmbedded (OE) et BitBake. Ils furent créés suite à des limitations constatées sur les fonctionnalités de Buildroot dans le cadre du projet OpenZaurus (mais c'était vers 2003 !). En 2010, EO et BitBake – ainsi que d'autres projets affiliés – furent rassemblés dans un projet « chapeau » nommé Yocto. A ce jour, Yocto est l'outil le plus fréquemment utilisé pour la construction d'un BSP. Les sources sont disponibles auprès du projet Yocto (et/ou de GitHub) [9] ou bien sur le dépôt du fabricant (cas de Toradex [10]).

REMARQUE
yocto (symbole y) est le préfixe du système international d'unités qui représente 10-24 fois cette unité (soit un quadrillionième). Adopté en 1991, il vient du grec ὀκτώ, (huit) car égal à $(10^{-3})^8$.

Les principes de fonctionnement de Yocto sont hérités de ceux d'OpenEmbedded :

- Yocto permet de créer des paquets binaires comme dans une distribution classique. Les formats pris en compte sont RPM, DEB et IPK.
- Yocto utilise la notion de couche (ou *layer*). Un layer peut ajouter le

support d'un composant matériel donné, mais également des composants logiciels. Les couches sont représentées par des « métadonnées » rassemblées dans des répertoires (meta-raspberrypi, meta-qt, meta-ivi, etc.).

- Du fait du principe précédent, Yocto fournit un environnement initial très léger (basé sur OE) qui permet de produire une distribution pour quelques cibles de référence (souvent émulées par QEMU). Le support de matériel réel est disponible à l'extérieur du projet sous forme de layers fournissant le BSP.
- Grâce à BitBake, Yocto peut paralléliser les actions. Il peut télécharger les sources d'un composant A tout en compilant le composant B et créer le paquet pour le composant C.
- Yocto permet d'utiliser plusieurs environnements de compilation (pour plusieurs cibles) dans le même répertoire de travail.
- Yocto ne dispose pas (réellement) d'un outil de configuration graphique similaire à make menuconfig et la configuration s'effectue dans des fichiers. L'outil *Toaster* fourni avec Yocto reste très peu utilisé car très limité.

De nombreux constructeurs de matériel (Intel, NXP, TI, Broadcom, AMD, etc.) sont membres de la communauté Yocto. De ce fait, la connaissance de cet outil est souvent nécessaire pour la production d'un système car le BSP est fourni sous la forme de métadonnées Yocto. Le moteur de Yocto (BitBake) est écrit en Python ainsi que la quasi-totalité du système mais la connaissance du langage Python n'est pas nécessaire pour une utilisation normale de Yocto.

Construction d'une image Yocto

A titre d'exemple, nous allons produire une image pour deux cibles :

1. L'émulateur QEMU (cible standard de Yocto) en version ARM
2. La célèbre carte Raspberry Pi (modèle 3)

REMARQUE
Mise à part la production d'une image simple pour la Raspberry Pi 3, la majorité des manipulations seront réalisées sur QEMU afin que le lecteur puisse les reproduire plus facilement sans disposer de matériel. Les fonctionnalités démontrées dans le livre sont indépendantes de la cible matérielle utilisée.

Nous choisissons la version 3.1 de Yocto nommée « Dunfell » car c'est une version dite LTS (pour *Long Term Support*) utilisée par de nombreux fabricants. Les principes exposés restent valables pour les autres versions, la dernière version étant la 3.3 au moment de la rédaction du livre. En premier lieu, on doit tout d'abord obtenir les sources par la commande suivante qui crée le répertoire `poky` (nom de la distribution d'exemple de Yocto).

```
$ git clone -b dunfell git://git.yoctoproject.org/poky.git
```

On peut alors se positionner dans le répertoire `poky` et créer un répertoire de travail que nous nommerons `qemuarm-build`. La commande nous place directement dans le répertoire et nous pouvons alors créer l'image minimale en utilisant la commande `bitbake`. Si on ne précise pas le répertoire, `bitbake` crée par défaut le répertoire `build`.

```
$ cd poky
$ source oe-init-build-env qemuarm-build
...
### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-ide-support
```

```
You can also run generated qemu images with a command like
'runqemu qemuarm'

Other commonly useful commands are:
 - 'devtool' and 'recipetool' handle common recipe tasks
 - 'bitbake-layers' handles common layer tasks
 - 'oe-pkgdata-util' handles common target package tasks
```

REMARQUE
La commande UNIX `source` exécute un script dans l'environnement du shell courant, et non du shell fils, ce qui permet d'affecter des variables d'environnement dans le shell courant (et donc de les conserver).

Le répertoire `conf` contient deux fichiers importants soit `local.conf` et `bblayers.conf` qui sont produits automatiquement lors de la création du répertoire de travail. Le premier fichier correspond à la configuration locale de l'environnement. Il permet de redéfinir des variables de BitBake dont les valeurs par défaut sont définies dans `meta/conf/bitbake.conf`. Yocto est fortement basé sur la notion d'héritage, et l'ajout de couches redéfinit fréquemment ces variables. Le fichier `local.conf` est le dernier niveau de modification possible. La cible par défaut correspond à l'émulateur QEMU pour l'architecture x86_64, il est donc nécessaire de modifier le fichier `local.conf` afin d'ajouter la ligne suivante, par exemple à la fin du fichier.

```
MACHINE = "qemuarm"
```

L'exécution de la commande `bitbake` conduit à un certain nombre d'actions, en premier lieu le calcul du nombre de tâches à exécuter (plus de 3000 !) afin de produire l'image. Par défaut, Yocto construit la chaîne de compilation et il est assez difficile de passer outre (même s'il est possible d'importer une chaîne binaire). Après plusieurs dizaines de minutes d'attente (voire plusieurs heures), on doit obtenir une image exécutable dans le répertoire `tmp/deploy/images/qemuarm`. Notons que `bitbake` utilise le sous-répertoire tmp pour stocker les fichiers produits lors de la construction des paquets et de l'image. On note également que suivant les dépendances

entre les tâches à réaliser, `bitbake` exécute en parallèle plusieurs actions, correspondant au maximum au nombre de cœurs présents dans la machine de compilation.

```
$ bitbake core-image-minimal
Parsing recipes: 100%
|##################################################################
##################################################################
#####################################| Time: 0:00:15
Parsing of 774 .bb files complete (0 cached, 774 parsed). 1326
targets, 61 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.46.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-18.04"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "qemuarm"
DISTRO               = "poky"
DISTRO_VERSION       = "3.1.6"
TUNE_FEATURES        = "arm armv7ve vfp thumb neon
callconvention-hard"
TARGET_FPU           = "hard"
meta
meta-poky
meta-yocto-bsp       =
"dunfell:1cb03844e124b88b98ddfdd4fce47fb545c6fe5e"

Initialising tasks: 100%
|##################################################################
##################################################################
#############################| Time:0:00:01
Sstate summary: Wanted 1144 Found 0 Missed 1144 Current 0 (0%
match, 0% complete)
NOTE: Executing Tasks
Currently 10 running tasks (334 of 3134)  10% |################
|
0: m4-native-1.4.18-r0 do_configure - 4s (pid 4263)
1: elfutils-native-0.178-r0 do_patch - 3s (pid 10032)
2: rpm-native-1_4.14.2.1-r0 do_patch - 3s (pid 11393)
```

```
3: openssl-native-1.1.1j-r0 do_compile - 1s (pid 17316)
4: libx11-1_1.6.9-r0 do_unpack - 0s (pid 18070)
5: base-files-3.0.14-r89 do_unpack - 0s (pid 18573)
6: expat-2.2.9-r0 do_unpack - 0s (pid 18624)
7: unzip-native-1_6.0-r5 do_compile - 0s (pid 18634)
8: xorgproto-native-2019.2-r0 do_patch - 0s (pid 18698)
9: shadow-native-4.8.1-r0 do_unpack - 0s (pid 18791)
...
```

Les sources des différents composants sont téléchargées dans le répertoire `downloads`. Comme indiqué lors de la création de l'environnement et après une longue attente, on peut finalement tester la distribution en utilisant la commande `runqemu nographic` qui exécute l'image en mode texte.

```
$ runqemu nographic
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 5.4.103-yocto-standard
(oe-user@oe-host) (gcc version 9.3.0 (GCC)) #1 SMP PREEMPT
Tue Mar 9 04:13:07 UTC 2021
[    0.000000] CPU: ARMv7 Processor [412fc0f1] revision 1
(ARMv7), cr=30c5387d
[    0.000000] CPU: div instructions available: patching
division code
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache,
PIPT instruction cache
[    3.815173] async_tx: api initialized (async)
[    3.815501] Block layer SCSI generic (bsg) driver
version 0.4 loaded (major 252)
[    3.815696] io scheduler mq-deadline registered
[    3.815820] io scheduler kyber registered
[    3.817103] pci-host-generic 3f000000.pcie: host bridge
/pcie@10000000 ranges:
...
[    5.043936] udevd[128]: starting version 3.2.9
[    5.105136] udevd[129]: starting eudev-3.2.9
[    6.752997] EXT4-fs (vda): re-mounted. Opts: (null)
[    6.753416] ext4 filesystem being remounted at /
supports timestamps until 2038 (0x7fffffff)
```

```
Poky (Yocto Project Reference Distro) 3.1.6 qemuarm
/dev/ttyAMA0

qemuarm login: root
root@qemuarm:~# uname -a
Linux qemuarm 5.4.103-yocto-standard #1 SMP PREEMPT Tue Mar
9 04:13:07 UTC 2021 armv7l GNU/Linux
```

Test de l'image Yocto pour la cible qemuarm

Le test sur une Raspberry Pi 3 (ou toute autre plate-forme réelle) nécessite l'ajout du BSP qui dans notre cas correspond au répertoire `meta-raspberrypi`. On note qu'il est nécessaire d'utiliser la branche correspondant à la version de Yocto.

```
$ cd poky
$ git clone -b dunfell
git://git.yoctoproject.org/meta-raspberrypi
$ source oe-init-build-env rpi3-build
```

Comme dans le test précédent, on affecte la variable MACHINE dans le fichier `local.conf`. Dans notre cas, la valeur à utiliser est `raspberrypi3` qui correspond à la cible Raspberry Pi 3 en mode 32 bits.

```
MACHINE = "raspberrypi3"
```

Cette valeur a du sens uniquement si on ajoute à l'environnement la couche (layer) concernant les cibles Raspberry Pi. La couche est déjà présente dans le répertoire des sources, mais il faut indiquer à Yocto de la prendre en compte en utilisant la commande suivante.

```
$ bitbake-layers add-layer ../meta-raspberrypi
```

Cette commande a pour effet de modifier le fichier `bblayers.conf`

comme suit (ajout d'une ligne).

```
BBLAYERS ?= " \
<path>/poky/meta \
<path>/poky/meta-yocto \
<path>/poky/meta-yocto-bsp \
<path>/poky/meta-raspberrypi \
"
```

REMARQUE :
Il est important de préciser que le layer peut être placé ailleurs que dans le répertoire des sources de Yocto.

Même si la cible est très différente de celle du test précédent, la plupart des sources sont communes aux deux architectures. On peut donc utiliser un lien symbolique afin d'éviter de télécharger de nouveau l'intégralité des sources.

```
$ ln -s ../qemuarm-build/downloads .
```

Une autre option permet de limiter de manière drastique l'espace utilisé pour la compilation. Pour cela il suffit d'ajouter l'option suivante à la fin du fichier `local.conf`.

```
INHERIT += "rm_work"
```

Une fois cette option activée, la plupart des fichiers temporaires créés dans `tmp/work` seront supprimés au fur et à mesure de la compilation.
Cette option permet donc d'économiser plusieurs dizaines de Go d'espace disque. A partir de là on peut produire l'image en utilisant de nouveau la commande `bitbake core-image-minimal`.
A la fin de la compilation, une image est produite dans `tmp/deploy/images/raspberrypi3`.

On pourra la copier sur la micro-SD grâce à la commande `dd`.

```
$ umount /dev/mmcblk0p*
$ sudo dd
if=tmp/deploy/images/raspberrypi3/core-image-minimal-
raspberrypi3.wic of=/dev/mmcblk0 status=progress
```

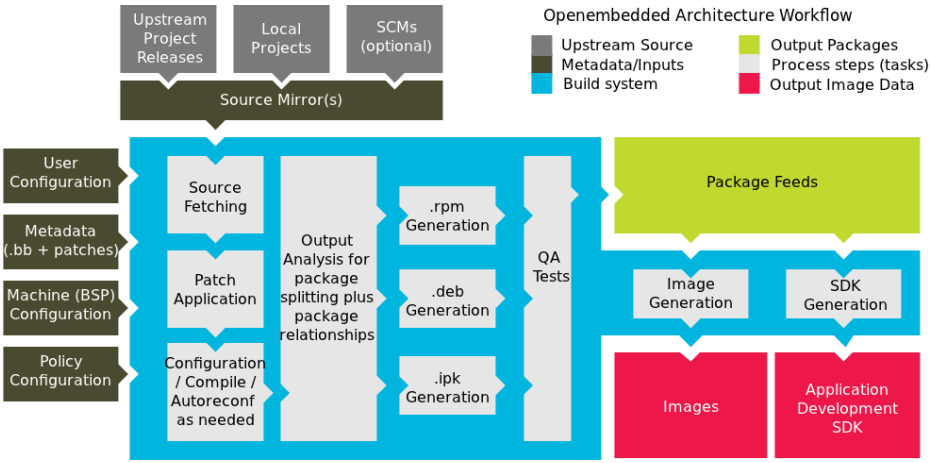
Le suffixe `.wic` correspond à un format désormais standard dans Yocto destiné à la manipulation des images binaires. La commande `wic` permettant de réaliser ces opérations (liste des images, affichage du contenu, etc.). On peut alors tester le démarrage de la Raspberry Pi 3 et on obtient la trace suivante. Notons que cette version minimale n'inclut pas de système de gestion des paquets sur la cible. Nous démontrerons ce point dans la suite du livre en utilisant de nouveau la cible QEMU. Notons également que la version de noyau est différente car le noyau de la Raspberry Pi 3 provient du BSP (et non de Yocto).

```
...
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 5.4.72-v7 (oe-user@oe-host) (gcc
version 9.3.0 (GCC
)) #1 SMP Mon Oct 19 11:12:20 UTC 2020
[    0.000000] CPU: ARMv7 Processor [410fd034] revision 4
(ARMv7), cr=10c5383d
...
Poky (Yocto Project Reference Distro) 3.1.6 raspberrypi3
/dev/ttyS0

raspberrypi3 login: root
root@raspberrypi3:~# uname -a
Linux raspberrypi3 5.4.72-v7 #1 SMP Mon Oct 19 11:12:20 UTC 2020
armv7l GNU/Linux
root@raspberrypi3:~#
```

Fonctionnement et configuration

Après ce test rapide, nous allons détailler un peu plus le fonctionnement de Yocto (en fait OpenEmbedded et BitBake). La figure suivante décrit le mécanisme global.



Fonctionnement de Yocto

La partie en haut à gauche correspond aux composants utilisés (externes ou internes). Les paramètres de l'environnement de construction (présents dans `local.conf` et `bblayers.conf`) sont représentés sur la gauche. La partie « moteur » est représentée en bleu clair et on remarque que les paquets binaires RPM, DEB ou IPK sont systématiquement produits. Le résultat (en rouge) correspond aux images du système ainsi qu'au SDK, soit l'environnement de développement produit (dont le compilateur croisé) utilisable pour le développement d'applications.

Notion de recette (.bb)

Outre la possibilité de produire des paquets binaires, une différence notable avec Buildroot est la notion de recette s'appliquant à n'importe quel composant, du simple « Hello World » à l'image complète, en passant par BusyBox ou le noyau Linux. Une recette correspond à un fichier `.bb` utilisable par BitBake par la commande suivante.

```
$ bitbake ma-recette
```

Pour que la recette soit utilisable, il faut bien entendu que le répertoire de métadonnées qui la contient soit déclaré dans le fichier `bblayers.conf`.

Notion de classe (.bbclass)

Les recettes partagent des procédures communes, par exemple « comment compiler un composant basé sur Autotools ». De ce fait, Yocto fournit un certain nombre de classes (fichiers `.class`) dans le répertoire `meta/classes`. On peut bien entendu créer ses propres classes dans des répertoires de métadonnées ajoutés. L'utilisation d'une classe implique la présence de la commande `inherit` dans le fichier `.bb`.

```
inherit autotools
```

On peut également hériter « globalement » d'une classe (i.e. pour toutes les recettes) en utilisant la variable `INHERIT` dans le fichier `local.conf`.

```
INHERIT += "rm_work"
```

Fichier à inclure (.inc)

Le dernier type correspond à un fichier pouvant être inclus dans n'importe quelle recette ou classe. Il correspond en général à des définitions de constantes. On utilise pour cela les directives `include` ou `require`. Dans le deuxième cas, la présence du fichier à inclure est obligatoire.

```
require linux.inc
include ../common/firmware.inc
include conf/distro/include/security_flags.conf
```

Utilisation de BitBake

Pour l'instant nous avons utilisé la commande `bitbake` sans aucun paramètre mis à part le nom de la recette à produire. La production d'un paquet binaire correspond à l'exécution de plusieurs étapes (fetch, unpack, patch, licence, configure, compile, install, package, deploy, etc.). Ces étapes correspondent à des fonctions `do_fetch()`, `do_unpack()`, etc.

Il est parfois utile d'exécuter seulement une étape en utilisant l'option `-c`, par exemple :

```
$ bitbake -c fetch ma-recette
```

Cette commande indique de télécharger les sources nécessaires à la construction de la recette. L'exemple suivant permet de reconstruire un paquet.

```
$ bitbake -c cleansstate ma-recette
$ bitbake ma-recette
```

Le but « cleansstate » efface la recette et l'entrée dans le cache des recettes (le dossier `cleansstate`). On peut également utiliser « clean » (qui n'efface pas le cache) ou « cleanall » (qui efface les sources téléchargées en plus du cache).

Prise en charge des paquets

L'image actuelle *core-image-mininal* testée sur les deux cibles est la plus simple possible. D'autres images comme *core-image-base* sont plus complètes mais nous allons plutôt ajouter des éléments à la distribution minimale afin de l'enrichir. Yocto permet d'ajouter des paquets soit à la construction de l'image, soit après la construction. Dans le deuxième cas, l'image doit bien entendu disposer d'un gestionnaire de paquets (ce qui n'est actuellement pas le cas).
Outre les paquets, Yocto permet d'ajouter des « features », qui correspondent à un ensemble de paquets constituant une fonctionnalité précise (x11, tools-debug, package-management, nfs-server, etc.).

REMARQUE
Une méthode utilisable sous Yocto consiste à réaliser des tests en modifiant `local.conf` puis à définir une nouvelle recette d'image dérivée d'une recette fournie par Yocto (voir plus loin dans le document).

Le format de paquet utilisé par défaut dans Yocto est RPM (créé par Red Hat). Mais on peut également utiliser DEB ou IPK. Ce dernier format fut créé

spécifiquement pour les systèmes embarqués à l'époque du projet OpenEmbedded et il consomme de ce fait beaucoup moins d'espace (quatre fois moins que le format RPM). Les lignes suivantes ajoutées à `local.conf` permettent d'ajouter à la cible le support des paquets IPK.

```
PACKAGE_CLASSES = "package_ipk"
EXTRA_IMAGE_FEATURES += "package-management"
```

L'utilisation de la gestion de paquets est un gros avantage dans la phase de développement car cela permet de manipuler proprement des paquets sur la cible (ajout, recherche, liste, retrait, etc.). Par contre, la quasi-totalité des produits industriels n'utilisent pas de gestionnaire de paquets et les mises à jour sont faites grâce à des outils dédiés (Mender, RAUC, SWUpdate) pour lesquels il existe des layers Yocto. Notons que le sujet de la mise à jour a fait l'objet d'un livre blanc Smile paru récemment [11].
Suite à la construction de la distribution par BitBake, on remarque que la taille de l'image EXT4 produite est légèrement supérieure à la précédente (20 Mo contre 18 Mo précédemment) mais la cible dispose désormais de la de gestion des paquets IPK (utilitaires `opkg`, base de données).

```
root@qemuarm:~# opkg info busybox
Package: busybox
Version: 1.31.1-r0
Depends: libc6 (>= 2.31+git0+df31c7ca92),
update-alternatives-opkg
Recommends: busybox-udhcpc
Status: install ok installed
Architecture: armv7vet2hf-neon
Installed-Size: 521843
Installed-Time: 1634498725
```

Une fois la fonctionnalité activée sur la cible, on peut configurer le PC de développement comme serveur de paquets. Pour cela, il faut tout d'abord produire l'index des paquets en utilisant la commande `bitbake` :

```
$ bitbake package-index
```

Le résultat dépend du format de paquet choisi et dans le cas du format IPK on notera l'apparition de fichiers `Packages.gz` .

```
$ find tmp/deploy/ipk -name Packages.gz
tmp/deploy/ipk/all/Packages.gz
tmp/deploy/ipk/x86_64-nativesdk/Packages.gz
tmp/deploy/ipk/qemuarm/Packages.gz
tmp/deploy/ipk/armv7vet2hf-neon/Packages.gz
```

On doit ensuite ouvrir un accès HTTP sur les répertoires contenant les paquets. Dans le cas d'un test, on peut utiliser Python qui utilise le port 8000.

```
$ cd tmp/deploy/ipk
$ python -m SimpleHTTPServer
```

Côté cible, il faut indiquer l'adresse IP et le port utilisé en complétant le fichier `/etc/opkg/opkg.conf`. Dans le cas d'une cible QEMU, la cible utilise l'adresse IP fixe 192.168.7.2 et le PC l'adresse 192.168.7.1. Les lignes à ajouter sont donc les suivantes :

```
src/gz all http://192.168.7.1:8000/all
src/gz armv7vet2hf-neon
http://192.168.7.1:8000/armv7vet2hf-neon
src/gz qemuarm http://192.168.7.1:8000/qemuarm
```

A partir de là, on peut mettre à jour la base de données sur la cible puis manipuler les paquets en utilisant la commande `opkg`. On note que seulement 36 paquets sont installés sur la cible.

```
root@qemuarm:~# opkg update
Downloading http://192.168.7.1:8000/all/Packages.gz.
Updated source 'all'.
Downloading
http://192.168.7.1:8000/armv7vet2hf-neon/Packages.gz.
```

```
Updated source 'armv7vet2hf-neon'.
Downloading http://192.168.7.1:8000/qemuarm/Packages.gz.
Updated source 'qemuarm'.

root@qemuarm:~# opkg list-installed | wc -l
36
```

L'empreinte mémoire de cette image est effectivement très réduite (moins de 10 Mo).

```
root@qemuarm:~# df -h
Filesystem                Size      Used Available Use%
Mounted on
/dev/root                18.3M      9.7M      7.2M  57% /
```

Création de recettes

Pour l'instant nous avons utilisé les recettes fournies par Yocto ou bien des recettes annexes comme le contenu de meta-raspberrypi. Nous allons désormais aborder la création de recette dans le cas d'exemples simples de type « Hello World ». Les recettes doivent être placées dans des layers externes car il n'est pas conseillé de modifier les layers existants. La commande `bitbake-layers` nous permet de créer un layer de test `meta-example-lb` ainsi qu'un exemple de recette (ne contenant aucun code) dans le répertoire des sources de Yocto.

On crée tout d'abord un layer dédié dans le répertoire de construction.

```
$ cd poky
$ source oe-init-build-env qemuarm-build
$ bitbake-layers create-layer ../meta-example-lb
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer
../meta-example-lb'
```

On ajoute donc la référence du layer au fichier `bblayers.conf`. en utilisant

de nouveau la commande `bitbake-layers`.

```
$ bitbake-layers add-layer ../meta-example-lb
```

On peut alors produire le paquet d'exemple.

```
$ bitbake example
...
NOTE: Executing Tasks
*********************************************
*                                           *
*  Example recipe created by bitbake-layers   *
*                                           *
*********************************************
NOTE: Tasks Summary: Attempted 675 tasks of which 657 didn't
need to be rerun and all succeeded.
```

Ajout d'une recette simple

L'exemple précédent ne contient pas de code source et il n'est donc pas possible de le tester sur la cible. Nous ajoutons donc au répertoire `recipe-example` une recette dont les sources utilisent l'outil CMake. Cette application correspond au fameux programme en C réalisant la conversion des degrés Celsius en Fahrenheit.

```
$ gcc -o fahr fahr.c
$ ./fahr 20
68
```

Une recette réelle correspond à un répertoire (soir `fahr` dans notre cas) et contenant un fichier de recette `fahr_2.0.bb`. La version de la recette peut être codée dans le nom du fichier ou bien définie dans le fichier `.bb` par la variable `PV`.
Le répertoire de la recette peut également contenir un sous-répertoire fournissant d'autres données pour la compilation de la recette. Le nom du sous-répertoire correspond le plus souvent au nom du fichier de recette (avec

ou sans la version) ou à `files` . L'exemple ci-dessous correspond à la recette *expat* fournie avec Yocto car notre recette n'utilise pas - pour l'instant - de sous-répertoire.

```
expat/
├── expat
│   ├── 0001-Add-output-of-tests-result.patch
│   ├── libtool-tag.patch
│   └── run-ptest
└── expat_2.2.10.bb
```

De même, les recettes doivent obligatoirement être placées dans un répertoire de type `recipes-<whatever>`, le plus souvent `recipes-core`, `recipes-kernel`, `recipes-graphics` en fonction de la catégorie de la recette. Notre recette utilise des sources externes et on doit donc fournir le checksum de l'archive ainsi que celui de la licence utilisée (GPL). La variable `SRC_URI` est également fondamentale car elle indique les URL d'accès à tous les composants nécessaires à la recette et ce en utilisant tous les protocoles connus (`git://`, `http://`, `file://`, etc.). Le cas de l'opérateur file:// correspond à une ressource disponible dans le sous-répertoire de la recette évoqué précédemment. En cas d'utilisation d'une archive externe, on doit préciser le checksum du fichier afin de garantir l'intégrité du paquet binaire produit. Dans le cas de notre exemple, le composant étant basé sur CMake, la méthode de compilation se borne à l'héritage de la classe « cmake » fournie par Yocto dans le répertoire `meta/classes` .

```
DESCRIPTION = "Celsius to Fahrenheit utility (CMake based)"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://COPYING;md5=8ca43cbc842c2336e835926c2166c28b"

# Package Revision
PR = "r0"

SRC_URI = "http://pficheux.free.fr/pub/tmp/fahr-2.0.tar.gz"
```

```
inherit cmake

SRC_URI[md5sum] = "8cd41891470ea0e909181d3a1ec6d47e"
```

Le paquet correspondant peut être construit par la commande :

```
$ bitbake fahr
```

On peut vérifier la présence et le contenu du paquet par la commande :

```
$ dpkg -c
tmp/deploy/ipk/armv7vet2hf-neon/fahr_2.0-r0_armv7vet2hf-neo
n.ipk
drwxr-xr-x root/root         0 2020-01-27 14:10 ./usr/
drwxr-xr-x root/root         0 2020-01-27 14:10 ./usr/bin/
-rwxr-xr-x root/root      5508 2020-01-27 14:10
./usr/bin/fahr
```

On peut alors mettre à jour la base de données sur le PC de développement puis sur la cible et enfin installer le paquet.

```
$ bitbake package-index

root@qemuarm:~# opkg update
Downloading http://192.168.7.1:8000/all/Packages.gz.
Updated source 'all'.
Downloading
http://192.168.7.1:8000/armv7vet2hf-neon/Packages.gz.
Updated source 'armv7vet2hf-neon'.
Downloading http://192.168.7.1:8000/qemuarm/Packages.gz.
Updated source 'qemuarm'.

root@qemuarm:~# opkg install fahr
Installing fahr (2.0) on root
Downloading
http://192.168.7.1:8000/armv7vet2hf-neon/fahr_2.0-r0_armv7v
```

```
et2hf-neon.ipk.
Configuring fahr.

root@qemuarm:~# fahr 20
68
```

Il peut être intéressant de construire une image intégrant directement les paquets (et les features) nécessaires. Pour cela, on peut dans un premier temps ajouter la ligne suivante au fichier `local.conf`.

```
IMAGE_INSTALL_append = " fahr"
```

Après construction de l'image, on note la présence du paquet correspondant.

```
$ bitbake core-image-minimal
$ runqemu nographic
...
[    4.977198] udevd[128]: starting version 3.2.9
[    5.032129] udevd[129]: starting eudev-3.2.9
[    6.661121] EXT4-fs (vda): re-mounted. Opts: (null)
[    6.661534] ext4 filesystem being remounted at /
supports timestamps until 2038 (0x7fffffff)

Poky (Yocto Project Reference Distro) 3.1.6 qemuarm
/dev/ttyAMA0

qemuarm login: root

root@qemuarm:~# fahr 20
68
```

Recette d'image

L'utilisation du fichier `local.conf` pour l'ajout de recette n'est cependant pas la méthode la plus usuelle et lorsque la configuration est prête, il est conseillé de créer une recette d'image dérivée des exemples fournis par

Yocto (ou par le BSP). Dans le cas de la Raspberry Pi, voici quelques exemples dans le répertoire `meta-raspberrypi/recipes-core/images` .

```
$ ls -1
rpi-basic-image.bb
rpi-hwup-image.bb
rpi-test-image.bb
```

On peut donc créer dans notre layer une recette d'image spécifique qui intègre notre paquet ajouté ce qui correspond au fichier de recette `meta-example-lb/recipes-core/images/images/lb-test-image.bb` dont le contenu est le suivant :

```
# Base this image on core-image-minimal
include recipes-core/images/core-image-minimal.bb

# Added packages
IMAGE_INSTALL += "fahr"

# Added features
IMAGE_FEATURES += "package-management"
```

On crée la nouvelle image puis on la teste en utilisant de nouveau `runqemu` en précisant le nom de l'image.

```
$ bitbake lb-test-image
```

```
$ runqemu lb-test-image nographic
[    5.059585] udevd[129]: starting eudev-3.2.9
[    6.728888] EXT4-fs (vda): re-mounted. Opts: (null)
[    6.729303] ext4 filesystem being remounted at /
supports timestamps until 2038 (0x7fffffff)

Poky (Yocto Project Reference Distro) 3.1.6 qemuarm
/dev/ttyAMA0
```

```
qemuarm login: root
root@qemuarm:~# fahr 20
68
```

Extension de recette (fichier `.bbappend`)

Yocto est un outil majeur dans l'écosystème du logiciel libre. Le principe même de logiciel libre est de s'appuyer sur le développement des communautés et si un composant est nécessaire à un projet, on veillera tout d'abord à vérifier s'il existe dans l'écosystème. Dans le cas de Yocto, on peut explorer les projets disponibles grâce à l'outil de recherche « Layer Index » sur http://layers.openembedded.org/layerindex. Cette page permet une recherche sur plusieurs catégories (recette, layer, cible matérielle, classe, etc.). Si le composant existe, on obtient une page indiquant entre autres l'URL du dépôt du projet ainsi que les dépendances par rapport à d'autres layers. Suivant les cas, le composant sera utilisé en l'état ou bien modifié pour les besoins du projet.

Modification d'une recette

La modification d'une recette passe souvent par la mise à jour des composants référencés dans la variable `SRC_URI` (distants ou locaux à la recette). On peut partir de l'exemple de la recette permettant de mettre en place un « splash screen » disponible dans `meta/recipes-core/psplash`, dont l'arborescence est donnée ci-dessous :

```
psplash
├── files
│   ├── psplash-init
│   ├── psplash-poky-img.h
│   ├── psplash-start.service
│   └── psplash-systemd.service
└── psplash_git.bb
```

Le logo utilisé est défini dans le fichier local `psplash-poky-img.h`. Si on veut modifier ce logo pour les besoins du projet, on peut imaginer de modifier

ce fichier mais les sources de Yocto étant gérées sous Git, nous rappelons qu'il est très déconseillé de modifier la recette initiale. On pourrait également dupliquer la recette dans le layer `meta-example-lb` puis modifier les composants mais cette méthode est limitée car les modifications devront être reportées à chaque mise à jour de la recette par la communauté Yocto.

Principe du .bbappend

Le principe est de créer - dans un autre layer - une recette étendue (suffixe `.bbappend`) permettant de modifier les éléments de la recette tout en conservant le fichier `.bb` initial.
Si on reprend l'exemple de la recette précédente, l'arborescence de notre nouvelle recette étendue est la suivante :

```
meta-poky/recipes-core/psplash/
├── files
│   └── psplash-poky-img.h
└── psplash_git.bbappend
```

Le fichier `psplash-poky-img.h` remplace donc celui de la recette initiale (car il porte le même nom). En l'occurrence, il s'agit du nouveau logo correspondant à la distribution « Poky » (définie dans le layer `meta-poky`) et non plus OpenEmbedded. Le fichier `.bbappend` contient uniquement la ligne suivante :

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

Cette ligne indique que l'on ajoute le sous-répertoire de la recette étendue au chemin de recherche des composants de la recette initiale et ce *uniquement* si la variable `DISTRO` (qui définit la distribution utilisée) est égale à `poky`. Ce dernier point est un cas particulier liée à la recette (qui concerne le logo). La plupart du temps, l'extension ne dépendra pas de la distribution. On peut connaître la liste des recette étendues en utilisant la commande :

```
$ bitbake-layers show-appends
```

Cette liste contient les `.bbappend` correspondant à des `.bb` pour la liste des layers chargés dans l'environnement de compilation. Suivant la valeur de la variable `DISTRO`, les images utilisent des paramètres définis par la distribution (voir `meta/conf/distro/poky.conf`). En plus de fournir des images spécifiques (voir précédemment), certains BSP fournissent des distributions spécifiques (cas des BSP pour les modules i.MX).

```
$ ls sources/meta-boundary/recipes-boundary/images/
boundary-image-multimedia-full.bb

$ ls sources/meta-freescale-distro/conf/distro/
fslc-framebuffer.conf  fslc-xwayland.conf    fsl-x11.conf
fslc-wayland.conf      fsl-framebuffer.conf
fsl-xwayland.conf
fslc-x11.conf          fsl-wayland.conf      include
```

Cas d'utilisation du .bbappend

L'utilité du `.bbappend` ne se borne pas à l'ajout de fichiers à une recette. On peut par exemple créer une extension de recette afin d'appliquer des patch aux sources liées à la recette. L'arborescence de l'extension sera la suivante :

```
recipes-core/mypack-auto/
├── files
│   └── hello.patch
└── mypack-auto_1.0.bbappend
```

Dans l'exemple ci-dessous, nous étendons la recette du noyau de la Raspberry Pi afin de charger automatiquement le module noyau i2c-dev au démarrage.

```
$ tree recipes-kernel/linux-raspberrypi
recipes-kernel/linux-raspberrypi/
└── linux-raspberrypi_%.bbappend

$ cat
recipes-kernel/linux-raspberrypi/linux-raspberrypi_%.bbappe
```

```
nd
KERNEL_MODULE_AUTOLOAD += "i2c-dev"
```

Au début de ce livre, nous avons évoqué les étapes (et donc les fonctions) utilisées par BitBake. Grâce à l'extension de recette, on peut facilement ajouter une étape exécutée avant ou après une étape standard. Dans l'exemple suivant, on active le bus I2C d'une Raspberry Pi en modifiant le fichier `config.txt` après son déploiement sur `tmp/deploy/image`.

```
$ cat recipes-bsp/bootfiles/rpi-config_git.bbappend

do_deploy_append() {
    # Enable i2c by default
    echo "dtparam=i2c_arm=on" >>
${DEPLOYDIR}/bootfiles/config.txt
}
```

Utilisation de Devtool

La commande `devtool` est fournie dans l'environnement Yocto et permet d'effectuer différentes opérations sur les recettes :

- Créer une recette à partir des sources du composant (add)
- Modifier une recette existante sur la base d'une modification des sources originaux (modify)
- Mettre à jour d'une recette suite à la mise à jour des sources « upstream » (update)

La deuxième option est souvent utilisée et nous allons la tester dans ce livre en reprenant l'exemple de la recette fahr pour modifier simplement le format d'affichage. Grâce à Devtool, nous allons créer un `.bbappend` pour la recette modifiée par un patch.
En premier lieu, nous créons un nouveau layer pour héberger la recette étendue.

```
$ bitbake-layers create-layer ../meta-example-lb-ext
```

```
$ bitbake-layers add-layer ../meta-example-lb-ext
```

La prochaine étape consiste à charger la recette fahr dans l'environnement de Devtool, ce qui a pour effet de créer un répertoire `workspace` ajouté à la liste des layers.

```
$ devtool modify fahr
```

On peut alors modifier le code source et utiliser Git pour valider la modification.

```
$ cd workspace/sources/fahr
$ vi fahr.c
$ git commit -a -m "updated format"
```

Finalement, on crée le fichier `.bbappend` dans le nouveau layer. Une fois l'opération terminée, on peut supprimer le répertoire `workspace`.

```
$ devtool finish fahr ../../../../meta-example-lb-ext
```

On constate que l'outil Devtool a créé l'arborescence suivante :

```
meta-example-lb-ext/recipes-example/fahr
├── fahr
│   └── 0001-Updated-format.patch
└── fahr_%.bbappend
```

Après installation du nouveau paquet sur la cible, on obtient le résultat attendu.

```
root@qemuarm:~# fahr 20
20 °C -> 68 °F
```

Production du SDK

Un compilateur croisé est produit lors de la première production de l'image mais cette version n'est pas utilisable à l'extérieur de Yocto (sauf en utilisant `bitbake -c devshell`, ce qui n'a que peu d'intérêt). Il est possible de créer un compilateur croisé que l'on pourra installer sur une machine compatible même si elle ne dispose pas de l'environnement Yocto. Pour cela on peut utiliser la commande `bitbake meta-toolchain` qui produit une archive du SDK dans `tmp/deploy/sdk` (sous la forme d'un script à exécuter). Une option plus avancée permet de créer un SDK contenant non seulement le compilateur mais également les bibliothèques spéciales installées sur la cible par des recettes ajoutées. Pour cela on utilise la commande :

```
$ bitbake -c populate_sdk <nom-recette-image>
```

L'utilisation de l'option `populate_sdk_ext` permet d'ajouter d'autres fonctionnalités dont l'outil Devtool évoqué précédemment (on parle alors de SDK *étendu* ou eSDK). Dans tous les cas de figure, et à l'issue de la compilation, le script contenant le SDK est disponible dans le répertoire `tmp/deploy/sdk`. L'installation s'effectue simplement en exécutant le script en tant qu'administrateur.

```
$ sudo
tmp/deploy/sdk/poky-glibc-x86_64-meta-toolchain-armv7vet2hf
-neonqemuarm-toolchain-3.1.6.sh
```

Utilisation du compilateur

Pour utiliser la chaîne de compilation, on charge les variables d'environnement nécessaire en utilisant de nouveau la commande `source` et un script fourni par Yocto.

```
$ source
/opt/poky/3.1.6/environment-setup-armv7vet2hf-neon-poky-lin
ux-gnueabi
```

La variable `CC` (habituellement vide) contient désormais le nom de la commande du compilateur croisée ainsi que les options indispensables à son utilisation.

```
$ echo $CC
arm-poky-linux-gnueabi-gcc -march=armv7ve -mthumb
-mfpu=neon -mfloat-abi=hard
--sysroot=/opt/poky/3.1.6/sysroots/armv7vet2hf-neon-poky-li
nux-gnueabi
```

Il est donc nécessaire d'utiliser `$CC` et non `arm-poky-linux-gnueabi-gcc` y compris pour un test très simple.

```
$ $CC -o hello hello.c
```

Bien entendu, ce compilateur peut être utilisé avec des outils classiques comme Autotools ou CMake. On peut par exemple compiler les sources de l'exemple précédent (le programme `fahr`) avec ce compilateur. L'outil CMake prend en compte le contenu de la variable `CC` pour utiliser le compilateur croisé adéquat et produire le fichier `Makefile`.

```
$ cd fahr-2.0
$ mkdir build && cd build
$ cmake ..
$ make
Scanning dependencies of target fahr
[ 50%] Building C object CMakeFiles/fahr.dir/fahr.o
[100%] Linking C executable fahr
[100%] Built target fahr
```

Dans le cas d'un exemple basé sur Autotools, on construit le script `configure` grâce à `autoreconf` puis on l'exécute en précisant le compilateur à utiliser ce qui produit le fichier `Makefile`.

```
$ cd mypack-auto
```

```
$ autoreconf -f -i -v
$ mkdir build && cd build
$ ../configure --host=arm-poky-linux-gnueabi
$ make
```

Intégration continue

L'intégration continue (IC ou CI) est un sujet phare dans l'industrie du logiciel. Ce point est particulièrement crucial pour les systèmes embarqués qui doivent avoir une fiabilité exemplaire soit parce qu'ils sont largement diffusés soit parce qu'ils assurent des tâches sensibles (et souvent les deux !). L'IC permet de vérifier que la mise à jour d'un composant ne provoque pas de régression sur le fonctionnement du système. L'outil Yocto permet de dérouler des tests à deux niveaux :

1. Au niveau du paquet en utilisant la classe « ptest » . Pour cela, on exécute un test fourni dans un paquet ajouté (`<nom-de-package>-ptest`).
2. Au niveau de l'image complète en utilisant la classe « testimage » . Yocto fournit pour celà des scripts de test (en Python) pour valider des fonctionnalités globale comme le réseau, l'accès SSH, etc. On peut bien entendu ajouter de nouveaux scripts dans un layer spécial.

Test de paquet (ptest)

L'utilisation de la classe « ptest » conduit à l'ajout de la commande `ptest-runner` qui permet d'exécuter les tests pour chaque paquet. Nous allons de nouveau utiliser la recette fahr afin de fournir un script de test et une liste de données à tester. Cette liste (que nous nommons `test.dat`) correspond à un tableau contenant les températures en °C (valeurs d'entrée) et celles en °F (valeurs de sortie / résultats).

```
10      50
20      68
30      86
```

Le script (qui doit être nommé `run-ptest`), lit le tableau et compare la valeur de la deuxième colonne avec le résultat produit par l'exécutable `fahr` en utilisant la valeur de la première colonne en entrée du programme. Ce script est écrit en shell, mais on pourrait très bien utiliser un autre langage.

```
#!/bin/sh
R=0
while read line
do
    set $line
    if [ "$(fahr $1)" != "$2" ]; then
      R=1
      break
    fi
done < test.dat

if [ $R -eq 0 ]; then
    echo PASS
else
    echo FAILED
fi
exit $R
```

Nous reprenons la recette précédente à laquelle nous ajoutons les éléments liés à la classe « ptest ». Le répertoire contient ainsi désormais les fichiers suivants :

```
fahr
├── fahr_2.0.bb
└── files
    ├── run-ptest
    └── test.dat
```

Le contenu du fichier de recette est également mis à jour comme suit. Nous faisons apparaître les modifications en gras.

```
DESCRIPTION = "Celsius to Fahrenheit utility (CMake based)"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM =
"file://COPYING;md5=8ca43cbc842c2336e835926c2166c28b"
PR = "r0"

SRC_URI = "http://pficheux.free.fr/pub/tmp/fahr-2.0.tar.gz"
SRC_URI += "file://run-ptest file://test.dat"

inherit cmake ptest

do_install_ptest () {
      cp ${WORKDIR}/test.dat ${D}${PTEST_PATH}
}

SRC_URI[md5sum] = "8cd41891470ea0e909181d3a1ec6d47e"
```

Nous ajoutons finalement les lignes suivantes au fichier `local.conf`. La feature « ptest-pkgs » permet de construire une image incluant les paquets liés à la classe « ptest ».

```
# Ajout du paquet fahr à la cible + les paquets ptest
IMAGE_INSTALL_append = " fahr"
EXTRA_IMAGE_FEATURES += "ptest-pkgs"
```

Après la compilation puis l'exécution de l'image nous pouvons utiliser `ptest-runner`. Nous constatons que plusieurs paquets fournis par Yocto utilisent la classe. Il est nécessaire de retirer le layer contenant le `.bbappend` (et donc le patch) afin d'obtenir la sortie attendue pour le programme `fahr`.

```
$ bitbake-layers remove-layer ../meta-example-lb-ext
$ bitbake core-image-minimal

$ runqemu nographic
...
Poky (Yocto Project Reference Distro) 3.1.6 qemuarm ttyAMA0
```

```
qemuarm login: root
root@qemuarm:~# ptest-runner -l
Available ptests:
acl    /usr/lib/acl/ptest/run-ptest
attr   /usr/lib/attr/ptest/run-ptest
busybox      /usr/lib/busybox/ptest/run-ptest
bzip2 /usr/lib/bzip2/ptest/run-ptest
fahr   /usr/lib/fahr/ptest/run-ptest
libxml2      /usr/lib/libxml2/ptest/run-ptest
lzo    /usr/lib/lzo/ptest/run-ptest
opkg   /usr/lib/opkg/ptest/run-ptest
util-linux   /usr/lib/util-linux/ptest/run-ptest
zlib   /usr/lib/zlib/ptest/run-ptest
```

Nous pouvons ensuite tester notre paquet.

```
root@qemuarm:~# ptest-runner fahr
START: ptest-runner
2021-10-19T16:41
BEGIN: /usr/lib/fahr/ptest
PASS
DURATION: 1
END: /usr/lib/fahr/ptest
2021-10-19T16:41
STOP: ptest-runner
```

Test d'une image (testimage)

La classe « testimage » utilise une approche différente car il s'agit de tester des fonctionnalités générales de l'image. La classe n'est donc pas utilisable dans une recette mais de manière globale, donc dans `local.conf`. Les scripts de test fournis par Yocto sont dans le répertoire suivant :

```
$ ls meta/lib/oeqa/runtime/cases/
apt.py              gstreamer.py       pam.py         scp.py
```

```
boot.py                kernelmodule.py    parselogs.py
skeletoninit.py
buildcpio.py           ksample.py         perl.py        ssh.py
buildgalculator.py     ldd.py             ping.py
stap.py
buildlzip.py           logrotate.py       ptest.py
storage.py
connman.py             ltp_compliance.py  __pycache__
systemd.py
date.py                ltp.py             python.py
weston.py
df.py                  ltp_stress.py      _qemutiny.py
x32lib.py
dnf.py                 multilib.py        rpm.py
xorg.py
gcc.py                 oe_syslog.py       runlevel.py
gi.py                  opkg.py            scons.py
```

On peut bien entendu reproduire cette arborescence dans un layer spécial et définir ses propres tests. Si on veut tester le fonctionnement du réseau (ping), on doit ajouter les lignes suivantes au fichier `local.conf` :

```
INHERIT += "testimage"
TEST_SUITES = "ping"
```

Dans le cas de QEMU, l'image est exécutée puis les tests sont déroulés. Suite à celà, l'exécution de l'image est interrompue avec affichage du résultat.

```
$ bitbake -c testimage core-image-minimal
...
RESULTS:
RESULTS - ping.PingTest.test_ping: PASSED (0.03s)
SUMMARY:
core-image-minimal () - Ran 1 test in 0.026s
core-image-minimal - OK - All required tests passed
(successes=1, skipped=0, failures=0, errors=0)
```

```
NOTE: Tasks Summary: Attempted 601 tasks of which 599
didn't need to be rerun and all succeeded.
```

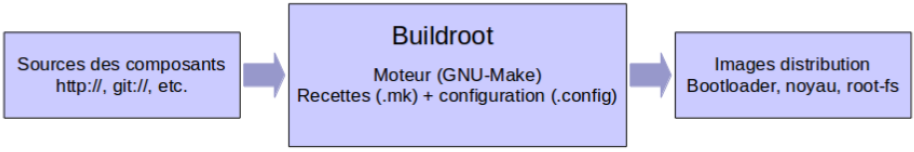
Dans le cas d'une carte réelle, on dispose de plusieurs méthodes de test, la méthode par défaut correspondant à « simpleremote » qui considère que la carte est déjà démarrée. Dans ce cas, on devra renseigner l'adresse IP du PC de test et celle de la cible dans le fichier `local.conf`.

```
# For real board (not QEMU)
TEST_TARGET = "simpleremote"
TEST_SERVER_IP = "<PC-IP-address>"
TEST_TARGET_IP = "<board-IP-address>"
```

Les autres méthodes sont décrites dans la documentation de Yocto [12] et la-aussi il est possible de définir sa propre méthode.

Quelques mots sur Buildroot

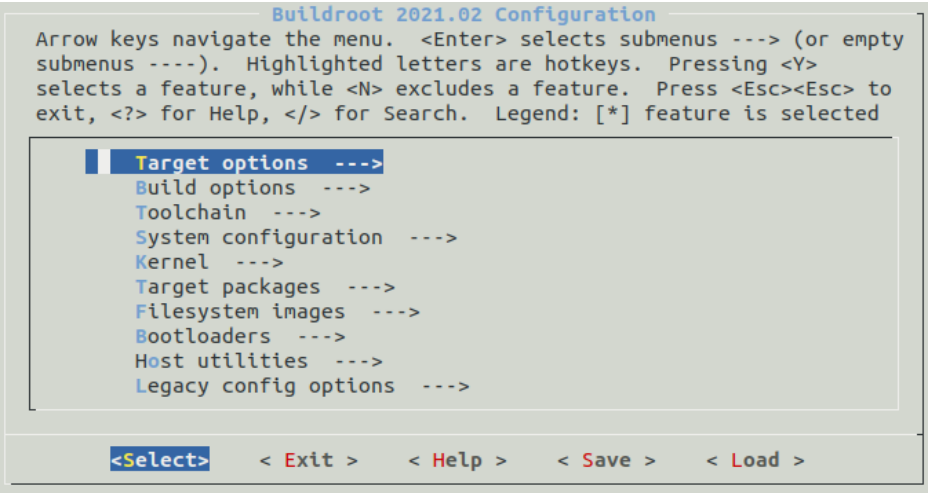
Ce livre est largement centré sur l'outil Yocto car ce dernier est désormais un standard de l'industrie. Le projet Buildroot (plus ancien que Yocto mais toujours très bien maintenu - avec une nouvelle version tous les trois mois) a cependant quelques avantages par rapport à Yocto. En premier lieu, Buildroot est plus simple d'approche même s'il utilise les mêmes principes que Yocto, comme on peut le constater sur la figure ci-dessous.



Fonctionnement de Buildroot

Buildroot dispose d'une interface graphique de configuration similaire à celle du noyau Linux ou de BusyBox. Une fois la configuration choisie parmi une

liste de fichiers `defconfig`, on peut aisément modifier les paramètres avec l'IHM.



Configuration de Buildroot

```
$ cd <BR-src-dir>
$ make raspberrypi3_defconfig # chargement de la
configuration pour Pi 3
$ make menuconfig             # modification de la
configuration (optionnel)
...
$ make                        # construction de l'image
```

Comme on peut le constater, Buildroot est basé sur GNU-Make (la commande `make` de GNU/Linux) et non sur BitBake. GNU-Make est un outil déjà bien connu des développeurs Linux et il est donc plus simple d'approche que BitBake. De plus, la construction d'une image Linux par Buildroot est plus rapide qu'avec Yocto et il est donc plus simple de réaliser un test rapide avec Buildroot si la cible est déjà prise en charge par un fichier `defconfig` fourni.

La principale limitation de Buildroot est son approche « statique » car il n'est pas basé sur la notion de layer externe. L'utilisation de la variable `BR2_EXTERNAL` [13] permet certes d'utiliser un répertoire externe pour ajouter des composants (répertoire `package`) des patch ou des fichiers de configuration mais cette approche reste moins souple que celle de Yocto.
Rappelons également qu'une image Buildroot ne peut disposer d'un gestionnaire de paquets comme avec Yocto. Il est vrai que la plupart des images utilisées en production n'utilisent pas de gestionnaire de paquets mais la disponibilité de cette option dans un environnement de développement peut être un gros avantage.

Conclusion

Après avoir introduit les concepts de base liés aux systèmes basés sur Linux embarqué, ce livre nous a permis d'évoquer le principal outil utilisable pour la production d'une distribution Linux réduite et maîtrisée.

Buildroot peut convenir dans le cas d'un système simple pour lequel la gestion de paquets n'est pas nécessaire. Cependant il a tout de même quelques limitations quant à l'intégration de composants externes si on le compare à Yocto, ce dernier étant beaucoup plus modulaire et dynamique. Yocto nécessite cependant un plus grand investissement pour sa prise en main (compter au minimum un mois pour un ingénieur expérimenté dans le domaine Linux industriel). Cependant son approche très modulaire et sa large utilisation du principe d'héritage en fait un outil très puissant pour la maintenance d'une offre logicielle variée (plusieurs cartes et plusieurs projets). Autre point et non des moindres, Yocto est de nos jours utilisé par la plupart des fournisseurs de matériel et une connaissance de base de cet outil est nécessaire pour l'exploitation des BSP disponibles. Enfin, Yocto est à la base de plusieurs outils de développement commerciaux (comme ceux de Wind River, SIEMENS, etc.) ce qui permettra de migrer sans trop d'effort les travaux réalisés vers ces environnements si cela est nécessaire.

Bibliographie

[1] Apollo Guidance Computer
https://en.wikipedia.org/wiki/Apollo_Guidance_Computer
[2] Calculateur D17-B https://en.wikipedia.org/wiki/D-17B
[3] RTOS/360
https://www.computer.org/csdl/proceedings/afips/1969/5073/00/50730015.pdf
[4] Simulateur Javascript de l'AGC
http://svtsim.com/moonjs/agc.html
[5] Système MULTICS https://en.wikipedia.org/wiki/Multics
[6] SPS7 et SM-90
http://www.feb-patrimoine.com/projet/unix/sps7.htm
[7] Utiliser ELBE https://www.youtube.com/watch?v=BwHzyCGB7As
[8] Utiliser DEBOS https://www.youtube.com/watch?v=57Lmv1hG9T0
[9] BSP Yocto pour Raspberry Pi
http://git.yoctoproject.org/cgit/cgit.cgi/meta-raspberrypi
[10] BSP Yocto pour les modules Toradex
https://developer.toradex.com/knowledge-base/board-support-package/openembedded-core
[11] Livre blanc « Mise à jour des systèmes embarqués » (Smile)
https://www.smile.eu/fr/livres-blancslivres-blancs/mise-jour-systemes-embarques
[12] Utilisation de la classe « testimage »
https://www.yoctoproject.org/docs/latest/mega-manual/mega-manual.html#performing-automated-runtime-testing
[13] Project specific customization (Buildroot)
https://buildroot.org/downloads/manual/manual.html#customize

www.smile.eu