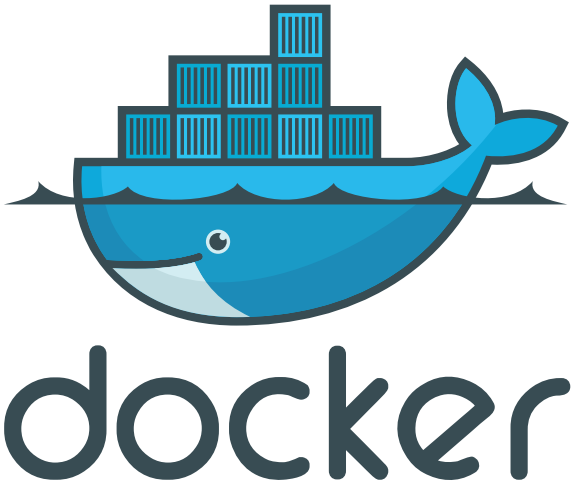

MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

État de l'art à la mi-projet de semestre Docker and embedded systems - Ou comment ne pas cross compiler Docker sur ARM

Auteur :
Gary Marigliano

Encadrant :
Jean-Roland Schüler

Contact :
gary.marigliano@master.hes-so.ch

Mandant :
Haute école d'ingénierie et d'architecture de Fribourg

Version 0.0.2
2 mai 2016

Historique

Version	Date	Auteur(s)	Modifications
0.0.1	15.04.16	Gary Marigliano	Création du document
0.0.2	01.05.16	Gary Marigliano	Modifications première page, ajout historique, repositionnement des images

Table des matières

1. Introduction

1.1 Contexte

Ce document s'inscrit dans le cadre du projet de semestre Docker and embedded systems actuellement réalisé par moi-même. Un des buts de ce projet est de cross compiler Docker à partir de ses sources pour produire un binaire exécutable sur un Odroid XU3 (ARMv7).

Lien : `https://github.com/krypty/docker_and_embedded_systems`

Il est important de noter que la vitesse de développement de Docker est assez hallucinante. En effet, sur Github (`https://github.com/docker/docker`) les commits se succèdent à vitesse grand V. Entre chaque version de Docker qui sortent environ tous les mois, il est courant d'avoir plus de 3000 commits qui ont été *pushés*. Tout ceci pour dire qu'à la lecture de ce document, il est quasiment sûr que certaines pistes explorées soient définitivement obsolètes ou au contraire deviennent la voie à suivre du à une mise à jour quelconque.

1.2 Objectifs

De manière plus précise, ce projet vise à maitriser les parties suivantes :

1. Construction d'un système Linux capable de faire tourner Docker et son *daemon* en utilisant Buildroot. Pour générer le dit système, on dispose d'un *repository* Gitlab hébergé à la Haute École de Fribourg
2. Cross compilation de Docker et de son *daemon*, capable de faire tourner des containers

L'objectif de ce document est d'énumérer les différentes techniques tentées pour (cross-)compiler Docker sur une cible ARM. De cette manière, le lecteur, en cas de reprise du projet ou par simple curiosité, aura une idée des pistes à explorer ou à éviter.

2. Objectif 1 - Construction d'un système GNU/Linux Docker-ready

Dans cette partie, on verra les ingrédients et pistes à suivre pour concevoir un système construit à partir de Buildroot capable de faire tourner Docker et son *daemon*.

2.1 Générer le système

Comme évoqué à la section 1.1, on dispose d'un Odroid XU3 sur lequel il faut générer un système GNU/Linux. Dans le cadre d'un cours, la Haute École de Fribourg met à disposition un *repository* git qui contient tout ce qu'il faut pour gérérer un tel système.

Toutes les ressources nécessaires à la génération du système se trouvent ici :

- Procédure de génération du système du cours CSEL et adresse du *repository* git : *p.02.2_mas_csel_environnement_linux_embarque_exercices.pdf*
- Script de génération de la carte utilisé : `https://github.com/krypty/docker_and_embedded_systems/blob/master/write_system_on_sd.sh` ou Appendix A
- Le PDF *01_IntroOdroidXu3.pdf* du cours SeS

Le système généré ne peut, dans sa configuration actuelle, permettre à Docker de se lancer. Pour pouvoir le faire, on a deux moyens à disposition : Buildroot et le kernel.

Grossièrement, Buildroot permet d'ajouter des packages et de configurer son système que le kernel permet d'ajouter des modules ou des drivers.

2.2 Vérifier que le système peut faire tourner Docker

Il faut en premier lieu mettre la main sur un binaire ARM Docker statiquement lié qui intègre le *daemon*. En effet, à l'heure actuelle, lorsqu'on compile Docker pour ARM de la manière officielle, le binaire résultant n'intègre pas le *daemon* mais uniquement le client (qui permet de se connecter à un *daemon* externe). Voir également à la section 3.1.4

Le seul binaire de ce type que j'ai trouvé actuellement est téléchargeable ici : `https://github.com/umiddelb/armhf/raw/master/bin/docker-1.9.1`.

Copiez ce fichier sur la cible et tentez de le lancer avec :

```
1 chmod +x docker-1.9.1
2 ./docker-1.9.1 deamon
```

S'il y a des erreurs c'est sûrement qu'il manque un ou plusieurs modules kernel. Pour vérifier que la configuration actuelle du noyau est correcte. L'équipe Docker met à disposition un script qui indique quels modules sont manquants.

Ce script est à télécharger ici : `https://github.com/docker/docker/blob/master/contrib/check-config.sh`

Voici un exemple de sortie où l'on voit qu'il manque certains modules :

```
/ # ./check-config.sh
info: reading kernel config from /proc/config.gz ...

Generally Necessary:
- cgroup hierarchy: nonexistent??
    (see https://github.com/tianon/cgroupfs-mount)
- CONFIG_NAMESPACES: enabled
- CONFIG_NET_NS: enabled
- CONFIG_PID_NS: enabled
- CONFIG_IPC_NS: enabled
- CONFIG_UTS_NS: enabled
- CONFIG_DEVPTS_MULTIPLE_INSTANCES: missing
- CONFIG_CGROUPS: enabled
- CONFIG_CGROUP_CPUACCT: enabled
- CONFIG_CGROUP_DEVICE: enabled
- CONFIG_CGROUP_FREEZER: enabled
- CONFIG_CGROUP_SCHED: missing
...
- CONFIG_NETFILTER_XT_MATCH_CONNTRACK: missing
- CONFIG_NF_NAT: missing
- CONFIG_NF_NAT_NEEDED: missing
- CONFIG_POSIX_MQUEUE: enabled

Optional Features:
- CONFIG_USER_NS: missing
- CONFIG_SECCOMP: enabled
- CONFIG_CGROUP_PIDS: missing
- CONFIG_MEMCG_KMEM: enabled
...
- CONFIG_EXT3_FS: enabled
- CONFIG_EXT3_FS_XATTR: missing
- CONFIG_EXT3_FS_POSIX_ACL: enabled
- CONFIG_EXT3_FS_SECURITY: enabled
    (enable these ext3 configs if you are using ext3 as backing filesystem)
- CONFIG_EXT4_FS: enabled
- CONFIG_EXT4_FS_POSIX_ACL: enabled
- CONFIG_EXT4_FS_SECURITY: enabled
- Storage Drivers:
  - "aufs":
    - CONFIG_AUFS_FS: missing
  - "btrfs":
    - CONFIG_BTRFS_FS: enabled (as module)
  - "devicemapper":
    - CONFIG_BLK_DEV_DM: enabled
    - CONFIG_DM_THIN_PROVISIONING: missing
  - "overlay":
    - CONFIG_OVERLAY_FS: enabled (as module)
  - "zfs":
    - /dev/zfs: missing
    - zfs command: missing
    - zpool command: missing
```

La suite consiste à modifier le kernel pour y ajouter les modules manquants, de reflasher le système et tester à nouveau si Docker se lance.

Je n'explique volontairement pas comment modifier la configuration d'un kernel Linux, car d'une part cette information se trouve facilement sur Internet ou sur les documents indiqués plus haut et d'autre une car ce n'est pas le but de ce rapport.

3. Objectif 2 - Techniques de compilation essayées

3.1 La manière officielle

C'est la manière recommandée et qui, un jour, sera celle qu'il faudra employer. Mais aujourd'hui, elle ne permet que de cross compiler un binaire ARM Docker qui n'embarque pas le *deamon*.

3.1.1 Principe utilisé

Pour compiler Docker de la manière officiellement supportée, on doit utiliser Docker. En effet, le Makefile fourni va lancer un container Docker qui va contenir un système d'exploitation ainsi que tous les pré-requis et dépendances puis lancer la compilation de Docker à l'intérieur de ce container.

3.1.2 Cheminement général

Sur une machine GNU/Linux

```
1 git clone https://github.com/docker/docker
2 cd docker
3 git checkout v1.10.3 -b tmp_build # vous pouvez remplacez v1.10.3 par la
  ↪ dernière version (tag) stable
4 make build
5 make binary # pour générer le binaire sur la plateforme sur laquelle on est
  ↪ en train de compiler (probablement x64)
6 make cross # pour générer le binaire ARM
```

Le binaire se trouve dans le dossier ./bundle. On se place volontairement sur un tag de release pour avoir un minimum de stabilité dans les fonctionnalités embarquées.

3.1.3 Schéma

Comme on peut le voir à la figure 3.1, pour compiler Docker, il faut disposer de Docker sur son PC. En faisant une commande `make`, Docker va créer un container basé sur une image Ubuntu et va installer tous les outils de compilation nécessaires. Une fois que cela est fait, Docker utilise ce container pour lancer la compilation. Le binaire est ensuite récupéré dans le dossier `./bundle`. Il ne reste plus qu'à copier le binaire sur la cible.

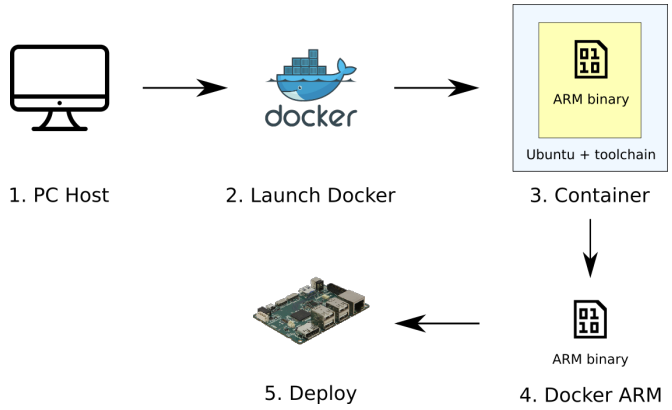

FIGURE 3.1 – Docker in Docker

3.1.4 Limitations

Actuellement, il est possible de générer un binaire Docker x64 et ARM mais seule l'architecture x64 intègre le *deamon* nécessaire à la création de containers.

Le binaire ARM est dit `CLIENT_ONLY` dans le sens où il peut être le client d'un *deamon* Docker remote (instancié sur une autre machine).

3.2 Compiler directement sur une machine ARM en utilisant la manière officielle

Même principe que la technique précédente à la différence que le PC Host est remplacé par la cible, une carte ARM. On va donc avoir besoin d'une carte qui propose une distribution GNU/Linux qui intègre Docker. Dans le cas de l'Odroid XU3, il existe Archlinux ARM [1].

3.2.1 Principe utilisé

Voir section 3.1.

Remarque : dans ce cas, seul `make binary` est nécessaire car on n'est pas obligé de cross compiler pour d'autres plateformes.

3.2.2 Cheminement général

Voir section 3.1.

3.2.3 Schéma

Voir section 3.1.

3.2.4 Limitations

Lors de mes tests, j'ai tenté à plusieurs reprises de compiler Docker en utilisant mon Odroid C1 personnel mais je ne suis jamais arrivé à finir la compilation car elle plantait au bout d'une bonne heure.

Évidemment, le temps de compilation est énormément long et on est limité par la machine avec laquelle on compile. Une autre possibilité pour l'avenir serait de passer par un prestataire Cloud qui fournit des machines ARM. Scaleway semble proposer ce genre d'offre [2]

1. Archlinux ARM : `https://archlinuxarm.org/`
2. Scaleway : `https://www.scaleway.com/`

3.3 Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et une image Debian

3.3.1 Principe utilisé

Cette technique (et les suivantes basées sur celle-ci) est différente. Ici, on utilise une machine virtuelle QEMU [3] pour compiler Docker. On ne fait donc plus de cross compilation mais de l'émulation. On verra dans au chapitre 4 que l'on peut utiliser QEMU pour exécuter des containers ARM sur une machine x64.

3.3.2 Cheminement général

Il faut d'abord installer QEMU sur une machine. J'utilise Archlinux mais cela devrait être sensiblement la même chose pour d'autres distributions.

```
1 yaourt -S qemu qemu-arch-extra
```

Pour la suite, j'ai suivi les tutoriels suivants :

1. A QEMU image for debian armel, `http://www.n0nb.us/blog/2012/03/a-qemu-image-for-debian-armel/`.
2. Debian Wheezy armhf images for QEMU, `https://people.debian.org/~aurel32/qemu/armhf/`

Je vais pas m'étendre sur cette solution, car Debian ARM ne propose pas de package Docker et cette VM Debian ne permet pas dans sa configuration actuelle d'exécuter le binaire ARM proposé à la section 2.2.

3.3.3 Schéma

Comme on peut le voir à la figure 3.2, on reprend le même principe qu'expliqué à la section : La manière officielle.

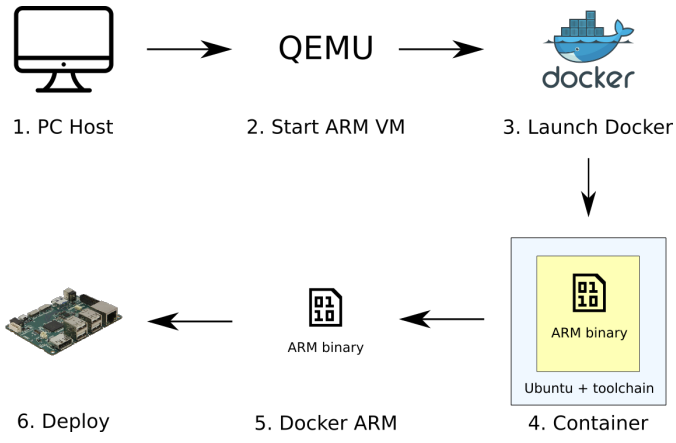

FIGURE 3.2 – Docker QEMU

La principale différence vient du fait qu'on encapsule le processus dans une machine virtuelle QEMU afin d'éviter d'utiliser une carte ARM. Une fois le système émulé, on lance Docker pour compiler Docker.

3. QEMU, émuler une machine complète : `http://wiki.qemu.org/Main_Page`

3.3.4 Limitations

Cette solution n'est pas une solution à suivre. C'est, à mon avis, une bonne idée mais inutilisable comme présenté ici. Par contre, cela peut devenir intéressant avec d'autres distributions qui intègre un package Docker et dont le kernel est correctement configuré. Par exemple, Raspbian (Une Debian modifiée pour les Raspberry Pi) ou Archlinux ARM. Les solutions suivantes reprennent cette hypothèse.

Autre point négatif, cette solution (et les dérivées utilisant QEMU) nécessite de disposer d'un binaire ARM Docker, celui de la section 2.2. Par conséquent, comme on ne sait pas comment produire ce binaire, on ne maîtrise pas l'ensemble de la pipeline de compilation.

3.4 Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et une image Raspbian

3.4.1 Principe utilisé

Même principe que la technique précédente à la différence près que l'image de la VM n'est plus une Debian mais une Raspbian[4], utilisée par les Raspberry Pi.

3.4.2 Cheminement général

Les étapes sont relativement les mêmes que pour la technique précédente. C'est principalement les fichiers à télécharger qui diffèrent.

Voici les tutoriels que j'ai suivis :

1. Émuler le Raspberry Pi sous Debian avec Qemu, `http://www.jdhp.org/hevea/tutoriel_rpi_qemu/tutoriel_rpi_qemu.html`
2. Émuler une Raspberry Pi sous Linux avec Qemu, `https://assos.centrale-marseille.fr/clubrobot/content/%C3%A9muler-une-raspberry-pi-sous-linux-avec-qemu`

3.4.3 Schéma

Similaire à la technique précédente.

3.4.4 Limitations

Similaires à la technique précédente.

Conclusion : aucune amélioration par rapport à la technique précédente. Impossible de compiler Docker.

4. Raspbian : `https://www.raspbian.org`

3.5 Compiler en émulant une machine ARM sur un PC de bureau avec QEMU et chroot

3.5.1 Principe utilisé

On va utiliser la commande `chroot` [5] afin d'isoler une machine QEMU dans un répertoire de notre machine de bureau.

3.5.2 Cheminement général

Le tutoriel utilisé est le suivant : `https://github.com/RoEdAl/linux-raspberrypi-wsp/wiki/Building-ArchLinux-ARM-packages-ona-a-PC-using-QEMU-Chroot`. Une copie de ces commandes se trouvent à l'Appendix B.

3.5.3 Schéma

Comme on peut le voir à la figure 3.3, la première couche représente le dossier *root* de notre machine et le niveau suivant, les dossiers qu'il contient. Dans un dossier (à créer) appelé *docker_ qemu*, on retrouve les fichiers utilisés par le tutoriel évoqué à la section précédente.

Dans le répertoire *archlinux_ rpi2*, on retrouve le système Archlinux ARM extrait dans lequel on vient copier `qemu-arm-static` permettant d'émuler le système ARM.

En clair, tous les dossiers en dessus de *archlinux_ rpi2* appartiennent à l'OS hôte et tous les fichiers en dessous, appartiennent au système invité, Archlinux ARM.

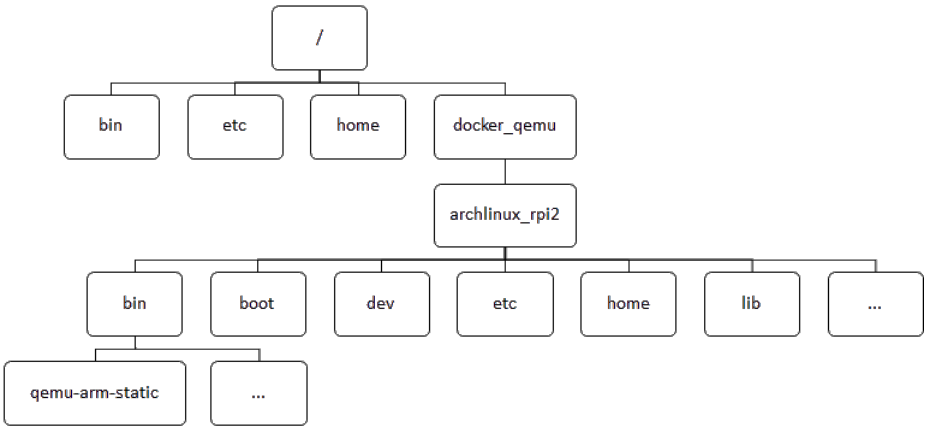

FIGURE 3.3 – Docker QEMU

3.5.4 Limitations

Bien que je sois parvenu à booter sur la machine virtuelle, je n'ai pas pu lancer la compilation de Docker car cette procédure à "détecté" que je me trouvais en `chroot` et s'est arrêtée.

5. chroot : `https://fr.wikipedia.org/wiki/Chroot`

3.6 Compiler Docker sans Docker

Avec cette technique, on essaie de court-circuiter la compilation officielle (Docker in Docker) en compilant Docker sans Docker. Pour ce faire, il faut veiller à avoir une machine qui puisse compiler Go, car Docker est écrit dans ce langage.

Attention : J'ai d'abord essayé de compiler Docker pour une machine x64. Cette technique était avant tout exploratoire et ne permet donc pas de compiler un binaire ARM.

3.6.1 Principe utilisé

On compile le programme de manière "classique" sans passer par un container Docker.

3.6.2 Cheminement général

Installation de Go sur Archlinux :

```
1  yaourt -S go
```

Ensuite, on effectue les commandes suivantes :

```
1  git clone https://github.com/docker/docker docker_tmp
2  export AUTO_GOPATH=1
3  cd docker_tmp
4  git checkout v1.10.3 -b gary
5
6  # parmis les pré-requis, il manquait btrfs-progs
7  yaourt btrfs-progs
8
9  # on ignore devicemapper, car il y avait des erreurs.
10 export DOCKER_BUILDTAGS='exclude_graphdriver_devicemapper'
11 ./hack/make.sh binary
12 # le binaire se trouve dans ./bundles/1.10.3/binary
```

3.6.3 Schéma

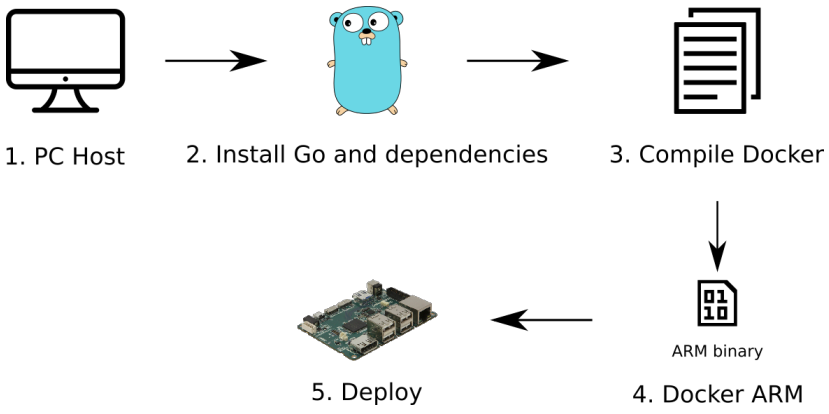

FIGURE 3.4 – Docker sans Docker

La figure 3.4 montre la différence par rapport à la manière officielle. En effet, Docker n'est plus utilisé mais uniquement ses sources.

3.6.4 Limitations

Cette technique n'est pas une bonne idée pour les raisons suivantes :

— Il faut installer et configurer soit même les outils de compilations alors qu'avec la manière officielle, c'est automatiquement fait dans un container.
— Lors de mes tests, je n'ai pas essayé de cross compiler vers ARM. C'est possible mais je ne l'ai pas testé.
— Cette solution n'apporte rien de plus par rapport à la manière officielle. C'est-à-dire qu'elle produit (ou produirait dans le cas d'une cross compilation ARM) le même binaire qu'avec la manière officielle. Donc, le *deamon* ne serait de toute façon pas disponible.

Néanmoins, cette solution m'a permis de comprendre qu'il fallait peut-être modifier le code source de Docker afin d'autoriser l'intégration du *deamon* tout en conservant la technique "Docker in Docker" afin de se faciliter la vie pour installer les outils de compilation.

3.7 Conclusion sur les techniques de compilation

Comme on a pu le voir, plusieurs techniques et tentatives ont été essayées. Aucune n'a permis de compiler ou de cross compiler Docker pour obtenir un binaire ARM statiquement lié.

Néanmoins, on sait que c'est possible car d'une part des distributions comme Archlinux ARM intègrent un package Docker incluant le *deamon* et d'une autre car on a pu mettre la main sur un binaire isolé, voir section 2.2.

Encore une fois, il n'est pas impossible que l'équipe Docker supporte officiellement le *deamon* pour la plateforme ARM d'ici quelques temps et ce document sera obsolète.

Pour la suite du projet, j'ai installé Archlinux ARM sur l'Odroid XU3 et vais continuer avec une partie concernant la sécurité de Docker et de ces containers.

4. Exécuter des containers ARM sur une machine x64

Dans ce chapitre, on va voir comment exécuter des containers basés sur des images ARM sur une machine x64. Ceci permet plusieurs choses intéressantes comme :

- Tester une image sur son poste avant de la déployer sur une cible
- Pouvoir utiliser des images prévues pour ARM sur une machine x64, voire d'en effectuer le portage
- Utiliser le container ARM comme outil de cross compilation
- Pouvoir développer une application ARM et la tester sur le même PC sans devoir la déployer sur la cible à chaque compilation
- ...

Lien utile : `http://blog.hypriot.com/post/close-encounters-of-the-third-kind/`

On va reprendre l'exemple du blog de Hypriot et lancer un container httpd ARM (un serveur web et une page HTML toute simple).

Il faut d'abord installer QEMU. Pour ce faire, voir section 3.3.2.

Ensuite, il faut créer un Dockerfile qui se base sur l'image ARM (ici `hypriot/rpi-busybox-httpd`) à lancer et une archive contenant `qemu-static-arm`.

Création d'un dossier de travail :

```
mkdir hypriot-qemu
cd hypriot-qemu
touch Dockerfile
```

Dockerfile à créer sur une machine x64 :

```
FROM hypriot/rpi-busybox-httpd
ADD qemu-arm-static.tar /
```

Pour l'archive, il faut effectuer les manipulations suivantes :

```
sudo cp /usr/bin/qemu-arm-static .
sudo chown gary:gary qemu-arm-static
mkdir usr
tar -cvf qemu-arm-static.tar usr
mkdir usr/bin
tar -uvf qemu-arm-static.tar usr/bin
mv qemu-arm-static usr/bin
tar -uvf qemu-arm-static.tar usr/bin/qemu-arm-static
```

Si tout s'est bien passé, vous devriez obtenir une sortie similaire :

```
tar vtf qemu-arm-static.tar
drwxrwxr-x gary/gary         0 2016-04-15 21:35 usr/
drwxrwxr-x gary/gary         0 2016-04-15 21:35 usr/bin/
-rwxr-xr-x gary/gary   2936324 2016-04-15 21:26 usr/bin/qemu-arm-static
```

Il ne reste plus qu'à lancer le container :

```
docker build -t rpi-busybox-httpd .
```

```
2 docker run -d -p 80:80 rpi-busybox-httpd # essayez un autre port s'il est
  ↪ déjà pris
```

Rendez-vous ensuite sur `http://localhost:80` pour observer la page web.

Appendices

A. Script : Write system on SD

```
#!/bin/bash

#Warning: please replace sdc by the location of your sd card
# Please review this script before use it. It can harm your system :-)

sudo dd if=/dev/zero of=/dev/sdc bs=4k count=32768
sudo parted /dev/sdc mklabel msdos
sudo parted /dev/sdc mkpart primary ext4 131072s 2228223s
sudo parted /dev/sdc mkpart primary ext4 2228224s 4325375s
sudo mkfs.ext4 /dev/sdc2 -L usrfs
#sudo mkfs.ext4 /dev/sdc1 -L rootfs
sync
sudo dd if=/tftpboot/xu3-bl1.bin of=/dev/sdc bs=512 seek=1
sudo dd if=/tftpboot/xu3-bl2.bin of=/dev/sdc bs=512 seek=31
sudo dd if=/tftpboot/xu3-tzsw.bin of=/dev/sdc bs=512 seek=2111
sudo dd if=/tftpboot/xu3-u-boot.bin of=/dev/sdc bs=512 seek=63
sudo dd if=/tftpboot/xu3-uImage of=/dev/sdc bs=512 seek=6304
sudo dd if=/tftpboot/exynos5422-odroidxu3.dtb of=/dev/sdc bs=512 seek=22688
sudo dd if=/tftpboot/xu3-rootfs.ext4 of=/dev/sdc1
sync
sudo resize2fs /dev/sdc1
```

B. Building ArchLinux ARM packages on a PC using QEMU Chroot

Source : `https://github.com/RoEdAl/linux-raspberrypi-wsp/wiki/Building-ArchLinux-ARM-packages-ona-a-PC-using-QEMU-Chroot`

On a PC witch ArchLinux or Manjaro installed :

Build and install binfmt-support package. Build and install qemu-user-static package. Install arch-install-scripts package.

Download and extract the root filesystem for Raspberry Pi or Raspberry Pi 2 :

```
wget http://archlinuxarm.org/os/ArchLinuxARM-rpi-latest.tar.gz
bsdtar -xpf ArchLinuxARM-rpi-latest.tar.gz -C archlinux-rpi
```

or

```
wget http://archlinuxarm.org/os/ArchLinuxARM-rpi-2-latest.tar.gz
bsdtar -xpf ArchLinuxARM-rpi-2-latest.tar.gz -C archlinux-rpi2
```

Enable ARM to x86 translation :

```
update-binfmts --enable qemu-arm
```

Copy the **QEMU** executable :

```
cp /usr/bin/qemu-arm-static archlinux-rpi/usr/bin
```

Chroot :

```
arch-chroot archlinux-rpi /bin/bash
```

Install base-devel and distcc packages :

```
pacman -Suy base-devel distcc
```

Exit and chroot again as normal alarm user :

```
exit
arch-chroot archlinux-rpi runuser -l alarm
```

Build your package.