OpenEmbedded
Découverte et premiers pas

Cyril Romain (cyril.romain@gmail.com)

Association Toulibre

17 décembre 2008

OpenEmbedded en quelques mots

OpenEmbedded est un framework de compilation de composants logiciels libres destinés à être déployés sur des systèmes embarqués.

- production d'un simple binaire jusqu'à une distribution complète
- support de nombreuses architectures et configurations
- flexible
- autosuffisant et déterministe

Construire une distribution Linux

Les briques logiciels de base pour construire un système Linux

- glibc: GNU C library.
- gcc: GNU C compiler. [1]
- binutils: outils manipulant les fichiers objet et binaire (as, ld, nm, objdump, etc.)

Ces éléments constituent une **toolchain**, avec laquelle peut ensuite être compilé tout autre composant (make, le kernel, X, KDE, ...)

[1] Problème de l'oeuf et de la poule: compiler gcc nécessite gcc ! gcc est en fait compilé en plusieurs étapes: phase de boostrap, puis compilation de gcc avec lui même

Cross-compilation

Une toolchain permet de compiler sur une machine hôte ('build', 'native') d'architecture A du code executable sur une machine cible ('target') d'architecture B.

[2] il existe aussi la canadian cross-toolchain: compilée sur une archi A pour être executée sur une archi B produisant du code pour une archi C avec A!=B!=C

Cross-compilation

Une toolchain permet de compiler sur une machine hôte ('build', 'native') d'architecture A du code executable sur une machine cible ('target') d'architecture B.

- si A == B: on parle de toolchain et de compilation

[2] il existe aussi la canadian cross-toolchain: compilée sur une archi A pour être executée sur une archi B produisant du code pour une archi C avec A!=B!=C

Cross-compilation

Une toolchain permet de compiler sur une machine hôte ('build', 'native') d'architecture A du code executable sur une machine cible ('target') d'architecture B.

- si A == B: on parle de toolchain et de compilation
- si A != B: on parle de cross-toolchain et de cross-compilation [2]

[2] il existe aussi la canadian cross-toolchain: compilée sur une archi A pour être executée sur une archi B produisant du code pour une archi C avec A!=B!=C

Compilation pour l'embarqué

Pour compiler du code pour une machine cible, il y a plusieurs possibilités

Compilation pour l'embarqué

Pour compiler du code pour une machine cible, il y a plusieurs possibilités

1. récupérer une standalone/prebuilt toochain/SDK s'executant sur la machine cible. Rarement viable par manque de mémoire et puissance de la machine embarquée

Compilation pour l'embarqué

Pour compiler du code pour une machine cible, il y a plusieurs possibilités

1. récupérer une standalone/prebuilt toochain/SDK s'executant sur la machine cible. Rarement viable par manque de mémoire et puissance de la machine embarquée
2. compiler une toochain/SDK executable sur la machine cible

Compilation pour l'embarqué

Pour compiler du code pour une machine cible, il y a plusieurs possibilités

1. récupérer une standalone/prebuilt toochain/SDK s'executant sur la machine cible. Rarement viable par manque de mémoire et puissance de la machine embarquée
2. compiler une toochain/SDK executable sur la machine cible
3. récupérer une cross-toochain s'executant sur une machine hôte (ex: x86) générant du code spécifique à la machine cible (ex: ARM)

Compilation pour l'embarqué

Pour compiler du code pour une machine cible, il y a plusieurs possibilités

1. récupérer une standalone/prebuilt toochain/SDK s'executant sur la machine cible. Rarement viable par manque de mémoire et puissance de la machine embarquée
2. compiler une toochain/SDK executable sur la machine cible
3. récupérer une cross-toochain s'executant sur une machine hôte (ex: x86) générant du code spécifique à la machine cible (ex: ARM)
4. compiler une cross-toochain

Systèmes de build

Liste non exhaustive des systèmes de build permettant la cross-compilation

Systèmes de build

Liste non exhaustive des systèmes de build permettant la cross-compilation

- à la main: long, fastidieux, connaissances techniques requises

Systèmes de build

Liste non exhaustive des systèmes de build permettant la cross-compilation

- à la main: long, fastidieux, connaissances techniques requises
- crossdev / crosstool: créer des cross-toolchain **uniquement**

Systèmes de build

Liste non exhaustive des systèmes de build permettant la cross-compilation

- à la main: long, fastidieux, connaissances techniques requises
- crossdev / crosstool: créer des cross-toolchain **uniquement**
- buildroot: basé sur des Makefiles, toolchain + gestion des dépendances

Systèmes de build

Liste non exhaustive des systèmes de build permettant la cross-compilation

- à la main: long, fastidieux, connaissances techniques requises
- crossdev / crosstool: créer des cross-toolchain **uniquement**
- buildroot: basé sur des Makefiles, toolchain + gestion des dépendances
- Scratchbox

Systèmes de build

Liste non exhaustive des systèmes de build permettant la cross-compilation

- à la main: long, fastidieux, connaissances techniques requises
- crossdev / crosstool: créer des cross-toolchain **uniquement**
- buildroot: basé sur des Makefiles, toolchain + gestion des dépendances
- Scratchbox
- T2: nécessite d'être root : −(

Systèmes de build

Liste non exhaustive des systèmes de build permettant la cross-compilation

- à la main: long, fastidieux, connaissances techniques requises
- crossdev / crosstool: créer des cross-toolchain **uniquement**
- buildroot: basé sur des Makefiles, toolchain + gestion des dépendances
- Scratchbox
- T2: nécessite d'être root : −(
- OpenEmbedded

Pourquoi avoir créer OpenEmbedded ?

Les systèmes de build existants ne gèrent pas les dépendances ou montrent vite leurs limites d'utilisation grande échelle par

OpenEmbedded a été crée en 2003 par Chris Larson

Pourquoi avoir créer OpenEmbedded ?

Les systèmes de build existants ne gèrent pas les dépendances ou montrent vite leurs limites d'utilisation grande échelle par

- manque de flexibilité
- lourdeur de maintenance (ex: des makefiles)

OpenEmbedded a été crée en 2003 par Chris Larson

Pourquoi avoir créer OpenEmbedded ?

Les systèmes de build existants ne gèrent pas les dépendances ou montrent vite leurs limites d'utilisation grande échelle par

- manque de flexibilité
- lourdeur de maintenance (ex: des makefiles)

OpenEmbedded a été crée en 2003 par Chris Larson

- pour fédérer les efforts de développement des différentes distro embarquées qui sont/étaient trop souvent spécifiques à **une seule** architecture cible.
- pour fournir un système de build flexible et puissant, utilisant une syntaxe et sémantique plus adaptée que des makefiles
- en s'inspirant de emerge et portage (voir `http://www.gentoo.org`)

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible
- plusieurs milliers de paquets disponibles: vim, python, qt4, etc.

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible
- plusieurs milliers de paquets disponibles: vim, python, qt4, etc.
- plusieurs images disponibles: console(2Mo), gpe-image, etc.

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible
- plusieurs milliers de paquets disponibles: vim, python, qt4, etc.
- plusieurs images disponibles: console(2Mo), gpe-image, etc.
- customisation: toolchain (glibc/uClibc/eglibc), scripts d'init, découpage des paquets, etc.

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible
- plusieurs milliers de paquets disponibles: vim, python, qt4, etc.
- plusieurs images disponibles: console(2Mo), gpe-image, etc.
- customisation: toolchain (glibc/uClibc/eglibc), scripts d'init, découpage des paquets, etc.
- kernel très récents supportés (2.6.24)

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible
- plusieurs milliers de paquets disponibles: vim, python, qt4, etc.
- plusieurs images disponibles: console(2Mo), gpe-image, etc.
- customisation: toolchain (glibc/uClibc/eglibc), scripts d'init, découpage des paquets, etc.
- kernel très récents supportés (2.6.24)
- syntax simple et puissante des .bb

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible
- plusieurs milliers de paquets disponibles: vim, python, qt4, etc.
- plusieurs images disponibles: console(2Mo), gpe-image, etc.
- customisation: toolchain (glibc/uClibc/eglibc), scripts d'init, découpage des paquets, etc.
- kernel très récents supportés (2.6.24)
- syntax simple et puissante des .bb
- pleins d'autres encore ...

Les atouts d'OpenEmbedded

- compilation de binaires/paquets/distributions complètes **ex nihilo** (from scratch)
- possibilité d'utiliser une toolchain pré-existante
- génération d'une même distribution déployable sur différente architecture cible
- plusieurs milliers de paquets disponibles: vim, python, qt4, etc.
- plusieurs images disponibles: console(2Mo), gpe-image, etc.
- customisation: toolchain (glibc/uClibc/eglibc), scripts d'init, découpage des paquets, etc.
- kernel très récents supportés (2.6.24)
- syntax simple et puissante des .bb
- pleins d'autres encore ...
- parce que c'est fun de créer sa propre mini distro ; −)

Qui utilise OpenEmbedded ?

- des laboratoires de recherches
- des sociétés spécialisées dans l'éléctronique embarqué
- des sociétés de télécom (téléphonie mobile / PDA)
- des projets libres de distributions Linux embarquée (ex: nslu2, OpenMoko)
- des particuliers (vous ?)

Quelques distributions basées sur OpenEmbedded

- Ångström: principale distribution supportée dans OE, basée sur Debian: `http://www.angstrom-distribution.org`
- Poky: `http://www.pokylinux.org`
- OpenMoko: `http://www.openmoko.org`

OpenEmbedded = bitbake + metadata

Techniquement, le projet OpenEmbedded se compose de

- bibake: **la** commande utilisée pour construire d'un simple paquet jusqu'à une distribution complète avec OpenEmbedded
- les metadata d'OpenEmbedded: l'ensemble des fichiers que bitbake exploite pour pouvoir faire cela

Exemple: builder vim

```
bitbake vim
```

Exemple: builder la distribution OpenMoko

```
bitbake openmoko-image
```

OpenEmbedded metadata repository

Le dépot des metadata contient l'ensemble des données nécessaires et suffisantes pour (cross)-compiler des composants logiciels open sources ex nihilo.
On distingue trois principaux types de metadata

1. **recipe** (*.bb): caractérise un composant logiciel (executable, bibliothèques, kernel, compiler, etc.)
2. **class** (*.bbclass): contient des tâches communes aux recipes
3. **conf** (*.conf): fichier de configuration

OpenEmbedded metadata: recipe (*.bb)

Un recipe contient l'ensemble des données permettant de compiler un composant depuis son code source. [3]

- description et licence
- liens vers les sources et patchs à appliquer
- dépendences (build dependencies, runtime dependencies)
- options de configuration
- customisation du découpage et contenu des paquets générés (-dev, -doc, etc.)
- ...

[3] on peut noter la similarité avec un .ebuild de Gentoo

OpenEmbedded metadata: recipe gnuchess

Exemple: packages/gnuchess/gnuchess_5.05.bb

```
DESCRIPTION = "Gnuchess is a chess playing engine."
HOMEPAGE = "http://www.gnu.org/software/chess/"
SECTION = "console"
PRIORITY = "optional"
LICENSE = "GPL"
SRC_URI = "$GNU_MIRROR/chess/$PN-$PV.tar.gz" [a]
S = "$WORKDIR/chess"
inherit autotools
```

[a] PN et PV (package name et version) déterminés par le nom du fichier .bb

Voir `documentation.conf` pour la doc des variables bitbake

OpenEmbedded metadata: class (*.bbclass)

Les classes permettent la génération à la volée de scripts shell executés pour builder un recipe. Leur usage est varié:

- build system tasks: autotools, scons, qmake, python, etc.
- packaging: découpage des paquets, support .deb .rpm .ipk
- SDK: pour packager des toolchains prêtes à emploi
- QA: assurance qualité (insane.bbclass, seppuku.bbclass)
- ...

Exemple: packages/gnuchess/gnuchess_5.05.bb

Gnuchess utilise les autotools. `inherit autotools` va inclure `autotools.bbclass` qui implémente les tâches `do_configure`, `do_compile`, `do_install` suivant `./configure`, `make` et `make install`

OpenEmbedded metadata: conf (*.conf)

Les fichiers de configuration

- `local.conf`: contient votre conf personnelle de build (TARGET_ARCH, DISTRO, ...)
- machine configurations: common architectures, routers, PDA, GSM phones, ...
- distribution policies: packaging, naming, preferred version of software, ...

bitbake

- un executeur de tâches et un gestionnaire de metadata

bitbake

- un executeur de tâches et un gestionnaire de metadata
- un peu le 'make' d'OpenEmbedded

bitbake

- un executeur de tâches et un gestionnaire de metadata
- un peu le 'make' d'OpenEmbedded
- insipiré de Portage, le gestionnaire de paquet de la distribution Gentoo

bitbake

- un executeur de tâches et un gestionnaire de metadata
- un peu le 'make' d'OpenEmbedded
- insipiré de Portage, le gestionnaire de paquet de la distribution Gentoo
- écrit en python

buts de bitbake

- gérer la cross-compilation

buts de bitbake

- gérer la cross-compilation
- gérer les dépendances inter-paquets
  - build time on target architectures
  - build time on native architectures
  - runtime

buts de bitbake

- gérer la cross-compilation
- gérer les dépendances inter-paquets
  - build time on target architectures
  - build time on native architectures
  - runtime
- linux distribution agnostic

buts de bitbake

- gérer la cross-compilation
- gérer les dépendances inter-paquets
  - build time on target architectures
  - build time on native architectures
  - runtime
- linux distribution agnostic
- architecture agnostic

buts de bitbake

- gérer la cross-compilation
- gérer les dépendances inter-paquets
  - build time on target architectures
  - build time on native architectures
  - runtime
- linux distribution agnostic
- architecture agnostic
- gérer les metadata conditionellement (target, OS, distro, machine)
- multiple target operating system (Linux, BSDs, etc.)

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`
2. **unpack** unpack data `do_unpack`

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`
2. **unpack** unpack data `do_unpack`
3. **patch** applies patches `do_patch`

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`
2. **unpack** unpack data `do_unpack`
3. **patch** applies patches `do_patch`
4. **configure** configures the sources `do_configure`

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`
2. **unpack** unpack data `do_unpack`
3. **patch** applies patches `do_patch`
4. **configure** configures the sources `do_configure`
5. **compile** compile the sources `do_compile`

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`
2. **unpack** unpack data `do_unpack`
3. **patch** applies patches `do_patch`
4. **configure** configures the sources `do_configure`
5. **compile** compile the sources `do_compile`
6. **stage** installs into the staging area `do_stage`

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`
2. **unpack** unpack data `do_unpack`
3. **patch** applies patches `do_patch`
4. **configure** configures the sources `do_configure`
5. **compile** compile the sources `do_compile`
6. **stage** installs into the staging area `do_stage`
7. **install** installs into the packaging area `do_install`

bitbake a recipe

Exemple: compiler foo

```
bitbake foo
```

Bitbake va effectuer dans l'ordre les tâches suivantes:

1. **fetch** downloads data from upstream `do_fetch`
2. **unpack** unpack data `do_unpack`
3. **patch** applies patches `do_patch`
4. **configure** configures the sources `do_configure`
5. **compile** compile the sources `do_compile`
6. **stage** installs into the staging area `do_stage`
7. **install** installs into the packaging area `do_install`
8. **package** creates packages `do_package`

bitbake a recipe

Fetch

- support de nombreux dépot: wget, svk, bzr, cvs, svn, git, perforce, hg

Configure

- paramètres par défaut + custom

Compile

- support de nombreux systèmes de build: autotools, scons, qmake, python distutils, etc.

Package

- découpage flexible des paquets à la Debian

bitbake en interne

Plus précisément, `bitbake foo` va consister à

1. parser les recipes pour lequels il est configuré à cherché
2. pour chaque recipe
   - la valeur de chaque variable est déterminée en fonction du `local.conf`, du recipe lui-même et des recipes/classes inclues
   - dépendances des tâches
3. un graph de tâches agglomérant les tâches de tous les recipes est généré
4. parcours du graph de tâches pour dresser la liste des tâches nécessaire pour builder foo
   - générer à la volée du script shell contenant les tâches de foo
   - executer le script shell
5. executer les tâches une à une chacune pour builder foo [4]

[4] les tâches dejà effectuées ne sont pas re-executée, sauf sur modification du recipe (ex: PR incrémenté)

OpenEmbedded configuration

Pour faire c'est premiers pas avec OpenEmbedded et bitbake:
`http://www.openembedded.org/wiki/GettingStarted`
Un seul fichier à configurer: `local.conf`

Exemple: local.conf pour un efika (PowerPC)

MACHINE = "efika"
DISTRO = "angstrom-2008.1"
MACHINE_KERNEL_VERSION = "2.6"
IMAGE_FSTYPES = "tar.gz squashfs"

Utilisation de bitbake

Exemple: build uclibc console image basée sur Ångström

```
bitbake console-image
```

Utilisation avancée: le bitbake shell

```
bitbake -i
```

Les paquets et images résultantes sont dans `tmp/deploy/`

Création d'un recipe 'hello world'

- syntaxe de bitbake détaillée dans un bitbake user manual: `http://bitbake.berlios.de/manual`
- voir aussi: `http://www.openembedded.org/user-manual`
- voir ce qui est fait dans les autres recipes

Tester les images/paquets générés

Booter l'image:

- Par émulation/virtualisation logicielle: qemu.
- Directement sur la machine cible: flasher la ROM.

Et après:

- installer des paquets: ipkg.
- débugger: gdb, strace, etc.

Ångström

Démo: boot d'Ångström depuis qemu

Contribuer à OpenEmbedded/Ångström

Pour résoudre les problèmes, un système de build ne suffit pas.
OpenEmbedded réuni (et à besoin d'autres):

- hackers kernel
- hackers de toolchain
- développeurs d'application
- développeurs de framework
- integrateur de systèmes

Participez aux développements d'OpenEmbedded !

- création/correction de recipes.
- documentation.
- tester Ångström/OpenMoko + rapport de bugs.
- patchs correctifs (cross-compilation, crashs, etc.)

A bientôt sur le channel IRC #oe sur freenode : −)