Linux embarqué avec Yocto

Créer un système Linux embarqué

Christophe BLAESS

christophe.blaess@logilin.fr

https://www.blaess.fr/christophe/

Ingénierie et formations sur Linux et les logiciels libres
https://www.logilin.fr

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap
- *Excalidraw* (en ligne) pour certains schémas

Linux embarqué avec Yocto

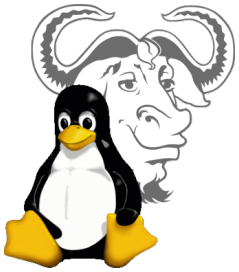
ILY v. 7.6

https://www.blaess.fr/christophe/

https://www.logilin.fr

Environnement Linux embarqué

Linux



- Noyau du système d'exploitation GNU/Linux compatible Unix,
- environnement constitué de logiciels libres,
- développement essentiellement bénévole.

Linux n'a jamais été prévu pour être un système industriel, temps-réel ou embarqué !

Système embarqué (embedded system)

« to embed (*embedded*, *embedding*) : (v. transitif) : enfoncer, sceller, incruster. »
(*Dictionnaire bilingue Larousse*).

- Système informatique souvent non-perçu comme tel,
- autonome, robuste, résilient,
- intégré comme élément d'un système plus large.

Source logo « Gnu/Linux » : https://commons.wikimedia.org/wiki/File:Gnulinux.png
Manchot « Tux » créé par Larry Ewing, mascotte « GNU Head » créée par Aurelio A. Eckert.

Influences de Linux :

1969 – Ken Thompson & Denis Ritchie : Unix AT&T

1978 – Université de Berkeley : Unix BSD

1984 – Richard Stallman : Projet Gnu (www.gnu.org)

1987 – Andrew Tanenbaum : Minix

Création de Linux :

1991 – Linus Torvalds : Linux 0.0.1

Voir aussi

- [TANENBAUM 2006] : Andrew Tanenbaum & Albert Woodhull – *Operating Systems, Design and Implementation* – Third edition – Pearson, 2006.
- [RAYMOND 2003] : Eric S. Raymond – *The Art of Unix Programming* – Addison-Wesley, 2003.
- [TORVALDS 2002] : Linus Torvalds & David Diamond – *Just for Fun : The Story of an Accidental Revolutionary* – Harper Business, 2002

Spécificités de l'environnement embarqué

Contraintes matérielles

Performance : puissance CPU, capacité mémoire, stockage, richesses I/O...

Encombrement : ventilateur / dissipateur, batterie, connecteurs...

Autonomie : batterie, panneau solaire, alimentation externe...

Contraintes logicielles

Performance : optimiser les ressources matérielles, temps réel...

Robustesse : fonctionnement 7/7 & 24/24, environnement hostile...

Sécurité : détection d'intrusion ou de modification, mises à jour, signatures...

Contraintes économiques

Concurrence : qualité, IHM, ergonomie...

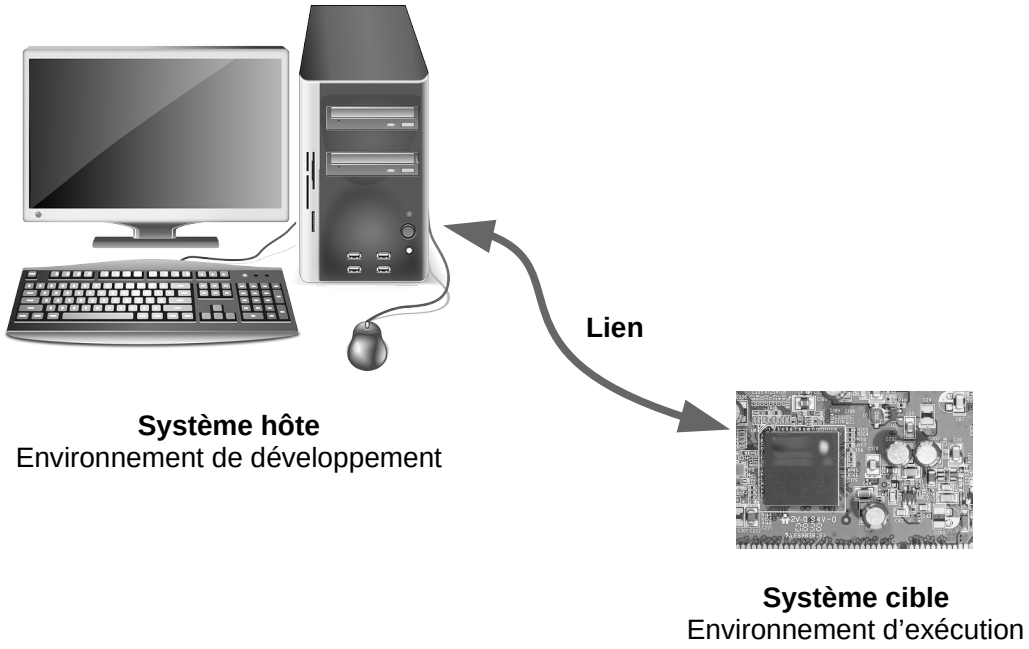
Coûts : choix des composants, licences logicielles...

Présence d'un « *market* » applicatif ouvert...

Voir aussi

- [FICHEUX 2017] : Pierre Ficheux – *Linux embarqué : Mise en place et développement* – Eyrolles, 2017.
- [BLANC 2013] : Gilles Blanc – *Linux embarqué : Comprendre, développer, réussir* – Pearson, 2013.

Développement applicatif embarqué, *cross-compilation*



Sources graphiques : images 158675 et 3262915 sur Pixabay.

Système hôte :

- Linux : une distribution classique intégrant Eclipse et une chaîne de cross-compilation Gnu,
- Windows : avec les outils Gnu provenant des projets Cygwin, Mingw ou WSL.

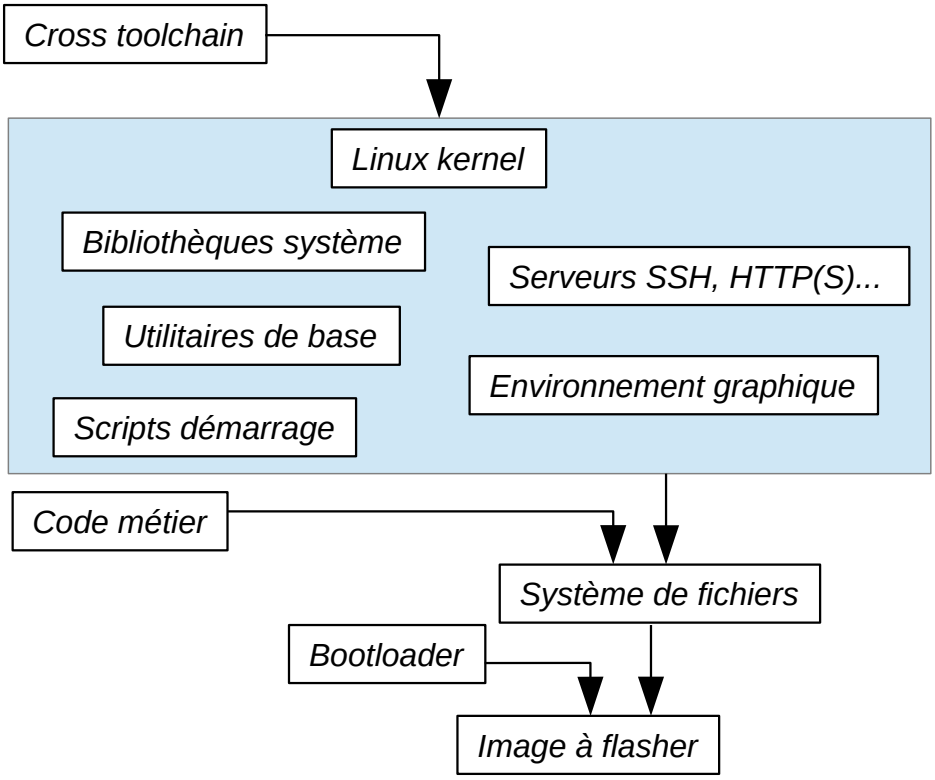
Système cible :

- Linux sur processeurs x86 (32/64 bits), Arm (32/64 bits), PowerPC, Motorola 68000, RiscV, Mips, etc.,
- systèmes non-Linux : microcontrôleurs, RTEMS, FreeRTOS...

Lien :

- Série RS/232 : terminal texte, transfert de fichiers,
- Ethernet et pile IP : *ssh*, *scp*, montage par *NFS*,
- Interface JTag : flashage direct et débogage au niveau du processeur...

Composants d'un système embarqué



La création manuelle des différents composants d'un système embarqué est de plus en plus rare (développement très spécifique et minimaliste, modification en profondeur des composants de base, aspect pédagogique, etc.)

On dispose aujourd'hui de plusieurs outils de génération de plates-formes Linux embarqué.

Voir aussi

[BLAESS 2012] Christophe Blaess – *Raspberry Pi from Scratch* – GNU/Linux Magazine France 155 & 158 – Diamond Editions – Disponible sur https://www.blaess.fr/christophe/articles/files-glmf/.

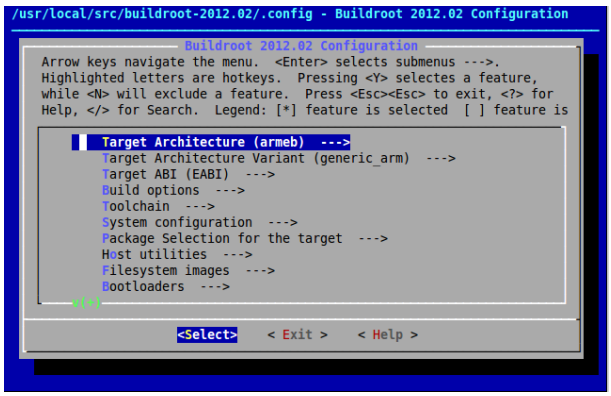
[FICHEUX 2013] Pierre Ficheux – « *Distributions embarquées pour Raspberry Pi* » – Open Silicium n° 7 – Diamond Editions.

Build systems

Buildroot

Buildroot est un ensemble de *Makefiles*, de scripts et de fichiers de configuration permettant la construction complète d'un système Linux pour une cible embarquée.

Il télécharge automatiquement les paquetages nécessaires pour la compilation et l'installation.



La configuration de Buildroot est réalisée à travers une interface assez simple d'utilisation.

Buildroot s'appuie intensivement sur l'utilitaire `make` comme moteur de compilation.

Buildroot permet de construire une image complète « prête à flasher » comprenant tout l'environnement d'exécution (noyau, bibliothèques, utilitaires, applications, système graphique, etc.).

Avantages :

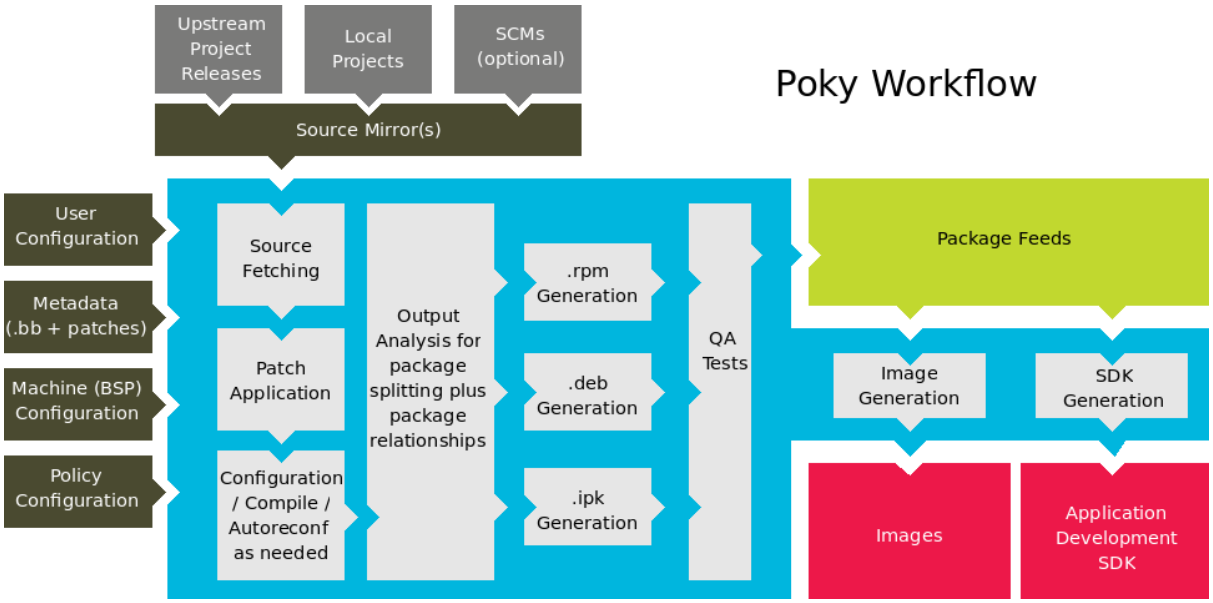
- Système relativement simple à comprendre et à modifier
- Possibilité d'écrire facilement des recettes pour ajouter des composants système.
- Intégration aisée du code métier.

Inconvénients :

- Pas de gestion de packages, génération d'une image contenant tout le système.
- Difficile de gérer des versions différentes (gamme de produits) dans la même arborescence.

Yocto Project

Outil complet pour créer une plate-forme embarquée et son environnement de développement d'applications-métier.



Source : http://www.yoctoproject.org

Avantages :

- Complet, riche, supporté par la majorité des industriels dans le domaine de Linux embarqué.
- Système de *packages* avec dépendances dans le code généré pour la cible.
- Possibilités de configuration et d'adaptation très complètes.

Inconvénients :

- Système assez complexe à maîtriser.
- Durées de compilation longues en raison des interdépendances entre packages.

Production d'une image de Yocto Project

Terminologie

Poky est la distribution de référence de Yocto Project. Elle regroupe l'outil de compilation **`bitbake`**, des scripts d'aide (`devtool`, `recipetool`, etc.) et des recettes provenant du projet ***Open Embedded Core.*** Deux versions de Poky sont publiées annuellement :

Nom	Version	Date
Dunfell	**3.1**	**2020-04**
Gatesgarth	3.2	2020-11
…	…	…
Kirkstone	**4.0**	**2022-04**
Langdale	4.1	2022-10
Mickledore	4.2	2023-04
Nanbield	4.3	2023-11
Scarthgap	5.0	2024-04

Poky contient plusieurs ***layers***. Un ***layer*** est un répertoire (préfixé par `meta-`) contenant des recettes (*recipes*) regroupées par fonctionnalités connexes.

Une **recette** (fichier `.bb`) décrit comment produire un *package* en téléchargeant les sources, appliquant éventuellement des *patches*, configurant, compilant et installant les exécutables.

Voir aussi

[SALVADOR 2014] : Otavio Salvador, Daiane Angolini – *Embedded Linux Development with Yocto Project* – Packt Publishing, 2014.

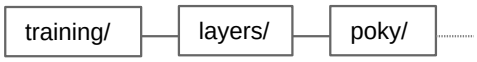
`http://www.yoctoproject.org/docs` : site de référence de Yocto Project.

Travaux pratiques : préparation de l'environnement de Yocto

```
[~]$ mkdir  training
[~]$ mkdir  training/layers
[~]$ cd  training/layers
```

Récupérer les sources de Poky en prenant une branche stable :

```
[layers]$ git clone  git://git.yoctoproject.org/poky  -b kirkstone
```



Prendre la dernière version stable :

```
[layers]$ cd poky
[poky]$ git tag
  ...
  yocto-4.0.17
[poky]$ git checkout yocto-4.0.17
[poky]$ ls
bitbake/              LICENSE.MIT    meta-skeleton/     README.poky
contrib/              MEMORIAM       meta-yocto-bsp/    README.qemu
documentation/        meta/          oe-init-build-env  scripts/
LICENSE               meta-poky/     README.hardware
LICENSE.GPL-2.0-only  meta-selftest/ README.OE-Core
[poky]$ cd ../..
```

On ne doit jamais modifier le contenu de l'arborescence de Poky !

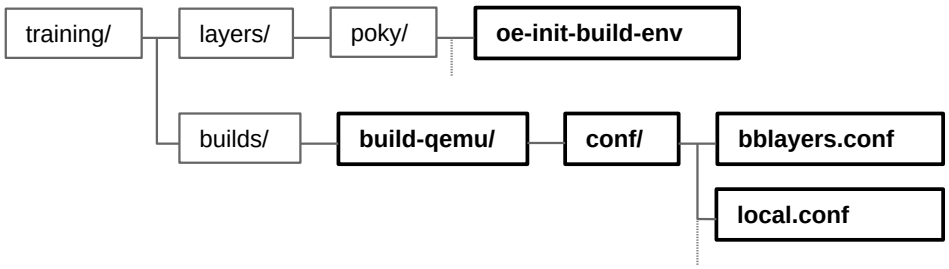
Configuration de Yocto Project

Il faut « sourcer » le script **`oe-init-build-env`** car il remplit des variables d'environnement et nous déplace dans l'arborescence.

```
[training]$ mkdir  builds
[training]$ source  layers/poky/oe-init-build-env  builds/build-qemu
```

Cette opération est à répéter à chaque début de session de travail. Le script crée initialement un répertoire de compilation (par défaut `build`). On peut préciser un chemin absolu.

```
[build-qemu]$ pwd
/home/USER/training/builds/build-qemu

[build-qemu]$ ls
conf/
[build-qemu]$ ls  conf/
bblayers.conf  local.conf  templateconf.cfg
```



On prépare la configuration du système en éditant `local.conf` et `bblayers.conf`.

Après compilation d'une image, on trouve les fichiers résultants dans un nouveau sous-répertoire `tmp/` (ce chemin peut être modifié dans le fichier `local.conf`).

Certains shells n'acceptent pas la commande « source », il faut la remplacer par un simple point :

```
[training]$  .  layers/poky/oe-init-build-env  <build_directory>
```

conf/local.conf

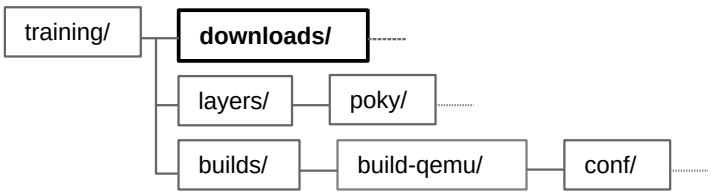
Ce fichier spécifie les paramètres pour la compilation. Les principales variables sont :

`MACHINE = "<nom de la cible>"`

Par défaut, Yocto Project prépare une image pour un émulateur de PC x86.

`DL_DIR = "${TOPDIR}/../../downloads"`

*Durant la compilation les packages nécessaires sont téléchargés et stockés dans le répertoire **`DL_DIR`**. Si on envisage plusieurs compilations successives il est intéressant de sortir ce répertoire de celui de compilation et de le conserver à part.*



Les produits de compilation intermédiaires sont stockés dans le répertoire indiqué par la variable **`SSTATE_DIR`** (*Shared State Files*). On peut les conserver d'une compilation à l'autre pour réduire le temps de production.

Basiquement Yocto Project ne propose qu'un support pour un nombre très réduit de cibles (*Qemu*, *Beaglebone*, *x86 32/64 générique*, *MPC8315e*). Il faut souvent ajouter un *layer* spécifique pour la cible désirée.

Pour trouver ce *layer*, cherchez sur :

https://layers.openembedded.org/layerindex/branch/master/machines/

ou dans le support du fournisseur de matériel.

Autoriser une connexion sans mot de passe

Nous verrons dans le prochain chapitre comment ajouter des utilisateurs et fixer les mots de passe.

Dans un premier temps nous souhaitons pouvoir se connecter avec l'identité `root` sans saisir de mot de passe.

Pour cela il faut ajouter la ligne

```
IMAGE_FEATURES += "empty-root-password"
```

dans le fichier `local.conf`.

Production d'une image

L'outil Bitbake est un moteur de compilation comme la commande `make` mais acceptant des recettes plus avancées écrites en Python et en shell.

Pour lancer la compilation (durée : environ deux heures) :

```
[build-qemu]$ bitbake  <nom de l'image>
```

Il existe de nombreuses images incluses directement dans Poky. Les principales sont :

- `core-image-base` : un shell, des outils classiques en ligne de commande et un serveur SSH pour une connexion distante.
- `core-image-minimal` : un système minimal en ligne de commande.
- `core-image-x11` : un environnement graphique X11 complet.
- `core-image-sato` : un environnement graphique orienté pour les applications mobiles.

Les recettes décrivant les images se trouvent dans `poky/meta/recipes-*/images/`

On peut demander à bitbake de ne faire que les téléchargements avec :

```
[build-qemu]$ bitbake <nom de l'image> --runall=fetch
```

Si les outils installés sur la machine hôte sont trop anciens, `bitbake` affiche un message du genre :

```
make version 4.2.1 is known to have issues on Centos/OpenSUSE and other
non-Ubuntu systems...
```

dans ce cas, installer des outils précompilés :

```
[build-qemu]$ ../../layers/poky/scripts/install-buildtools
```

et avant chaque session de travail exécuter le script

```
[build-qemu]$ source ../../layers/poky/buildtools/environment-setup-x86_64-poky-sdk-linux
```

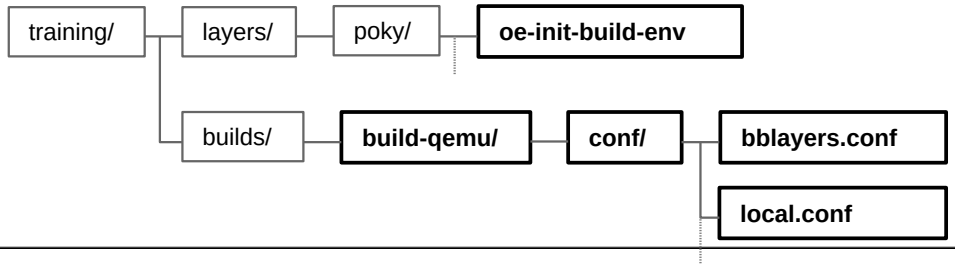
Travaux pratiques : produire une image pour émulateur Arm

Le projet **Qemu** fournit des émulateurs pour différentes architectures (x86 32/64 bits, Arm 32/64 bits, PowerPC, MIPS, Sparc...).

Il est disponible librement dans les distributions Linux classiques.

Yocto Project supporte la production d'images pour Qemu et même le lancement de l'émulateur avec les paramètres nécessaires.

1. Créez un environnement compilation nommé **`build-qemu`**.
2. Éditez `conf/local.conf` pour choisir la machine **`qemuarm`**
3. Lancez la production d'une image « **`core-image-base`** »



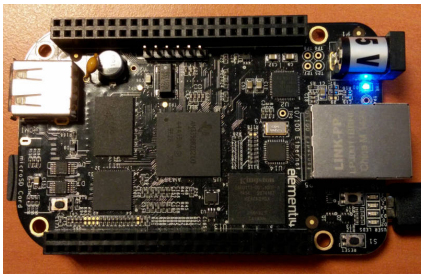
Voir aussi :

https://www.qemu.org : le site officiel du projet Qemu.

Solution :

```
[training]$ source  layers/poky/oe-init-build-env  builds/build-qemu

[build-qemu]$ nano  conf/local.conf
            MACHINE = "qemuarm"
            DL_DIR = "${TOPDIR}/../../downloads"
            IMAGE_FEATURES += "empty-root-password"

[build-qemu]$ bitbake  core-image-base
```

Travaux pratiques : produire une image pour Beaglebone Black

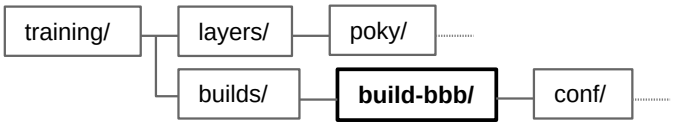


La carte ***Beaglebone Black*** est un ordinateur monocarte produit par *Texas Instruments*, *Digikey* et *Element 14* :

- processeur Sitara AM335x cadencé à 1GHz
- 512 Mo de ram
- 2Go de mémoire flash eMMC
- Ethernet, USB, UART, GPIO, SPI, CAN, i²c...

Yocto Project supporte directement la production d'images pour la famille des cartes *Beagleboard* et *Beaglebone*.

1. Créez environnement de compilation nommé **`build-bbb`**.
2. Éditez `conf/local.conf` et indiquez **`beaglebone-yocto`** dans la variable **`MACHINE`**
3. Lancez la compilation d'une image « **`core-image-base`** ».



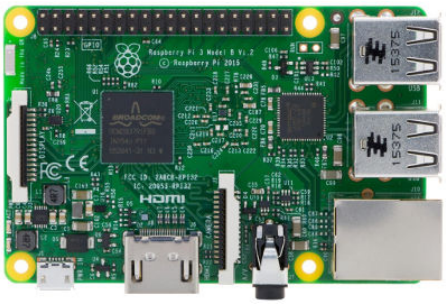
Voir aussi

[SADIQ 2015] « *Using Yocto Project with BeagleBone Black* » - H M Irfan Sadiq – Packt Publishing Ltd, 2015.

Solution :

```
[training]$ source  layers/poky/oe-init-build-env  builds/build-bbb

[build-bbb]$ nano  conf/local.conf
             MACHINE = "beaglebone-yocto"
             DL_DIR = "${TOPDIR}/../../downloads"
             IMAGE_FEATURES += "empty-root-password"

[build-bbb]$ bitbake  core-image-base
```

Travaux pratiques : produire une image pour Raspberry Pi



Le ***Raspberry Pi*** est un ordinateur monocarte produit par la *Raspberry Pi Fundation*. Son coût modéré et sa grande communauté d'utilisateurs lui valent un succès notable.

Processeur Broadcom BCM2708 (modèle 0 et 1), BCM2709 (modèles 2 et 3) BCM2711 (modèle 4)

512 Mo à 8Go de Ram

Pas de mémoire flash, fonctionne sur carte micro-SD

Ethernet, USB, GPIO, SPI, i²C, UART, Bluetooth, Wifi

Il n'est pas supporté directement par Poky, il faut ajouter un layer spécifique :

```
git://git.yoctoproject.org/meta-raspberrypi
```

Les modèles supportés sont décrits dans `meta-raspberrypi/conf/machine/`.

Les images proposées sont dans `meta-raspberrypi/recipes-core/images/`.

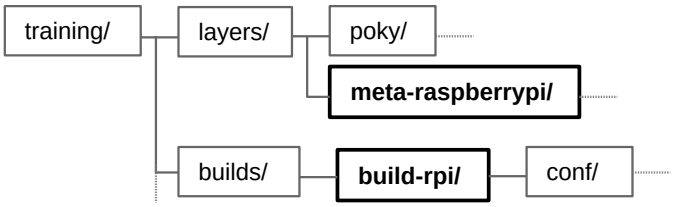
MACHINE	Modèle
`raspberrypi0`	Raspberry Pi Zero, modèle minimal sans connecteurs
`raspberrypi0-wifi`	Raspberry Pi Zero W, modèle Zero plus Wifi et Bluetooth
`raspberrypi0-2w` `raspberrypi0-2w-64`	Raspberry Pi Zéro 2 Wifi 32 et 64 bits
`raspberrypi`	Le Raspberry Pi original
`raspberrypi2`	Raspberry Pi 2, modèle quad-core.
`raspberrypi3` `raspberrypi3-64`	Raspberry Pi 3, plus rapide, ajoute Wifi et Bluetooth 32 et 64 bits
`raspberrypi-cm`	Version *System-on-Module* du Raspberry Pi 1.
`raspberrypi-cm3`	Version *System-on-Module* du Raspberry Pi 3.
`raspberrypi4` `raspberrypi4-64`	Raspberry Pi 4 (USB 3 et Ethernet Gigabit) 32 et 64 bits

Voir aussi

[MOCQ 2016] François Mocq – *Raspberry Pi 3 ou Pi Zero* – Editions ENI, 2016.

Compilation du BSP pour Raspberry Pi 4

1. Téléchargez le layer spécifique pour Raspberry Pi à côté du répertoire de Poky.
2. Créez un environnement de compilation nommé **`build-rpi`**.
3. Éditez le fichier `local.conf` et indiquez **`raspberrypi4`** dans la variable **`MACHINE`**.
4. Définissez également la variable `ENABLE_UART` à `"1"`.
5. Ajoutez le layer `meta-raspberrypi`.
6. Lancez la compilation d'une image « **`core-image-base`** ».



Voir aussi

[TEXIER 2016] : Pierre-Jean Texier, Petter Mabäcker - *Yocto for Raspberry Pi* – Packt Publishing, 2016

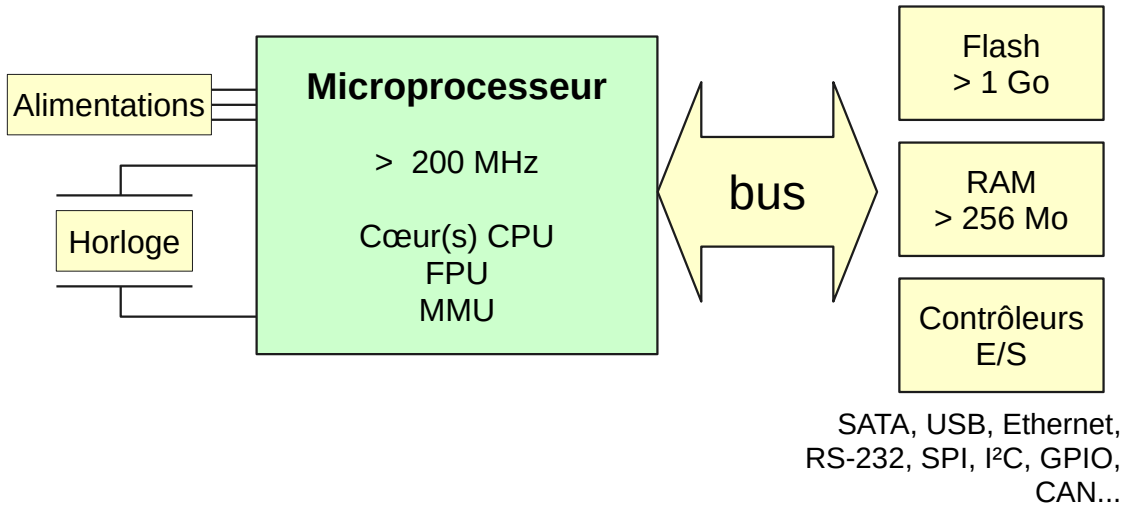
Solution :

```
[training]$ cd layers/

[layers]$ git  clone  git://git.yoctoproject.org/meta-raspberrypi  -b
kirkstone

[layers]$ cd ..
[training]$ source  layers/poky/oe-init-build-env  builds/build-rpi

[build-rpi]$ bitbake-layers  add-layer  ../../layers/meta-raspberrypi/

[build-rpi]$ nano  conf/local.conf
             MACHINE = "raspberrypi4"
             ENABLE_UART = "1"
             DL_DIR = "${TOPDIR}/../../downloads"
             IMAGE_FEATURES += "empty-root-password"

[build-rpi]$ bitbake  core-image-base
```

Composition d'un système Linux embarqué

Aspects matériels

Microprocesseur



Entrées-sorties réalisées par des contrôleurs externes au processeur.

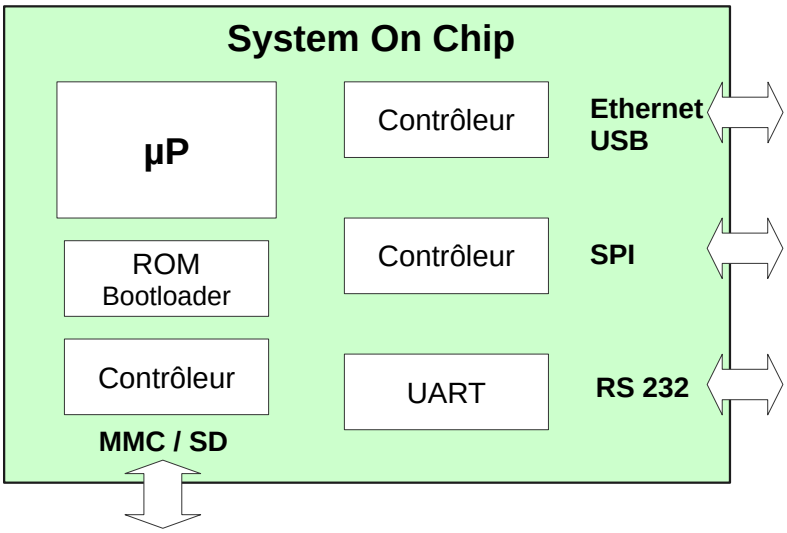
Mise en œuvre directe électroniquement très complexe.

Optimisé pour l'utilisation d'un système d'exploitation.

Quelques exemples de microprocesseurs :

- Famille Arm : ARMv7 (Cortex-A8, Cortex-A9, Cortex-A15, Cortex-M…) ARMv8 (Cortex-A35, Cortex-A53, Cyclone, Exynos, Cortex-A76, ...)
- Famille x86 : Intel (Atom, Core i5, i7...), AMD (Opteron, Phenom...), Via (C3, C7, Nano...)
- Famille M68k : Motorola 680x0, Coldfire (MCF5xxx), Dragonball.
- Famille PowerPC : Apple (G5), IBM (Power 6, Power 7, Power 8, Cell, Xenon)

System on Chip (S.O.C.)



Contrôleurs d'entrées-sorties déjà incorporés.

Intégration électronique encore assez complexe.

Entrées-sorties industrielles (CAN) ou analogiques (ADC/DAC, PWM) assez rares.

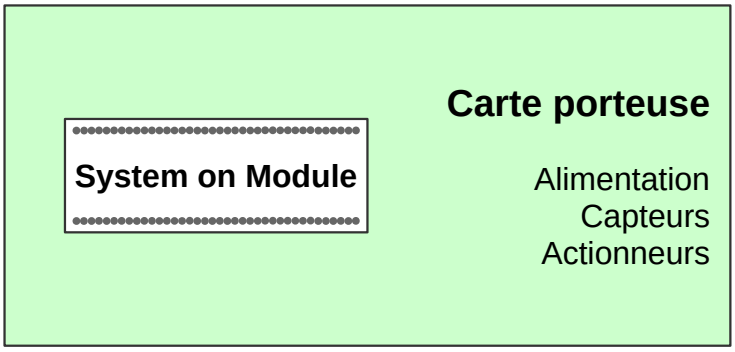
Quelques *systems-on-chip* Arm :

- Allwinner : A13, A20, A63, A80...
- Broadcom : BCM2835, BCM2837...
- NXP (Freescale) : LPC, i.MX21, i.MX23, i.MX6, i.MX8 OorIQ...
- Marvell : 88SE6, 88SE9...
- Rockchip : RK30xx, RK31xx, RK32xx, RK33xx...
- Texas Instruments : OMAP, DaVinci...

System-on-module (SOM)

Un **module** regroupe un *system-on-chip* et quelques composants (mémoire RAM et flash par exemple).

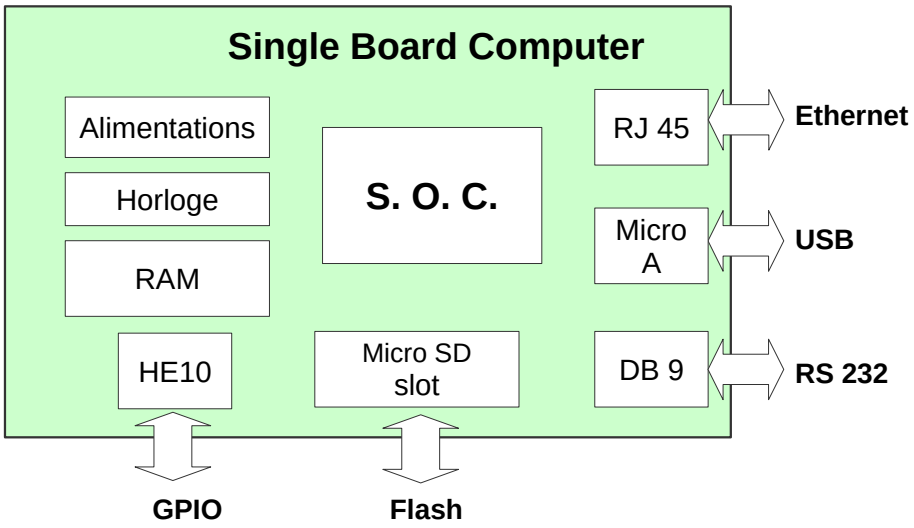
On le connecte à une carte porteuse contenant l'alimentation et l'électronique de communication externe.



Le développement de la carte porteuse est facilement réalisable par un bureau d'étude électronique.

Exemples de fournisseurs de modules : Phytec (phyCore, phyFlex...), Armadeus Systems (OposSOM, APF...), Marvell (Armada...), Acme Systems (Aria, Arietta, Acqua…), etc.

Single-Board-Computer (SBC)



Ordinateur **mono-carte** intégrant *system-on-chip*, mémoire, connecteurs d'E/S, etc.

Très utilisé pour le prototypage et la mise au point initiale.

Exemples : BeagleBone Black, Raspberry Pi, Banana Pi, OLinuXino...

Certains SBC (sans Linux) reposent sur des microcontrôleurs : Arduino, Launchpad, Teensy...

Spécificités logicielles d'un système Linux embarqué		
Applications finales	Open Office, Firefox, Gimp, VLC, Eclipse, etc.	Logiciels spécifiques, code métier.
Environnement graphique	X-Window (serveur Xorg). Environnements Gnome, Kde, etc.	DirectFB. libEFL. QtEmbedded.
Outils de développement	Chaîne de compilation GNU native, outils de mise au point.	Chaîne de compilation croisée sur l'hôte et débogage distant.
Shell	Bash, Ksh, Zsh, Dash, etc.	Ash dans Busybox
Utilitaires système	GNU Coreutils, Net-tools, Procps. Open SSH, Apache...	Busybox. Dropbear, Lighttp...
Initialisation Boot	Systemd, Init system V NetworkManager, Kmod...	Scripts pour Busybox
Bibliothèques	Standards. Gnu GlibC...	Réduites et adaptées, uClibC, musl...
Noyau	Universel, fourni par la distribution, nombreux modules.	Ajusté précisément. Uniquement les modules nécessaires.
Matériel	**Poste de travail. Serveurs.**	**Systèmes embarqués.**

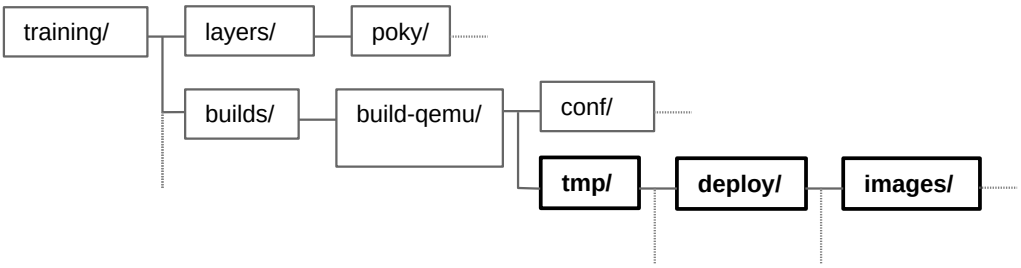
Résultats de la compilation

Yocto Project stocke tous les fichiers qu'il écrit dans le répertoire `tmp/` (configuration dans `conf/local.conf`).

- `tmp/work/` : est le répertoire où il réalise ses compilations
- `tmp/deploy/` : contient ses résultats :
- `tmp/deploy/licenses/` : la liste des licences des packages compilés
- `tmp/deploy/rpm/` : est un dépôt des packages binaires produits
- `tmp/deploy/images/<machine>` : contient toutes les images pour la cible.

On trouve (variable suivant les cibles) :

- un *bootloader* primaire dépendant de la plate-forme (ex : `MLO` pour Beaglebone)
- un *bootloader* secondaire plus standard (`u-boot.img`)
- une image du noyau et du *device tree* (fichiers `zImage` et `.dtb`)
- une archive (`.tar.bz2`) de l'arborescence du *root filesystem* ;
- parfois une image complète de la carte SD à produire.



Travaux pratiques : lancement de l'image sur l'émulateur Arm

Les fichiers produits par la compilation sont dans `tmp/deploy/images/qemuarm`

Il faut fournir plusieurs fichiers à l'émulateur :

- Le root filesystem : `core-image-minimal-qemuarm.ext4`
- Le noyau : `zImage-qemuarm.bin`
- Éventuellement les fichiers du *device tree*.

Pour simplifier le démarrage, Yocto Project nous fournit un script de lancement nommé **`runqemu`**.

Démarrage de l'émulateur :

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

Démarrage de l'émulateur sur la même console :

```
[build-qemu]$ runqemu  nographic
```

Travaux pratiques : installation de l'image sur une cible matérielle

- Vérifiez le résultat de la compilation dans le répertoire

```
tmp/deploy/images/beaglebone-yocto/
      ou
tmp/deploy/images/raspberrypi4/
```

- Insérez une carte micro-SD dans un adaptateur USB de votre PC.
- Démontez le système de fichiers de la carte s'il a été auto-monté.
- Copiez l'image produite (extension `.wic` ou `.rpi-sdimg`) directement sur la carte.
- Insérez la carte micro-SD dans le Beagle Bone Black ou le Raspberry Pi.

Solutions :

Toute cible :

```
[build-???]$ lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda      8:0    0  1,8T  0 disk
├─sda1   8:1    0  1,8T  0 part /
└─sda5   8:5    0    3G  0 part [SWAP]
sdb      8:32   1  7,4G  0 disk
└─sdb1   8:33   1    4G  0 part /media/$USER/SDCARD
sr0     11:0    1 1024M  0 rom

[build-???]$ sudo umount /dev/sdb?
```

Beagle Bone Black :

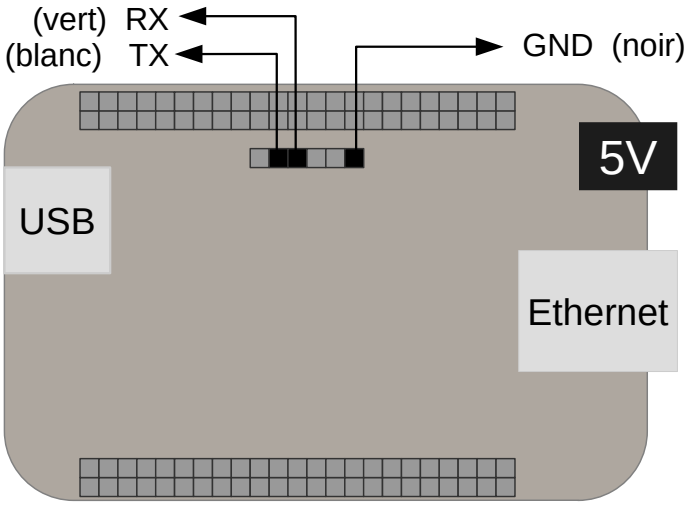
```
[build-bbb]$ sudo cp tmp/deploy/images/beaglebone-yocto/core-image-
base-beaglebone-yocto.wic  /dev/sdb
```

Raspberry Pi :

```
[build-rpi]$ sudo cp  tmp/deploy/images/raspberrypi4/core-image-base-
raspberrypi4.rpi-sdimg  /dev/sdb
```

Connexion sur BeagleBone Black

Branchez un câble de *debug* série pour accéder à la Beaglebone Black et lancez l'utilitaire **`minicom`** pour vous connecter sur la cible.

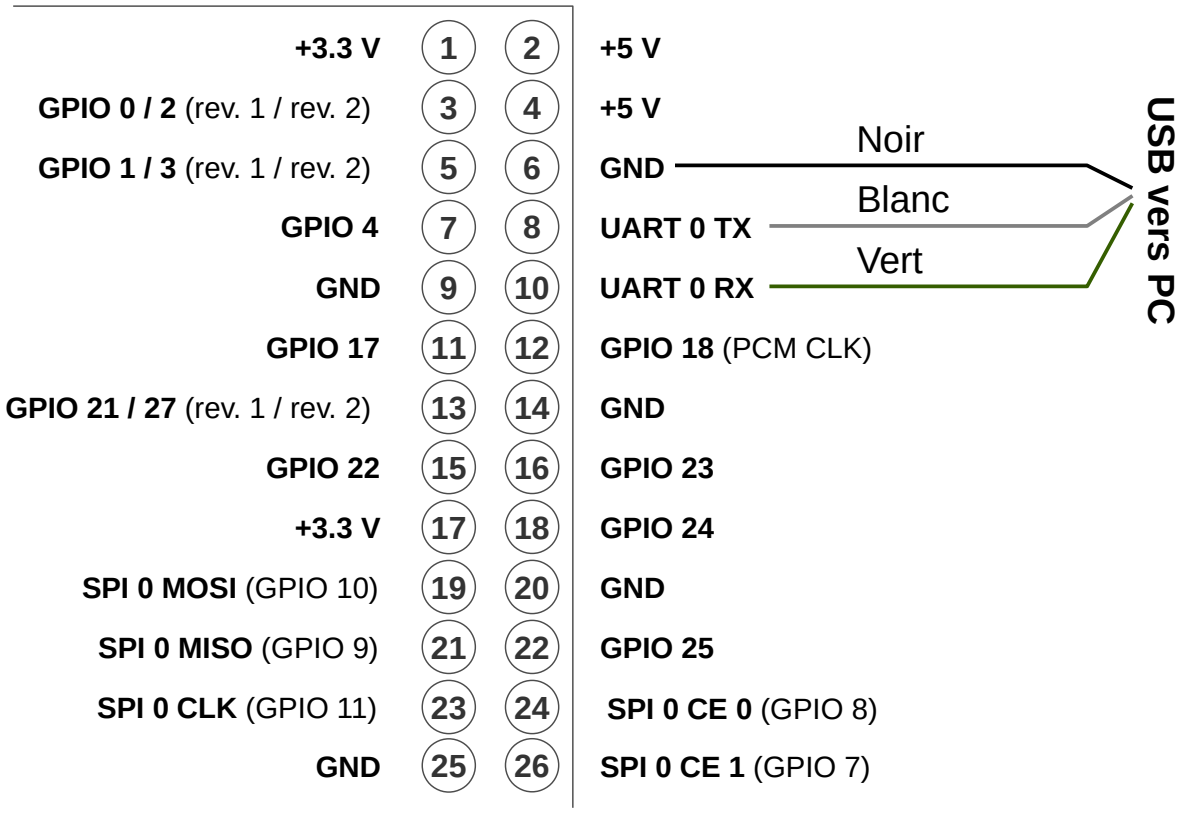



```
$ sudo minicom -b 115200 -D /dev/ttyUSB0
```

NB : si la saisie ne fonctionne pas, désactivez le contrôle de flux matériel dans la configuration du port série de minicom.

Connexion sur Raspberry Pi

Connectez vous avec un câble de *debug* série et l'utilitaire **`minicom`**.

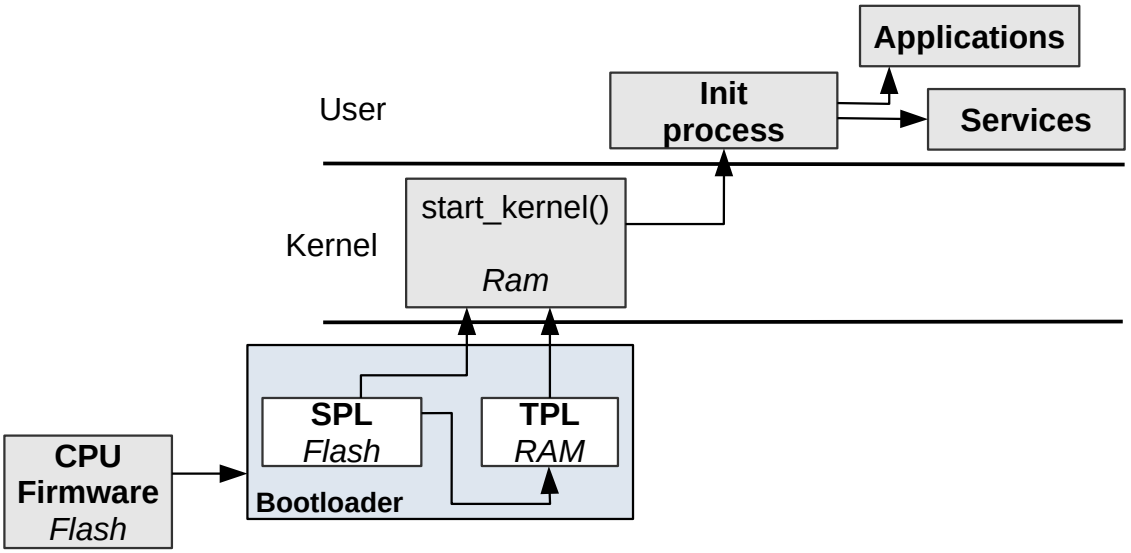



```
[build-rpi]$ sudo  minicom  -b  115200  -D  /dev/ttyUSB0
```

NB : si la saisie ne fonctionne pas, désactivez le contrôle de flux matériel dans la configuration du port série de minicom.

Boot du système Linux

Bootloader et kernel



Il existe plusieurs *bootloaders* par exemple : Uboot, Barebox, Grub, etc.

1. Une mémoire flash, EEPROM, ou ROM contient un ***firmware*** (micro-code) immuable qui initialise le CPU, et transmet le contrôle au *bootloader*. Ce micro-code contient parfois une interface minimale pour paramétrer le chargement du *bootloader*.
2. Le ***bootloader*** (chargeur de démarrage) prend le relais, pour charger une image du noyau Linux (placée en mémoire flash, sur une partition de carte SD, depuis un réseau, etc.).
3. Le **noyau Linux** s'initialise, détecte les périphériques et monte son système de fichiers principale. Il cherche un fichier `init` qu'il exécute.
4. Le processus **init** configure le système depuis l'espace utilisateur et démarre les applications-métier.

Démarrage de l'espace utilisateur : init

Lors du *boot* du système, le noyau :

- détecte le type de processeur, la mémoire, le contrôleur d'interruptions, les bus, etc. ;
- prépare toutes les structures de données internes nécessaires au fonctionnement du noyau (table des processus, descripteurs de fichiers...) ;
- détecte et initialise les périphériques dont les pilotes (*drivers*) sont compilés statiquement dans le noyau : contrôleurs scsi, disques durs, cartes réseau, cartes graphiques, etc. ;
- monte le système de fichiers se trouvant sur le périphérique bloc dont le nom est fourni par l'option « **`root=...`** » (en lecture seule si l'option « `ro` » a été indiquée dans les arguments de boot) et monte éventuellement `/dev` ;
- lance un premier processus utilisateur, dont le PID est toujours 1, et dont le fichier exécutable est recherché ainsi :
  - fichier indiqué dans l'option « **`init=...`** » de la ligne de commande du noyau
  - si cette option n'est pas fournie, on recherche le premier fichier exécutable existant parmi (dans cet ordre) : `/sbin/init`, `/etc/init`, `/bin/init`, `/bin/sh`.

Voir aussi

Le fichier `main.c` dans le sous-répertoire `init/` des sources du noyau Linux. En particulier la fonction `init_post()` :

```
[...]
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s.  Attempting "
                            "defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found.  Try passing init= option to kernel.");
}
[...]
```

Rôles du processus init

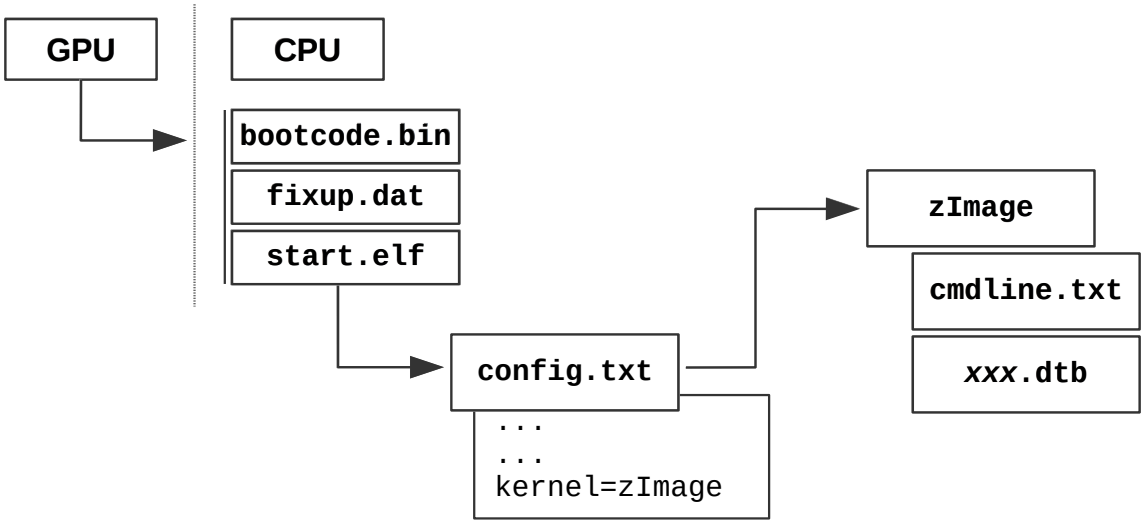
Le travail du processus `init` est double :

1) terminer l'initialisation du système depuis l'espace utilisateur ;
   - monter les systèmes de fichiers spéciaux (`/proc`, `/sys`...) ;
   - remonter le système de fichiers initial en lecture et écriture ;
   - charger les modules du noyau supplémentaires dont on a besoin ;
   - initialiser les adresses des interfaces réseau ;
   - démarrer les « démons » qui fonctionneront en arrière-plan ;
   - lancer les applications principales du système.

2) en fonctionnement normal, « adopter » les processus orphelins, et lire les codes de terminaison à leurs morts.

Sur les postes de travail et serveurs, le processus `init` hérité des Unix Système V est souvent remplacé par celui du projet *Systemd*.

Cas particulier : boot du Raspberry Pi



La GPU du Raspberry Pi charge un *bootloader* propriétaire (fourni par Broadcom) en mémoire, pour qu'il soit exécuté par le CPU.

Ce *bootloader* s'appelle **`bootcode.bin`** et doit se trouver sur la première partition de la carte micro-SD, formatée avec le système de fichiers `vfat` (fat 32).

Le *bootloader* lit un fichier de configuration nommé **`config.txt`** se trouvant sur la même partition.

Le *bootloader* utilise ensuite deux fichiers : **`fixup.dat`** et **`start.elf`** qui chargeront à leur tour le noyau dont le nom est précisé dans `config.txt` (ici **`zImage`**). En outre le contenu du fichier **`cmdline.txt`** est passé en paramètre au noyau ainsi que la configuration matérielle décrite dans le *device tree* (fichier **`.dtb`**)

Partitionnement du système