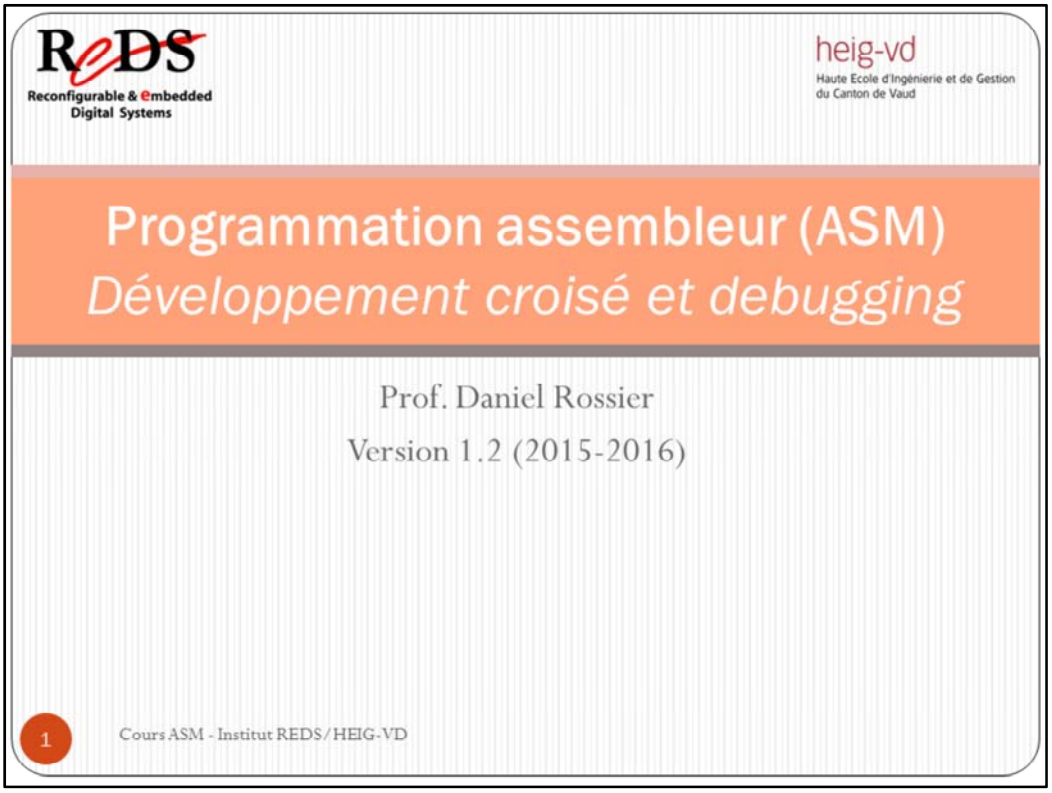
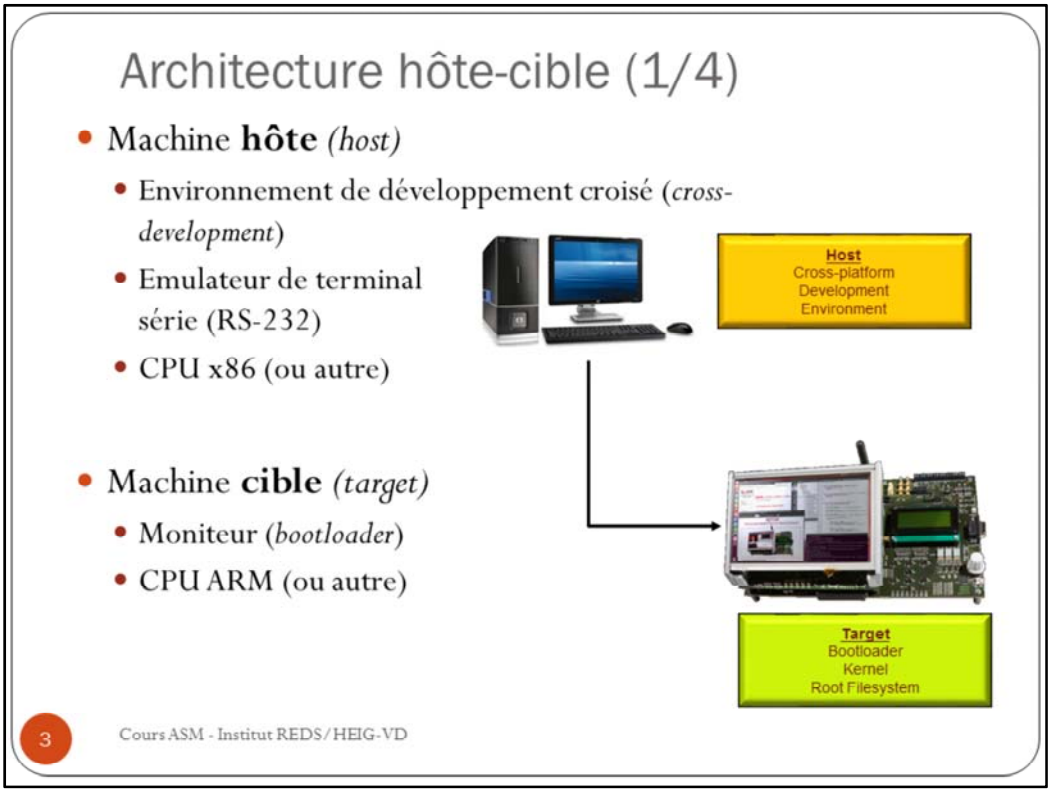

ReDS
Reconfigurable & embedded Digital Systems
heig-vd
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud
Programmation assembleur (ASM)
Développement croisé et debugging
Prof. Daniel Rossier
Version 1.2 (2015-2016)
1
Cours ASM - Institut REDS/HEIG-VD

Plan

- Architecture hôte/cible
- Chaînes de *cross-compilation*
- *Board Support Package (BSP)*
- *Makefile*

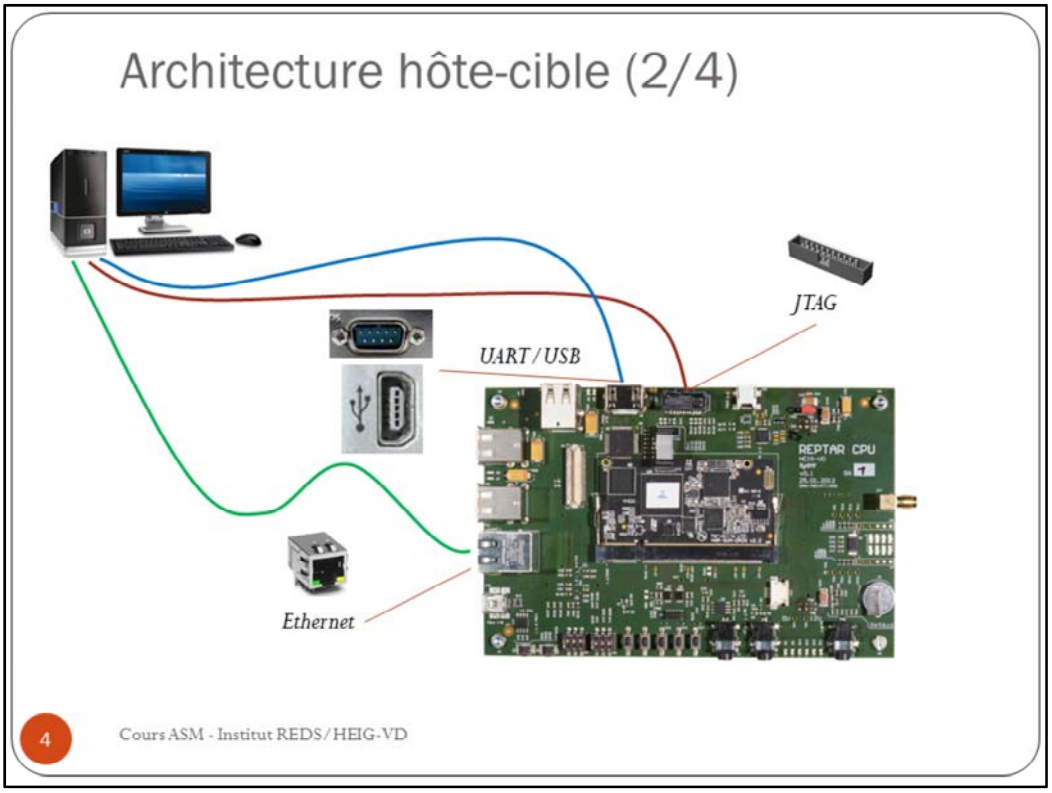


Aujourd'hui, il est de plus en plus fréquent de voir des applications tournées dans des environnements matériels très différents: un PC, un *smartphone*, une tablette PC, une *set-top-box*, des bornes interactives, etc.

Les environnements de développement ont eux aussi beaucoup évolués. Si les applications se destinent à tourner sur un système embarqué, il n'est cependant pas dit qu'elles seront développées et mises au point sur le système final. La plupart du temps, le développement s'effectuera sur un PC avec des outils confortables et performants, et l'application sera déployée sur le système cible pour la tester. C'est ce que l'on appelle du développement croisé ***(cross-development)***.

Un environnement matériel de *cross-development* se compose principalement d'une machine-hôte ***(host)*** *– ou hôte –* et d'une machine-cible ***(target)*** *– ou cible –.* Ces deux plates-formes peuvent bien entendu être équipées de processeurs différents.

La communication entre l'hôte et la cible peut prendre différentes formes: connexion sérielle *UART*, connexion réseau *TCP/IP*, connexion *JTAG*, etc.



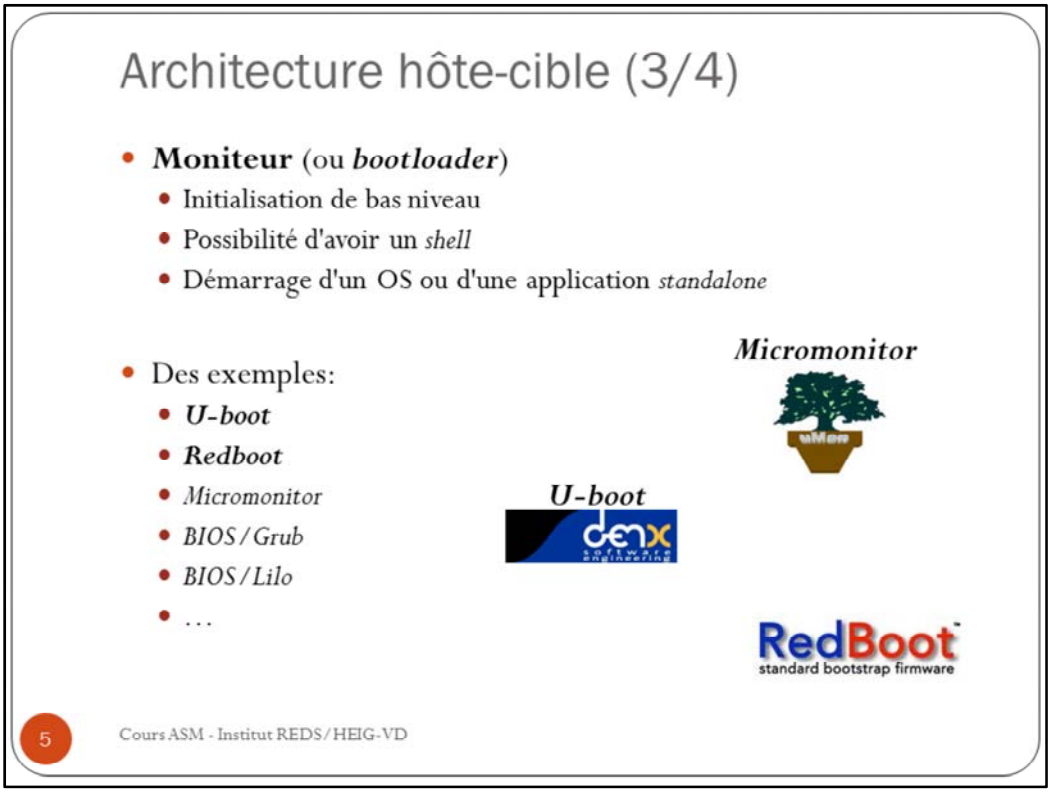
Différents niveaux de connexions/communications existent en fonction de l'équipement disponible sur la cible et du genre d'application que l'on souhaite déployer.

On fait la différence entre 3 types principaux de connexion.

- La connexion *JTAG* est une connexion de très bas niveau qui fait intervenir le-s composant-s électronique-s directement. Elle est requise afin de tester un équipement au niveau électronique et peut s'avérer nécessaire pour le déploiement des composants logiciels de base (moniteur, application *standalone*, etc.).
  Typiquement, une connexion *JTAG* est nécessaire lorsqu'aucun composant logiciel ne fonctionne et qu'un déploiement de base (d'usine) est nécessaire. Le système *JTAG* sera décrit plus loin en détail.

- La connexion sérielle de type *UART* est très utilisée. Simple et efficace, elle permet d'accéder une plate-forme avec un minimum de moyen. Le *driver UART* est léger et peut facilement être intégré dans un moniteur.

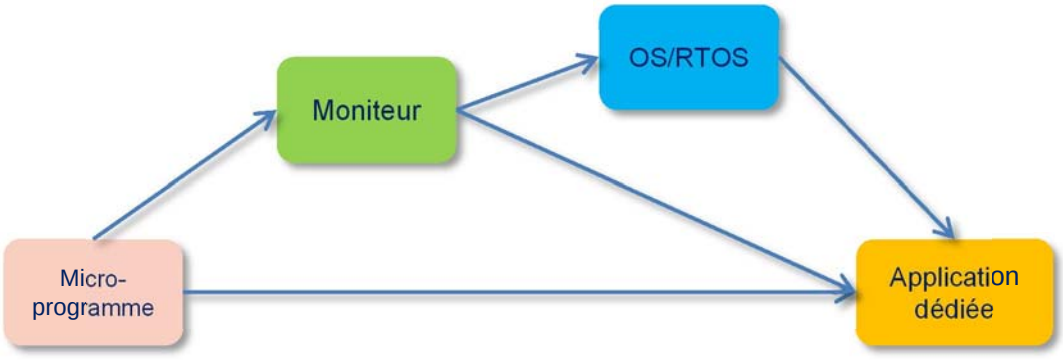
- Finalement, la connexion réseau de type *TCP/IP* permet un transfert de données plus élaboré et nécessite l'utilisation d'adresses *IP*, et de ce fait, la plate-forme doit être connecté à un réseau local (sous-réseau éventuel).



Lorsqu'un processeur/microcontrôleur démarre, son pointeur d'instruction est généralement initialisé à une adresse mémoire contenant la première instruction à exécuter. Cette instruction peut faire partie d'un **moniteur** ou ***bootloader*** et correspond en principe au vecteur d'interruption *reset* (c-à-d qu'à chaque fois que l'on appuie sur un bouton *reset* si pour autant il y en a un, le processeur va poursuivre son exécution à cet emplacement).

Le code associé au *reset* va amener le processeur à exécuter du code d'initialisation. Le moniteur est responsable d'initialiser les périphériques de base, le contrôleur mémoire, un *timer*, etc. Il peut fournir un *shell* et intègre généralement les pilotes de périphériques pour une interface UART, voire réseau.

Un moniteur peut démarrer un OS ou une application *standalone*. Les différents scénarios de démarrage possibles sont montrés sur la figure ci-dessous.



Architecture hôte-cible (4/4)

- **Emulateur de terminal série**
  - Liaison très simple entre la machine hôte et la cible
  - Protocole UART (RS-232)

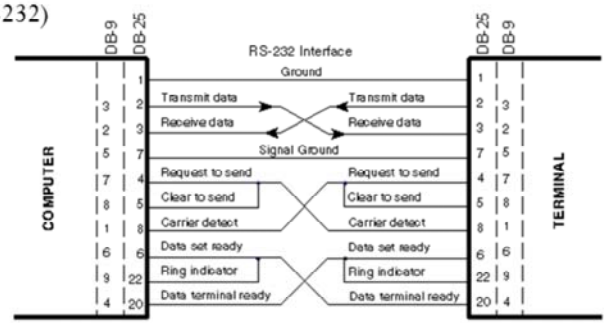


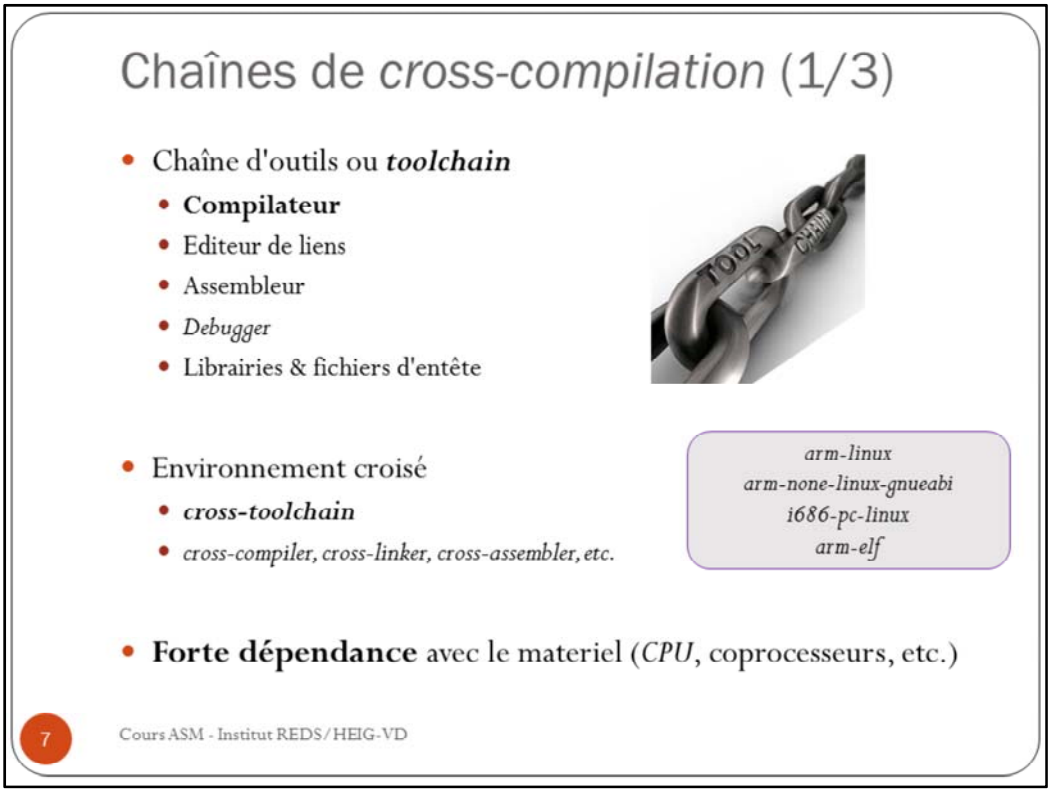
Figure 1. Direct-to-computer RS-232 Interface

- Des exemples:
  - *minicom*
  - *picocom*
  - *Hyperterminal*
  - …

6 Cours ASM - Institut REDS/HEIG-VD

L'utilisation d'une connexion sérielle de type *UART* nécessite l'utilisation d'un terminal série côté machine-hôte. Autrefois, un terminal série était matériellement caractérisé par un écran, un clavier et un connecteur série; ce matériel de base suffisait pour se connecter sur un système équipé d'une telle connexion.

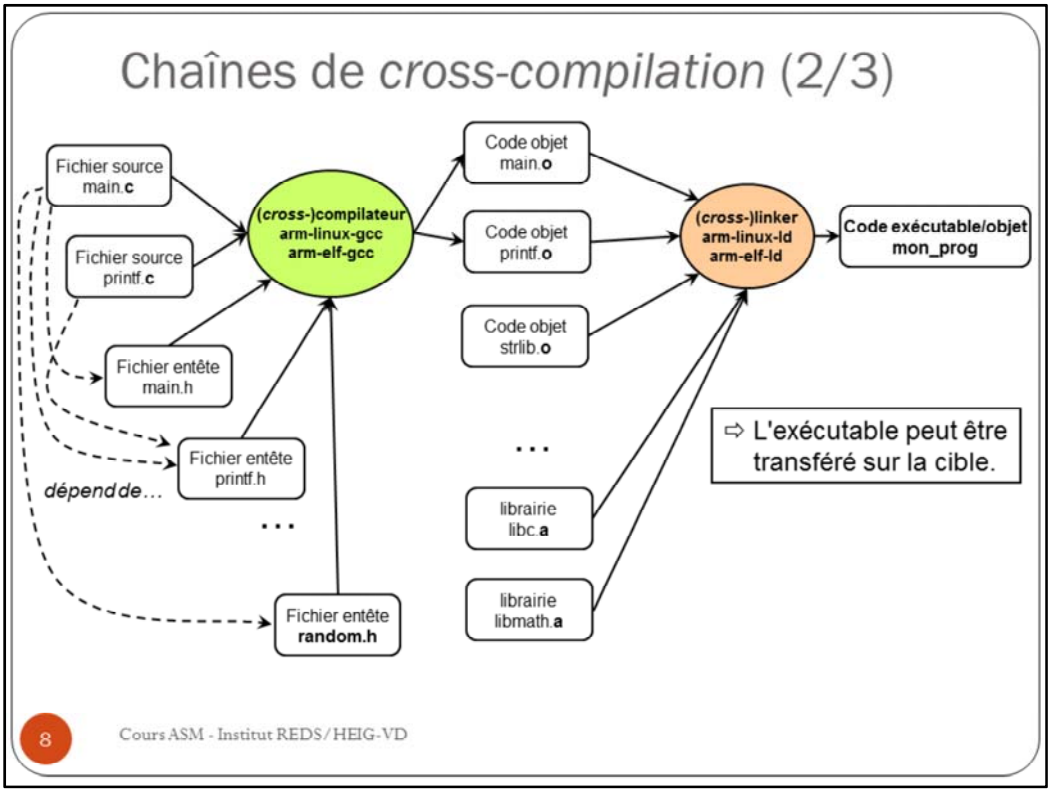
Aujourd'hui, un terminal série est simplement une application logicielle (on parle aussi *d'émulation* de terminal série). La connexion matérielle est, elle, bien réelle; il peut s'agir d'un connecteur 9 ou 25 *pins.* Comme ce type de connexion tend à être remplacé par des standards plus évolués, on trouve aujourd'hui des convertisseurs *UART-USB* permettant l'utilisation de simples ports USB.



Le développement d'une application quelle qu'elle soit nécessite l'utilisation d'une chaîne d'outils *(toolchain)* permettant la génération d'un exécutable (fichier binaire) à partir du code source. Par conséquent, il existe en principe une *toolchain* native pour chaque environnement matériel. De plus, le développement croisé nécessite une *toolchain* croisée (*cross-toolchain*) permettant la génération d'un binaire pouvant être exécuté sur un processeur différent de celui sur lequel la *toolchain* fonctionne. Par exemple, une *cross-toolchain* pour un processeur ARM peut tourner sur un processeur Intel/x86.
Les composants principaux que constitue une *toolchain* sont les suivants:

- **Compilateur/*cross-compiler***: compile un fichier contenant un code dans un certain langage vers des instructions binaires (hôte/cible). Le résultat d'une compilation est un ***code objet***.
- **Compilateur d'assemblage/*cross-assembler*:** compile un fichier contenant un programme écrit en assembleur. Le résultat est également un code objet contenant des instructions machine.
- **Editeur de liens/*cross-linker***: lie tous les *codes objet* produits par le compilateur, y compris éventuellement ceux en provenance de librairies pour former un exécutable.

La *toolchain* comprend également les librairies et fichiers d'entête nécessaire à l'utilisation de fonctions standards ou spécifiques. Elle peut comprendre aussi un *debugger*.



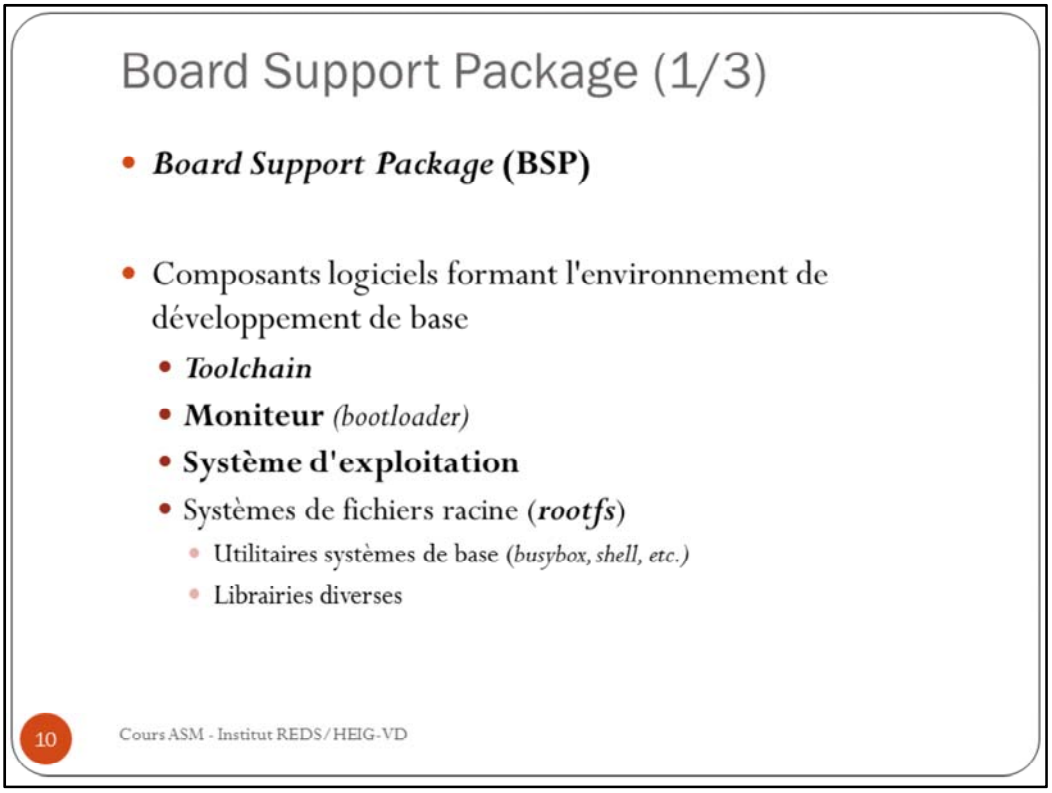
Le cycle de construction d'une image binaire passe principalement par une phase de compilation et une phase d'édition des liens *(linkage).*

La compilation fonctionne de manière similaire entre les différents langages: le compilateur traduit le code du langage de départ vers un code assembleur; le code assembleur est lui-même compilé (traduit) vers un code machine. Le fichier résultat est un code objet; il s'agit de la traduction des instructions du fichier source uniquement. C'est dire que si une référence vers une autre fonction (ou variable) est sollicitée, cette référence ne peut être *résolue* lors de la compilation: cette référence apparaît dans le code objet comme une ***référence symbolique*** (il faut s'imaginer qu'une référence à ce stade représente une adresse).

L'édition des liens permet d'assembler l'ensemble des codes-objet intervenant dans l'application afin de produire un binaire exécutable ***(linkage statique)***. Les références symboliques pourront être remplacées par des références relatives ou absolues. La différence entre ces deux types d'adresse dépend du type d'objet référencé ainsi que des instructions machine résultantes de la compilation. Une adresse relative peut dépendre de la position courante lors de l'exécution (adresse de l'instruction en cours d'exécution) ou d'une référence de base. Une adresse absolue quant à elle réfère une position définitive dans le plan d'adressage.
Dans le cas de librairies partagées, la résolution des adresses symboliques peut se faire au chargement du binaire (dans ce cas, le *linker* est re-sollicité à ce moment), ou durant l'exécution. Dans les deux cas, on parle de ***linkage dynamique***.

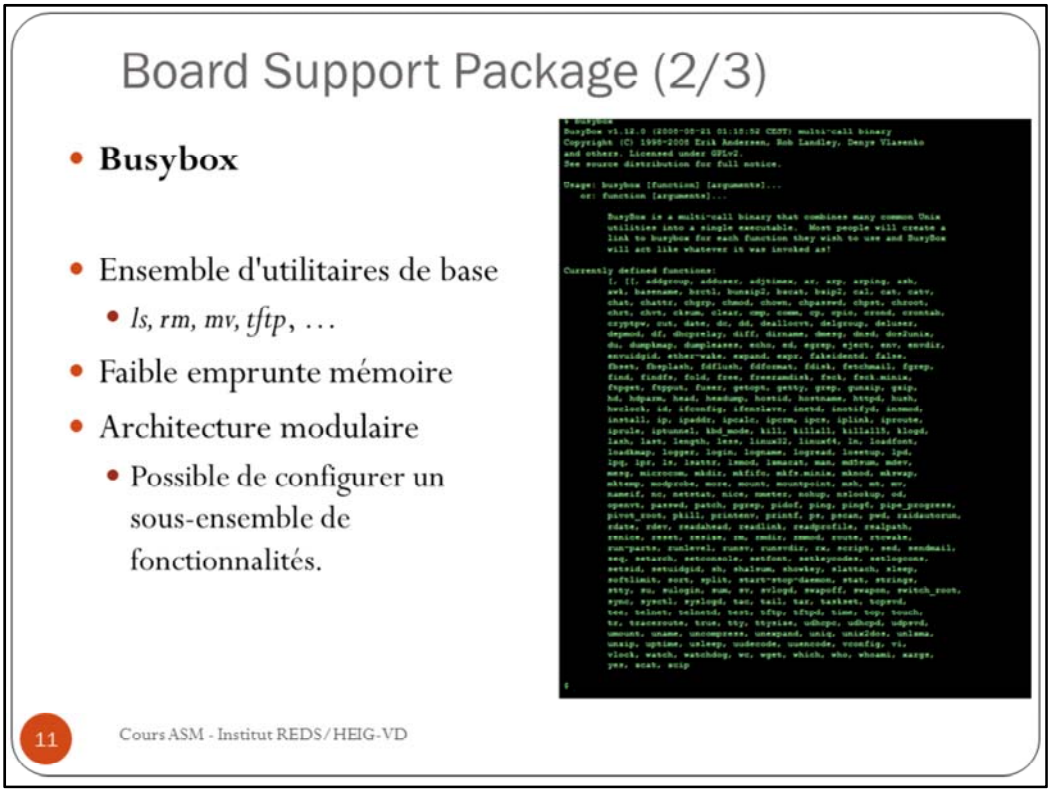
Chaînes de *cross-compilation* (3/3)

- Pourquoi un code objet n'est pas exécutable ?
- Quels types d'adresse peuvent se trouver dans un exécutable (*image binaire*) ?
- Que se passe-t-il au moment du chargement de l'exécutable en mémoire ?



Le *BSP* (*Board Support Package*) constitue l'ensemble des logiciels de base (et de référence) pour le développement d'applications sur une plate-forme. On y trouve notamment la *toolchain*, le moniteur (ou *bootloader*), le système d'exploitation et le système de fichiers contenant les utilitaires de base et les applications censées tourner sur la cible.

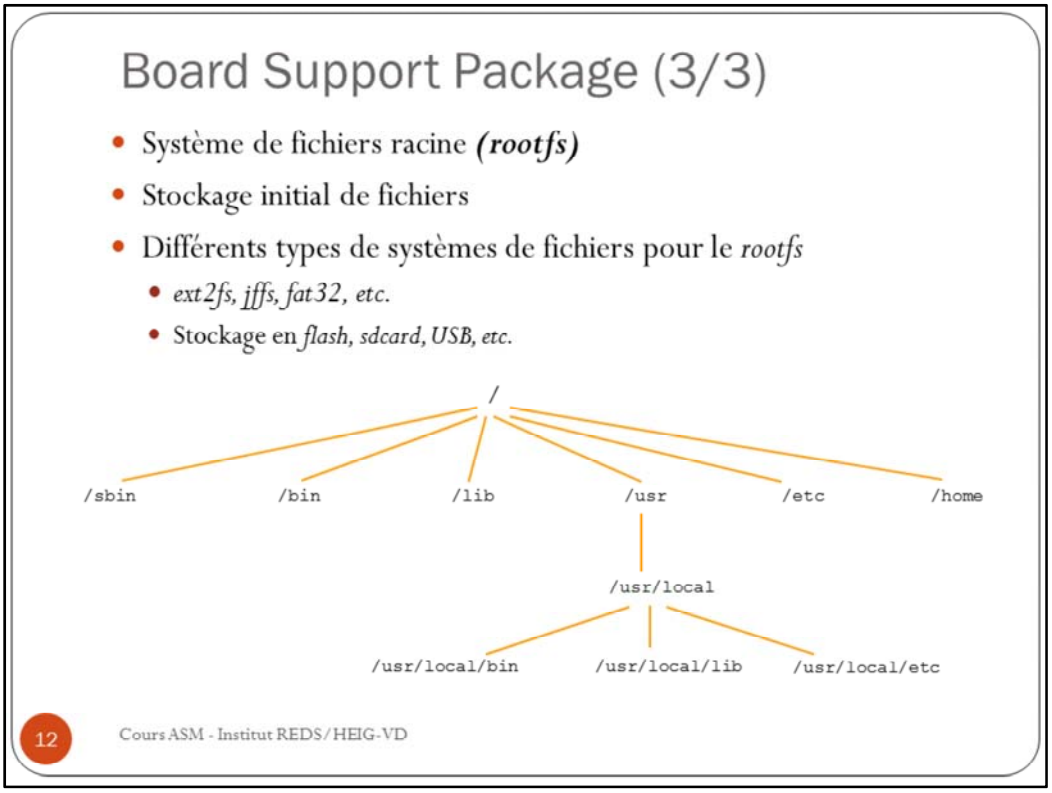
Généralement, le BSP est propre à chaque plate-forme; la *toolchain* est construite pour supporter les caractéristiques du microcontrôleur, le moniteur est configuré pour initialiser les périphériques de base de la plate-forme, l'OS a été également préconfiguré et testé pour la plate-forme cible, et les librairies doivent permettre une utilisation optimale de l'environnement logiciel et matériel.



Parmi les utilitaires que l'on trouve souvent dans un environnement embarqué, ***busybox*** occupe une place de choix. C'est une application qui implémente la plupart des utilitaires disponibles sous *Linux* sous une forme simplifiée et réduite. Ainsi, au lieu de déployer une grande quantité d'utilitaires système de base *(ls, mv, rm, tftp, mount, etc.),* une seule application regroupe l'ensemble de ces utilitaires. Leur utilisation est similaire à l'environnement standard grâce à l'utilisation de liens symboliques. Par exemple, l'application *ls* est un lien symbolique sur *busybox*. On se rappelle que le premier argument de la fonction *main()* d'une application C est précisément le nom de l'application que l'on entre dans le *shell*. De cette manière, *busybox* peut déterminer de quel utilitaire il s'agit et exécuter la fonction correspondante.

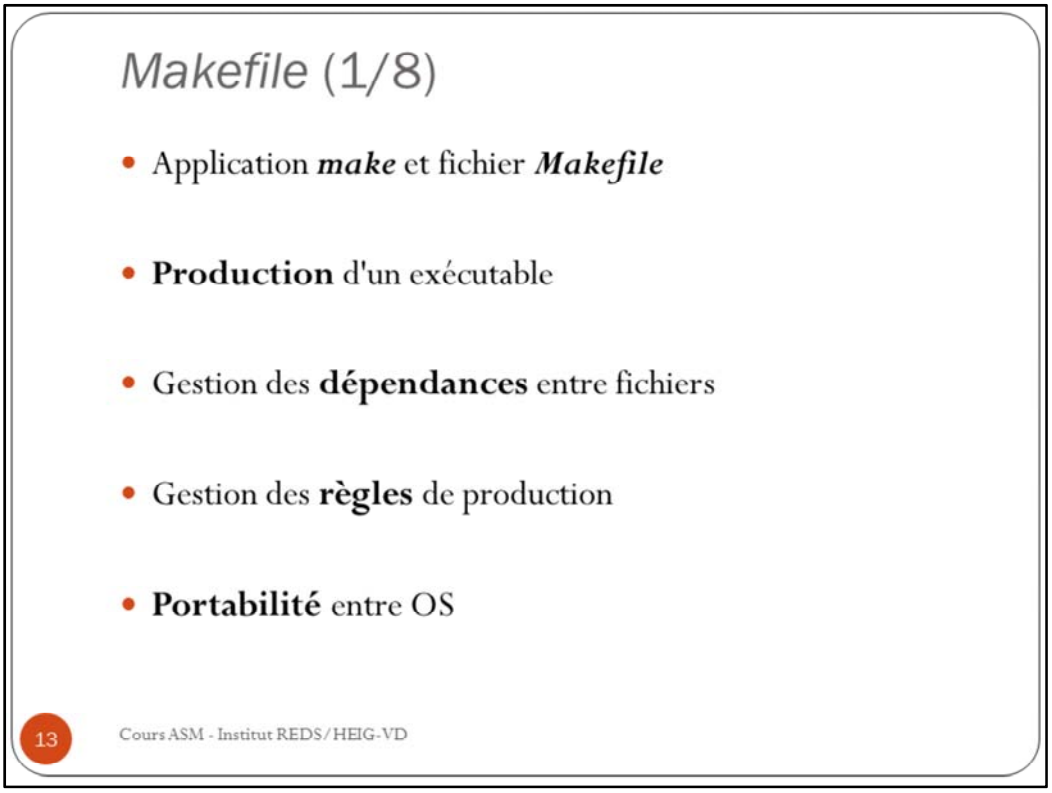
L'application *busybox* peut facilement être téléchargée gratuitement (*http://www.busybox.net*) et se laisse configurer de la même manière qu'un noyau *Linux*.

Dans un environnement embarqué, il est aussi utile de mentionner l'existence d'une librairie d'appels système simplifiée et réduite (appelée *uclibc*) en remplacement de la fameuse *libc* plus gourmande en place et en occupation RAM (*Android* possède un équivalent appelé *bionic*).



Le système de fichiers racine ***(rootfs)*** constitue l'ensemble des fichiers et applications déployées sur une cible organisés sous forme d'une arborescence dont le répertoire racine est "/" (ou "\").
C'est aussi le premier système de fichiers *monté* dans un PC traditionnel. A partir du *rootfs*, d'autres systèmes de fichiers peuvent être utilisés (montés). Il est habituel d'avoir un premier *rootfs* (pouvant être stocké au même endroit que le noyau) monté en RAM après le *bootstrap* du noyau, avant de monter un second *rootfs* plus conséquent.

Il faut comprendre que le *rootfs* ne constitue pas un *type* de système de fichiers, mais caractérise le système de fichier principal (de base) utilisé par l'OS. C'est lui qui contient les utilitaires systèmes notamment. Le *rootfs* peut être de différents types: *jjfs* ou *yaffs* s'il est stocké en flash, *ext2fs* si on peut le stocker sur une grosse mémoire de masse *(SD-card*), *fat32* s'il est stocké dans une clé USB, *ramfs* s'il doit être monté en RAM, etc. Par conséquent, le type de système de fichiers dépendra fortement du dispositif de stockage et de leurs caractéristiques propres.



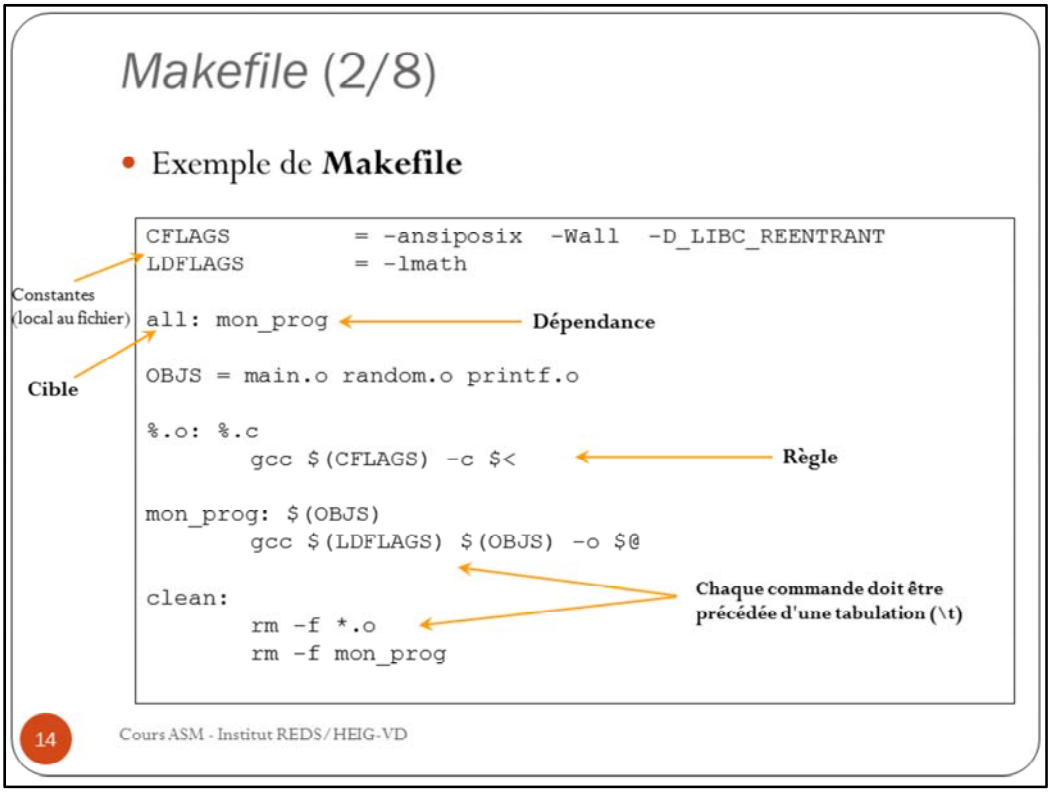
L'application *make* est très répandue dans les systèmes Unix/Linux/MacOSX et permet la fabrication d'un exécutable à partir des fichiers de base (code source). Un fichier texte structuré appelé ***Makefile*** contient l'ensemble des règles de production qui facilitent la compilation et l'édition de liens pour de gros projets comprenant plusieurs centaines de fichiers. *Make* gère de manière efficace les dépendances entre fichiers et permet de recompiler uniquement les fichiers ayant subis des modifications. L'application *make* fonctionne de la manière suivante:

- On l'invoque sans paramètre. Dans ce cas, *make* recherche un fichier par défaut, nommé ***Makefile*** contenant les règles nécessaires à la construction d'une application.

- On invoque *make* avec le nom d'une cible. L'application recherche la cible passée en argument dans le fichier *Makefile.* Si aucun argument n'est passé à l'application *make,* celle-ci considère la première cible du *Makefile.*

- On invoque *make* avec l'option *–f* et en donnant explicitement le nom de fichier en remplacement du fichier *Makefile.*

Comme toute application et commande sous Linux/Mac OS X, le bon réflexe pour obtenir plus d'information: les **pages *man*** (*man make*).



Un *Makefile* est composé d'un triplet ***cible-dépendance-règle***.

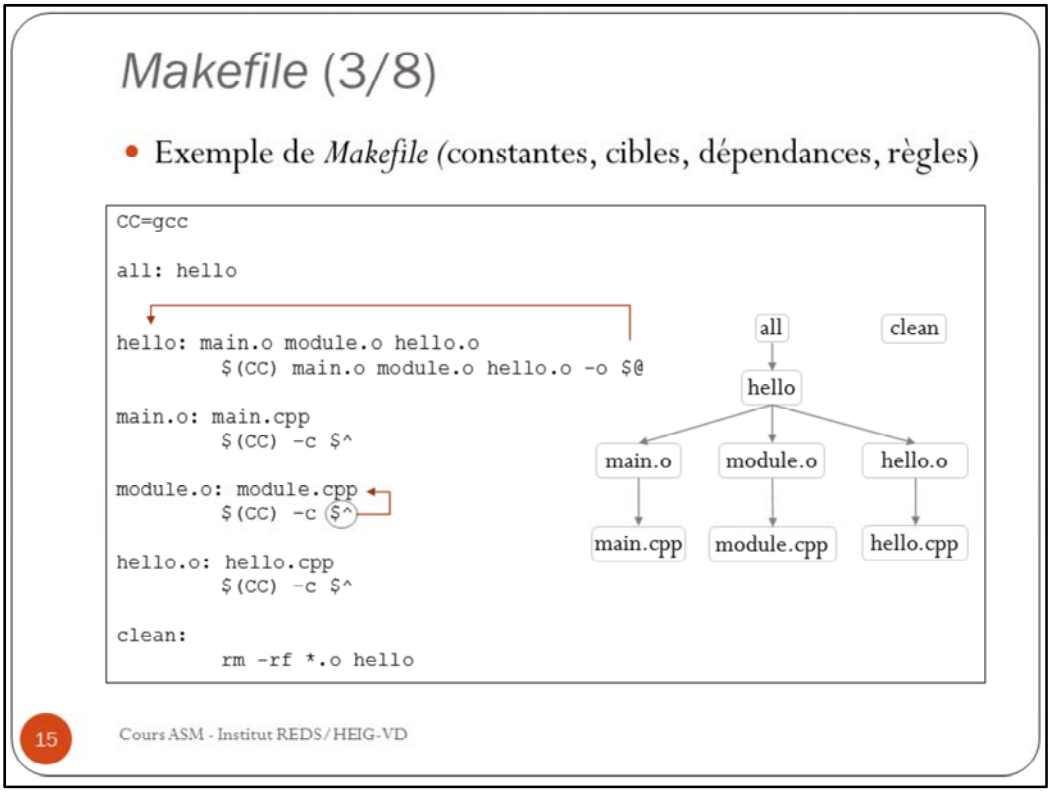
- La **cible** dénote généralement un "objet" à créer, générer, mettre à jour. Il s'agit la plupart du temps d'un fichier, mais pas seulement: la cible peut être de type *logique* auquel cas aucun fichier n'est associé. Un exemple typique est la cible *clean*.

- La (ou les) **dépendance** représente l'ensemble des fichiers nécessaires à la construction de la cible correspondante. Si les fichiers n'existent pas, *make* recherchera automatiquement une cible correspondante à cette dépendance, et effectuera la construction de la cible selon les règles associées. Si l'un des fichiers constituant les dépendances n'est pas à jour, la règle est exécutée.

- La (ou les) **règle** représente une (ou des) *commande* nécessaire à la génération de la cible. Ces commandes peuvent faire référence à l'une ou l'autre (ou toutes) des dépendances.

Un schéma traditionnel consiste au triplet suivant: (*binaire, fichier source, compilation du fichier source*) ou sous format d'un *Makefile*:

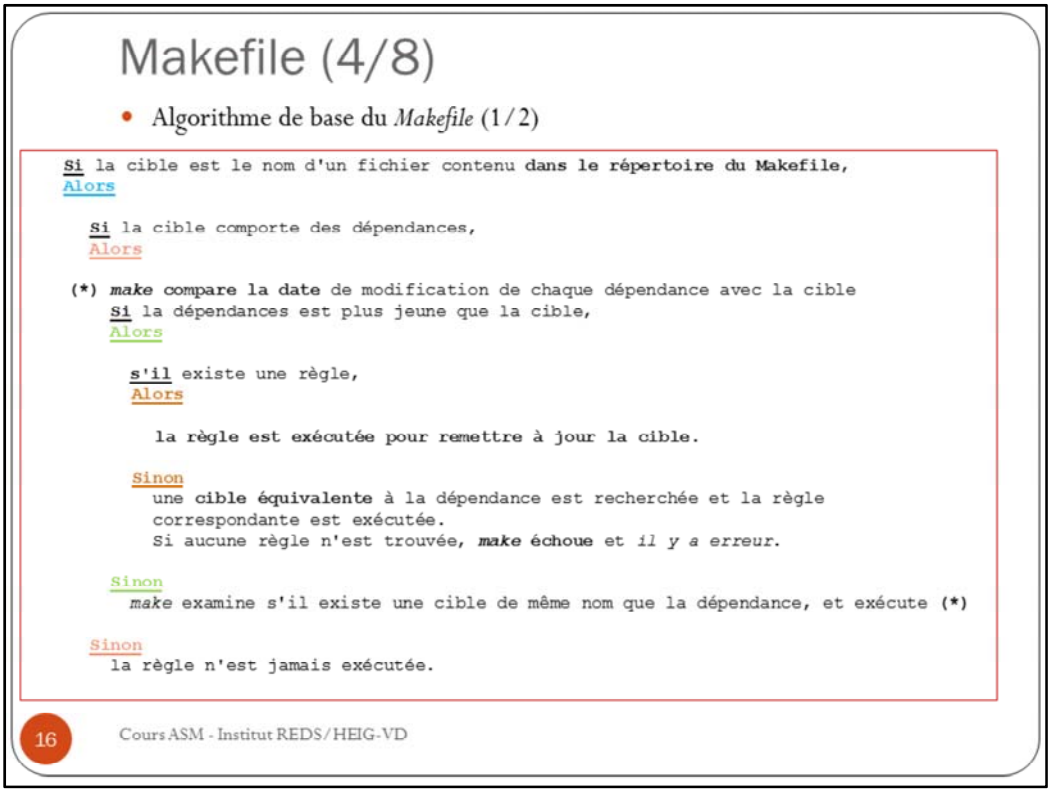
binaire: *fichier source*
<t> *compilation du fichier source*



L'exemple ci-dessus mets en évidence l'utilisation de constantes locales (référencées à l'aide de *$(<constante>)* et de *variables internes* au sein des règles.

La variable ***$@*** permet de faire référence à la cible, alors que la variable ***$^*** fait référence à la liste des dépendances. On voit ainsi qu'il est possible de décrire une règle de manière très générique.

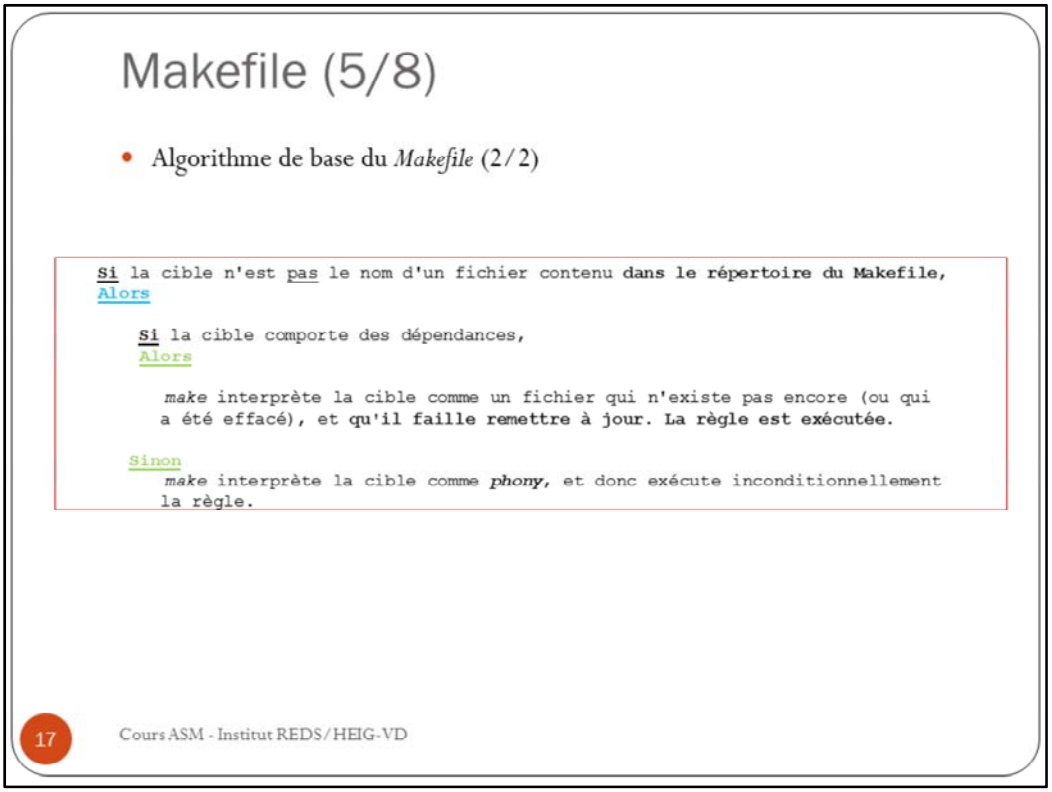
Nous verrons plus tard l'utilisation de *règles d'inférence* qui permet de décrire un triplet (cible, dépendances, règles) avec une grande flexibilité.



Normalement, la cible fait référence à un fichier. Si celui-ci existe déjà, les dépendances correspondantes à cette cible vont être considérées par l'application *make.* Lorsque les dépendances correspondent à des fichiers existants, *make* compare la date de dernière modification avec la date courante, et décide s'il faut appliquer la règle le cas échéant. En revanche, si la dépendance n'existe pas (encore), *make* va chercher dans le *Makefile* une cible correspondant à cette dépendance afin de la construire. S'il n'est pas possible de trouver une telle cible, la dépendance ne peux pas être générée et *make* termine en erreur.

De plus, pour chaque dépendance examinée, s'il n'y a pas de différence de date avec la cible, *make* recherche une éventuelle cible avec le même nom que la dépendance, et compare les dates des dépendances sous-jacentes avec cette cible, et cela de manière récursive.

Il peut arriver également que la cible ne se réfère pas à un (vrai) fichier, mais plutôt à une **cible *logique***. C'est le cas par exemple pour la cible *clean* (lorsque l'on tape *make clean* typiquement, il n'y a pas génération d'un fichier *"clean"*. On peut donc définir de telles cibles logiques. Ces cibles sont aussi appelées cibles *phony*. La directive ***.PHONY:*** *clean* dans un *Makefile* précise que si un fichier *clean* devait exister dans le répertoire courant, il faut l'ignorer et considérer la cible comme étant une cible logique.



En résumé, les propriétés suivantes sont appliquées par l'application *make*:

1) Descente récursive

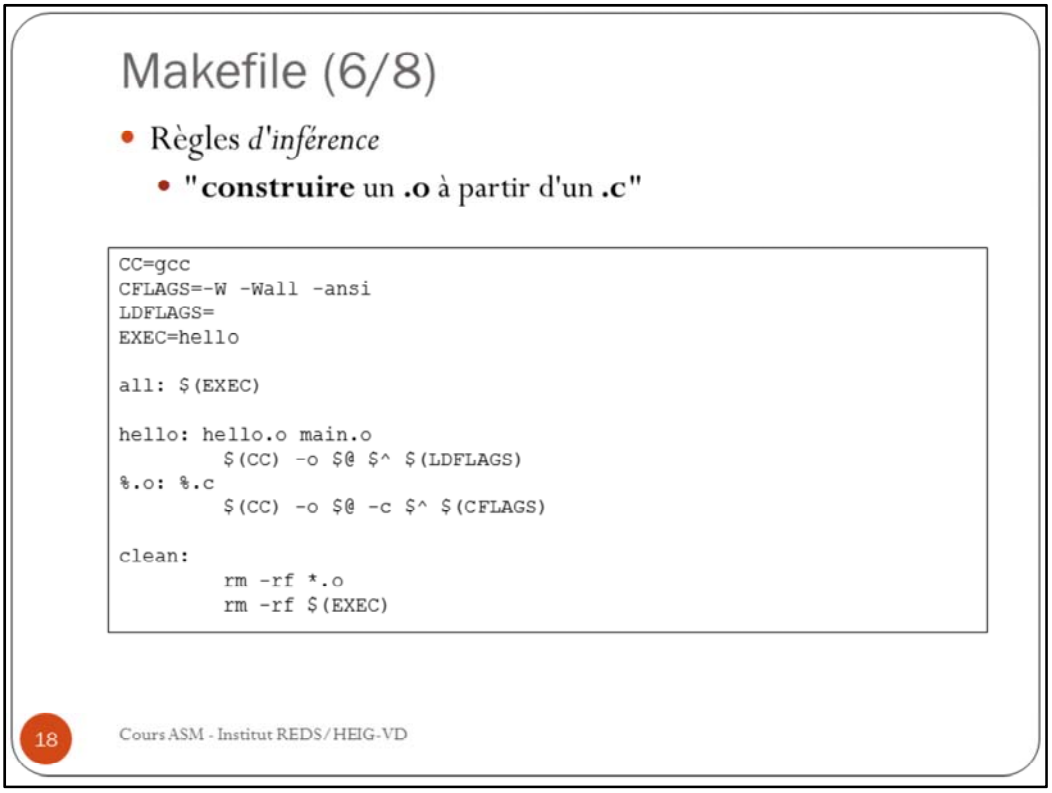
Pour chaque dépendance, *make* parcours le *Makefile* complètement pour voir s'il existe une cible équivalente et applique l'algorithme sur celle-ci (mise à jour des dépendances sous-jacentes tant que possible).

2) Exécution itérative

Make essaie toujours de trouver une cible associée à une règle, permettant ainsi la construction (ou mise à jour) de la cible. Quelque soit l'emplacement d'un triplet, *make* peut chercher à ré-appliquer des règles définies *avant* le triple en question.

3) Sélection opportuniste d'une règle

Dès qu'une règle est applicable, elle est exécutée, quelque soit le nombre de dépendances (typiquement un triplet comportant un nombre de dépendances inférieur à un autre triplet ayant une cible identique peut être sélectionné et la règle exécutée).



Les règles d'inférence permettent d'optimiser grandement l'écriture de triplets, notamment en rendant les règles très génériques. Ainsi, on peut énumérer une règle de compilation qui sera appliquée sur l'ensemble des fichiers ayant une extension spécifique (*.c, .S, .c++, etc.*).

De plus, nous disposons également de constructions syntaxiques pour obtenir une liste de fichiers selon certains critères.
Par exemple, plutôt que d'énumérer la liste des fichiers objets comme dépendances d'une cible, il est possible de les **générer automatiquement** à partir de la liste des fichiers sources, en utilisant la construction suivante :

OBJ = $(SRC:.c=.o)

De la même manière, il peut être utile de gérer la liste des fichiers sources de manière automatique :

SRC = $(wildcard *.c)

Makefile (7/8)

- Soit un programme *calcul2D* constitué d'un fichier *main.c*, des fichiers sources *liste.c*, *vecteur.c*, *matrice.c* et des fichiers entêtes correspondants: *liste.h*, *vecteur.h* et *matrice.h*

  ⇒ Ecrire un *Makefile*, de telle sorte que si seul un sous-ensemble des fichiers sources a été modifié, alors **seul ce sous-ensemble** soit recompilé (attention aux fichiers *.h*)

Makefile (8/8)

- Sur la base de l'exercice précédent, on suppose que les fichiers *vecteur.c* et *matrice.c* font partie d'une librairie *libm.a*
- Adapter le *Makefile* de telle sorte que l'application soit *liée* avec la librairie *libm.a* et que si une modification intervient dans l'un des fichiers source de la librairie, celle-ci soit mise à jour également.

- *Indication*: la construction d'une archive (librairie) s'effectue de la manière suivante: ***ar r [archive] [member…]***

Références

- Moniteurs, *bootloaders*
  - *http://www.denx.de/wiki/U-Boot*
  - *http://sourceware.org/redboot*
  - *http://www.gnu.org/software/grub*
- Références sur *JTAG*
  - *http://grouper.ieee.org/groups/1149/1*
  - *http://www.jtag.com*
- Environnement sur cibles
  - *http://www.busybox.net*
  - *http://www.uclibc.org*
- Pierre Ficheux, « Mise au point à distance avec GDB et QEMU », *OpenSilicium* n°1 (janvier 2011)
- Eclipse C/C++ Indexer
  - *http://help.eclipse.org/indigo/topic/org.eclipse.cdt.doc.user/concepts/cdt_c_indexer.htm*