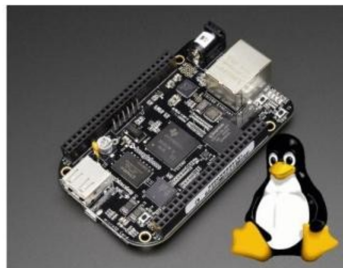
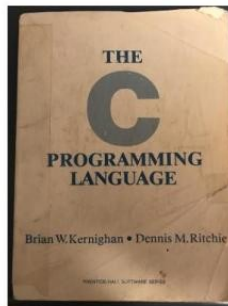
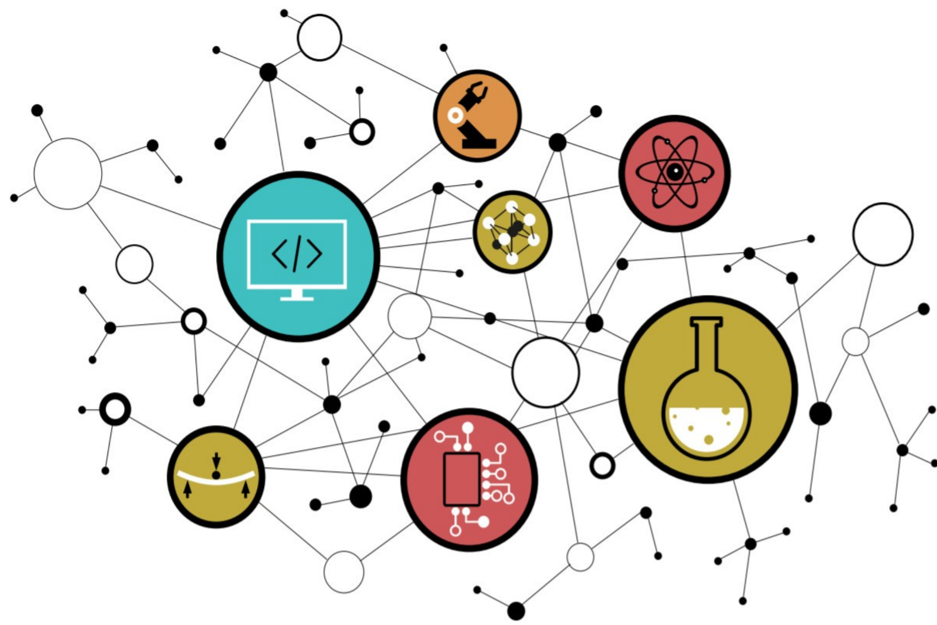
LINUX EMBARQUE

COURS







CONTACTS

Équipe enseignante

hugo descoubes - COURS BSP
hugo.descoubes@ensicaen.fr
+33 (0)2 31 45 27 61

André Lépine - COURS DRIVER
andre.lepine@ensicaen.fr

Dimitri Boudier
dimitri.boudier@ensicaen.fr

ENSICAEN
6 boulevard Maréchal Juin
CS 45053
14050 CAEN cedex 04

RESSOURCES

Les différentes ressources numériques sont accessibles sur la plateforme pédagogique de l'ENSICAEN. Télécharger l'archive complète de travail elinux.zip

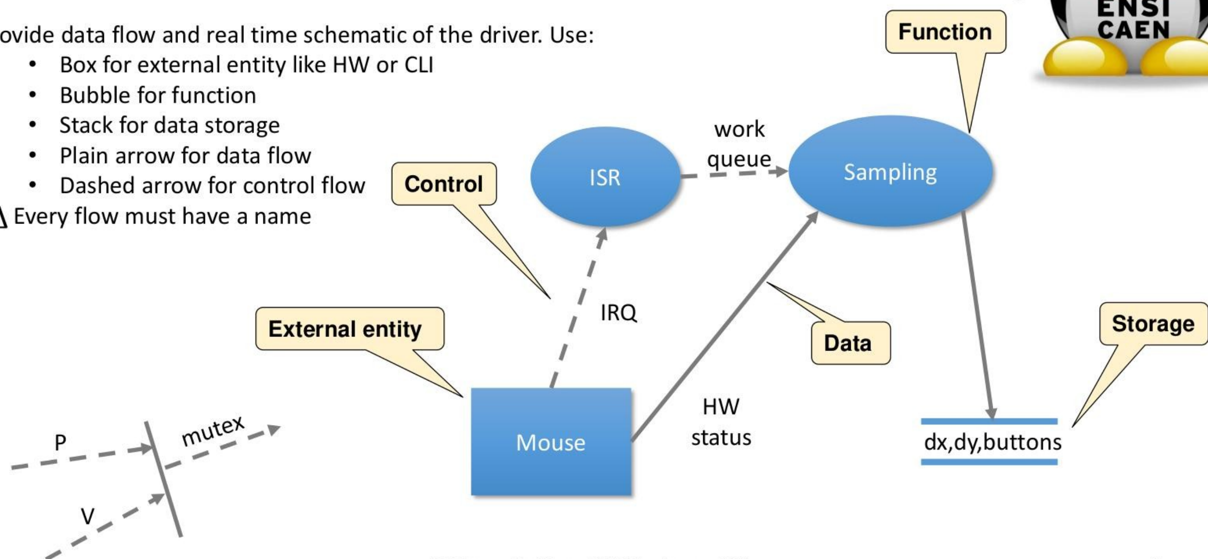
https://foad.ensicaen.fr/course/view.php?id=232

ÉVALUATION

- Examen terminal sur table de 2h :

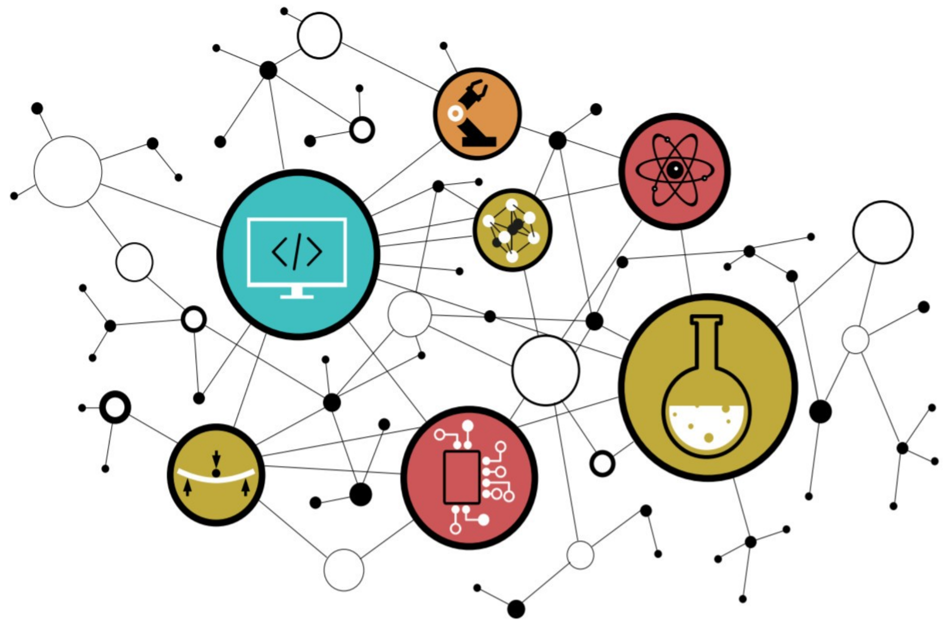
L'évaluation sur table portera sur les séances de Cours Magistral (potentiellement sur tout point présent dans les supports ou présenté à l'oral) ainsi que sur la trame de Travaux Pratiques. S'aider des conseils et exercices présents dans l'archive de travail en ligne (elinux/cm/eval) :

  - 1h30 partie de André : Analyse d'un driver Linux existant et dessin du SART associé



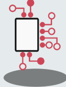
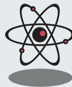
  - 30mn partie de hugo : Questions de culture générale pouvant traiter sur tout point abordé en séance de cours présentiel ou présent dans le support de travail. *Connaissances fondamentales et culture scientifique de l'ingénieur électronicien.*

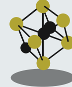
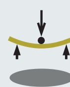
WHAT IS LINUX



Chapitre 1

Le choix Linux

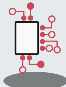
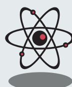



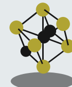
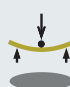
2021-2022

SYSTÈMES D'EXPLOITATION

Historique des systèmes d'exploitation

Noyau d'un OS

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Un **système d'exploitation** ou ***Operating System* (OS)** est un programme qui pilote un système informatique et électronique en contrôlant ses ressources.

Cela inclut :

- Gestion des fonctions périphériques matérielles (drivers ou pilotes)
- Gestion virtuelle de la mémoire vive principale (RAM) par gestion de l'unité de contrôle de la mémoire (MMU)
- Gestion de la mémoire de masse (HDD, SSD, SD, etc) et des fichiers dans une arborescence (*File System*) : organisation, implantation
- Gestion des processus et threads (gestion du/des CPU) : création, ordonnancement, planification, coopération, dépendance, …
- Gestion de la sécurité et des politiques d'accès : multi-utilisateurs, multi-tâches
- etc

En revanche, un OS n'est pas :

- Un logiciel applicatif (traitement de texte, compilateur, navigateur Web, …)
- Une interface graphique (GUI, CLI)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

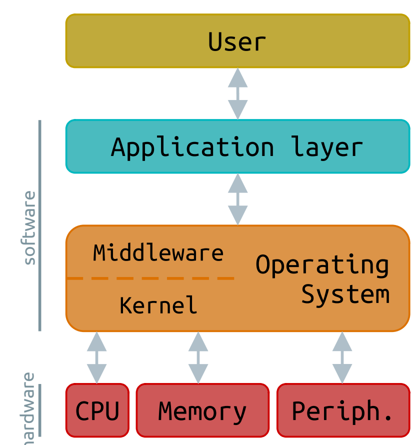
On peut considérer l'OS comme étant un intermédiaire entre les applications logicielles et les ressources matérielles de la machine sur laquelle il fonctionne.

Un OS est un gestionnaire de ressources proposant des services logiciels accessibles depuis les applications :

Gestion des ressources matérielles (CPU, mémoire de masse ou de travail, périphériques, …) pour les attribuer de manière optimale entre les différents processus qui les demandent.

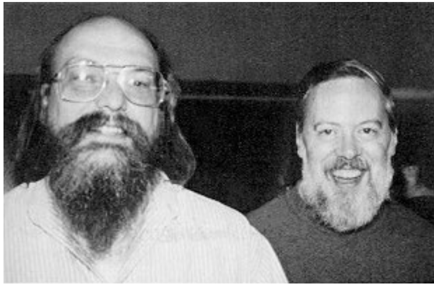
Il peut aussi être vu comme une couche d'abstraction :

Il masque les mécanismes de fonctionnement de la machine dans le but de présenter une interface simple au développeur.

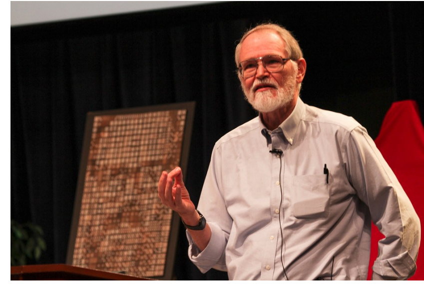


Premiers OS

- Années 40 et 50 : pas d'OS, les systèmes sont mono-tâche et mono-utilisateur
- Années 60 : le MIT crée le premier OS, Multics. Il est multi-programmes, multi-utilisateurs
- Années 70 : UNIX développé par Thompson et Ritchie (Bell Labs), OS pour lequel le C est créé (Kernighan et Ritchie)



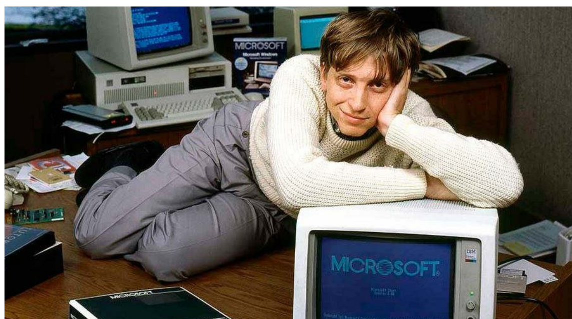
Ken Thompson et Dennis Ritchie.



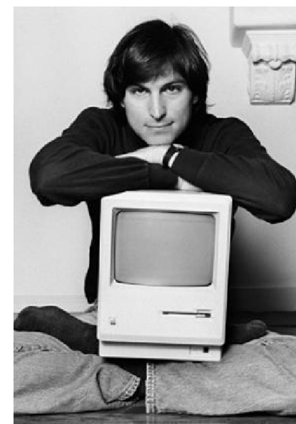
Brian Kernighan,
Hommage à D. Ritchie en 2012 (Bell Labs).

OS propriétaires

- Années 80 : IBM et Microsoft fondent le MS-DOS
- En parallèle : Xerox et Steve Jobs développent Xerox Star, abandonné puis réadapté pour Macintosh



Bill Gates

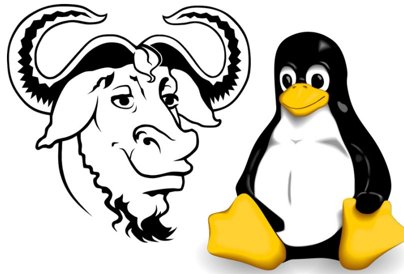


Steve Jobs

OS GNU/Linux

- 1983 : Stallman (MIT) crée GNU. 1er sous licence libre (la GNU GPL) mais c'est un OS sans noyau
- 1991 : Torvalds (Univ. d'Helsinki) développe le noyau Linux (licence GPL)
- 1994 : GNU et Linux s'associent pour donner naissance au premier OS 100 % libre : GNU/Linux

Richard Matthew Stallman



Logos GNU et Linux

Linus Torvalds

Évolutions technologiques

Années 1960 — **Première génération : Système de traitement par lots**

Exécution de grands calculs successifs, peu d'intervention utilisateur

Années 1970 — **Deuxième génération : Systèmes multi-programmés**

Exécution des programmes par *scheduling* (ordonnancement ou planification)

Troisième génération : Systèmes en temps partagé

Années 1980 — Le *scheduling* cherche à répondre aux demandes de plusieurs utilisateurs en communication directe

Années 2000 — **Quatrième génération : Systèmes temps-réel**

L'objectif est de garantir l'exécution des tâches en un temps donné

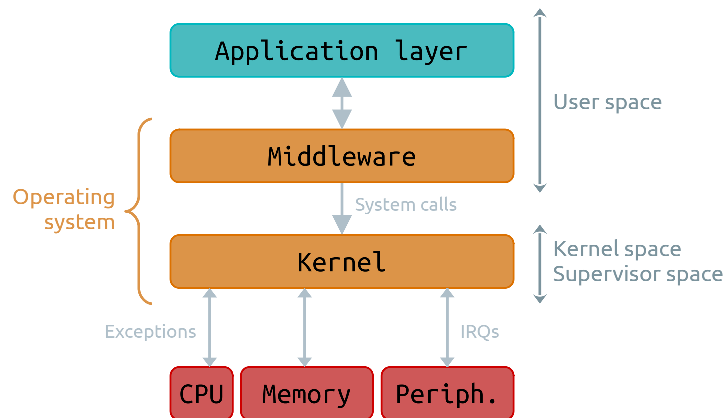
Années 2010 — **Cinquième génération : Système distribué**

Gestion des ressources de plusieurs ordinateurs en simultané, via réseau informatique

Ce groupe de machine est vu comme une unique machine virtuelle, aux capacités importantes

Dans un OS, Le ***kernel*** est l'interface la plus basse du modèle en couches logicielles de l'application.

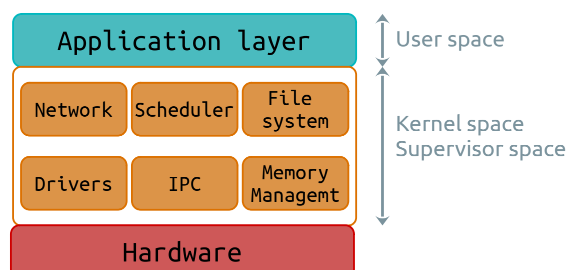
Il propose des services de bas niveaux (*scheduler, drivers, file system, network, memory management*, …) et son espace d'adressage est souvent virtualisé et associé à la notion de niveaux de privilège processeur (mécanisme de protection mémoire).



Il existe différentes familles de *kernels*. Découvrons les 4 principales.

Monolithic kernel

Tous les services bas niveaux évoluent dans le même espace d'adressage et sont encapsulés dans un binaire unique. Ceci implique une forte dépendance des outils systèmes entre eux (un bug dans un driver peu faire tomber le système complet) et une grande difficulté à maintenir de gros systèmes (ex. : GNU/Linux < 1.2 …).

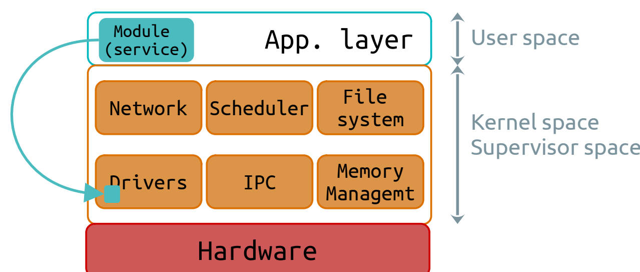


Monolithic modular kernel

Il s'agit de noyaux monolithiques avec une approche modulaire dynamique, impliquant le chargement à chaud de modules *kernel* (*runtime*).

Cette solution permet de n'inclure que les services nécessaires dans l'espace *kernel* puis d'en rajouter à chaud en fonction des besoins (solution modulable très pratique pour des phases de prototypage de drivers).

Quelques exemples : GNU/Linux > 1.2, FreeBSD, Solaris …

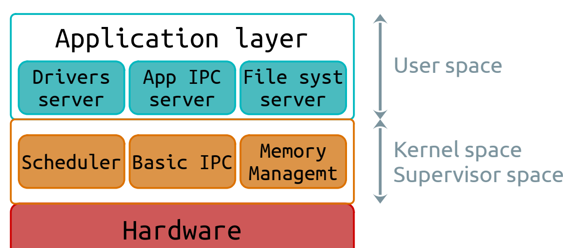


Microkernel

Le kernel n'offre **que les services les plus élémentaires et critiques** dans son espace d'adressage non protégé. Les autres services sont alors répartis dans des serveurs en espace utilisateur possédant leurs propres espaces d'adressages (confinement des défauts).

Il s'agit de solution plus fiable et robuste et surtout plus simple à maintenir … mais moins performantes (énormément d'appels systèmes).

Exemples de systèmes : MINIX, L4, SPARTAN …



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Hybrid kernel

Le grand manque de performance des microkernels à amené la création des noyaux hybrides. Il s'agit de technologies mixtes entre les solutions monolithiques et les microkernels.

Nous pouvons retrouver dans l'espace noyau des services non critiques mais générateurs de grand nombre d'appels système.

Nous retrouvons notamment dans cette famille les OS suivants :

Windows XP, Vista, 7, 8, tous basés sur le noyau hybride NT (New Technology, ex : NT6.3 pour windows 8.1) ;

Mac OS X basé sur le noyau hybride XNU (X is Not Unix) dérivé de Mach kernel (microkernel) et de FreeBSD kernel (monolithic kernel).

LINUX EN BREF

De UNICS à Linux
Le kernel Linux
GNU, distributions GNU/Linux

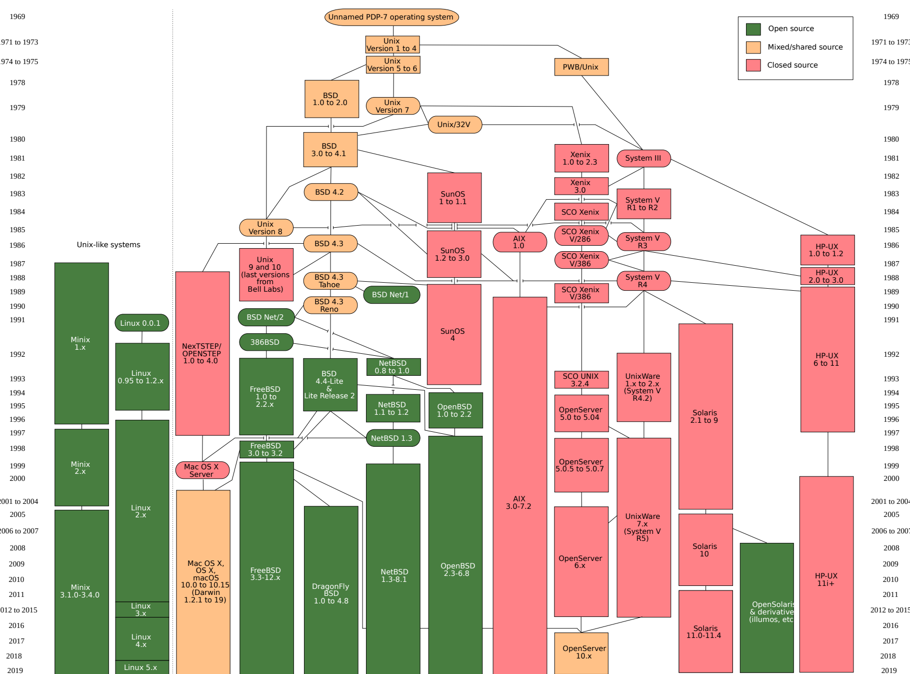
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

UNIX (historiquement nommé UNICS) est un système d'exploitation multi-tâches, multi-utilisateurs (cloisonnement des espaces utilisateurs, *root* a tous les droits) créé en 1969 et basé sur un kernel monolithique.

UNIX est à la base d'un (très) grand nombre de systèmes : nous parlons souvent de famille UNIX ou UNIX-like (GNU/Linux, FreeBSD, Mac OS X, Minix, Solaris …).

Dans cette famille, nous pouvons rencontrer aussi bien des solutions propriétaires que des solutions libres et OpenSource soumises à différents types de licences (GPL, LGPL, Apache, BSD …).

Vous aurez à suivre une conférence propre au monde de l'OpenSource et aux règles de déploiement de solutions logicielles mixtes (propriétaires/libres).



Source : https://fr.wikipedia.org/wiki/Unix

Le kernel Linux est une implémentation libre d'UNIX respectant les spécifications POSIX (norme IEEE 1003).

Il a été créé par Linus Torvalds et est né en réponse à Andrew Tanenbaum (créateur microkernel MINIX) qui ne souhaitait pas intégrer des contributions visant à améliorer MINIX.

Linux a été créé *from scratch* par Torvalds sur un modèle collaboratif décentralisé via Internet et réutilise les composants logiciel du projet GNU.

Pour info, Torvalds se mit à utiliser un logiciel de gestion de version en 2002 seulement. En 2005, il crée Git notamment pour correspondre à la philosophie libre de Linux.

WHAT IS LINUX?

Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of hackers across the Net. It aims towards POSIX and Single UNIX Specification compliance.

It has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management, and multistack networking including IPv4 and IPv6.

ON WHAT HARDWARE DOES IT RUN?

Although originally developed first for 32-bit x86-based PCs (386 or higher), today Linux also runs on (at least) the Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, Cell, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64, AXIS CRIS, Xtensa, Tilera TILE, AVR32, ARC and Renesas M32R architectures.

Linux is easily portable to most general-purpose 32- or 64-bit architectures as long as they have a paged memory management unit (PMMU) and a port of the GNU C compiler (gcc) (part of The GNU Compiler Collection, GCC). Linux has also been ported to a number of architectures without a PMMU, although functionality is then obviously somewhat limited.

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/admin-guide/README.rst?h=v6.0-rc3

Linux est probablement le noyau supportant en 2022 le plus d'architectures matérielles de CPU (tout type de kernels et de domaines confondus).

À titre indicatif, en 2020 le kernel représente environ 27.800.000 (66 492 fichiers et plus de 21000 auteurs) de lignes de code contre 176.250 pour la *release* 1.0.

https://www.kernel.org/

Le modèle de développement, de supervision et de validation des *releases* reste très hiérarchisé. Linus Torvalds supervise les changements de code et des *releases* des dernières versions, pilote toujours une grande partie des commits (28.815 commits soit 6.775.000 de lignes "pushées" à la lui seul) mais délègue néanmoins à d'autres développeurs experts le suivi d'une grande partie du noyau ainsi que la maintenance des *releases* antérieures.

Linux se veut interopérable, modulable et multiplateforme (avec de très nombreuses couches d'abstractions des couches inférieures du système). Tux est sa mascotte :

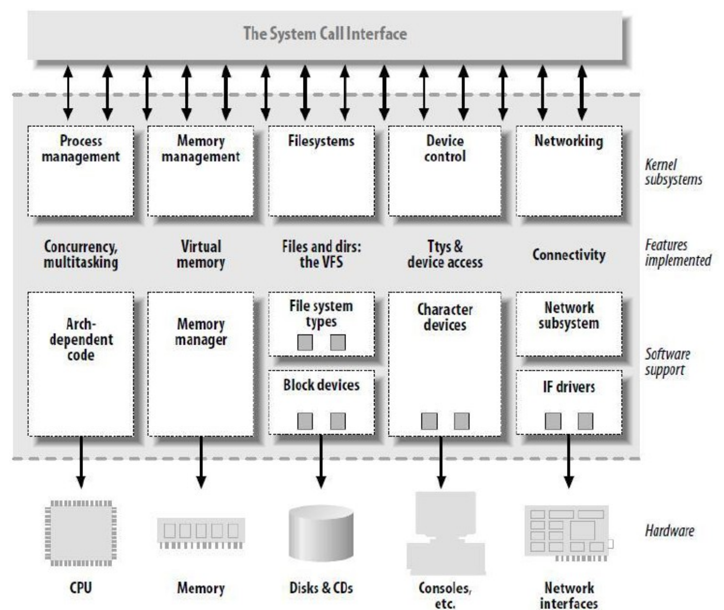
En fin 2020, on estime que depuis 2005 environ 21 000 développeurs issus de plus de 1 000 sociétés différentes ont contribué à l'écriture du kernel Linux.

Observons en 2020, les principales entreprises impliquées dans le développement de Linux : Intel, Red Hat, Huawei, Google, AMD, Linaro, Samsung …

Linux n'est qu'un noyau !

Observons les principaux services qu'il propose,

nous nous pencherons sur les détails plus tard.



Richard Matthew Stallman (rms) est considéré comme le fondateur du mouvement du logiciel libre, avec notamment la Free Software Fondation qu'il crée en 1985.

Il développe **GNU (GNU's Not UNIX)** en 1983, projet depuis repris par le GNU Project.

Avec **GNU**, Stallman cherche à créer une famille de composants logiciels tels qu'un compilateur C (**gcc**), un debugger (**gdb**), des bibliothèques système (**glibc** …), un éditeur de texte (**emacs**), un interpréteur de commande, une ré-implémentation des commandes standards …

GNU est donc un middleware, ou un OS sans kernel !

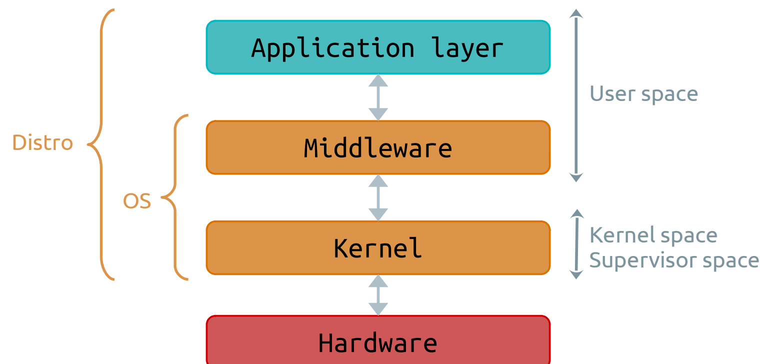
Rappelons qu'à l'époque UNIX n'était pas libre. Il fallut attendre 1990 avant d'avoir une premier kernel GNU nommé Hurd.

Une distribution GNU/Linux est une solution logicielle prête à l'emploi intégrant :

→ le kernel Linux

→ le middleware (GNU, BSD, commandes UNIX-like commandes, shell, GUI …)

→ des outils d'installation, d'administration et de mise à jour du système (`aptitude` sous Debian, `yum` sous Red Hat …).



Il existe un très grand nombre de distributions, sans compter le domaine de l'embarqué. Néanmoins, la grande majorité des distributions existantes descendent des trois suivantes :

Slackware : plus ancienne distribution encore en activité (1993)

Debian : éditée par une communauté de développeurs (1993)

Red Hat : éditée par l'entreprise Américaine du même nom (1994)

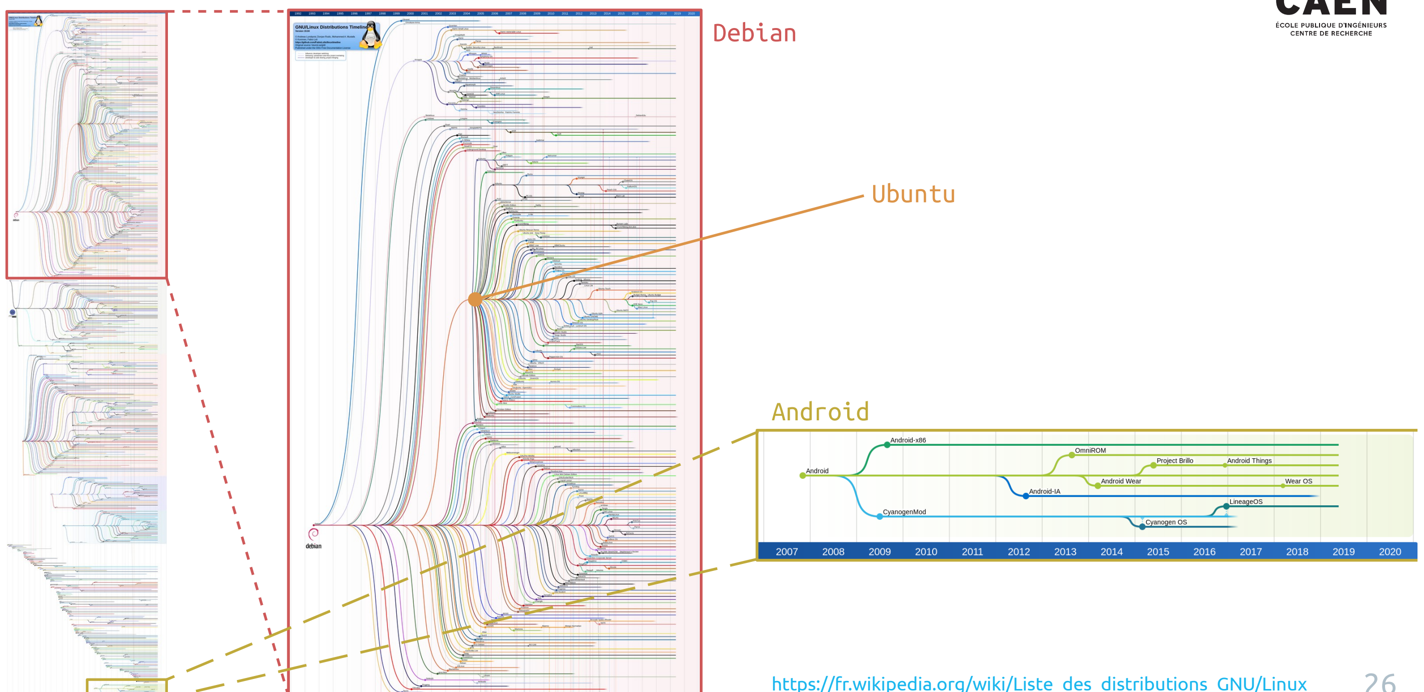
De nombreuses distributions sont dérivées des précédentes.
Voici les plus connues.

Ubuntu : distribution communautaire grand public éditée par Canonical et basé sur Debian. Souvent dépréciée des développeurs chevronnés pour son côté trop « *user friendly* ».

Linux Mint : orientée grand public, basée sur Debian et Ubuntu

Fedora : distribution communautaire supervisée et basée sur Red Hat.

Les autres : SUSE (Novell), Gentoo, ArchLinux …
taper Linux From Scratch sur internet pour les curieux !



https://fr.wikipedia.org/wiki/Liste_des_distributions_GNU/Linux

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Beaucoup de concepts rencontrés dans GNU/Linux sont issues de la philosophie UNIX, qui est largement répandue dans le domaine de l'ingénierie (*The Art of Unix Programming*, Eric S. Raymond).

La plupart de ces règles sont régies par le principe du rasoir d'Ockham ou principe de parcimonie :

Modularité :	Écrire des éléments simples reliés par de bonnes interfaces.
Clarté :	La clarté vaut mieux que l'ingéniosité.
Composition :	Concevoir des programmes qui peuvent être reliés à d'autres programmes.
Séparation :	Séparer les règles du fonctionnement et les interfaces du mécanisme.
Simplicité :	Concevoir pour la simplicité et n'ajouter de la complexité seulement par obligation.
Transparence :	Concevoir pour la visibilité de façon à faciliter la revue et le déverminage.
Robustesse :	La robustesse est l'enfant de la transparence et de la simplicité.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Parcimonie :	Écrire un gros programme seulement lorsqu'il est clairement démontrable que c'est l'unique solution.
Représentation:	Inclure le savoir dans les données, de manière à ce que l'algorithme puisse être bête et robuste.
La moindre surprise :	Pour la conception d'interface, réaliser la chose la moins surprenante.
Silence :	Quand un programme n'a rien d'étonnant à dire, il doit se taire.
Dépannage :	Si le programme échoue, il faut le faire bruyamment et le plus tôt possible.
Économie :	Le temps de programmation est cher, le préserver par rapport au temps de la machine.
Génération :	Éviter la programmation manuelle. Écrire des programmes qui écrivent des programmes autant que possible.
Optimisation :	Prototyper avant de fignoler. Mettre au point avant d'optimiser.
Diversité :	Se méfier des affirmations de « unique bonne solution ».
Extensibilité :	Concevoir pour le futur, car il arrivera plus vite que prévu.

ENSI CAEN ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

À partir de maintenant, vous avez compris ce qu'est Linux, et que dans une discussion non technique « Linux » veut tout et rien dire. Alors précisons.

Techniquement, Linux est un kernel libre.

Mais dans le langage courant, cela désigne n'importe quelle distribution basée sur l'association de GNU et Linux (GNU/Linux).

Clarifions le vocabulaire utilisé pendant ce module :

Kernel	= noyau = Linux
OS	= kernel + Middleware = Linux + GNU
Distribution	= OS + outils d'administration (Ubuntu, Red Hat, Fedora, Raspbian, …)

LINUX POUR L'EMBARQUÉ

Informatique vs Embarqué

Pourquoi Linux

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Le monde de Linux pour l'embarqué est par essence différent du monde du PC.

En effet, le principe même du domaine de l'embarqué est de fournir une solution logicielle spécifique, sur une plateforme matérielle précise (grande hétérogénéité des plateformes et développement en milieu contraint, ressources CPU et mémoire) pour une application particulière …

Ceci est à comparer à l'universalité des solutions logicielles (multiplateformes) dans le monde des ordinateurs (processeurs généralistes) et la philosophie même des systèmes UNIX.

Computer-world Linux

Embedded Linux
faster, lighter, optimized, ...

Le domaine des systèmes embarqué offre une grande richesse d'architectures matérielles différentes (malgré un marché dominé par les architectures ARM) et peut exiger selon les volumes en jeux de développer des applications en milieux contraints. Observons les principales optimisations associées au domaine.

Vitesse d'exécution (CPU)

- temps de boot (bootloader, services réseaux et interfaces I/O minimalistes, décompression noyau, minimiser taille et services kernel et user space …) ;
- options de compilation (architecture dépendant) ;
- exécutables GNU/UNIX et librairies optimisées (Busybox, uClibc …) ;
- conception logicielle (mono-processus, inlining …) ;
- outils de trace et de profilage …

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Empreinte mémoire

- contradictions avec le domaine de l'optimisation CPU (*uninlining*, options de compilation `-Os`, compression kernel et *file system*, librairies partagées …) ;
- taille optimale du kernel et de la distribution (temps de chargement en RAM rapide) ;
- utiliser *initramfs* au lieu de *initrd* ;
- exécutables GNU/UNIX et librairies optimisées (Busybox, uClibc … en désaccord avec la philosophie UNIX) ;
- *stripping* ;
- gestionnaire dynamique de paquets souvent inutile …

Consommation

- option d'alimentation ;
- fréquence de travail du cœur ;
- périodicité de préemption du kernel …

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Coût matériel

Optimiser les aspects précédemment cités permet de jouer grandement sur le design de l'architecture matérielle finale (coût processeur, RAM DDR, *Mass storage* MMC/SDcard, eMMC, …). Aspects non négligeables en fonction des volumes en jeu et marchés ciblés.

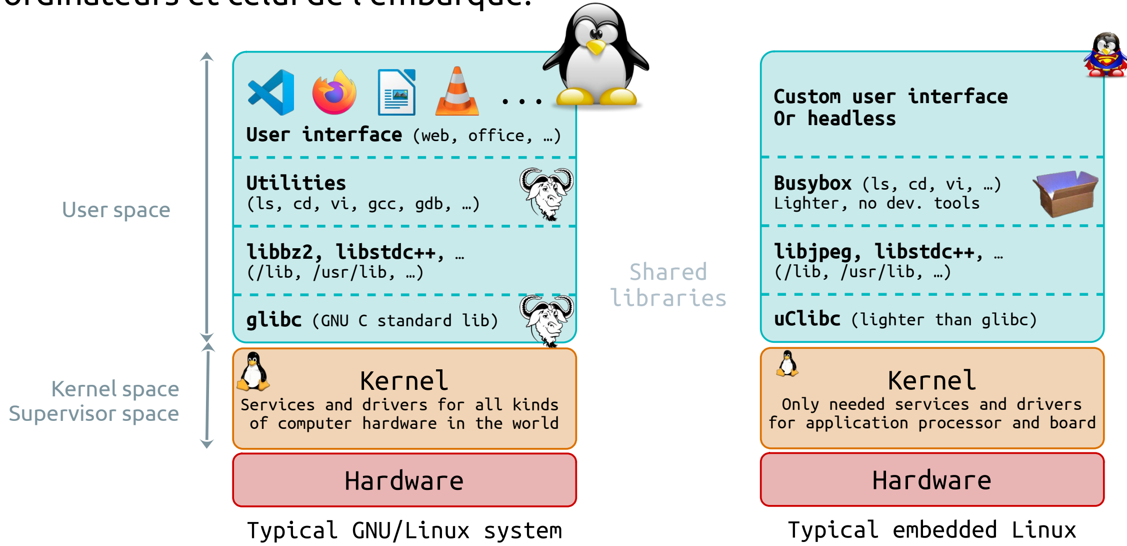
Vitesse d'exécution (CPU)

Permet de travailler avec un processeur à plus faible coût et peut jouer sur la consommation de l'application finale (coût de la batterie et du système d'alimentation de la plateforme).

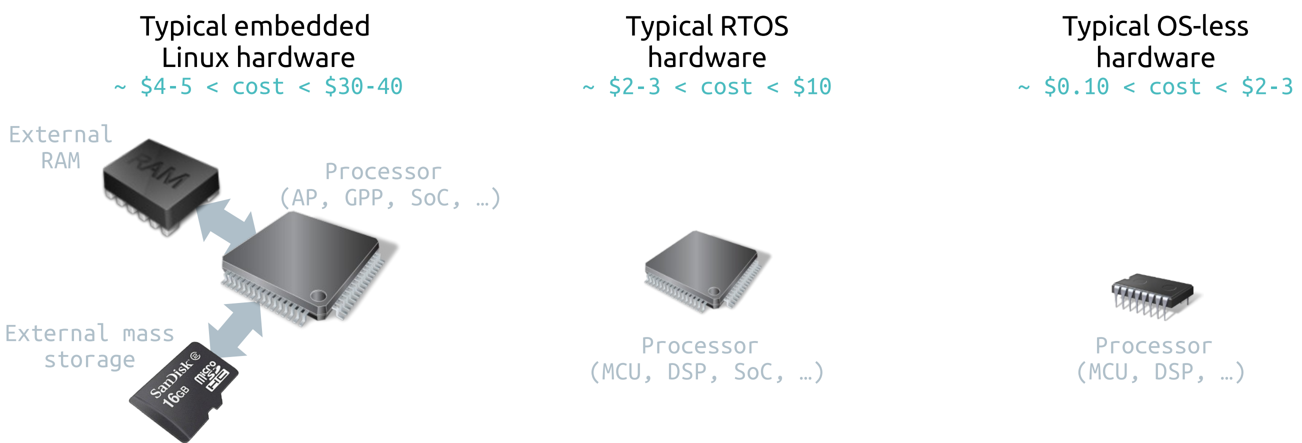
Empreinte mémoire

Moins de *swapping* et un usage réduit de la RAM (mémoire vive et de stockage de masse moins coûteuse), besoin de moins de cache processeur (processeur moins coûteux).

Observons l'architecture typique d'un système GNU/Linux dans le monde des ordinateurs et celui de l'embarqué.



Prenons quelques exemples du coût global d'un processeur associé à sa mémoire (principale et secondaire), sans interfaces de communication, pour une application dans l'embarqué. Les coûts sont donnés à titre indicatif (contre-exemples nombreux).



ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Contraintes des systèmes propriétaires

- Les coûts de license sont élevés, voire très élevés
- L'éditeur du logiciel peut disparaître, et le support technique avec
- Ou alors, paiement d'un surplus pour s'assurer l'accès aux sources
- Logiciel pas sur-mesure (fonctionnalités superflues ou peu adaptées)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Avantages de l'open source

- Redistribution sans royalties
- Disponibilité du code source
- Modification du code source pour adapter une solution sur-mesure
- Pour Linux, la taille du projet assurer une pérennité des solutions sur le très long terme

Et Linux en particulier

- Fiabilité (robustesse, qualité)
- Performances
- Portabilité matérielle (architectures processeurs)
- Adaptabilité (systèmes de fichiers, modularité, …)
- Support technique

Critères de sélection d'un OS,
vu par les pros.

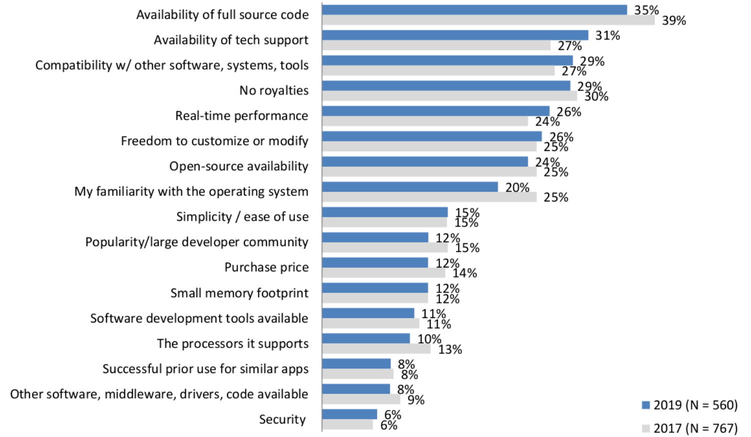
Source :

2019 Embedded Markets Study

Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments March 2019

Presented by : EETimes, embedded

© 2019 AspenCore All Rights Reserved



Part de marché
des OS pour l'embarqué

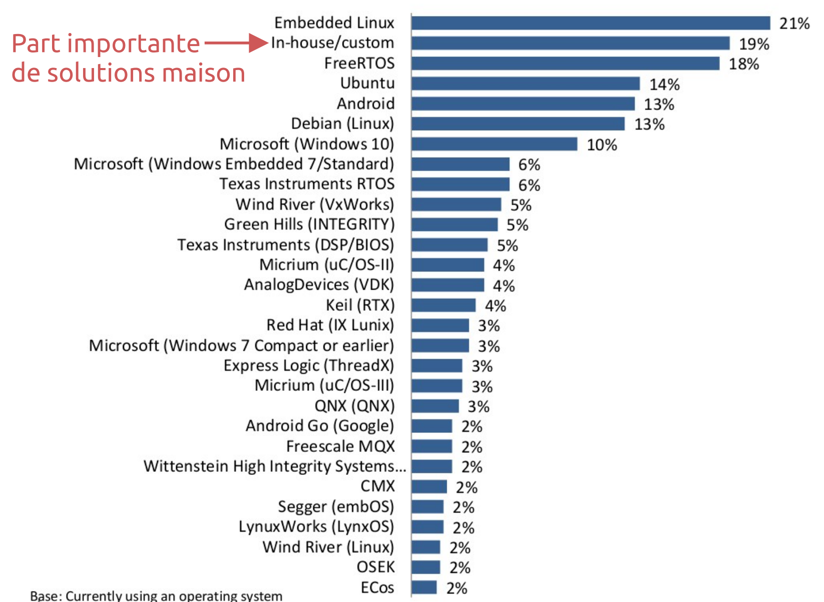
Source :

2019 Embedded Markets Study

Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments March 2019

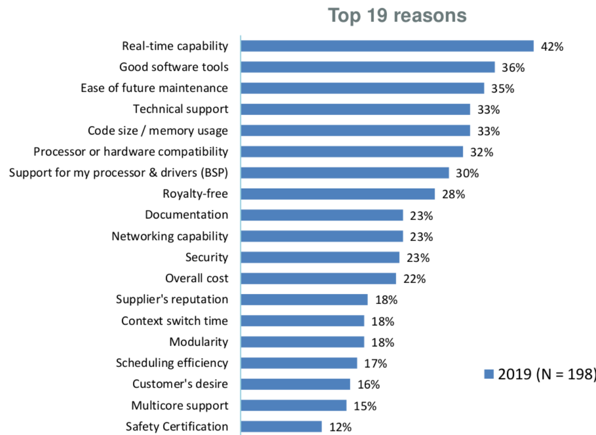
Presented by : EETimes, embedded

© 2019 AspenCore All Rights Reserved

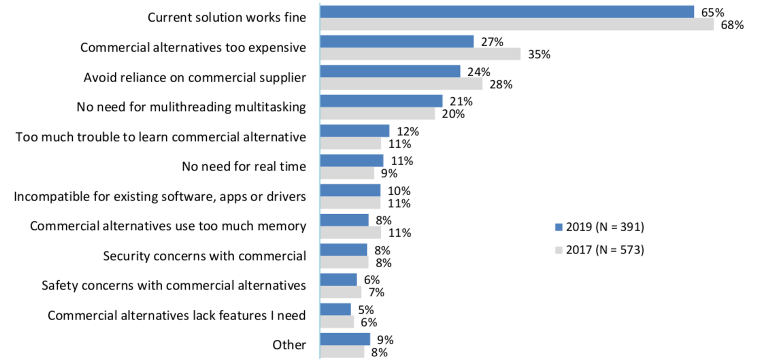


Pour et contre un OS propriétaire

Which factors most influenced your decision to use a commercial operating system?



What are your reasons for not using a commercial operating system?



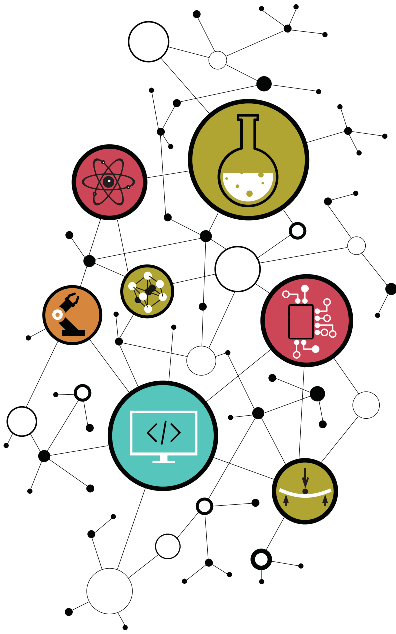
Quand passer sous Linux pour l'embarqué ?

- Quand il faut gérer un système "complexe" (trop complexe pour un RTOS) : Par exemple, pilotage de plusieurs librairies réseau (WIFI, BT, BLE, IP, etc) voire avec une interface graphique évoluée.
- Quand il faut gérer une IHM avancée

Quand éviter Linux ?

- Sur des applications industrielles à fort volume
- Sur des applications avec des contraintes temps réel très lourdes
- Sur des applications faiblement évolutive à architecture système "simple"
- Matériel inadapté (mémoires trop faibles, pas de MMU (sauf uClinux), …)
- Si la license GPL/LGPL ne correspond pas au projet
- Si on doit garantir la conformité à des standards (aéronautique, ...)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE



Sites références concernant le projet Linux (notamment en France) :

http://www.linux.org/ , http://linuxfr.org/ , http://www.linux-france.org/

Archives du kernel Linux :

https://www.kernel.org/

Site TI pour Linux sur architecture OMAP :

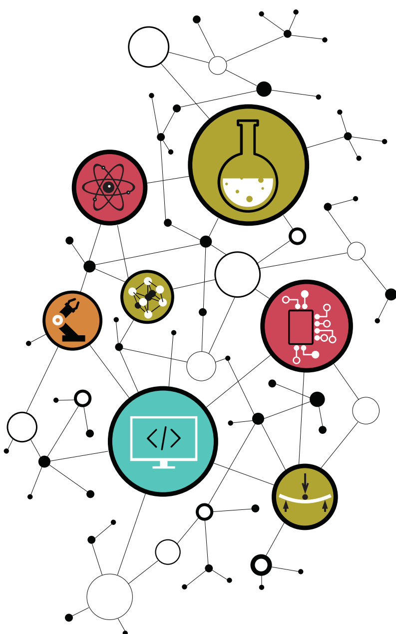
http://linux.omap.com/

bootlin, société française de développement, services et formation autour de Linux embarqué :

https://bootlin.com/

RESSOURCES EN LIGNE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE



Livre Linux Embarqué de Pierre Ficheux et édité par eyrolles :

http://www.eyrolles.com/Informatique/Livre/linux-embarque-9782212134827

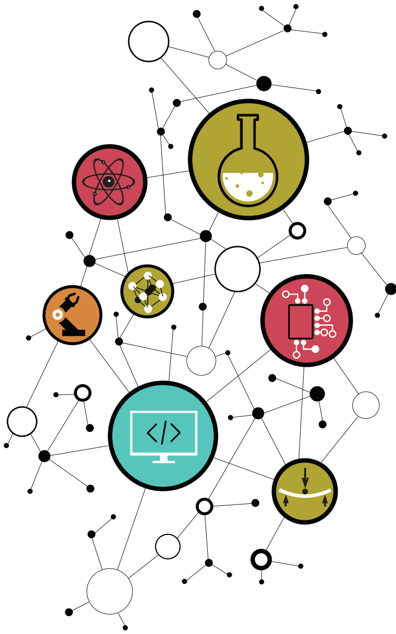
Cours en ligne proposé par l'ENST Bretagne de Jean-Marie Gilliot :

http://public.enst-bretagne.fr/~jgilliot/linux/LinuxEmbarque.html

Cours en ligne proposé par l'ENSEIRB-MATMECA de Patrice Kadionik :

http://kadionik.vvv.enseirb-matmeca.fr/

CONTACT



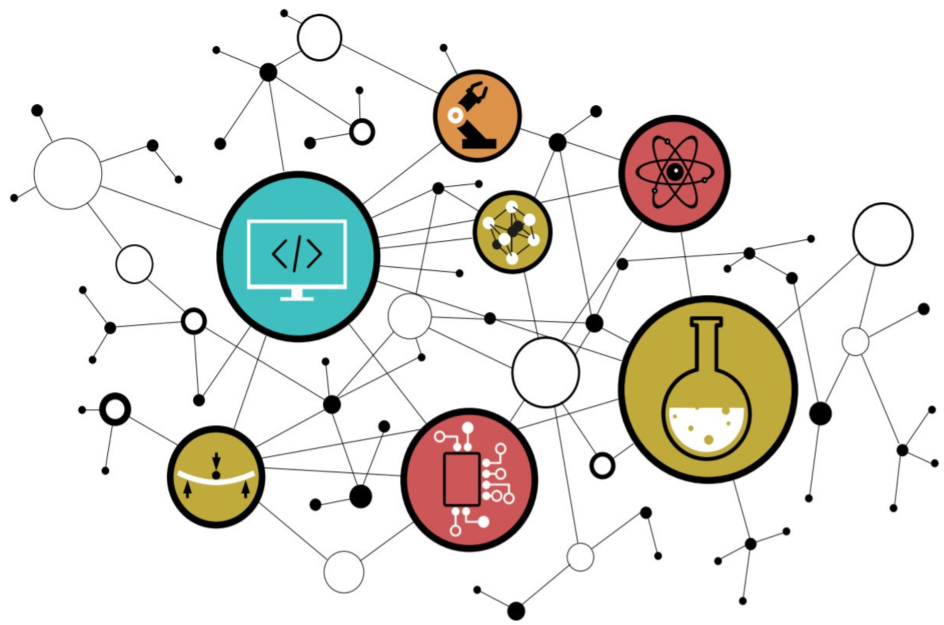
Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)

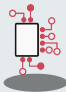
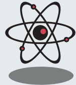
Except where otherwise noted, this work is licensed under
https://creativecommons.org/licenses/by-nc-sa/3.0/

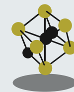
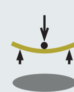
SYSTÈME GNU\LINUX



Chapitre 2

Système GNU/Linux

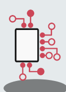
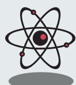



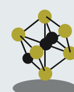
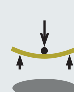



2021-2022

KERNEL LINUX

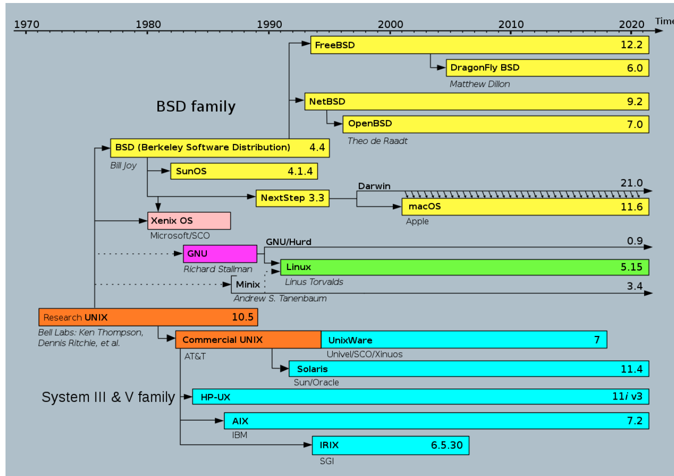
Historique
Composants
Filesystem

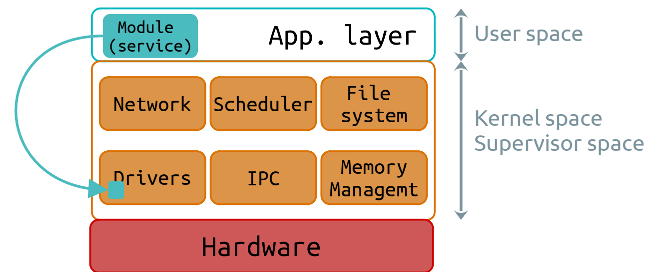



Le kernel Linux est un noyau monolithique modulaire.

Il est développé sous licence GPLv2 en suivant un modèle collaboratif décentralisé via internet. Les sources sont librement accessibles sur www.kernel.org .



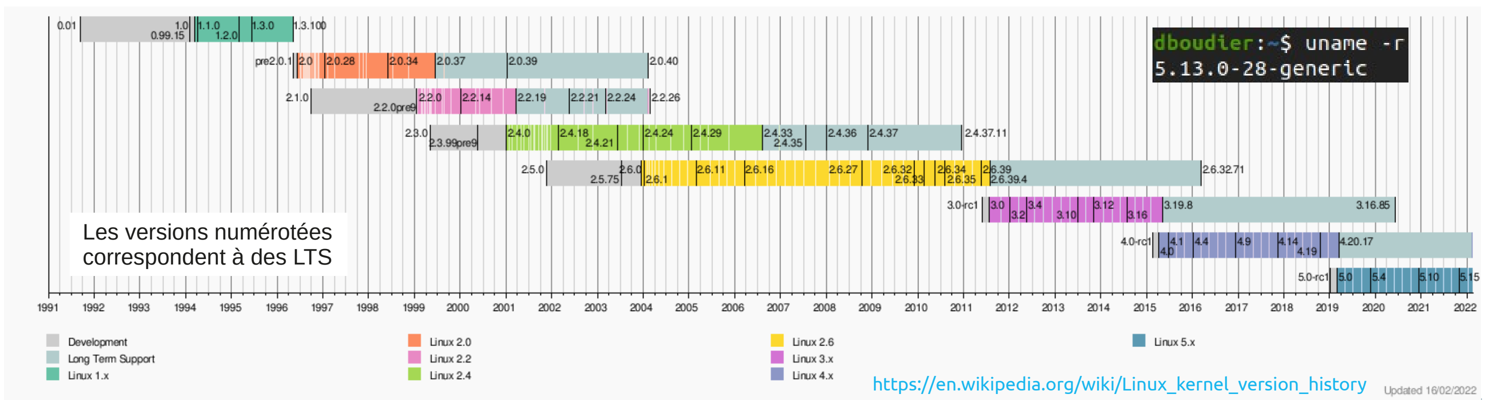


https://en.wikipedia.org/wiki/Linux#/media/File:Unix_timeline.en.svg

Le développement Linux a été lancé en 1991 par Linus Torvalds (8 ans après GNU).

En 2020, le projet comptait 27,8 millions de lignes de code. En 2020 on dénombrait en tout 887.925 commits avec près de 21.000 contributeurs au total.

Aujourd’hui (2022), Torvalds reste le *maintainer* des versions majeures (*-rc* et *mainline*).



https://en.wikipedia.org/wiki/Linux_kernel_version_history

Git et Linux

Pour faciliter son développement, le projet Linux s'appuie dès 2002 sur Bitkeeper, un logiciel de gestion de version. Mais ce logiciel propriétaire devint payant trois ans après, ce qui contraint les contributeurs à lâcher le projet Linux.

Constatant le manque de solutions libres, Torvalds lance alors en 2005 le projet Git : l'objectif principal est de répondre aux besoins du développement du noyau Linux.

En deux semaines, Git fait accomplit ses premiers *merges*.
En deux mois, Git est finalisé et héberge le kernel Linux 2.6.12.

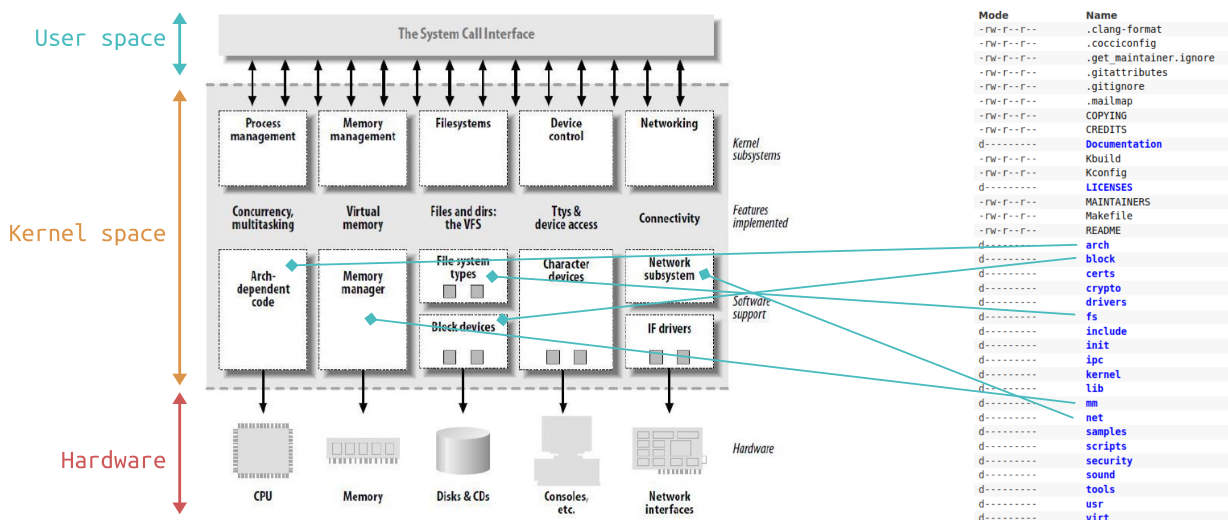
D'après Torvalds :

```
"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."
```


Services fournis

Jouant son rôle de kernel, Linux propose des services bas niveaux :

Ordonnanceur, gestion des systèmes de fichiers, pilotes de périphériques (LDD), gestion/protection mémoire, ...



Mode	Name
-rw-r--r--	.clang-format
-rw-r--r--	.cocciconfig
-rw-r--r--	.get_maintainer.ignore
-rw-r--r--	.gitattributes
-rw-r--r--	.gitignore
-rw-r--r--	.mailmap
-rw-r--r--	COPYING
-rw-r--r--	CREDITS
d---------	Documentation
-rw-r--r--	Kbuild
-rw-r--r--	Kconfig
d---------	LICENSES
-rw-r--r--	MAINTAINERS
-rw-r--r--	Makefile
-rw-r--r--	README
d---------	arch
d---------	block
d---------	certs
d---------	crypto
d---------	drivers
d---------	fs
d---------	include
d---------	init
d---------	ipc
d---------	kernel
d---------	lib
d---------	mm
d---------	net
d---------	samples
d---------	scripts
d---------	security
d---------	sound
d---------	tools
d---------	usr
d---------	virt

Les fichiers d'en-tête du kernel Linux sont généralement accessibles depuis une machine host, dans le répertoire `/usr/src/linux-headers-<version_no>/`. Ceci permet de compiler du code applicatif pouvant communiquer avec les interfaces logicielles du noyau (appels système)

(C'est Ubuntu qui récupère les sources du kernel et les place dans ce dossier)

```
dboudier:/usr/src$ ls
linux-headers-5.11.0-46-generic  linux-headers-5.8.0-44-generic    linux-hwe-5.8-headers-5.8.0-44
linux-headers-5.13.0-27-generic  linux-headers-5.8.0-63-generic    linux-hwe-5.8-headers-5.8.0-63
linux-headers-5.13.0-28-generic  linux-hwe-5.11-headers-5.11.0-46  tty0tty-1.3
linux-headers-5.4.0-59           linux-hwe-5.13-headers-5.13.0-27  virtualbox-6.1.26
linux-headers-5.4.0-59-generic   linux-hwe-5.13-headers-5.13.0-28
linux-headers-5.8.0-43-generic   linux-hwe-5.8-headers-5.8.0-43
dboudier:/usr/src$
dboudier:/usr/src$
dboudier:/usr/src$ cd linux-headers-5.13.0-28-generic/
dboudier:/usr/src/linux-headers-5.13.0-28-generic$ ls
arch   crypto         fs       ipc      kernel    mm              samples   sound   usr
block  Documentation  include  Kbuild   lib       Module.symvers  scripts   tools   virt
certs  drivers        init     Kconfig  Makefile  net             security  ubuntu
```

`linux-headers-<version_no>-generic` = version stable du kernel

`linux-hwe-<version_no>` = Hardware Enablement = version kernel spécifique au matériel utilisée par Ubuntu

Observons succintement les différents répertoires du kernel (www.kernel.org).

arch/ Architecture. Parties spécifiques aux architectures CPU et plateformes supportées ainsi que certains services propres aux coeurs (core DMA, cache, timer, vecteur d'interruptions, board support, device tree, …)

Il s'agit du second plus gros répertoire de l'arborescence.

```
dboudier:/usr/src/linux-headers-5.13.0-28-generic/arch$ ls
alpha  arm    csky   hexagon  Kconfig  microblaze  nds32  openrisc  powerpc  s390  sparc  x86
arc    arm64  h8300  ia64     m68k     mips        nios2  parisc    riscv    sh    um     xtensa
```

Le répertoire `arch/<arch>/boot/` contient les premiers fichiers (en assembleur) lancés au démarrage du kernel

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

`Documentation/` Documentation du kernel et sous-services proposés (drivers, ...).

`drivers/` Pilotes de périphériques, (GPIO, I²C, CAN, USB, … multiplateformes).

Rappelons que Linux reste un kernel monolithique, tout service matériel supporté par la mainline doit pouvoir être ajouté à la compilation du noyau.

Il s'agit du répertoire le plus volumineux de l'arborescence.

`fs/` *File system*.

Systèmes de fichiers réels et virtuels supportés (ext4, FAT, NTFS, …).

`include/` Principaux fichiers d'en-tête pour compilation du noyau.

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

`init/` Fichiers d'amorçage et d'initialisation système (`main.c`, `initramfs.c`, …) indépendants de l'architecture.

`ipc/` *Inter-Process Communication*.

Mécanismes de communication inter-processus (mémoire partagée, message queue, sémaphores, …).

`kernel/` Cœur du noyau (scheduler, signaux UNIX, …).

`lib/` *Libraries*.

Petite bibliothèque C interne au kernel (algorithmes de décompression, manipulation de string "`strcmp`, `strlen`, …", …).

`mm/` *Memory management.*

Mécanismes de gestion et protection mémoire (segmentation, pagination, swap, …).

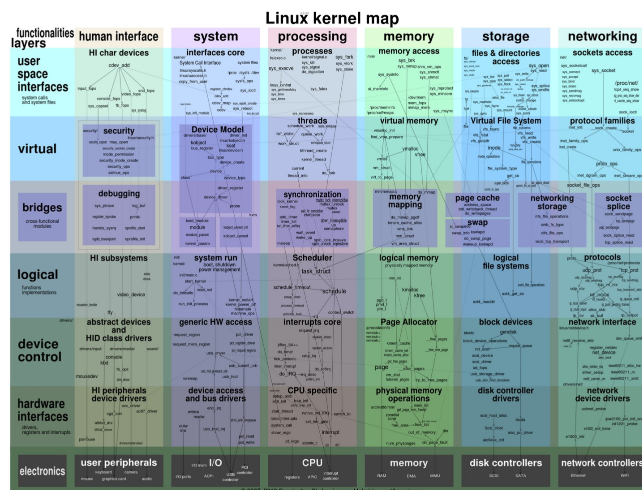
`net/` *Network.*

Interfaces non-architecture-dépendantes et protocoles réseaux (Ethernet, IPv4, IPv6, CAN, …).

`scripts/` Code sources des outils de configuration et de compilation du kernel.

`tools/` Utilitaires de prototypage et d'analyse du kernel et de l'architecture cible (profilage, trace, consommation, échauffement thermique, …).

Afin d'assurer un bonne interopérabilité, maintenabilité et portabilité, le kernel Linux est découpé en couches proposant plusieurs niveaux d'abstraction, notamment au regard du matériel. Observons le *kernel map* du noyau :



https://makelinux.github.io/kernel/map/

GNU

Historique
Composants
Filesystem

GNU est un projet de système d'exploitation lancé en 1983 par Stallman, avec pour objectif principal de fournir un OS libre et collaboratif. GNU repose sur une implémentation libre de l'OS UNIX (1969), ce dernier étant à la fois populaire et modulaire (donc facile à réimplémenter morceau par morceau).

Notons que l'acronyme GNU signifie "GNU's Not UNIX".

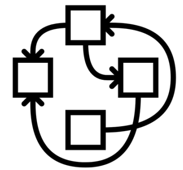
En quelques années, GNU fournissait un certain nombre de services (application ou librairie) :

- GCC (Initialement *GNU C Compiler*, maintenant *GNU Compiler Collection*)
- Emacs (éditeur de texte), un shell, des bibliothèques
- gdb, make, coreutils, binutils, libc, bash, grub, nano, grep, gimp, etc

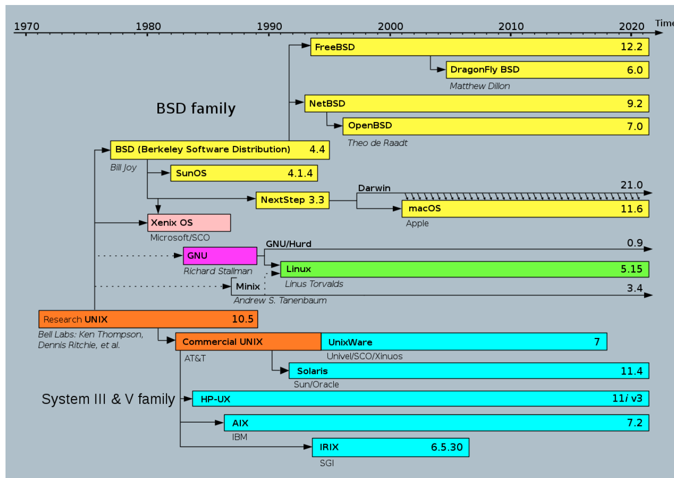
Mais il manquait toujours l'élément central de l'OS : **le noyau**

https://www.gnu.org/
https://www.gnu.org/software/software.html

En 1990, *The GNU Project* lance la production de son noyau nommé Hurd.



Basé à l'origine sur BSD 4.4, puis sur Mach, Hurd est toujours en cours de développement, soutenu par la *Free Software Fundation* et *The GNU Project*.



En 2012, *The GNU Project* décide d'intégrer officiellement un fork de Linux dans son projet : Linux-libre.

Sans code propriétaire, sans binary blob.

On parle alors de système **GNU/Linux**.

Un système GNU signifiant GNU (dont Hurd).

NB : les distributions utilisent majoritairement GNU/Linux, mais The GNU Project incite à utiliser GNU/Linux-libre.

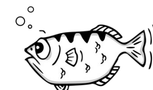
https://en.wikipedia.org/wiki/Linux#/media/File:Unix_timeline.en.svg

GNU est composé uniquement de logiciels libres (ce qui est le cas pour GNU/Linux-libre, mais pas GNU/Linux et distributions dérivées).

Les principaux outils proposés par GNU sont :

`GCC` GNU Compiler Collection (initialement *GNU C Compiler*)

`GDB` GNU Debugger



`binutils` GNU Binary Utilities

`glibc` GNU C library

`coreutils` GNU Core Utilities (`cat`, `ls`, `mv`, `rm`, …)

`Bash` Bourne Again Shell

`GRUB` Grand Unified Bootloader

Liste des 373 packages GNU ici : https://directory.fsf.org/wiki/GNU

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Les systèmes GNU/Linux utilisent par convention le *Filesystem Hierarchy Standard* (FHS), mais d'autres variantes d'UNIX l'utilisent également.

/ Le répertoire racine du système de fichiers (hiérarchie principale).

/home Répertoire des dossiers des utilisateurs (fichiers personnels).
`/home/user1`, `/home/user2`, …

/root Répertoire `home` spécifique au super-utilisateur (`root`).

GNU
FHS – Filesystem Hierarchy Standard

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

/boot Fichiers de démarrage (image des kernels, `System.map`, `initrd`).

/etc *Editable Text Configuration*. Fichiers de configuration du système.

/opt *Optional*. Répertoire pour l'installation de logiciels externes.

/var Stockage de données variables, i.e. fichiers dont le contenu est censé varier au fil de l'exécution du système (*logs*, *lock files*, *spool*, …).

/proc Système de fichiers virtuel contenant sous forme de fichier les informations des processus et du kernel. Répertoire forcément dynamique.

/proc/<pid> Contient tous les paramètres du processus désigné.

```
dboudier:/proc$ ls
1     114   1240  14    1522  170    17354  181    1927   20056  20577  2091  220   27    313   394   47    55    6149  6766  78   904  98             filesystems  meminfo         sysvipc
10    1140  1242  1401  1543  171    17355  18116  1932   20095  20590  2094  222   28    3138  4     48    5512  62    68    780  906  acpi           fs           misc            thread-self
100   1145  1244  1402  1549  172    17356  182    1935   20122  20614  2097  2221  282   316   40    49    5582  6336  6965  783  908  asound         i8k          modules         timer_list
1002  1146  1247  1403  1556  1723   17357  183    1938   20124  20664  2099  2267  2830  32    402   50    56    636   6979  79   909  bootconfig     interrupts   mounts          tty
101   1147  1248  1426  1561  173    17358  18440  1940   2013   20689  2101  2274  29    320   41    52    5610  637   7     790  91   buddyinfo      iomem        mtrr            uptime
102   1148  1253  1435  1565  17339  17359  185    1953   20147  2069   2105  2293  298   34    416   521   5665  638   70    80   910  bus            ioports      net             version
103   1149  1257  146   1570  17340  17360  1852   1956   20149  20691  2107  23    299   340   42    522   5732  639   71    82   912  cgroups        irq          pagetypeinfo    version_signature
104   115   1264  1466  1574  17341  17361  1867   1961   2015   20719  2108  2315  3     341   423   523   5736  64    7123  83   913  cmdline        kallsyms     partitions      vmallocinfo
106   1150  1268  1467  1581  17344  1741   1883   1967   20191  20747  2109  2393  30    346   426   524   58    640   72    84   914  consoles       kcore        pressure        vmstat
107   1151  127   1481  1590  17345  1746   18859  197    20199  20755  2113  24    300   35    427   526   59    641   724   85   917  cpuinfo        keys         schedstat       zoneinfo
108   1152  1279  1482  1596  17346  175    18882  198    20217  20788  2115  2401  302   357   43    527   5958  646   73    86   918  crypto         key-users    scsi
109   1153  1283  15    16    17347  1754   18905  19876  20228  20801  2117  2402  304   36    44    5273  6     6472  733   88   92   devices        kmsg         self
11    116   1284  1507  164   17348  177    18918  19946  2025   20816  212   2405  305   37    4415  5278  60    65    734   886  920  diskstats      kpagecgroup  slabinfo
110   1167  1285  1509  165   17349  1774   19     19950  20372  2086   2121  2413  306   38    4418  528   6014  6544  735   887  921  dma            kpagecount   softirqs
1103  1168  1286  1512  1656  17350  178    190    1998   20382  2087   2133  2443  307   382   46    53    607   66    739   888  94   driver         kpageflags   stat
111   12    13    1516  166   17351  179    1909   2      20388  2088   2137  25    308   385   467   5314  61    67    74    89   95   dynamic_debug  loadavg      swaps
112   1235  1308  1519  169   17352  18     1911   20     2040   2089   2143  2571  309   386   468   5353  613   6712  76    9    96   execdomains    locks        sys
113   1236  1396  1521  17    17353  1807   1919   20024  205    209    22    26    31    391   469   54    614   6742  77    90   97   fb             mdstat       sysrq-trigger
```

En vrac : `ps -Af` `ls /proc` `ls -l /proc/<pid>` `cat /proc/<pid>/status` `cat /proc/cpuinfo` `cat /proc/modules` `cat /proc/filesystems` `cat /proc/meminfo`

Voir /Documentation/filesystems/proc.rst

/dev *Devices files*. Liste des périphériques sous forme de fichiers (*device nodes*).

/media Points de montage pour les supports amovibles (clé USB, CD, DD externe, …).

/mnt Points de montage temporaire pour les systèmes de fichiers.

/sys Système de fichiers virtuel contenant sous forme de fichier les informations du matériel et des pilotes.

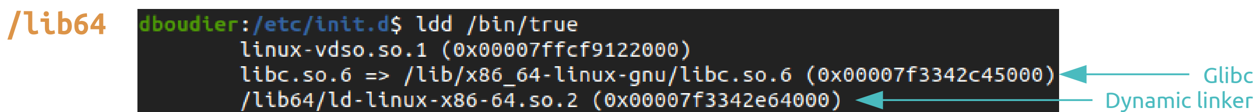
Par opposition à /dev, /sys est géré dynamiquement.

/sys est pour le matériel ce que /proc est pour le logiciel.

/bin Binaires des commandes utilisateurs (`apt`, `cat`, `cd`, `cp`, `ls`, `mv`, …).
Le FHS impose une liste minimale de commandes.

/sbin Binaires des commandes systèmes (`init`, `route`, … réservées au *super-user*).
Certaines sont en accès restreint aux autres utilisateurs (`ifconfig`, …).

/lib Bibliothèques partagées pour les binaires de `/bin` et `/sbin`.
/lib32 La commande `ldd <binary>` indique les bibliothèques utilisées par le binaire.
/lib64



/usr USR = **UNIX Shared Resources** (et non "~~User~~").
Hiérarchie secondaire (sous-répertoires) en lecture seule et accès partagé.

/usr/bin Binaires des commandes non-essentielles.

/usr/sbin Binaires des commandes systèmes non-essentielles .

/usr/lib Librairies pour les répertoires `/usr/bin` et `/usr/sbin` .

/usr/src Fichiers sources des kernels.

```
dboudier:/$ ls -l
total 2097240
lrwxrwxrwx   1 root root          7 août  31  2020 bin -> usr/bin
drwxr-xr-x   4 root root       4096 févr. 18 11:08 boot
drwxr-xr-x   2 root root       4096 août  31  2020 cdrom
drwxr-xr-x  24 root root       5680 févr. 18 10:07 dev
drwxr-xr-x 150 root root      12288 févr. 18 11:08 etc
drwxr-xr-x   4 root root       4096 avril 15  2020 home
lrwxrwxrwx   1 root root          7 août  31  2020 lib -> usr/lib
lrwxrwxrwx   1 root root          9 août  31  2020 lib32 -> usr/lib32
lrwxrwxrwx   1 root root          9 août  31  2020 lib64 -> usr/lib64
lrwxrwxrwx   1 root root         10 août  31  2020 libx32 -> usr/libx32
drwx------   2 root root      16384 août  31  2020 lost+found
drwxr-xr-x   3 root root       4096 janv. 18  2021 media
drwxr-xr-x   2 root root       4096 juil. 31  2020 mnt
drwxr-xr-x  11 root root       4096 janv. 14 18:04 opt
dr-xr-xr-x 460 root root          0 févr. 18 10:07 proc
drwx------  18 root root       4096 janv. 14 18:05 root
drwxr-xr-x  39 root root       1100 févr. 18 11:14 run
lrwxrwxrwx   1 root root          8 août  31  2020 sbin -> usr/sbin
drwxr-xr-x  26 root root       4096 sept. 22 10:33 snap
drwxr-xr-x   2 root root       4096 juil. 31  2020 srv
-rw-------   1 root root 2147483648 janv. 11 10:26 swapfile
dr-xr-xr-x  13 root root          0 févr. 18 10:07 sys
drwxrwxrwt  26 root root      12288 févr. 18 11:28 tmp
drwxr-xr-x  14 root root       4096 oct.  13  2020 usr
drwxr-xr-x  14 root root       4096 juil. 31  2020 var
```

Notons que certains distributions (ici Ubuntu) prennent la liberté d'effectuer un lien symbolique (symlink) entre différents répertoires.

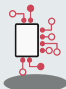
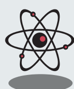
BOOTLOADER

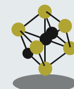
Chargeur d'amorçage

Firmwares

Bootloaders

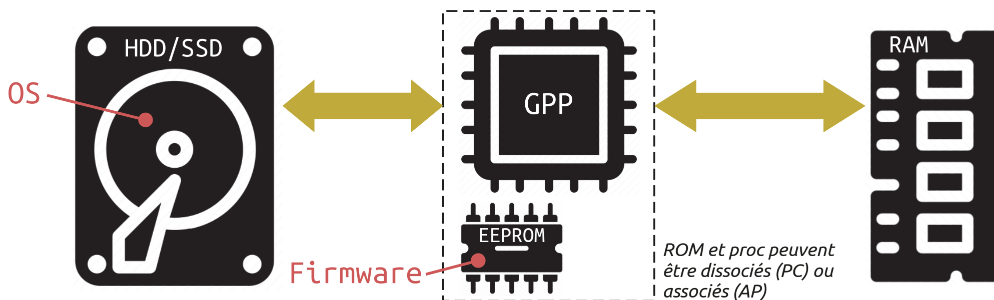
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE



Un système d'exploitation a vocation a être modifié régulièrement, que ce soit par l'utilisateur ou par l'entreprise qui en assure la distribution et/ou le support. Il est donc judicieux de déposer l'OS sur une mémoire de masse externe au processeur (disque dur, Flash, clé USB, …).

Par opposition, un *firmware* est censé être figé pour la durée de vie du système. Il est intégré à la ROM interne (ou associée) du processeur.



Au démarrage d'un ordinateur, il faut alors charger le système d'exploitation depuis la mémoire de stockage de masse (HDD, SSD, SD, etc) vers la mémoire de travail (RAM).

C'est le rôle du *bootloader* ou chargeur d'amorçage.

Cette fonction ne peut pas être effectuée par un OS standard du marché, puisque la stratégie de boot (amorçage) peut évoluer dans le temps en fonction du besoin de l'utilisateur (multi-boot, multi-OS, média de masse HDD, SSD, USB ou réseau, …).

Le système d'amorçage peut être constitué de plusieurs étages de bootloader. Les premiers étages de boot sont très primitifs (peu de services inclus) et peuvent potentiellement se trouver sur des médias physiques séparés (BIOS, UEFI, sur ROM, Flash, MCU externes). Les niveaux de boot évolués (GRUB, U-Boot, Barebox, etc) se trouve en général sur le même média que l'OS.

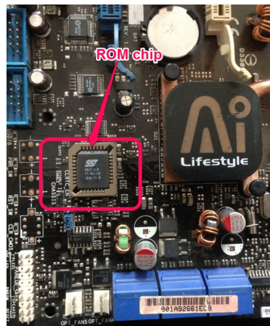
Le firmware charge le bootloader en RAM, puis ce dernier chargera l'OS en RAM.

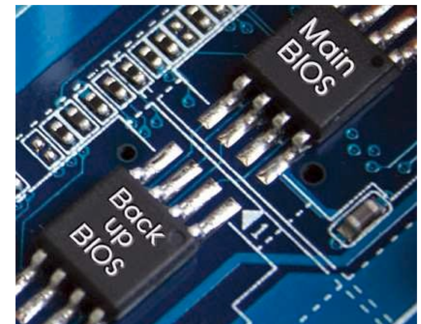
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Le BIOS (*Basic Input Output System*) est un firmware très répandu sur les cartes mères.

Outre l'initialisation des composants et l'identification des périphériques, il a pour mission la lecture du MBR (*Master Boot Record* ou secteur d'amorçage) d'un disque.

Parmi les informations contenues dans le MBR se trouve l'adresse du bootloader.





ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Le successeur du BIOS est l'UEFI (*Unified Extensible Firmware Interface*).

Il offre évidemment plus de fonctionnalité que le vieux BIOS écrit en assembleur, mais supporte ce standard pour satisfaire des OS anciens.

L'UEFI est capable de travailler avec le réseau, d'utiliser une IHM de meilleure qualité et de supporter des disques de taille supérieure à 2,2 To.

En ce qui nous intéresse (chargement du bootloader), l'UEFI supporte le système de partitionnement GPT (*Globally unique identifier Partition Table*) en plus du MBR.

Cependant le système GPT est encore peu répandu dans l'embarqué (pas de besoins).

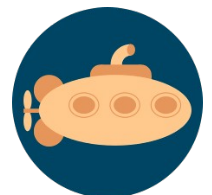
Sur les ordinateurs personnels, les bootloaders sont par défaut transparents.
On n'y accède qu'en interrompant le BIOS/UEFI, ou en cas de dual-boot.

GNU Grub (*Grand Unified Bootloader*) est le bootloader par défaut des distributions GNU/Linux. Il est capable de démarrer différents OS à partir de tous les systèmes de fichiers connus (contrairement aux deux homologues ci-dessous).

Windows Boot Manager est le bootloader Microsoft utilisé depuis Windows Vista.

Boot Camp est le bootloader d'Apple, capable de lancer un OS Windows.

Avec **Barebox**, le bootloader le plus connu pour les OS GNU/Linux embarqués est **Das U-boot** (the Universal Bootloader) (https://www.denx.de/wiki/U-Boot/).



Il fonctionne sur x86, ARM, RISC-V, MicroBlaze, MIPS, …
Il supporte plusieurs filesystems et peut charger un kernel depuis une interface SATA, SD, I2C, eMMC, USB, port série, réseau, …

U-boot propose une CLI (console ou liaison série) pour régler ses paramètres en direct.

Il est décomposé en deux étages :

- Un SPL (*Secondary Program Loader*) très léger qui fera les initialisations nécessaires au bootloader complet
- U-boot lui-même, qui configurera les contrôleurs mémoire et chargera le kernel en RAM.

Das U-boot : Exemple de stage 3A SATE chez BOOTLIN en 2020

Bootlin contributes SquashFS support to U-Boot

SquashFS is a very popular read-only compressed root filesystem, widely used in embedded systems. It has been supported in the Linux kernel for many years, but so far the U-Boot bootloader did not have support for SquashFS, so it was not possible to load a kernel image or a Device Tree Blob from a SquashFS filesystem in U-Boot.

Between February 2020 and August 2020, João Marcos Costa from the ENSICAEN engineering school, has worked at Bootlin as an intern. João's internship goal was specifically to implement and contribute to U-Boot the support for the SquashFS filesystem. We are happy to announce that João's effort has now completed, as the support for SquashFS is now in upstream U-Boot. It can be found in fs/squashfs/ in the U-Boot source code.

https://bootlin.com/blog/bootlin-contributes-squashfs-support-to-u-boot/

BareBox : bootloader pour l'embarqué

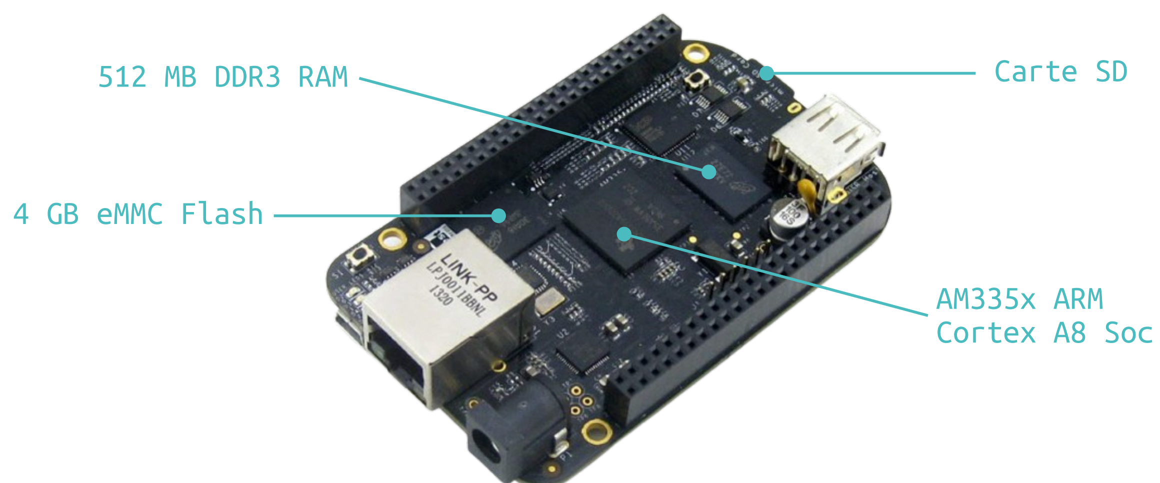
Initialement u-boot-v2, BareBox est un fork de Das U-boot.
https://www.barebox.org/

Il vise les systèmes embarqués, et support de nombreuses architectures (ARM, x86, MIPS, PowerPC, …) et filesystems.

SÉQUENCE D'AMORÇAGE

Cas de la BeagleBone Black

Prenons comme exemple la BeagleBone Black sur laquelle nous travaillerons en TP.



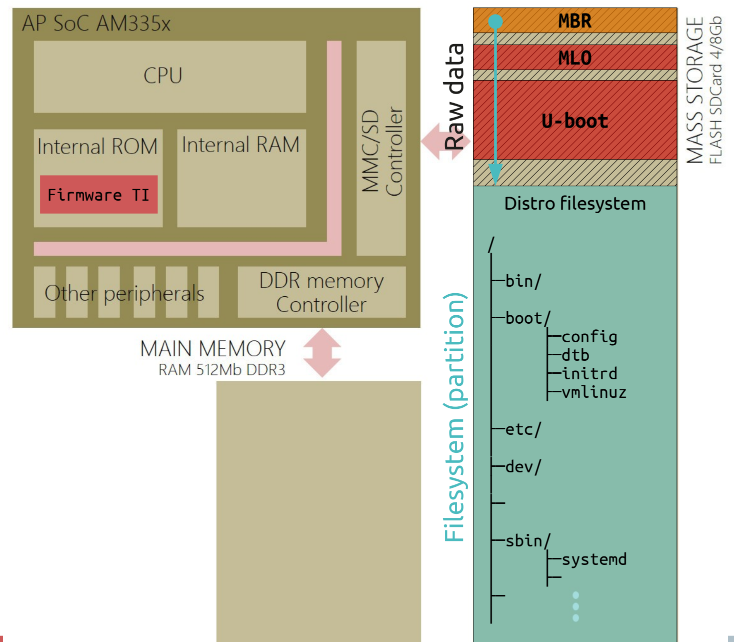
https://beagleboard.org/black/

Le firmware Boot Rom est contenu dans la ROM interne au SoC AM335x de TI.

MLO (édité par TI) et U-boot (édité par DENX) sont les bootloaders externes, stockés en binaire dans le média de masse.

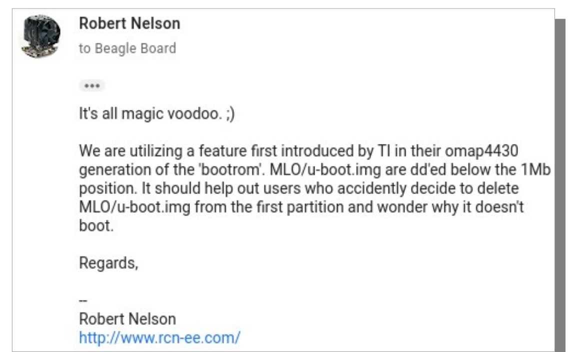
La distribution GNU/Linux (et donc le kernel) se trouvent sur une partition (système de fichier).

La partition est indiquée par le MBR, présent sur les 512 premiers octets de mémoire de masse.



Précisions sur les bootloaders.

Le firmware Boot Rom est capable d'aller chercher un bootloader sur un système de fichier FAT. Ici, le développeur Linux pour BBB (R.C. Nelson) a fait le choix de le placer dans un espace en donnée brute (hors fs).


Robert Nelson
to Beagle Board

It's all magic voodoo. ;)

We are utilizing a feature first introduced by TI in their omap4430 generation of the 'bootrom'. MLO/u-boot.img are dd'ed below the 1Mb position. It should help out users who accidently decide to delete MLO/u-boot.img from the first partition and wonder why it doesn't boot.

Regards,

--
Robert Nelson
http://www.rcn-ee.com/

U-boot est un bootloader complexe, décomposé en deux étages.

Le premier étage est un SPL (*Secondary Program Loader*). Ce rôle est rempli par MLO (Minimal Bootloader), intégré au code source de U-boot par Texas Instruments.

TI fournit ce programme pour assurer la portabilité de U-boot (solution ultra-répandue) pour leur processeur.

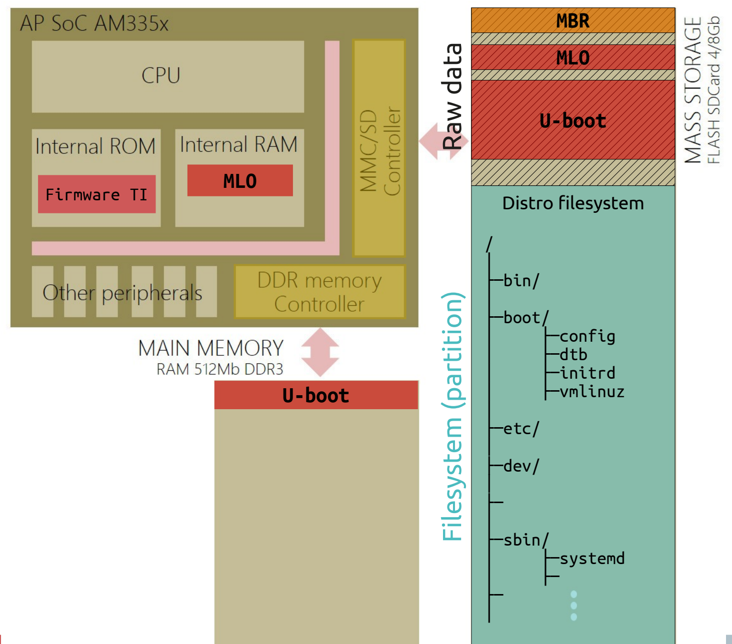
Mise sous tension

Le firmware TI active le contrôleur MMC/SD.

Il parcourt la partition binaire brute, identifie le MLO et le charge alors en RAM interne au SoC.

À son tour, le MLO active le contrôleur DDR puis localise U-boot.

Il transfère U-boot en RAM externe.



Précision

Comment Boot Rom détecte MLO dans un espace binaire brut (sans filesystem).

26.1.11 Table of Contents

The Table of Contents (TOC) is a header needed only in GP devices while using MMC RAW mode. This must not be confused with the TOC used in HS devices. The TOC is 512 bytes long and consists of a maximum of 2 TOC items (32 bytes long each), located one after the other. The second TOC item must be filled by FFh. Each TOC item contains information required by the ROM Code to find a valid image in RAW mode, as illustrated in Table 26-38. To detect RAW mode, the ROM also needs the magic values mentioned in Table 26-39. Other than the TOC item fields and magic values, all the other bytes in the 512-byte TOC must be zero.

Table 26-38. The TOC Item Fields

Offset	Field	Size (bytes)	Description
0000h	Start	4	0x00000040
0004h	Size	4	0x0000000C
0008h	Flags	4	Not used, should be zero.
000Ch	Align	4	Not used, should be zero.
0010h	Load Address	4	Not used, should be zero.
0014h	Filename	12	12 character long name of sub image, including the zero ('\0') terminator. The ASCII representation is "CHSETTINGS".

Table 26-39. Magic Values for MMC RAW Mode

Offset	Value
40h	0xC0C0C0C1
44h	0x00000100

The ROM Code recognizes the TOC based on the filename described in Table 26-40.

Table 26-40. Filenames in TOC for GP Device

Filename	Description
CHSETTINGS	Magic string used by ROM

SPRUH73Q–October 2011–Revised December 2019 *Initialization* 5077

C'est défini dans le *Technical Reference Manual* du processeur, encore faut-il le trouver !

```
1 00000000: 4000 0000 0c00 0000 0000 0000 0000 0000  @...............
2 00000010: 0000 0000 4348 5345 5454 494e 4753 0000  ....CHSETTINGS..
3 00000020: ffff ffff ffff ffff ffff ffff ffff ffff  ................
4 00000030: ffff ffff ffff ffff ffff ffff ffff ffff  ................
5 00000040: c1c0 c0c0 0001 0000 0000 0000 0000 0000  ................
6 00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
7 00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Extrait du fichier MLO (après conversion binaire → texte)

MMC Raw mode, p5077

AM335x and AMIC110 Sitara™ Processors
Technical Reference Manual

https://www.ti.com/lit/pdf/spruh73

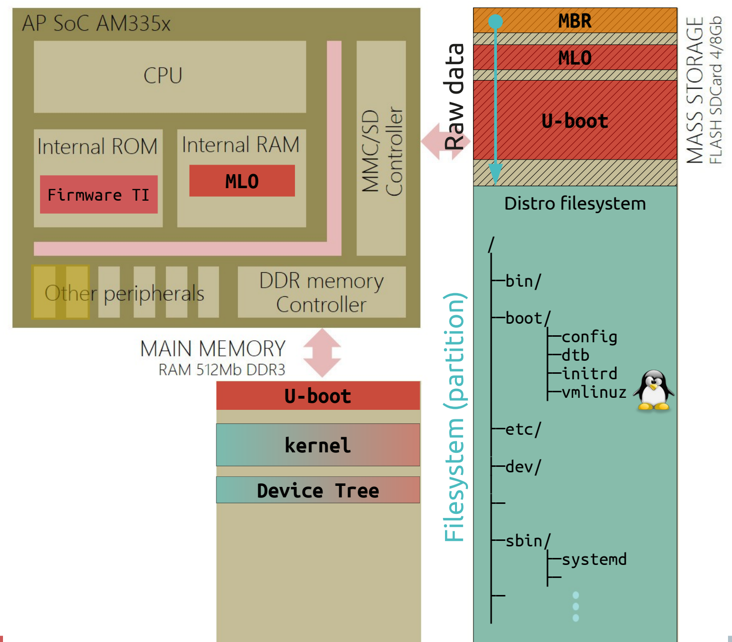
Mise sous tension (la suite)

La première tâche d'U-boot est de configurer certains périphériques (notamment la liaison série pour IHM).

Une fois prêt, U-boot lit le MBR pour identifier les partitions de fichiers dans la mémoire de masse.

Il parcourt le système de fichier de chaque partition jusqu'à trouver l'image compressée du kernel (vmlinuz) et le device tree.

Il les charge en RAM, puis donne la main au kernel quand sa mission est terminée.



Lancement du kernel

Le point d'entrée dans le kernel dépend de l'architecture. C'est le fichier assembleur :

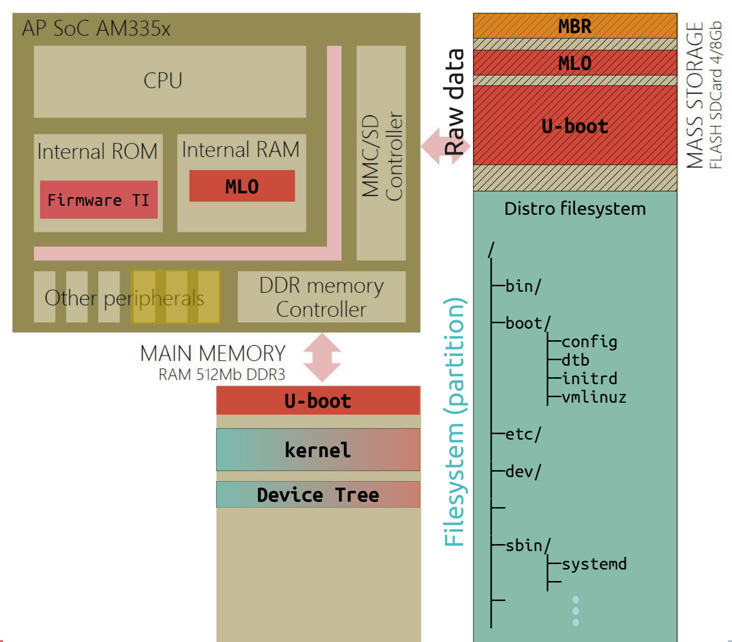
`arch/arm/boot/compressed/head.S`
label « `start` »

Le kernel initie sa décompression.

Toujours en lien avec le matériel (donc asm), le kernel configure le cache et la MMU.

L'exécution du kernel se poursuit, entrant maintenant dans la phase indépendante de l'architecture :

fichier `init/main.c`, fonction `start_kernel()`



SÉQUENCE D'AMORÇAGE

Lancement du kernel

Le fichier head.S (à gauche) est le premier point d'entrée dans le kernel et dépend de l'architecture.

Le fichier init/main.c (à droite) est le premier point d'entrée architecture-agnostique du kernel.

```
  1 /* SPDX-License-Identifier: GPL-2.0-only */
  2 /*
  3  *  linux/arch/arm/boot/compressed/head.S
  4  *
  5  *  Copyright (C) 1996-2002 Russell King
  6  *  Copyright (C) 2004 Hyok S. Choi (MPU support)
  7  */
  8 #include <linux/linkage.h>
  9 #include <asm/assembler.h>
 10 #include <asm/v7m.h>
 11
 12 #include "efi-header.S"

185                 .align
186                 /*
187                  * Always enter in ARM state for CPUs that support the ARM ISA.
188                  * As of today (2014) that's exactly the members of the A and R
189                  * classes.
190                  */
191  AR_CLASS(      .arm    )
192 start:
193                 .type   start,#function
194                 /*
195                  * These 7 nops along with the 1 nop immediately below for
196                  * !THUMB2 form 8 nops that make the compressed kernel bootable
197                  * on legacy ARM systems that were assuming the kernel in a.out
198                  * binary format. The boot loaders on these systems would
199                  * jump 32 bytes into the image to skip the a.out header.
200                  * with these 8 nops filling exactly 32 bytes, things still
201                  * work as expected on these legacy systems. Thumb2 mode keeps
202                  * 7 of the nops as it turns out that some boot loaders
203                  * were patching the initial instructions of the kernel, i.e
204                  * had started to exploit this "patch area".
205                  */
206                 __initial_nops
207                 .rept   5
208                 __nop
```

```
927 asmlinkage __visible void __init __no_sanitize_address start_kernel(void)
928 {
929         char *command_line;
930         char *after_dashes;
931
932         set_task_stack_end_magic(&init_task);
933         smp_setup_processor_id();
934         debug_objects_early_init();
935         init_vmlinux_build_id();
936
937         cgroup_init_early();
938
939         local_irq_disable();
940         early_boot_irqs_disabled = true;
941
942         /*
943          * Interrupts are still disabled. Do necessary setups, then
944          * enable them.
945          */
946         boot_cpu_init();
947         page_address_init();
948         pr_notice("%s", linux_banner);
949         early_security_init();
950         setup_arch(&command_line);
951         setup_boot_config();
952         setup_command_line(command_line);
953         setup_nr_cpu_ids();
954         setup_per_cpu_areas();
955         smp_prepare_boot_cpu(); /* arch-specific boot-cpu hooks */
956         boot_cpu_hotplug_init();
957
958         build_all_zonelists(NULL);
959         page_alloc_init();
```

Initialisation des composants matériels et logiciels

SÉQUENCE D'AMORÇAGE

Lancement de l'OS et applications

Le kernel est capable de s'auto-décompressé !

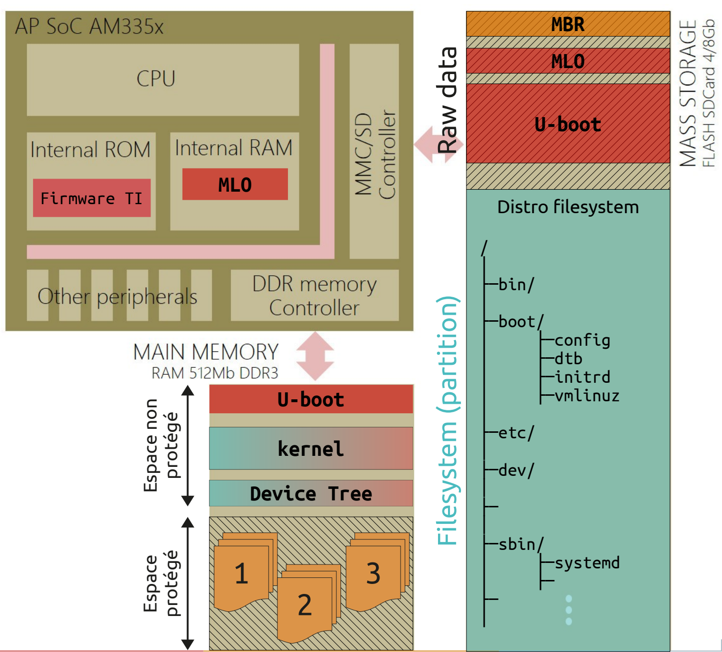
De même, il a configure et exploite la **MMU**. Cela implique que le kernel travaille en d'adressage réel non protégé (lien direct CPU vers RAM avec bypass de la MMU), voir boot/system.map.

Tandis que les processus (applications chargées puis exécutées depuis la RAM) tourneront en adressage virtualisé (via la MMU). On appelle cet espace mémoire « espace protégé ».

Après son démarrage, le kernel lance le premier exécutable (PID = 1) :

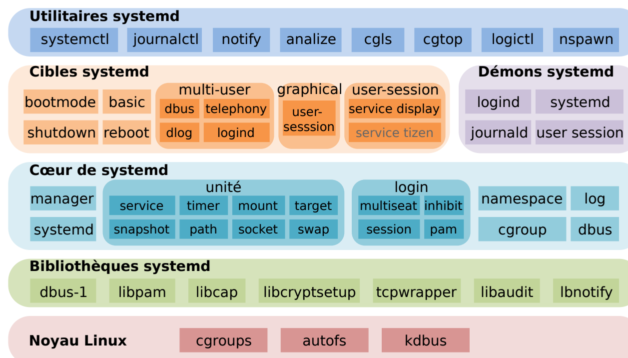
/sbin/systemd

Les segments de *code*, *stack* et *heap* arrivent en RAM. La machine est lancée, d'autre processus seront chargés puis exécutés automatiquement en parallèle par systemd.

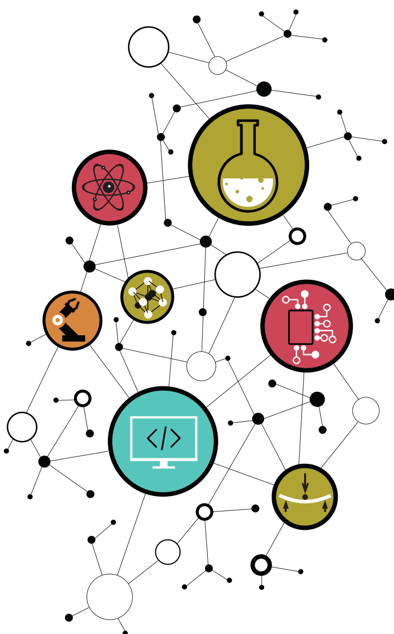


systemd est une suite logicielle qui fournit une gamme de composants système pour les systèmes d'exploitation Linux. Il est une alternative à SysV Init. Systemd a notamment pour but d'offrir un meilleur cadre pour la gestion des dépendances entre services, de permettre le chargement en parallèle des services au démarrage et de réduire les appels aux scripts shell.

En 2020, systemd représente à lui seul 1.273.896 lignes de code et plus de 300 développeurs ont contribué à son développement cette même année.



CONTACT



Dimitri Boudier – PRAG ENSICAEN

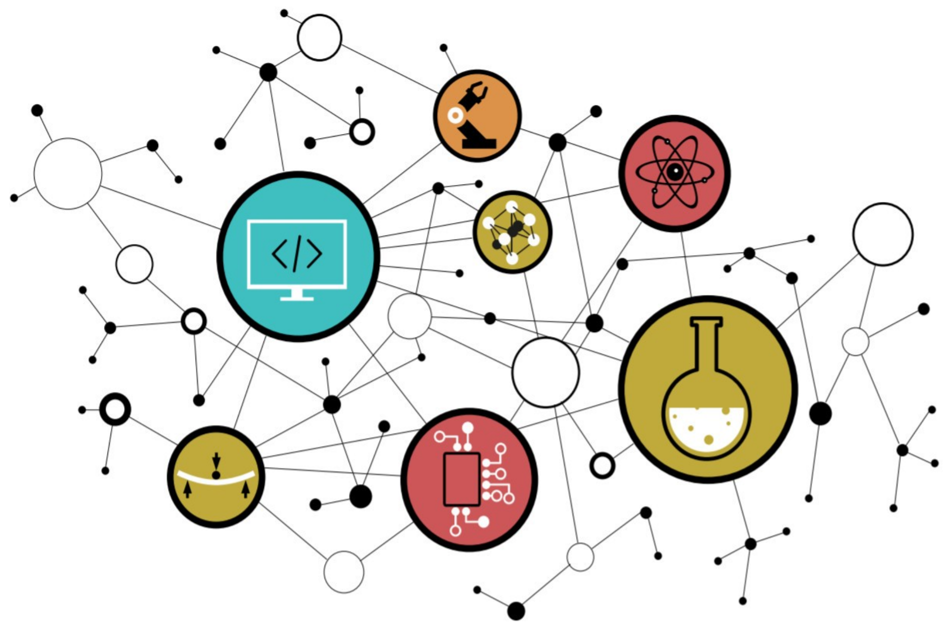
dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)

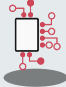
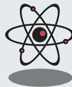
Except where otherwise noted, this work is licensed under https://creativecommons.org/licenses/by-nc-sa/3.0/

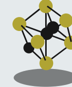
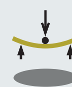
LINUX FROM SCRATCH



Chapitre 3

Linux from scratch

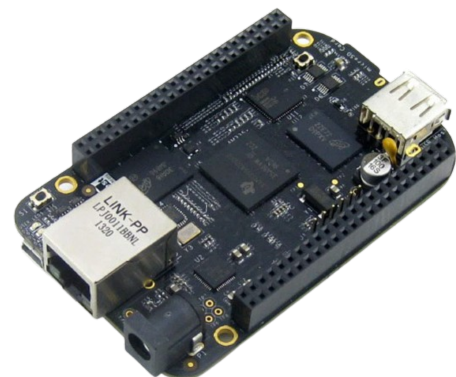
2021-2022

Nous aborderons dans ce chapitre les principes de la création d'un système ***Linux From Scratch***, comprenez "en partant des sources".

Nous ne verrons ici que les grandes lignes, puisque la trame de TP portera dans son intégralité sur la création d'un OS embarqué *from scratch*.

En TP, nous utiliserons comme cible la BeagleBone Black équipée d'un SoC Sitara AM3358 de TI.

Nous chercherons en plus à lui ajouter la fonctionnalité CAN, désactivée par défaut.



Les exemples présentés seront basés sur ce cahier des charges.

Le développement d'un Linux embarqué se fait, par définition, pour une cible spécifique que l'on appellera "*target*".

Pour faciliter la création d'un *Linux from scratch*, il est fortement conseillé de procéder depuis un système GNU/Linux hôte, puisqu'il contient les mêmes outils que ceux qu'on essaiera de créer. Dans notre cas, le "*host*" sera une distribution Ubuntu.

/!\ Parti pris /!

Notez que la méthode présentée ici (et vue en TP, donc) est une technique "à la main", sans outil extérieur. L'objectif est de comprendre toutes les étapes qui sont réalisées par des outils plus haut niveau (Yocto, …) que nous aborderons plus tard.

Ce chapitre est découpé selon les étapes de la création d'un *Linux from scratch*.

1. Téléchargement des sources du kernel depuis le *repository* (non détaillé ici)
2. Téléchargement de la chaîne de compilation croisée
3. Configuration des services du noyau
4. Configuration du device tree
5. Compilation du *kernel*, des modules et du *device tree*
6. Déploiement sur cible

TÉLÉCHARGEMENT DES SOURCES

Étape 1

1. TÉLÉCHARGEMENT DES SOURCES

La première étape consiste, évidemment, à récupérer les sources du kernel Linux souhaité.

```
git clone https://github.com/RobertCNelson/bb-kernel

cd bb-kernel/
git checkout origin/am33x-rt-v4.14 -b tmp
```

Exemple appliqué à la BeagleBone Black

Rappel : ce ne sont que les grandes lignes (il faut aussi penser au *bootloader* par exemple)

CHAÎNE DE COMPILATION CROISÉE

Étape 2

Téléchargement

La deuxième étape consiste à récupérer la chaîne de compilation croisée.

En effet, on compilera le kernel pour une cible embarquée différente de notre *host*.

Il est difficile de créer sa propre *cross-toolchain* (dépendances *toolchain* ← *libc* ← kernel).

Nous utiliserons donc **Linaro**, une toolchain optimisée pour les architectures ARM (Cortex-A8, A9, …) et dont le projet est activement suivi et maintenu.

Après téléchargement :

```
export CC=<full_path>/gcc-linaro-6.5.0-2018.12-x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-

${CC}gcc --version
```

```
arm-linux-gnueabihf-gcc (Linaro GCC 6.5-2018.12) 6.5.0
Copyright © 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

CONFIGURATION DES SERVICES

Étape 3

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

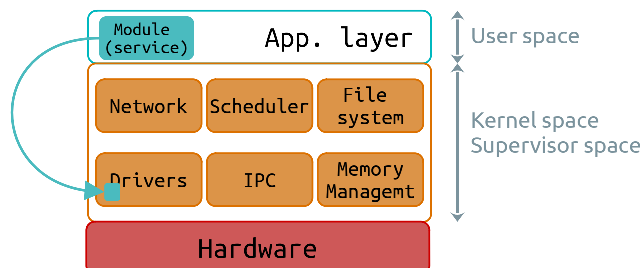
ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Monolithic modular kernel

Il s'agit de noyaux monolithiques avec une approche modulaire dynamique, impliquant le chargement à chaud de modules *kernel* (*runtime*).

Cette solution permet de n'inclure que les services nécessaires dans l'espace *kernel* puis d'en rajouter à chaud en fonction des besoins (solution modulable très pratique pour des phases de prototypage de drivers).

Quelques exemples : GNU/Linux > 1.2, FreeBSD, Solaris …



Linux est donc un **kernel monolithique modulaire**. Ainsi un service est sera :

- soit directement intégré au kernel (compilé dans le même exécutable) ;
- soit chargé dynamiquement pendant l'exécution, depuis le *user-space* ;
- soit pas disponible du tout (sauf recompilation).

Pour la création d'un *Linux from scratch*, ce choix est bien évidemment laissé au développeur pour CHAQUE module (même s'ils ont une valeur par défaut).

Ce choix se fait en créant le fichier `.config` à la racine du projet.

Par ex. la configuration du *host* est accessible ici : `/boot/config-<supported-versions>`

Le fichier de configuration `.config` permet l'inclusion de directives de compilation présents dans les différents **Makefile** du projet.

À titre d'exemple, voici le Makefile propre à l'ajout de drivers et services CAN au kernel (`/drivers/net/can/Makefile`) :

```
1887 #
1888 # CAN Device Drivers
1889 #
1890 CONFIG_CAN_VCAN=m
1891 CONFIG_CAN_VXCAN=m
1892 CONFIG_CAN_SLCAN=m
1893 CONFIG_CAN_DEV=m
1894 CONFIG_CAN_CALC_BITTIMING=y
1895 CONFIG_CAN_JANZ_ICAN3=m
1896 CONFIG_CAN_KVASER_PCIEFD=m
1897 CONFIG_CAN_C_CAN=m
```

Sur le kernel host (v 5.13.0-28) `/usr/src/<header>/.config`

path: root/drivers/net/can/Makefile

blob: 1e660afcb61bf11bfc30557212cbd52717868afb (plain)

```
1  # SPDX-License-Identifier: GPL-2.0
2  #
3  #  Makefile for the Linux Controller Area Network drivers.
4  #
5
6  obj-$(CONFIG_CAN_VCAN)          += vcan.o
7  obj-$(CONFIG_CAN_VXCAN)         += vxcan.o
8  obj-$(CONFIG_CAN_SLCAN)         += slcan.o
9
10 obj-y                           += dev/
11 obj-y                           += rcar/
12 obj-y                           += spi/
13 obj-y                           += usb/
14 obj-y                           += softing/
15
16 obj-$(CONFIG_CAN_AT91)          += at91_can.o
17 obj-$(CONFIG_CAN_CC770)         += cc770/
18 obj-$(CONFIG_CAN_C_CAN)         += c_can/
19 obj-$(CONFIG_CAN_FLEXCAN)       += flexcan/
20 obj-$(CONFIG_CAN_GRCAN)         += grcan.o
21 obj-$(CONFIG_CAN_IFI_CANFD)     += ifi_canfd/
22 obj-$(CONFIG_CAN_JANZ_ICAN3)    += janz-ican3.o
23 obj-$(CONFIG_CAN_KVASER_PCIEFD) += kvaser_pciefd.o
24 obj-$(CONFIG_CAN_MSCAN)         += mscan/
25 obj-$(CONFIG_CAN_M_CAN)         += m_can/
26 obj-$(CONFIG_CAN_PEAK_PCIEFD)   += peak_canfd/
27 obj-$(CONFIG_CAN_SJA1000)       += sja1000/
28 obj-$(CONFIG_CAN_SUN4I)         += sun4i_can.o
29 obj-$(CONFIG_CAN_TI_HECC)       += ti_hecc.o
30 obj-$(CONFIG_CAN_XILINXCAN)     += xilinx_can.o
31 obj-$(CONFIG_PCH_CAN)           += pch_can.o
32
33 subdir-ccflags-$(CONFIG_CAN_DEBUG_DEVICES) += -DDEBUG
```

Sources du *repo* du kernel (v5.13.0-28)

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Il existe plusieurs manières de générer le fichier de configuration .config.

La plus rudimentaire consiste à éditer manuellement ce fichier, mais elle est aussi la plus dangereuse car elle ne prend pas en compte les dépendances entre les services du kernel. Mais surtout : 11241 lignes dans ce fichier (kernel 5.13.0-28) !

```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # Linux/x86 5.13.0-28-generic Kernel Configuration
4 #
5 CONFIG_CC_VERSION_TEXT="gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
6 CONFIG_CC_IS_GCC=y
7 CONFIG_GCC_VERSION=90300
8 CONFIG_CLANG_VERSION=0
9 CONFIG_AS_IS_GNU=y
10 CONFIG_AS_VERSION=23400
11 CONFIG_LD_IS_BFD=y
```

```
418 #
419 # Performance monitoring
420 #
421 CONFIG_PERF_EVENTS_INTEL_UNCORE=y
422 CONFIG_PERF_EVENTS_INTEL_RAPL=m
423 CONFIG_PERF_EVENTS_INTEL_CSTATE=m
424 # CONFIG_PERF_EVENTS_AMD_POWER is not set
425 # end of Performance monitoring
```

y = service monolithique

m = service modulaire

CONFIG_*** is not set = service non compilé

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Les autres solutions de génération du fichier .config se basent sur des interfaces graphiques (ou presque). Celles-ci utilisent les fichiers Kconfig présents dans chaque répertoire de l'arborescence.

On appelle l'IHM souhaitée en l'appelant depuis la console :

- make config : interface shell, navigation difficile
- make menuconfig : interface ncurse, la plus utilisée car simple et rapide
- make xconfig : interface X Window
- make gconfig : interface GTK+

Quelle que soit l'interface, l'avantage est la gestion des dépendances.

Exemple issu de la trame de TP.

Le SoC AM335x de la famille Sitara présent sur la BBB embarque deux contrôleurs CAN (DCAN0 et DCAN1).

Hors, aucune des distributions présentes sur internet et aucune des configurations du kernel pour la BBB n'incluent nativement de services d'interface.

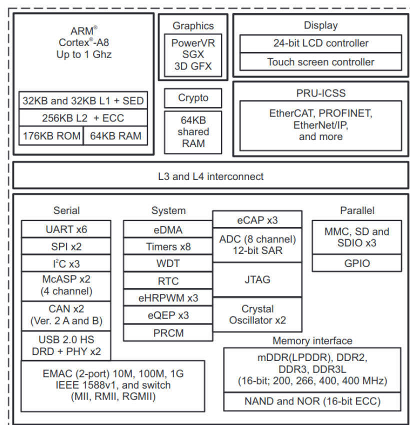


Figure 1-1. AM335x Functional Block Diagram

AM335x Sitara™ Processors datasheet (Rev. L)
https://www.ti.com/lit/gpn/am3359

AM335x and AMIC110 Sitara™ Processors Technical Reference Manual (Rev. Q)
https://www.ti.com/lit/pdf/spruh73

www.ti.com

23.1 Introduction

23.1.1 DCAN Features

The general features of the DCAN controller are:

- Supports CAN protocol version 2.0 part A, B (ISO 11898-1)
- Bit rates up to 1 MBit/s
- Dual clock source
- 16, 32, 64 or 128 message objects (instantiated as 64 on this device)
- Individual identifier mask for each message object
- Programmable FIFO mode for message objects
- Programmable loop-back modes for self-test operation
- Suspend mode for debug support
- Software module reset
- Automatic bus on after Bus-Off state by a programmable 32-bit timer
- Message RAM parity check mechanism
- Direct access to Message RAM during test mode
- CAN Rx / Tx pins configurable as general purpose IO pins
- Two interrupt lines (plus additional parity-error interrupt line)
- RAM initialization
- DMA support

Nous devons évidemment commencer par rechercher les informations de configuration du kernel pour ce processeur.

Processor SDK Linux Software Developer's Guide :

https://software-dl.ti.com/processor-sdk-linux/esd/docs/latest/linux/Foundational_Components/Kernel/Kernel_Drivers/DCAN.html

Extrait de :

Processor SDK Linux for AM335X
→ Fundational Components
→ Kernel
→ DCAN
→ Linux Driver Configuration

```
[*] Networking support ->
   <*|M> CAN bus subsystem support ->
      <*|M> Raw CAN Protocol (raw access with CAN-ID filtering)
      <*|M> Broadcast Manager CAN Protocol (with content filtering)
      <*|M> CAN Gateway/Router (with netlink configuration)
         CAN Device Drivers ->
            <*|M>   Platform CAN drivers with Netlink support
            [*]     CAN bit-timing calculation
            <*|M>   Bosch C_CAN/D_CAN devices ->
               <M> Generic Platform Bus based C_CAN/D_CAN driver
```

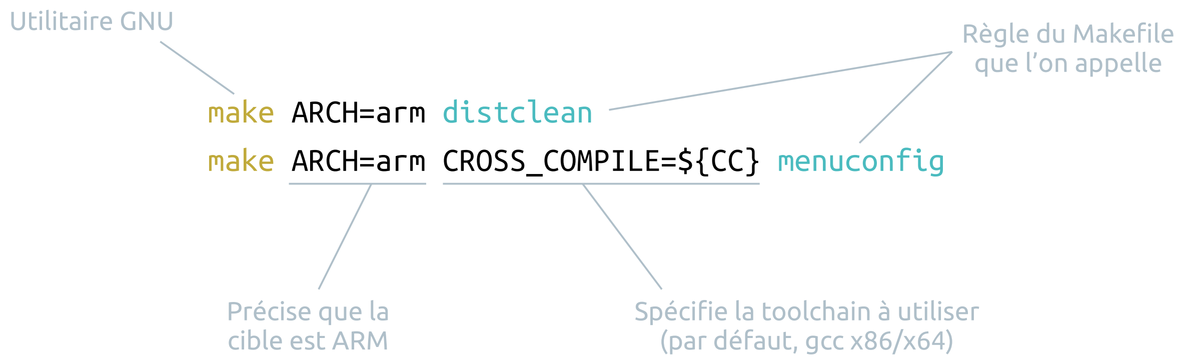
NOTE *|M means can be either be built into the kernel or enabled as a kernel module.

Exemple : service CAN

ENSI CAEN ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Ayant connaissance des modules à intégrer au kernel, lançons l'édition du fichier de configuration `.config` avec l'interface menuconfig.

Il faut au préalable se placer dans le répertoire racine des sources du kernel (pour utiliser son `Makefile`).



3. CONFIGURATION DES SERVICES

Exemple : service CAN

ENSI CAEN ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

Navigation dans le menuconfig et inclusion ('*') des services nécessaires au fonctionnement du CAN.

```
    Power management options  --->
[*] Networking support  --->
    Device Drivers

--- Networking support
      Networking options  --->
[*]   Amateur Radio support  --->
<*>   CAN bus subsystem support  --->
<M>   Bluetooth subsystem support

--- CAN bus subsystem support
<*>   Raw CAN Protocol (raw access with CAN-ID filtering)
<*>   Broadcast Manager CAN Protocol (with content filtering)
<*>   CAN Gateway/Router (
<M>   ISO 15765-2:2016 CAN
      CAN Device Drivers

<*> Virtual Local CAN Interface (vcan)
<M> Virtual CAN Tunnel (vxcan)
<M> Serial / USB serial CAN Adaptors (slcan)
<*> Platform CAN drivers with Netlink support
[*]   CAN bit-timing calculation
[ ]   Enable LED triggers for Netlink based drivers
< >   Support for Freescale FLEXCAN based chips
< >   Aeroflex Gaisler GRCAN and GRHCAN CAN devices
<M>   Janz VMOD-ICAN3 Intelligent CAN controller
< >   TI High End CAN Controller
<*>   Bosch C_CAN/D_CAN devices

--- Bosch C_CAN/D_CAN devices
<M>   Generic Platform Bus based C_CAN/D_CAN driver
<M>   Generic PCI Bus based C_CAN/D_CAN driver
```

3. CONFIGURATION DES SERVICES

Exemple : service CAN

Sauvegarde et apparition du fichier `.config`.

```
1503 #
1504 # CAN Device Drivers
1505 #
1506 CONFIG_CAN_VCAN=y
1507 CONFIG_CAN_VXCAN=m
1508 CONFIG_CAN_SLCAN=m
1509 CONFIG_CAN_DEV=y
1510 CONFIG_CAN_CALC_BITTIMING=y
1511 # CONFIG_CAN_LEDS is not set
1512 # CONFIG_CAN_FLEXCAN is not set
1513 # CONFIG_CAN_GRCAN is not set
1514 CONFIG_CAN_JANZ_ICAN3=m
1515 # CONFIG_CAN_TI_HECC is not set
1516 CONFIG_CAN_C_CAN=y
1517 CONFIG_CAN_C_CAN_PLATFORM=m
1518 CONFIG_CAN_C_CAN_PCI=m
1519 CONFIG_CAN_CC770=m
1520 CONFIG_CAN_CC770_ISA=m
1521 CONFIG_CAN_CC770_PLATFORM=m
1522 CONFIG_CAN_IFI_CANFD=m
1523 CONFIG_CAN_M_CAN=m
1524 CONFIG_CAN_PEAK_PCIEFD=m
```

En faisant une configuration sans rien modifier, ces éléments apparaissent en tant que module et non comme des services natifs.

3. CONFIGURATION DES SERVICES

Exemple : service CAN

Si **à ce stade** on compilait le kernel et l'intégrait dans la cible, on observerait le comportement suivant.

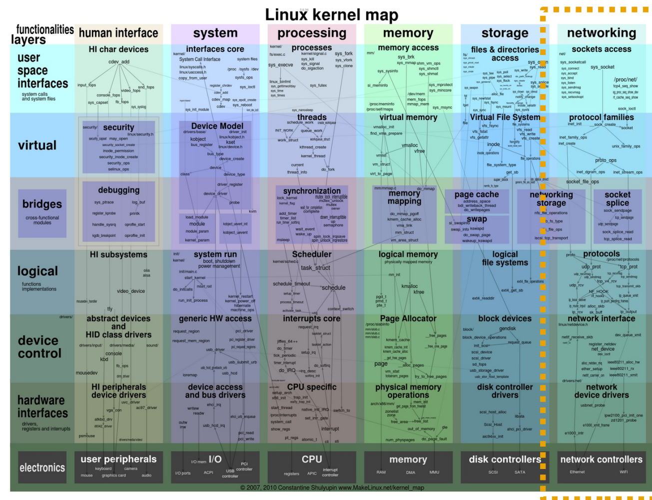
Au démarrage de la cible, le kernel affiche une série de messages pendant son démarrage. Nous pouvons trier et n'afficher que eux concernant le service CAN avec la commande *display messages* : « `dmesg | 'CAN\|can'` » .

```
root@arm:~# dmesg | grep 'CAN\|can:'
[    0.116922] platform 481d0000.d_can: alias fck already exists
[    0.726646] vcan: Virtual CAN interface driver
[    0.726674] CAN device driver interface
[    0.727011] c_can_platform 481d0000.d_can: invalid resource
[    0.732965] c_can_platform 481d0000.d_can: control memory is not used for ra
[    0.763904] c_can_platform 481d0000.d_can: c_can_platform device registered
[    0.895113] can: controller area network core (rev 20120528 abi 9)
[    0.895292] can: raw protocol (rev 20120528)
[    0.895310] can: broadcast manager protocol (rev 20120528 t)
```

Par contre, l'interface CAN n'apparaît pas dans les interfaces réseaux suite à un « `ifconfig -a` ».

Les services sont bien intégrés au kernel Linux, mais le périphérique CAN n'est pas configuré correctement !

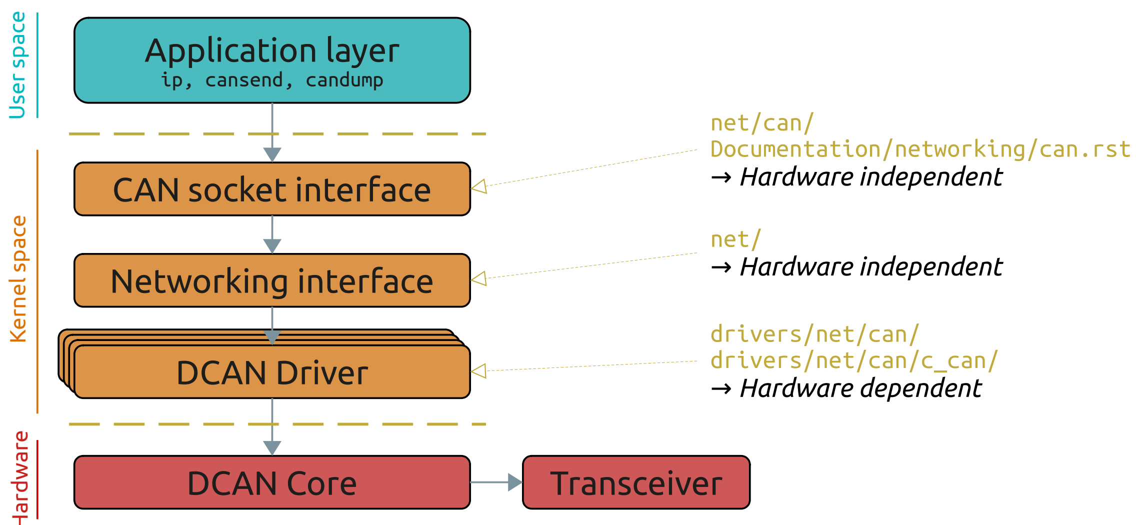
Afin d'assurer un bonne interopérabilité, maintenabilité et portabilité, le kernel Linux est découpé en couches proposant plusieurs niveaux d'abstraction, notamment au regard du matériel. Observons le *kernel map* du noyau :



Networking
Linux Kernel Layers

https://makelinux.github.io/kernel/map/

Observons la décomposition en couche du système d'exploitation déployé.



DEVICE DRIVER

Exemple de stage 3A SATE

EXEMPLE DE STAGE 3A SATE

Exemple de stage 3A SATE réalisé par Clément Perrochaud sur Caen chez Effinov (https://www.effinnov.com/). Cette entreprise propose notamment des services de développement pour la société NXP (conception et développement de composant électronique).

NXP développe notamment sur Caen des composants sécurisés. Ce stage s'inscrit dans le but d'ajouter le support de l'I2C sous Linux pour leur composant NCI.

```
index : kernel/git/torvalds/linux.git
Linux kernel source tree

about  summary  refs  log  tree  commit  diff  stats

author     Clément Perrochaud <clement.perrochaud@nxp.com>   2015-03-09 11:12:05 +0100
committer  Samuel Ortiz <sameo@linux.intel.com>              2015-03-26 11:21:41 +0100
commit     6be88670fc59d50426f90f734a36b90e1de7d148 (patch)
tree       8296eaff5b2747937b4239fcee58f5fa380926dc
parent     dece45855a8b0d1dcf48eb01d0822070ded6a4c8 (diff)
download   linux-6be88670fc59d50426f90f734a36b90e1de7d148.tar.gz

NFC: nxp-nci_i2c: Add I2C support to NXP NCI driver

Add a module to the NXP-NCI driver to support NFC controllers with an
I2C control interface, such as the NPC100.

Signed-off-by: Clément Perrochaud <clement.perrochaud@effinnov.com>
Signed-off-by: Samuel Ortiz <sameo@linux.intel.com>

Diffstat
-rw-r--r-- Documentation/devicetree/bindings/net/nfc/nxp-nci.txt  35
-rw-r--r-- drivers/nfc/nxp-nci/Kconfig                             12
-rw-r--r-- drivers/nfc/nxp-nci/Makefile                             2
-rw-r--r-- drivers/nfc/nxp-nci/i2c.c                              415
4 files changed, 464 insertions, 0 deletions

diff --git a/Documentation/devicetree/bindings/net/nfc/nxp-nci.txt b/Documentation/devicetree/bindings/net/nfc/nxp-nci.txt
new file mode 100644
index 0000000000000..5b6cd9b3f628a
--- /dev/null
+++ b/Documentation/devicetree/bindings/net/nfc/nxp-nci.txt
@@ -0,0 +1,35 @@
+* NXP Semiconductors NXP NCI NFC Controllers
+
```

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.0-rc4&id=6be88670fc59d50426f90f734a36b90e1de7d148

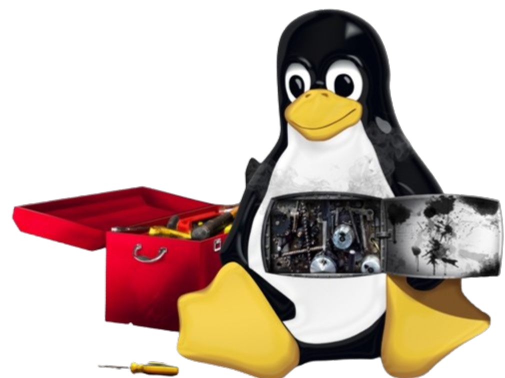
DEVICE TREE

Étape 4

Pour rappel, les processeurs ont généralement plus de périphériques que de broches disponibles. Autrement dit, les périphériques doivent se partager l'accès aux GPIO.

Nous devons donc décrire l'architecture matérielle cible au kernel.

L'idée est simple : nous utiliserons des fichiers de description matérielle.



Historiquement, les *board files* étaient utilisés pour remplir ce rôle.

Il s'agit de fichiers C présents dans le répertoire arch/<cpu_arch>/.

Exemple issu du kernel 5.17-rc5

/arch/arm/mach-omap1/board-nokia770.c

Il s'agit d'un fichier C typique de ce qu'on peut produire en embarqué.

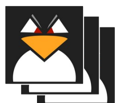
```
267 static void __init omap_nokia770_init(void)
268 {
269         /* On Nokia 770, the SleepX signal is masked with an
270          * MPUIO line by default.  It has to be unmasked for it
271          * to become functional */
272 
273         /* SleepX mask direction */
274         omap_writew((omap_readw(0xfffb5008) & ~2), 0xfffb5008);
275         /* Unmask SleepX signal */
276         omap_writew((omap_readw(0xfffb5004) & ~2), 0xfffb5004);
277 
278         platform_add_devices(nokia770_devices, ARRAY_SIZE(nokia770_devices));
279         nokia770_spi_board_info[1].irq = gpio_to_irq(15);
280         spi_register_board_info(nokia770_spi_board_info,
281                                 ARRAY_SIZE(nokia770_spi_board_info));
282         omap_serial_init();
283         omap_register_i2c_bus(1, 100, NULL, 0);
284         hwa742_dev_init();
285         mipid_dev_init();
286         omap1_usb_init(&nokia770_usb_config);
287         nokia770_mmc_init();
288         nokia770_cbus_init();
289 }
```


Même si les *boards files* ont été progressivement abandonnés, voyons leurs avantages et inconvénients

- Édition en C (compétences déjà présentes)

- Temps de boot plus rapide (pas de *parsing*)

- Surcharge importante du répertoire /arch/
  - Depuis l'arrivée de Linux dans l'embarqué, le nombre d'architectures supportés a explosé
  - Maintenabilité complexe, impossible à concevoir pour Torvalds.



- Modifier une seule configuration matérielle implique de recompiler l'intégralité du kernel

Les *board files* ont été progressivement remplacés par les ***Device Trees***.

Les Devices Trees (DT) sont construits sous forme de structure de données qui décrivent les composants matériels d'un système informatique.

Cela comprend les CPU, les bus, la mémoire, les périphériques internes au processeur et périphériques externes (ceux de la carte).

La documentation du kernel explique comment utiliser les DT avec Linux :

`/Documentation/devicetree/usage-model.rst`

Les spécifications du format sont quant à elles gérées par :

https://www.devicetree.org/specifications/

Les *Device Trees* sont des fichiers de description. Le langage employé est donc un **langage de description** et non de programmation.

Le Device Tree Compiler (dtc) est un compilateur dédié à ce langage. Nous pouvons constater que le processus de compilation est proche de celui du C :

- `.dts` : Device Tree Source, contient généralement la description de la *board* cible.
- `.dtsi` : Device Tree Source Include, contient généralement la description du processeur ou SoC cible.
- `.dtb` : Device Tree Blob (Binary Large Object), binaire de sortie généré par le Device Tree Compiler.
- `.dtbo` : DTB Object, DT pré-compilé et chargeable dynamiquement en *user-space*. Se rapproche de la notion de *firmware*.

Une fois le dtb chargé en mémoire, on parle de *Flattened Device Tree* (FDT).

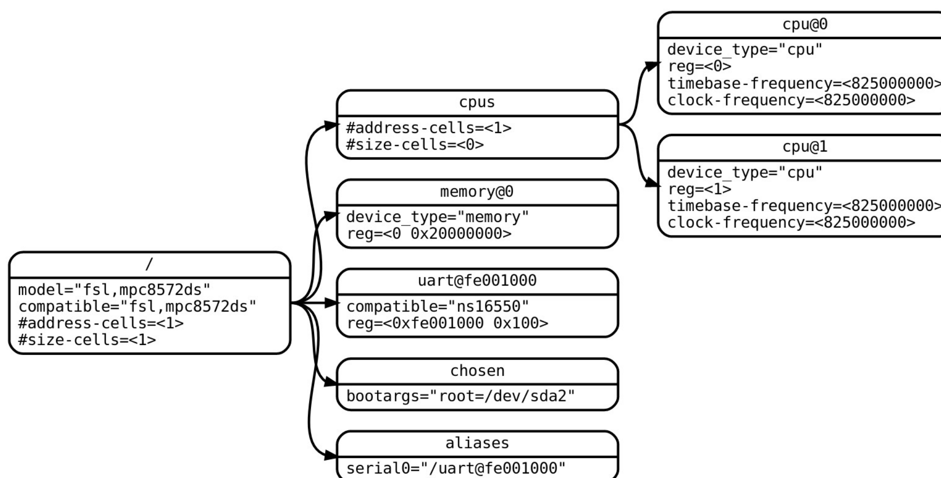
L'idée principale du Device Tree est de partir d'un kernel complètement indépendant de l'architecture matérielle cible (portabilité, interopérabilité), puis de fournir une description du-dit matériel en argument au lancement du kernel.

Le principal avantage est que la modification de la configuration matérielle n'impacte que le dtb (10-100 ko) et ne demande pas la recompilation complète du kernel.

La représentation en Device Tree est aujourd'hui répandue pour les systèmes embarqués sur lesquels tourne Linux : PowerPC, ARM, RISC-V, ...

En revanche, ceci ne concerne pas les ordinateurs classiques (architectures x86 et x64) puisque ceux-ci utilisent plutôt des protocoles de détection du matériel (ex. ACPI).

Comme son nom l'indique, le DT est une arborescence.



https://github.com/devicetree-org/devicetree-specification/releases/tag/v0.4-rc1

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Exemple de syntaxe du Device Tree

- Tree of **nodes**
- Nodes with **properties**
- Node ≈ a device or IP block
- Properties ≈ device characteristics
- Notion of **cells** in property values
- Notion of **phandle** to point to other nodes
- `dtc` only does syntax checking, no semantic validation

```
/ {
    node@0 {
        a-string-property = "A string";
        a-string-list-property = "first string", "second string";
        a-byte-data-property = [0x01 0x23 0x34 0x56];

        child-node@0 {
            first-child-property;
            second-child-property = <1>;
            a-reference-to-something = <&node1>;
        };

        child-node@1 {
        };
    };

    node1: node@1 {
        an-empty-property;
        a-cell-property = <1 2 3 4>;

        child-node@0 {
        };
    };
};
```


« Device Tree 101 », Février 2021, thomas.petazzoni@bootlin.com , https://bootlin.com/docs/

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Observons les DT pour les processeurs ARM depuis les sources du kernel :

`/arch/arm/boot/dts/`

Mode	Name
-rw-r--r--	Makefile
-rw-r--r--	aks-cdu.dts
-rw-r--r--	alphascale-asm9260-devkit.dts
-rw-r--r--	alphascale-asm9260.dtsi
-rw-r--r--	alpine-db.dts
-rw-r--r--	alpine.dtsi
-rw-r--r--	am335x-baltos-ir2110.dts
-rw-r--r--	am335x-baltos-ir3220.dts
-rw-r--r--	am335x-baltos-ir5221.dts
-rw-r--r--	am335x-baltos-leds.dtsi
-rw-r--r--	am335x-baltos.dtsi
-rw-r--r--	am335x-base0033.dts
-rw-r--r--	am335x-bone-common.dtsi
-rw-r--r--	am335x-bone.dts
-rw-r--r--	am335x-boneblack-common.dtsi
-rw-r--r--	am335x-boneblack-hdmi.dtsi
-rw-r--r--	am335x-boneblack-wireless.dts
-rw-r--r--	am335x-boneblack.dts
-rw-r--r--	am335x-boneblue.dts
-rw-r--r--	am335x-bonegreen-common.dtsi
-rw-r--r--	am335x-bonegreen-wireless.dts
-rw-r--r--	am335x-bonegreen.dts
-rw-r--r--	am335x-chiliboard.dts
-rw-r--r--	am335x-chilisom.dtsi
-rw-r--r--	am335x-cm-t335.dts

```
7   #include "am33xx.dtsi"
8   #include "am335x-bone-common.dtsi"
9   #include "am335x-boneblack-common.dtsi"
10  #include "am335x-boneblack-hdmi.dtsi"
11
12  / {
13          model = "TI AM335x BeagleBone Black";
14          compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
15  };
16
17  &cpu0_opp_table {
18          /*
19           * All PG 2.0 silicon may not support 1GHz but some of the early
20           * BeagleBone Blacks have PG 2.0 silicon which is guaranteed
21           * to support 1GHz OPP so enable it for PG 2.0 on this board.
22           */
23          oppnitro-1000000000 {
24                  opp-supported-hw = <0x06 0x0100>;
25          };
26  };
27
28  &gpio0 {
29          gpio-line-names =
30                  "[mdio_data]",
31                  "[mdio_clk]",
32                  "P9_22 [spi0_sclk]",
33                  "P9_21 [spi0_d0]",
34                  "P9_18 [spi0_d1]",
35                  "P9_17 [spi0_cs0]",
36                  "[mmc0_cd]",
37                  "P8_42A [ecappwm0]",
38                  "P8_35 [lcd d12]",
39                  "P8_33 [lcd d13]",
40                  "P8_31 [lcd d14]",
41                  "P8_32 [lcd d15]",
42                  "P9_20 [i2c2_sda]",
43                  "P9_19 [i2c2_scl]",
44                  "P9_26 [uart1_rxd]",
45                  "P9_24 [uart1_txd]",
46                  "[rmii1_txd3]",
47                  "[rmii1_txd2]",
48                  "[usb0_drvvbus]",
49                  "[hdmi_cec]",
```

Revenons (très brièvement) à l'exemple de TP :

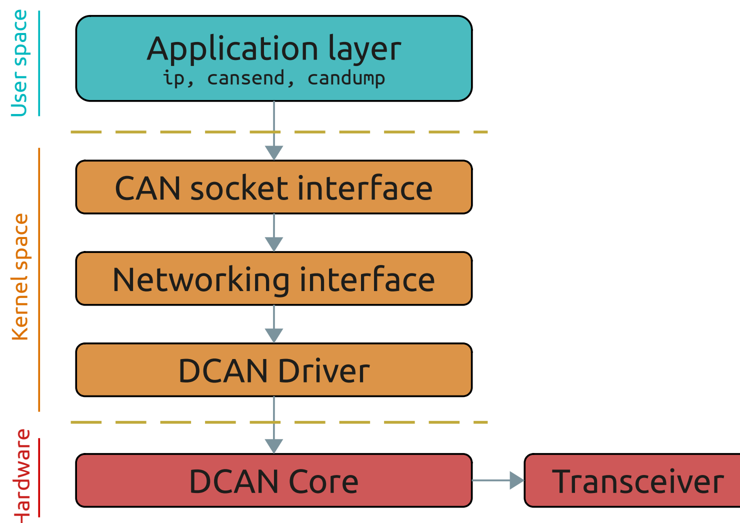
Nous voulons activer le périphérique CAN pour notre processeur AM3358 sur BeagleBone Black.

Il faut pour cela naviguer entre les différents fichiers du DT et la documentation du processeur. Tout cela sera vu en TP.

À la fin, on obtiendra un nouveau dtb à charger aux côté du kernel. Le composant CAN sera alors reconnu par le kernel.

```
root@arm:/home/debian# ifconfig -a
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          UP NOARP  MTU:16  Metric:1
          RX packets:3 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:24 (24.0 B)  TX bytes:0 (0.0 B)
          Interrupt:71

eth0      Link encap:Ethernet  HWaddr 90:59:af:50:39:d0
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:56
```


Avec le DTB permettant au kernel de reconnaître le périphérique CAN, nous pouvons désormais communiquer avec d'autres composants CAN, grâce aux logiciels utilisant l'interface kernel socketCAN.



COMPILATION

Étape 5

5. COMPILATION

Considérons qu'on a déjà récupéré notre chaîne de compilation croisée, et que son chemin est désigné par la variable d'environnement CC.

```
make ARCH=arm distclean
make ARCH=arm CROSS_COMPILE=${CC} menuconfig            → <kernel>/.config
make ARCH=arm CROSS_COMPILE=${CC} zImage -j16           → <kernel>/arch/arm/boot/zImage
make ARCH=arm CROSS_COMPILE=${CC} am335x-boneblack.dtb  → <kernel>/arch/arm/boot/dts/am335x-boneblack.dtb
```

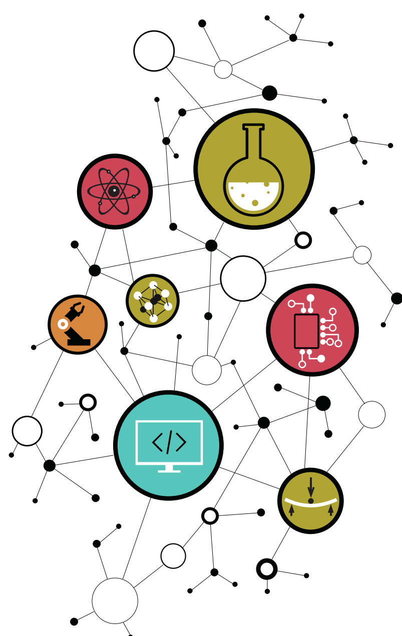
Si on était amené à changer soit les services du kernel, soit l'architecture matérielle sur laquelle porter l'OS, alors il suffirait de n'effectuer que la commande correspondante.

DÉPLOIEMENT SUR CIBLE

Étape 6

Trop archi-spécifique, sera vu en TP

CONTACT



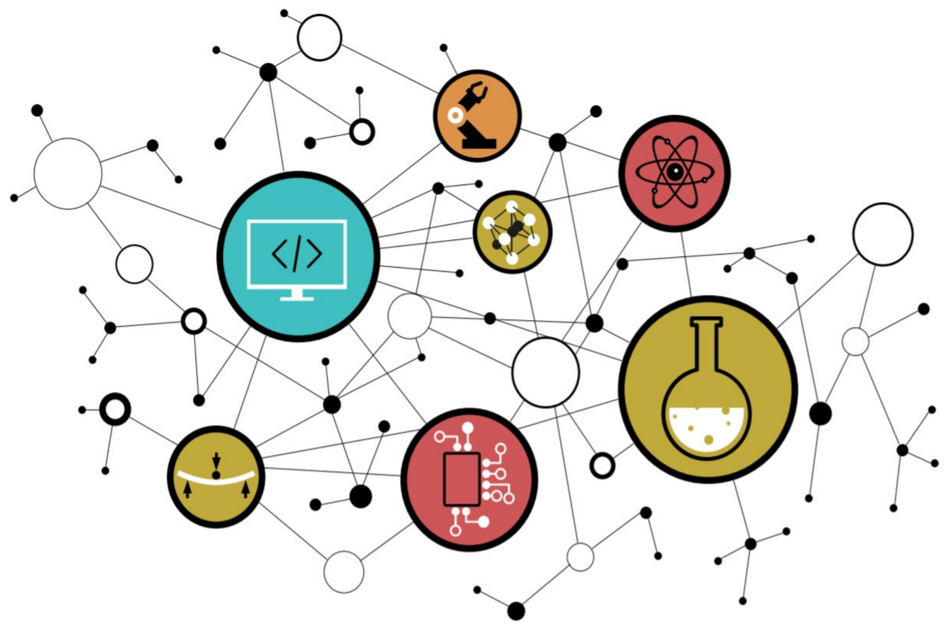
Dimitri Boudier – PRAG ENSICAEN
dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)

Except where otherwise noted, this work is licensed under
https://creativecommons.org/licenses/by-nc-sa/3.0/

BSP LINUX
BOARD SUPPORT PACKAGE



ENSICAEN
Embedded Systems

BOARD SUPPORT PACKAGE

Embedded Linux

Hugo Descoubes – mars 2014

Une BSP Linux (Board Support Package) est un système d'exploitation complet combinant un jeu de composants logiciel en user space, des librairies, des drivers et services kernel prêts à l'emploi pour une plateforme matérielle donnée (interfaces matérielles de la board connues). *Ceci est à comparer aux distributions dans le monde du PC qui sont censées tourner sur n'importe qu'elle machine au monde (souvent x86/x64).*

La Beagle Bone est une plateforme d'évaluation, il est néanmoins possible de travailler sur de boards durcies (souvent fournies avec BSP) pour une future intégration en milieu industriel (ex. EOLANE sur Caen) :

Le marché Allemand étant beaucoup plus riche que celui français sur le domaine des BSP's complètes Linux/Windows/Android (BSP essentiellement sur famille i.MX6 Freescale et AM335x de TI). Un grand nombre d'acteurs Allemands sont présents sur ce secteur, notamment PHYTEC :

Pour information, Freescale propose la famille i.MX6 de processeurs qui est extrêmement bien pensé (single/dual/quad cores), très modulable, software et pins compatible :

Red indicates change from column to the left

i.MX 6SoloLite
- Single ARM® Cortex™- A9 at 1.0GHz
- 256KB L2 cache, Neon, VFPvd16, Trustzone
- 2D graphics
- 32-bit DDR3 and LPDDR2 at 400MHz
- Integrated EPD controller

i.MX 6Solo
- Single ARM Cortex-A9 at 1.0GHz
- 512KB L2 cache, Neon, VFPvd16, Trustzone
- 3D graphics with 1 shader
- 2D graphics
- 32-bit DDR3 and LPDDR2 at 400MHz
- Integrated EPD controller

i.MX 6DualLite
- Dual ARM Cortex-A9 at 1.0GHz
- 512KB L2 cache, Neon, VFPvd16, Trustzone
- 3D graphics with 1 shader
- 2D graphics
- 64-bit DDR3 and 2-channel 32-bit LPDDR2 at 400MHz
- Integrated EPD controller

i.MX 6Dual
- Dual ARM Cortex-A9 at 1/1.2GHz
- 1 MB L2 cache, Neon, VFPvd16, Trustzone
- 3D graphics with 4 shaders
- Two 2D graphics engines
- 64-bit DDR3 and 2-channel 32-bit LPDDR2 at 533MHz
- Integrated SATA-II

i.MX 6Quad
- Quad ARM Cortex-A9 at 1.2GHz
- 1 MB L2 cache, Neon, VFPvd16, Trustzone
- 3D graphics with 4 shaders
- Two 2D graphics engines
- 64-bit DDR3 and 2-channel 32-bit LPDDR2 at 533MHz
- Integrated SATA-II

Pin-to-pin Compatible

Software Compatible

Afin de créer un système Linux complet plusieurs solutions sont possibles :

- ***Distribution précompilée :*** *rapide à mettre en œuvre mais manque de flexibilité. Seules certaines architectures sont supportées. Ce que nous avons fait jusqu'à présent.*

- ***Compilation manuelle des composants :*** *grande flexibilité mais exercice long, fastidieux et non reproductible qui nécessite une gestion manuelle des dépendances inter-paquets.*

- ***Outils d'automatisation de génération de système :*** *solution flexible et reproductible compilant directement les sources de chaque package (BuildRoot, OpenEmbedded, Yocto …)*

*Plaçons nous dans le cas d'une BSP sur laquelle nous souhaitons rajouter quelques services, pour une phase de prototypage par exemple. Première solution, passer par un gestionnaire de paquets, si le package est proposé (**apt-get** sous Debian, **opk** sous Angstrom, **yum** sous Red Hat …). Bien s'assurer de la configuration réseau de la cible (proxy système, serveur DNS …) :*

/etc/network/interfaces

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
        address 172.24.123.13
        netmask 255.255.0.0
        gateway 172.24.0.1
        dns-nameservers 193.49.200.14
```

/etc/network/ifstate

```
lo=lo
eth0=eth0
service restart networking
```

/etc/resolv.conf *(vérifier config. serveur DNS)*

```
domain localdomain
search localdomain
nameserver 193.49.200.14
```

```
export http_proxy=http://proxy.ensicaen.fr:3128
export ftp_proxy=http://proxy.ensicaen.fr:3128
```

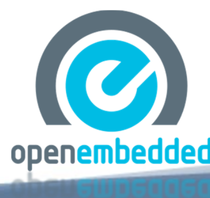

*Beaucoup de packages nécessites la récupération et la recompilation des sources, c'est le cas pour **can-utils** (gestion du nombre de plateformes cibles et de versions quasi impossible à maintenir). Néanmoins, si vous avez besoin d'installer un grand nombre de paquets, cette solution peut devenir très longue et fastidieuse, notamment pour des problématiques de gestion de version et de dépendance entre packages (toujours regarder les dates de mise à jour des solutions en ligne). Observons les dépendances nécessaires afin d'installer can-utils (http://baydogar.blogspot.fr/2013/05/cross-compiling-can-utils.html) :*

- ***iproute** (ici, vs 2.6.39)*
- ***libsocketcan** (ici, vs 0.0.8)*
- ***canutils** (ici, vs 3.0.2)*

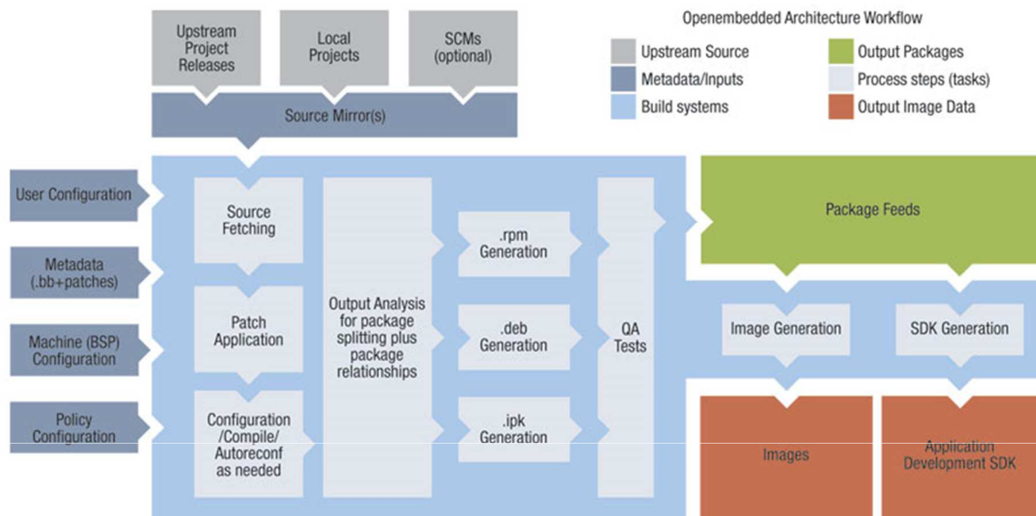
Il existe sinon plusieurs solution permettant l'automatisation du processus de génération de systèmes Linux complets :

- ***BuildRoot** (http://buildroot.uclibc.org) : outil simple et efficace de génération de systèmes Linux basé sur la commande **make**. Configuration via interface **ncurses** (cf. make menuconfig) bien adaptée au développement de petits systèmes. BR manque néanmoins d'un gestionnaire dynamique de packages (souvent inutile pour une application embarquée) et supporte environ 1000 paquets en 2014, peu être problématique pour la génération de grosses distributions, notamment pour le monde du PC.*

- ***OpenEmbedded** (http://www.openembedded.org/) : outil puissant mais plus lourd à prendre en main que BuildRoot adapté à la génération de tout type de distribution basé sur le moteur d'exécution et de gestion de métadonnées **BitBake** (moteur écrit en python à comparer à make). OpenEmbedded travaille avec un ensemble de **métadonnées** composées de fichiers de configuration, de classes et de recettes/recipes basé sur le principe de l'héritage. Chaque recette (.bb) décrit les tâches à effectuer afin de construire une image, un package et gérant les dépendances associées (plusieurs milliers de paquets supportés).*

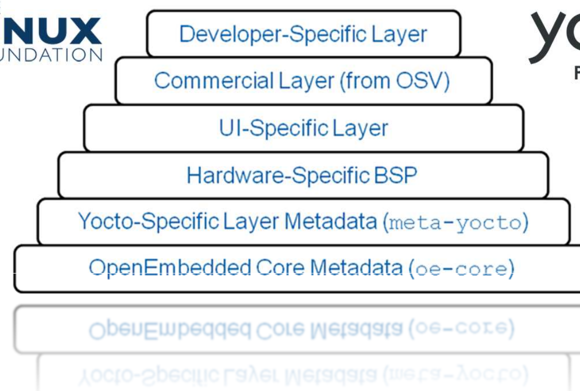


Observons le processus de génération de système ou workflow utilisé par OpenEmbedded :



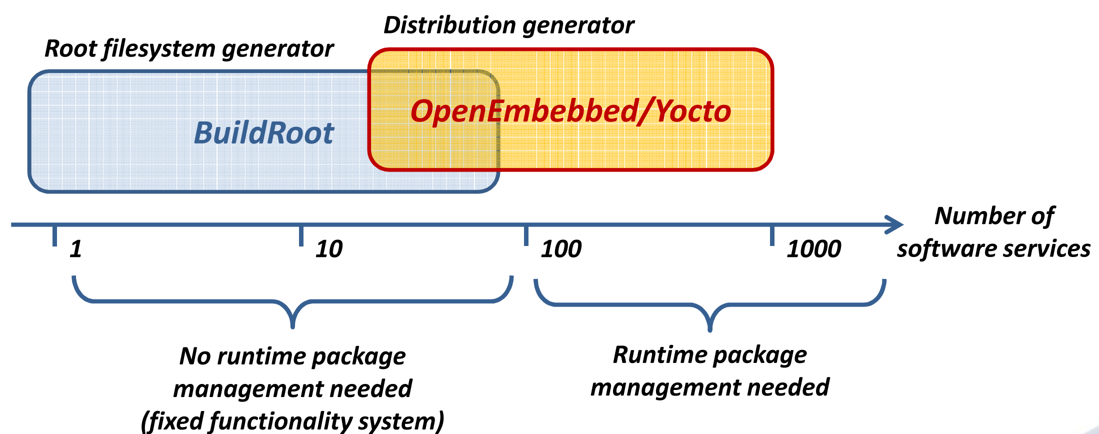
ENSICAEN Embedded Systems
BSP LINUX
Board Support Package – packages – Buildroot – Yocto – autres projets

- ***Yocto*** *(https://www.yoctoproject.org/) : projet OpenSource collaboratif porté par la Linux Foundation dit projet ''chapeau'' (umbrella project) travaillant notamment autour du projet OpenEmbedded (Poky, Eglibc, Matchbox …).*



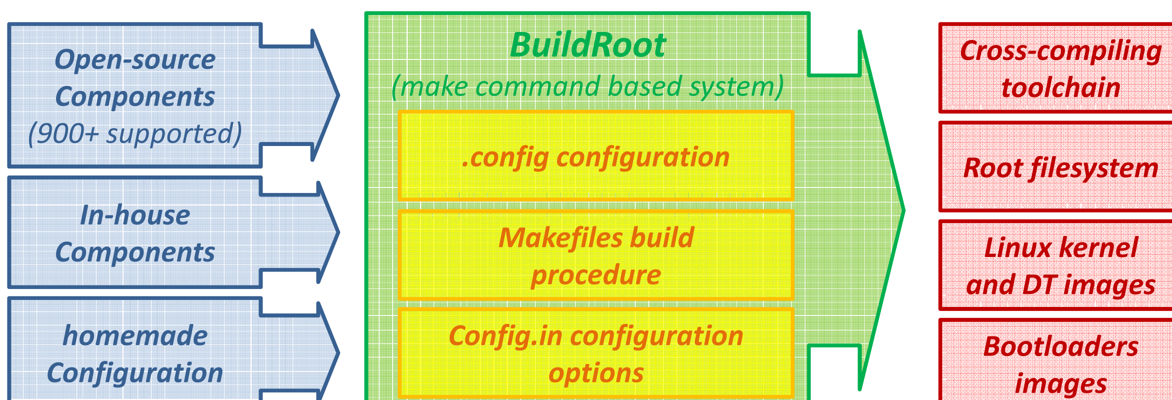
- ***Scratchbox, OpenWRT*** *(dérivé de BuildRoot avec gestion de paquets IPK) …*

En fonction de l'application, du marché ciblé et du nombre de composants logiciels à déployer en user space, nous pouvons adapter l'outil de génération de système à utiliser en fonction de nos besoins :



ENSICAEN Embedded Systems
BSP LINUX
Board Support Package – packages – Buildroot – Yocto – autres projets

*Intéressons-nous maintenant à **BuildRoot**, projet initialement créé en 2001 par la communauté de développeur du projet uCLibc pour la génération de petits systèmes Linux (début réel du projet en tant que tel en 2005). Observons le workflow typique de BuildRoot :*



*Se placer dans le répertoire **~/elinux/work/buildroot/** et récupérer la dernière version de Buildroot sur le site officiel dans la section download (documentation, http://buildroot.uclibc.org/downloads/manual/manual.pdf) :*

```
cd  ~/elinux/work/buildroot/
wget http://buildroot.org/downloads/buildroot-<current-version>.tar.bz2
tar xjf buildroot-<current-version>.tar.bz2
cd buildroot-<current-version>
```

Un certain nombre de packages sont également nécessaires côté host, s'assurer de leur bonne installation :

```
sudo apt-get update
sudo apt-get install \
build-essential gawk bison flex gettext texinfo patch \
gzip bzip2 perl tar wget cpio python unzip rsync
```


Observons succinctement le système de fichiers de BuildRoot. Nous pouvons constater qu'avant la génération d'un premier système, les différents répertoires sont vides et ne contiennent que des fichiers de configuration et d'automatisation de procédures :

- ***arch/*** *: pré-configurations fournies architecture CPU dépendant.*
- ***board/*** *: pré-configurations kernel fournies pour des boards du marché. Permet également de sauver les différents fichiers et composants maison complémentaires propres à notre plateforme de développement (non vu dans cet enseignement).*
- ***output/*** *: répertoire de destination, notamment des images (kernel, device tree, Bootloader …) et rootfs générés.*

- ***configs/*** *: pré-configurations BuildRoot fournies pour des plateformes du marché (beaglebone, raspberryPI, pandaboard …).*
- ***docs/*** *: documentation BuildRoot*
- ***fs/*** *: recettes ou recipes utilisées par BR pour la génération du système de fichiers choisi.*
- ***linux/*** *: recettes ou recipes utilisées par BR pour la génération de l'image kernel.*
- ***boot/*** *: recettes ou recipes utilisées par BR pour la génération du bootloader choisi.*
- *…*

ENSICAEN Embedded Systems
BSP LINUX
Board Support Package – packages – Buildroot – Yocto – autres projets

Appliquer la configuration par défaut donnée pour la beaglebone et ouvrir l'interface ncurse de configuration :

```
cd ~/elinux/work/buildroot/buildroot<current-version>/
make distclean
cp configs/beaglebone_defconfig .config
make menuconfig
cp .config board/beaglebone/br-bbb-config
```

```
/home/vmlinux/Desktop/elinux/work/buildroot/buildroot-2013.11/.config
Buildroot 2013.11 Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->. Hig
Pressing <Y> selectes a feature, while <N> will exclude a feature.
<?> for Help, </> for Search. Legend: [*] feature is selected [

Target options --->
Build options --->
Toolchain --->
System configuration --->
Kernel --->
Target packages --->
Filesystem images --->
Bootloaders --->
Host utilities --->
Legacy config options --->
```

Etudier les configurations proposées et adapter cette configuration à notre projet. Nous n'utiliserons BuildRoot que pour la génération d'un rootfs (minimisation du temps de génération), nous garderons donc dans un projet séparé la récupération, l'application des patches et la compilation du kernel, du device tree, des modules, des firmwares et des deux étages de bootloader.

- ***Target options :*** *description précise de l'architecture CPU, utile afin de lever des options d'optimisation à la compilation. Par exemple :*
  arm-linux-gnueabihf-gcc –mcpu=cortex-a8 –mfpu=vfpv3-d16 –mfloat-abi=hard …

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages
Filesystem images
Bootloaders  ---
Host utilities
Legacy config op
```

https://wiki.linaro.org/WorkingGroups/ToolChain/FAQ

```
Target Architecture (ARM (little endian))  --->
Target Architecture Variant (cortex-A8)  --->
Target ABI (EABIhf)  --->
Floating point strategy (VFPv3-D16)  --->
ARM instruction set (ARM)  --->
```


ENSICAEN
Embedded Systems
BSP LINUX
Board Support Package – packages – Buildroot – Yocto – autres projets

- ***Build options :*** *options de compilation (options d'optimisation, nombre de jobs, différents chemins utiles au projet, librairies statiques ou dynamiques, stripper/épurer les exécutables …). Dans notre cas, modifier le nombre de jobs (nombre de CPU's utilisés côté host pour la compilation).*

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options
```

```
    Commands  --->
($(CONFIG_DIR)/defconfig) Location to save buildroot config
($(TOPDIR)/dl) Download dir
($(BASE_DIR)/host) Host dir
    Mirrors and Download locations  --->
(2) Number of jobs to run simultaneously (0 for auto)
[ ] Enable compiler cache
[ ] Show packages that are deprecated or obsolete
[ ] build packages with debugging symbols
    strip command for binaries on target (strip)  --->
()  executables that should not be stripped
()  directories that should be skipped when stripping
    gcc optimization level (optimize for size)  --->
    *** enabling Stack Smashing Protection requires support
[ ] prefer static libraries
($(TOPDIR)/local.mk) location of a package override file
()  global patch directory
```

- ***Toolchain :*** *choix de la chaîne de cross-compilation ou ABI (interne à BuildRoot ou externe à télécharger ou locale préinstallée) et de la librairie standard associée.*

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options  --->
```

```
    Toolchain type (External toolchain)  --->
    Toolchain (Custom toolchain)  --->
    Toolchain origin (Pre-installed toolchain)  --->
(/home/vmlinux/Desktop/elinux/work/toolchain/linaro/gcc-linaro
(arm-linux-gnueabihf) Toolchain prefix
    External toolchain C library (glibc/eglibc)  --->
[*] Toolchain has RPC support?
[ ] Toolchain has C++ support?
()  Extra toolchain libraries to be copied to target
[ ] Copy gdb server to the Target
[ ] Build cross gdb for the host
[ ] Purge unwanted locales
()  Generate locale data
[*] Enable MMU support
(-pipe) Target Optimizations
()  Target linker options
[ ] Register toolchain within Eclipse Buildroot plug-in
```


Board Support Package – packages – Buildroot – Yocto – autres projets
ENSICAEN
Embedded Systems
BSP LINUX

A une ABI (ARM dans notre cas) utilisée est associée une librairie standard, outil indispensable utilisé par la suite par tout composant logiciel du système. Présentons les 3 principales librairies standards rencontrés dans l'embarqué :

- ***glibc*** *(GNU C Library) : librairie standard du C, la même que celle portée sur des distribution GNU/Linux dans le monde du PC. Elle a pour avantage d'être très riche nativement en services, très robuste et très bien maintenue. Néanmoins, il s'agit de la librairie offrant la plus forte empreinte mémoire. Depuis la version 2 de la glibc (nommée libc6), elle est connue sur système Linux sous le nom libc.so.6 (présente sur le rootfs dans* ***/lib/<target-architecture>/*** *)*

- ***eglibc*** *(Embedded GNU C Library) : variante configurable et optimisée pour l'embarqué de librairie standard du C glibc (en 2009, Debian annonça la migration de la glibc vers la eglibc pour leurs systèmes). eglibc est depuis 2011 un composant du projet Yocto. Cette librairie est un projet très bien maintenue offrant une meilleure empreinte mémoire que la glibc mais néanmoins plus large que uClibc.*

- ***uClibc*** *(microcontroller libc) : projet optimisé pour l'embarqué initialement prévu pour uCLinux (variante de Linux pouvant tourner sur processeur sans MMU). Cette bibliothèque est la plus compacte, elle est hautement configurable au détriment des services proposés et des performances.*

- ***System configuration :*** *configuration du système durant la phase d'amorçage (login, mdp, /dev stratégie de management …). Il faut savoir qu'en user space, le répertoire /dev peu être géré statiquement (plus performant mais moins évolutif) ou dynamiquement (cf. distributions dans le monde du PC, utilise souvent udev).*

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options  --->
```

```
(beagleboneblack) System hostname
(Welcome to Ensicaen BSP) System banner
    Passwords encoding (md5)  --->
    Init system (Busybox)  --->
    /dev management (Dynamic using udev)  --->
(system/device_table.txt) Path to the permission tables
    Root FS skeleton (default target skeleton)  --->
()  Root password
[ ] Run a getty (login prompt) after boot
[ ] remount root filesystem read-write during boot
()  Root filesystem overlay directories
()  Custom scripts to run before creating filesystem images
()  Custom scripts to run after creating filesystem images
```

- ***Kernel :*** *récupération, application des patches et configuration du kernel Linux (compression éventuelle) et du device tree. Cette étape, notamment l'application des patches, pouvant être difficilement configurable en fonction de la stratégie de récupération des patches, nous préférerons garder le processus de compilation du kernel dans un projet annexe.*

- ***Target packages :*** *customisation du rootfs cible. En 2013, près de 900 packages sont supportés par BuildRoot. Rajouter les composants nécessaires à du prototypage d'application CAN (rappel, "/" permet une recherche rapide et une validation des dépendances sous interface ncurses).*

Observons deux des principales stratégies rencontrées afin de gérer un jeu de packages binaires exécutables :

- ***Standard :** chaque package est un objet binaire exécutable indépendant, solution équivalente aux distributions GNU/Linux rencontrées dans le monde du PC.*
- ***BusyBox :** objet binaire exécutable unique, configurable à la compilation et pouvant encapsuler des centaines de commandes systèmes (via liens symboliques). Cette solution permet de minimiser grandement l'empreinte mémoire du jeu de commande cible.*

```
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 ash -> busybox
-rwxr-xr-x  1 vmlinux vmlinux 601288 févr.  8 00:54 busybox
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 cat -> busybox
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 catv -> busybox
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 chattr -> busybox
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 chgrp -> busybox
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 chmod -> busybox
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 chown -> busybox
lrwxrwxrwx  1 vmlinux vmlinux      7 févr.  8 00:43 cp -> busybox
```


- ***Filesystem images :** type d'image du rootfs à générer (ext2, ext3, ext4 …) et compression éventuelle. Dans notre cas, nous générerons un rootfs sous forme d'une simple archive de type tarball sans processus de compression (facilite le déplacement, le partage et l'installation par la suite).*

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options
```

```
[ ] cloop root filesystem for the target device
[ ] cpio the root filesystem (for use as an initial RAM filesystem)
[ ] cramfs root filesystem
[ ] ext2/3/4 root filesystem
    *** initramfs requires a Linux kernel to be built ***
[ ] jffs2 root filesystem
[ ] romfs root filesystem
[ ] squashfs root filesystem
[*] tar the root filesystem
      Compression method (no compression)  --->
()    other random options to pass to tar
[ ] ubifs root filesystem
```

- ***Bootloaders :*** *sélection du bootloader (3rd stage bootloader) et éventuellement du second étage de bootloader, dans notre cas le MLO proposé par TI. Concernant notre projet, nous possédons déjà les sources et un binaire précompilé des différents bootloaders.*

```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options
```

```
[ ] Barebox
[ ] mxs-bootlets
[ ] U-Boot
[ ] X-loader
```



```
Target options  --->
Build options  --->
Toolchain  --->
System configuration  --->
Kernel  --->
Target packages  --->
Filesystem images  --->
Bootloaders  --->
Host utilities  --->
Legacy config options
```

```
[ ] host dfu-util
[ ] host dosfstools
[ ] host e2fsprogs
[ ] host genext2fs
[ ] host genimage
[ ] host genpart
[ ] host lpc3250loader
[ ] host mtools
[ ] host omap-u-boot-utils
[ ] host openocd
[ ] host sam-ba
[ ] host sunxi-tools
[ ] host u-boot tools
```

- ***Host utilities :*** *installation éventuelle d'utilitaires pour la machine host. Non nécessaire dans notre cas.*

- ***Legacy config options :*** *options de configuration BuildRoot retirée entre la version courante et les versions antérieures. Non utilisé dans notre cas.*

*Observons le contenu du répertoire de sortie de BuildRoot (**~/elinux/work/buildroot/buildroot<current-version>/output/**) :*

- ***build :** répertoire correspondant aux points d'extractions et de compilation des packages, et contenant donc les sources.*
- ***host :** répertoire correspondant aux points d'extractions et d'installation d'utilitaires pour le host. **host/usr/<toolchain-prefix>/sysroot/** contient le sysroot utilisé par la toolchain.*
- ***staging :** lien symbolique vers le **sysroot** (rootfs-like minimaliste utilisé par la chaîne de cross-compilation pour les fichiers d'en-tête et les bibliothèques)*
- ***target :** point d'installation des bibliothèques et composants cibles*
- ***toolchain :** vide si utilisation d'une toolchain externe*
- ***images :** répertoire d'installation du rootfs (tarball, ext2, ext4 …) et images binaires pour la cible (kernel, device tree, bootloaders …)*

ENSICAEN
Embedded Systems
BSP LINUX
Board Support Package – packages – Buildroot – Yocto – autres projets

Une fois la configuration effectuée, la sauvegarder côté host et lancer le processus d'exécution. Extraire ensuite le rootfs généré vers la MMC/SDcard :

```
cp .config  board/beaglebone/br-bbb-config
make
cd output/images/
tar –xvf rootfs.tar –C /media/rootfs
```

Modifier le fichier de configuration uEnv.txt de U-Boot en adaptant le point d'entrée du rootfs à notre système :

```
optargs=quiet init=/sbin/init
```

*Vous constaterez une erreur au démarrage du système, le processus **init** ne trouve pas le chemin vers les bibliothèques dynamiques :*

```
[    0.864441] pinctrl-single 44e10800.pinmux: could not request pin 21 on device pinctrl-single
/sbin/init: error while loading shared libraries: libc.so.6: cannot open shared object file: No such file
[    1.210266] Kernel panic - not syncing: Attempted to kill init! exitcode=0x00007f00
```

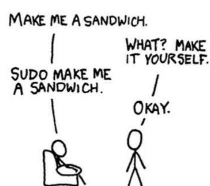
*Importer le **sysroot** utilisé par la chaîne de compilation vers le rootfs précédemment généré. Rebooter le système … et vous voilà sur votre premier système Linux créé from scratch !*

```
cd  ~/elinux/work/buildroot/buildroot<current-version>/output/staging/
tar –vcf sysroot.tar .
cd ../images/
mkdir  rootfs
tar –xvf rootfs.tar –C rootfs/
tar –xvf ../staging/sysroot.tar –C rootfs/
cd rootfs/
tar –vcf rootfs.tar .
tar –xvf rootfs.tar –C /media/rootfs
```


*A ce stade là, il ne reste plus qu'à croiser les doigts et tester les composants installés. Commençons par les services **canutils** (maybe in few minutes it's coffee time … again) :*

http://www.armadeus.com/wiki/index.php?title=CAN_bus_Linux_driver

```
Welcome to Ensicaen BSP
beagleboneblack login: root
# cansend can0 500#R
write: Network is down
# ip link set can0 up type can bitrate 125000
# ifconfig can0 up
# cansend can0 500#R
# can
can-calc-bit-timing  cangen          cansend
canbusload           cangw           cansniffer
candump              canlogserver
canfdtest            canplayer
```



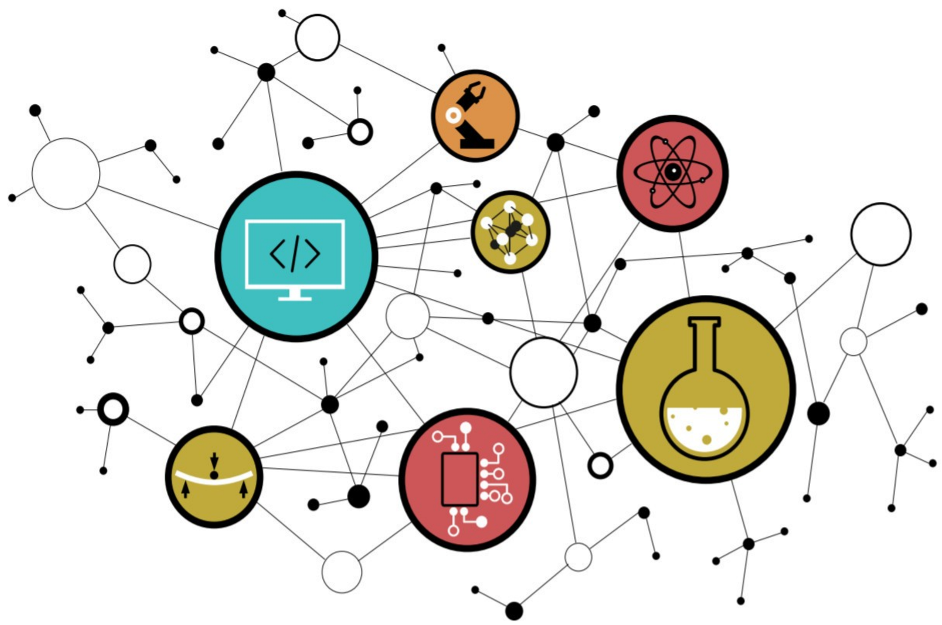
A faire :

- ***Partie sur Yocto :*** *Le projet manquant encore un peu de maturité et d'investissement de la part de certains acteurs en début 2014, notamment Freescale, cette partie sera créée dans un futur proche mais avec un peu de prudence.*

ENSICAEN
Embedded Systems

Merci de votre attention !

LINUX TEMPS REEL



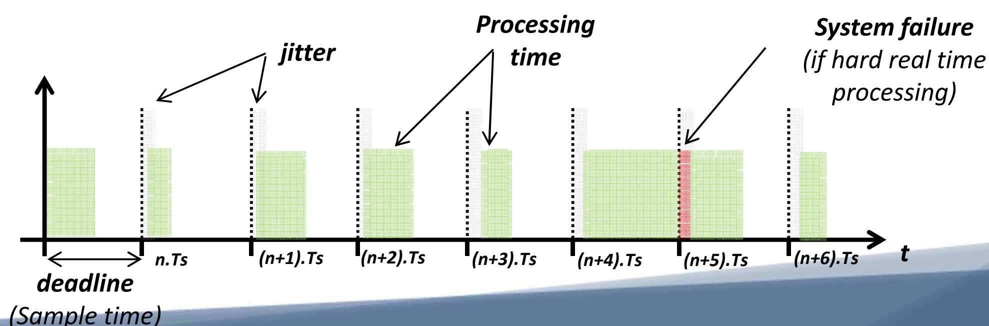
ENSICAEN
Embedded Systems

REAL TIME LINUX

Embedded Linux

Hugo Descoubes – mars 2014

Un système temps réel doit satisfaire à des contraintes temporelles et garantir un déterminisme logique et temporel à l'exécution. *Attention, temps réel ne veut pas forcément dire rapide (régulation de température, régulation de procédés chimiques, applications climatiques …). Une solution technologique se voulant temps réel est le plus souvent la résultante d'une alchimie entre choix de la solution matérielle (MCU, FPGA, SoC, GPP … cache-less, out-of-order-engine CPU …) et logicielle portée (OS-less, OS, RTOS) :*



Rappelons les différentes hiérarchies et terminologies de temps réel cohabitant souvent dans une même application :

- ***Temps réel dur (Hard Real Time) :*** *échéance temporelle vitale/critique à garantir (assistance au freinage, guidage de missile, pacemaker …)*

- ***Temps réel ferme (Firm Real Time) :*** *échéance temporelle à garantir mais non vitale, résultat inutile si deadline dépassée mais applicatif toujours viable (vidéo, audio …)*

- ***Temps réel mou (Soft Real Time) :*** *échéance à garantir mais réponses tardives et latences tolérées (IHM, connectivités …)*

- ***Non temps réel :*** *échéance non garantie (debug, profilage …)*

Rappelons que dans le cadre de développement d'applications pouvant être temps réel, plusieurs solutions s'offrent au développeur :

- ***Séquenceur (homemade) :*** *Système OS-less sans OS/RTOS avec fonction ''main'' implémentant un séquenceur cyclique (exemple de l'offline scheduling). Solutions peut évolutives et pouvant être difficiles à maintenir.*

- ***Real Time Operating System (exécutif temps réel) :*** *exécutifs proposant le plus souvent très peut de services (scheduling, parfois stack IP, file system … souvent sans gestion de MMU et drivers non inclus) mais offrant une empreinte mémoire faible (on-chip memory) ainsi qu'un grand déterminisme temporel avec très peu de jitter. Solutions évolutives et permettant un coût matériel global réduit. Exemple :* ***FreeRTOS, RTX, VxWorks, uC/OS-III, SYS/BIOS …***

- ***Standard Operating System :*** *les systèmes d'exploitation dans le monde du PC (famille Windows, MAC OS X, distro GNU/Linux …) sont des systèmes temps réel mou. Toute application lancée aura, tôt ou tard, sa chance afin d'accéder à une ressource CPU, ceci est du à la politique d'ordonnancement (scheduling policy) de ces systèmes et à la nature non vitale/critique des applicatifs.*

Dans ce chapitre, nous allons nous intéresser à rendre un système GNU/Linux temps réel puis nous validerons les performances de notre système à l'aide d'outils de benchmarking.

ENSICAEN Embedded Systems
REAL TIME LINUX
Temps Réel – Solutions – patch PREMPT-RT – Xenomai - Benchmarking

Observons les principaux acteurs du marché proposant une évolution vers une solutions Linux temps réel :

- ***Xenomai*** *(http://www.xenomai.org/)* ***:*** *solution proposant un co-noyau temps réel (nucleus) cohabitant avec le noyau standard Linux. Xenomai est porté sur un hyperviseur (Adeos ou Adaptative Domain Environment for Operating Systems) ou nanokernel HAL (Hardware Abstraction Layer). Cet hyperviseur assure notamment le routage des interruptions des couches matérielles vers les systèmes ou domaines portés (Interrupt Pipe). Plusieurs API temps réel ou skins sont accessibles. Néanmoins, des skins maison peuvent être développées même si POSIX reste à privilégier pour des soucis de portabilité. En 2003, Xenomai fusionne avec le projet RTAI pour devenir un projet indépendant en 2005.*

- ***Linux-rt ou patch PREEMPT-RT*** *(https://rt.wiki.kernel.org)* ***:*** *branche du kernel proposant un patch pour le noyau linux standard permettant de le rendre temps réel. Le patch PREEMPT-RT influe sur la politique d'ordonnancement en rendant notamment préemptible la majeure du code du noyau et mettant par exemple en œuvre des mécanisme de d'héritage de priorité pour la gestion des sémaphores et des spinlocks (API kernel).*

- ***RTAI (Real Time Application Interface) :*** *extension temps réel du noyau travaillant également avec l'hyperviseur Adeos (cf. Xenomai). Il s'agit d'un projet universitaire initialement conçu comme une variante de RTLinux (projet maintenu puis abandonné en 2011 par Wind River).*

Avant d'installer nos premiers systèmes Linux temps réel, validons les outils de test sur un système standard. Nous utiliserons les utilitaires ***rt-tests*** *et notamment l'outil* ***Cyclictest*** *(cf. https://rt.wiki.kernel.org/index.php/Cyclictest). Récupérer les sources des utilitaires, les cross-compiler et installer l'utilitaire Cyclictest dans le rootfs de la BBB :*

Côté host

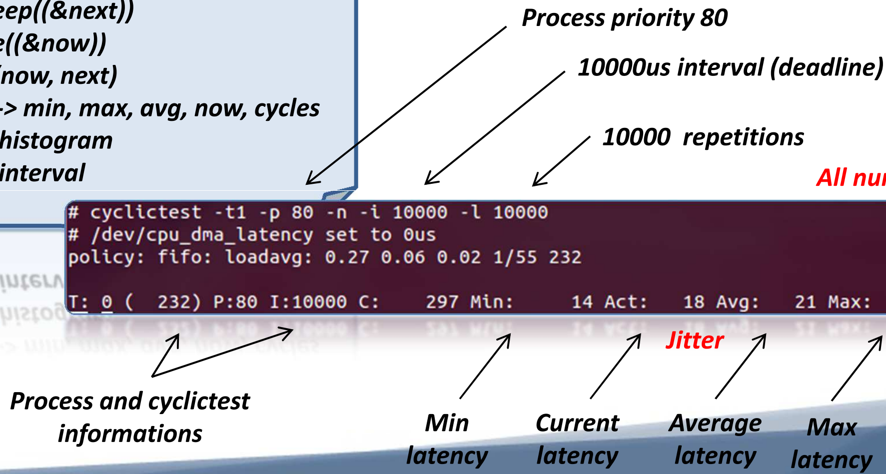
```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git
cd rt-tests
export PATH=/<cross_compiler_path>/bin/:$PATH
export CC=arm-linux-gnueabihf-gcc
make CROSS_COMPILE=arm-linux-gnueabihf- all
cp ./cyclictest /media/rootfs/usr/bin/
```

Côté BBB

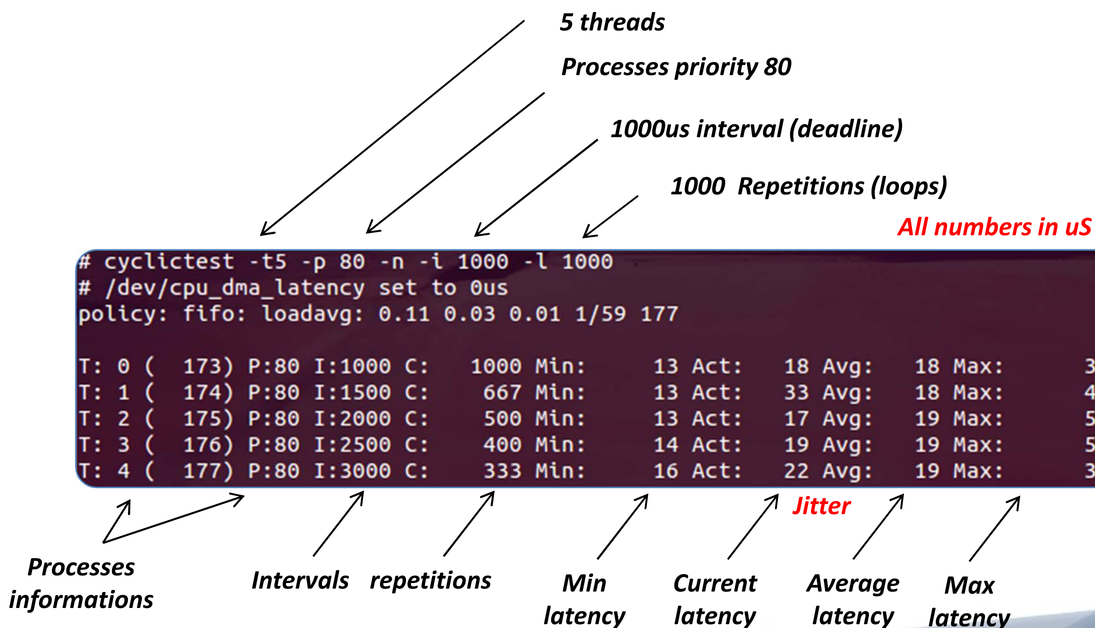
```
cyclictest –help
cyclictest -t1 -p 80 -n -i 10000 -l 10000
```

Observons en pseudo code le travail de stress réalisé par l'utilitaire Cyclictest et analysons les informations de sortie :

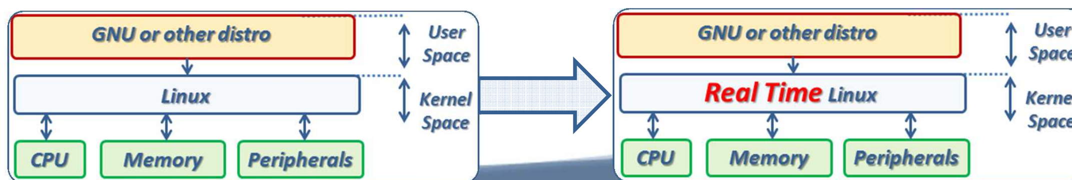
```
clock_gettime((&now))
next = now + interval

while (!shutdown) {
        clock_nanosleep((&next))
        clock_gettime((&now))
        diff = calcdiff(now, next)
        # update stat-> min, max, avg, now, cycles
        # Update the histogram
        next = now + interval
}
```



Prenons un exemple de test :



*Le patch **PREMPT-RT** consiste en l'application d'une série de patchs au kernel ainsi qu'en la modification du fichier de configuration du noyau. Les patchs officiels pour PREEMPT-RT sont directement accessible sur kernel.org (https://www.kernel.org/pub/linux/kernel/projects/rt/). Nous constaterons que cette solution est très simple et rapide à appliquer et ne nécessite aucune couche de virtualisation supplémentaire (cf. Adeos). Le système résultant est plus léger qu'avec Xenomai/RTAI. De plus, aucune API user space spécifique n'est nécessaire (API POSIX standard), cette solution ne nécessite donc aucune modification des applicatifs déjà développés en user space.*



ENSICAEN Embedded Systems
REAL TIME LINUX
Temps Réel – Solutions – patch PREMPT-RT – Xenomai - Benchmarking

*Se rendre sur le wiki officiel de PEEMPT-RT (https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO), récupérer la dernière version stable des patchs associée à notre release de kernel (3.8.13) et les appliquer. Se placer dans le répertoire du kernel **~/elinux/work/kernel/kernel/** :*

```
cd ~/elinux/work/kernel/kernel/
wget http://www.kernel.org/pub/linux/kernel/projects/rt/3.8/stable/patch-3.8.13.13.bz2
cd kernel
bzcat ../patch-3.8.13.13.bz2 | patch -p1
```

Par curiosité, extraire une archive des patchs appliqués et observer le type de modifications appliquées au code kernel :

```
--- a/arch/sparc/kernel/prom_common.c
+++ b/arch/sparc/kernel/prom_common.c
@@ -64,7 +64,7 @@ int of_set_property(struct device_node *dp,
	err = -ENODEV;

	mutex_lock(&of_set_property_mutex);
-	write_lock(&devtree_lock);
+	raw_spin_lock(&devtree_lock);
```

Nous allons maintenant modifier le fichier de configuration du kernel avant de relancer une compilation complète. Suivre le tutoriel en ligne sur le site officiel du patch PREEMPT-RT. Attention, certaines des configurations à appliquer sont dépendantes de l'architecture cible, l'exemple donné en ligne est pour une architecture x86 (architectures les mieux supportées par le patch en 2014). Activer les services suivants (interface ncurse, recherche avec /) :

- ***PREEMPT** : modèle de préemption du kernel*

```
( ) No Forced Preemption (Server)
( ) Voluntary Kernel Preemption (Desktop)
(X) Preemptible Kernel (Low-Latency Desktop)
```

- ***PREEMPT_RCU** (read Copy Update, mécanisme de type exclusion mutuelle), **TREE_PREEMPT_RCU, RCU_BOOST, RCU_BOOST_PRIO=1, RCU_BOOST_DELAY=500,***

Il me reste à faire :

- *Benchmark sur jitter avec et sans charge (cyclictest et hackbench)*
- *Partie Xenomai : présentation, procédure d'installation, mise en œuvre sur BBB, benchmark, API skins, dév d'une application (CAN)*

ENSICAEN
Embedded Systems

Merci de votre attention !

PRISE DE NOTES

notes

notes

PRISE DE NOTES

notes

notes