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)
Introduction

Prof. Daniel Rossier

Version 1.5 (2017-2018)

Contact & infos diverses

- Contact
  - daniel.rossier@heig-vd.ch
  - **skype**: rossierd
- **Institut REDS**
  - *Reconfigurable Embedded Digital Systems*
  - http://www.reds.ch
- Bureau **A21**
- ***http://www.linkedin.com***

Cours POS - Compétences-métier

- Etre capable de porter un OS (comme *Linux*) sur différentes plates-formes matérielles

- Cette unité d'enseignement est essentiellement orientée sur la pratique (projet/labo)

Déroulement du cours POS (1/2)

- Matériel du cours et laboratoires
  - **Moodle** : espace de cours **17_HEIG-VD_POS-A**

- Note cours (60 %)
  - 2 travaux écrits (60 %)
  - 1 présentation orale du projet + démo (40 %)
- Note labo (40 %) - Projet
  - Note intermédiaire #1 - Rapport intermédiaire + démo
  - Note intermédiaire #2 - Rapport intermédiaire + démo
  - Note finale - Rapport

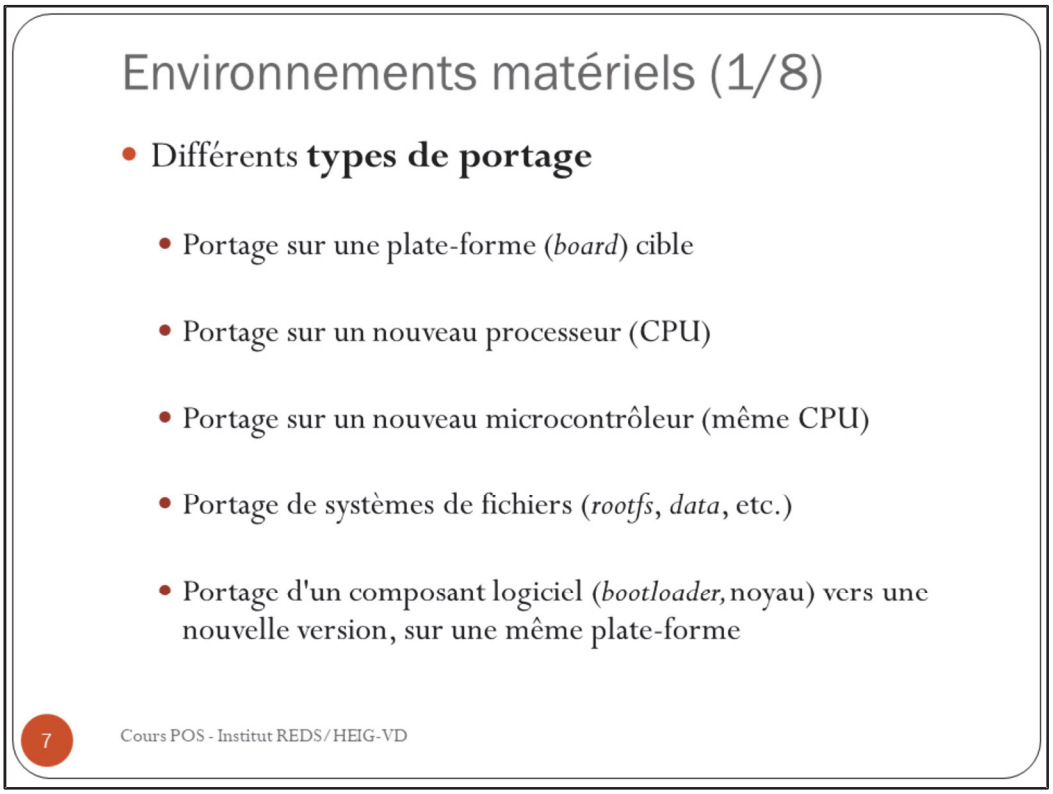
- Pas d'examen final !

Déroulement du cours POS (2/2)

- Introduction
- Accès au matériel
- Code d'amorçage
- Evénements asynchrones
- Accès mémoire
- Portage de composants divers

Plan (Introduction)

- Environnements matériels
- Environnements logiciels
- Portage de moniteur / *bootloader*
- Portage de systèmes d'exploitation (OS)

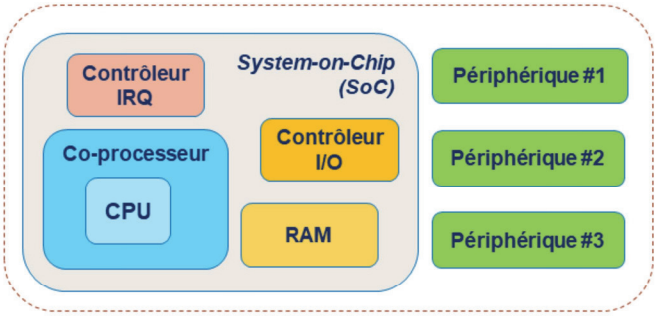


Le portage d'un *OS* sur une cible matérielle nécessite une bonne compréhension de l'architecture système de celle-ci. En particulier, le microcontrôleur joue un rôle crucial puisqu'il indique la nature du jeu d'instructions du binaire exécuté. Au-delà des instructions, c'est l'ensemble des contrôleurs *I/O* (dont celui de la mémoire) qui sont implantés d'une certaine manière en fonction du microcontrôleur. Le plan mémoire physique, la technologie des contrôleurs de base (mémoire, *timers, IRQs,* etc.), le code de démarrage interne lié à une *ROM* éventuelle sont des éléments fondamentaux intervenant lors du démarrage d'un *bootloader* ou d'un *OS.* Un des premiers objectifs à atteindre lors d'un portage d'*OS* consistera à obtenir un "signe de vie" de l'OS, montrant que du code *souhaité* s'exécute bien aux bonnes adresses.

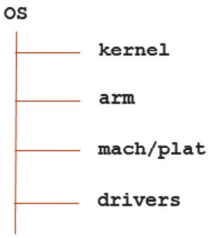
Autour du microcontrôleur, la connectique qui lie celui-ci aux périphériques externes de la plate-forme impacte sur la configuration des *GPIOs*. C'est un point essentiel à un portage d'OS; les *drivers* de périphériques reposent souvent sur plusieurs sous-systèmes de l'OS qui prennent en charge le traitement lié aux accès à ces périphériques via les *GPIOs* (par exemple, l'accès en lecture/écriture à des registres, le traitement des interruptions, etc.).

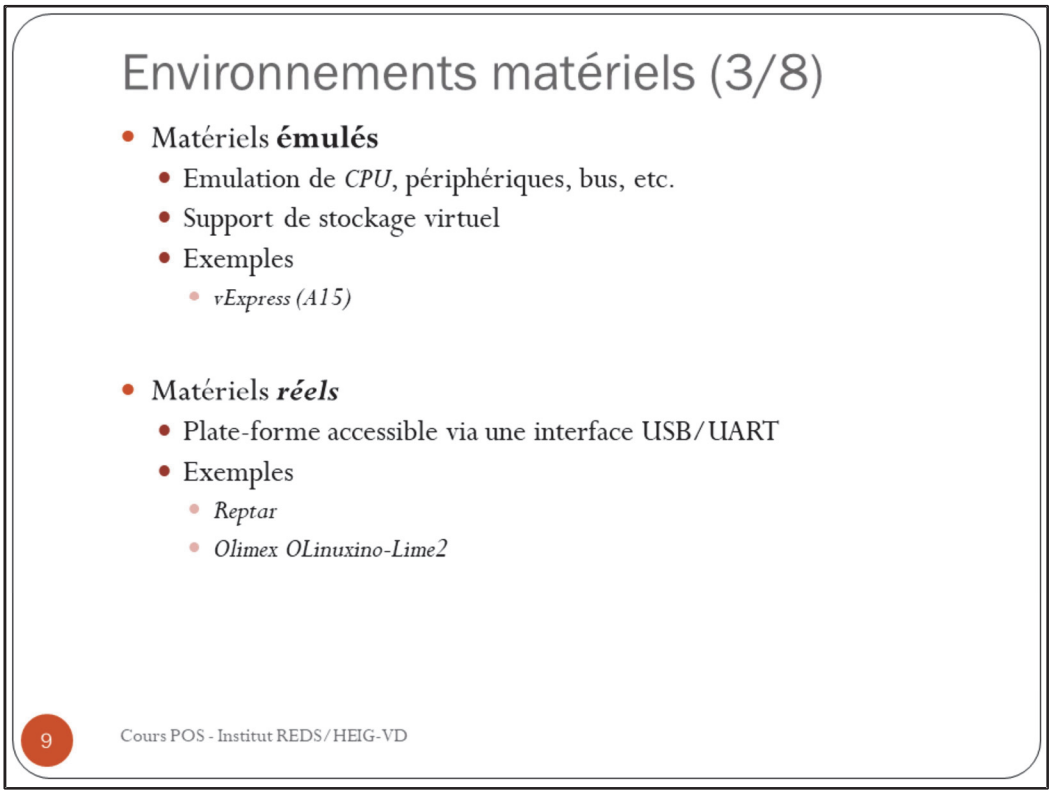
Finalement, le portage d'un système de fichiers ou d'autres composants logiciels nécessite un support de stockage tel qu'une *flash, sdcard* ou disques externes, ou encore un accès réseau. L'accès au système de fichiers racine (rootfs) doit donc être adapté en fonction de la *board*.

Environnements matériels (2/8)



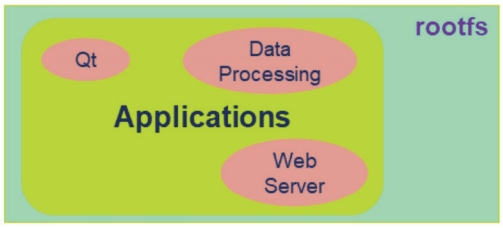
- Code générique
- Code dépendant CPU
- Code dépendant Plate-forme
- Code dépendant Périphérique



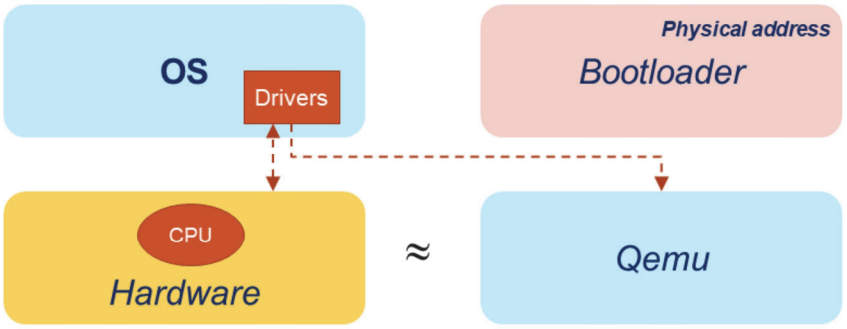


Le portage d'un OS complet sur une nouvelle *board* peut s'avérer relativement complexe. En particulier, la mise au point d'une séquence de démarrage (*bootstrap*) peut s'avérer difficile à effectuer due à l'absence d'une console; en effet, l'initialisation d'une interface série nécessite que le code à exécuter soit déjà en place en mémoire, au bon endroit, et le passage à un mode d'adressage virtuel (avec *MMU*) implique que les adresses *I/O* de cette interface soit correctement mappées.
C'est pourquoi, l'utilisation d'un émulateur de matériel comme *Qemu, Bochs,* etc. est vivement recommandé pour la validation de la séquence de démarrage, et peut être d'une très grande aide pour la suite du portage et du développement en général.

Environnements matériels (4/8)



- Environnement matériels/logiciels
- Composants

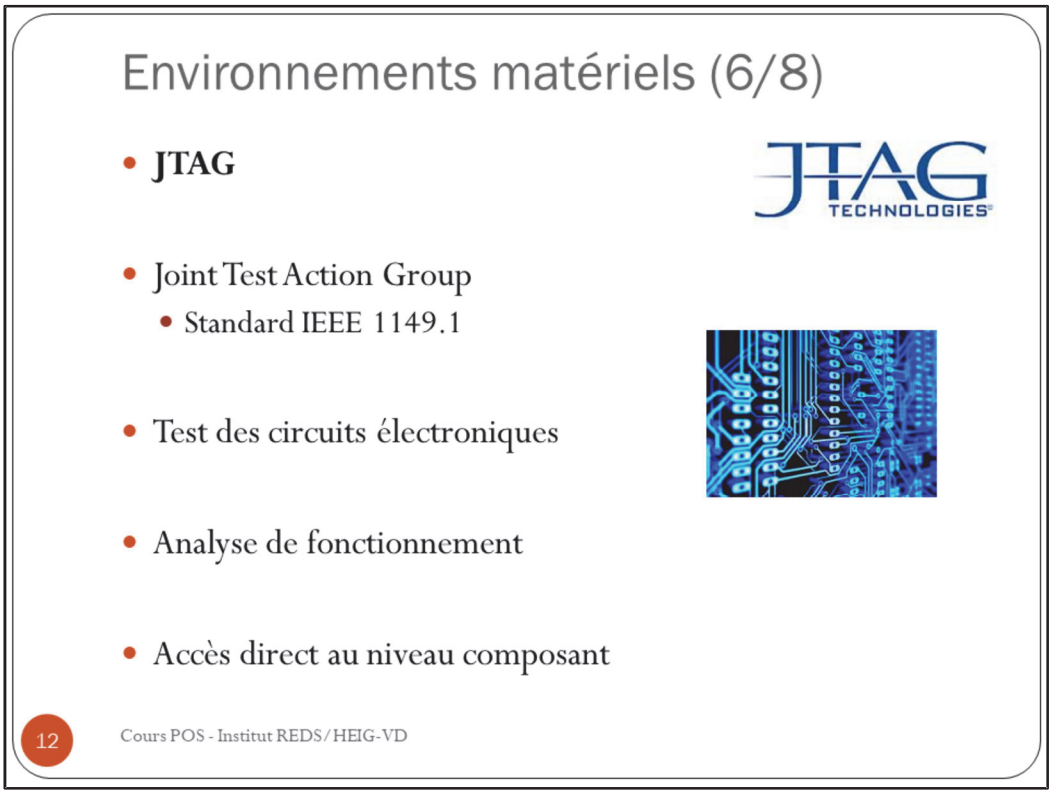


Environnements matériels (5/8)

- *Reptar*
  - *Cortex-A8*, *board* de l'institut *REDS* – *ARMv7*
  - http://reds.heig-vd.ch/formations/bachelor/pos/cours-exercices-et-laboratoires
  - Carte *REDS* et support dans *qemu*

- *vExpress A15*
  - https://wiki.linaro.org/Boards/Vexpress
  - *ARM Cortex-A15* – *ARMv7*
  - Support dans *qemu*

- *OLinuxino-Lime2*
  - *Allwinner A20 dual core Cortex-A7 processor* – *ARMv7*

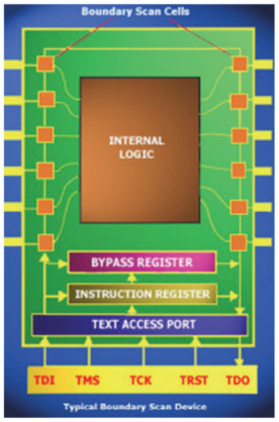


Le système *JTAG* (*Joint Test Action Group*) a été créé au départ pour tester des circuits électroniques pour l'industrie des cartes PC. Le JTAG correspond aussi à un standard IEEE (1149.1) définissant les types de signaux intervenant dans ce type de communication, ainsi qu'aux protocoles utilisés. Le terme *JTAG* s'apparente également au terme *Boundary Scan* (scrutation des frontières).

Le principe du *JTAG* réside dans l'utilisation de cellules connectées sur les *pins* du processeur/microcontrôleur, ou du FPGA ou autre circuit numérique. Les cellules permettent de piloter les entrées-sorties (I/Os) indépendamment de leurs fonctions initiales. Ce mécanisme permet également de piloter l'accès aux mémoires (RAM, *flash*, *sdcard*, etc.) et de pouvoir écrire et lire du contenu. De plus, le *JTAG* est également associé à des mécanismes plus élaborés permettant d'accéder directement le processeur (*registres, mémoires caches, points d'arrêt, etc.*), ce qui donne d'importants moyens pour la mise au point de programmes, sans perturber l'exécution en cours.

Environnements matériels (7/8)

- Le JTAG possède **3 entités.**
  - Le contrôleur TAP
  - Un registre à décalage d'instructions
  - Un registre à décalage de données
- Le contrôleur TAP possède **4 broches** (+ 1 optionnelle).
  - ***TCK***: horloge (*Test ClocK*)
  - ***TMS***: mode test de contrôle TAP (*Test Mode Select*)
  - ***TDI***: entrée donnée série (*Test Data In*)
  - ***TDO***: sortie donnée série (*Test Data out*)
  - (***TRST***: remise à zéro du *TAP*)



13 Cours POS - Institut REDS/HEIG-VD

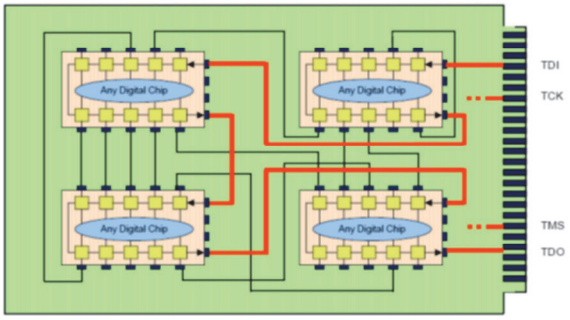
La norme *JTAG* définit clairement les signaux intervenant dans la connexion inter-composants et la communication entre le contrôleur associé – appelé *TAP* (*Test Access Port*) et les cellules. Le contrôleur implémente également une série d'instructions permettant l'automatisation de tests électroniques, ainsi que le pilotage des accès sur les composants.

On remarque que le contrôleur dispose de son propre signal d'horloge; celui-ci doit être cadencé par une source externe au composant. De ce fait, le contrôleur peut évoluer indépendamment au système, sans interférer sur le fonctionnement en cours. C'est pourquoi cette technique est particulièrement appréciée lors de la mise au point d'applications temps-réel strict par exemple, où les contraintes temporelles peuvent être sévères.

Mais cette architecture met également en évidence qu'une connexion JTAG nécessite l'utilisation d'une interface *cliente* particulière, tant matérielle que logicielle. C'est pourquoi une communication *JTAG* depuis un PC nécessite généralement une **sonde** *(matérielle) JTAG* pouvant être reliée sur un port parallèle ou *USB* du PC, ainsi qu'un logiciel permettant le pilotage des signaux et commandes envoyés au contrôleur TAP.

Environnements matériels (8/8)

- Boucle série
  - Un point d'entrée, un point de sortie
- Boucle active (écriture)
- Boucle passive (lecture)
- *Boundary Scan*



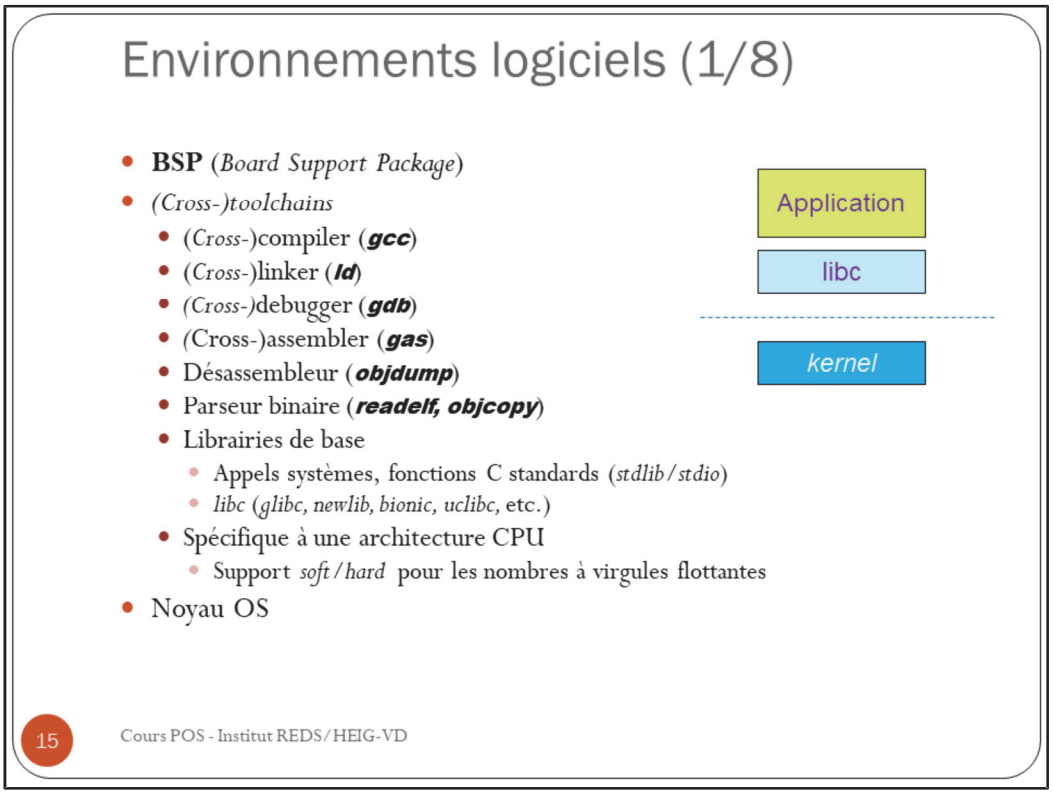
- Signal d'horloge indépendant

Nous examinons ici les détails du fonctionnement *JTAG* à l'intérieur d'un composant.

Chaque bit du registre à décalage de données du JTAG correspond à la valeur instantanée d'une broche de circuit. On peut lire sa valeur courante ou lui affecter une nouvelle valeur (écriture). Le contrôleur *TAP* permettra la configuration de chaque cellule, et permettra la synchronisation des accès.

Le JTAG forme une boucle série interconnectant les broches d'entrée-sortie des circuits à surveiller/analyser. Le transfert s'effectue en série et la boucle peut être active (écriture) et/ou passive (lecture). On récupère ainsi un flux de données série plus ou moins important. On va donc balayer périodiquement les composants JTAG.

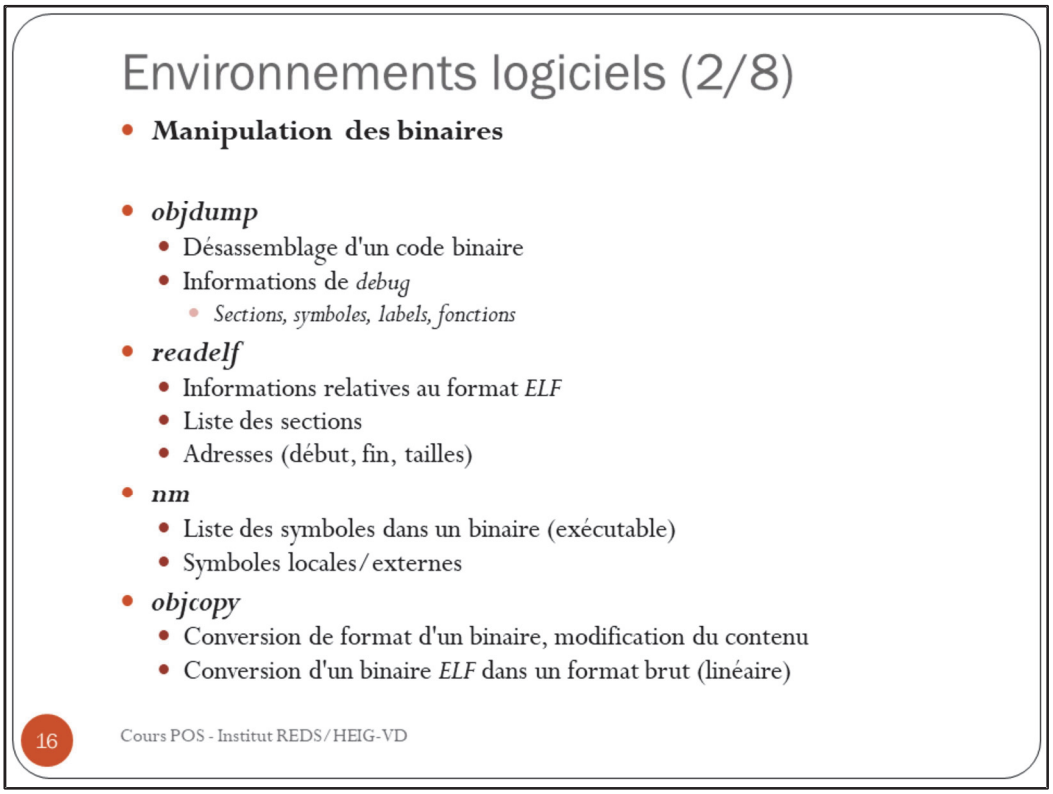
Le contrôleur *TAP* comprend une machine à 16 états avec 10 instructions normalisées pour le test des circuits. Cela comprend le test des connexions inter-circuits et des connexions internes au circuit.



Le *BSP* contient l'essentiel des composants logiciels permettant le développement, le déploiement et l'exécution d'applications embarquées.

La *toolchain* représente l'ensemble des applications tournant sur la machine hôte permettant la génération d'un binaire exécutable sur la cible. Par conséquent, elle dépend fortement du jeu d'instructions de cette dernière et des spécificités du *CPU* cible; par exemple, la présence ou non d'un coprocesseur arithmétique à virgule flottante nécessite d'utiliser une *toolchain* appropriée afin de pouvoir tenir compte des instructions complexes supportées par le coprocesseur.

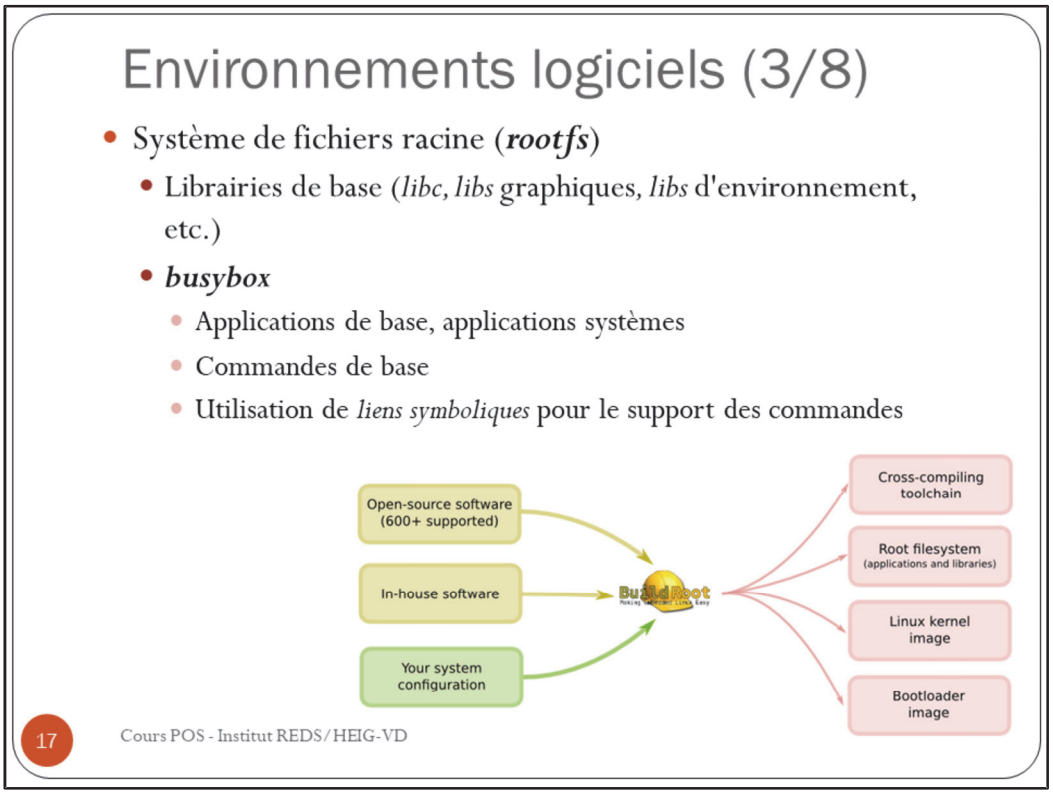
De plus, les librairies de base tel que la ***libc*** permettent de *linker* du code de bas niveau (statiquement ou dynamiquement) contenant les fonctions standards du langage C ainsi que le *code d'entrée (stub)* des appels systèmes. La *toolchain* peut être lié de ce fait au système d'exploitation sur lequel tournera l'exécutable.



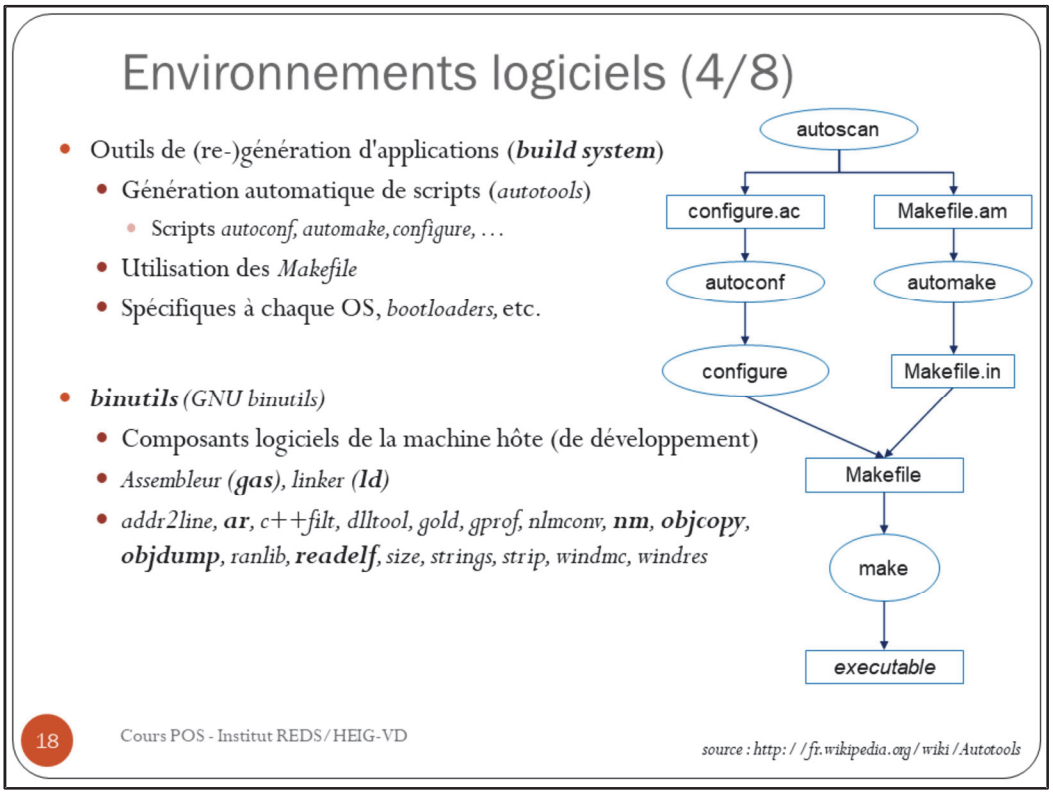
Le compilateur produit un code objet (.o) dans un format bien défini; sous *Linux*, c'est le format *ELF* qui est largement répandu. Il en va de même pour l'exécutable (image binaire).

Différents outils permettent d'inspecter et de transformer un binaire.

- *objcopy* est utilisé pour "aplatir" un fichier *ELF*, c-à-d qu'un nouveau fichier avec un contenu totalement identique à ce qui sera implanté en mémoire lors du chargement, en considérant que le fichier sera simplement copié (au sens d'un *memcpy*) à l'adresse correspondant au début de l'image.
  Il faut faire attention avec la section *BSS.* En effet, l'utilitaire *objcopy* ne va pas allouer *par défaut* les *bytes* nécessaires à cette section. Il existe cependant des options permettant de prendre en compte cet espace (cf laboratoires).



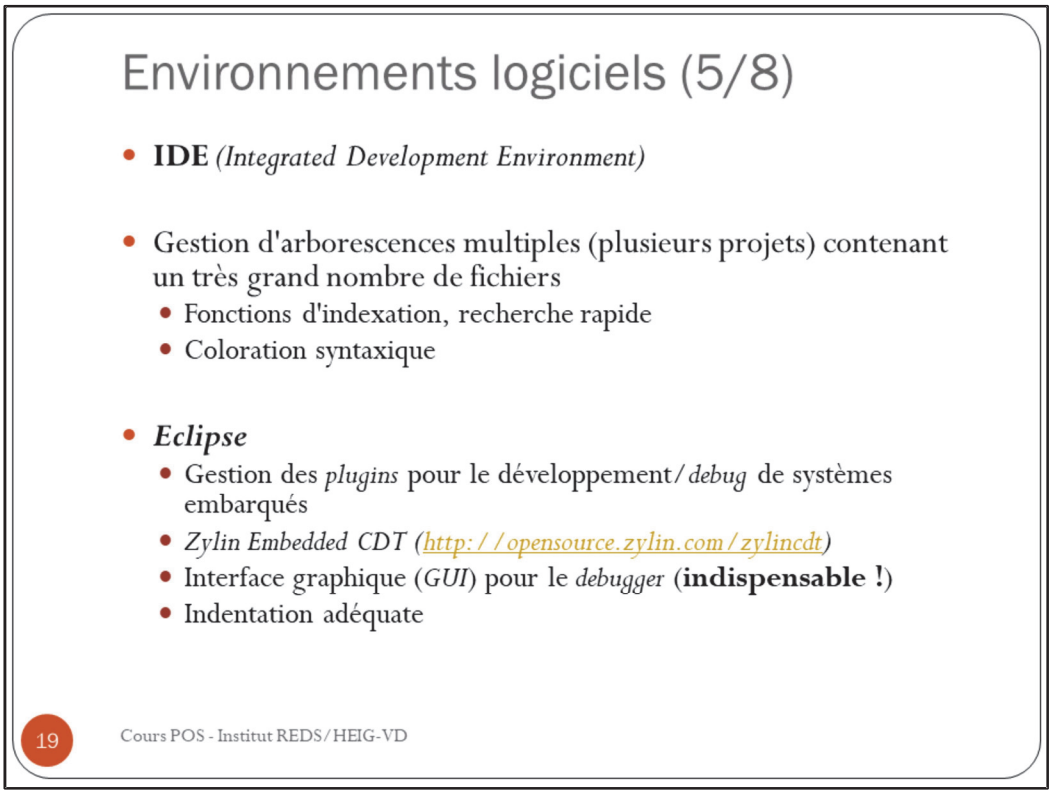
Un (ou plusieurs) OS peut aussi faire partie du *BSP*. En particulier, le système de fichiers racine (*rootfs*) doit être présent et contenir des applications de base. Une application très courante dans les systèmes embarqués et *busybox* qui implémente la plupart des utilitaires d'un environnement *Linux* classique *(ls, more, rm, cat, mount,* etc.*),* mais en version simplifiée. Pour chaque utilitaire, un lien symbolique existe et redirige l'exécution sur *busybox*.



Les *binutils* constituent un ensemble minimal d'applications permettant la manipulation de binaires exécutables (compilateur d'assemblage, *linker*, *parser* binaire, liste des noms symboliques, etc.). Typiquement, ces ensemble est utilisé pour la compilation d'un compilateur, et donc d'une *toolchain.*

La génération de binaires exécutables de très gros projets comme un OS nécessite un ensemble d'utilitaires pour la génération des *Makefiles*, à partir d'une analyse de l'environnement courant. Ces utilitaires font partie des *autotools*.

- *autoscan* parcours les répertoires à la recherche de fichiers sources et génère un fichier *configure.scan* qui pourra être renommé en *configure.ac*
- *autoconf* système pour décrire la structure de l'application, en permettant au développeur d'écrire un *Makefile* simplifié. Cet outil génère un fichier 'moule' du Makefile à partir d'un série de macros.
- *automake* lit les fichiers *Makefile.am* et génère les fichiers *Makefile.in*. outil qui fournit une structure de portabilité, basée sur un ensemble de tests des spécificités de la machine au moment de l'installation.



La manipulation de gros projets nécessite l'utilisation d'un *IDE* afin de pouvoir naviguer aisément dans des centaines de milliers de lignes de code.

Eclipse est largement répandu dans le développement de systèmes embarqués; il est performant et comporte une interface graphique pour *gdb* (disponible avec le *plugin* de *Zylin Embedded CDT*) très puissante.

On veillera à respecter une indentation classique de type *K&R* (*Kernighan & Ritchie*) pour le code C bas niveau. Dans *eclipse*, il est possible de formater automatiquement le code source suivant un style pré-défini.

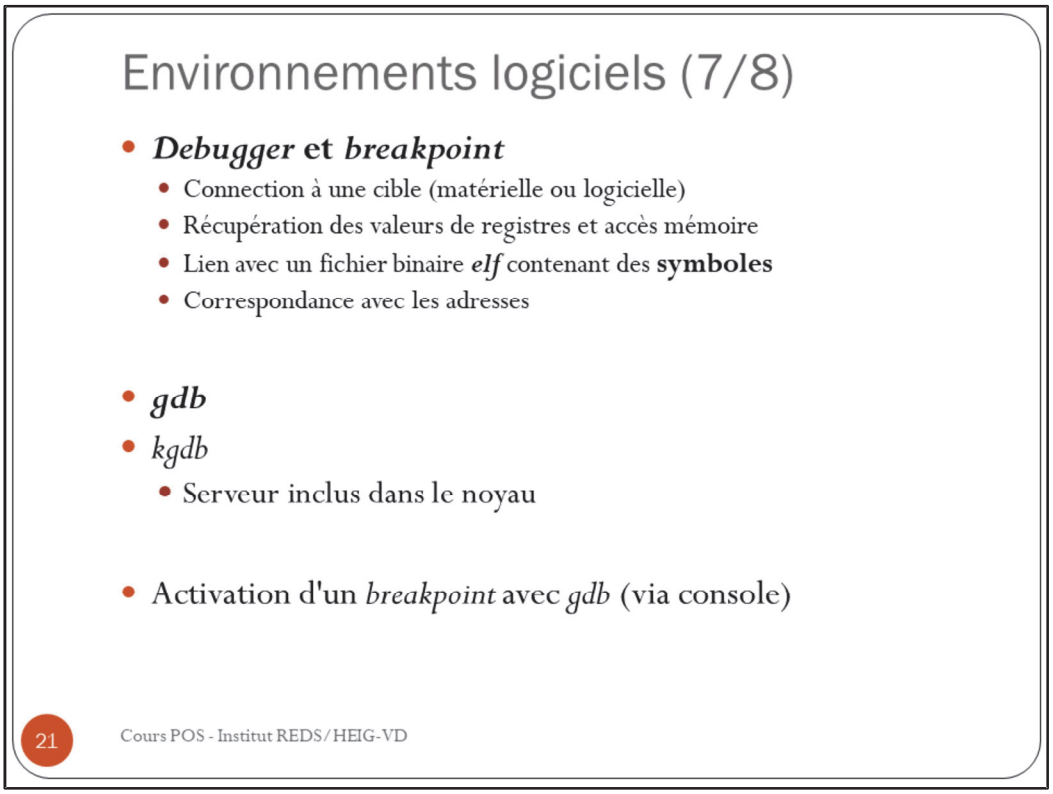
Environnements logiciels (6/8)

- **Documentation**

- Schéma de la *board, datasheet*
  - Schéma de connexions GPIO, *chip select*, etc.

- Manuel de référence du microcontrôleur (*SoC*)

- Documents propriétaires décrivant le *hardware* et ses interfaces
  - *FPGA register layout*
  - Adressage *FPGA*
  - etc.



Lorsqu'un *debugger* tel que *gdb* se connecte à un serveur *gdb* (celui-ci peut être une application tournant dans l'espace utilisateur, ou embarqué dans le noyau ou dans un émulateur comme *qemu*), il permet de "visualiser" le contenu de la mémoire en temps-réel. A l'aide de *breakpoint*, on peut donc examiner le contenu des registres du *CPU* à n'importe quel moment, effectuer une exécution pas-à-pas et examiner les instructions désassemblées en cours d'exécution.

Il faut bien comprendre que le *debugger* désassemble les instructions qu'il trouve dans la mémoire aux adresses contenus dans le registre *pc*. Si la compilation du code source a été faite de telle manière à retrouver les symboles dans le fichier binaire (option *–g*), le *debugger* fera un lien avec le code source correspondant et montrera l'exécution courante dans celui-ci. Le format du binaire exécutable contenant les symboles est classiquement dans le format *ELF.*
Bien entendu, il se peut tout à fait qu'il y ait un décalage entre le code source et l'instruction en cours d'exécution.

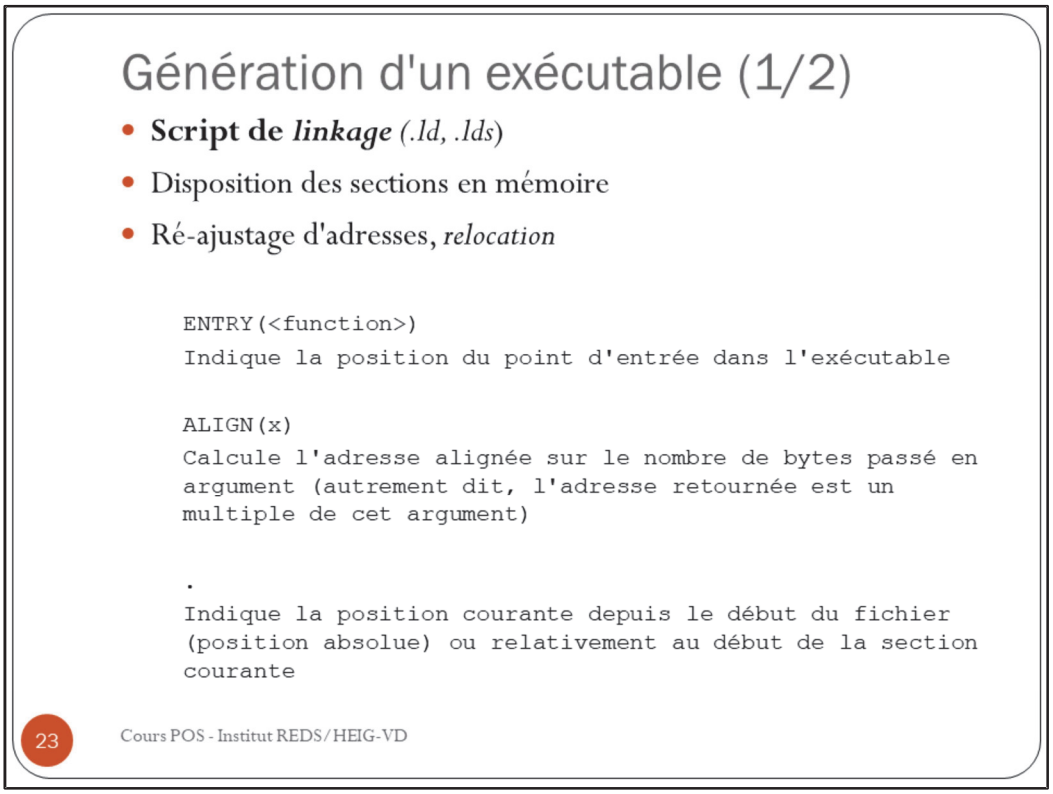
Lors du *debug* d'un code de démarrage (comme dans *U-boot* par exemple), l'emplacement des instructions/données peut changer durant l'exécution. Dans ce cas, il peut être nécessaire de recharger dans le *debugger* les symboles à la nouvelle adresse, à partir de l'exécutable, afin de pouvoir resynchroniser avec le code source.

Les adresses gérées par le serveur *gdb* sont soit virtuelles, soit physiques, selon que la *MMU* soit activée ou non.

Environnements logiciels (8/8)

- Correspondance du contenu mémoire avec les symboles
  - ***symbol-file***
  - ***add-symbol-file <elf_file> <address>***
    - ***add-symbol-file u-boot 0x420f1c00***

- ***Breakpoint***
  - ***Software*** *breakpoint*
    - Modification de l'instruction en mémoire
    - Génération d'une exception (*syscall*)
  - ***Hardware*** *breakpoint*
    - Registres spécialisés contenant les adresses des *breakpoints*
    - Comparaison constante entre le *PC* et ces registres
    - Modification de l'instruction (bit de *breakpoint*)



Le *linker* produit une image binaire exécutable contenant les différentes sections de code et de données contenus dans les fichiers objets qui lui sont indiqués. Si rien n'est spécifié explicitement à ce moment, les sections apparaissent dans un ordre dicté par l'emplacement des codes objets passés au *linker*. Dans le cas d'une application, cela ne pose généralement aucun problème puisque le *linker* réajustera les adresses de saut et de référence selon l'emplacement des fonctions et des données référencées.

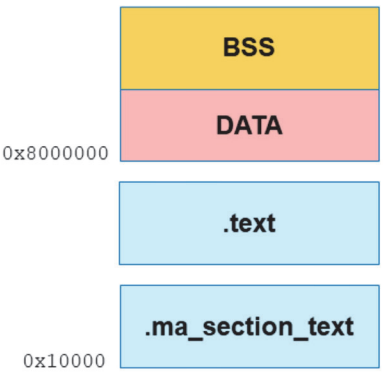
En revanche, il peut arriver que l'on définisse des sections réservées à un usage précis et que celles-ci doivent être placées à un endroit prédéfini; c'est le cas par exemple des fonctions d'initialisation d'un OS. Une fois exécutées, celles-ci peuvent être effacées de la mémoire afin de récupérer de la place. Si toutes ces fonctions sont regroupées dans une section particulière, il est aisée d'effacer la zone mémoire associée.

Le code de démarrage d'un noyau ou *bootloader* doit également se situer à une adresse précise; c'est pourquoi, il convient d'indiquer au *linker* où la fonction doit se trouver (la fonction sera dans une section réservée à cet effet).

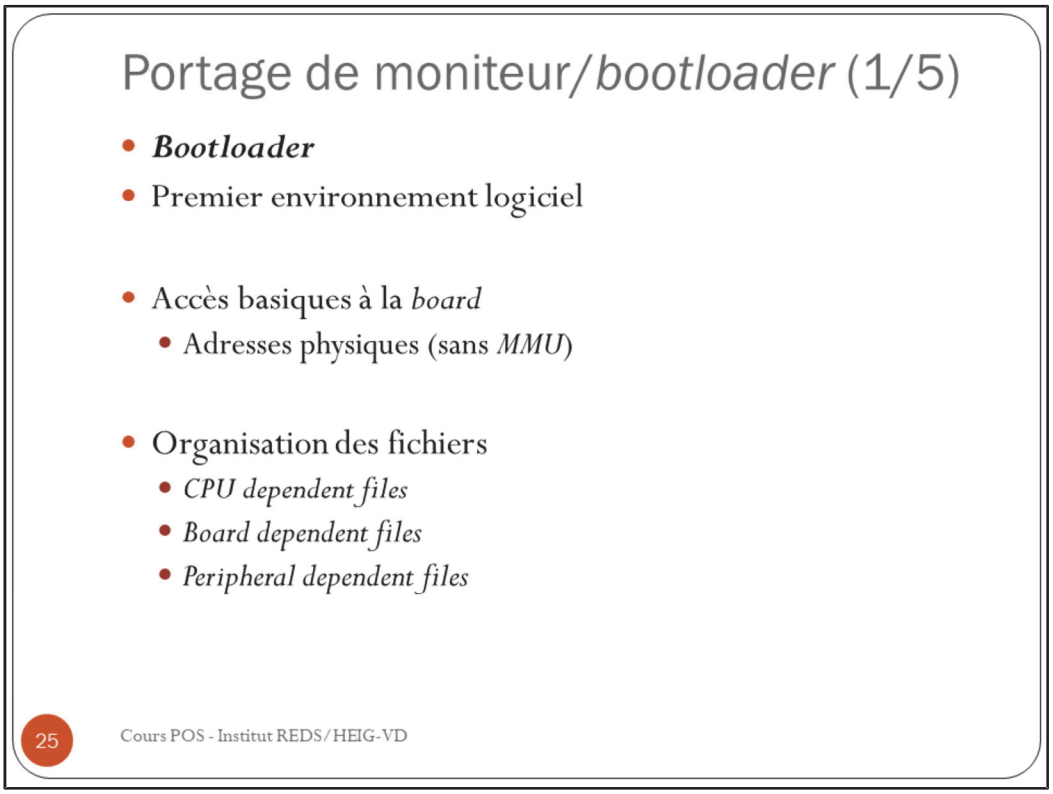
Génération d'un exécutable (2/2)

```
3   SECTIONS
4   {
5       . = 0x100
6       .text: {
7         *(.text)
8       }
9       . = 0x500
10      .data: {
11        *(.data)
12        . += 0x600
13      }
14  }
15
```

```
3  ENTRY(start)
4
5  SECTIONS
6  {
7   . = 0x10000;
8   .ma_section.text :
9   {
10   *(.ma_section.text);
11  }
12  . = ALIGN(4);
13  .text : { *(.text) }
14
15  . = 0x8000000;
16  .data : { *(.data) }
17  .bss : { *(.bss) }
18 }
19
```

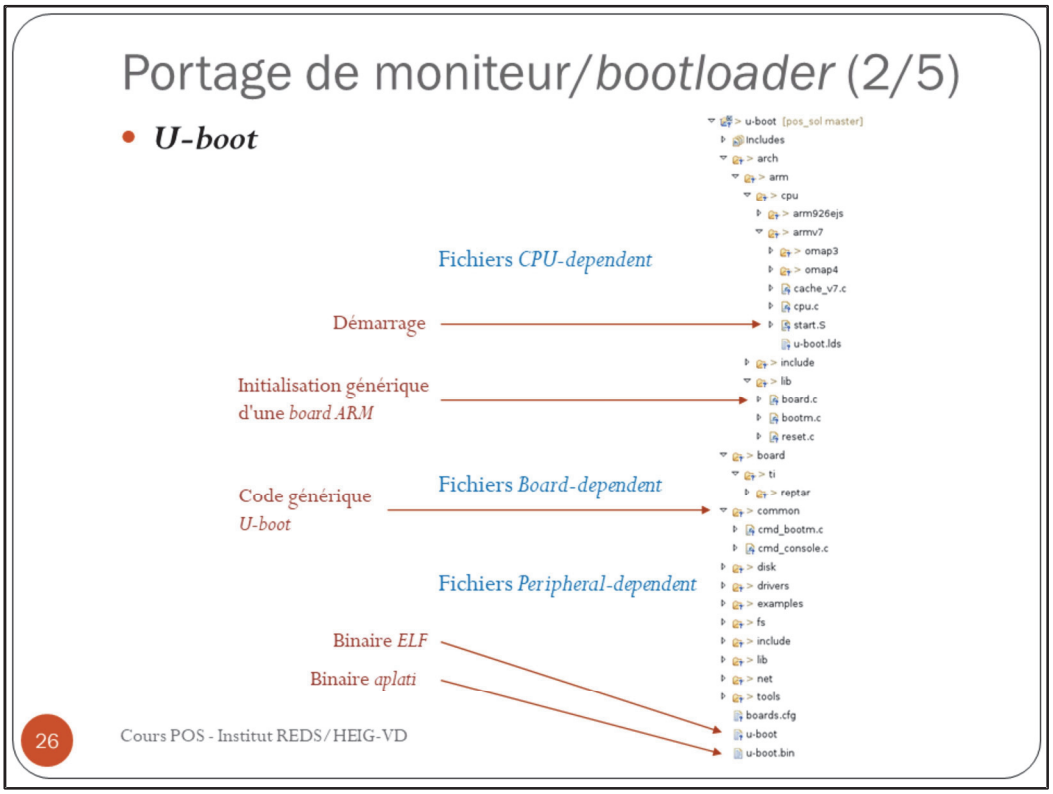



```
arm-linux-gnueabihf-gcc -O0 -g -c asmA1.S
arm-linux-gnueabihf-ld -T asm.lds asmA1.o -o asmA1.elf
arm-linux-gnueabihf-objcopy -O binary --set-section-flags .bss=alloc,load,contents asmA1.elf asmA1.bin
```



Le moniteur effectue une première initialisation des périphériques de base (contrôleur mémoire, *UART*, horloge système, etc.) afin d'offrir à l'utilisateur un environnement minimal lui permettant d'interagir avec la plate-forme.

Souvent, les OS démarrés depuis le moniteur bénéficient de cette configuration initiale et n'effectuent pas une réinitialisation de ces composants. Mais il peut arriver aussi qu'un OS (comme *Linux* par exemple) effectue une initialisation complète de la plate-forme et réinitialise les composants de base.

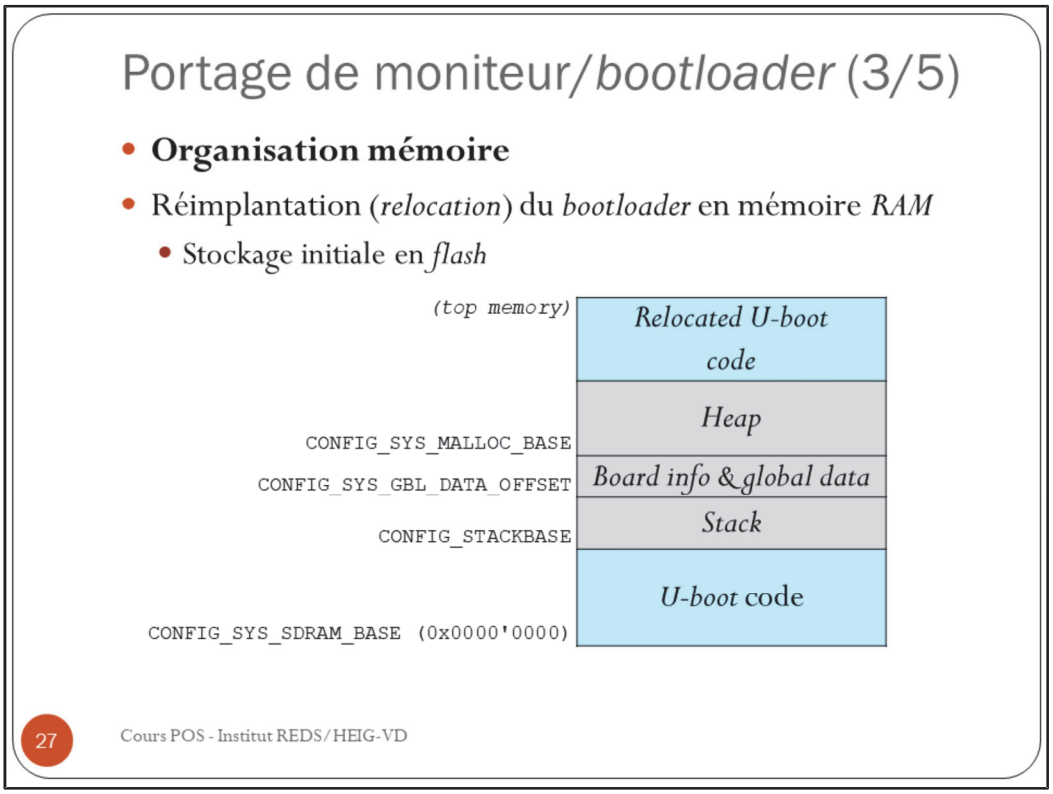


Le portage de *U-boot* sur une nouvelle plate-forme consiste à créer les répertoires et fichiers spécifiques à la *board* en se basant sur une *board* existante.

- Le répertoire *include/configs* contient l'ensemble des constantes liées à la *board*.
- Le répertoire *board/* contient l'ensemble des fichiers correspondant à la *board.*

Les fichiers suivants sont importants (exemple avec la *board REPTAR*) :

- *board/ti/reptar/**reptar.c***: Initialisation de bas niveau – fonction *board_init(void)*
- *board/ti/reptar/**config.mk***: Constante *CONFIG_SYS_TEXT_BASE* pour adresse de base
- *arch/arm/cpu/armv7/omap3/**lowlevel_init.S**:* Initialisation *RAM* + horloge
- *arch/arm/cpu/armv7/**start.S**:* Code de démarrage
- *arch/arm/cpu/armv7/**uboot.lds**:* Script de *linkage*



Le *bootloader* peut se trouver dans une mémoire de stockage de type *flash*. Son exécution peut débuter en *flash*, mais il est préférable de le reloger en mémoire *RAM* dès que possible afin d'accélérer son exécution et de préserver la *flash.*

Portage de moniteur/*bootloader* (4/5)

`arch/arm/include/asm/global_data.h`

```
typedef struct global_data {
    bd_t *bd;
    unsigned int    baudrate;
    unsigned long   cpu_clk;        /* CPU clock in Hz!    */
    unsigned long   bus_clk;
    unsigned long   pci_clk;
    unsigned long   mem_clk;
    #ifdef CONFIG_BOARD_TYPES
    unsigned long   board_type;
    #endif
    unsigned long   have_console;   /* serial_init() was called */
    unsigned long   env_addr;       /* Address  of Environment struct */
    unsigned long   env_valid;      /* Checksum of Environment valid? */
    /* TODO: is this the same as relocaddr, or something else? */
    unsigned long   dest_addr;      /* Post-relocation address of U-Boot */
    unsigned long   dest_addr_sp;
    unsigned long   ram_top;        /* Top address of RAM used by U-Boot */

    unsigned long   relocaddr;      /* Start address of U-Boot in RAM */
    phys_size_t     ram_size;       /* RAM size */
    unsigned long   mon_len;        /* monitor len */
    unsigned long   irq_sp;         /* irq stack pointer */
    unsigned long   start_addr_sp;  /* start_addr_stackpointer */
    unsigned long   reloc_off;      /* relocation offset */
    struct global_data * new_gd;    /* relocated global data */
    const void *    fdt_blob;       /* Our device tree, NULL if none */
    void *          new_fdt;        /* Relocated FDT */
    unsigned long   fdt_size;       /* Space reserved for relocated FDT */
    char            env_buf[32];    /* buffer for getenv() before reloc. */
    struct arch_global_data arch;   /* architecture-specific data */
} gd_t;
```

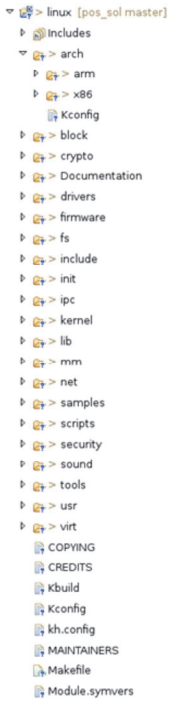
```
typedef struct bd_info {
    unsigned long   bi_memstart;    /* start of DRAM memory */
    phys_size_t     bi_memsize;     /* size  of DRAM memory in bytes */
    unsigned long   bi_flashstart;  /* start of FLASH memory */
    unsigned long   bi_flashsize;   /* size  of FLASH memory */
    unsigned long   bi_flashoffset; /* reserved area for startup monitor */
    unsigned long   bi_sramstart;   /* start of SRAM memory */
    unsigned long   bi_sramsize;    /* size  of SRAM memory */
#ifdef CONFIG_ARM
    unsigned long   bi_arm_freq;    /* arm frequency */
    unsigned long   bi_ddr_freq;    /* ddr frequency */
#endif
    unsigned long   bi_bootflags;   /* boot / reboot flag (Unused) */
    unsigned long   bi_ip_addr;     /* IP Address */
    unsigned char   bi_enetaddr[6]; /* OLD: see README.enetaddr */
    unsigned short  bi_ethspeed;    /* Ethernet speed in Mbps */
    unsigned long   bi_intfreq;     /* Internal Freq, in MHz */
    unsigned long   bi_busfreq;     /* Bus Freq, in MHz */
    unsigned int    bi_baudrate;    /* Console Baudrate */
    unsigned int    bi_procfreq;    /* CPU (Internal) Freq, in Hz */
    unsigned char   bi_pci_enetaddr[6]; /* PCI Ethernet MAC address */
#endif
    unsigned char   bi_enetXaddr[6];  /* OLD: see README.enetaddr */
    ulong           bi_arch_number;   /* unique id for this board */
    ulong           bi_boot_params;   /* where this board expects params */
#ifdef CONFIG_NR_DRAM_BANKS
    struct {        /* RAM configuration */
        ulong       start;
        ulong       size;
    } bi_dram[CONFIG_NR_DRAM_BANKS];
#endif /* CONFIG_NR_DRAM_BANKS */
} bd_t;
```

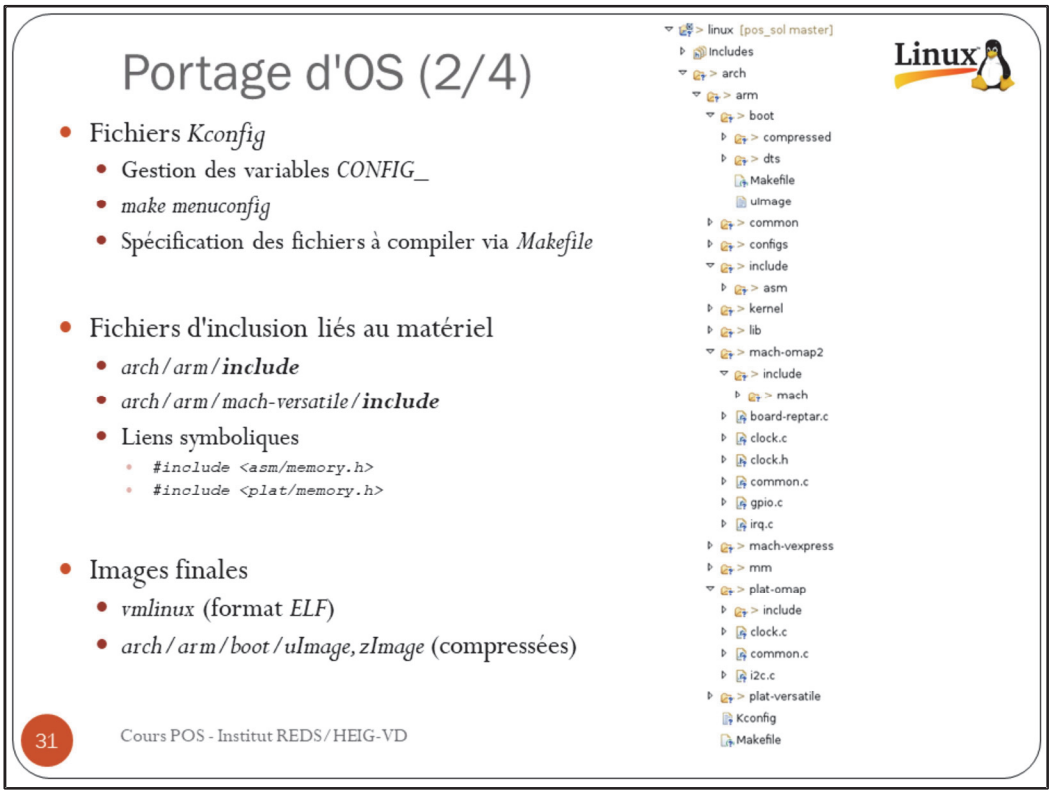
Portage de moniteur/*bootloader* (5/5)

- **Applications *standalone***
- Fonctions du moniteur
- Conventions *ABI*
  - Appels de fonctions C depuis assembleur
  - Nécessité d'avoir une pile pour les *prologues/épilogues*
- *Toolchain*
  - Fichier d'amorçage par défaul d'une application : `crt0.S`
  - Script de *linkage* par défaut (*.lds*)

Portage d'OS (1/4)

- **Portage de *Linux***
- Organisation des fichiers
  - *arch/* matériel
  - *drivers/* pilotes
  - *fs/* Systèmes de fichiers
  - *kernel/* Processus, *threads*, *printk()* etc.
  - *mm/* Gestionnaire mémoire





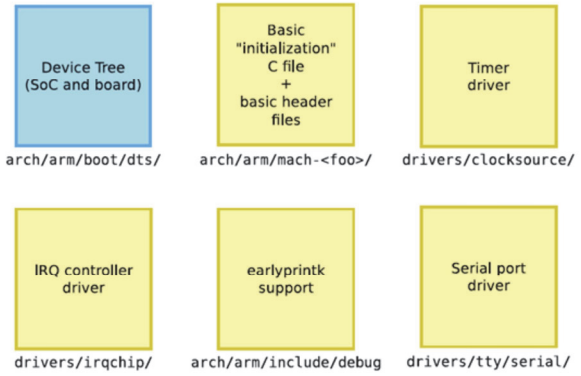
Les fichiers *Kconfig* contiennent la déclaration des variables de configuration (*CONFIG_**) et les dépendances entre elles. L'activation de l'une ou l'autre variable dépendra de la configuration du noyau spécifiée par l'utilisateur.

Les liens symboliques utilisés lors d'inclusion *(#include)* sont créés par les *Makefile* lors de la première compilation. Ils dépendront de la configuration matérielle choisie.

Le fichier *vmlinux* à la racine de l'arborescence est le résultat de la compilation du noyau; il s'agit d'une image binaire au format *ELF*. Les images résultantes stockées dans le répertoire *arch/arm/boot/* sont compressés et contiennent éventuellement une entête spéciale utilisée par le *bootloader* (par exemple *U-boot*) ainsi qu'un code d'initialisation et de décompression.

Portage d'OS (3/4)

- Sous-systèmes & éléments principaux



Portage d'OS (4/4)

- Activation d'une console *UART*
- Activation de l'option `CONFIG_DEBUG_LL`
- Fonctions disponibles dans `arch/arm/kernel/debug.S`
  - `printascii(), printch(), printhex8(), etc.`
- Sélection d'une *UART* dans `arch/arm/include/debug`
  - Mappage de l'*UART* sur des adresses virtuelles
  - Accès au contrôleur UART pour l'écriture

Références

- Marta Rybczynska, *Porting Linux to a New Architecture*, Embedded Linux Conference 2014.
  http://events.linuxfoundation.org/sites/events/files/slides/Rybczynska_Porting_Linux_to_a_new_architecture_ELC2014.pdf
- Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support.
  http://free-electrons.com
- Pierre Ficheux, *Linux embarqué*, 3e édition, Éditions Eyrolles (2010)