REDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Portage de systèmes d'exploitation (POS)
Accès au matériel

Prof. Daniel Rossier

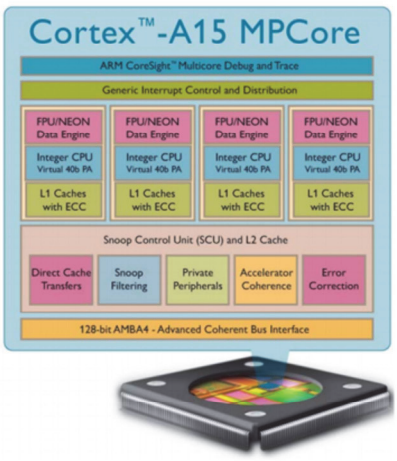
Version 1.4 (2017-2018)

Plan

- Architecture matérielle
- Plan d'adressage
- Introduction au *device tree*
- Propriétés de base des DT
- Propriétés avancées des DT
- Couche d'abstraction matérielle
- *Framework* (sous-systèmes) des *clocks, GPIOs, et pinctrl*

Architecture matérielle (1/6)

- ***CPU (coeur)***
- Jeu d'instructions
  - *armv4, armv5, armv6, armv7, armv8*
  - Adéquation de la *toolchain*
- Modes d'exécution
  - **Noyau**, hyperviseur, **utilisateur**, interruption, etc.
- Gestion mémoire
  - *Memory Management Unit*
  - Architecture des tables de page
  - *Memory Protection Unit*
- *Floating Point Unit (FPU)*

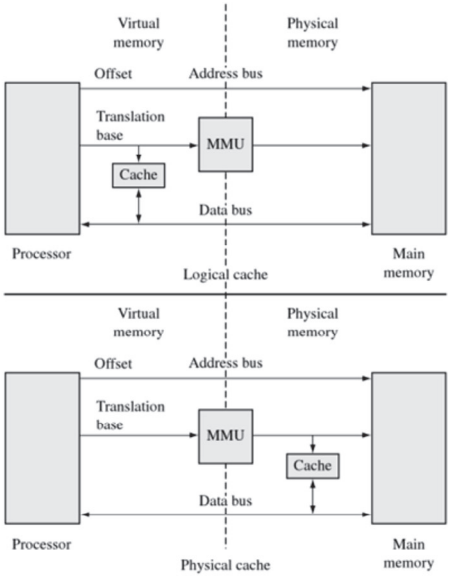


Le coeur du microcontrôleur (ou du *SoC*) définit le type de jeu d'instructions ainsi que l'architecture des mémoires *caches* et des coprocesseurs arithmétiques.

L'architecture de la *MMU* est également dépendant du type de microcontrôleur et définit les types de pagination, l'architecture des tables de pages ainsi que l'ensemble des bits d'attributs des entrées de table de page *(Page Table Entry*).

Architecture matérielle (2/6)

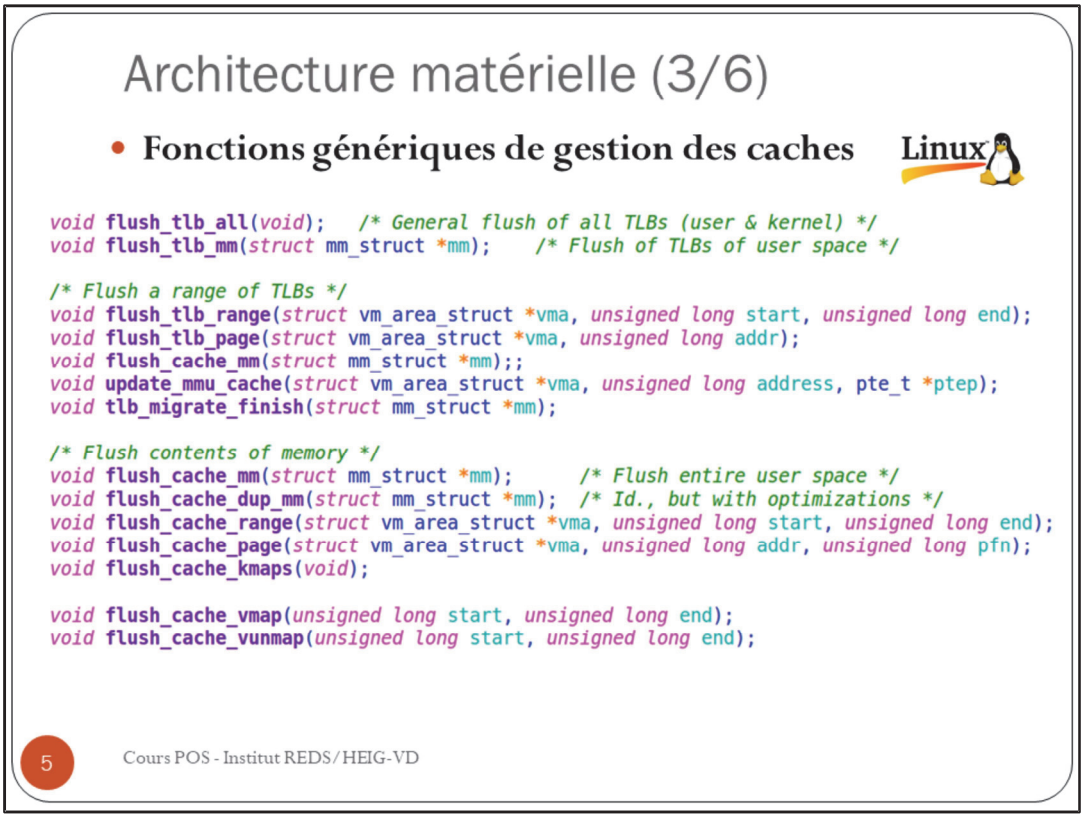
- ***Caches***
- Structure des *caches L1* et *L2*
  - Instructions (*I-cache*) & données *(D-cache)*
  - *Lookaside Translation Buffers (TLBs)*
- Fonctions génériques
  - *Flush* des *TLBs* et contenu mémoire
- Adresses virtuelles décomposées en trois éléments
  - ***Tag, index, offset***



La structure des *caches* définit les niveaux de *caches* (*L1 et L2*) et les politiques mises en œuvre dans la gestion de ceux-ci.

Le contenu des caches des différents niveaux ainsi que leurs architectures dépendent fortement du type de microcontrôleur. On y retrouve en tout cas les éléments suivants : instructions (*I-cache*), données (*D-cache*) ainsi que *TLBs* (*Translation Lookaside Buffer*). Ces derniers permettent de conserver l'association entre une page virtuelle et une page physique.

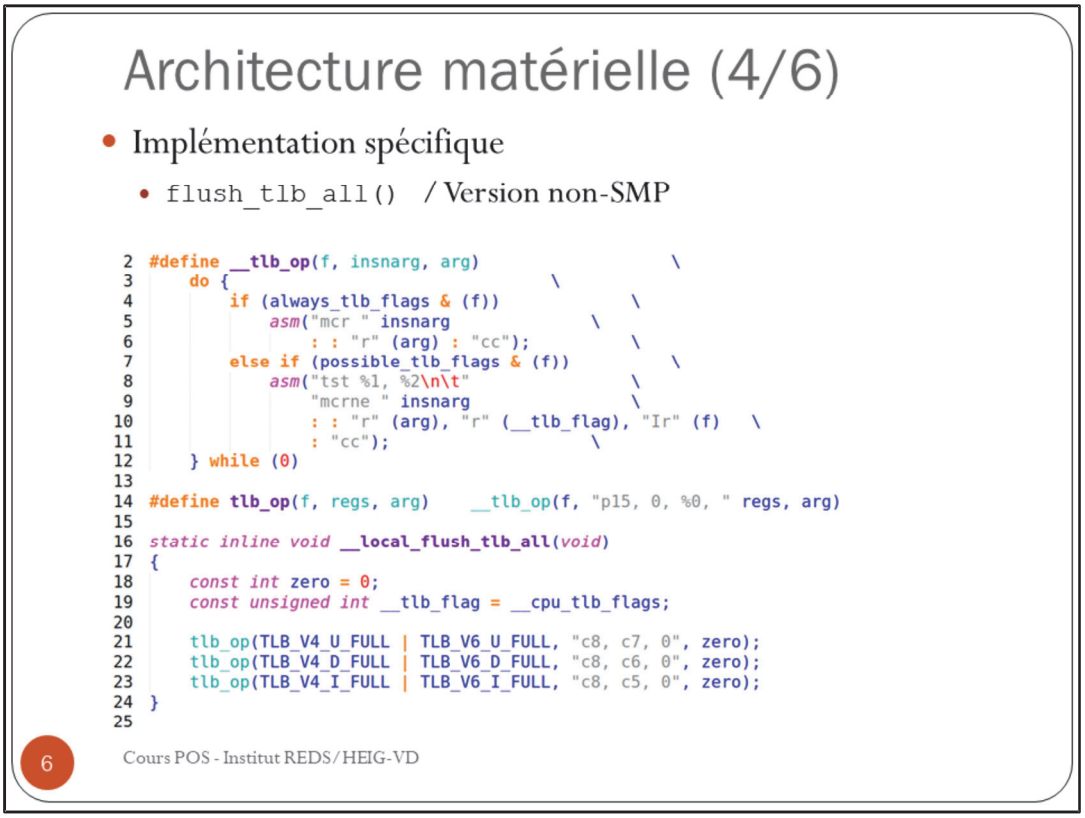
Les accès aux *caches* sont caractérisés par l'emplacement de ces mémoires, pouvant se trouver entre le processeur et la *MMU*, ou entre la *MMU* et la mémoire physique.



Les fonctions ci-dessus sont génériques pour l'ensemble des processeurs. Elles constituent ainsi une famille de fonctions pouvant être appliquées sur des régions mémoires de tailles différentes.

La première fonction (*flush_tlb_all()*) s'applique au cache mémoire de niveau 2 ou 3 selon l'architecture, et contient typiquement les associations entre numéros de pages virtuelles et numéros de pages physiques (cf chapitre "pagination" du cours SYE).
Cette fonction procède à une synchronisation complète du cache avec la RAM; cette opération est coûteuse et doit être "occasionnelle".

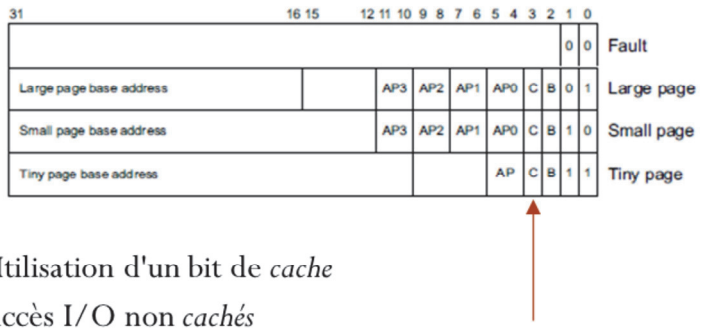
Dans le cas où des opérations de synchronisation sont fréquentes, il est nécessaire de *flusher* (synchroniser) que les entrées du caches concernées par les régions mémoires altérées. C'est pourquoi, la plus part des fonctions ci-dessus prennent comme arguments le début et la fin d'une région mémoire, ou alors un descripteur vers un certain contexte mémoire, typiquement lié à un processus, le processus courant par exemple.



Le code ci-dessus doit être exécuté rapidement (d'où l'intérêt des *pragmas #define* ainsi que les fonctions *inline*), et nécessite l'utilisation d'instructions spécifiques propres au processeur. Dans le contexte d'un processeur ARM, il s'agit d'une instruction ***mcr***, instruction très souvent utilisé lorsqu'il s'agit de manipuler registre de coprocesseurs (au sens large, les caches mémoires ayant des opérations internes de cette nature).

Architecture matérielle (5/6)

- Structure d'une PTE (second niveau)
  - Exemple ARM

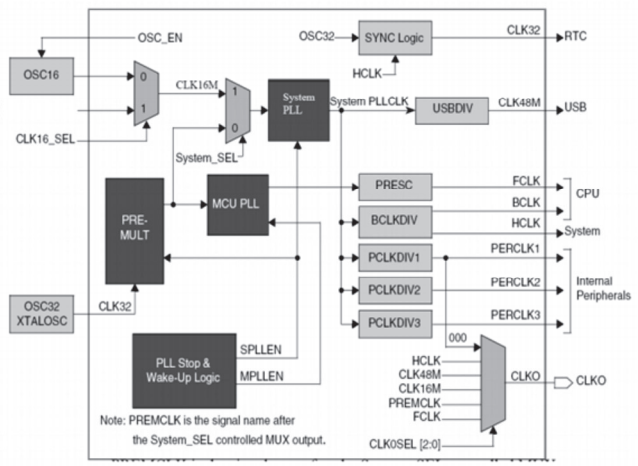


  - Utilisation d'un bit de *cache*
  - Accès I/O non *cachés*

La synchronisation des caches mémoires avec la RAM peut dépendre de la configuration des bits de la PTE (*Page Table Entry*) intervenant dans la traduction des adresses virtuelles vers les adresses physiques. Ces bits indiquent au processeur si les accès doivent être effectués obligatoirement dans la RAM (ou une région I/O) sans avoir recours aux caches. C'est le cas par exemple pour les registre de périphériques (I/O) où il est nécessaire d'avoir les valeurs courantes. Bien entendu, il s'en suit que les accès prendront un peu plus de temps.

Architecture matérielle (6/6)

- ***Horloges***
- Plus de 100 horloges dans un système embarqué
- Forte interdépendance
- Structure hiérarchique



8 Cours POS - Institut REDS/HEIG-VD

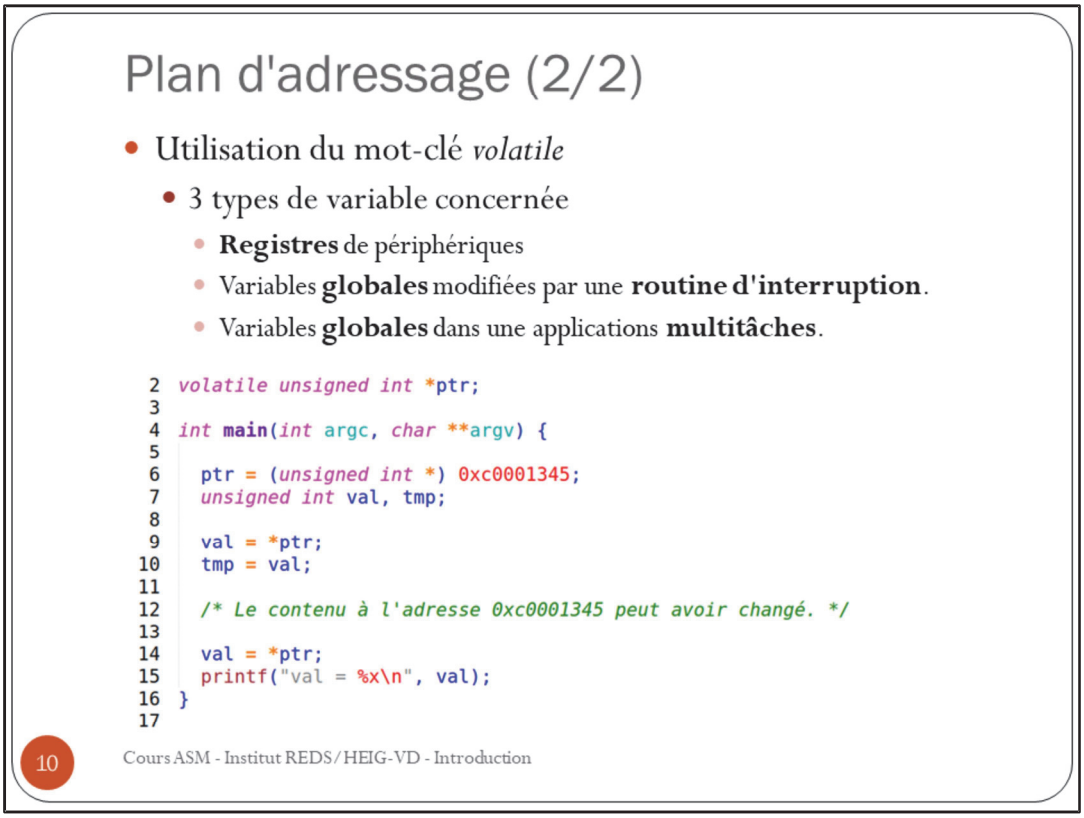
Les contrôleurs de périphériques nécessitent l'utilisation d'horloge (parfois plusieurs) afin qu'ils puissent fonctionner correctement; il en va de même d'un OS qui doit réguler l'ensemble de ses activités (processus, *threads*, alarmes, etc.) sur la base de *timers*.

C'est pourquoi, un système embarqué utilise beaucoup d'horloges de fréquences différentes avec des spécificités propres (tensions, polarités, etc.).

L'ensemble des horloges se présente sous la forme d'une architecture complexe, hiérarchique, où les fréquences sont dérivées à partir de différentes sources. La configuration des horloges est généralement fastidieuse et c'est pourquoi une abstraction matérielle à ce niveau est vivement souhaitable.

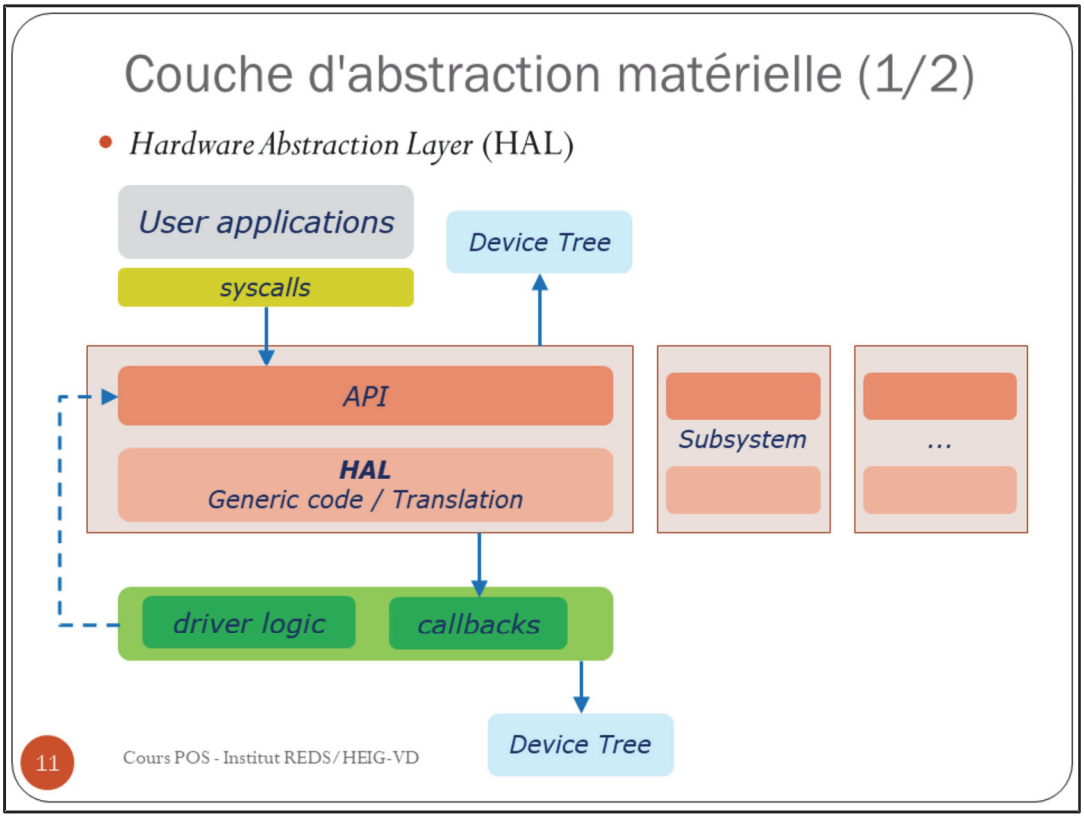
Plan d'adressage (1/2)

- Configuration dépendante du microcontrôleur
- Linéaire & *byte*-adressable
- Accès I/O
- Adressage virtuel / physique
  - Configuration de l'espace virtuel dépendant de l'OS
  - Présence d'une MMU pour les I/Os



Selon le niveau d'optimisation, le compilateur C essayera de minimiser les accès mémoires en supprimant les lignes de code *apparemment* sans effet, comme c'est le cas ci-dessus : sans le mot-clé *volatile*, l'affectation en ligne 14 serait simplement supprimée par le compilateur. Or, en admettant que l'adresse se réfère à un registre I/O d'un *timer* contenant le temps courant, ou à une variable partagée avec d'autres *threads*, les valeurs affichées seraient inexactes.

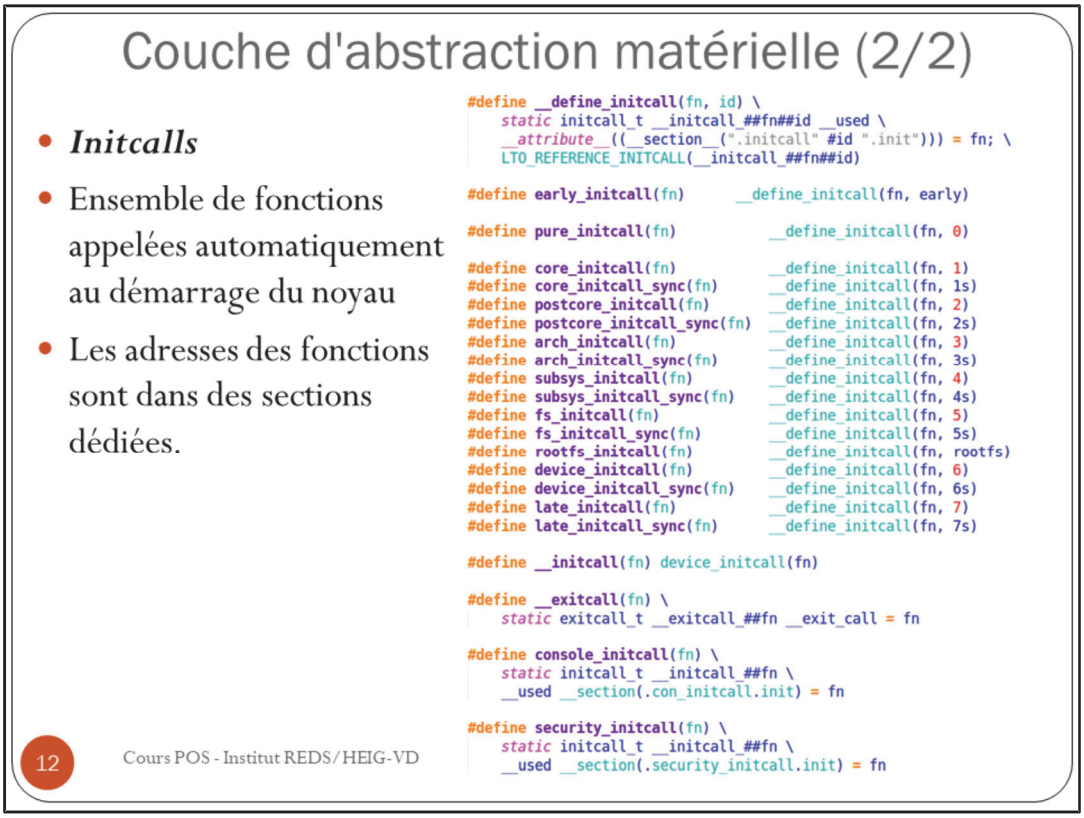
Par conséquent, les pointeurs associés à des adresses I/O doivent être déclarés avec *volatile* afin d'éviter des mauvaises surprises lors de l'exécution.



Un système d'exploitation ou un *bootloader* comporte normalement une partie générique, indépendante des périphériques ou de la plate-forme, afin de le rendre portable entre différentes architectures. Ce concept d'abstraction bien connu au niveau des applications (machine virtuelle *Java* par exemple, ou environnement graphique *Qt*) s'applique aussi dans le code noyau, comme pour les *drivers* par exemple. C'est la raison pour laquelle l'utilisation de fonctions de type *callback* est impérative, car elle permette le **découplage** entre la partie générique et la partie spécifique. Les couches d'abstractions apparaissant à ce niveau constitue ce que l'on appelle conventionnellement le ***HAL (Hardware Abstraction Layer)***.

Un HAL ne constitue pas seulement une API avec les couches supérieures mais implémentent également un comportement générique (par exemple, des fonctions d'interrogation de présence d'un périphérique d'un certain type sur un certain bus), ou encore des fonctions qui contrôlent l'allumage ou l'extension d'une LED (indépendamment du type de LED). Les principes de pagination font apparaître également des fonctions génériques (la mise à jour d'une PTE avec des attributs génériques indiquant si une page est mise en cache ou non, si elle a été modifiée, etc.). Le mappage entre les attributs génériques et les *vrais* attributs dépendant du type de MMU pourront se faire dans l'implémentation spécifique du HAL.

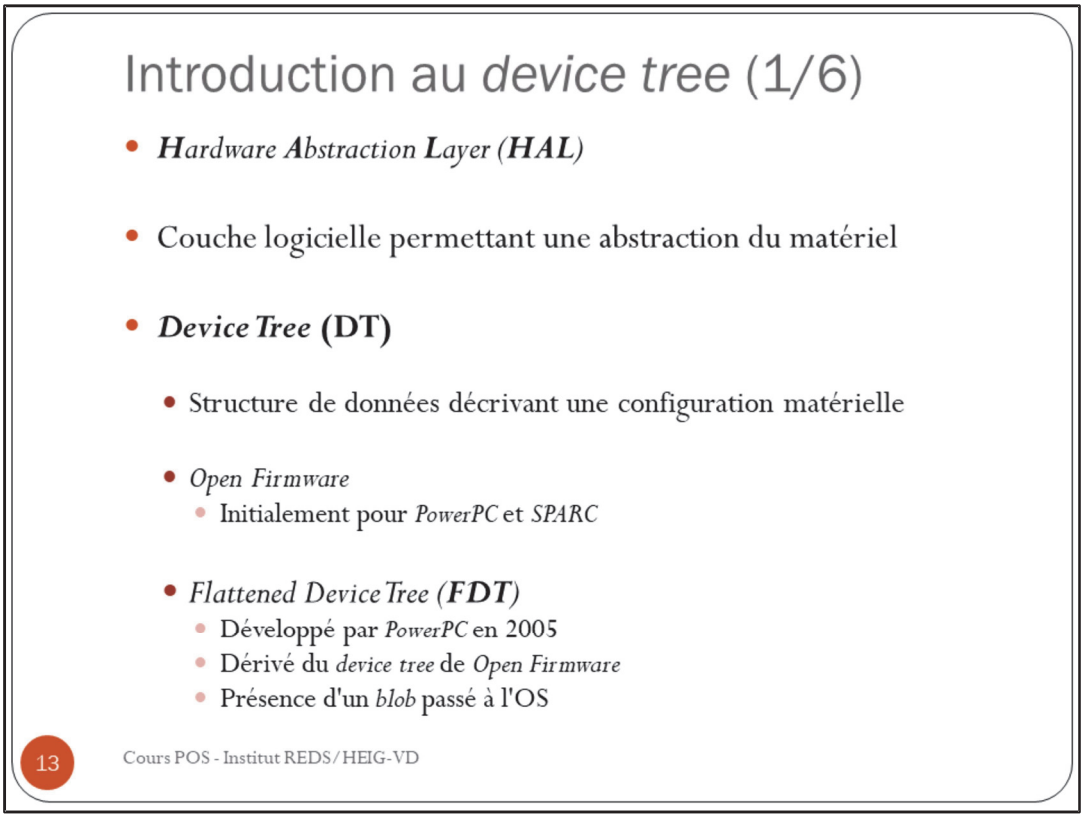
Pour faciliter l'implémentation d'un HAL, la description du système au sens large peut se faire à l'extérieur du noyau (donc hors du code) grâce à l'introduction du ***device tree***.



Dans l'esprit de *découpler* les dépendances entre le code spécifique à une configuration matérielle et les parties génériques, le mécanisme d'*initcall* joue un rôle important.

Le principe est basé sur une approche déclarative au niveau des fonctions d'initialisation (d'un driver, d'un sous-système, d'une *board*, etc.) dont leurs adresses sont collectées et placées dans des sections binaires dédiées. C'est donc le compilateur, au travers de l'utilisation de directives spéciales, qui générera les adresses des fonctions dans les sections spécifiques. Le *linker* arrangera ces sections au bon endroit en fonction du script de linkage.

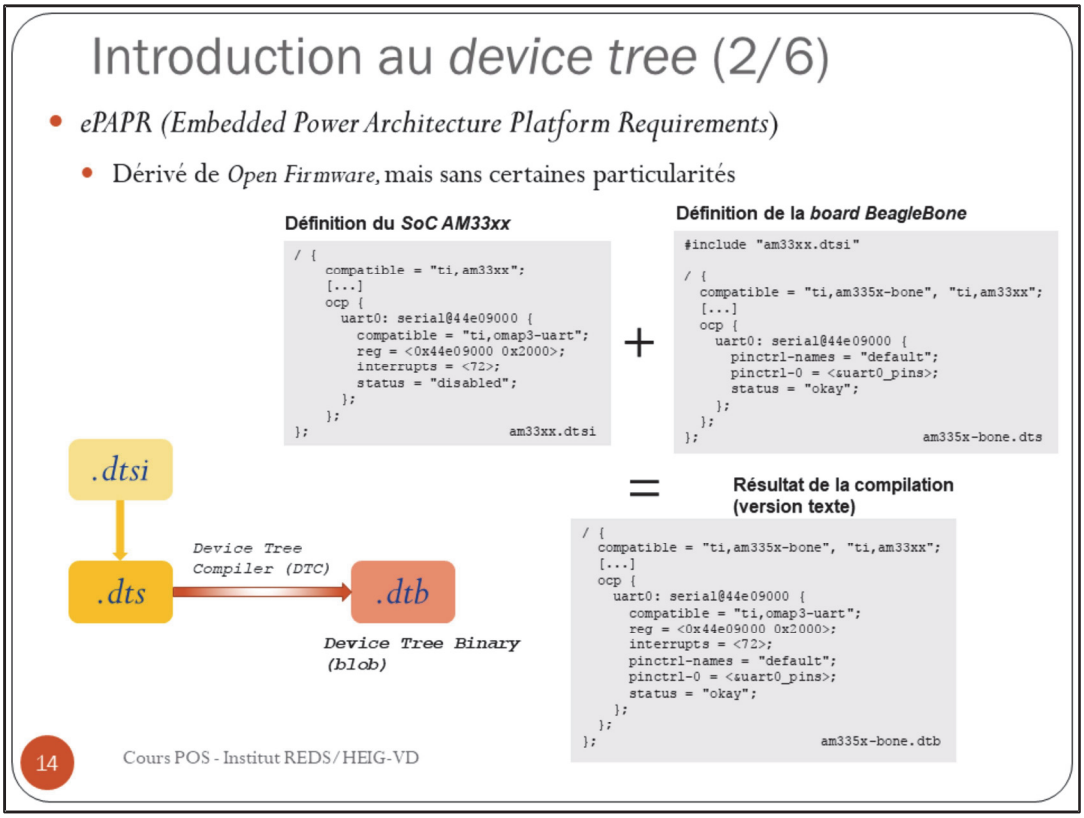
Le code d'initialisation pourra ainsi parcourir l'ensemble des adresses de fonction qu'il trouvera dans les sections réservées à cet effet, dans un ordre prédéfini.



L'abstraction matérielle permet de faciliter grandement le portage de composants de bas niveau d'une plate-forme à l'autre. Cela nécessite une couche logicielle qui doit transformer les détails matériels en descriptions plus ou moins génériques – donc portables – de telle manière à ce que les composants d'un noyau d'OS, par exemple, puisse accéder facilement au matériel. Cette couche logicielle porte généralement le nom de ***Hardware Abstraction Layer (HAL)***. Le *HAL* gère le nombre et le type de *CPU*, les adresses de base, les tailles des régions mémoires, les bus et, les lignes d'interruption, etc.

Dans ce contexte, le matériel consiste en une large palette de composants : les dispositifs reliés au *SoC*, les bus (*SPI, I^2C, i^2S, SDIO,* etc.), les contrôleurs d'interruption, les contrôleurs *GPIO*, les régulateurs de consommation, etc.

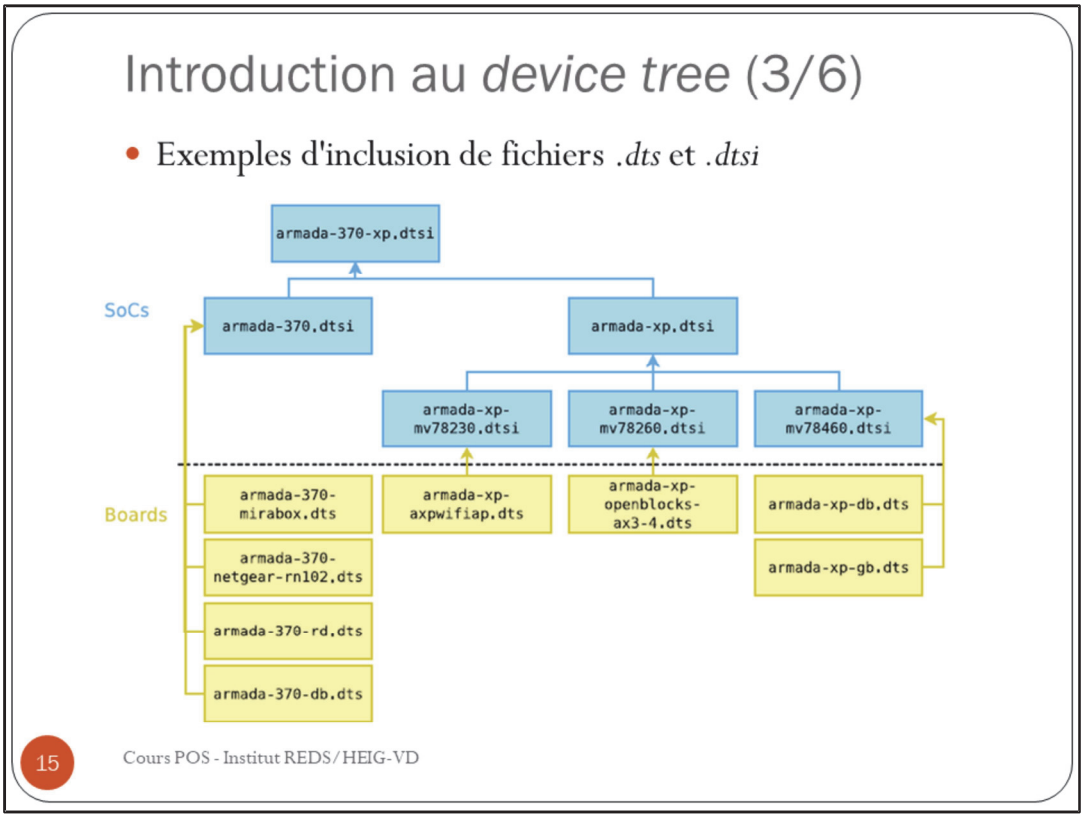
Le standard *Open Firmware (IEEE 1275)* est utilisé pour décrire les caractéristiques d'une plate-forme et ses périphérique sous forme d'un *device tree.* Le *device tree* est une structure de données se présentant sous forme d'un arbre qui peut être transmis du *bootloader* (initialement de type *Open Firmware*) au système d'exploitation. Le standard *IEEE 1275* a dérivé sur différentes variantes de *device tree*, dont notamment **le *Flattened Device Tree (FDT)***. C'est cette forme de *device tree* qui est la plus utilisée aujourd'hui dans *Linux*.



Dans le contexte des systèmes embarqués, le standard *ePAPR* définit le format *DTS* pour un *FDT*. Le *device tree* peut se présenter sous plusieurs fichiers *DTS*.

Un *DTS* peut inclure d'autres *DTS* (de base) décrit dans un fichier dont l'extension est *.dtsi*. Puis, le compilateur (*dtc*) produit un *device tree* sous la forme d'une image binaire, aussi appelé *blob*. Le fichier résultant porte habituellement l'extension *.dtb*.

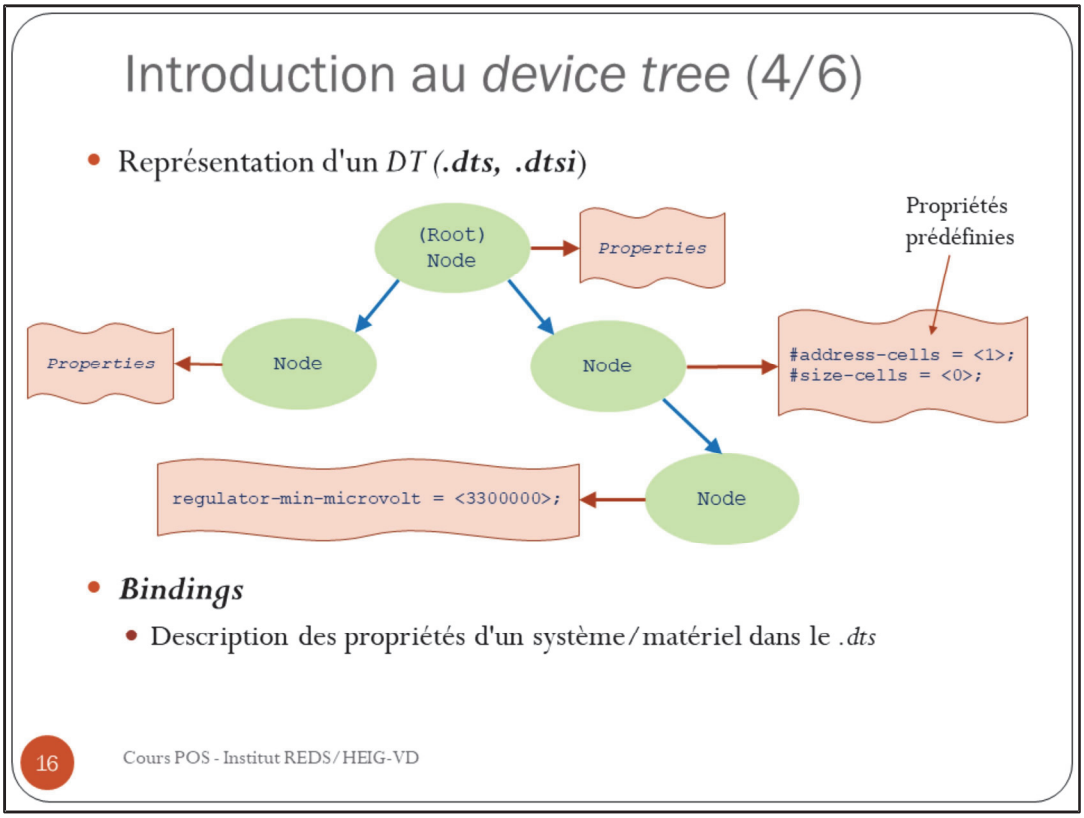
Puis, le fichier binaire (*.dtb*) peut être chargé par le *bootloader* avant le démarrage du noyau (l'adresse étant transmise à celui-ci), mais il peut être également concaténé au noyau directement. La méthode d'implantation en mémoire du *device tree* et la façon dont le noyau le récupère dépend de la configuration du *bootloader* et du noyau.



L'exemple ci-dessus montre l'inclusion de *dts* décrivant respectivement le coeur *CPU* de la plate-forme, le *SoC* de la plate-forme et la plate-forme elle-même.

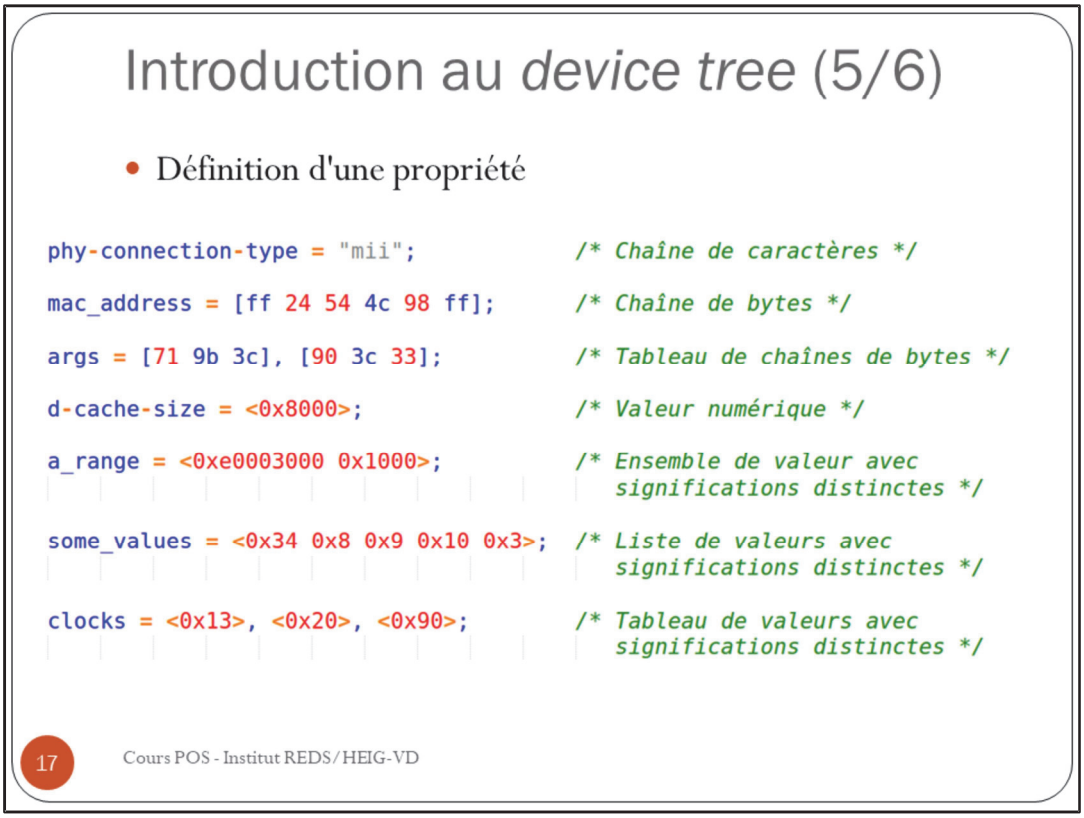
Les fichiers *.dtsi* sont destinés à être utilisés comme un *device tree* de base (générique) qui doivent être complétés par un *.dts*.

Comme dans la plupart des langages de haut niveau, une directive *#include* (ou aussi */include/*) existe et permet de spécifier un fichier à être inclus **avant** la compilation. Le **préprocesseur** est responsable de *parser* ces directives au même titre que les directives *#define*.



Un *device tree* se présente sous la forme d'une arborescence hiérarchique composée de nœuds et de propriétés.

La notion de ***bindings*** permet de décrire la liste des nœuds et propriétés spécifiques à un dispositif (processeur, plate-forme, périphérique, etc.). Les *bindings* consistent en un document (textuel) qui décrit ces conventions de description afin que le système d'exploitation puisse reconnaître (*matcher)* le dispositif avec son *device tree* correspondant, et de pouvoir ainsi récupérer toutes les données associées. Les *bindings* doivent être clairement définis et documentés.

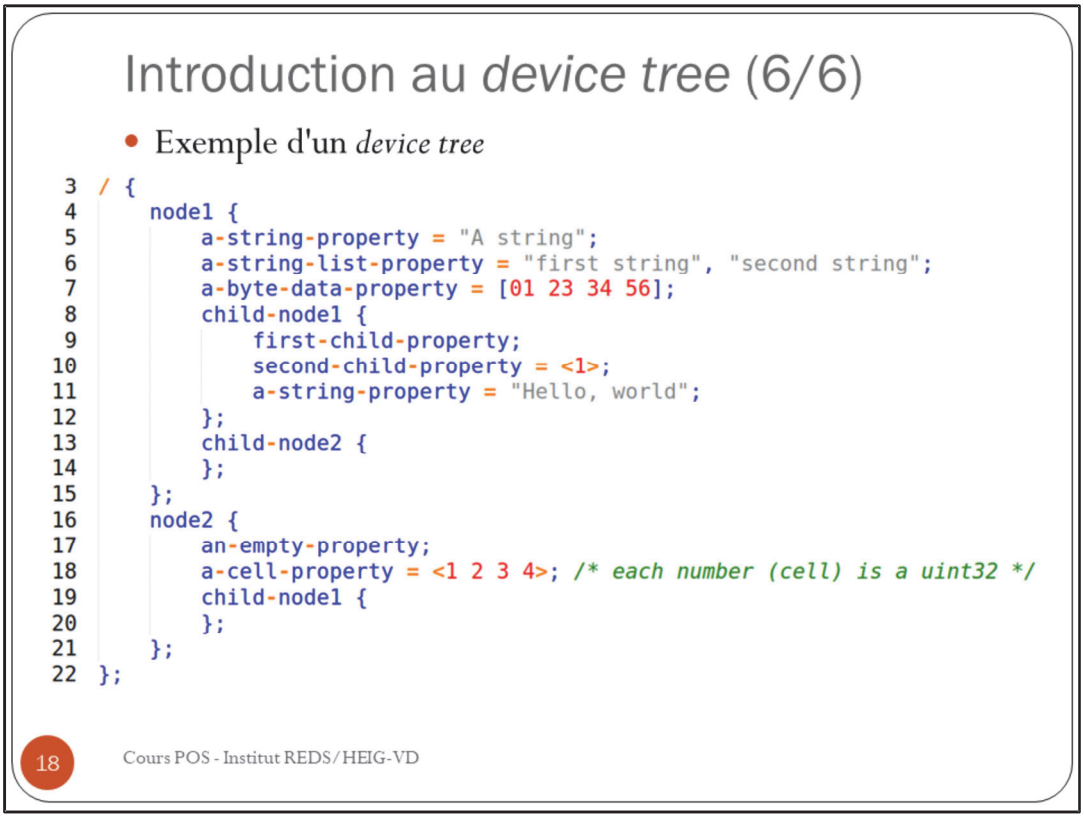


Les propriétés d'un DT peuvent être très variées et seront converties dans un type ad-hoc lors du *parsing* du DT. C'est le cas pour une chaîne de caractères, un entier, un tableau, etc.

Les propriétés doivent impérativement précéder la définition des noeuds.

Le nom d'une propriété peut contenir les caractères du tableau ci-dessous.

Character	Description
0-9	digit
a-z	lowercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash
?	question mark
#	hash



La description d'un *device tree* commence par le nœud racine. Les propriétés sont définies en premier suivies de la description des nœuds-fils.

Un *device tree* doit contenir au minimum un nœud racine.

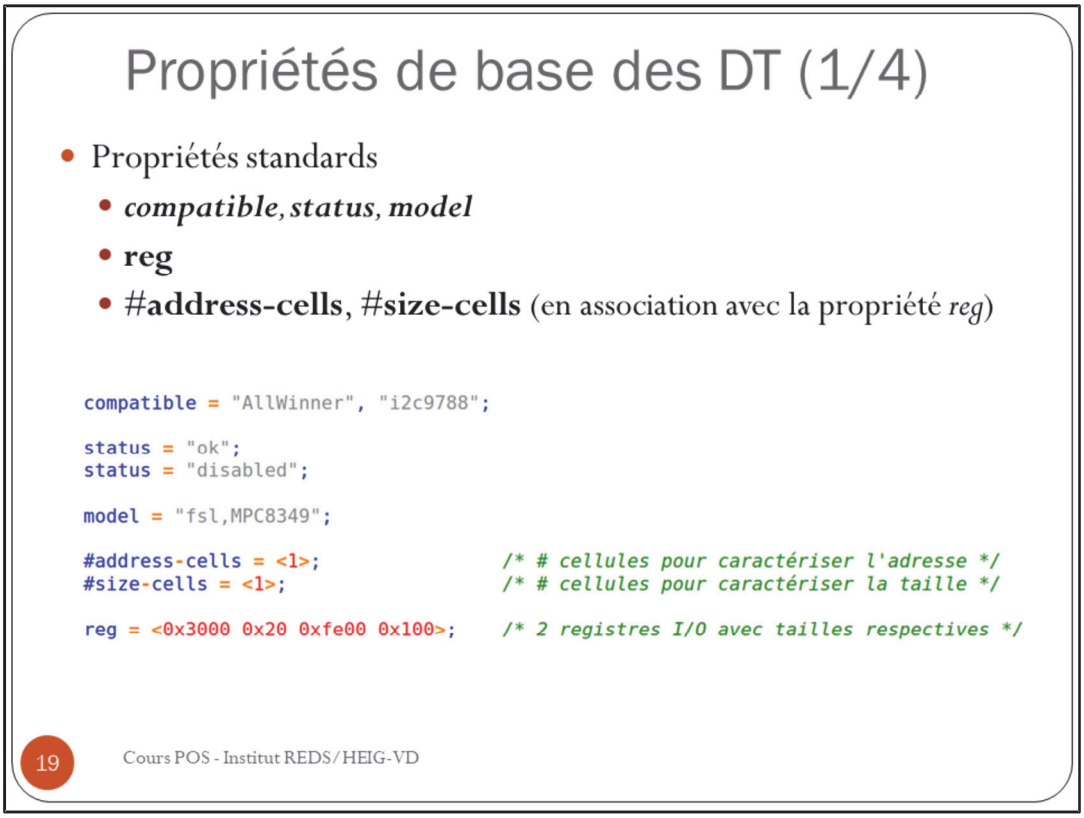
Le nom d'un nœud doit avoir la forme suivante : *<node_name>[@<unit_address>]*

Exemples : ***my-device, mem@0x13000, irq@3, i2c***

Un nœud est identifié de façon unique sous forme d'un **chemin** (***path***) absolu à partir du nœud racine (*/node_name_1/node_name_2/.../node_name_N*

Un noeud peut contenir les caractères du tableau ci-dessous.

Character	Description
0-9	digit
a-z	lowercase letter
A-Z	uppercase letter
,	comma
.	period
_	underscore
+	plus sign
-	dash

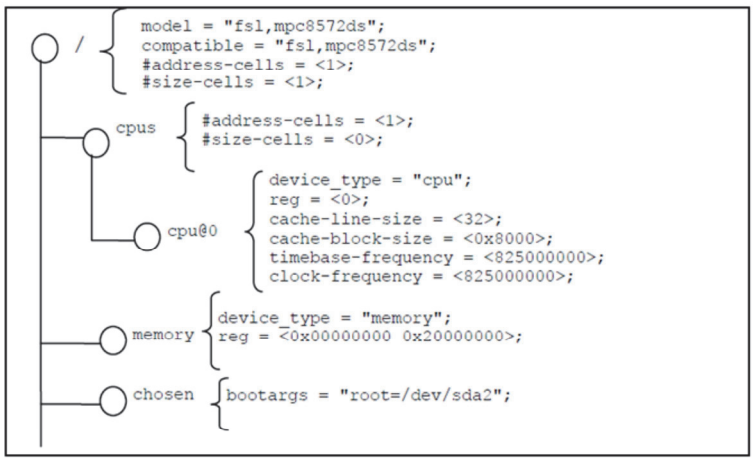


Les propriétés ***#address-cells*** et ***#size-cells*** concernent uniquement les propriétés *reg* définis dans les nœuds-fils uniquement, et non au-delà (niveaux inférieurs).

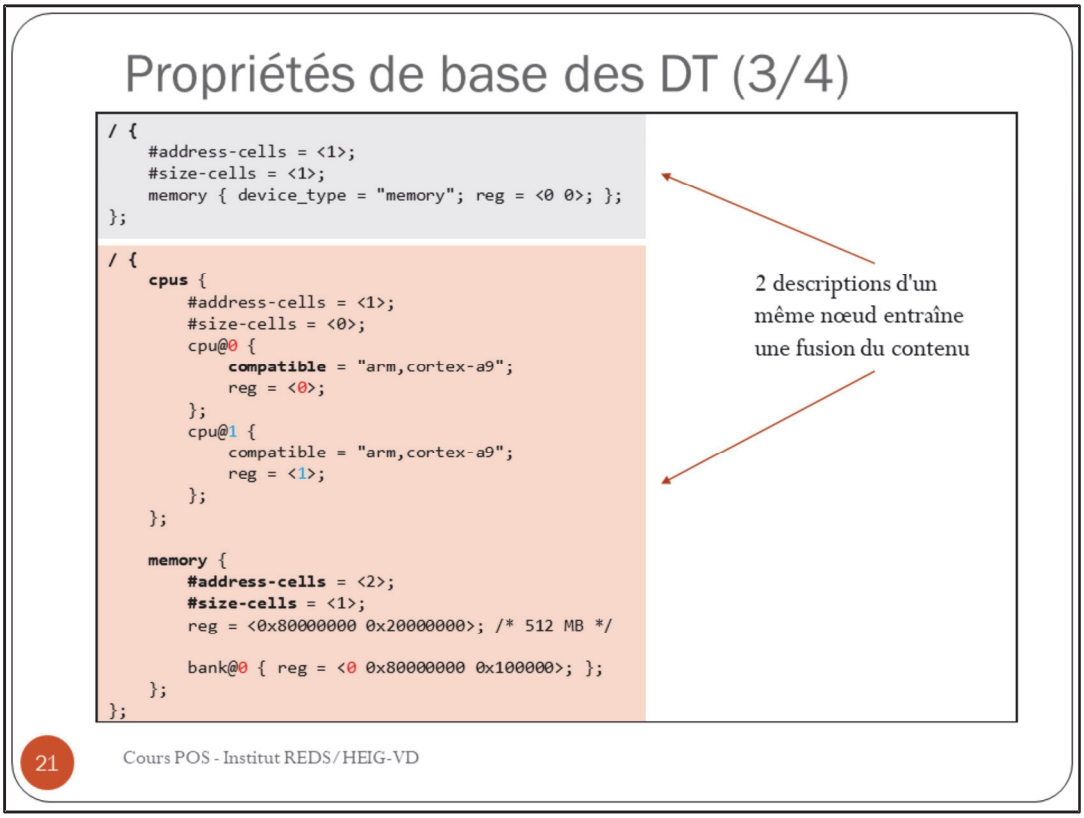
Ces propriétés sont décrites dans les pages suivantes.

Propriétés de base des DT (2/4)

- Exemple - *source: Power.org (ePAPR)*



La description ci-dessus montre l'utilisation des propriétés mentionnées précédemment.

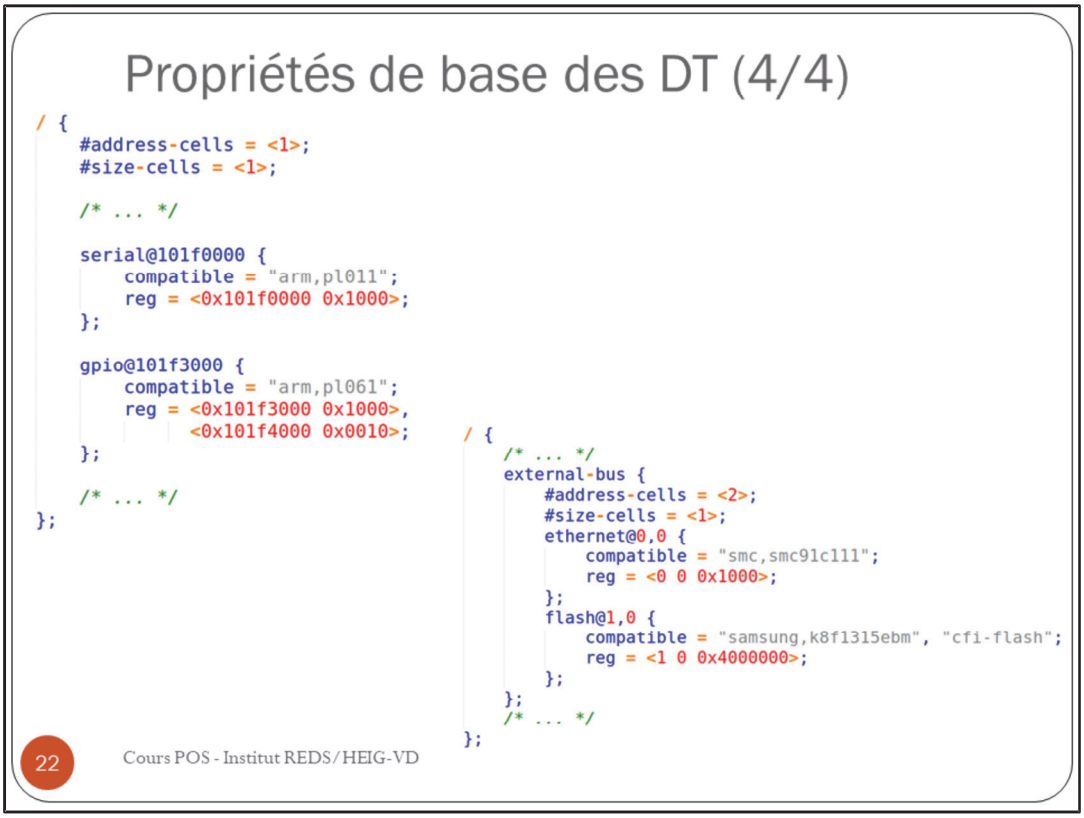


La description d'un *DTS* commence avec le nœud racine. Si plusieurs descriptions sont associées à un nœud identique, les descriptions sont fusionnées (superposées). On peut utiliser cette propriété dans un schéma d'inclusion de *DTS*.

La propriété *compatible* associée à un nœud détermine le *bindings* et permet à l'OS d'effectuer le *matching* entre un *driver* et le *device* par exemple, mais aussi d'associer les données du *DTS* avec la bonne famille de *CPU*, le bon *SoC*, la bonne plate-forme, etc.

Les propriétés ***#address-cells*** et ***#size-cells*** d'un nœud sont prédéfinies et renseignent sur le format de la propriété *reg* utilisée dans les nœuds-fils éventuels, et seulement à ce niveau; ces propriétés ne sont pas propagées au-delà des noeuds-fils. La propriété *reg* contient deux champs, un **champ *adresse*** et un **champ *taille*** spécifiant la taille de la région mémoire définie.

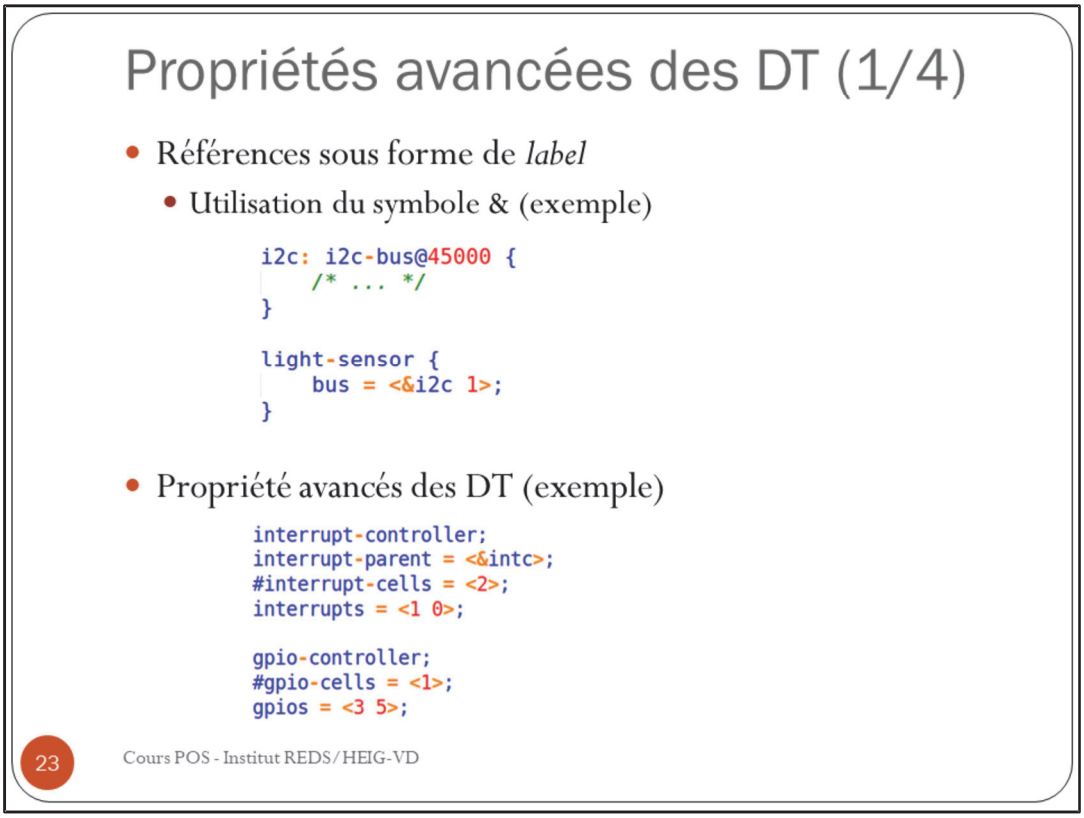
Dans le cas d'un nœud décrivant une zone mémoire, plusieurs façons de décrire cette zone sont possibles : la propriété *#address-cells* indique le nombre d'éléments (appelés *cellules*) exprimés sous forme d'entiers 32 bits nécessaires à la représentation du champ *adresse*, alors que la propriété *#size-cells* indique le nombre de cellules nécessaire pour représenter le champ *taille*. D'une manière générale, la propriété *reg* peut être utilisée pour tout type de valeur.



Une propriété peut être décrite sous forme d'un tableau. Ci-dessus, la propriété *reg* contient un ensemble de plusieurs adresses dont les caractéristiques liées aux cellules sont définies par les propriétés *#address-cells* et *#size-cells* dans le nœud-parent, comme discuté précédemment.

On remarque la présence du symbole @ dans des noms de nœud : il permet de construire un nom de nœud en deux parties : une partie *générique* et une partie *spécifique.* La partie spécifique est généralement un nombre représentant une adresse ou un identifiant. Ce nombre doit normalement se retrouver dans la première cellule de la propriété *reg*.

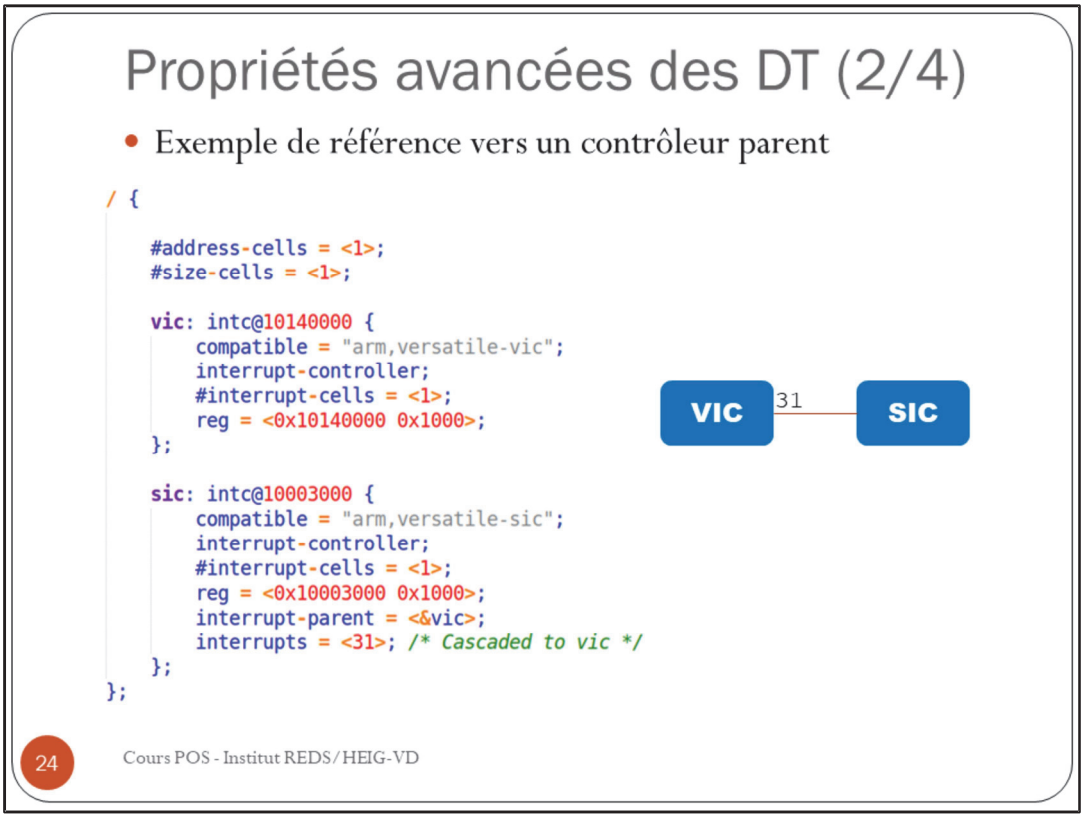
L'exemple de droite montre la possibilité de donner plus qu'un nombre (ou adresse) associé à un nœud. Dans ce cas, ce sont les premières cellules de la propriété *reg qui* devront correspondre à ces nombres, comme le montre l'exemple avec le nœud *flash*.



Un DT peut contenir des références vers des *labels.* Un *label* est rattaché à un noeud (il le précède avec ":". Une fois défini, le *label* peut être référencé à l'aide du symbole "&".

Une référence peut porter sur n'importe quel *label* quelque soit sa position dans l'arborescence, en avant ou en arrière. Etant donné que la définition des propriétés doit se faire avant la déclaration d'un noeud, il se peut que la propriété utilise des références vers des *labels* déclarés plus loin dans le *device tree*.

Parmi les propriétés avancées, on peut mentionner la propriété ***interrupt-controller*** et les propriétés associées (***interrupt-cells, interrupts, interrupt-parent***). Ces propriétés très utiles permettent de représenter les liens entre périphériques et contrôleur(s) d'interruption, en indiquant les numéros d'interruption qui pourront être récupérés par le *driver*. La propriété *interrupt-cells* indique le nombre de cellules que doit contenir la propriété *interrupts.*

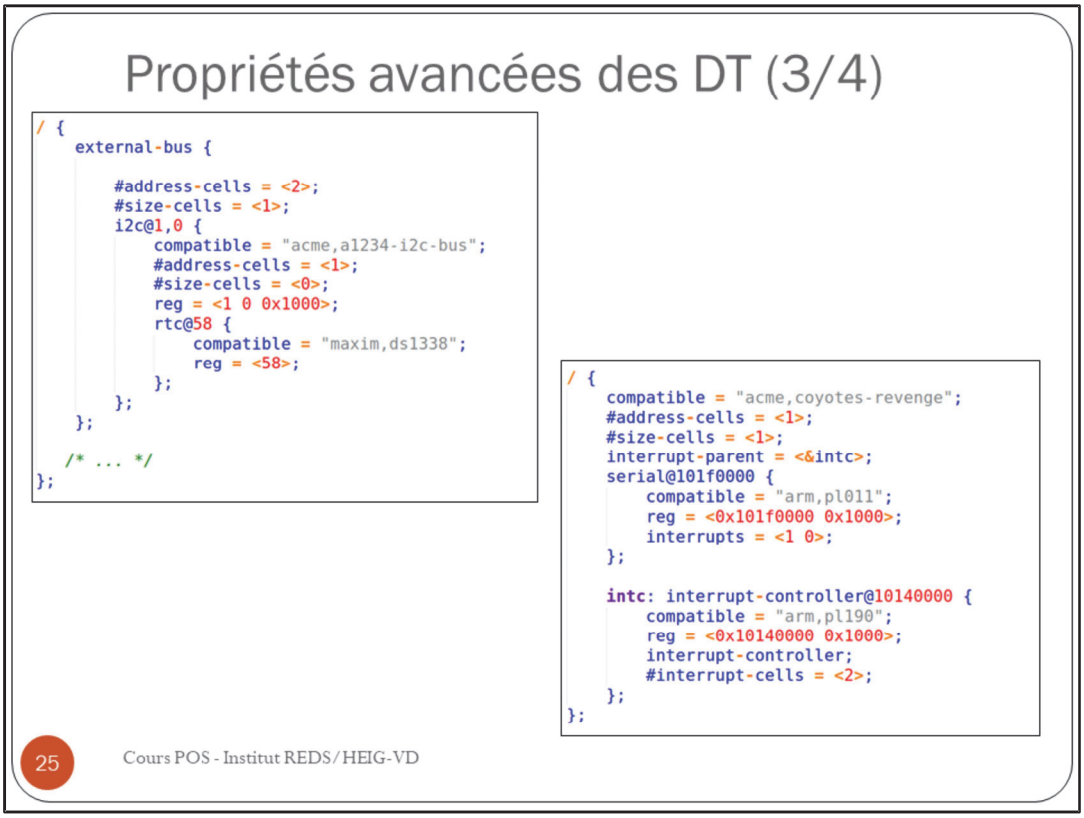


La propriété ***interrupt-controller*** stipule que le noeud associé dispose de la capacité de traiter une interruption (par exemple, un noeud associé à un contrôleur d'interruption ou un contrôleur GPIO puisque les lignes peuvent être configurées en mode *IRQ*).

Le noeud faisant référence à un tel noeud (capable de traiter l'interruption) doit utiliser la propriété ***interrupt-parent*** pour établir formellement le lien et indiquer quelle(s) ligne(s) d'interruption est(sont) sollicitée(s).

Il faut mentionner que le compilateur de DT n'effectue pas de vérification de cohérence entre ces différentes propriétés ou sur le nombre de cellules présentes dans la propriété ***interrupts***.

Ces propriétés sont encore détaillées sur la page suivante.



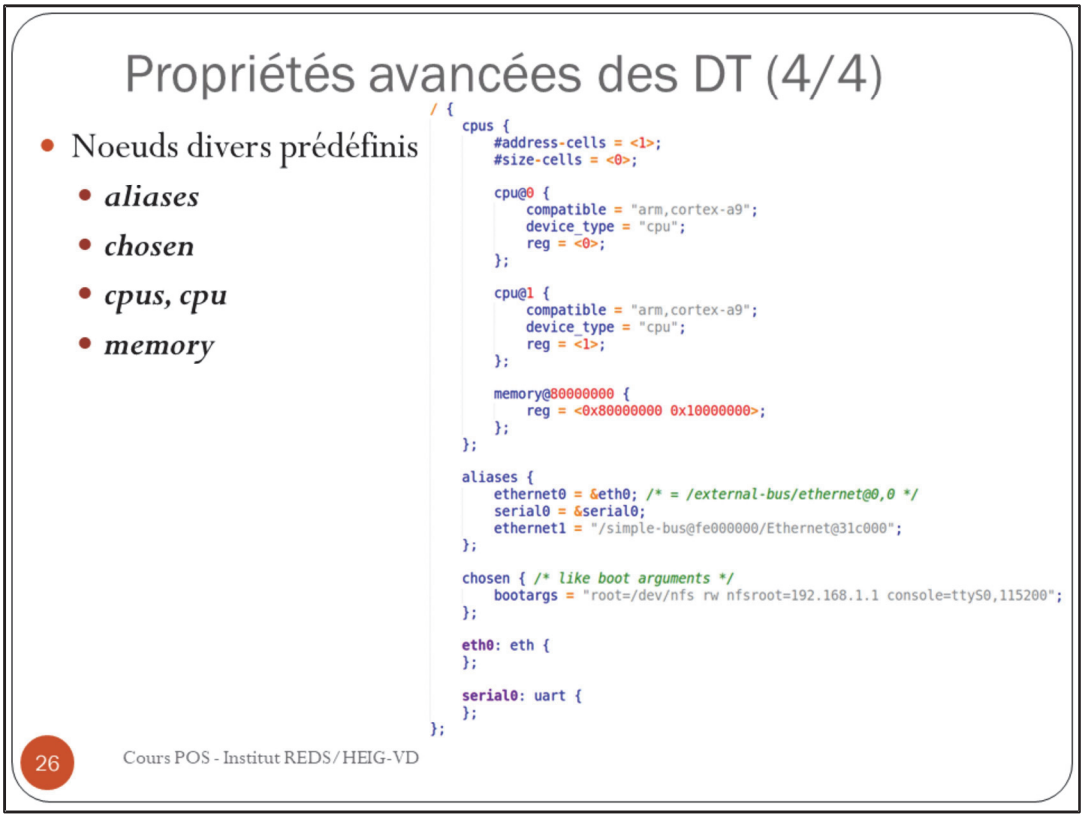
La propriété ***interrupt-controller*** est de type déclarative (sans autre argument); elle indique que le nœud courant représente un contrôleur d'interruption.

La propriété ***interrupt-parent*** permet de lier le nœud courant à un contrôleur d'interruption (la notion de *parent* se réfère au contrôleur et non au nœud). Cette propriété contient une référence vers un nœud de type contrôleur.

La propriété ***interrupt-cells*** spécifie le nombre de cellules que l'on pourra définir dans la propriété ***interrupts***, cette dernière permettant de décrire les caractéristiques liées aux lignes d'interruption gérées par le dispositif associé au nœud.

La propriété ***interrupts*** permet de définir les numéros d'interruption - les détails doivent être décrits dans les documents de *bindings* - qui sont gérés par le contrôleur d'interruption référencé dans la propriété ***interrupt-parent***.

Il faut noter qu'il n'y a pas de contraintes sur la position dans l'arborescence des noeuds intervenant dans ces relations.



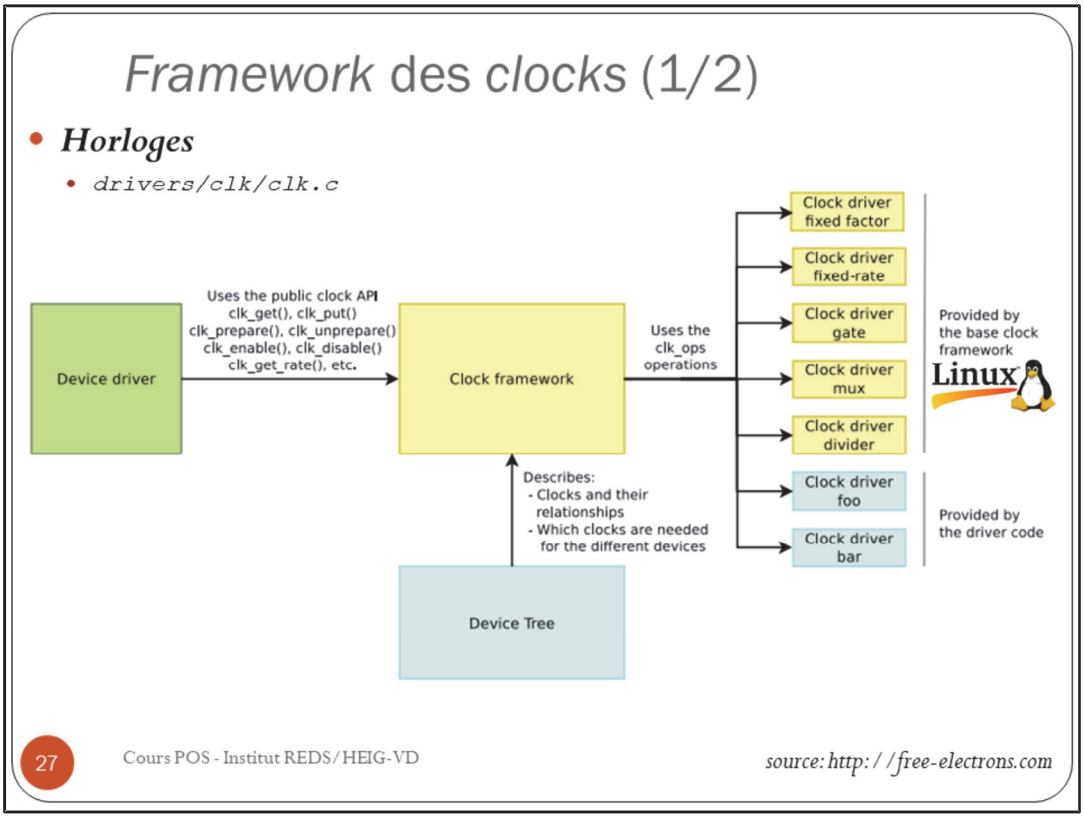
La propriété ***aliases*** permet de définir un nom (*alias*) associé à une référence vers un *label*.

La propriété ***chosen*** permet de sélectionner une configuration particulière lors du démarrage du système. Sous *Linux*, par exemple, cela correspond aux arguments introduits dans la "*command line*" récupéré par le noyau lors du *bootstrap.*

Cette propriété doit être un noeud-fils du noeud *root*.

Autre exemple :

```
{
   stdout_path =          → boot console
   stdin_path =           → input for console at boot time
   bootargs = ...
}
```



Linux possède un sous-système complet dédié à la gestion des horloges composé d'une partie commune (ou générique) et d'une partie dépendante du matériel.

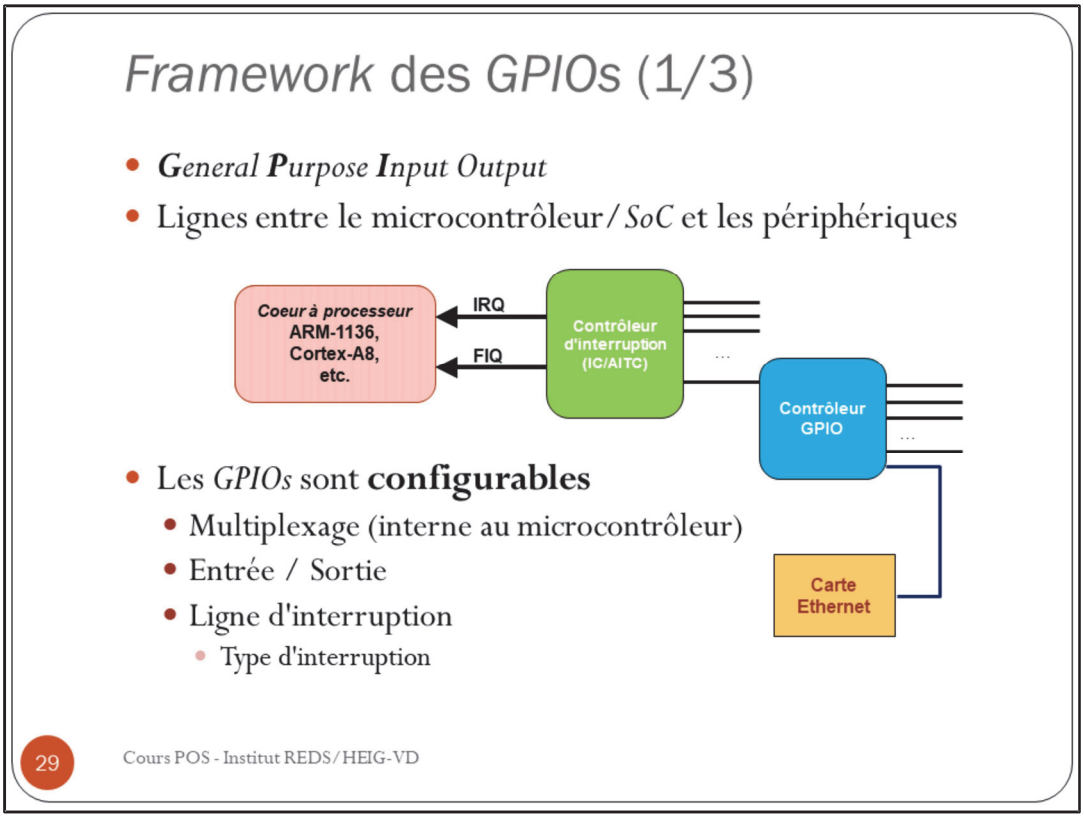
Le sous-système se présente sous la forme d'un *framework* appelé *CCF (Common Clock Framework*). Il contient des fonctions génériques comme *clk_get(), clk_get_rate(), clk_enable(),* etc.

Les fonctions dépendantes du matérielles sont appelées en fonction des propriétés découvertes dans le *device tree* lors du démarrage du noyau.

Framework des *clocks* (2/2)

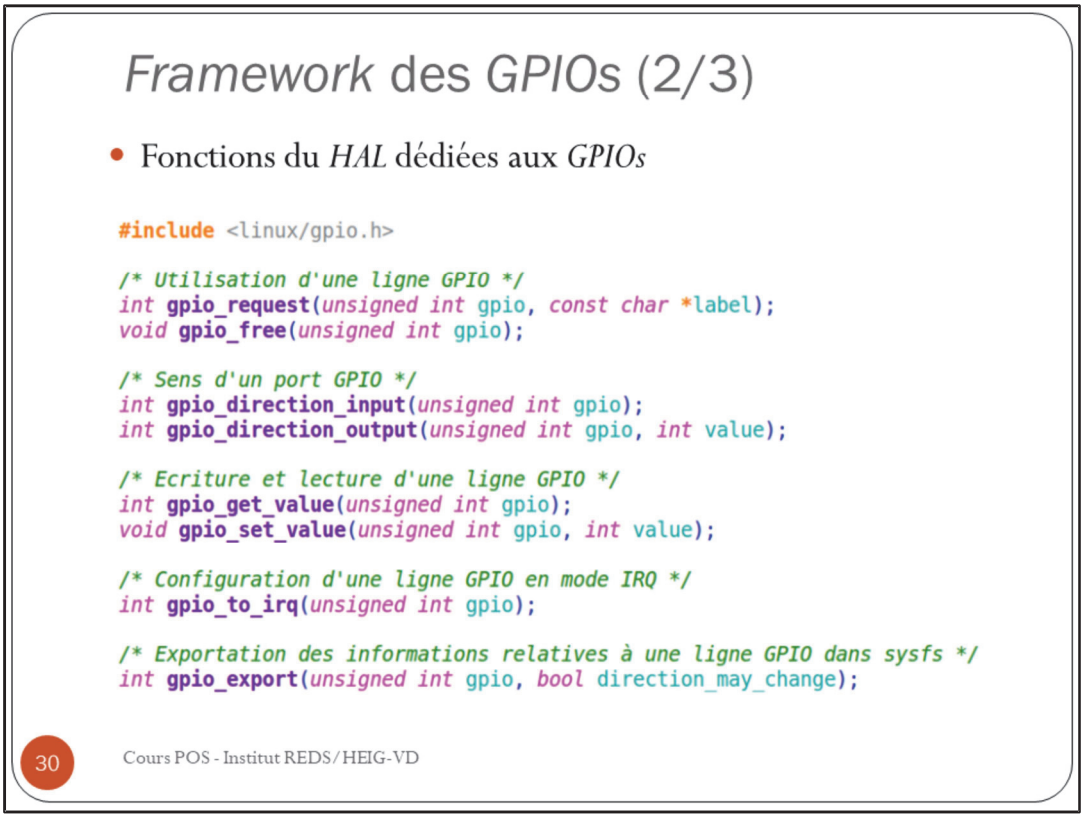
- Pour chaque matériel, les *callbacks* liés au sous-système *clock* doivent être implémentés.

```
struct clk_hw_ops {
  int             (*prepare)(struct clk *clk);
  void            (*unprepare)(struct clk *clk);
  int             (*enable)(struct clk *clk);
  void            (*disable)(struct clk *clk);
  unsigned long   (*recalc_rate)(struct clk *clk);
  long            (*round_rate)(struct clk *clk, unsigned long, unsigned long *);
  int             (*set_parent)(struct clk *clk, struct clk *);
  struct clk     *(*get_parent)(struct clk *clk);
  int             (*set_rate)(struct clk *clk, unsigned long);
};
```



Le contrôleur *GPIO* est un composant essentiel et présent dans tous les microcontrôleurs. Les *GPIOs* permettent d'interconnecter le microcontrôleur (ou *SoC*) avec des périphériques internes ou externes. Plusieurs dispositifs peuvent être reliés sur une même *GPIO*; dans ce cas, il est nécessaire de spécifier lequel est utilisé à un moment ou à un autre (il y a en principe une configuration par défaut).

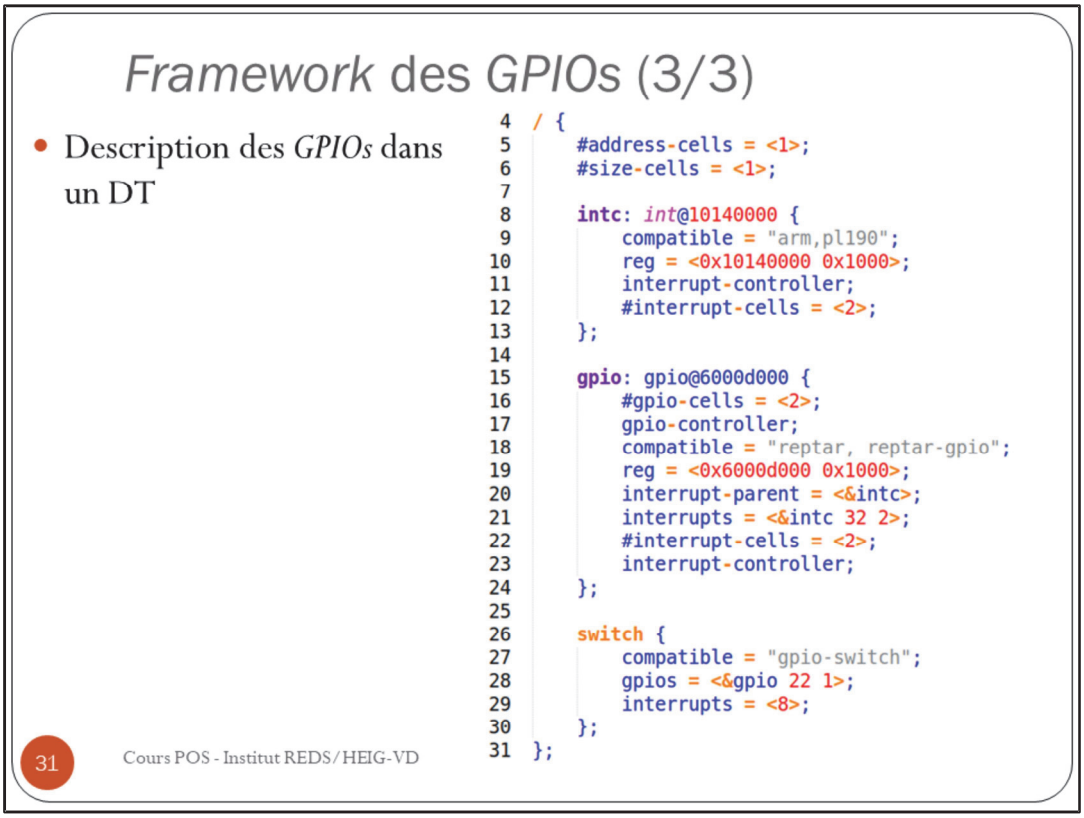
Dans le cas où une *GPIO* est associée à une ligne externe, la *GPIO* peut être configurée en entrée ou en sortie, ou encore en mode *IRQ*. Dans ce dernier cas, la *GPIO* pourra solliciter une interruption (le contrôleur *GPIO* est relié au contrôleur d'interruption via - au moins - une ligne dédiée).



Le *HAL* peut contrôler les *GPIOs* avec un ensemble de fonctions génériques telles que décrites ci-dessus. Les composants du noyau (*drivers*, sous-système d'initialisation, sous-système de type *"plug & play*, etc.) peuvent donc sans autre utiliser directement ces fonctions.

Dans le cas de *Linux*, par exemple, la dernière fonction – *gpio_export()* – permet d'exporter les informations relatives à la *GPIO* dans le système de fichiers ***sysfs***. Grâce à ce mécanisme, on peut également configurer les *GPIOs* directement depuis l'espace utilisateur.

L'implémentation de ces fonctions appartient au *HAL*; elle sera ainsi dépendante du matériel, c-à-d du contrôleur *GPIO* du microcontrôleur cible. Il est à noter que la dépendance au matériel dans ce cas précis porte essentiellement sur les adresses des registres du contrôleur et la structure des registres. Ces informations peuvent sans autre être décrites dans *device tree*.



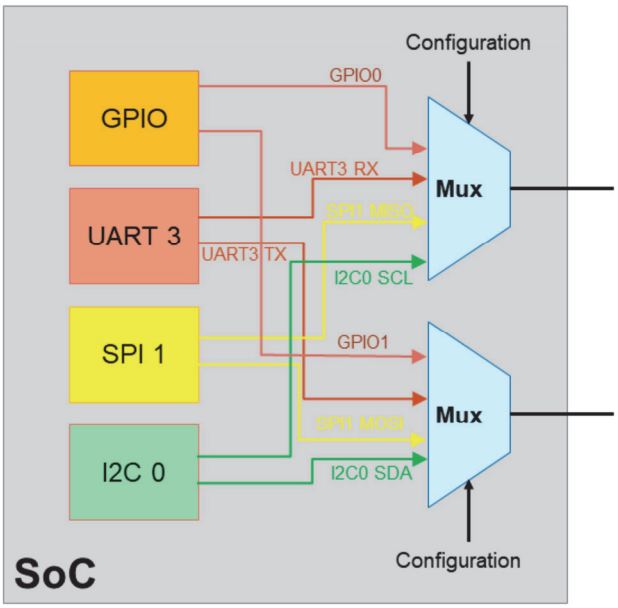
Les caractéristiques des *GPIOs* d'une plate-forme peuvent être sans autre décrites dans un *device tree*, comme le montre l'exemple ci-dessus.

A l'instar d'un contrôleur d'interruption, il existe des propriétés avancées pour un contrôleur *GPIO* comme ***<#gpio-cells>*** et ***<gpio-controller>*** par exemple. Ainsi, on peut également décrire une hiérarchie entre les deux types de contrôleur et caractériser chaque ligne *GPIO* du mieux que possible.

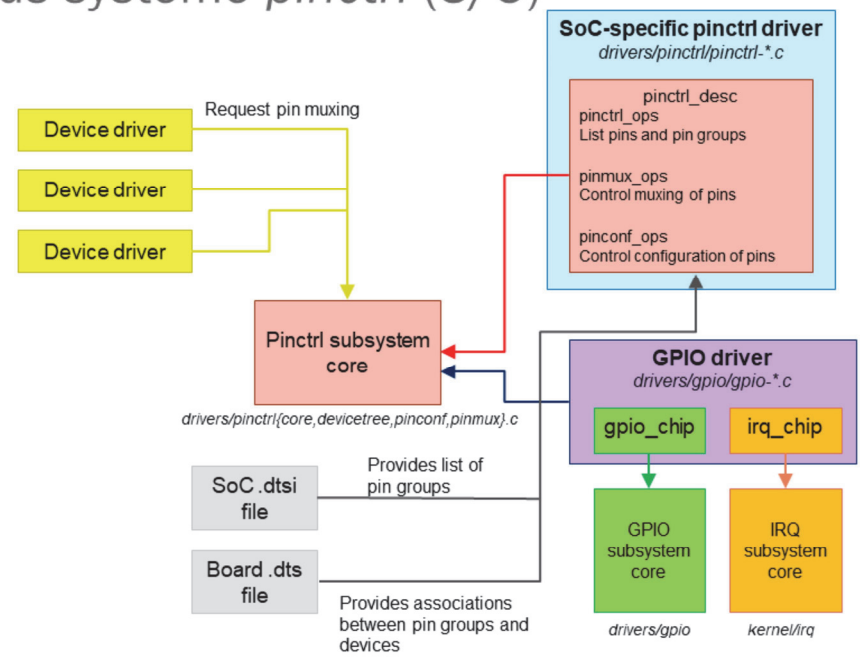
Sous-système *pinctrl* (1/5)

- Sous-système ***pinctrl***
  - Déclaration des lignes d'entrée-sortie (*pins*)
  - Association entre les lignes I/O et les fonctionnalités internes
- `drivers/pinctrl`
  - Gestion du multiplexage des *pins*
  - Interface pour les *drivers*
  - Interface vers l'implementation des fonctions du multiplexeur du *SoC*.

Sous-système *pinctrl* (2/5)



Sous-système *pinctrl* (3/5)



Sous-système *pinctrl* (4/5)

- Utilisation des lignes dans une configuration spécifique (*board level*)
  - Propriétés du *Device Tree*
    - *pinctrl-<x>*
    - *pinctrl-names*

- *pinctrl-<x>*
  - Référence à une description existante pour une fonction particulière
  - *pinctrl-0*, *pinctrl-1*, *pinctrl-<x>*

- *pinctrl-names*
  - Liste des noms associés aux différentes fonctions multiplexées
  - "*default*" signifie que le périphérique utilise la configuration de base

```
i2c@0x11000 {
  pinctrl-0 = <&pmx_twsi0>;
  pinctrl-names = "default";
  ...
};
```

Sous-système *pinctrl* (5/5)

SoC Level

arch/arm/boot/dts/sun7i-a20.dtsi

```
/ {
    soc@01c00000 {
      pio: pinctrl@01c20800 {
        compatible = "allwinner,sun7i-a20-pinctrl";
        reg = <0x01c20800 0x400>;
        interrupts = <0 28 1>;

        uart0_pins_a: uart0@0 {
          allwinner,pins = "PB22", "PB23";
          allwinner,function = "uart0";
          allwinner,drive = <0>;
          allwinner,pull = <0>;
        };
        ...
      };
    };
  };
```

UART 0 pin mux config

Board Level

arch/arm/boot/dts/sun7i-a20-olinuxino-micro.dts

```
/ {
    soc@01c00000 {

      pio: pinctrl@01c20800 {
        led_pins_olinuxino: led_pins@0 {
          allwinner,pins = "PH2";
          allwinner,function = "gpio_out";
          allwinner,drive = <1>;
          allwinner,pull = <0>;
        };
      };

      uart0: serial@01c28000 {
        pinctrl-names = "default";
        pinctrl-0 = <&uart0_pins_a>;
        status = "okay";
      };
    };

    leds {
      compatible = "gpio-leds";
      pinctrl-names = "default";
      pinctrl-0 = <&led_pins_olinuxino>;

      green {
        label = "a20-olinuxino-micro:green:usr";
        gpios = <&pio 7 2 0>;
        default-state = "on";
      };
    };
  };
```

Enable UART0 and associate pin mux config

Declare LED device and associate pin mux config

Références

- Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support.
  http://free-electrons.com
- Wiki – Device Tree Usage
  http://devicetree.org/Device_Tree_Usage
- ePAPR Version 1.1
  https://www.power.org/wp-content/uploads/2012/07/Power_ePAPR_APPROVED_v1.1.pdf
- Experiences with device tree support development for ARM based SoC's
  http://elinux.org/images/4/48/Experiences_With_Device_Tree_Support_Development_For_ARM-Based_SOC%27s.pdf
- Device Trees
  http://elinux.org/Device_Trees
- U-BOOT cmd fdt
  http://www.denx.de/wiki/view/DULG/UBootCmdFDT