Chapitre 3

Linux from scratch

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

2021-2022

Disclaimer

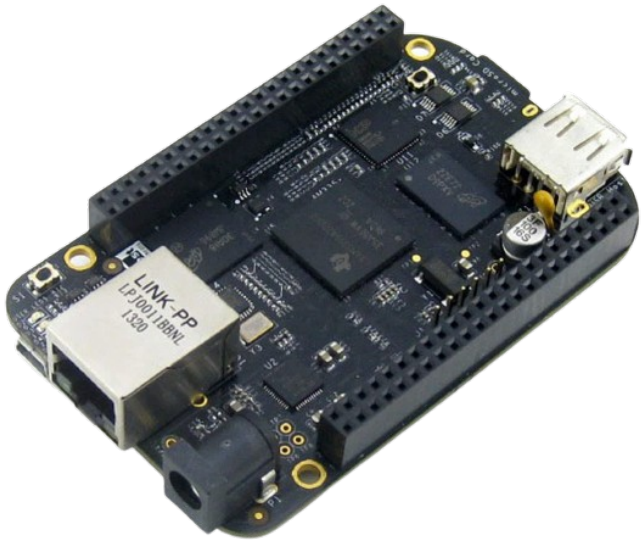
Nous aborderons dans ce chapitre les principes de la création d'un système ***Linux From Scratch***, comprenez "en partant des sources".

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.



Disclaimer

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

ENSI CAEN

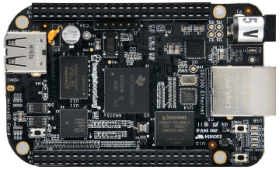
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

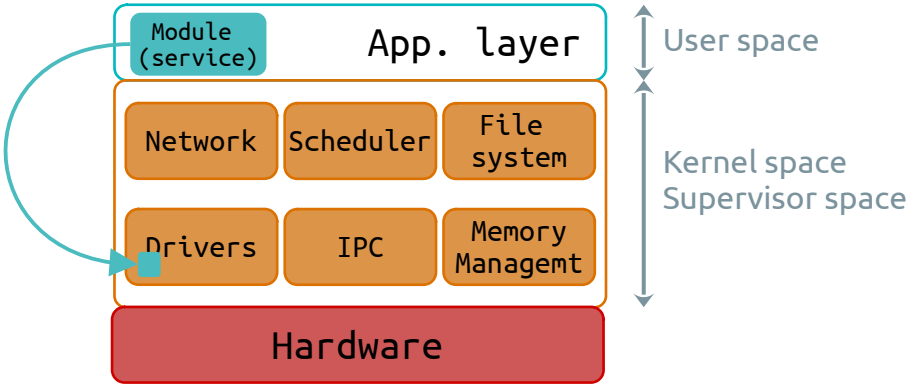
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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 …



Fichier de configuration `.config`

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>`

Fichier de configuration `.config`

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`

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

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
```

Fichier de configuration `.config`

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) !

```
#
# Automatically generated file; DO NOT EDIT.
# Linux/x86 5.13.0-28-generic Kernel Configuration
#
CONFIG_CC_VERSION_TEXT="gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=90300
CONFIG_CLANG_VERSION=0
CONFIG_AS_IS_GNU=y
CONFIG_AS_VERSION=23400
CONFIG_LD_IS_BFD=y
```

```
#
# Performance monitoring
#
CONFIG_PERF_EVENTS_INTEL_UNCORE=y
CONFIG_PERF_EVENTS_INTEL_RAPL=m
CONFIG_PERF_EVENTS_INTEL_CSTATE=m
# CONFIG_PERF_EVENTS_AMD_POWER is not set
# end of Performance monitoring
```

`y` = service monolithique

`m` = service modulaire

`# CONFIG_*** is not set` = service non compilé

Fichier de configuration `.config`

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.

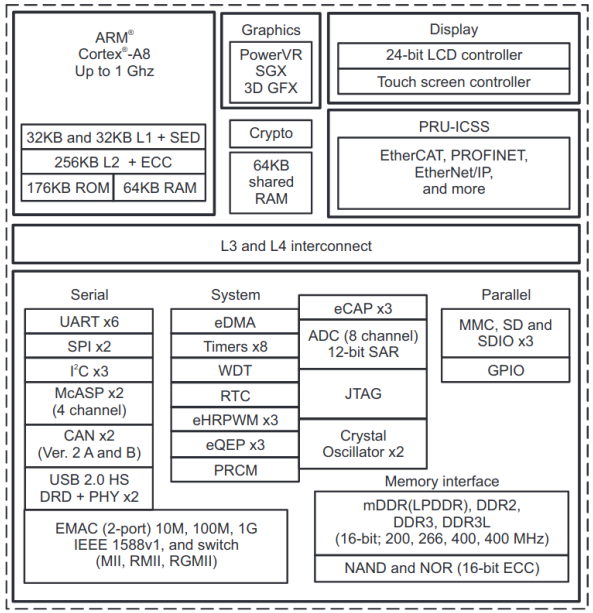


Figure 1-1. AM335x Functional Block Diagram

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

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

Exemple : service CAN

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

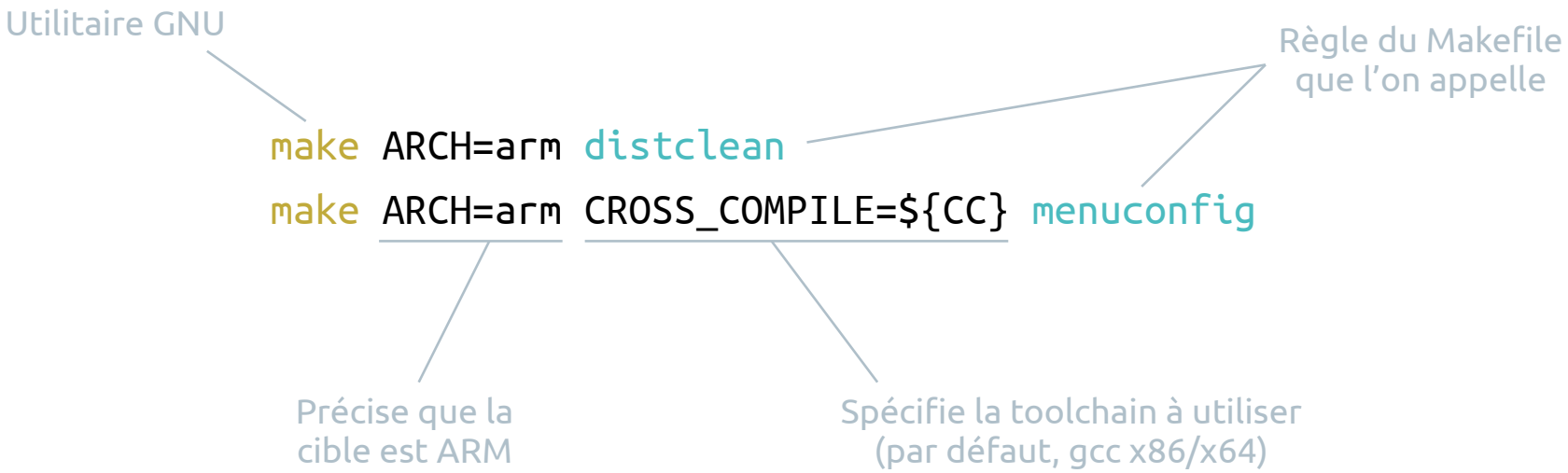
```
[*] Networking support ->
   <*|M> CAN bus subsystem support ->
      <*|M> Raw CAN Protocol (raw access with CAN-ID filtering)
      <*|M> Broadcast Manager CAN Protocol (with content filtering)
      <*|M> CAN Gateway/Router (with netlink configuration)
         CAN Device Drivers ->
            <*|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

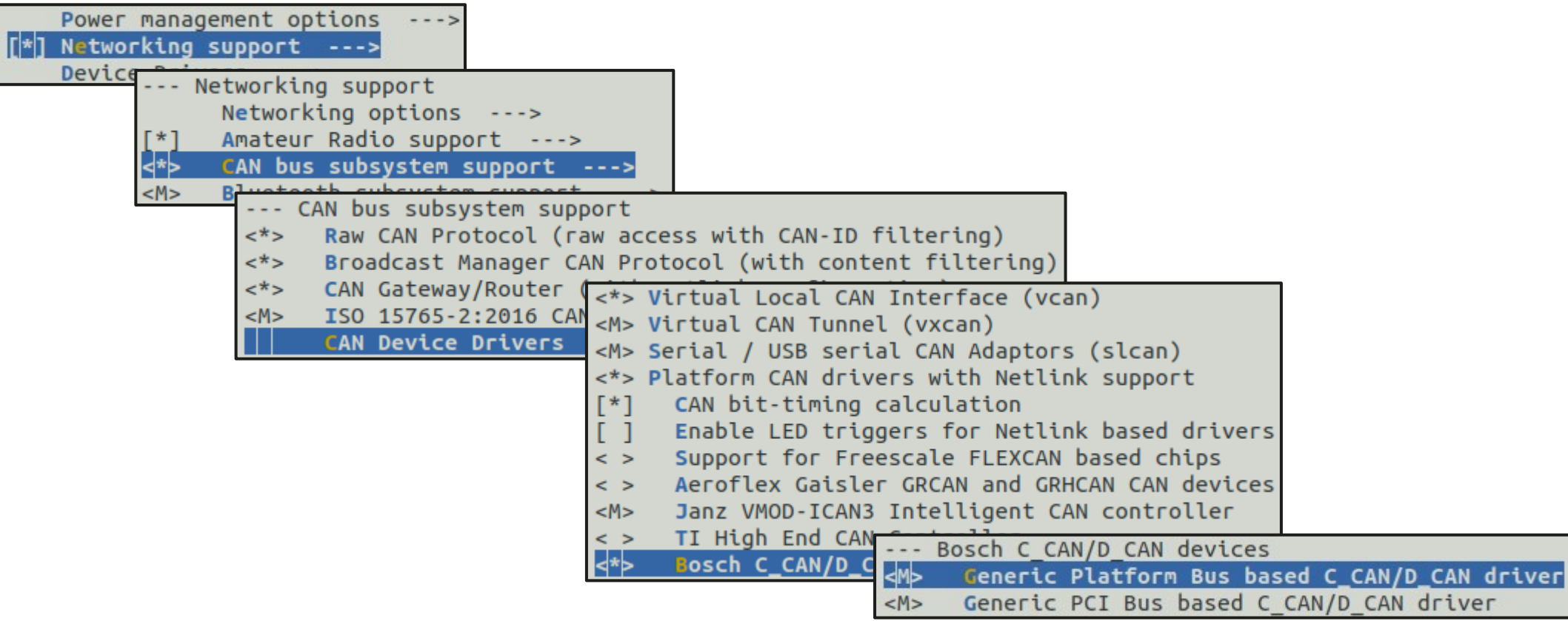
Ayant connaissance des modules à intégrer au kernel, lançons l'édition du fichier de configuration `.config` avec l'interface menuconfig.

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



Exemple : service CAN

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



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.

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'` » .

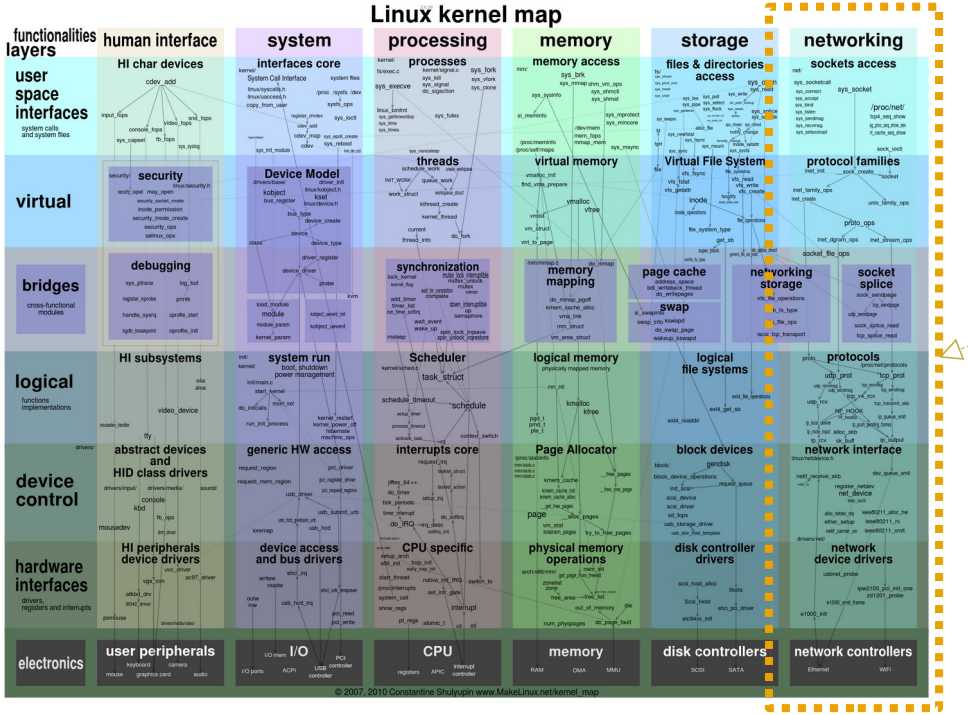
```
root@arm:~# dmesg | grep 'CAN\|can:'
[    0.116922] platform 481d0000.d_can: alias fck already exists
[    0.726646] vcan: Virtual CAN interface driver
[    0.726674] CAN device driver interface
[    0.727011] c_can_platform 481d0000.d_can: invalid resource
[    0.732965] c_can_platform 481d0000.d_can: 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 !

RAPPEL : Kernel map

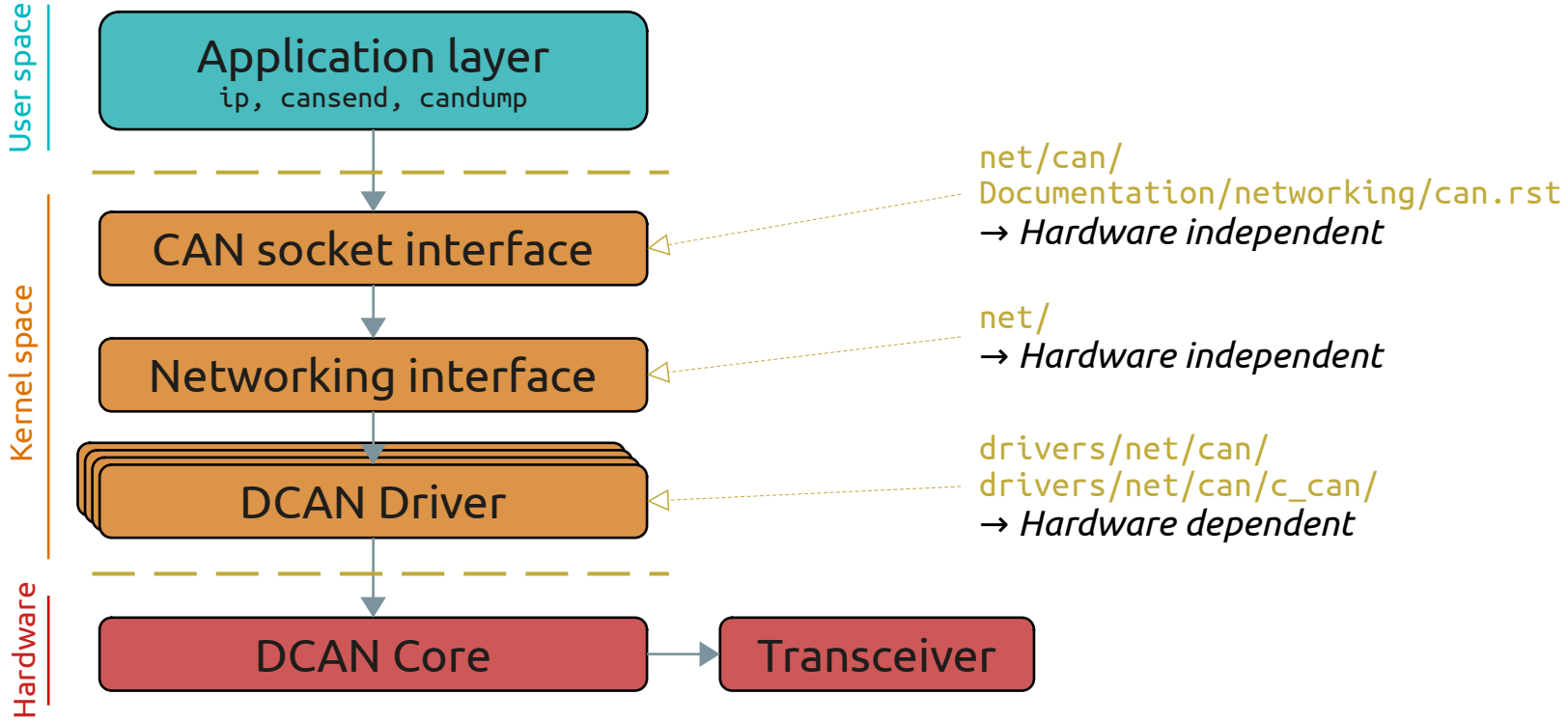
Afin d'assurer un bonne interopérabilité, maintenabilité et portabilité, le kernel Linux est découpé en couches proposant plusieurs niveaux d'abstraction, notamment au regard du matériel. Observons le *kernel map* du noyau :



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

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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

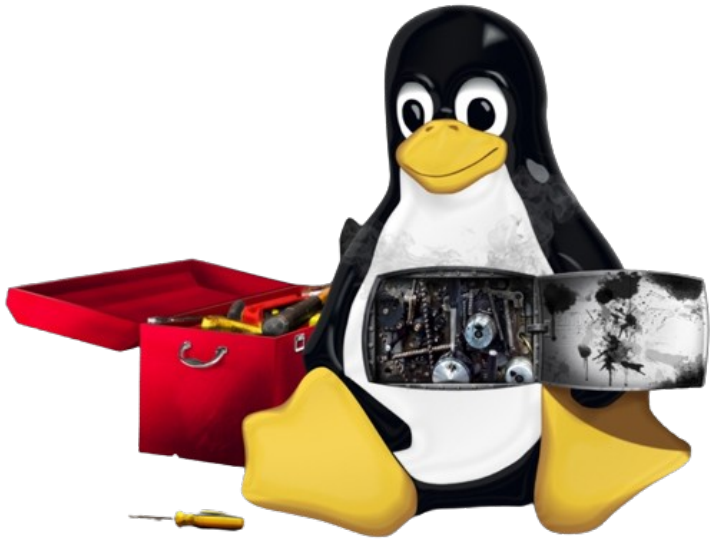
ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.



Board file

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 }
```

Board file

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/

Fichiers liés au Device Tree

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.

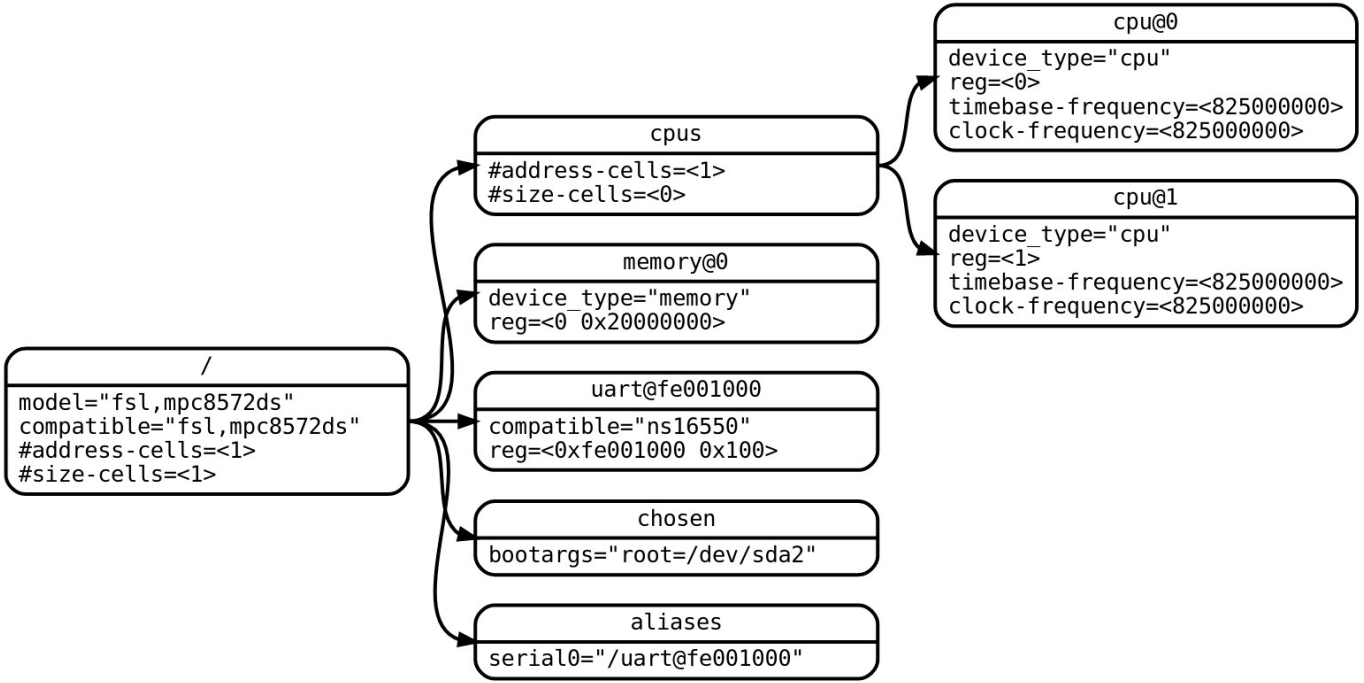
Le principal avantage est que la modification de la configuration matérielle n'impacte que le dtb (10-100 ko) et ne demande pas la recompilation complète du kernel.

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).

Construction d'un Device Tree

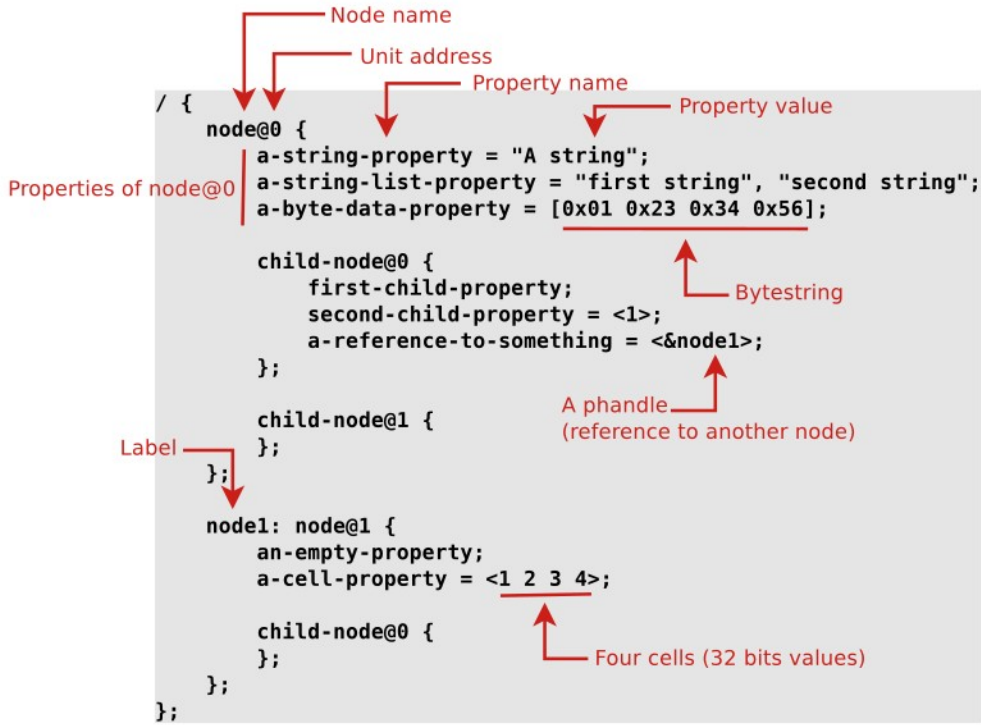
Comme son nom l'indique, le DT est une arborescence.



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

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



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

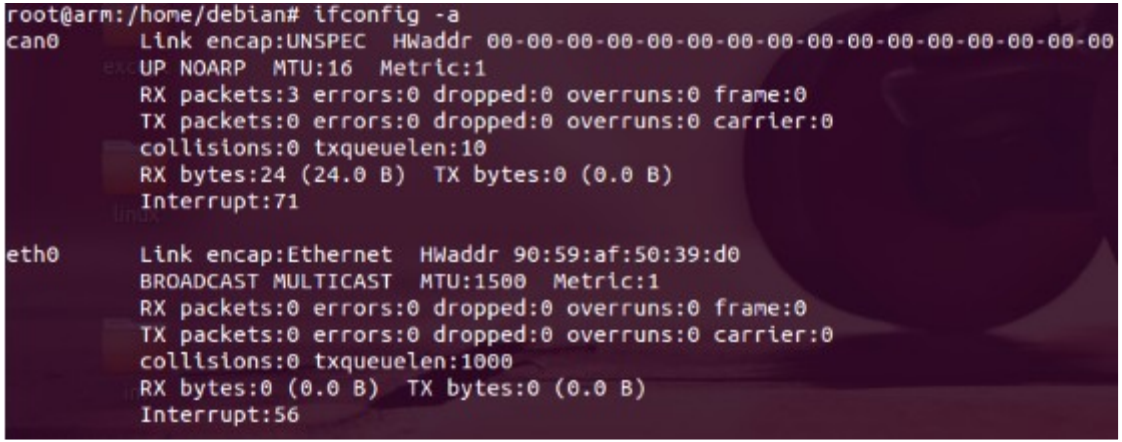
```
7  #include "am33xx.dtsi"
8  #include "am335x-bone-common.dtsi"
9  #include "am335x-boneblack-common.dtsi"
10 #include "am335x-boneblack-hdmi.dtsi"
11
12 / {
13         model = "TI AM335x BeagleBone Black";
14         compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
15 };
16
17 &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.

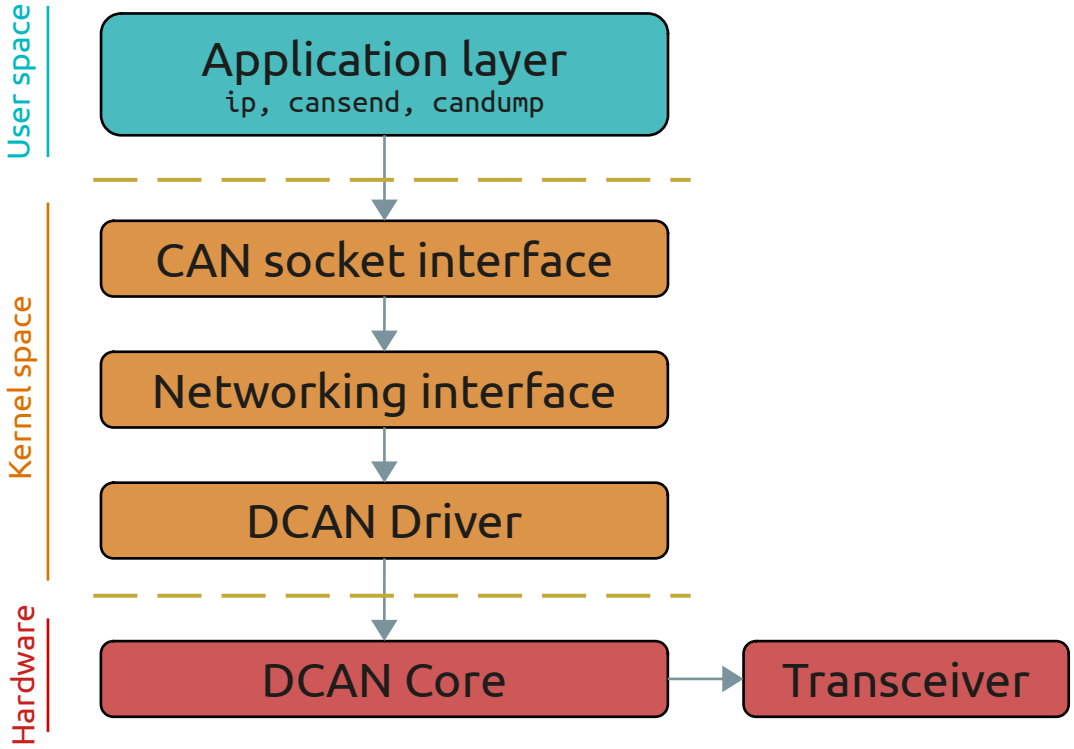
Il faut pour cela naviguer entre les différents fichiers du DT et la documentation du processeur. Tout cela sera vu en TP.

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



Structure du système d'exploitation

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

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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.

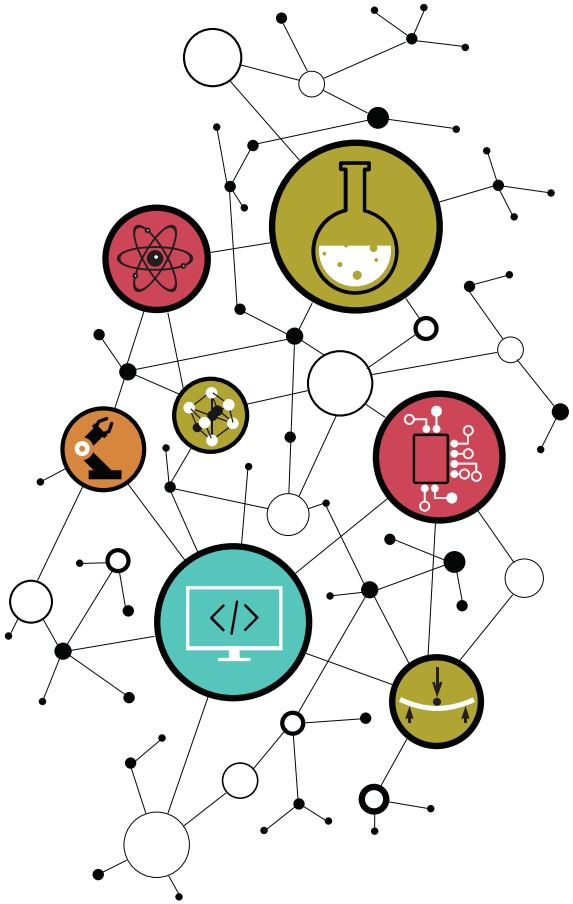
DÉPLOIEMENT SUR CIBLE

Étape 6

Trop archi-spécifique, sera vu en TP

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

CONTACT



Dimitri Boudier – PRAG ENSICAEN

dimitri.boudier@ensicaen.fr

Avec l'aide précieuse de :

- Hugo Descoubes (PRAG ENSICAEN)