REDS
Reconfigurable & embedded Digital Systems

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

Systèmes d'exploitation et Environnements d'Exécution Embarqués (SEEE)

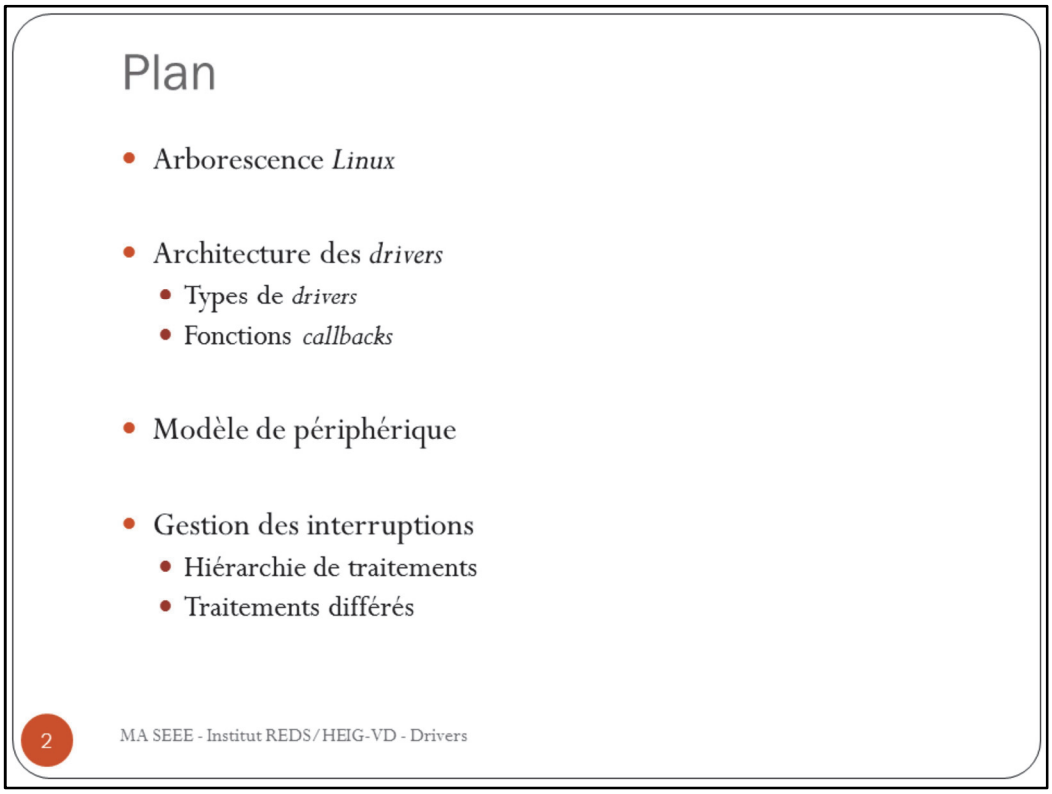
Drivers

Module d'approfondissement MSE

Prof. Daniel Rossier

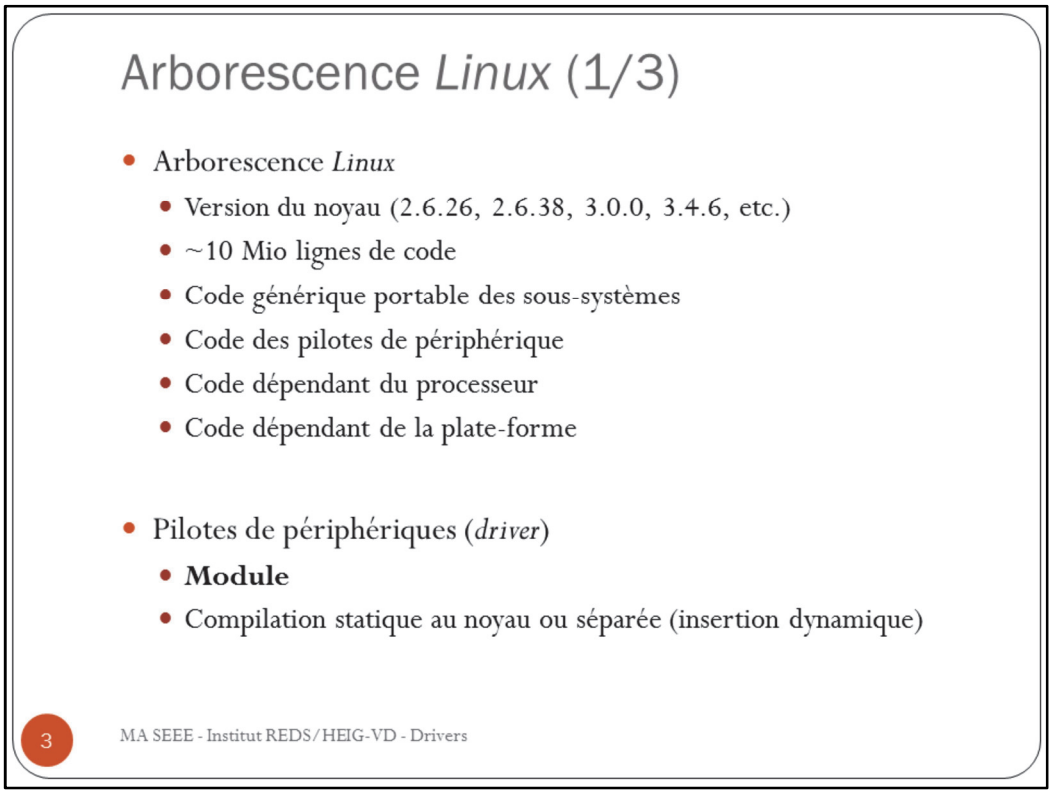
Version 1.5 (2017)

REDS
Reconfigurable & embedded Digital Systems



Les pilotes de périphériques (ou *drivers*) constituent des composants essentiels dans tout système embarqué. Ils permettent de gérer les interactions entre les applications de l'espace utilisateur et les périphériques embarqués. Ils jouent donc un rôle central dans l'architecture logicielle et apparaissent souvent comme des éléments critiques d'un système puisqu'ils ont un contrôle complet sur le matériel qu'ils pilotent. Aussi, les interactions avec le noyau du système d'exploitation sont nombreuses et doivent être maîtrisées afin de pouvoir garantir une sécurité maximale et des temps de réponse adéquats.

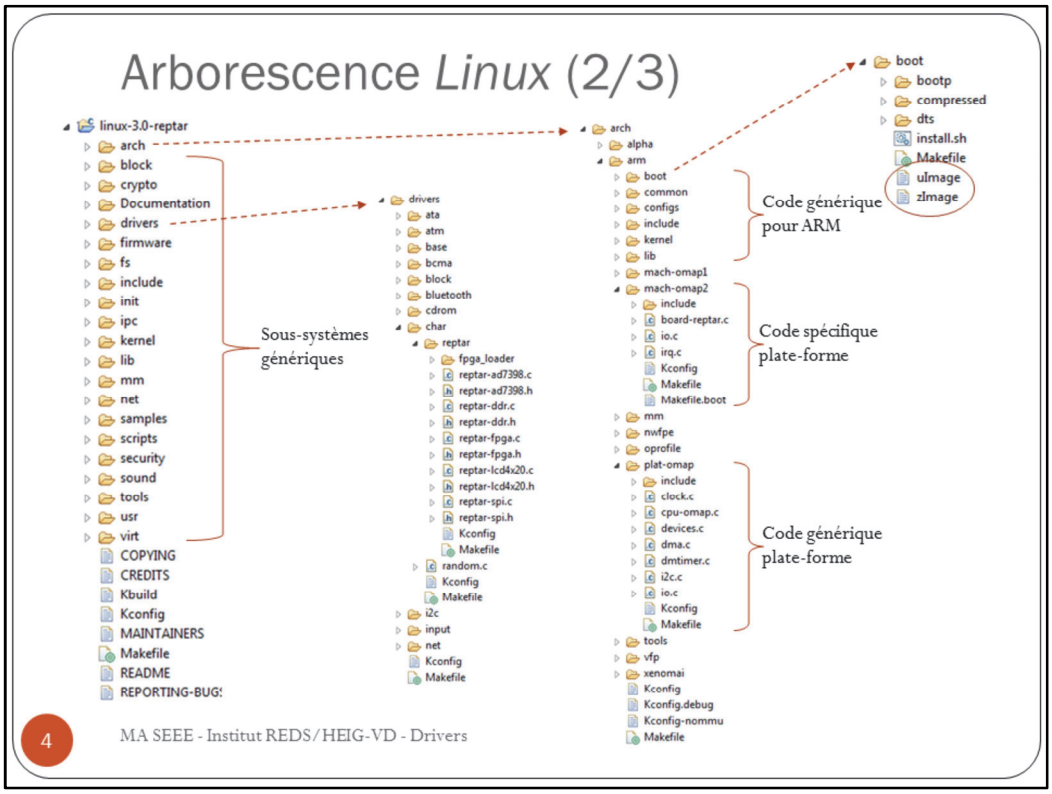
Dans ce chapitre, nous examinons les caractéristiques principales d'un *driver* sous *Linux*. Ce type de *driver* est très souvent considéré dans le développement de systèmes embarqués 32 bits et couvre une large palette de périphériques. De plus, une bonne compréhension des mécanismes de gestion au niveau d'un *driver* permet d'appréhender des environnements d'exécution plus complexes de plus en plus présents sur les microcontrôleurs de dernière génération.



Le développement de *driver* sous *Linux* nécessite une bonne compréhension de l'organisation du code du noyau. En effet, il faut se souvenir que *Linux* est une architecture d'OS de type **monolithique**, c-à-d que l'ensemble du code système se trouve dans l'espace noyau. Un *driver* fait partie de cette catégorie de code et peut donc accéder les différentes structures de données et fonctions disponibles dans le noyau.

L'arborescence comprend une partie générique commune à toutes les architectures de CPU et aux plates-formes. Les algorithmes d'ordonnancement, d'allocation mémoire, de stratégie de sécurité, de gestion de protocoles, etc. comportent une partie indépendante du matériel. L'arborescence comprend également une partie dépendante de l'architecture du processeur et du matériel en général. C'est au travers de la configuration du noyau que le code correspondant au processeur, à la plate-forme et aux différentes matériels utilisés, **sera compilé**.

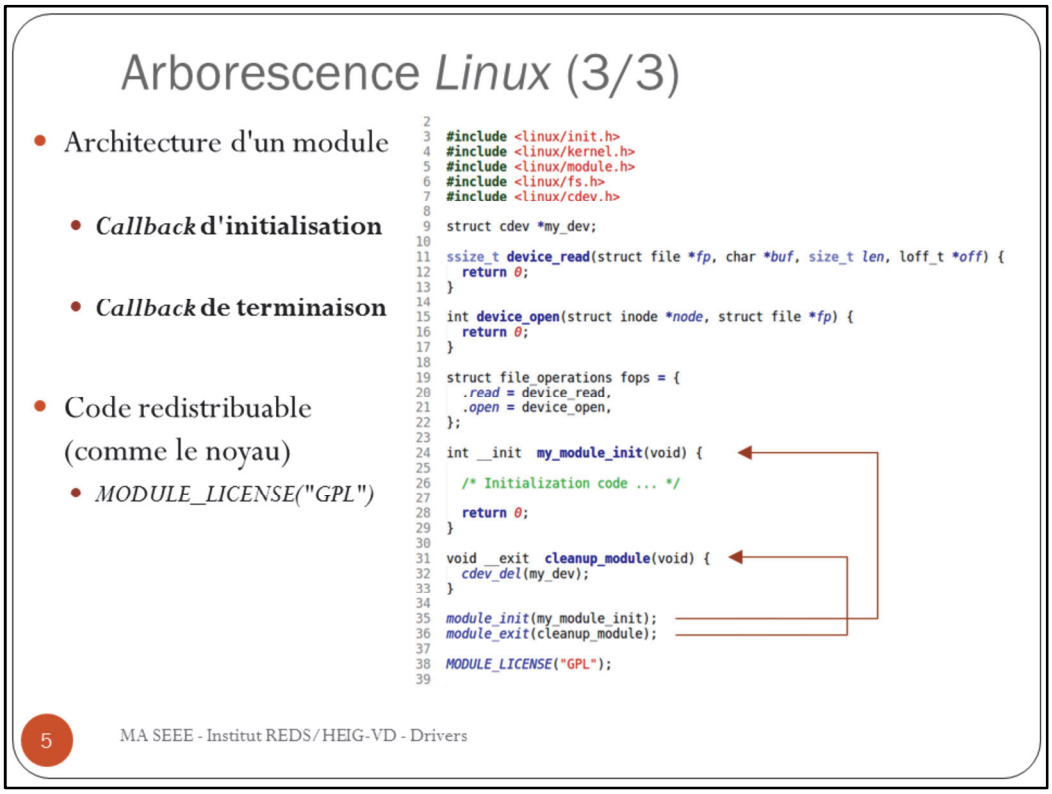
Sous *Linux,* les drivers quel qu'ils soient, sont tous considérés comme des ***modules***. Un module a une structure bien défini en terme de fonction d'initialisation et de terminaison. Il peut être compilé de manière statique, faisant partie intégrante du noyau et initialisé lors du *bootstrap* du noyau, ou être compilé de manière séparée sous forme d'un fichier binaire avec l'extension ***.ko***. Dans ce cas, le module sera inséré dynamiquement après l'initialisation du noyau et montage du *rootfs*. C'est lors de son insertion qu'il sera initialisé (à l'aide de la commande ***insmod*** ou ***modprobe*** par exemple).



L'arborescence *Linux* contient un sous-répertoire ***arch*** contenant le code spécifique aux processeurs et aux plates-formes. Dans le cas d'un CPU ARM, il existe également une partie de code générique aux types de plates-formes; par exemple, les architectures OMAP disposent de codes génériques que l'on trouve dans le sous-répertoire ***plat-omap***. Le code dépendant de la plate-forme se trouve dans un répertoire commençant par ***mach-***. L'image binaire du noyau résultant de la compilation se trouve dans ***arch/arm/boot***. Le fichier ***zImage*** est une image standard compressée incluant le code de décompresseur. Le fichier ***uImage*** contient quelques octets d'en-tête utilisés par la commande ***bootm*** de *U-boot.* Ce fichier est produit avec l'utilitaire *mkimage (*disponible dans le *package u-boot-tools*).

Le code des *drivers* est généralement situé dans un sous-répertoire ***drivers***.

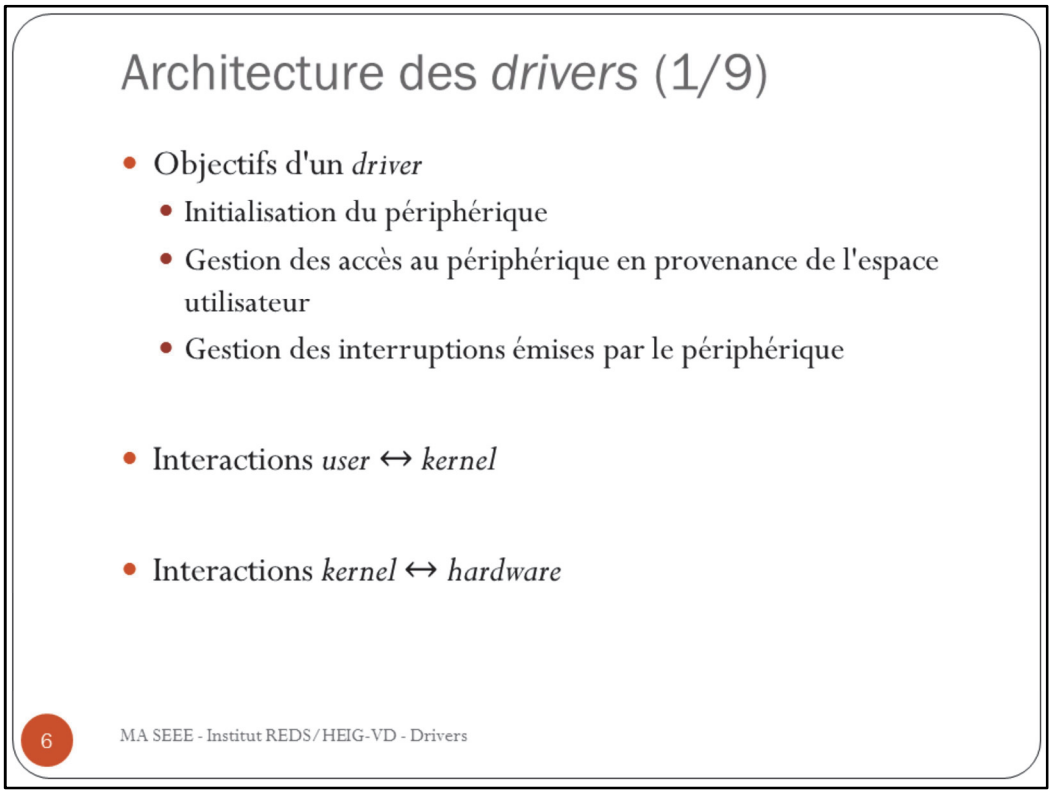
Il faut noter la présence des répertoires ***include*** situés à différents endroits: à la racine de l'arborescence, le répertoire ***include*** contient l'ensemble des définitions requises par le code générique, celui présent dans ***arch/arm/*** contient les définitions du code générique ARM, celui présent dans ***arch/arm/mach-omap2*** l'ensemble des définitions nécessaires pour ce type de plate-forme, etc.
Malheureusement, cette belle organisation n'a pas été toujours présentée sous cette forme et les versions antérieures à 2.6.27 ne comprennent qu'un répertoire ***include*** à la racine avec des liens symboliques vers des sous-répertoires au même niveau dépendant de l'architecture.



Un module au sens *Linux* est un programme contenant au minimum deux fonctions de type *callback* enregistrées au sein du noyau avec les macros ***module_init()*** et ***module_exit*()**. Ces deux *callbacks* sont appelés respectivement lorsque le module est chargé dans le noyau (soit durant le démarrage du noyau si le module est compilé statiquement, soit durant l'opération de chargement dynamique via ***insmod*** par exemple), et lorsque le module est retiré du noyau (opération ***rmmod***) (il n'est pas possible de retirer un module compilé statiquement avec le noyau).

La compilation d'un module doit s'effectuer avec le ***Makefile* du noyau**. C'est pourquoi, la compilation nécessite d'avoir soit l'arborescence complète du noyau, soit une version minimum (appelée *linux-headers*) contenant les fichiers d'entête ***.h*** ainsi que les *Makefile* nécessaires. La version minimum peut-être présente dans le répertoire ***/lib/modules*** du *rootfs* de la cible (si toutefois ce répertoire a été installé).

Un *driver* est donc caractérisé par un fichier contenant les *callbacks* mentionnés ci-dessus, ainsi qu'une série d'appels de fonction noyau permettant de s'enregistrer auprès du noyau en tant que *driver*.

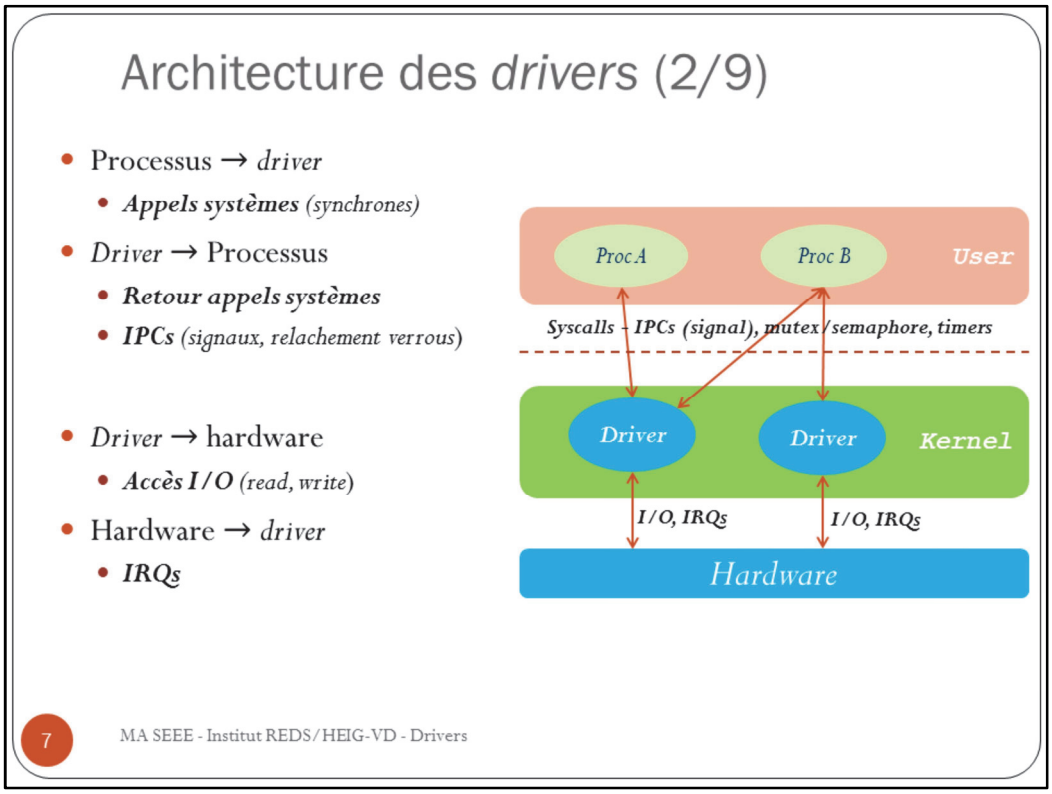


Un *driver* est responsable de gérer les accès aux matériels par les applications utilisateurs. L'approche monolithique conventionnelle consiste à placer le code des *drivers* dans l'espace noyau de telle manière à ce que le *driver* puisse disposer librement des fonctionnalités et des structures de données du noyau. La présence des *drivers* dans l'espace noyau permet de limiter les interactions avec l'espace utilisateur et donc d'obtenir d'excellentes performances.

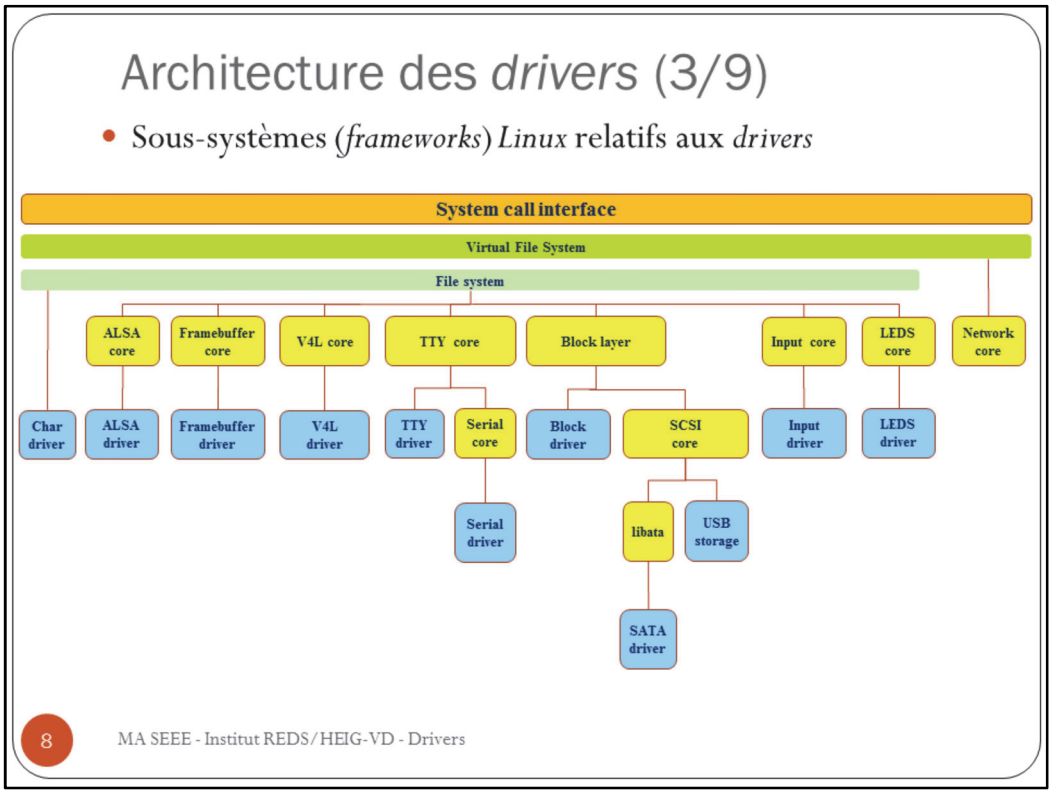
Un *driver* se trouve au cœur de l'architecture logicielle et doit gérer les interactions de l'espace utilisateur vers l'espace noyau, et les interactions entre le noyau et le matériel. Par conséquent, le développement d'un *driver* est généralement plus complexe que le développement d'une application logicielle "traditionnelle", car il doit gérer de multiples requêtes en provenance des processus (utilisateur) et du matériel via les interruptions (IRQs).

En particulier, le code est généralement réentrant car il n'est pas rare que plusieurs applications souhaitent utiliser le même périphérique (dispositif graphique de type *framebuffer* (LCD), écran tactile, souris, etc.). Les mécanismes de verrous doivent être utilisés de manière judicieuse, les sections critiques clairement identifiées.

La gestion des *buffers* doit être optimale, car très souvent le *driver* doit être réactif; on ne peut se permettre d'introduire trop de latence (délai) au risque de ralentir considérablement les applications utilisateur.



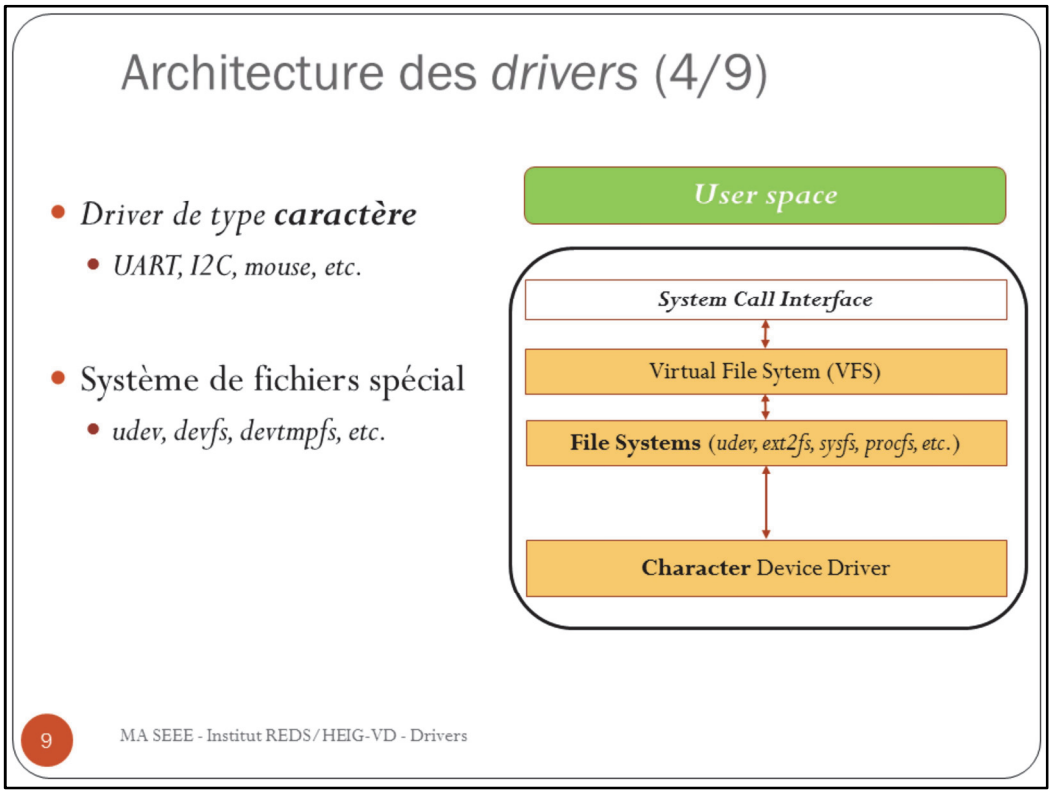
Sans grande surprise, un processus doit utiliser les appels systèmes s'il souhaite accéder aux fonctionnalités d'un périphérique. L'appel système peut être synchrone ou asynchrone; dans ce dernier cas, le processus sera informé via des signaux POSIX par exemple. Un *thread* dédié pourrait également être utilisé pour gérer les interactions avec le *driver*. Il est également possible de **mapper une zone mémoire** de type I/O dans l'espace utilisateur afin de permettre aux processus d'accéder directement la mémoire du périphérique. C'est le cas avec les cartes graphiques par exemple. L'accès au matériel par le *driver* s'effectue généralement avec des **adresses I/O**. On se rappelle que le *driver* ne peut pas utiliser directement des appels système au même titre qu'un processus (puisqu'il se trouve déjà dans le noyau), et que l'utilisation de certains services peut engendrer certaine complication (utilisation de fichiers par exemple). C'est pourquoi, il est indispensable de bien **découpler** les fonctionnalités de *driver* et des fonctionnalités applicatives qui doivent rester au niveau de l'espace utilisateur. Les **interruptions matérielles (IRQs)** permettront au matériel de signaler un événement spécifique (données prêtes, alarmes, actions de l'utilisateur, etc.) mais le processus en cours d'exécution n'a peut-être rien à voir avec celui qui a initié la requête. C'est pourquoi, le *driver* doit souvent gérer des **queues de processus** en attente de données. Finalement, le matériel peut exécuter **des accès DMA** (en lecture/écriture) vers une zone mémoire dédiée. Une interruption est généralement levée pour signaler la disponibilité des données. Ce mécanisme peut nécessiter l'intervention d'un **contrôleur DMA** externe au périphérique (intégré dans le microcontrôleur) à moins que le périphérique dispose de son propre contrôleur DMA.



Un *driver* peut s'exécuter dans le noyau de manière relativement indépendante lorsqu'il implémente les ***callbacks*** directement associés aux appels systèmes. Cependant, la plupart des *drivers* pilotent des périphériques classiques d'entrées-sorties appartenant à une catégorie (classe) de **fonctionnalités** bien définie (communication sérielle de type UART, communication réseau, dispositif d'acquisition, sortie vidéo, etc.). C'est pourquoi le *driver* s'interface généralement sur d'autres composants du noyau constituant un sous-système particulier.

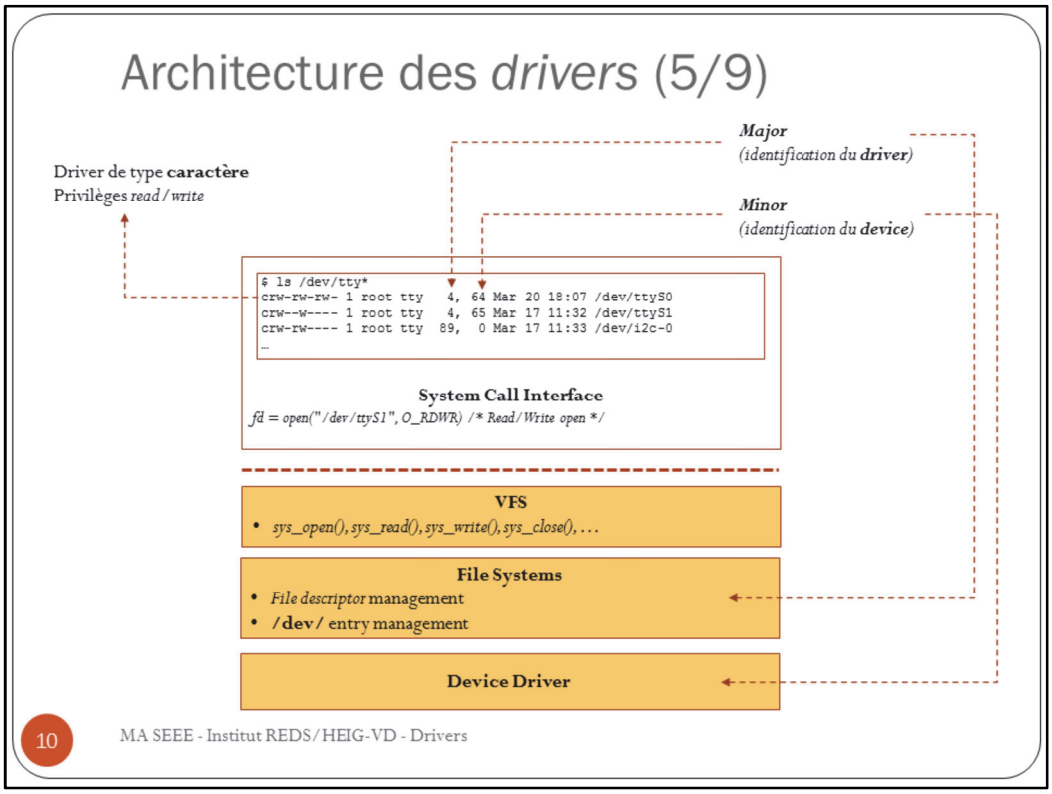
C'est le cas des dispositifs de type ***input*** comme une souris, un clavier, un écran tactile, etc. Dès qu'une information en provenance de ce type de dispositif est récupéré par le *driver,* celle-ci est traitée est transférée dans le sous-système des *inputs* de *Linux*. A titre d'exemple, l'appui sur la touche retour (*carriage return*) ou un *clique* de souris aura souvent le même effet.

Un exemple pratique de sous-système pouvant être utilisé par un *driver* de périphérique embarqué est celui des ***leds***. *Linux* met à disposition une **classe *leds*** permettant de contrôler l'affichage, l'extinction et le clignotement de *leds*. Le *driver* doit alors s'enregistrer auprès de cette classe afin de pouvoir réagir correctement à ces différents comportements. L'application utilisateur disposera ensuite d'une entrée spécifique dans */sys*.



Les *drivers* de type **caractère** sont simples à réaliser et très utilisés dans les systèmes embarqués. Les périphériques de ce type permettent l'accès au *byte* près. Comme le montre l'architecture ci-dessus, aucun sous-système particulier n'est requis (hormis le système de fichiers). Les accès sont généralement de type séquentiel (non aléatoire). Ils implémentent les fonctions de type *callback* permettant le transfert de données depuis et vers l'espace utilisateur. La complexité du *driver* dépendra naturellement du périphérique à gérer.

Comme tout driver sous *Linux*, un *driver* de type caractère doit implémenter des *callbacks* définis par le système de fichiers responsable de gérer les entrées dans le répertoire */dev*.



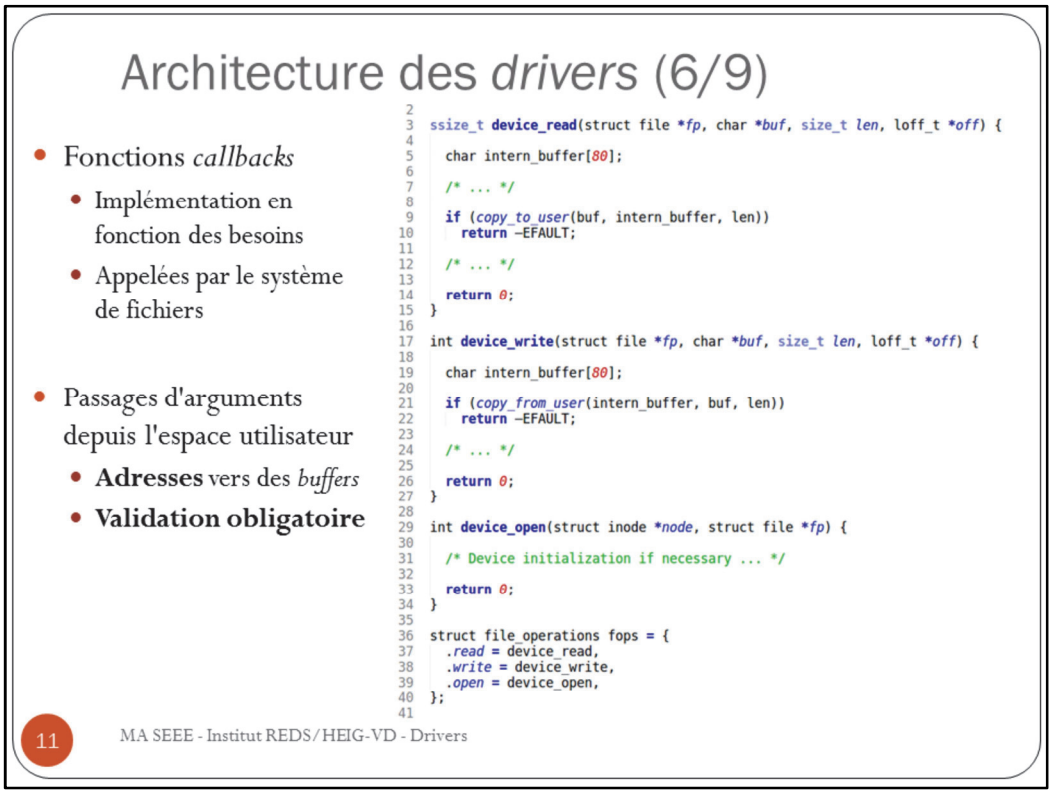
Les *drivers* de type caractère exposent leurs interfaces vers l'espace utilisateur au moyen du système de fichiers. Il n'y a donc pas un **lien direct** avec les applications. On voit nettement sur la figure ci-dessus que l'application doit utiliser une interface commune aux fichiers pour utiliser les fonctionnalités des *drivers*. Le développeur peut ainsi bénéficier d'un ensemble commun d'appels systèmes qui sont utilisés à la fois pour la gestion des fichiers et des périphériques. Au niveau du noyau, seul le type de système de fichiers diffère: pour les *drivers*, ce sont les entrées dans le répertoire système ***/dev*** qui sont gérées d'une manière très particulière. Chaque entrée de */dev* est associée avec deux constantes: le ***major*** et ***le minor***.

Le *major* permet au système de fichier d'identifier le *driver* associé avec l'entrée de répertoire, le *minor* est transmis au *driver* et permet d'identifier (de manière unique) un des périphériques qu'il doit gérer (par exemple, dans le cadre de l'UART, il peut y avoir plusieurs ports de communication). La plupart des *drivers* n'ont cependant qu'un seul *minor* à gérer (valeur 0).

Les entrées de */dev* peuvent être créées avec l'utilitaire ***mknod:***

mknod [*options***]** *nom* **{bc}** *numéro_majeur numéro_mineur*

où **b** signifie le type *bloc,* **c** signifie le type *caractère*.

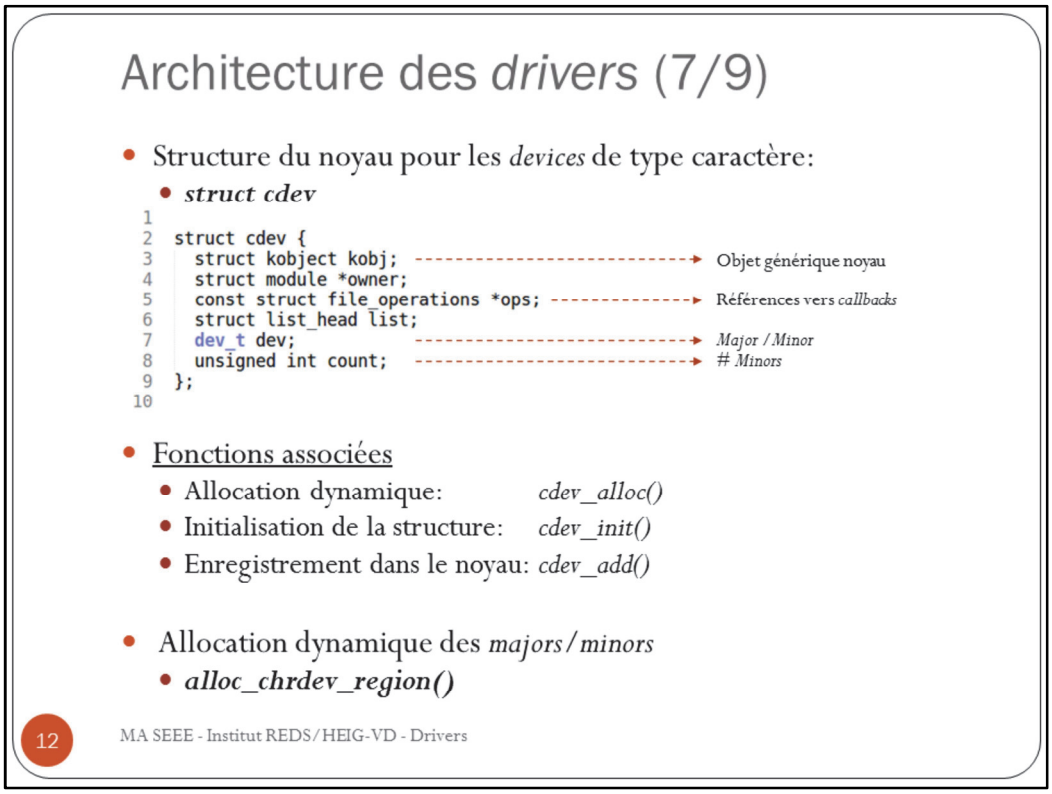


Lorsque le *driver* est identifié par le système de fichiers via le *major*, les fonctions du *driver* peuvent être appelées (par le FS) sous forme de ***callback***.

Examinons le chemin d'appel de fonction pour l'appel système *open()*:

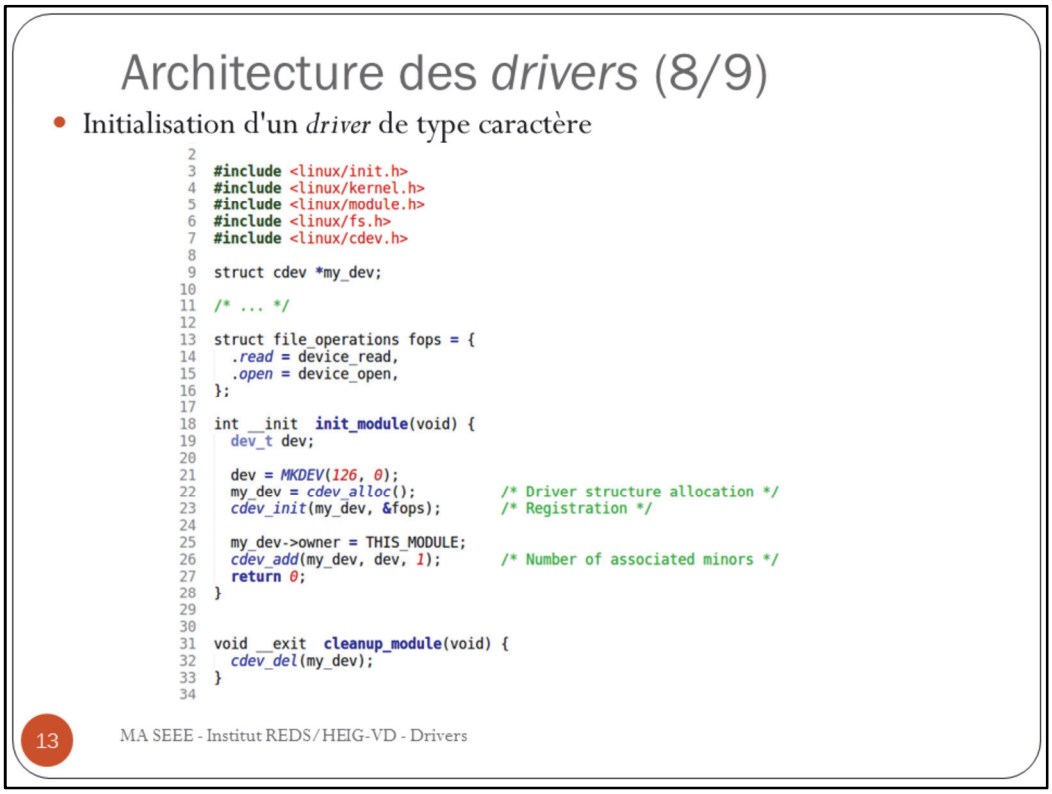
1) Le processus exécute l'appel système *open("/dev/ttyS2", O_RDWR).*
2) La couche VFS identifie que cette entrée de répertoire appartient à */dev* qui est "*monté*" avec le système de fichier de type *udev (*ou *devfs*). VFS gère également le *file descriptor* qui sera retourné par l'appel système *open()*.
3) Le système de fichiers identifie le *driver* via le *major* et invoque le *callback open()* correspondant dans le *driver* pour autant que celui-ci l'ait défini.
4) Le *driver* peut effectuer des initialisations du périphérique dont il a la responsabilité.

Lorsque le processus doit transmettre des données au *driver*, récupérer des données, un pointeur vers un *buffer* du processus (espace utilisateur) est transmis au *driver*. Généralement, le *driver* travaille avec des *buffers* internes et des opérations de copie sont effectuées.
Afin de **valider** les adresses transmises par l'espace utilisateur (via les *syscalls*), les fonctions ***copy_from_user()*** et ***copy_to_user()*** doivent être utilisées. Sinon, il se pourrait que l'application transmette des adresses illégales, ce qui pourrait corrompre le système. Dans ce cas, il s'agirait clairement d'une faille de sécurité dans le *driver*.



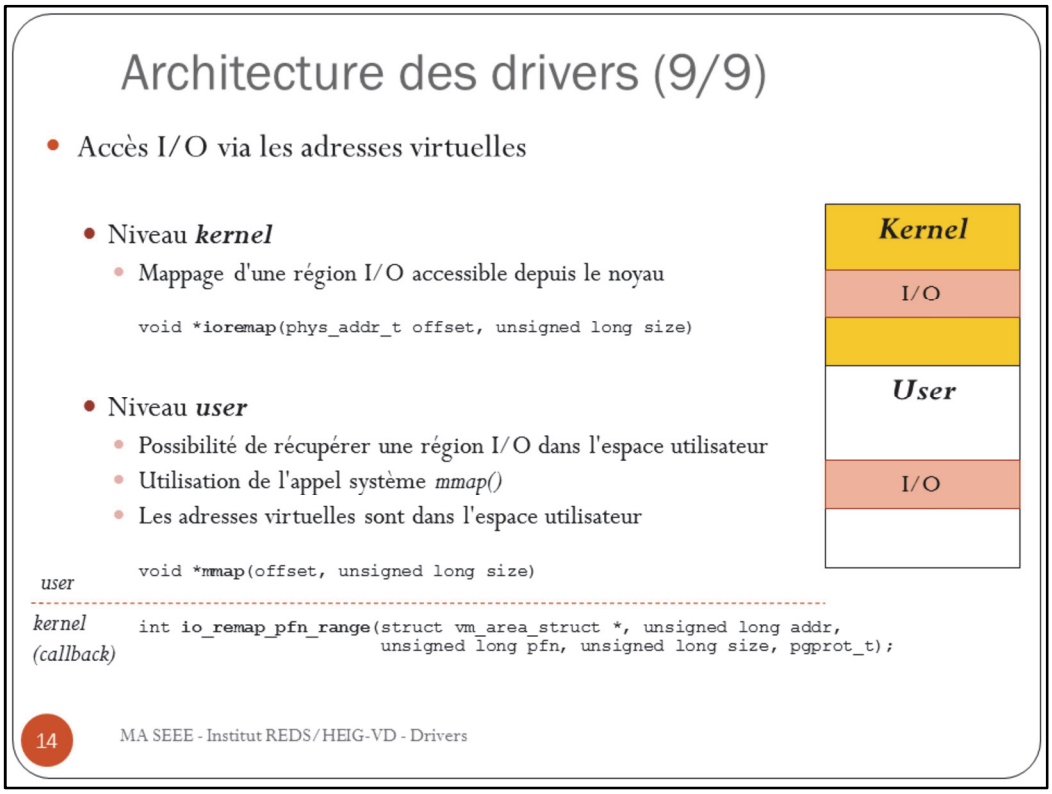
Un *driver* de type caractère nécessite l'utilisation de la structure du noyau ***cdev***. Cette structure permet de faire le lien avec les *callbacks* du *driver* ainsi que le *major* et *minor(s)* associés. La structure peut être allouée dynamiquement avec la fonction *cdev_alloc()*.

L'attribution des *majors* peut s'avérer relativement compliquée si la plate-forme dispose d'un grand nombre de périphériques. C'est pourquoi *Linux* dispose d'un mécanisme d'allocation dynamique de *major* et de *minor.* La fonction *alloc_chrdev_region()* peut être sollicitée afin de récupérer ces informations et les stocker dans la structure *cdev*.



Le lien principal entre l'espace utilisateur et l'espace noyau, dans le cadre d'un *driver* de type caractère, se réalise grâce au *major* et au *minor.* Comme nous l'avons vu précédemment, une entrée spéciale est crée dans le répertoire */dev* associée avec ces deux numéros. Dans l'espace noyau, la macro *MKDEV()* (cf ci-dessus) permet d'initialiser une structure de type *dev_t* qui permettra de stocker le majeur et le mineur.

Comme le montre l'exemple ci-dessus, la structure *cdev* contient toutes les informations relatives au *driver*. La structure de type *file_operations* permettra de déclarer les *callbacks* qui seront implémentés par le *driver*.

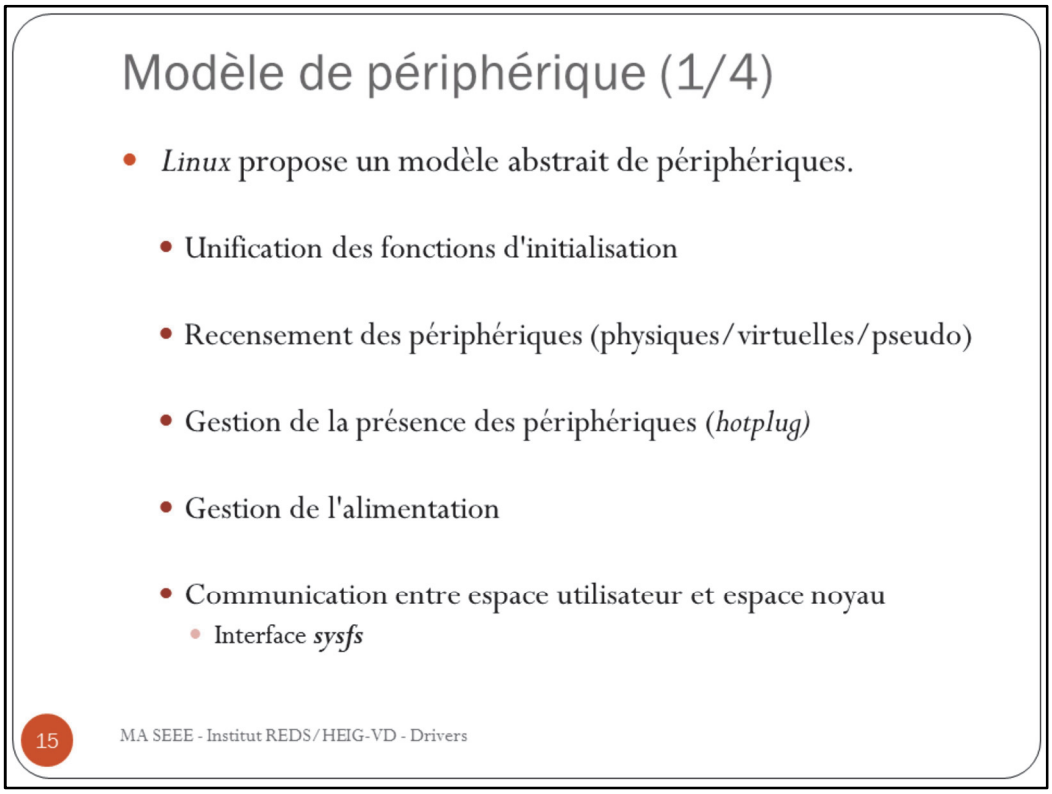


Il peut arriver qu'un *driver* demande au noyau de mapper une zone physique I/O afin de pouvoir l'accéder (adresses virtuelles); ce mappage n'est pas forcément défini lors du *bootstrap* du noyau et peut s'effectuer à la volée. La fonction ***ioremap()*** est utilisée dans ce but.

Comme il a été évoqué au début de ce chapitre, un *driver* peut aussi exposer une zone mémoire I/O accessible par le processus dans l'espace utilisateur. Le principe repose sur l'utilisation d'un appel système puissant: ***mmap()***. Lorsque le processus utilise cet appel système, le *driver* peut retourner l'adresse virtuelle d'une zone mémoire mappée sur une zone I/O. Bien entendu, le mappage doit être contrôlé par le *driver* (typiquement avec la fonction ***io_remap_pfn_range()***). Cette technique permet d'éviter l'utilisation intensive d'appels systèmes (*read() / write()*) dans certain cas, comme avec une carte graphique par exemple.

Il est cependant déconseillé d'adopter systématiquement cette approche pour tout accès I/O; Le mappage I/O dans l'espace utilisateur peut laisser la porte ouverte aux failles de sécurité et ne s'avère pas forcément plus performante par rapport aux appels systèmes. De plus, certaines architectures matérielles ne supportent pas bien l'utilisation d'adresses virtuelles I/O en mode utilisateur.

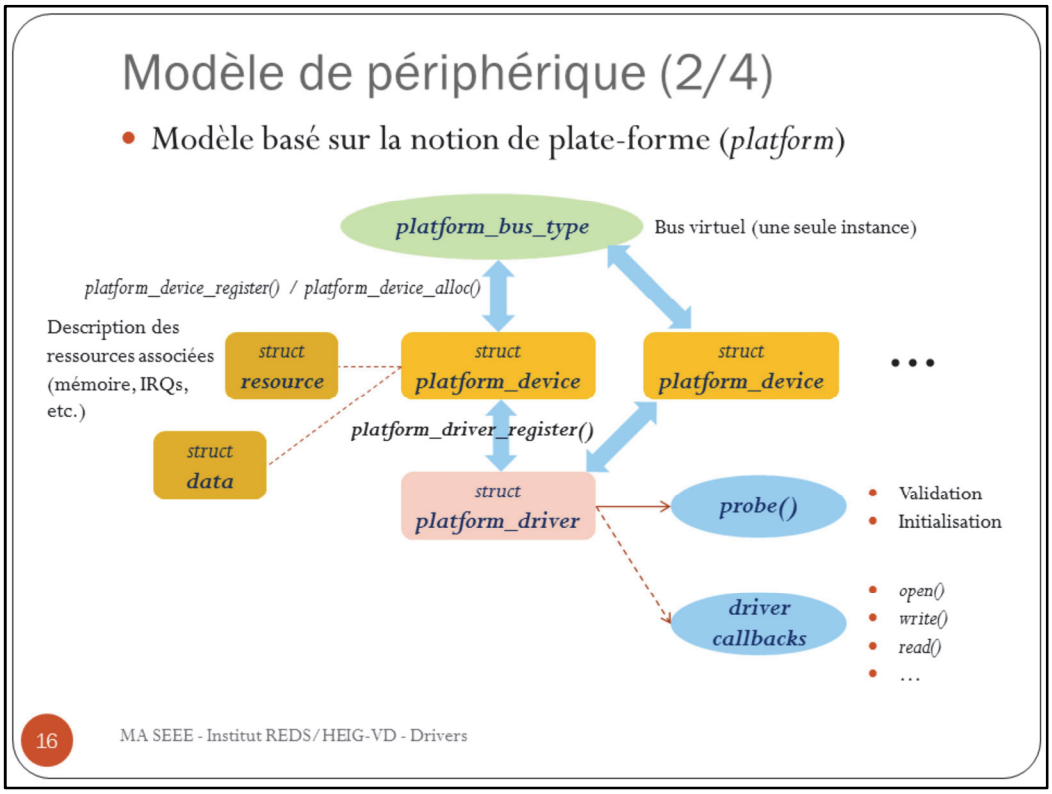
(La structure *vm_area* définit un mappage d'adresses virtuelles contigües possédant les mêmes propriétés)



Afin de faciliter l'implémentation des *drivers* et les mécanismes de communication entre objets du noyau et l'espace utilisateur, *Linux* a élaboré au fil des dernières années un modèle abstrait de périphérique. La gestion de la complexité et de la diversité des *drivers* ont nécessité le développement d'un modèle unifié, mais tous les *drivers* disponibles dans le noyau n'ont pas encore été *alignés* sur ce modèle. C'est pourquoi, on trouve dans le noyau des *drivers* qui sont parfaitement intégrés à ce modèle (qui utilisent les mécanismes inhérents au modèle) et des *drivers* (plutôt anciens) qui n'ont aucun lien avec le modèle.

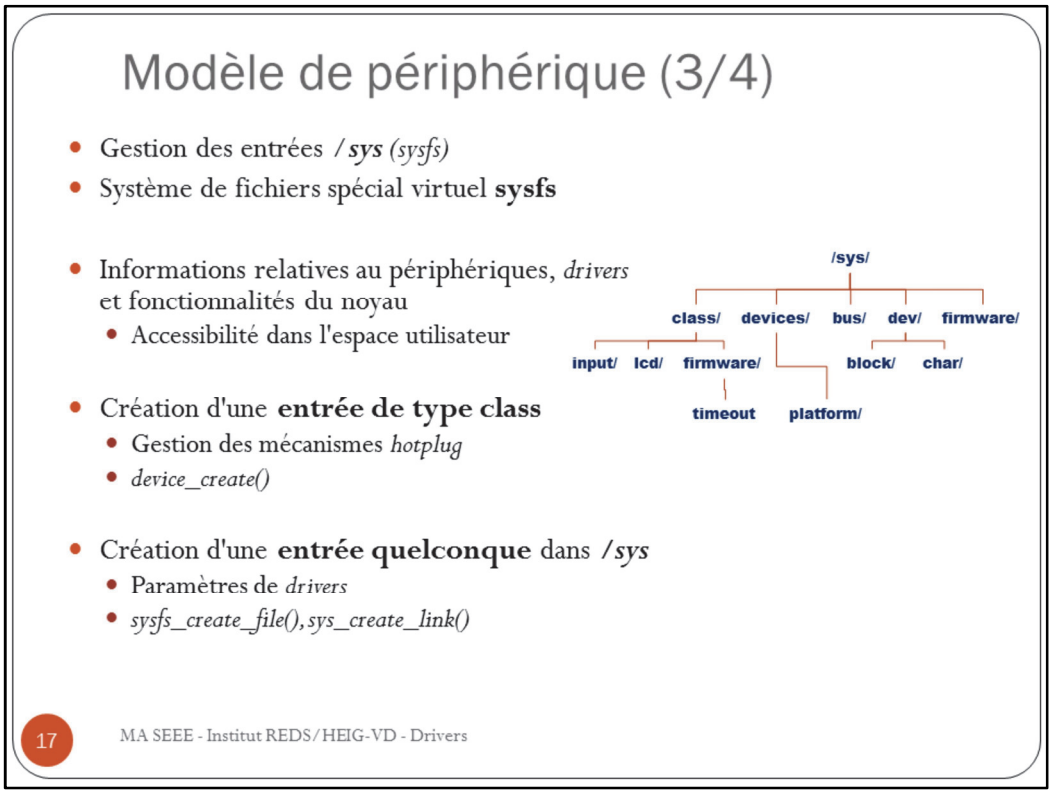
Les avantages d'utiliser un modèle abstrait de périphériques sont nombreux: ils permettent de classifier les périphériques en fonction de leur type et des fonctionnalités supportées, de gérer des fonctions *génériques* comme la gestion de l'alimentation, le *hotplug* (introduction de périphérique "*à chaud*", retirement de périphériques), le recensement des périphériques, la possibilité de consulter l'ensemble des *drivers* appartenant à un bus commun (PCI par exemple), et d'une manière générale, d'optimiser les structures du noyau requises par les *drivers*.

Une des grandes forces du modèle est la possibilité d'utiliser une interface dans l'espace utilisateur commune à tous les périphériques, au travers du système de fichier spécial ***sysfs***. Les informations contenues dans ce système de fichiers renseignent sur l'ensemble des périphériques du système (état, variables de configuration, etc.), et, dans la mesure où les *drivers* le permettent, des paramètres peuvent être modifiés aisément à l'aide d'appels systèmes de type VFS.



Si les notion de *bus, device* et *class* permettent de gérer efficacement l'ensemble des périphériques d'un système, en facilitant grandement la gestion du *hotplug* ou encore du *power management*, ce modèle peut s'avérer compliqué (et parfois inutile) dans le contexte des systèmes embarqués. En effet, il est plus rare de voir apparaître ou disparaître des périphériques à la volée (*hotplug)* et le nombre de périphériques est souvent plus limité. C'est pourquoi, *Linux* présente un modèle simplifié pour les plates-formes embarquées, avec les structures ***platform_device*** et ***platform_driver.*** Le modèle est ainsi étendu avec ces structures, ce qui garantit une compatibilité avec l'ensemble des composants du noyau qui reposent sur la notion de *bus, device* et *class.*
Une particularité du **modèle orienté plate-forme** est la présence d'un **bus *virtuel*** sur lequel se greffe l'ensemble des périphériques enregistrés. Ce bus permet de disposer aisément de facilités comme des ***itérateurs*** sur l'ensemble des *devices* et des *drivers* sans toutefois à se préoccuper de mécanismes de *hotplug* ou de liaison dynamique *device-driver*.

La liaison (***binding***) dynamique entre *device* et *driver* est automatiquement faite lorsque le *driver* **s'enregistre** dans le noyau: une comparaison ***(matching)*** entre **nom** de *device* et nom de *driver* est effectuée afin d'identifier le-s *device-s* concerné-s. Il est toutefois possible d'implémenter un mécanisme de *matching* plus élaboré basé sur une table d'identification. Dès qu'un *binding* est effectué, un *callback* spécifique appelé ***probe*** est exécuté est permet au *driver* d'effectuer les opérations d'initialisation. **Le *device* est passé comme argument au *probe*.**

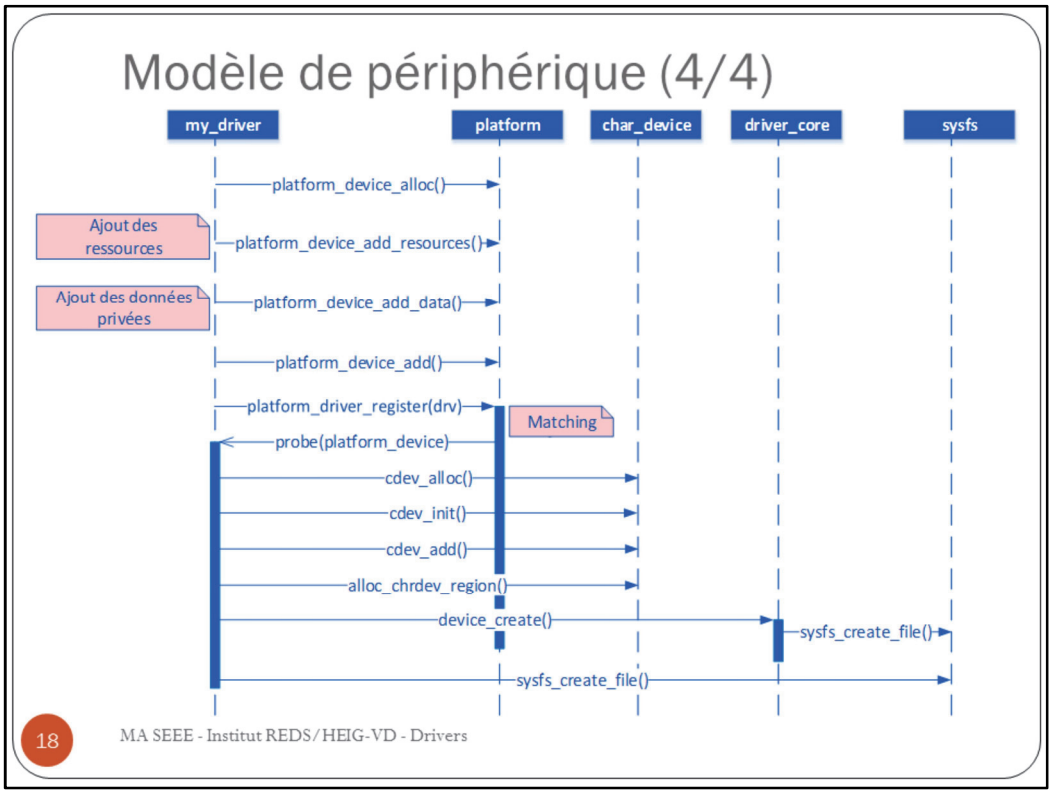


Comme il a déjà été évoqué précédemment, les entrées de répertoire dans */sys* sont virtuelles et gérées directement par le noyau au moyen d'un système de fichiers spécial appelé ***sysfs***.

Grâce à ces entrées, il est possible, depuis l'espace utilisateur, d'accéder à diverses informations relatives aux périphériques (physiques et virtuels) de la plate-forme en cours de fonctionnement. Dans ce contexte, on retrouve à la racine de l'arborescence les objets fondamentaux du noyau correspondant aux modèles de périphérique; le répertoire ***sys/devices/platform*** contient l'ensemble des *devices* enregistrés dans le modèle orienté plate-forme, avec la possibilité éventuelle pour certaines entrées de lire ou écrire des données de configuration.

Le système de fichiers *sysfs* est également utilisé pour la création automatique des entrées de répertoire dans ***/dev,*** autre type de répertoire virtuel gérer par le noyau et permettant l'accès aux périphériques via leurs drivers. Le principe de base repose sur le parcours des entrées dans ***/sys/class***. En fonction de l'application et des fichiers de règles/configuration associés, les entrées correspondantes sont crées dans */dev*.

La création de nouvelles entrées peut s'effectuer à différents instants: lors du *bootstrap* du noyau (module ***devtmpfs***), au lancement des scripts d'initialisation (***/etc/init.d/udev***, ***/etc/init.d/mdev***, *etc.*) ou à n'importe quel moment (application *hotplug* gérant l'apparition/disparition à chaud de périphériques).

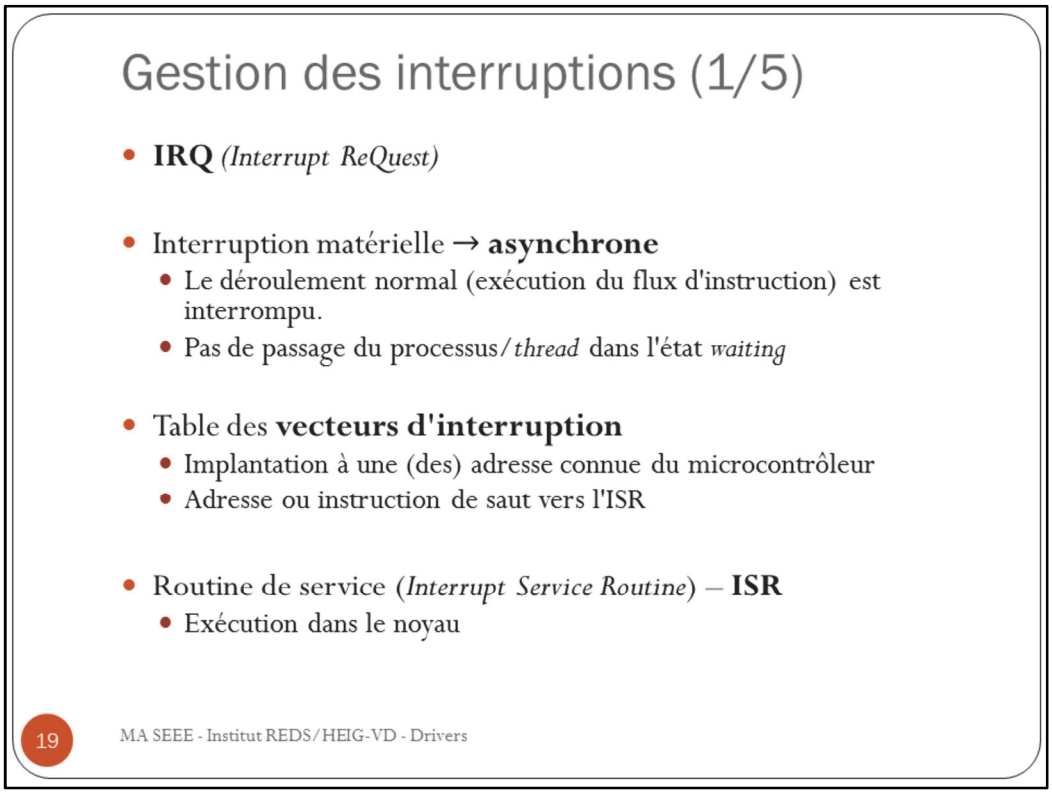


Le diagramme de séquence ci-dessus met bien en évidence les fonctions du modèle orienté plate-forme, qui sont appelées pour l'enregistrement des ***devices*** et des ***drivers*** au sein du noyau.

Les deux fonctions de base ***platform_device_register()*** et ***platform_driver_register()*** permettent alors d'activer le *callback* de type *probe* qui permettra d'effectuer le ***matching*** entre le *device* et le *driver*.

On remarque la création d'une structure de type *cdev* indiquant l'utilisation des mécanismes liés à un *driver* de type caractère. La fonction ***device_create()*** crée l'entrée dans */sys* qui permettra la création automatique d'une entrée associée au *major/minor* correspondant dans */dev*.

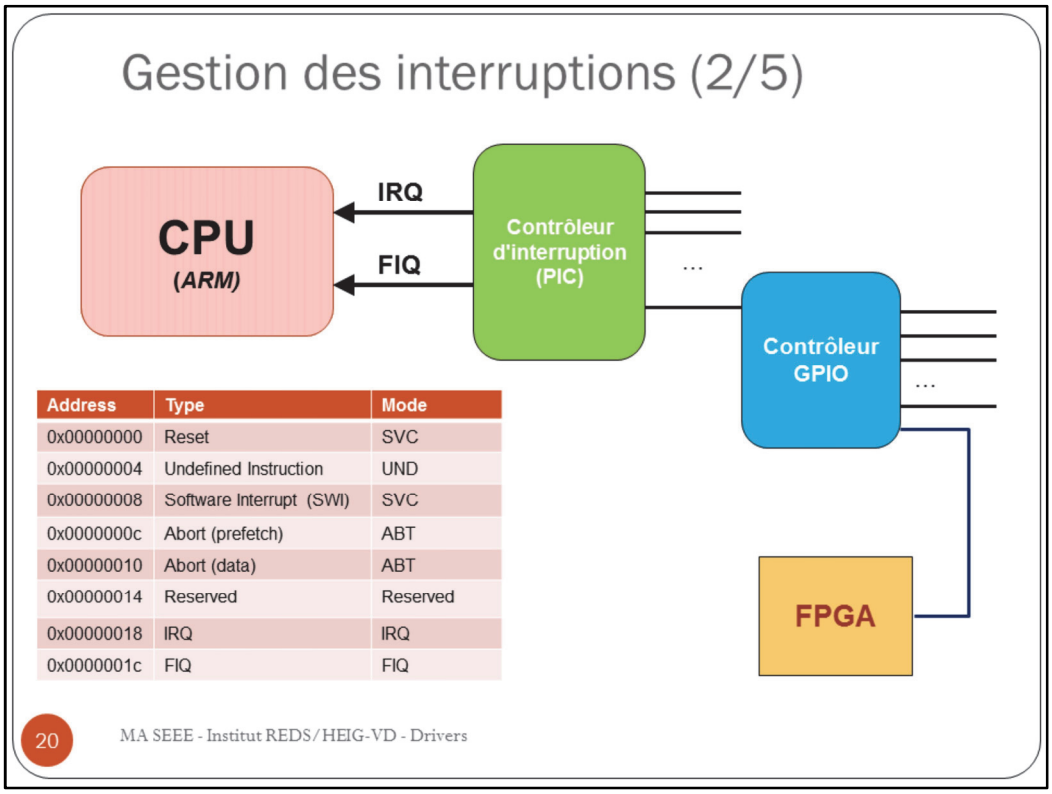
Finalement, la création d'une entrée privée dans */sys* permet à une application tournant dans l'espace utilisateur d'accéder à des paramètres exportés en lecture et/ou écriture.



L'interruption matérielle est utilisée lorsque le périphérique souhaite transmettre une information (de type événement) au processeur (CPU), par exemple, lorsqu'une donnée est prête, lors d'une alarme, ou tout autre demande qui peut être traitée ensuite par une routine de service.

La routine de service (ISR) est une fonction du noyau; son emplacement est déterminé par un vecteur d'interruption qui se trouve en mémoire à une adresse précise fixée par le CPU. Lorsqu'une interruption survient, le processeur interrompt l'instruction en cours d'exécution, passe en mode *kernel* et désactive (masque) la ligne d'interruption sur laquelle l'IRQ est arrivée. Il consulte la table des vecteurs pour extraire le vecteur associé à l'IRQ, pour finalement exécuter l'ISR correspondante. A la fin de l'ISR, le processeur pourra repasser en mode *user* et **poursuivre** l'exécution du code interrompu lors de l'interruption (l'instruction qui a été préemptée est ré-exécutée). La réactivation de la ligne d'interruption peut-être gérée complètement dans l'ISR, à n'importe quel moment (très tôt ou durant le traitement, voire à la fin de l'ISR juste avant la terminaison de la fonction).

Lorsque l'entrée IRQ est réactivée, d'autres interruptions peuvent survenir, alors même que l'ISR n'est pas terminée. Dans ce cas, le mécanisme décrit précédemment est parfaitement récursif (le CPU est toutefois déjà en mode noyau). On parle alors d'**interruptions imbriquées**. Différents scénarios peuvent alors se produire: une autre ISR est exécutée, ou la même ISR est exécutée une nouvelle fois. Dans ce dernier cas, on parle de **réentrance**, et la fonction doit gérer les éventuels accès concurrents de manière adéquate à l'aide de *mutex*, sémaphores, etc.



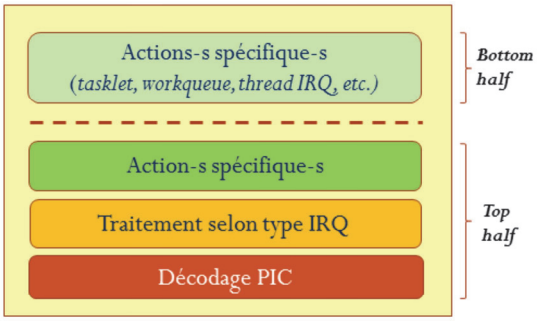
La figure ci-dessus présente une architecture matérielle classique avec un microcontrôleur de la famille ARM. La gestion des interruptions nécessite un contrôleur d'interruption ou ***PIC*** *(**P**rogrammable **I**nterrupt **C**ontroller)* sur lequel seront *reliés* les différents périphériques capables d'émettre une IRQ. En revanche, il est à noter que, du côté du CPU, seul deux lignes d'interruption (*IRQ, FIQ*) sont disponibles; dans ce cas, deux vecteurs seront suffisants. En amont du PIC, on peut trouver un contrôleur GPIO offrant la possibilité de connecter toute sorte de périphériques externes. Dans ce cas, une IRQ pourra être propagée via le **contrôleur GPIO** et le PIC.

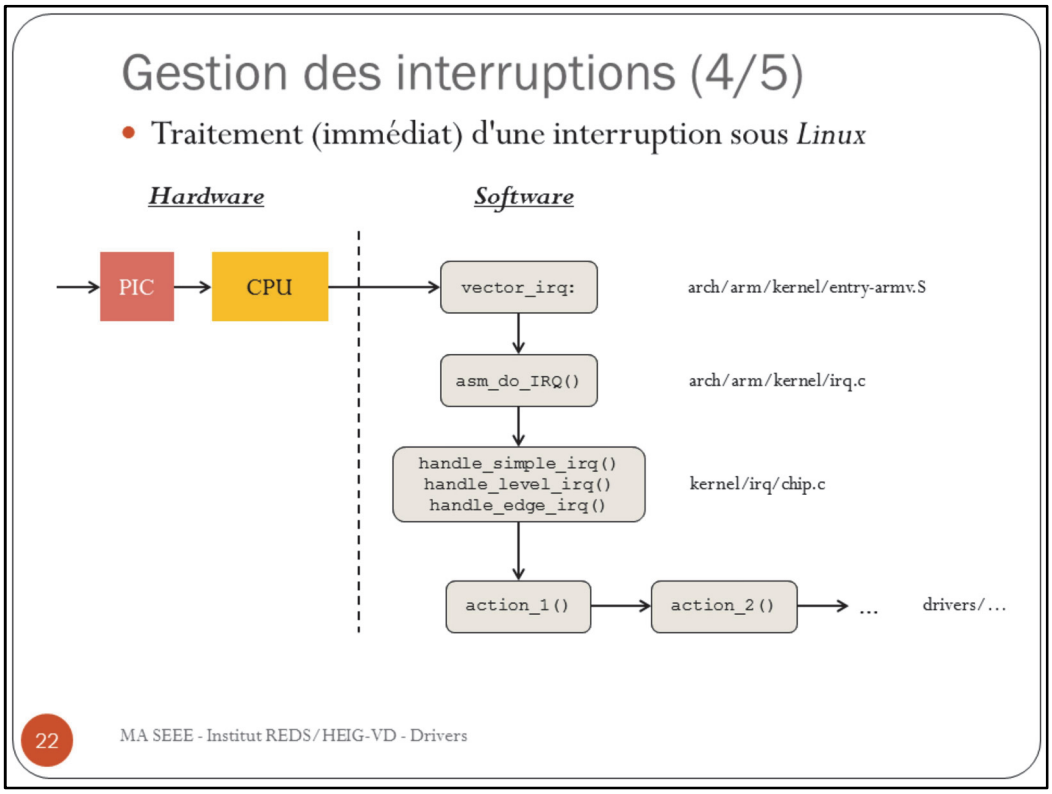
Cette ***hiérarchie matérielle*** met bien en évidence la nécessité de traiter l'interruption à différents niveaux. Lorsque l'ISR est exécutée, la source de l'interruption doit être identifiée. Il s'agit pour cela d'*interroger* le PIC et le contrôleur GPIO. La plupart du temps, il est également nécessaire *d'acquitter* l'interruption au niveau du PIC, du contrôleur GPIO et du périphérique. Le masquage de l'interruption peut se faire "à la source", c-à-d au niveau du contrôleur GPIO. De ce fait, le *canal IRQ* du PIC et, bien entendu la ligne IRQ du CPU (respectivement FIQ) peuvent être réactivées le plus rapidement possible afin de garantir une réactivité maximale.

Le PIC peut également temporiser les IRQs en cas de masquage et les délivrer selon des priorités, en fonction de la configuration.

Gestion des interruptions (3/5)

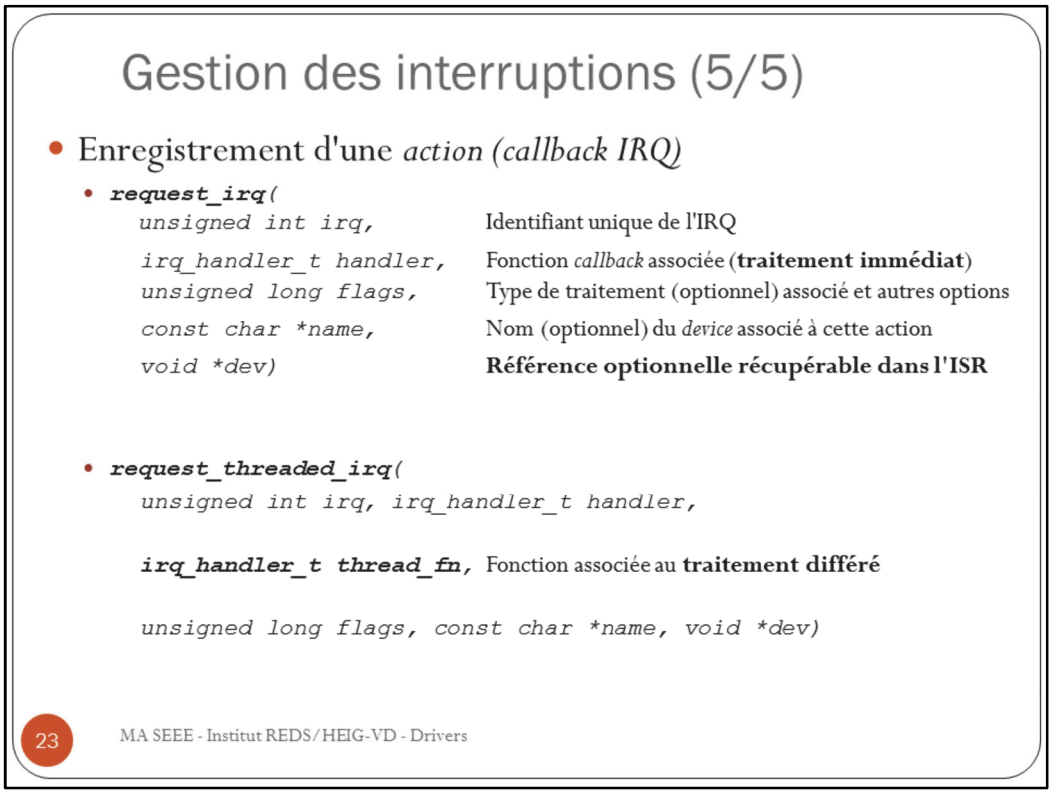
- Hiérarchie de traitements
  - **Traitement immédiat**
  - *Top half*
  - Traitement au niveau PIC
  - Traitement selon type IRQ
  - Action-s spécifique(s)
  - **Traitement différé**
  - *Bottom half*





Le traitement immédiat sous *Linux* est découpé en plusieurs fonctions correspondant aux différents niveaux de traitement décrits précédemment. L'entrée de l'ISR correspond au point d'entrée *vector_irq* dans le fichier *arch/arm/kernel/entry-armv.S*. La gestion des piles systèmes et du mode d'exécution est effectuée à cet endroit. Cette fonction réalise également le premier décodage de l'IRQ en interrogeant le contrôleur d'interruption et en associant un **numéro unique** correspondant à la source de l'interruption (attention! Pour le contrôleur GPIO, il n'y aurait ici qu'un numéro associé correspondant à la ligne d'interruption associée. Un numéro unique d'IRQ associé à chaque GPIO est calculé par des fonctions des niveaux supérieurs).
Puis la fonction de traitement est exécutée. Il existe principalement trois fonctions de traitement de base: ***handle_simple_irq()***, ***handle_level_irq()*** et ***handle_edge_irq()***. Ces trois fonctions implémentent des comportements différents au niveau de l'acquittement et du masquage de l'interruption. La première laisse l'acquittement et le démasquage au niveau de l'action. La seconde effectue un acquittement et démasquage, la troisième un acquittement et préserve le masquage. Le comportement dépendra donc du type d'interruption selon qu'il s'agisse d'une interruption à niveau, à flanc (montant/descendant) ou de type pulse. D'autres comportements peuvent être implémentés par le développeur.

Les fonctions de traitement appellent ensuite la fonction *callback* correspondant à l'action. C'est à ce niveau le plus souvent que le *driver* doit implémenter l'action correspondante au périphérique qui a déclenché l'interruption.



La fonction ***request_irq()*** (implémentée dans *kernel/irq/manage.c*) est très utilisée dans le noyau *Linux*, en particulier par les *drivers*. Elle permet d'associer une fonction C comme l'action d'une ISR liée à une interruption. La fonction sera appelée selon le schéma décrit précédemment et exécuté en tant que *traitement immédiat* (c-à-d qu'elle fait partie intégrante de l'ISR).

Un traitement différé peut être démarré soit grâce à une fonction *threadée* passée en argument dans la fonction ***request_threaded_irq()***; ce n'est pas la seule manière de différer le traitement, comme déjà évoqué. ***Tasklets*** et ***workqueues*** sont deux mécanismes bien connus qui permettent d'associer une fonction à l'interruption qui sera exécutée de manière différée. Nous ne détaillerons pas ces mécanismes dans le cadre de ce cours.

L'argument *irq* correspond à l'identifiant unique de l'interruption (identifiant retourné par le traitement de premier niveau ou par une fonction ***gpio_to_irq()*** par exemple si l'interruption provient d'un périphérique connecté au contrôleur GPIO.

L'argument *flags* permet de stipuler un certain type de traitement (si ce n'est pas celui par défaut pour cette interruption), à savoir *IRQF_TRIGGER_RISING, IRQF_TRIGGER_FALLING, IRQF_TRIGGER_HIGH, IRQF_TRIGGER_LOW, etc.* Il est également possible de donner d'autres options (voir *include/linux/interrupt.h*).

Références

- Linux Device Drivers (3rd Edition), Jonathan Corbet, Alessandro Rubini & Greg Kroah-Hartman
  - **Version online**: http://lwn.net/images/pdf/LDD3
- GCC, Declaring Attributes of Functions, http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html
- Linux Assembly HOWTO, Calling conventions, http://asm.sourceforge.net/howto/conventions.html