Solutions pour Linux embarqué

Panorama et critères de choix

Christophe Blaess
http://christophe.blaess.fr
http://www.logilin.fr

Ingénierie et formations Linux industriel

www.logilin.fr christophe.blaess.fr

Introduction

Qu'est-ce qu'un système embarqué ?
Contraintes sur un système embarqué

Microcontrôleur vs microprocesseur

Introduction - terminologie
Différences matérielles
Démonstration / Comparaison

Choix d'une architecture matérielle

Prototypage – Projet personnel
Petite série – Startup
Grande série – production industrielle

Apports d'un système d'exploitation

Exécution des tâches
Mémoire virtuelle et MMU
Abstraction des périphériques
Systèmes d'exploitation libres pour l'embarqué

Linux pour l'embarqué

Composants d'un système Linux
Les distributions Linux
Démonstration

Build systems

Buildroot
Yocto Project
Démonstration

Environnement logiciel

Licences libres
Temps réel libre pour système embarqué
Environnement de développement
Prototypage – Projet personnel
Mise en production

Conclusion

Questions ?

Introduction

Qu'est-ce qu'un système embarqué ?

*To **embed**, (embedded, embedding) : (v. t.) enfoncer, sceller, noyer, enchâsser, incruster. (Larousse)*

Un système embarqué est un ensemble électronique et/ou informatique intégré comme composant d'un environnement plus important.

Un système embarqué se définit surtout par les contraintes auxquelles il est soumis.

L'identification précise des contraintes doit se faire dès la conception du système.

Le terme « système embarqué » − pourtant très utilisé − n'est pas très exact, on lui préfère parfois « logiciel enfoui » ou « système incorporé ».

Pour en savoir plus :

- [FICHEUX 2012] Pierre Ficheux et Éric Bénard, *Linux embarqué*, Eyrolles 2012
- [BLANC 2011] Gilles Blanc, *Linux embarqué : comprendre, développer, réussir*, Pearson 2011.

Contraintes sur un système embarqué

Contraintes matérielles

- **Performance** : puissance CPU, capacité mémoire, stockage, richesses I/O…
- **Encombrement** : ventilateur / radiateur, 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...

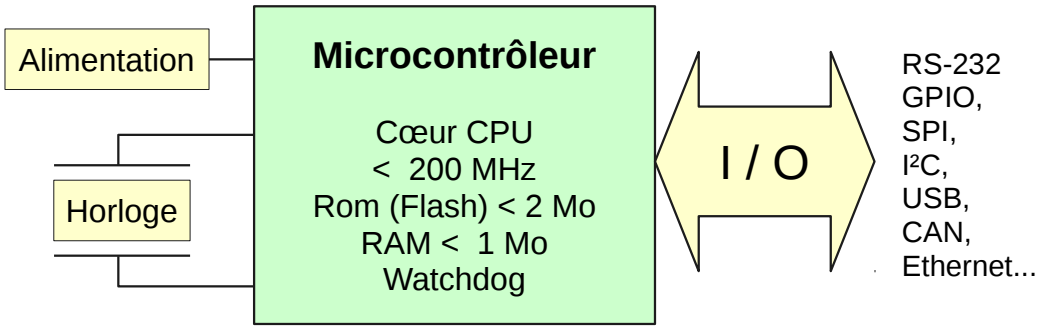
Contraintes économiques

- **Concurrence** : qualité, IHM, ergonomie...
- **Coûts** : choix des composants, licences logicielles...
- **Évolutivité** : Présence d'un « *market* » applicatif ouvert...

Microcontrôleur *vs* microprocesseur

Introduction - terminologie

Microcontrôleur (µC)

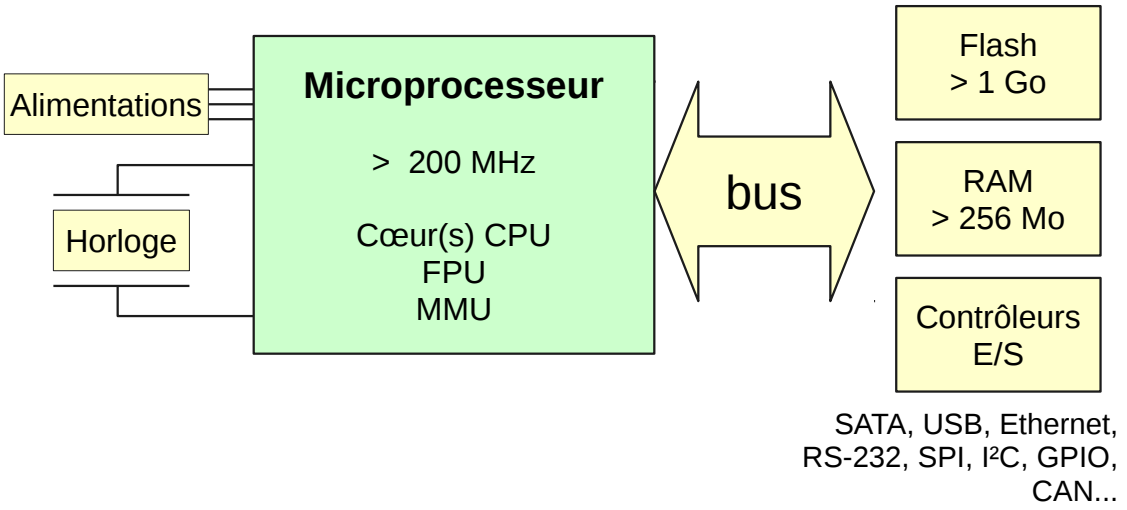


- Mise en œuvre électronique simple.
- Déterminisme et fiabilité de fonctionnement.
- Généralement pas de système d'exploitation (ou minimal).

Quelques exemples de microcontrôleurs :

- Atmel : AT91 (AT91SAM9xx, AT91SAM7xx), AVR (TinyAVR, MegaAVR…)
- Freescale : 68HC05, 68HC08, 68HC11
- Hitachi / Renesas : H8 (H8/300, H8/500…), SuperH (SH-1, SH-2, SH-3...)
- Intel : 8051, 8052, 8085
- Microchip : PIC (PIC-10F, PIC-12F, PIC-16F, PIC-24F)
- STMicroelectronics : ST6, STM8, STM32
- Texas Instruments : MSP430, TMS320, TM4C

Microprocesseur (µP)



Entrées-sorties réalisées par des contrôleurs externes au processeur

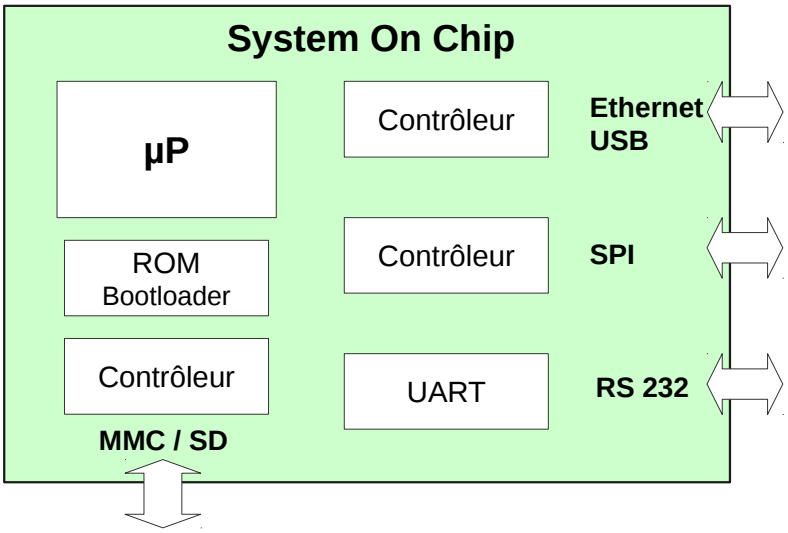
Mise en œuvre électronique beaucoup plus complexe

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

Quelques exemples de microprocesseurs :

- Famille Arm : ARM7 (Arm 720T), ARM9 (Arm 926) ARM11 (Arm 1176jzf), Cortex-A (Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15),
- Famille x86 : Intel (Atom, Core 2, Core i5, Core i7), AMD (Opteron, Phenom), Via (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

Souvent peu d'entrées-sorties industrielles (CAN) ou analogiques (ADC/DAC, PWM)

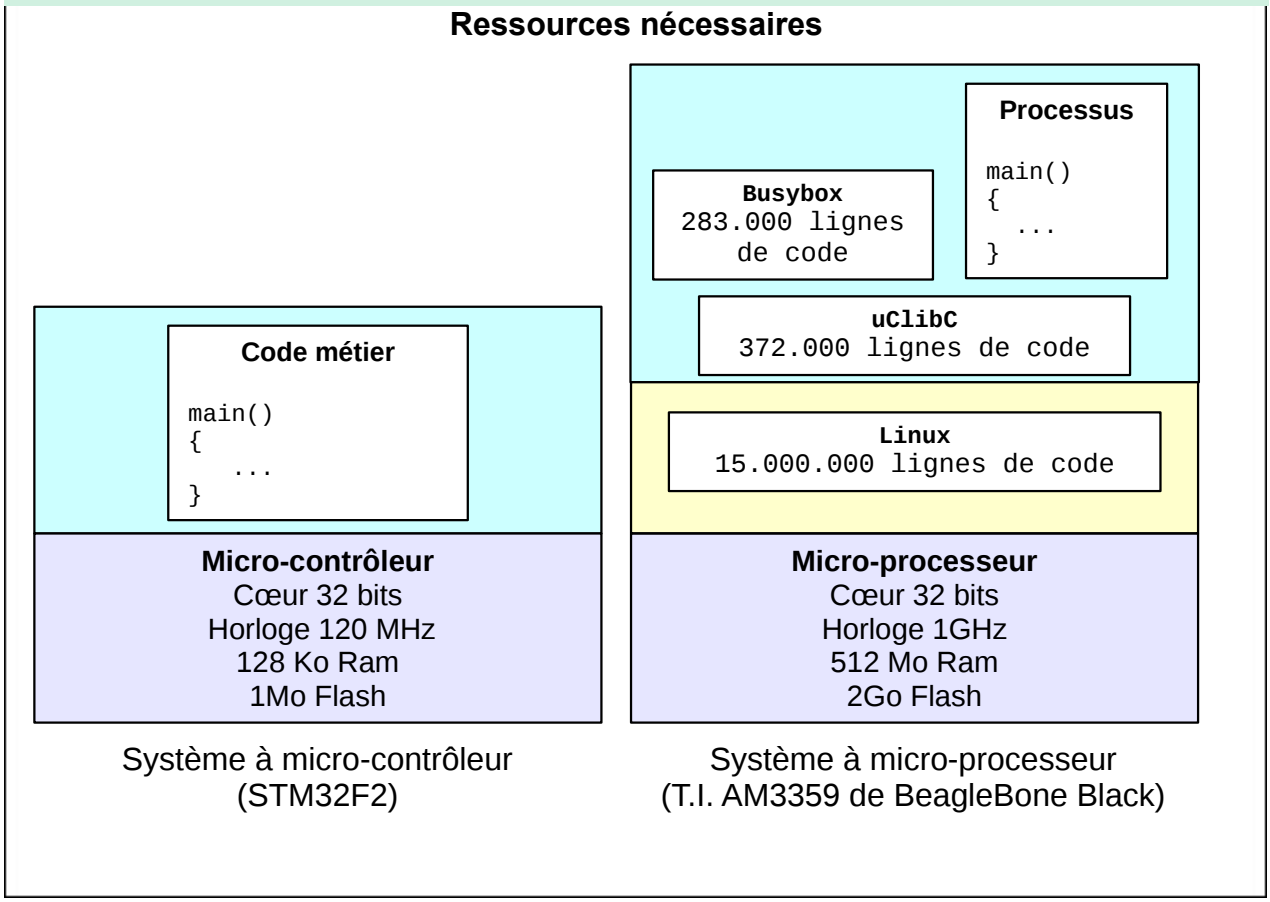
Quelques *systems-on-chip* Arm :

- Allwinner : A13, A20, A80
- Broadcom : BCM2835, BCM2836
- Freescale : i.MX21, i.MX23, i.MX6
- Marvell : 88SE6, 88SE9
- Rockchip : RK30, RK31
- Texas Instruments : OMAP, DaVinci

Différences matérielles

Comparatif

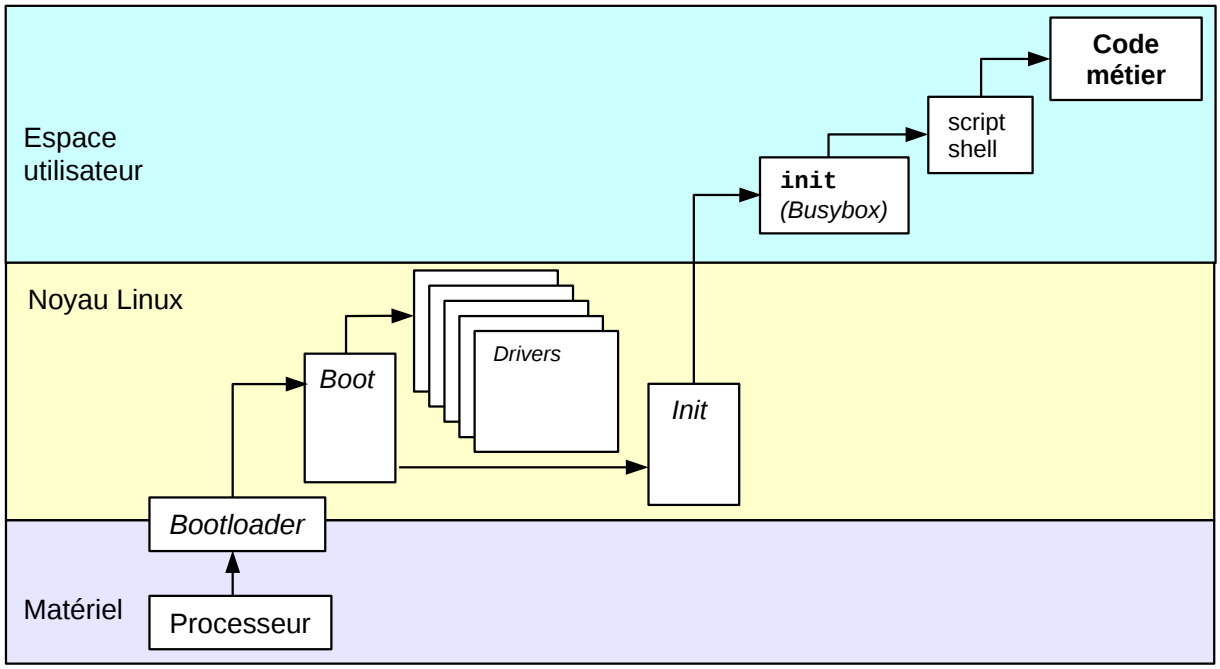
	Microcontrôleur	***System-on-chip***
Coût moyen	< 10 €	> 20 €
PCB support	Simple – 2 couches	Complexe – 6 couches
Alimentations	Simple – 3.3 V	Multiples 3.3V / 5V / 12 V
Volume de code métier	~ 100 ko	> 10 Mo
Environnement de développement	Propriétaire > 3000 € Libre (Gnu) pas toujours disponible.	Libre (Gnu) et gratuit si développement sous et pour Linux.
Mise au point du code	Complexe. Débogueur spécifique.	Plus simple grâce à l'O.S.
Déploiement, mise à jour	Complexe.	Simple. Plusieurs possibilités.
Protection du code métier	Facile (fusibles).	Difficile.



Sur un système à micro-contrôleur, le code métier est le seul maître à bord, il accède à volonté aux périphériques, à la mémoire, etc.

Au contraire, dans un système à micro-processeur, le code métier n'est qu'une petite partie de l'ensemble du logiciel. Il est soumis à l'ordonnancement et au sous-système de gestion mémoire du noyau. Il s'appuie sur des bibliothèques et des utilitaires externes.

Temps de boot



Suivant le type de processeur et la complexité du matériel, le temps de *boot* du noyau dure de deux à cinq secondes environ.

Le démarrage du processus `init`, les tâches administratives (montage systèmes de fichiers, configuration paramètres de `/proc`, etc.) prennent une à deux secondes supplémentaires.

Le lancement de tous les services (réseau, authentification, environnement graphique, etc.) peut demander une dizaine de secondes.

Pour en savoir plus

« *Optimisation du temps de boot d'un système Linux embarqué* » Christophe Blaess – Open Silicium 9. Voir http://www.blaess.fr/christophe/articles/ : 4 décembre 2013.

Démonstration / Comparaison

Microcontrôleur Texas Instrument MSP430

Carte de développement « *Launchpad MSP430* ».

Contenu : fichier `blinking-leds.c`

Taille mémoire flash utilisée : 16 Ko.

Temps de compilation / installation : 1 minute.

Microprocesseur Texas Instrument AM3358

Carte « *BeagleBone Black* »

Contenu : Linux 3.12.10, Busybox 1.23.1, uClibC 0.9.33.2 + script `blinking-leds.sh`

Taille mémoire flash utilisée : 3 Mo.

Temps de compilation / installation (système complet) : environ 30 minutes.

blinking-leds.c :

```
#include <stdlib.h>
#include <msp430g2553.h>

int main(void)
{
    unsigned int i, j;

    WDTCTL = WDTPW + WDTHOLD; // Watchdog off.
    P1DIR |= BIT0 | BIT6;     // Leds P1.0 and 01.6 on output
    P1OUT &= ~(BIT0 | BIT6);  // Leds off.

    for (;;) {
        for (j = 0; j < 4; j ++) {
            P1OUT |= BIT0 | BIT6; // Leds on
            for (i = 0; i < 15000; i ++)
                ;

            P1OUT &= ~(BIT0 | BIT6); // Leds off
            for (i = 0; i < 15000; i ++)
                ;
        }
        for (i = 0; i < 50000; i++)
                ;
    }
    return 0;
}
```

Choix d'une architecture matérielle

Microcontrôleur

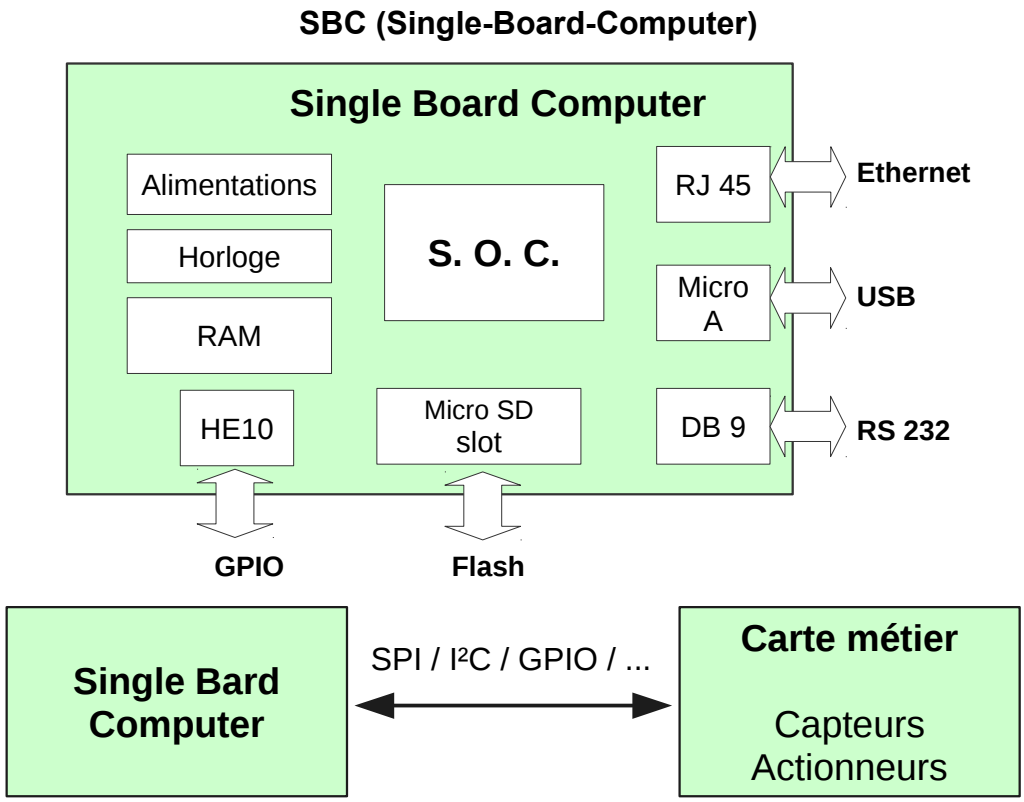
- **Prix** : conception et réalisation PCB, coût unitaire.
- **Simplicité** : fiabilité, code certifiable.
- **Prédictibilité** : temps d'exécution, déterminisme.

Microprocesseur (système d'exploitation)

- **Puissance** : calcul, mémoire, optimisation.
- **Évolutivité** : isolation du code métier par rapport au matériel, portabilité.
- **Richesse applicative** : piles de protocoles, services…

Sauf cas très spécifiques, l'emploi d'un microprocesseur sans système d'exploitation manque d'efficacité dans l'utilisation des ressources disponibles.

Prototypage – Projet personnel

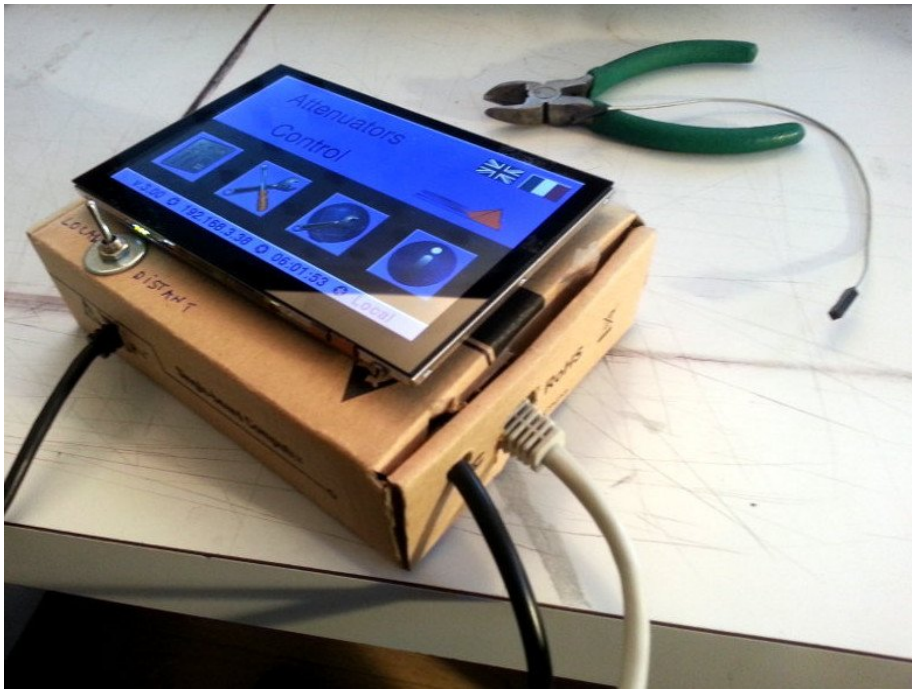


Ordinateur **mono-carte** intégrant *system-on-chip*, mémoire, connecteurs d'E/S, etc.

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

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

On teste éventuellement plusieurs SBC afin de déterminer le processeur le plus adapté.



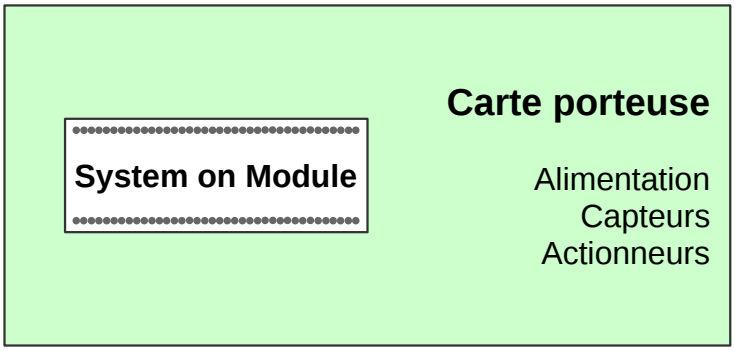
Exemple de prototype construit autour d'un S.B.C.

IHM applicative conçue et développée par Logilin. Pour vos projets : ingenierie@logilin.fr

Petite série – Startup

Computer-on-module (C.O.M.) - System-on-module (S.O.M)

Un **module** est une petite carte de dimension réduite contenant l'équivalent d'un ordinateur mono-carte (S.O.C, mémoire…) sans connecteurs.



La liaison avec la carte porteuse se fait soit par broches HE10 soit par collage CMS.

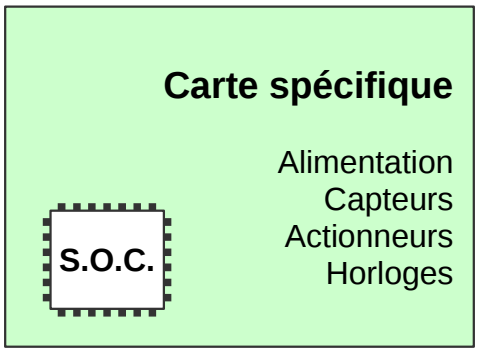
La carte porteuse doit être étudiée pour s'intégrer dans le boîtier final du projet.

C'est une phase qui fait souvent appel à un financement participatif (*crowd-funding*)

Grande série – production industrielle

Intégration d'un *system-on-chip*

Rarement intéressant en dessous d'une dizaine de milliers d'unité.



Coûts importants de design, routage, banc de test, validation, etc.

Les frais de production sont avantageux.

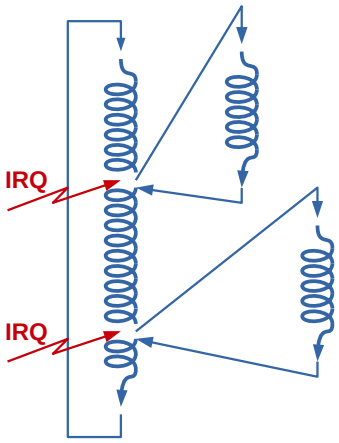
Externalisation de la conception : attention à la propriété intellectuelle

Apports d'un système d'exploitation

Exécution des tâches

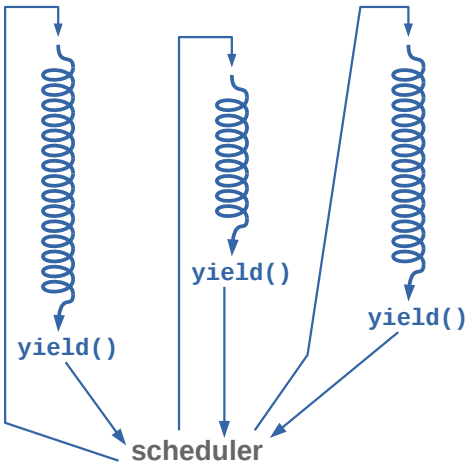
Monotâche

« *super-loop* »

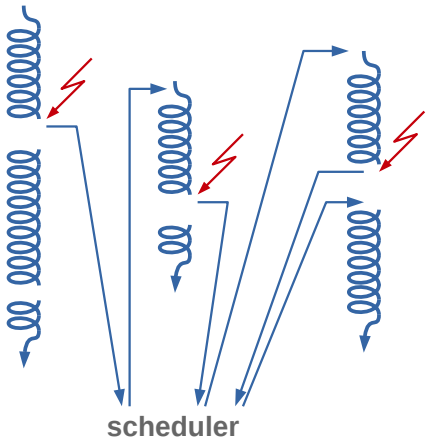


Monotâche
+
interruptions

Ce type de programmation, hérité des automates, est plus adapté aux microcontrôleurs.



Multitâche coopératif



Multitâche préemptif

L'**ordonnanceur** (*scheduler*) est une fonctionnalité essentielle des systèmes d'exploitation pour exécuter des tâches (non organisées entre-elles) sur un même processeur.

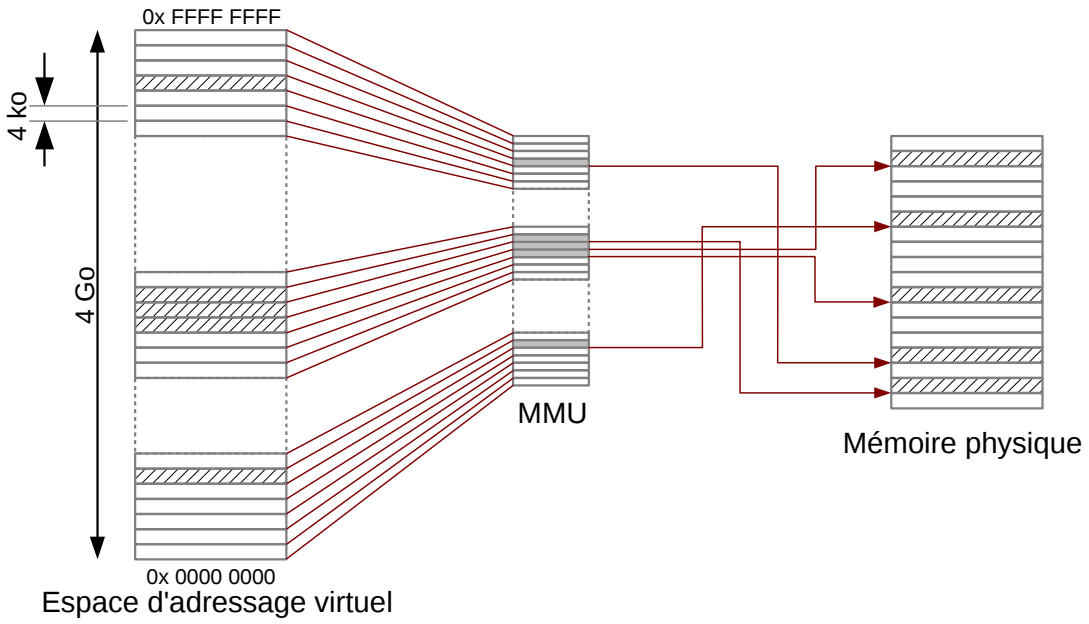
Il existe plusieurs modes d'ordonnancement :

- **temps partagé** (*time sharing system*) : comportement par défaut sur les O.S. comme Linux
- **temps réel** (*realtime scheduling*), suivant des algorithmes comme *Round Robin* ou *Fifo* basés sur des priorités entre tâches ou *Earliest Deadline First* utilisant des temps d'expiration des tâches.

Pour en savoir plus…

« *Solutions temps réel sous Linux* » - Christophe Blaess – Éditions Eyrolles 2012.

Mémoire virtuelle et MMU



Un processus voit un espace de mémoire virtuelle, au sein du quel il peut accéder à n'importe quelle adresse de `0x000000` à `0xFFFFFFFF` (sur processeur 32 bits).

Cet espace est découpé en pages, et la MMU – Memory Managment Unit (un composant intégré dans le processeur) – associe une page de mémoire virtuelle avec une page de mémoire physique en effectuant la modification d'adresse lors de l'accès à la mémoire.

Certaines pages de mémoire virtuelle n'ont pas de correspondance en mémoire physique : une tentative d'accès déclenche une interruption « faute de page ».

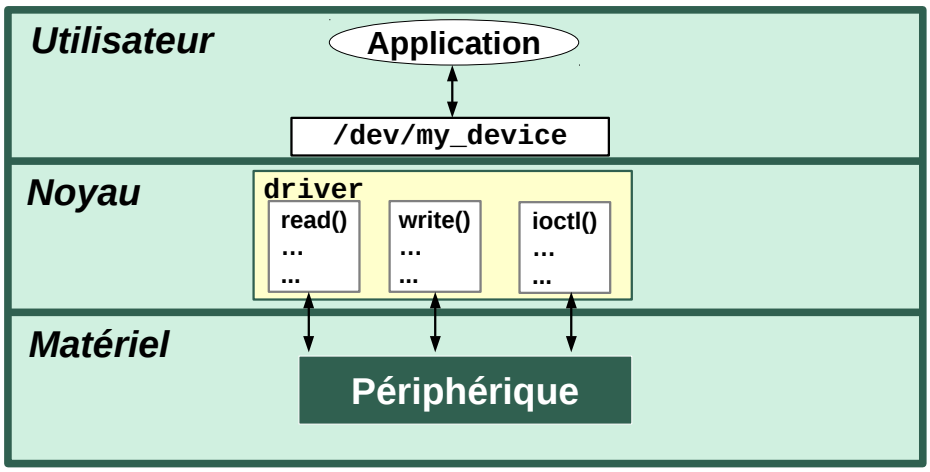
Chaque processus dispose d'une configuration personnelle de la MMU. Cette dernière est programmée à chaque commutation entre deux processus.

Un processus ne voit que les pages de mémoire physique qui lui ont été attribué par le noyau ; les pages des autres processus ne sont projetées à aucun emplacement de sa mémoire virtuelle.

Abstraction des périphériques

Le support pour les périphériques est assuré par des pilotes (*drivers*) qui peuvent être développés de manière externe au noyau (modules).

Les applications de l'espace utilisateur communiquent avec les périphériques en réalisant des opérations (lecture, écriture paramétrage...) sur des pseudo-fichiers (généralement stockés dans `/dev`)



Le développement de drivers robustes et portables entre différentes versions du noyau n'est pas une opération triviale (surtout sur des architectures multi-cœurs) !

Systèmes d'exploitation libres pour l'embarqué

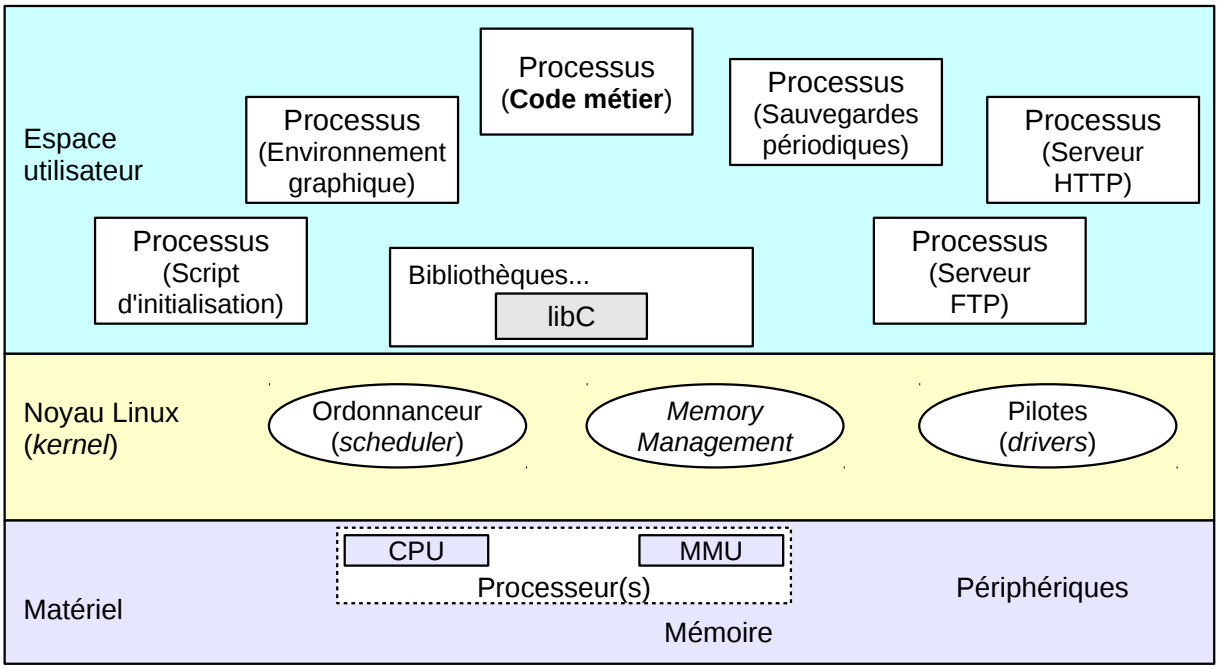
	Lepton	**FreeRTOS**	**RTEMS**	**Linux**
Ordonnancement préemptif	√	√	√	√
Temps-réel	+ +	+ + +	+ + +	+
Séparation mémoire (via MMU)				√
Isolation matériel (drivers)	+			+ + +
Pile protocoles (réseau, USB...)	+	+	+	+ + +
Systèmes de fichiers	+ +		+	+ + +

Lepton, FreeRTOS et RTEMS sont surtout des systèmes à destination de microcontrôleurs puissants (32 bits) plutôt que pour microprocesseurs.

Linux n'offre nativement que des performances assez limitée pour le temps réel, des extensions libres comme PREEMPT_RT ou Xenomai permettent de l'améliorer sensiblement.

Linux pour l'embarqué

Composants d'un système Linux

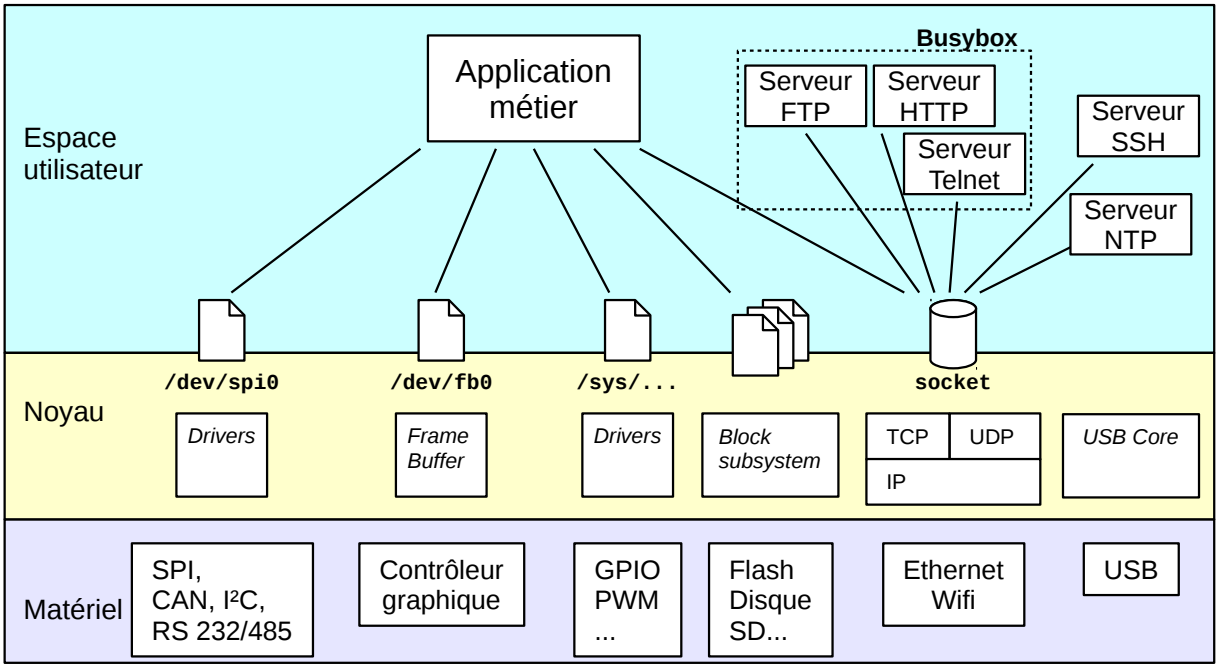


Le **rôle du noyau Linux** : mettre les ressources offertes par le matériel à disposition des applications de l'espace utilisateur.

Lorsqu'il exécute le code du noyau, le processeur est en mode superviseur (privilégié). Pour exécuter du code en espace utilisateur, il passe en mode protégé (non-privilégié).

Les tâches (*threads*) de l'espace utilisateur s'exécutent dans des processus (espaces de mémoire disjoints).

Services offerts par le noyau Linux



Le noyau Linux permet directement d'accéder à plusieurs centaines de périphériques et protocoles de communications, de réaliser des opérations d'entrées-sorties aisément, d'afficher une interface graphique, de lire des dizaines de formats de systèmes de fichiers, de dialoguer avec de nombreux protocoles réseau, etc.

Si un élément n'est pas présent dans une configuration embarquée, il suffit généralement de recompiler le noyau (voire de ne compiler qu'un module) pour l'intégrer.

Les distributions Linux

Une **distribution** =noyau Linux, bibliothèques systèmes et utilitaires de base

\+ quelques milliers de packages prêts à installer

\+ un système d'installation et mise à jour.

Famille **Debian**	Famille **Redhat**
Debian 8 Jessie **Ubuntu** 15.04 Snappy Ubuntu Core (IoT) Linux **Mint** 17.1	Red Hat Enterprise Linux (**RHEL**) 7.1 **Fedora** Linux 21 **CentOS** 7.1
Packages = fichiers `.deb` `dpkg -l` `apt-get install` <package>	Packages = fichiers `.rpm` `rpm -qa` `yum install` <package>

Il existe également des distributions plus spécifiquement conçues pour l'embarqué comme ArchLinux ou Ångström.

Android peut être considéré comme une distribution Linux particulière où l'essentiel des packages sont des applications pour machine virtuelle Java.

Démonstration

Distribution Raspbian sur Raspberry Pi

1. Télécharger le fichier `2015-05-05-raspbian-wheezy.zip` (990 Mo)
   depuis https://www.raspberrypi.org/downloads

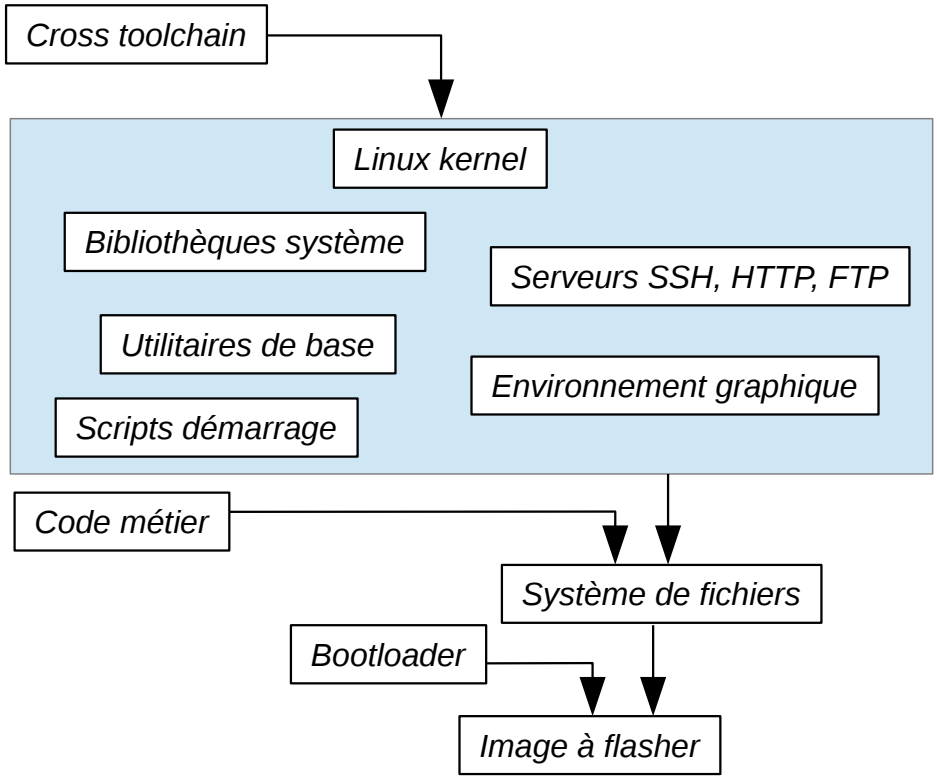
2. Copier le fichier sur une carte SD vierge
   ```
   dd if=2015-05-05-raspbian-wheezy.zip  of=/dev/sdc  bs=1M
   ```

3. Insérer la carte dans le Raspberry Pi et le démarrer
   (brancher écran / clavier / souris sur le Raspberry PI auparavant)

L'utilisation d'une distribution est plutôt réservée à un poste de travail ou à un serveur.

Pour un système embarqué, on préfère généralement maîtriser l'ensemble des composants du système. On utilise alors un « *build system* ».

Build systems



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 plate-forme Linux embarqué.

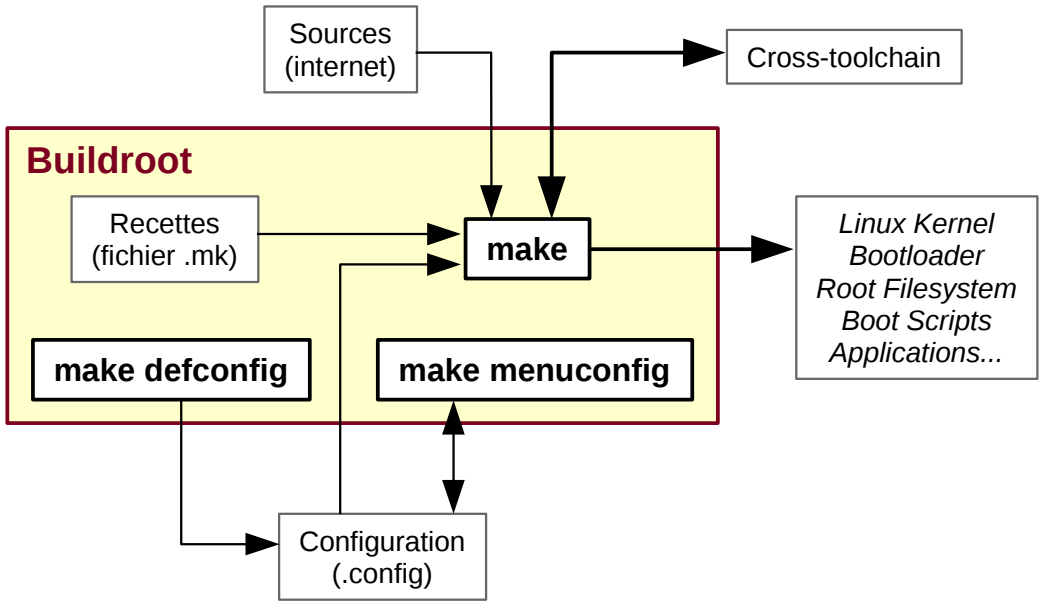
Voir aussi

« *Distributions embarquées pour Raspberry Pi* » - Pierre Ficheux – Open Silicium n° 7

« *Raspberry Pi from scratch* » - Christophe Blaess – Linux Magazine France n° 155 et 158.

Buildroot

Buildroot est un ensemble 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.



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

Avantages :

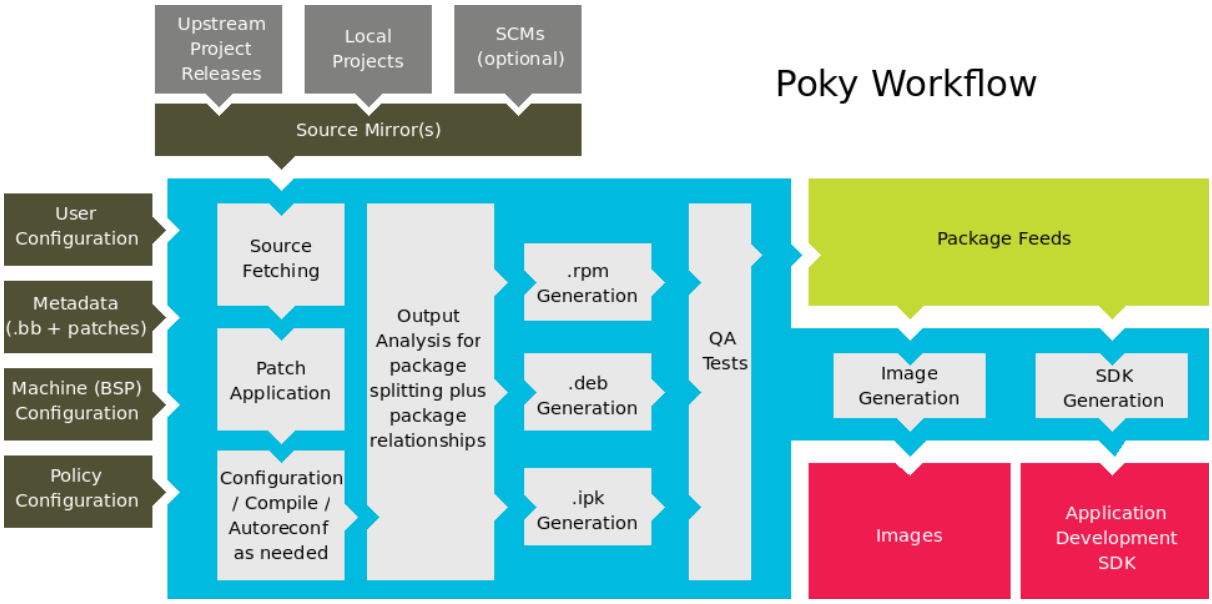
- Système relativement simple à comprendre et à modifier
- Possibilité d'ajouter assez aisément des *patches* pour les composants système.
- Intégration aisée du code métier.

Inconvénients :

- Difficile de gérer plusieurs architectures en parallèles pour du code métier à rendre disponible sur plusieurs plates-formes.
- Pas de gestion de packages, génération d'une image contenant tout le système.

Yocto Project

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



Source : http://www.yoctoproject.org

Avantages :

- Très complet et très riche.
- 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.

Démonstration

Construction d'une image pour BeagleBone Black avec Buildroot.

Boot sur carte micro-SD avec deux partitions :

Partition 1 « Boot » Système de fichiers Fat	Partition 2 « Root » Système de fichiers Ext2

`MLO`
`u-boot.img`
 	
`uImage`
 	
`am335x-boneblack.dtb`
`am335x-bone.dtb`
	`/bin`
	`/boot`
	`/dev`
	`/etc`
	`/home`
	`/lib`
	`/mnt`
	`/proc`
	`/root`
	`/sbin`
	`/sys`
	`/tmp`
	`/usr`
	`/var`

Environnement logiciel

Licences libres

GPL – *General Public License*

Une majorité des outils libres, y compris le noyau Linux lui-même, sont sous licence GPL.

Elle garantit **quatre droits** fondamentaux à l'utilisateur :

- le droit d'**exécution** et d'utilisation du programme
- le droit d'**analyse**, de correction, de modification du programme
- le droit de **partage** du programme
- le droit de **partage** d'une version **modifiée**.

La licence GPL impose aussi à l'utilisateur un **devoir**, exprimé par la clause du « ***copyleft*** » :

La redistribution d'un programme doit s'accompagner des mêmes droits que ceux obtenus initialement.

La GPL garantit la liberté d'un code, et la pérennité de cette liberté.

Il existe d'autres licences (BSD, MPL, MIT, etc.).

LGPL – *Lesser General Public License*

La licence LGPL permet de disposer de **bibliothèques libres**, dont l'implémentation bénéficie des mêmes critères que le code GPL, mais qui peuvent être employées dans des applications propriétaires.

Il est parfaitement possible de développer des **applications propriétaires** sous Linux, même si elles emploient des bibliothèques libres.

Le noyau – Les drivers

Le noyau étant sous licence GPL, tout code compilé statiquement dans le kernel doit être sous une licence compatible.

Il est possible (même si peu recommandé) de développer du code propriétaire pour le noyau (drivers, protocoles, etc.), à condition de le compiler sous forme de **module externe**, chargé (automatiquement, par un script) après le *boot*.

Temps réel libre pour système embarqué

Temps réel souple (soft realtime)

Contraintes temporelles en **millisecondes**

Comportement moyen, pas de garantie (*best effort*)

→ **Linux *vanilla***

Contraintes temporelles en **centaines de microsecondes**

Comportement prévu pour gérer les pires circonstances (*worst cases*)

→ Linux avec patch **`PREEMPT_RT`**

Temps réel strict (*hard realtime*)

Non-certifiable

Contraintes temporelles en **dizaines de microsecondes**

Comportement dans le pire des cas vérifiable en pratique mais pas prouvable à cause des millions de lignes de code du noyau Linux sous-jacent.

→ Linux avec extension **Xenomai / Adeos**

Certifiables

Contraintes temporelles en **microsecondes**

Comportement vérifiable (code minimal)

→ **RTEMS, FreeRTOS.**

Temps réel absolu

Contraintes en **dizaines de nanosecondes**

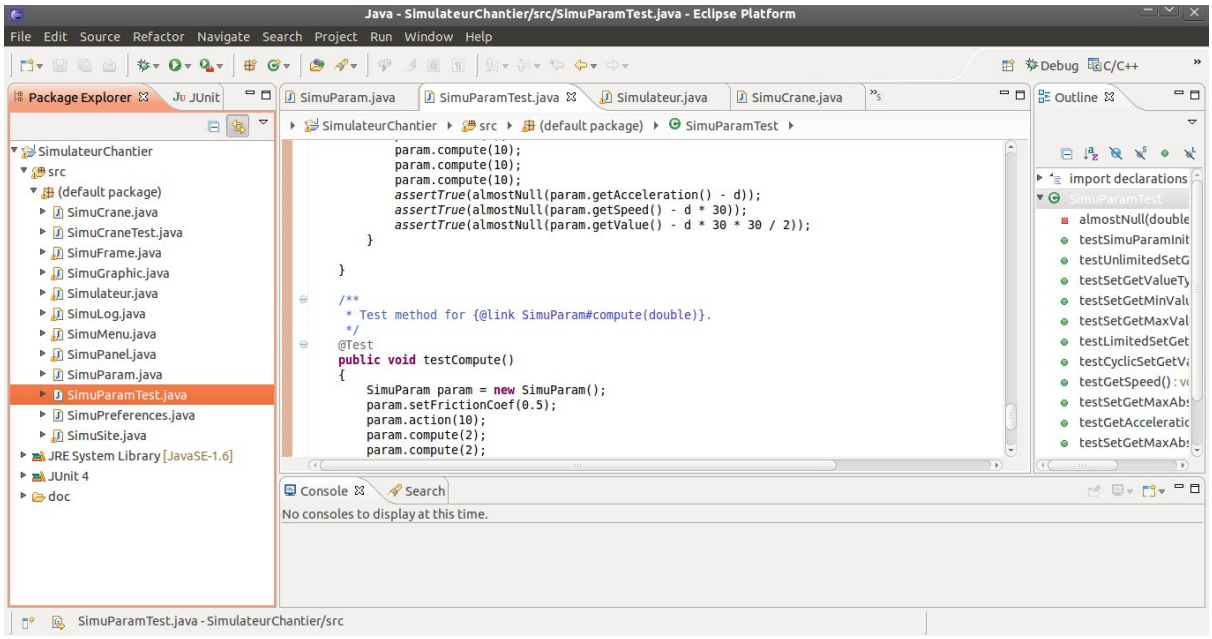
→ **FPGA,** logique **électronique.**

Pour en savoir plus…

« *Raspberry Pi et temps réel* » - Christophe Blaess – Linux Magazine Hors Série n° 75 : « *Raspberry Pi avancé* », Éditions Diamond, novembre 2014.

Environnement de développement

Éclipse



Éclipse est un environnement de développement intégré permettant de gérer des projet Java, C, C++, Python, etc.

Éclipse permet de réaliser du débogage local ou sur une cible distante.

Il existe d'autres environnements de développement sous Linux : Geany (minimal), Code::Blocks, Netbean, Kdevelop, etc.

Prototypage – Projet personnel

Utilisation de **distributions Linux pré-compilées** :

- Raspbian pour Raspberry Pi
- Ubuntu Core
- Arch Linux,
- etc.

Développement du code métier directement sur la cible.

Prototypage rapide par **scripts** (Python par exemple).

Inscription du code métier dans les scripts de démarrage de l'application.

Démonstration, *proof-of-concept* : débogage et mise au point réduits.

Pas de souci d'industrialisation, de déploiement ou de mise à jour pour le moment.

Mise en production

L'**industrialisation** du processus logiciel est cruciale.

Les systèmes de construction d'image (Buildroot, Yocto, etc.) sont préférés aux distributions.

La configuration du noyau est optimisée (temps de boot, occupation mémoire, choix des drivers).

Il faut mettre au point des mécanismes de déploiement et de mise à jour.

Le logiciel doit être sécurisé pour être fiable (non-*brickable*) pour éviter les retours S.A.V.

C'est un bon moment pour mettre en place un environnement d'intégration continue, un système de gestion de version, etc.

Pour des conseils, des informations ou de l'assistance, contactez-nous : ingenierie@logilin.fr.

Conclusion

Plusieurs facteurs sont à prendre en compte lors de la conception d'un système embarqué :

- puissance de calcul, quantité de mémoire, types d'entrées-sorties…
- contraintes physiques (dimensions, poids, autonomie, etc.) ;
- nécessité d'un O.S et choix éventuel ;
- méthodes de développement logiciel, déploiement et mise à jour du code ;
- ...

Les choix peuvent évoluer au cours de la mise au point du projet : la plate-forme utilisée pour le prototype ne sera pas la même que les premières séries ou la production en nombre.

Questions ?

N'hésitez pas à me contacter sur

http://christophe.blaess.fr

ou

http://www.logilin.fr