Institut
Mines-Télécom

Structure des systèmes programmables

Bonjour systèmes embarqués

Alexis Polti

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr

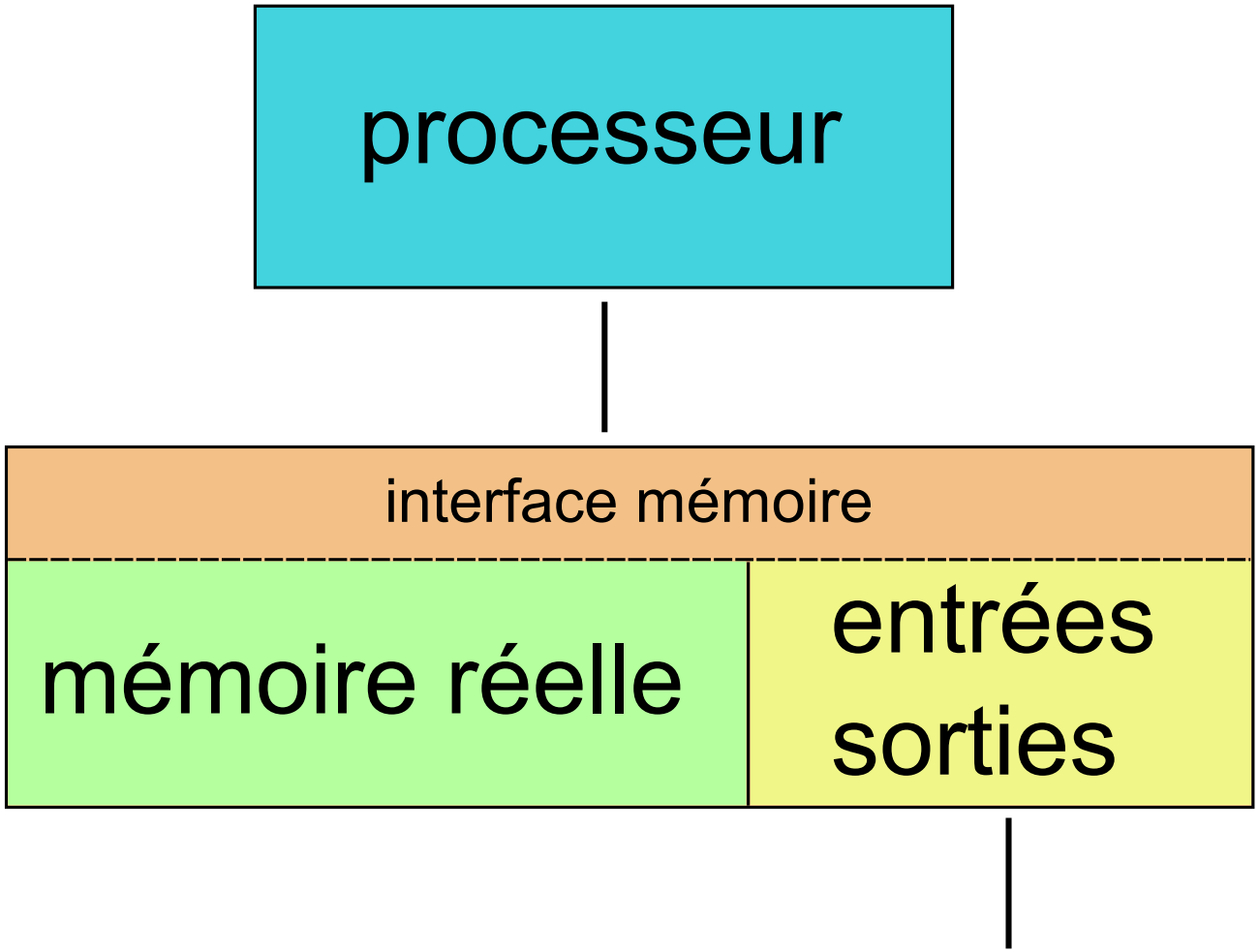
tl;dr

- **Ce qu'on va apprendre :**
  - le vocabulaire et les structures de base des systèmes embarqués,
  - que tout est soit processeur soit mémoire,
  - que code et variables ont généralement deux adresses : LMA et VMA,
  - ce qui se passe avant et après `main()`,
  - ce que fait un bootloader,
  - les différentes façons de débugger.

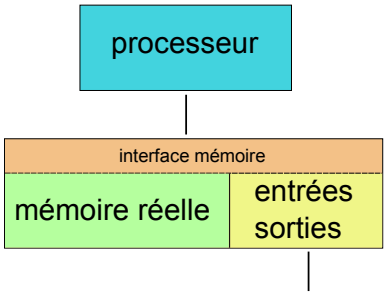
Plan

- **Les systèmes à processeur**
  - architecture, mapping mémoire
  - modes d'exécution et exceptions
  - types de systèmes : bare metal vs. hosted
  - vie des exécutables
  - bootloaders
  - debug

Architecture globale

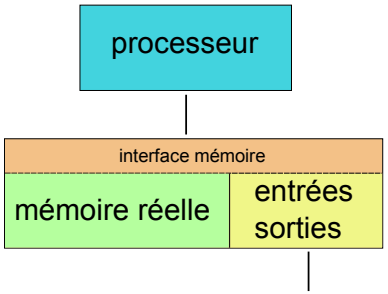


Architecture globale



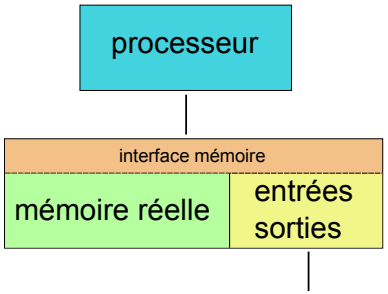
- **Mémoire**
  - « mémoire » = « présentant une interface similaire à celle d'une mémoire »
  - dispositif défini par la sémantique des accès :
    - accès = triplet (type [load / store], adresse, valeur)
  - physiquement
    - mémoire réelle
    - bancs de registres
    - périphériques
    - fonctions virtuelles (bit-banding, ...)

Architecture globale



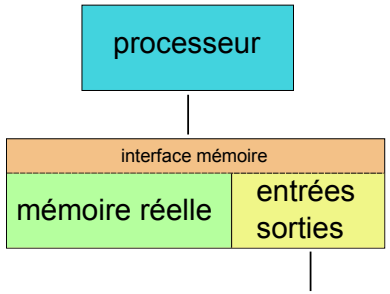
- **Plan d'adressage mémoire**
  - souvent : plusieurs « mémoires »
  - partage de l'espace mémoire du processeur
    - allocation d'une ou plusieurs plages d'adresses
    - accès : par des lectures / écritures de mots (load / store)
    - → définition : « périphérique mappé en mémoire »
  - décodage d'adresse
    - usuellement sur bits de poids fort d'adresse
    - fait par :
      - périphérique
      - éventuellement en partie par bus / NoC
    - facilité par le contrôleur de bus (chip select)
    - possibilité de configuration au vol (ex : PCI)

Architecture globale

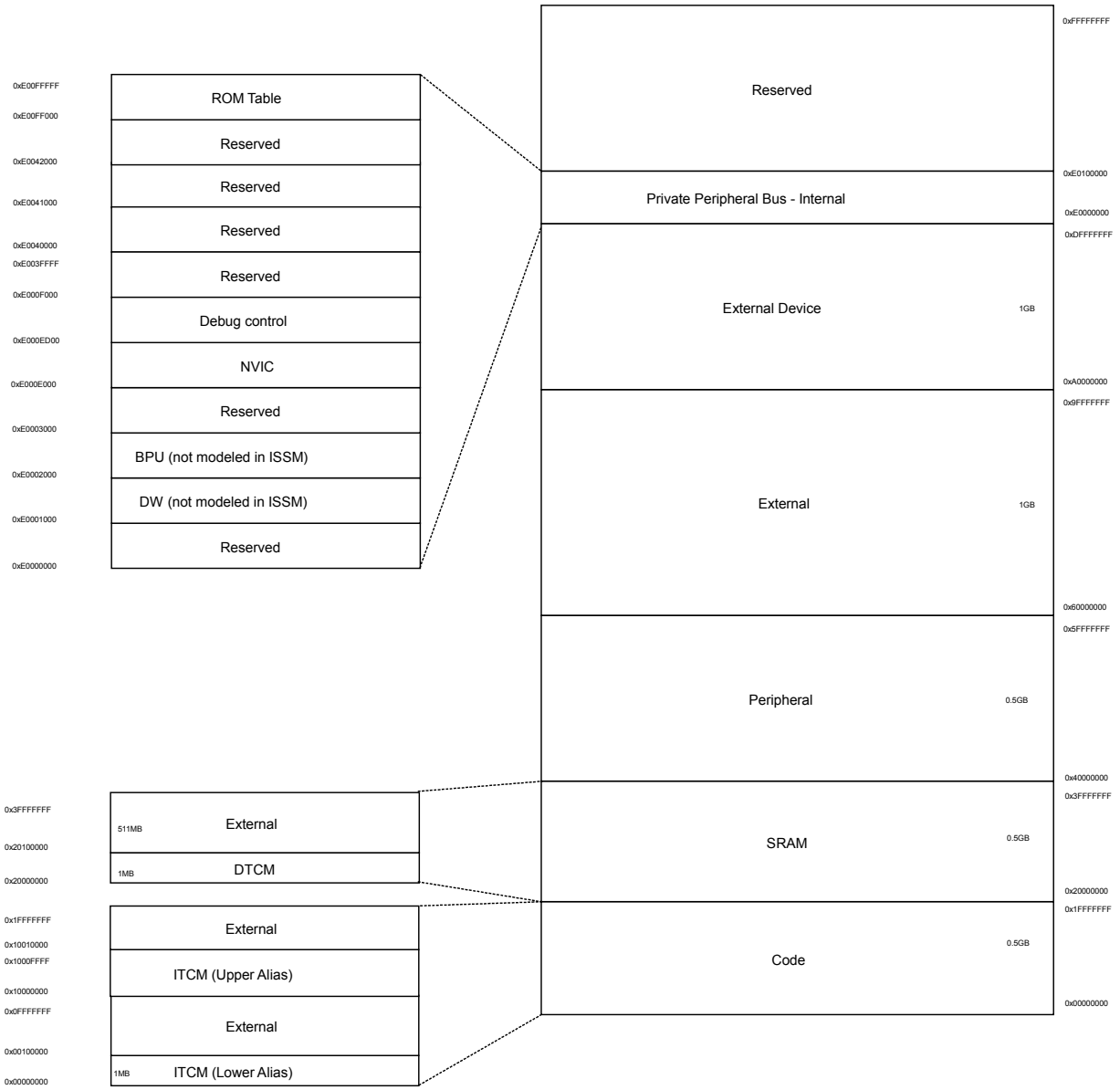


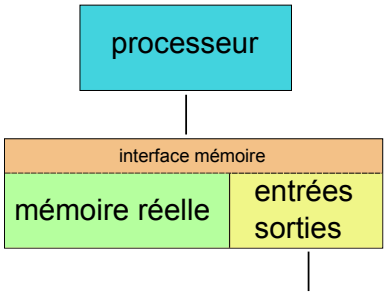
- **Plan d'adressage mémoire**
  - nécessité d'un plan mémoire (memory map)
    - pour le contrôleur de bus
    - pour le NoC
    - pour le système (bootloader, OS)
    - pour le programmeur
  - peut dépendre du mode du processeur
    - user / kernel
    - protégé, réel, ...

Architecture globale



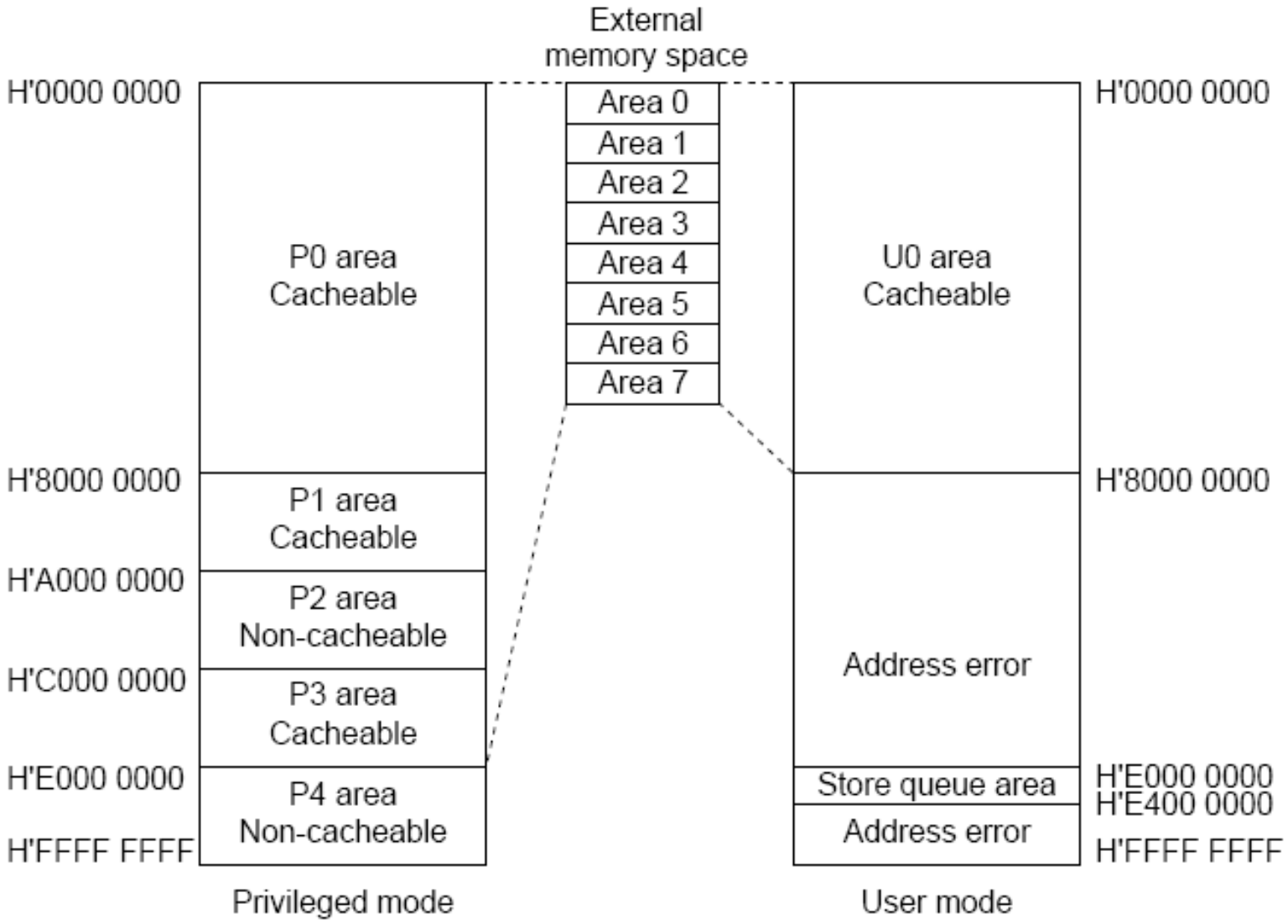
- **Exemple : ARM Cortex M0+**



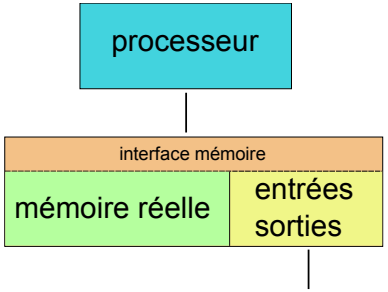


Architecture globale

- **Exemple : SH4**



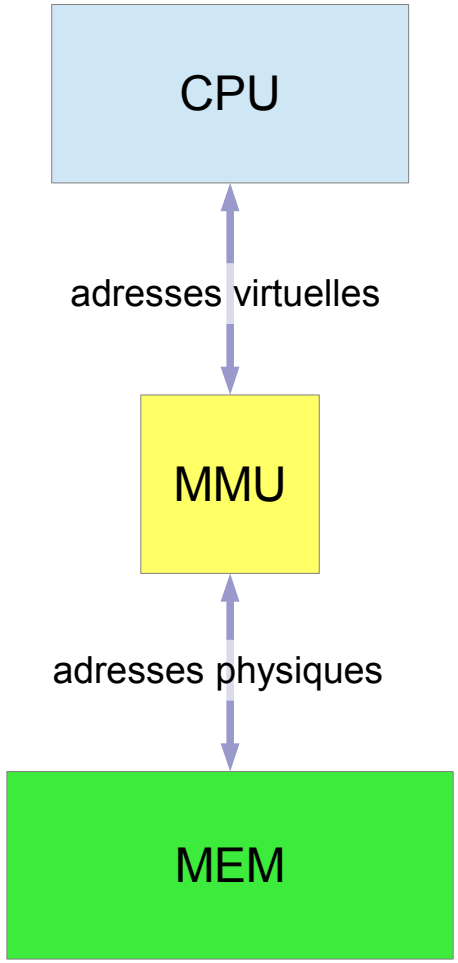
Architecture globale



- **Mémoires physiques**
  - vive
    - cache, centrale
    - SRAM, (S / DDR-S / DDR2-S / DDR3-S / ...) DRAM, …
    - rapides
  - morte
    - stockage de masse, bios, configuration
    - Flash, (E)EPROM, (M/P/OTP)ROM, …
    - lecture lente, écriture très lente
  - internes ou externes
  - la distinction vive-morte, RAM/ROM, RW/RO est floue...

Pause : les MMU

- **Memory Management Unit**
  - gestion de la mémoire :
    - traduction
    - protection (MPU)
    - contrôle de cache
    - …

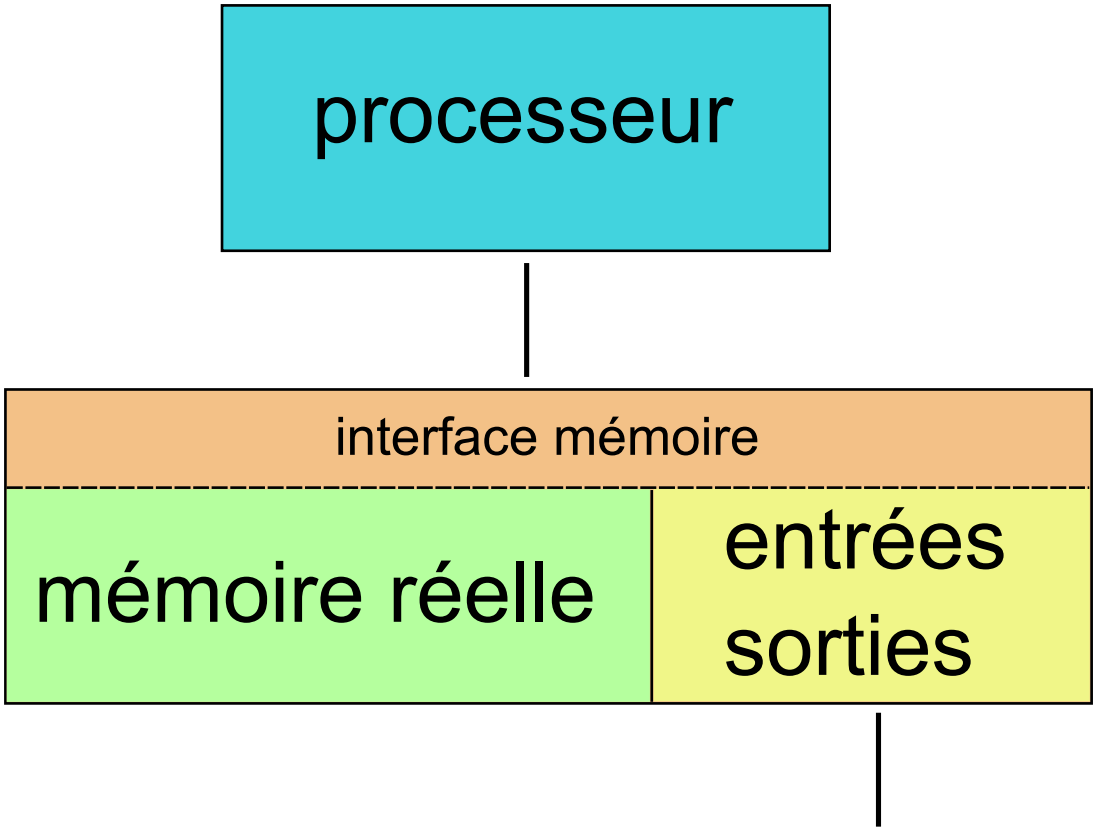


- **Quizz : quel est le lien entre MMU / MPU et swap ?**

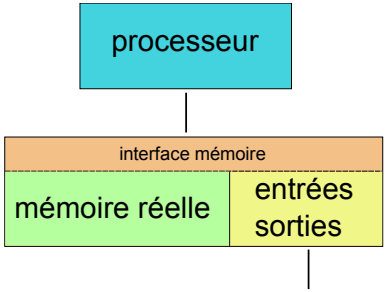
processeur
interface mémoire
mémoire réelle
entrées sorties

Harvard / von Neumann

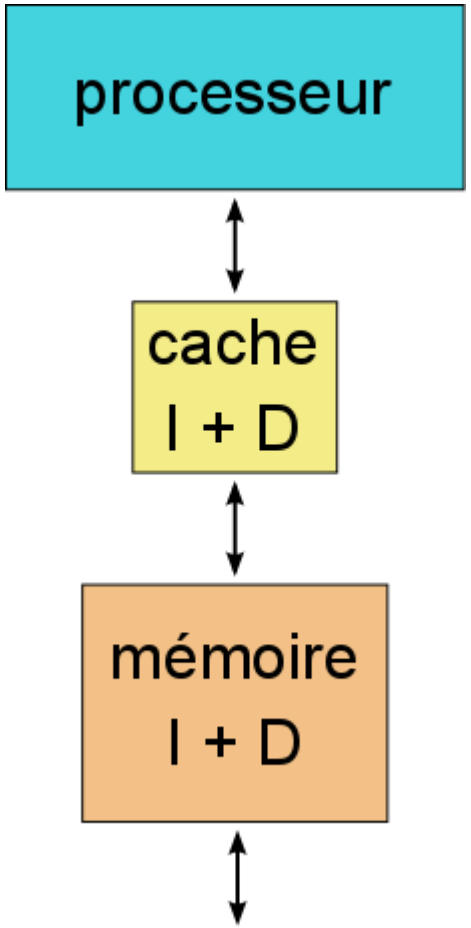
- **Zoom**
  - plusieurs architectures possibles
  - pour chaque architecture, deux définitions



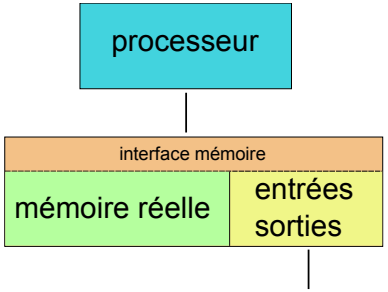
Harvard / von Neumann



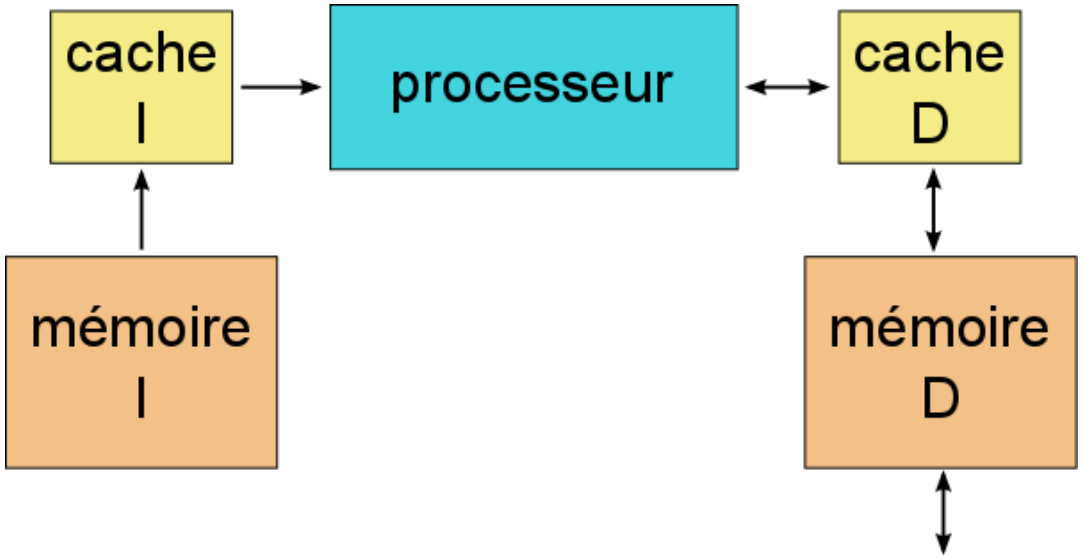
- **Architecture Von Neumann**
  - matériel : un seul chemin de données
  - logiciel : un seul espace mémoire
  - permet de traiter le code comme des données
    - est-ce bien ?
  - goulot d'étranglement : deux cycles pour chaque instructions accédant à des données en mémoire.



Harvard / von Neumann



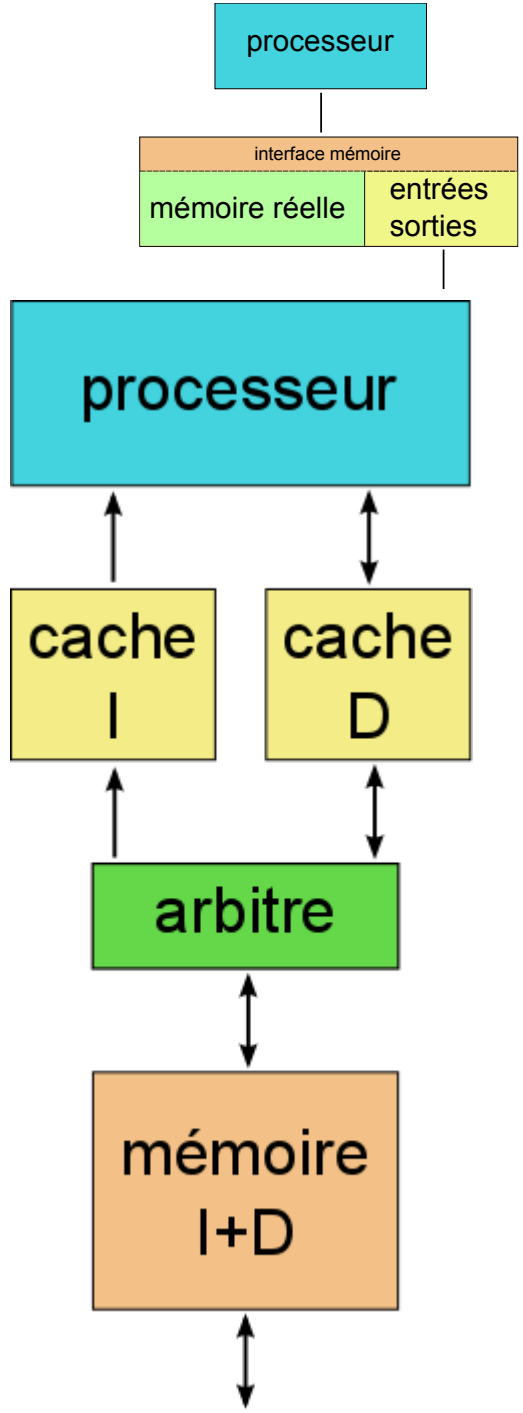
- **Architecture Harvard**
  - matériel : deux chemins de données distincts
    - un pour les instructions
    - un pour les données
  - logiciel : deux espaces mémoire distincts



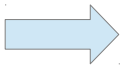
  - Problèmes en C :
    - sur les contrôleur où la flash contient le code, et la RAM les données, où stocker les données constantes ?
    - comment recopier / modifier des instructions ?

Harvard / von Neumann

- **Architecture Harvard unifiée**
  - appelée aussi Harvard modifiée
  - prend le meilleur des deux mondes
  - problèmes :
    - cohérence de cache lors de la manipulation d'instructions
    - performances réduites en cas de cache miss répétés
  - architecture (de loin) la plus répandue !



Où en est-on ?



- **Les systèmes à processeur**
  - architecture, mapping mémoire
  - modes d'exécution et exceptions
  - types de systèmes : bare metal vs. hosted
  - vie des exécutables
  - bootloaders
  - debug

Modes et exceptions

- **Modes de fonctionnement**
  - il existe souvent au moins 2 modes :
    - superviseur
    - utilisateur
  - chaque mode correspond à des ressources propres
    - permet la mise en place de protection pour les OS
    - les ressources propres sont généralement
      - des registres : PC, pointeur de pile, …
      - des instructions spéciales
  - Le passage d'un mode à un autre s'effectue par les exceptions : interruption du flot normal d'exécution.

Modes et exceptions

- **Exceptions :**
  - permettent de signaler un événement particulier au processeur pour qu'il arrête son traitement courant et effectue temporairement un traitement spécifique.
  - peuvent être générées :
    - par des instructions spéciales
      - appelées usuellement TRAP, exception logicielle, interruption logicielle
      - permet d'implémenter des appels système
    - à la demande d'un périphérique externe
      - appelé usuellement interruption matérielle (IRQ)
    - par un dysfonctionnement du système (échec d'un accès mémoire, instruction non reconnue, …)
      - appelé usuellement exception matérielle

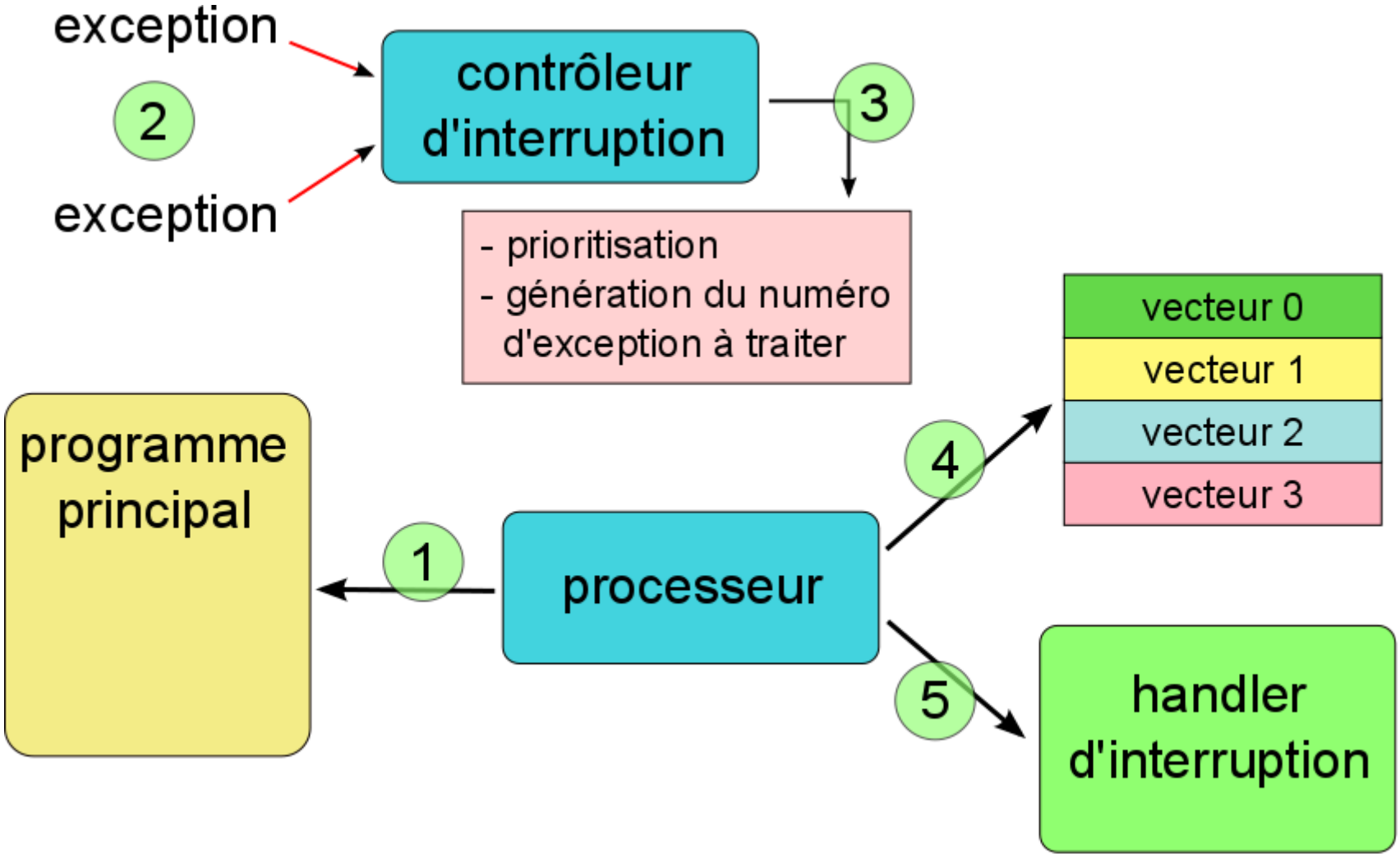
Modes et exceptions

- **Typologie des exceptions**
  - synchrone : si elle a toujours lieu au même moment pour un même set de données et la même allocation mémoire. Exemples :
    - divisions par 0, page fault, undefined instruction
    - TRAP
  - asynchrone : si elle n'a pas de relation temporelle avec le programme courant. Exemples :
    - interruptions externes
    - data memory error
    - reset

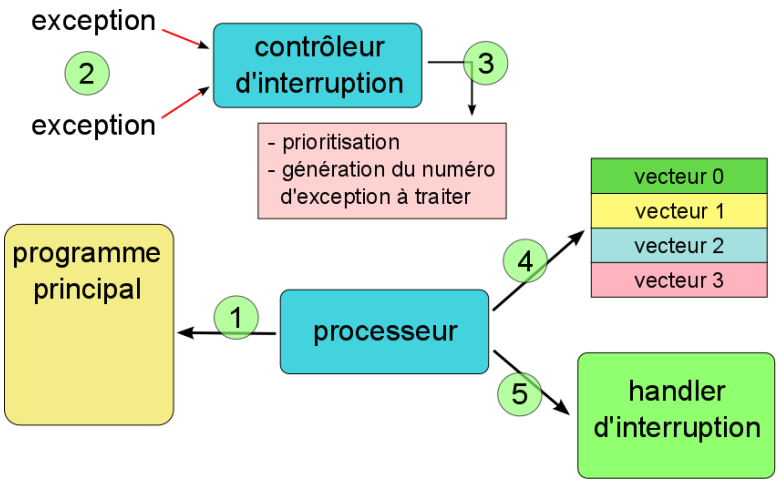
Modes et exceptions

- **Handlers, vecteurs, table des vecteurs**
  - définitions :
    - à chaque exception correspond un code à exécuter appelé *handler*
    - la première instruction de ce handler est située à une adresse appelée *vecteur d'interruption*
  - les vecteurs sont souvent regroupés au sein d'une table
    - appelée *table des vecteurs d'interruption*
    - hard codée / en ROM / en RAM
    - il peut y avoir plusieurs tables (modes, …)
  - pour limiter la taille de la table, certaines interruptions partagent le même vecteur. On parle alors d'interruptions chaînées.

- **Traitement des exceptions**



Modes et exceptions



- **Traitement des exceptions**
  - selon le type d'exception :
    - retour à l'instruction interrompue (si problème)
    - retour à l'instruction d'après (si pas de problème)
  - en cas de double faute, beaucoup de processeurs se mettent en mode *erreur*
    - arrêt
    - l'examen des registres permet de savoir ce qui s'est passé
  - certaines architectures autorisent les IRQ préemptibles
  - certaines architectures autorisent les NMI

Où en est-on ?

- **Les systèmes à processeur**
  - architecture, mapping mémoire
  - modes et exceptions
  - types de systèmes : bare metal vs. hosted
  - vie des exécutables
  - bootloaders
  - debug

Types de systèmes embarqués

- **Types**
  - approche minimale : carte nue (bare-metal)
  - approche maximale : hébergé (avec OS / hosted)
  - beaucoup d'approches mixtes entre les deux

Types de systèmes embarqués

- **OS**
  - offre un ensemble de services (fonctions) aux exécutables
  - en général, au moins ces services usuels :
    - abstraction des périphériques
    - primitives de synchronisation
    - gestion du temps
    - gestion des tâches
    - gestion de la mémoire
    - gestion des fichiers
  - responsable de l'initialisation complète du matériel
  - charge les exécutables en mémoire et contrôle leur exécution

Types de systèmes embarqués

- **OS**
  - peut se présenter comme une bibliothèque normale, linkée avec l'application
    - les services de l'OS sont alors des appels de fonction normaux
  - peut être un code indépendant
    - les services de l'OS sont alors usuellement appelés par des TRAP (exceptions logicielles) : *syscalls*
  - dans tous les cas, l'OS installe ses propres handlers d'interruption

Types de systèmes embarqués

- **OS : exemples**
  - généralistes
    - Linux, xBSD, Solaris
    - Windows
    - OS/2, Plan9, …
  - dédiés aux systèmes embarqués
    - Windows Embedded, VxWorks, QNX, …
    - Symbian OS, Android, uCLinux, …
    - FreeRTOS, ChibiOS/RT, eCos, RTEMS, …
    - TinyOS, Contiki, ...
    - Forth (!)
  - Plus de 900 OS répertoriés sur Wikipedia

Types de systèmes embarqués

- **Bare-metal (free-standing)**
  - pas d'OS
    - alone in the dark : stress !
    - pas (ou très rarement) de bibliothèques partagées
  - souvent un seul exécutable auto-contenu en charge de la gestion de tout le système
  - différentes fonctionnalités sont à gérer :
    - initialisation
    - fonctionnement
    - mise à jour
    - debug

Où en est-on ?

- **Les systèmes à processeur**
  - architecture, mapping mémoire
  - modes d'exécution, exceptions
  - types de systèmes : bare metal vs. hosted
  - vie des exécutables
  - bootloaders
  - debug

Vie des exécutables

- **Architecture d'un programme**
  - stockage : en ROM, flash, mémoire de masse
  - exécution : selon l'optimisation souhaitée,
    - soit depuis la RAM (optimisation vitesse)
    - soit directement depuis la ROM (économie de RAM)
  - tout ne peut pas être laissé en ROM :
    - exemple : variables
    - → séparation données modifiables / non modifiables
  - économiser la place en ROM / mémoire de masse :
    - on ne stocke que le nécessaire (pas la pile, ni BSS)
    - sous forme plus ou moins compacte : formats d'exécutables

Vie des exécutables

- **Code**
  - généralement non modifiable
  - si exécuté à partir de la RAM :
    - nécessite souvent une recopie préalable
  - si exécuté directement à partir de la ROM :
    - XIP (eXecute In Place)
      - problèmes :
        - vecteurs d'interruption,
        - breakpoints,
        - support des bibliothèques,
        - clashs d'adresses, …
  - on peut le séparer en différentes sections, certaines XIP, d'autres en RAM...

Vie des exécutables

- **Données**
  - modifiables :
    - lors de l'exécution : situées en RAM (par définition)
    - quizz : mais alors quand la RAM n'est pas encore initialisée, on ne peut pas utiliser de variables ???
  - non modifiables (read-only) :
    - lors de l'exécution : situées en ROM
    - … ou éventuellement en RAM (avec protection d'accès)
    - quizz : si elles sont non modifiables, pourquoi les mettre en RAM ???
  - problèmes des variables initialisées :
    - avant exécution : stockées en ROM
    - nécessitent une recopie ROM → RAM

Vie des exécutables

- **Bilan : différentes parties d'un exécutable**

Nom	Description	Stockage	À l'exécution
TEXT	code (à priori non modifiable)	en ROM	en ROM ou recopié en RAM
BSS	variables non initialisées ou initialisées à zéro	pas stocké	créé en RAM
DATA	variables modifiables initialisées	en ROM	recopié en RAM
RODATA	constantes	en ROM	en ROM ou recopié en RAM

Attention : sera complété / modifié / nuancé par la suite !..

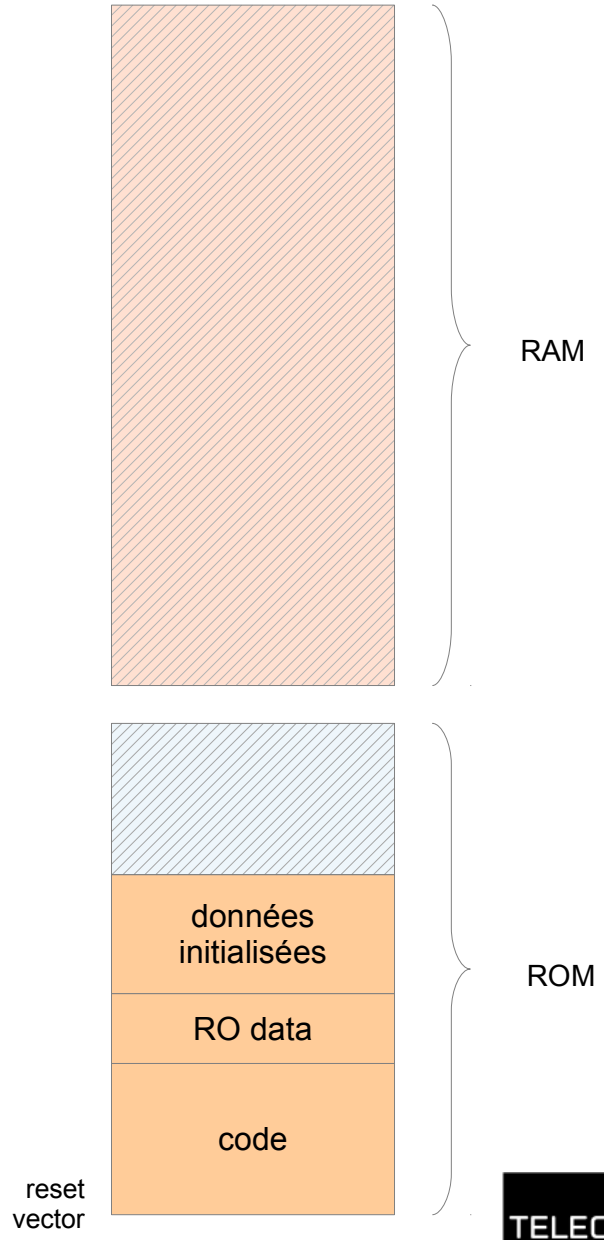
Vie des exécutables

- **Emplacements d'exécution**
  - déterminés à la compilation / édition de lien
  - trois possibilités :
    - code fixe
    - code relogeable
    - position-independant code (pic)
  - parfois nécessaire de déplacer certaines parties des exécutables (ROM → RAM) avant le lancement du main : qui, quand ?

- **Cas usuels extrêmes**
  - exemple 1 : système nu (bare-metal), sans MMU, exécutable stocké en ROM, exécuté depuis la RAM ou la ROM
  - exemple 2 : système avec OS et MMU, exécutable stocké sur disque, exécuté depuis la RAM

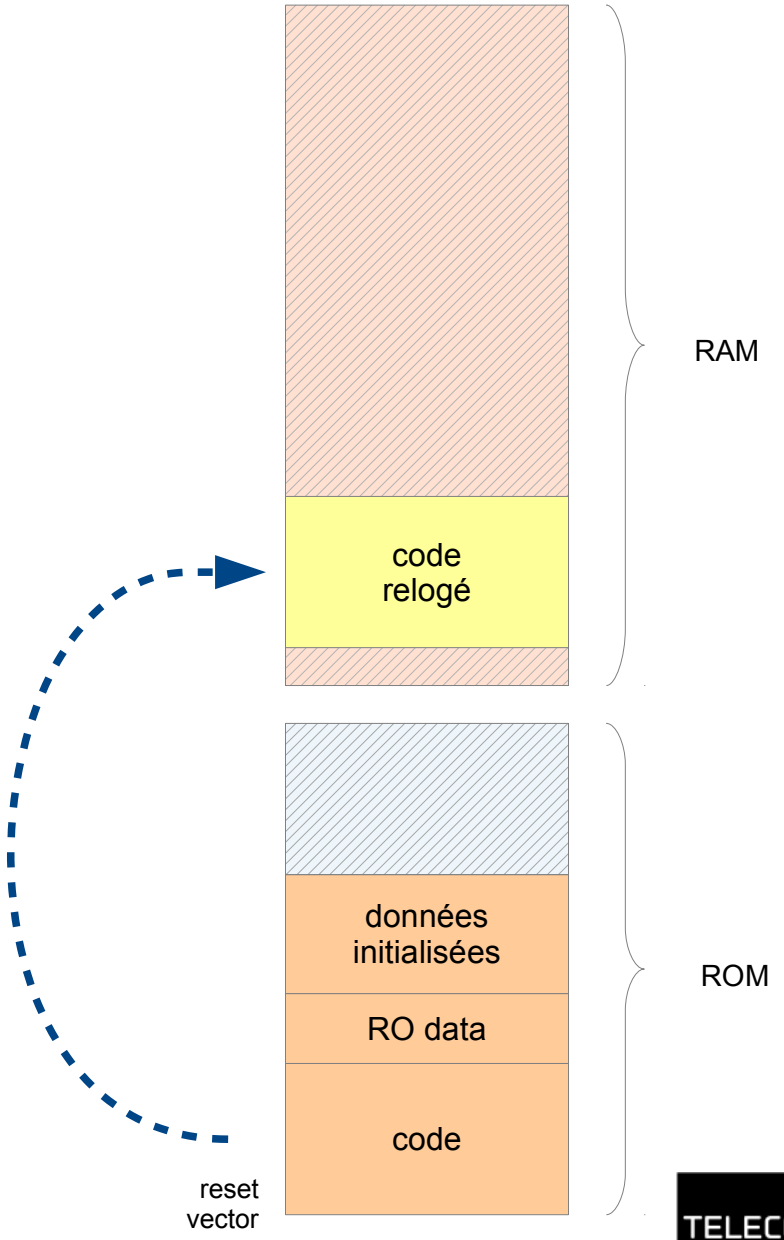
Vie des exécutables : exemple bare-metal

- **Lancement de l'exécutable**
  - stocké en ROM



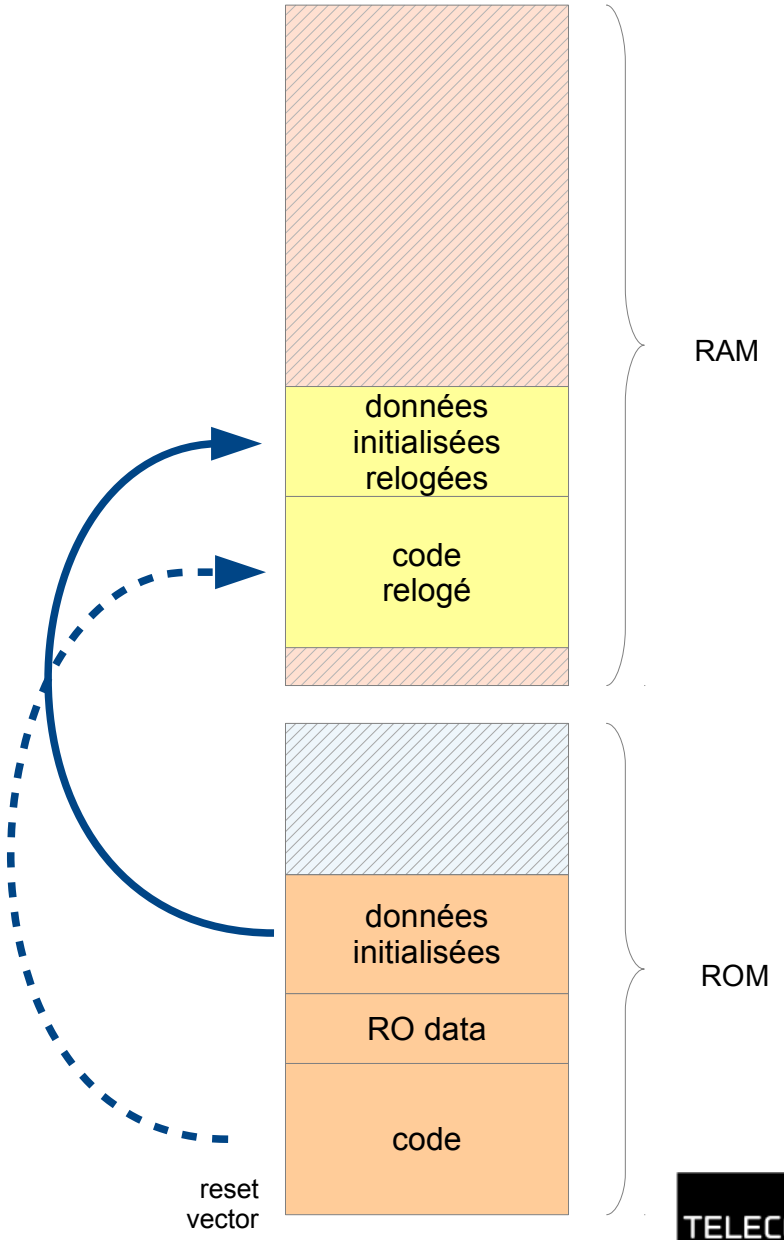
Vie des exécutables : exemple bare-metal

- **Lancement de l'exécutable**
  - stocké en ROM
  - éventuellement, recopie du code en RAM
    - par exécutable externe (bootloader, …)
    - ou par le code lui-même (`crt0.s`, …)



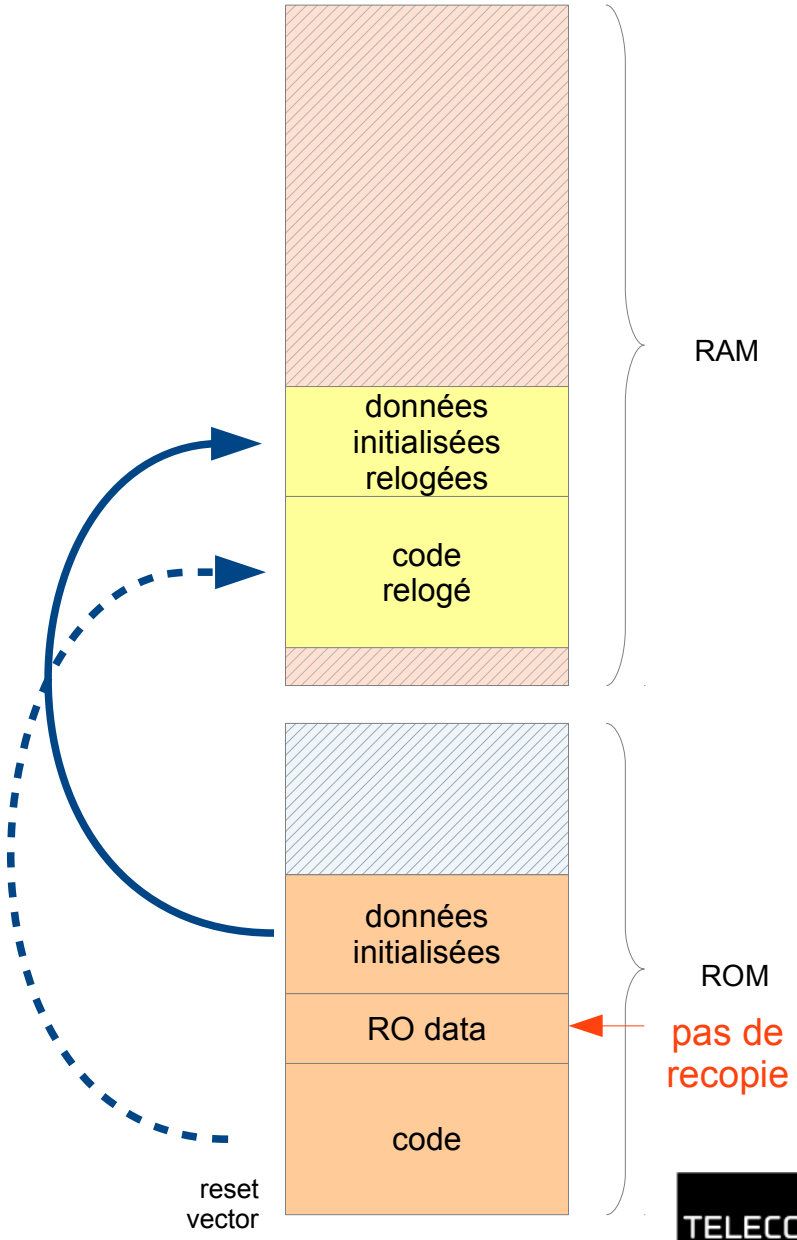
Vie des exécutables : exemple bare-metal

- **Lancement de l'exécutable**
  - stocké en ROM
  - éventuellement, recopie du code en RAM
    - par exécutable externe (bootloader, …)
    - ou par le code lui-même (`crt0.s`, …)
  - recopie des données en RAM par `crt0.s` (inséré par la chaîne de compilation)



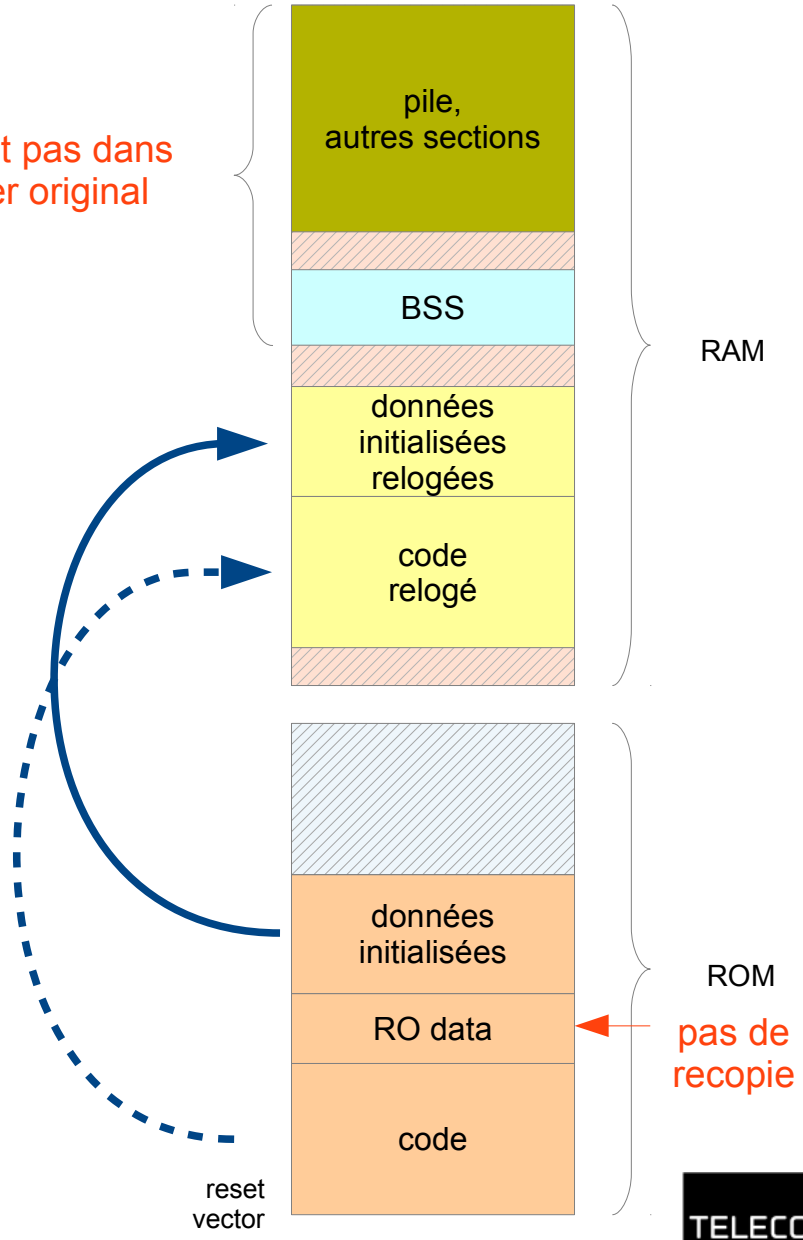
Vie des exécutables : exemple bare-metal

- **Lancement de l'exécutable**
  - stocké en ROM
  - éventuellement, recopie du code en RAM
    - par exécutable externe (bootloader, …)
    - ou par le code lui-même (`crt0.s`, …)
  - recopie des données en RAM par `crt0.s` (inséré par la chaîne de compilation)
  - les données non modifiables sont laissées en ROM



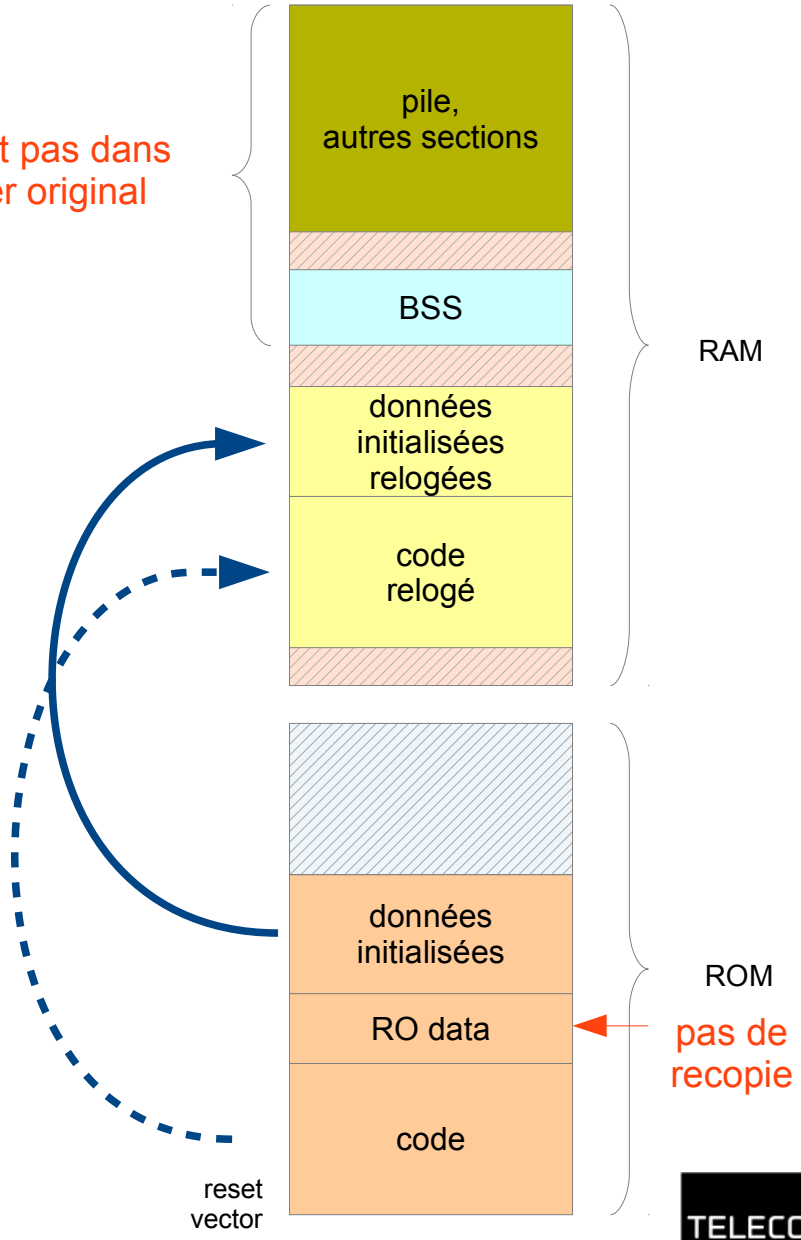
Vie des exécutables : exemple bare-metal

- **Lancement de l'exécutable**
  - stocké en ROM
  - éventuellement, recopie du code en RAM
    - par exécutable externe (bootloader, …)
    - ou par le code lui-même (`crt0.s`, …)
  - recopie des données en RAM par `crt0.s` (inséré par la chaîne de compilation)
  - les données non modifiables sont laissées en ROM
  - création du BSS, de la pile, … par `crt0.s`



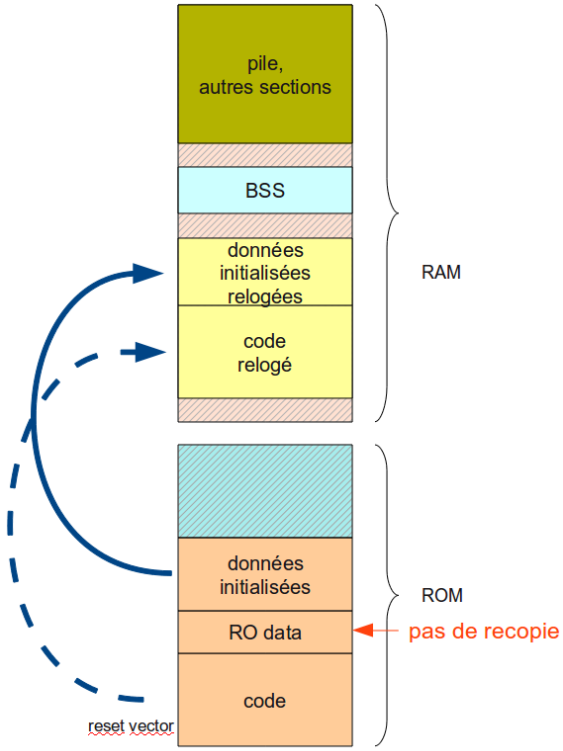
Vie des exécutables : exemple bare-metal

- **Lancement de l'exécutable**
  - stocké en ROM
  - éventuellement, recopie du code en RAM
    - par exécutable externe (bootloader, …)
    - ou par le code lui-même (`crt0.s`, …)
  - recopie des données en RAM par `crt0.s` (inséré par la chaîne de compilation)
  - les données non modifiables sont laissées en ROM
  - création du BSS, de la pile, … par `crt0.s`
  - saut au `main()`



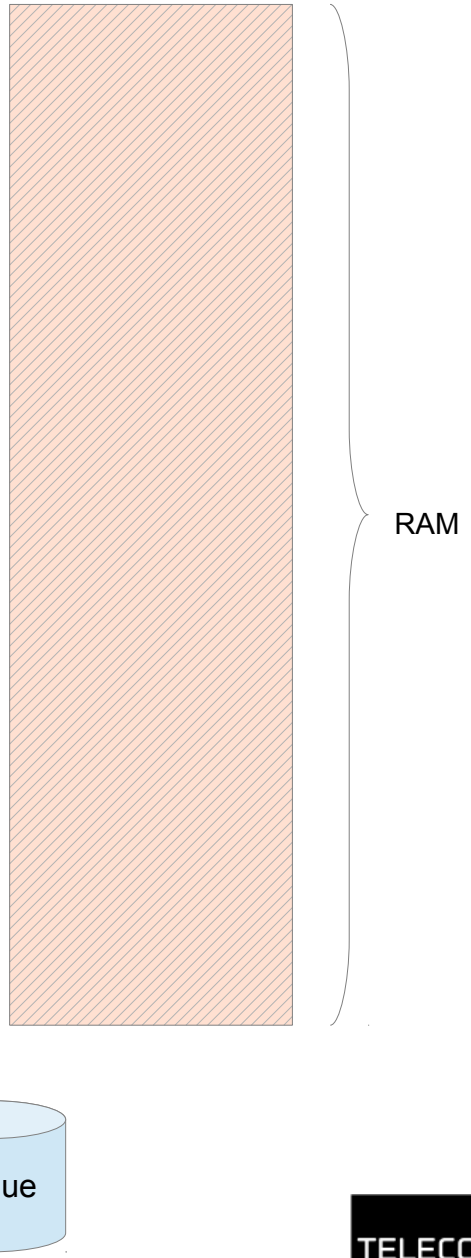
Vie des exécutables

- **Un objet, deux adresses**
  - données relogées
    - une adresse en ROM avant recopie
    - une adresse en RAM après recopie
  - code logé en flash et exécuté en RAM
    - une adresse en ROM avant recopie
    - une adresse en RAM après recopie
  - **adresse avant recopie : LMA**
  - **adresse après recopie : VMA**
  - Attention :
    - aucun rapport avec les adresses virtuelles de la MMU...
    - le code et les variables peuvent donc avoir **deux** adresses !



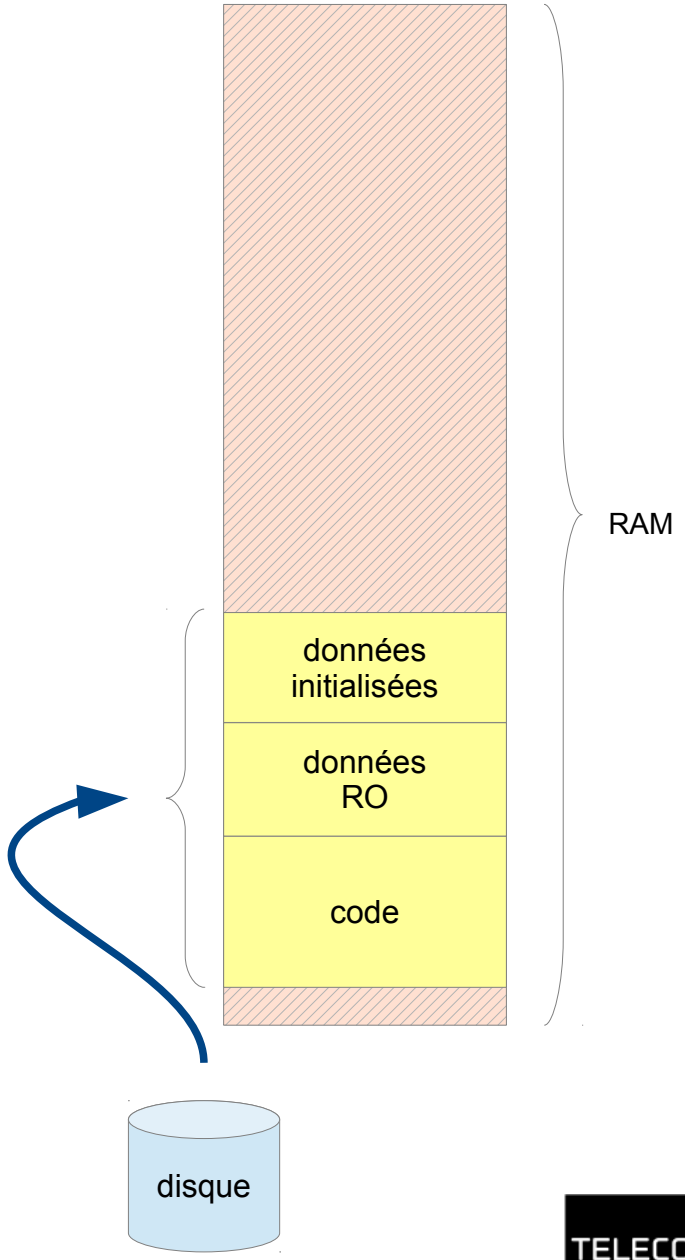
Vie des exécutables : exemple hosted

- **Lancement de l'exécutable**
  - stocké sur disque



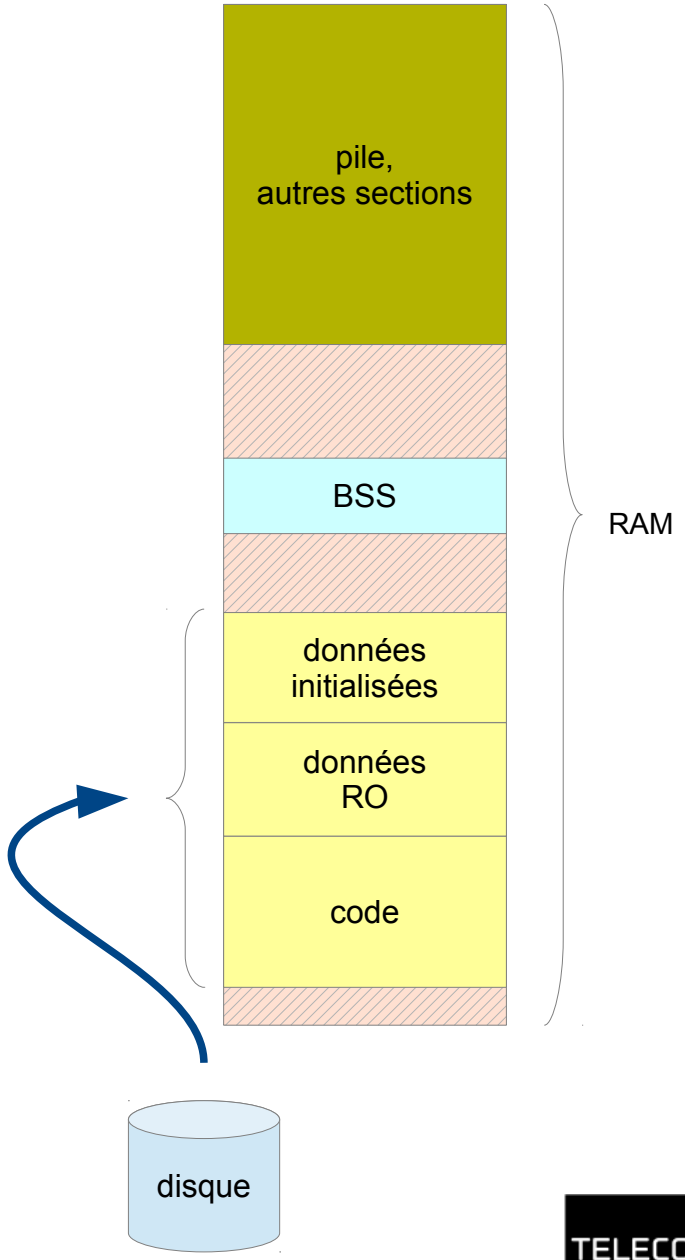
Vie des exécutables : exemple hosted

- **Lancement de l'exécutable**
  - stocké sur disque
  - le loader de l'OS lit un exécutable depuis le disque et le mappe en RAM directement aux bonnes adresses



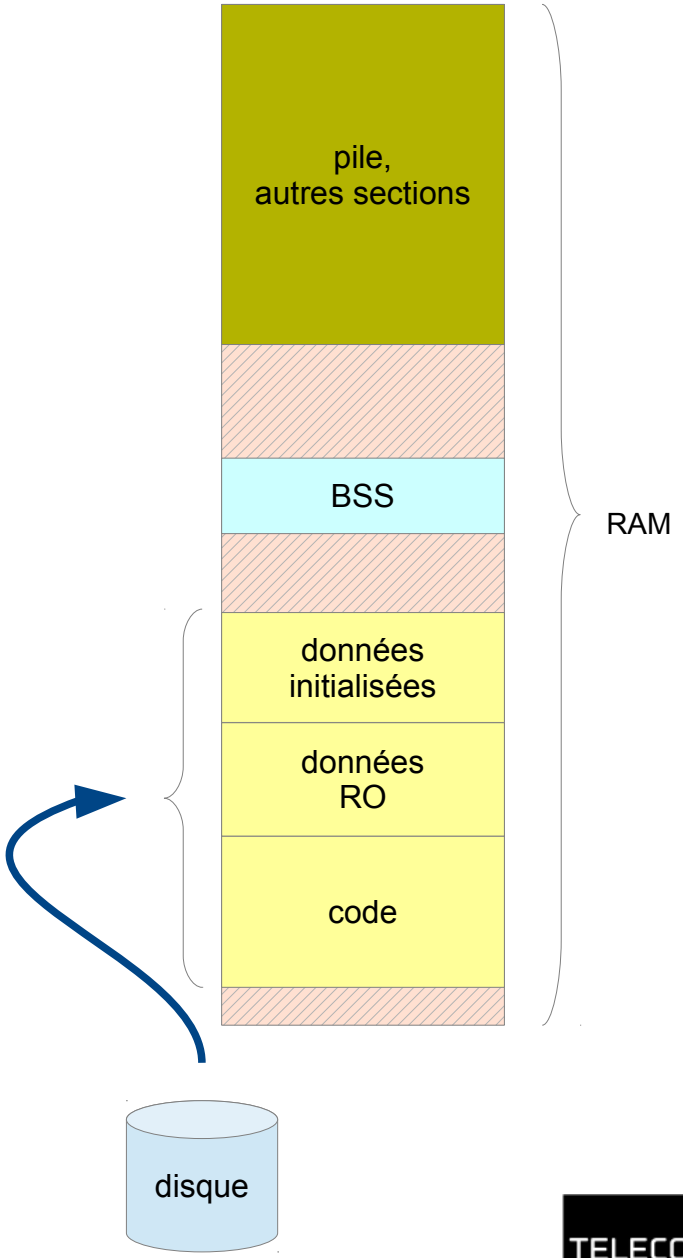
Vie des exécutables : exemple hosted

- **Lancement de l'exécutable**
  - stocké sur disque
  - le loader de l'OS lit un exécutable depuis le disque et le mappe en RAM directement aux bonnes adresses
  - création de la pile, du BSS, …
    - généralement par le loader
    - très rarement par le `crt0.s`



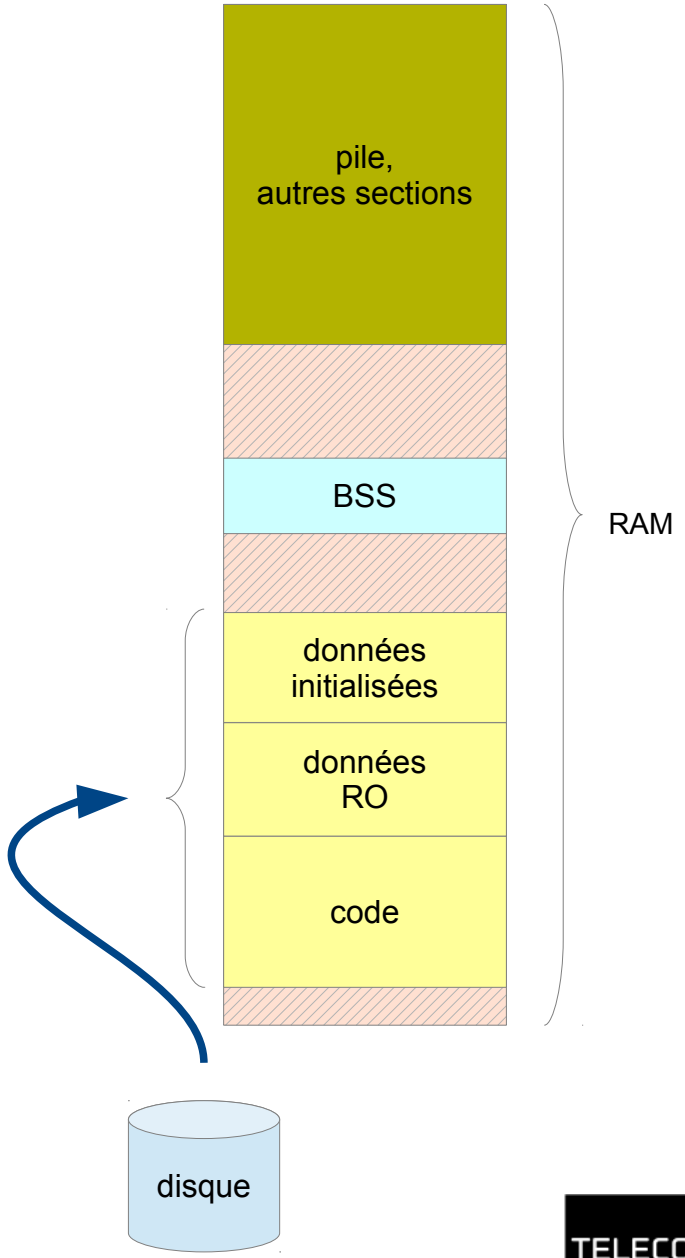
Vie des exécutables : exemple hosted

- **Lancement de l'exécutable**
  - stocké sur disque
  - le loader de l'OS lit un exécutable depuis le disque et le mappe en RAM directement aux bonnes adresses
  - création de la pile, du BSS, …
    - généralement par le loader
    - très rarement par le `crt0.s`
  - saut au `main()`



Vie des exécutables : exemple hosted

- **Un objet, une adresse**
  - la MMU permet de mapper le fichier en mémoire
  - aucune recopie nécessaire
  - dans ce cas, LMA = VMA
  - généralement, la MMU met aussi en place des protections lors du mapping
  - cas particulier : activation de la MMU
    - avant activation : code exécuté depuis sa LMA
    - après activation : code exécuté depuis sa VMA



Pause question piège

- **Attention**
  - dans les deux exemples précédents, on a sous-entendu un fait qui n'est pas forcément vérifié : lequel ?

(indice : on l'a vu en début du cours...)

Où en est-on ?

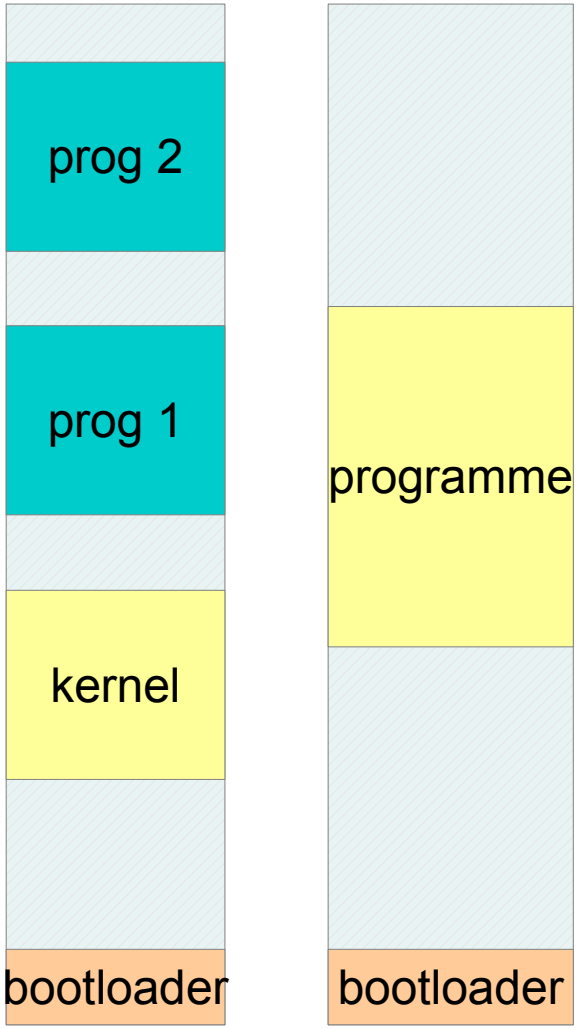
- **Les systèmes à processeur**
  - architecture, mapping mémoire
  - modes d'exécution et exceptions
  - types de systèmes : bare metal vs. hosted
  - vie des exécutables
  - bootloaders
  - debug

Boot et bootloaders

- **Séparation des fonctionnalités**
  - initialisation → bootloader
  - fonctionnement → exécutable
  - debug → moniteur ou debugger
    - moniteur : reprise de contrôle en cas d'exception
    - debugger : fonctions avancées

Boot et bootloaders

- **Bootloaders**
  - premier programme à s'exécuter
  - initialise le matériel
  - prépare un environnement d'exécution correct
  - lance l'exécutable principal
  - éventuellement sur condition
  - peut permettre la mise à jour du programme principal

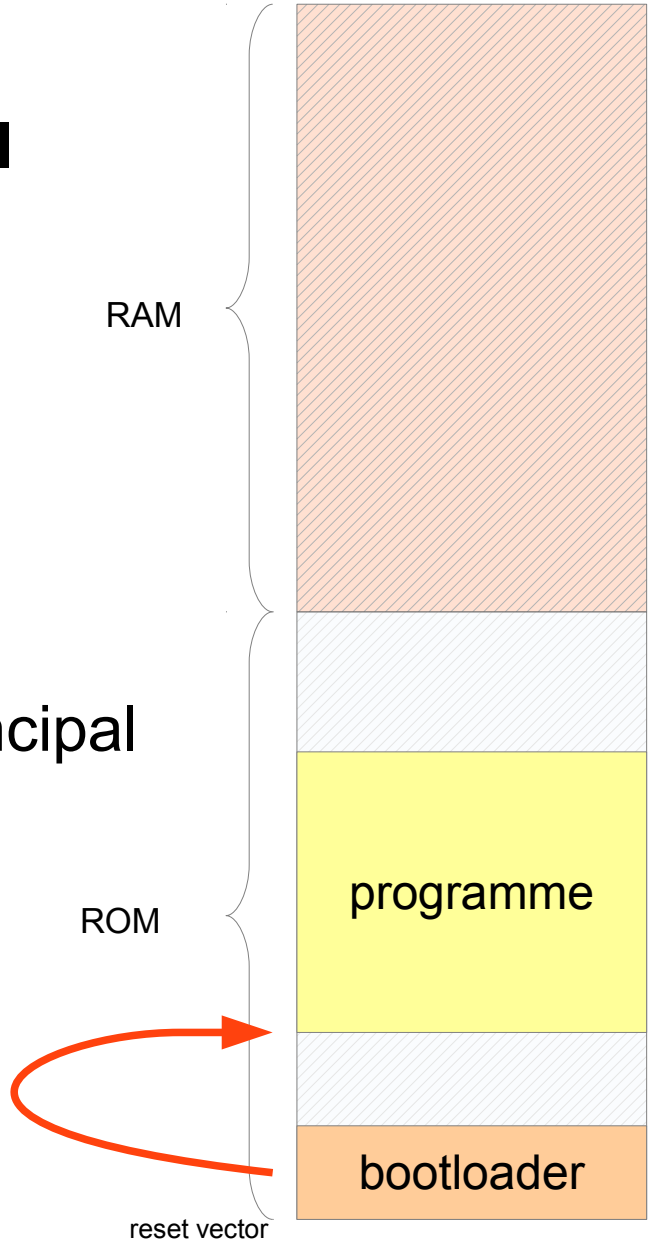


Boot et bootloaders

- **Initialisation typique d'un système**
  - configuration du processeur :
    - désactivation des interruptions
    - initialisation des registres de contrôle du processeur
  - initialisation de la RAM
    - configuration du contrôleur RAM
    - mise en place éventuelle d'environnement d'exécution spécifique au langage utilisé (C, Forth, ...) : pile(s), BSS, ...
  - mise en place des handlers d'interruptions critiques (Sparc)
  - initialisation des périphériques critiques
    - PLL
    - énumération des bus critiques (PCI, AGP, HT, ...)
    - IOMMU, contrôleurs de bus
    - contrôleurs de stockage de masse
  - activation des caches
  - configuration MMU, passage en mode virtuel
  - activation des interruptions critiques

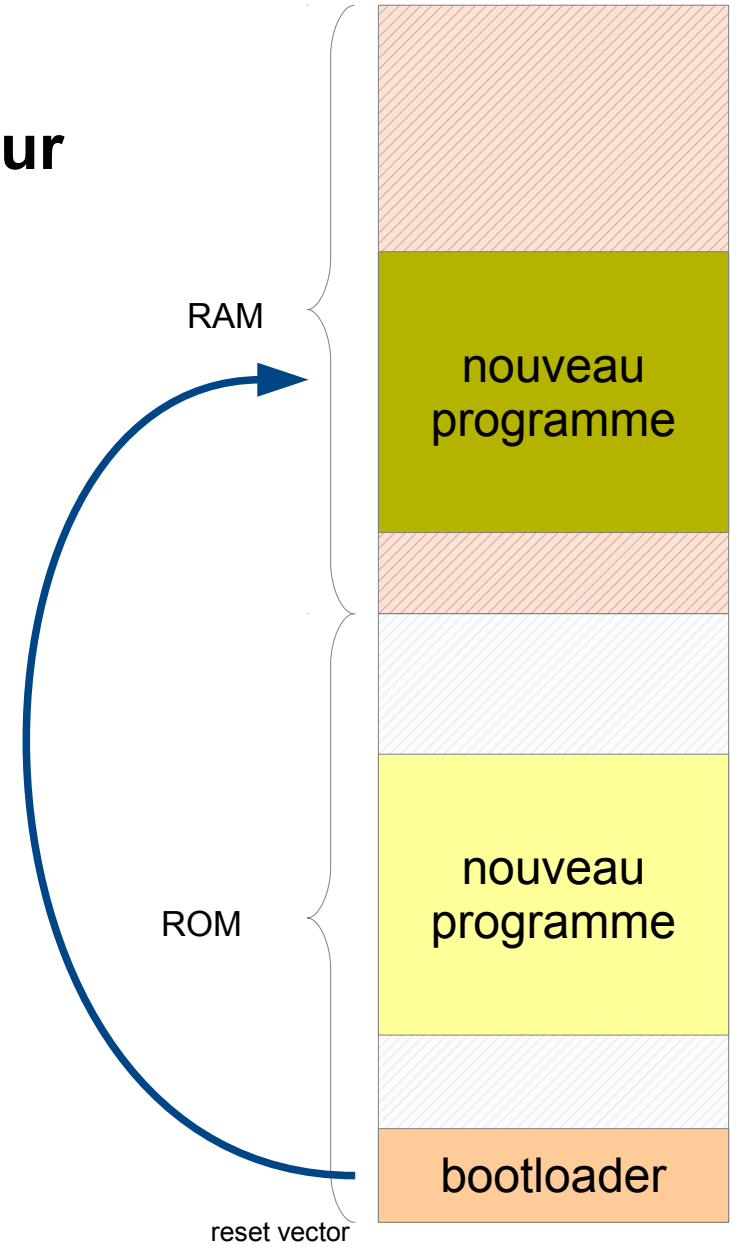
Boot et bootloaders

- **Exemple de fonctionnement : boot normal**
  - lancement du bootloader
  - initialisation du système
  - attente d'une condition
    - caractère sur port série,
    - bouton poussoir,
    - …
  - transfert de l'exécution au programme principal



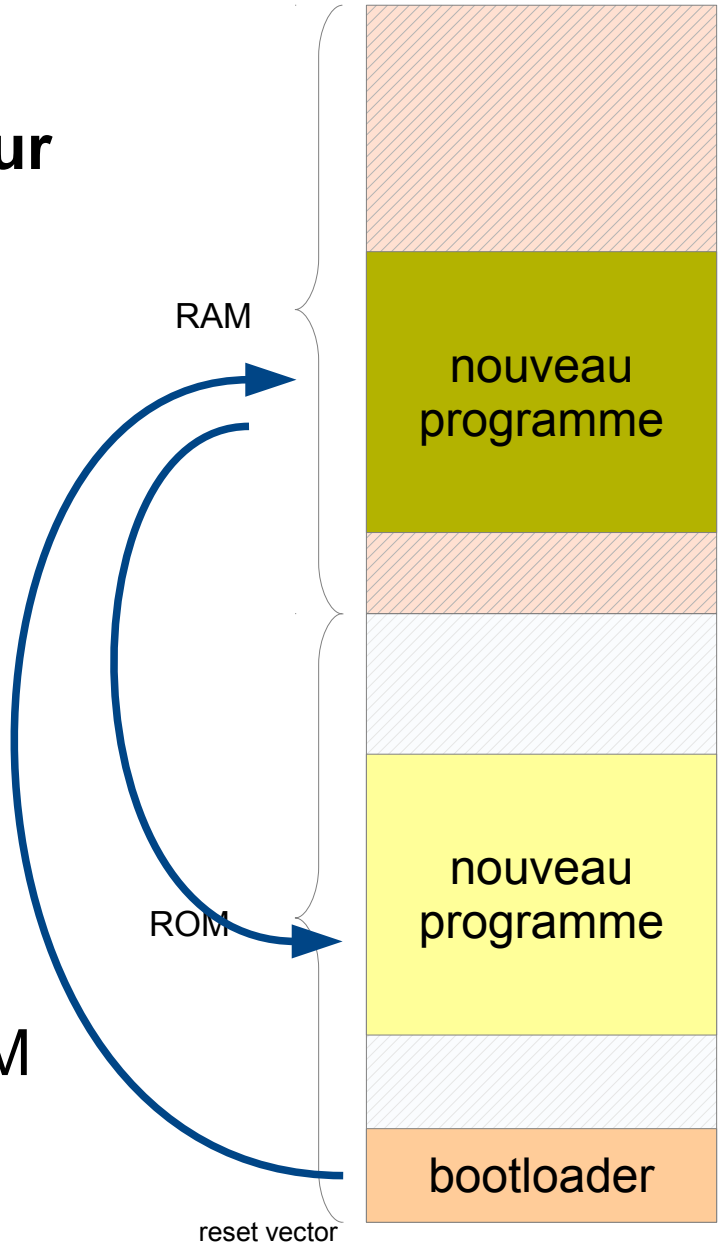
Boot et bootloaders

- **Exemple de fonctionnement : mise à jour**
  - lancement du bootloader
  - initialisation du système
  - attente d'une condition
    - caractère sur port série,
    - bouton poussoir,
    - …
  - récupération du nouveau programme
    - port série,
    - NFS, TFTP, …
    - …



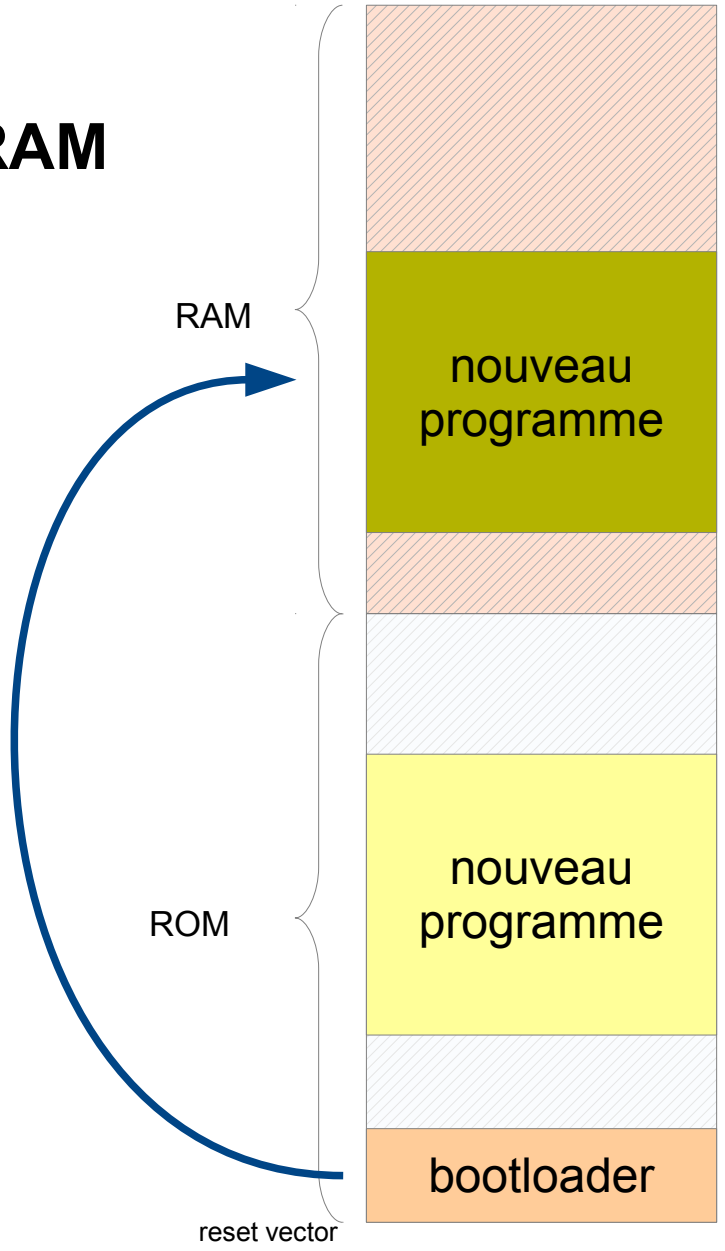
Boot et bootloaders

- **Exemple de fonctionnement : mise à jour**
  - lancement du bootloader
  - initialisation du système
  - attente d'une condition
    - caractère sur port série,
    - bouton poussoir,
    - …
  - récupération du nouveau programme
    - port série,
    - NFS, TFTP, …
    - …
  - recopie du nouveau programme en RAM vers l'ancien en ROM / flash



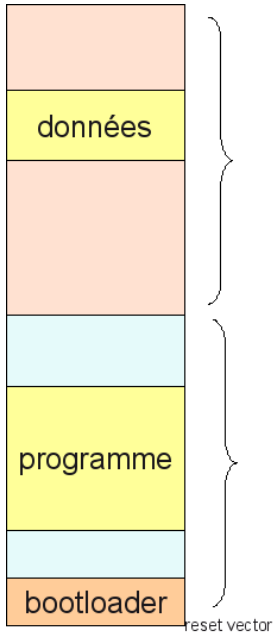
Boot et bootloaders

- **Exemple de fonctionnement : boot en RAM**
  - lancement du bootloader
  - initialisation du système
  - attente d'une condition
    - caractère sur port série,
    - bouton poussoir,
    - …
  - récupération du nouveau programme
    - port série,
    - NFS,
    - …
  - transfert de l'exécution au programme en RAM

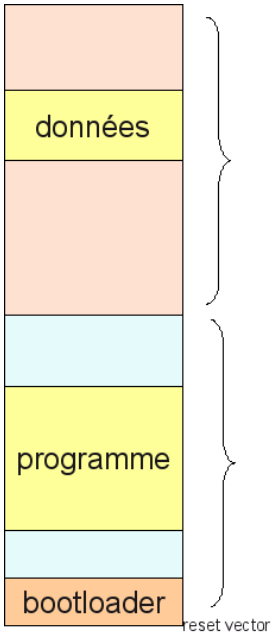


Boot et bootloaders

- **Fonctions additionnelles**
  - passage d'arguments / structures de données
  - services à l'exécution (syscalls)
  - fonctions de debug
    - diagnostic
    - examen et manipulation de la mémoire
    - reprise de contrôle sur exception : moniteur

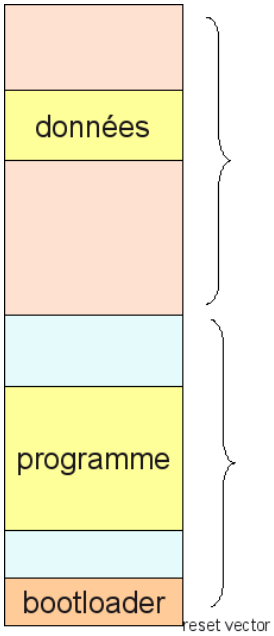


Boot et bootloaders



- **Boot complexes**
  - Un démarrage complexe peut être effectué en plusieurs étapes
  - exemple : PC avec GRUB
    - BIOS :
      - exécution commence en 0xFFFF0000 (BIOS)
      - POST
      - chargement du MBR (finit par 0xAA55) et exécution en 0x7C00
    - MBR (GRUB stage 1) :
      - 512 octets (440 utiles)
      - va chercher un exécutable un peu plus complet (GRUB stage 1.5) à un endroit spécifique du disque, le charge et l'exécute
    - GRUB stage 1.5 :
      - sait lire les systèmes de fichiers
      - charge GRUB stage 2 depuis le système de fichier prévu et le lance
    - GRUB stage 2 :
      - va chercher ses fichiers configuration
      - affiche éventuellement une interface utilisateur
      - va chercher un noyau ou un autre bootloader (généralement propriétaire), le charge en mémoire et l'exécute.

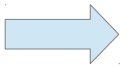
Boot et bootloaders



- **Exemples de bootloaders**
  - BIOS des PC
    - complexe, fournit des services basique (INT)
    - ne sait que lire un MBR et l'exécuter
  - U-Boot
    - dédié à l'embarqué
    - configurable et multi-plateforme
    - support RS232 et réseau (Ethernet)
    - permet le reflashage et l'exécution au vol
  - OpenFirmware (IEEE-1275)
    - interpréteur de bytecode basé sur Forth
    - utilisé par Sun, Apple, IBM et la plupart des chipsets non x86
  - EFI / UEFI

Où en est-on ?

- **Les systèmes à processeur**
  - architecture, mapping mémoire
  - modes d'exécution et exceptions
  - types de systèmes : bare metal vs. hosted
  - vie des exécutables
  - bootloaders
  - debug



- **Pourquoi débugger ?**
  - Règle : un processeur ne "plante" pas !
  - Un processeur :
    - exécute l'instruction située en PC
    - est bloqué en mode erreur, généralement après une double faute (ARM, SPARC, …)
  - En cas de problème, c'est le programme que vous lui avez donné à exécuter qui ne correspond pas à ce que vous avez en tête.

1er commandement de l'UE

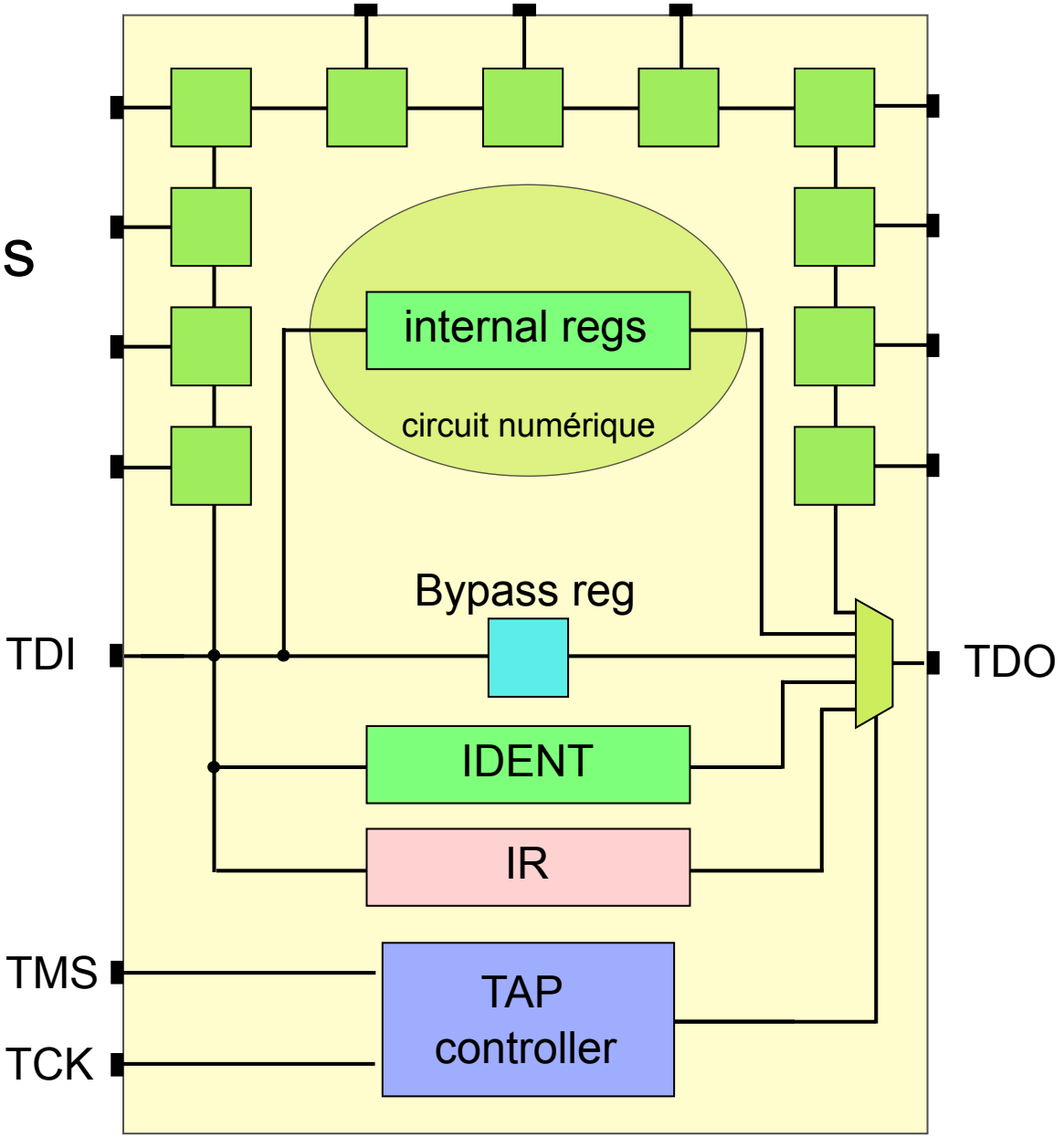
- À partir de maintenant, un processeur **ne plante plus**.
- L'emploi d'un débugger **au niveau assembleur** pour comprendre ce qui s'est passé doit être un **réflexe** :
  - exécution pas à pas (en assembleur)
  - examen des registres à chaque step
  - examen de la mémoire à chaque step

Débug

- **Debugger**
  - débug local :
    - le debugger s'exécute nativement sur le système
    - utilisation des ressources du système pour interagir avec l'utilisateur
    - débug des applications : ok
    - débug de l'OS : difficile
  - débug distant
    - le debugger s'exécute sur une machine hôte
    - communication avec le système à débugger (cible)
      - avec coopération : RS232, Ethernet (stubs, gdbserver, moniteur)
      - sans coopération : JTAG, SWD, BDM, 1Wire, AUD/HUDI, ...

Débug

- **JTAG**
  - protocole basé sur des registres à décalage
  - interface JTAG : TAP
  - contrôleur TAP
    - machine à 16 états
  - 4 ou 5 signaux
    - TDI
    - TDO
    - TMS
    - TCK
    - [TRST]

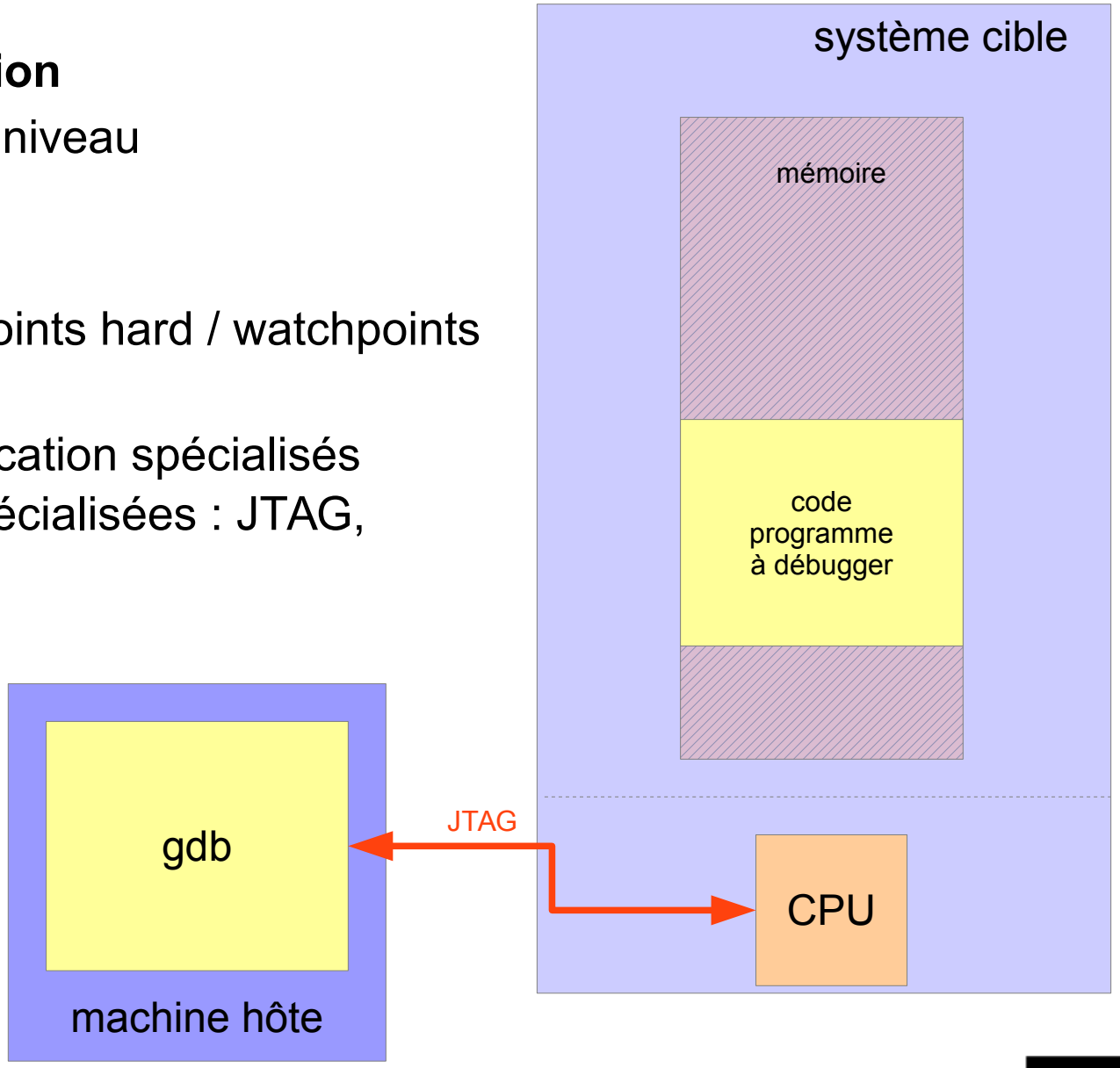


Débug

- **Breakpoints**
  - logiciels :
    - remplacement d'une instruction assembleur par une TRAP
    - handler spécial pour cette TRAP → donne la main à gdb
    - remise en place de l'instruction originelle
    - quizz : peut-on mettre des breakpoints en flash ?
  - matériel :
    - comparateur sur PC / data / addr / signaux de contrôle
    - match : arrêt du processeur et communication en JTAG des informations qui ont déclenché l'arrêt
    - inconvénient : ressource limitée !

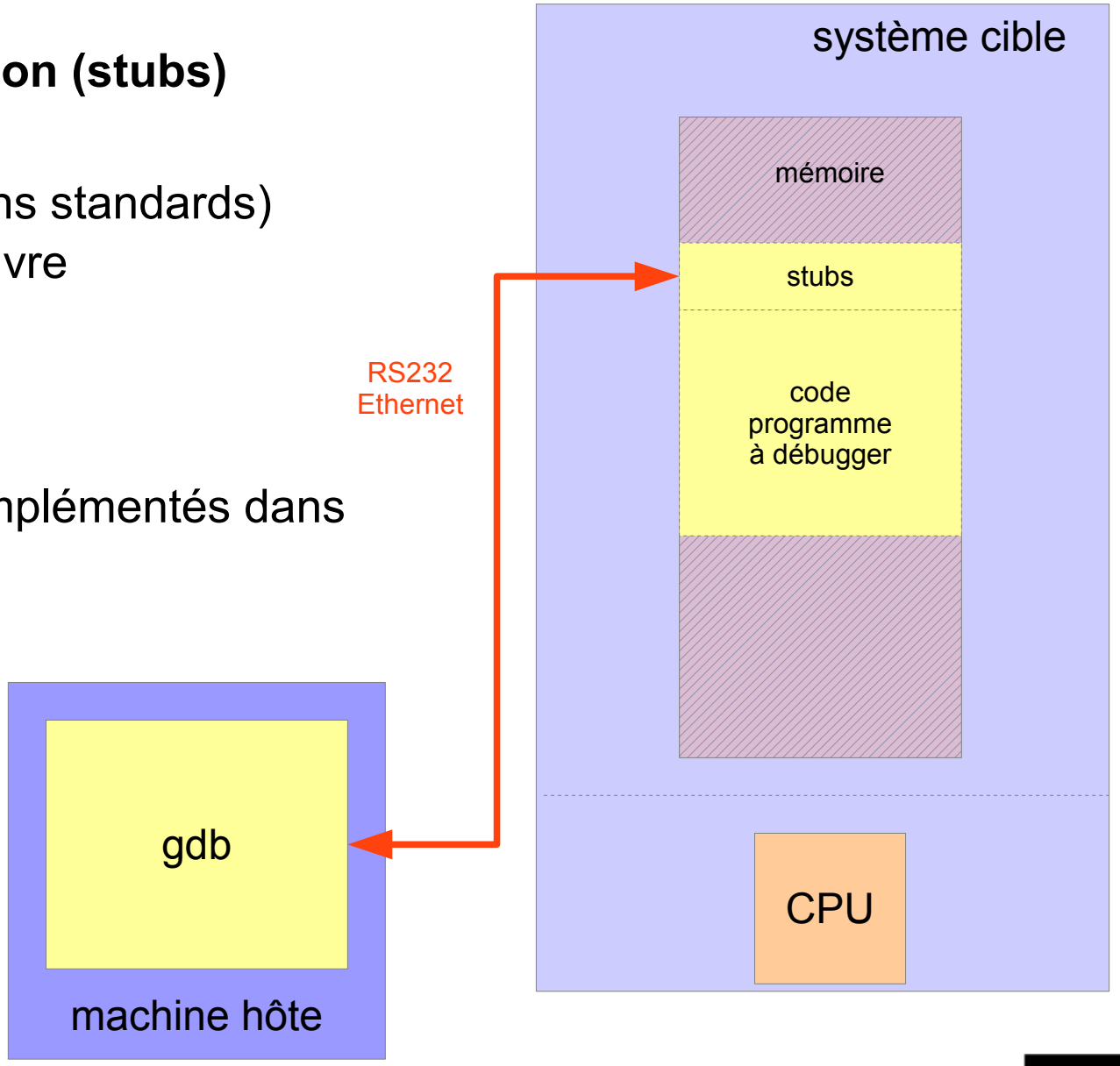
Debug

- **Carte nue, sans coopération**
  - communication très bas niveau
  - avantages :
    - non intrusif
    - possibilité de breakpoints hard / watchpoints
  - inconvénients :
    - moyens de communication spécialisés
    - besoin de sondes spécialisées : JTAG, SWD, …



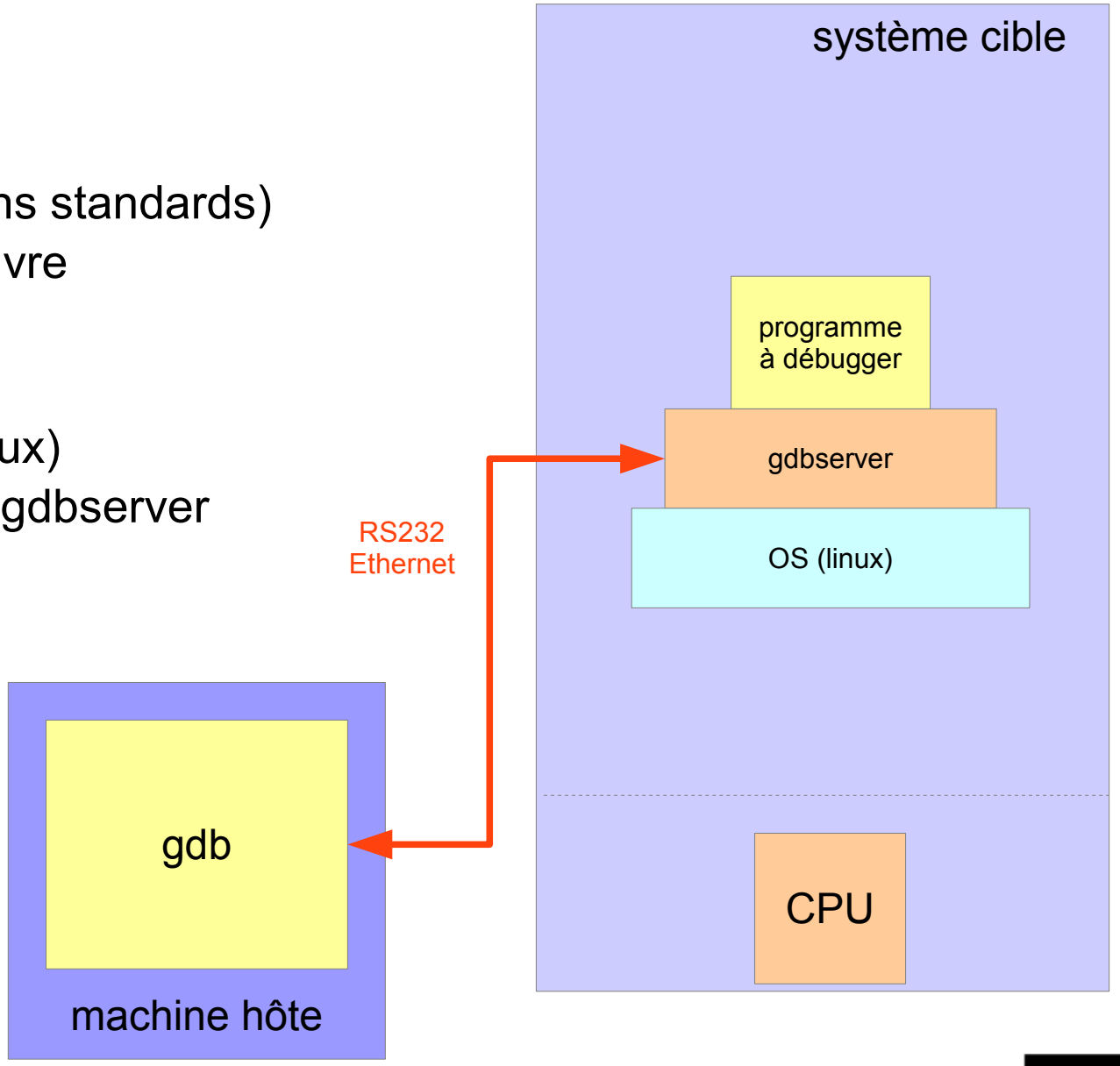
Debug

- **Carte nue, avec coopération (stubs)**
  - avantages :
    - très peu onéreux (liens standards)
    - facile à mettre en œuvre
  - inconvénients :
    - intrusif : stubs
    - code en RAM
  - les stubs peuvent être implémentés dans un moniteur



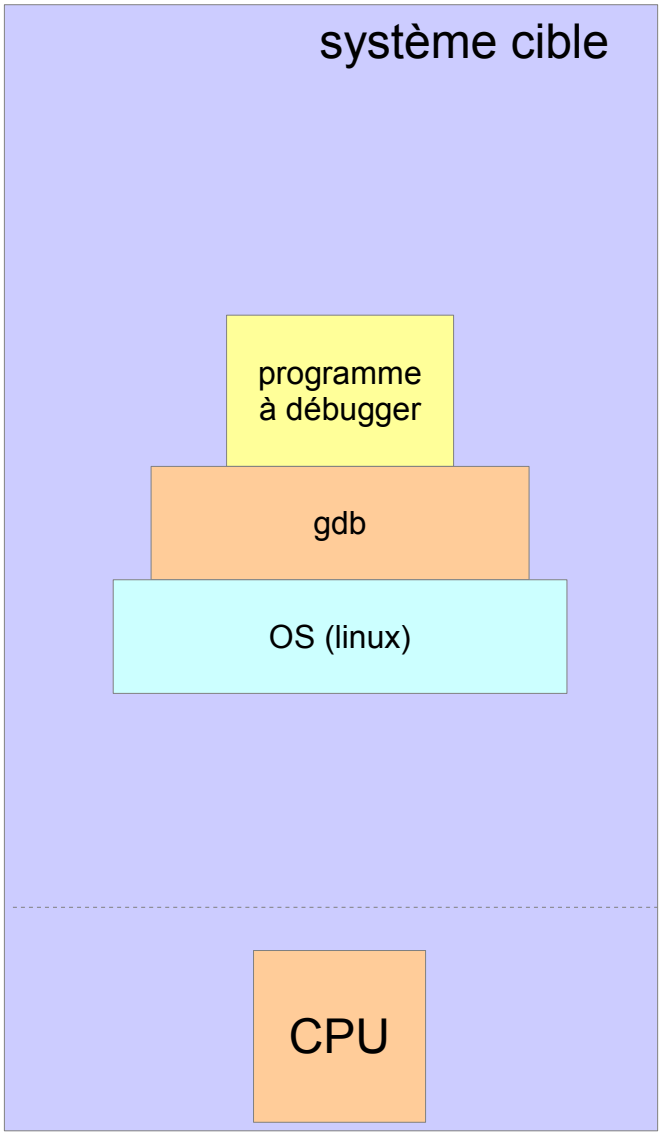
Debug

- **OS : débug distant**
  - avantages :
    - très peu onéreux (liens standards)
    - facile à mettre en œuvre
    - peu intrusif
  - inconvénients :
    - nécessite un OS (Linux)
    - moniteur spécialisé : gdbserver

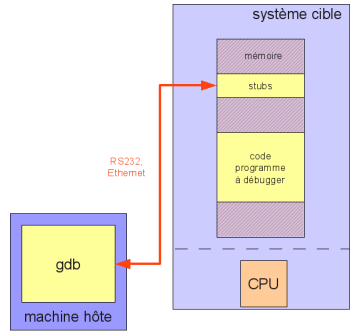


Debug

- **OS : débug natif**
  - avantages :
    - ne coûte rien
    - facile à mettre en œuvre
    - peu intrusif
  - inconvénients :
    - gourmand...
    - nécessite un OS (Linux) et une IHM

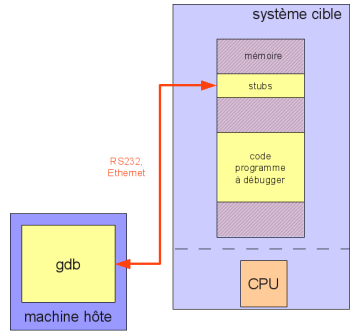


Debug



- **Stubs**
  - contiennent :
    - handlers spécialisés
    - routines de communication avec gdb
  - « linkés » avec le programme à débugger
  - fournis pour la plupart des processeurs
  - nécessitent souvent des modifications (mineures)

Debug



- **Stubs**
  - nécessitent les fonctions suivantes :

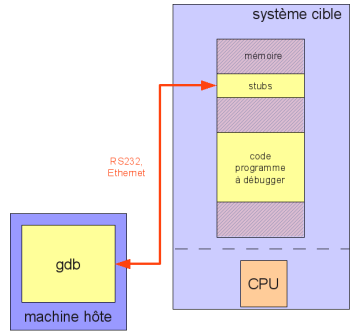
```
/* Fonctions à fournir aux stubs */
int getDebugChar();
void putDebugChar(int);
void exceptionHandler (int exception_number, void *exception_address);
void flush_i_cache(); // sur SPARC seulement
```

  - fournissent les fonctions suivantes :

```
/* Fonctions fournies par les stubs */
void set_debug_traps();
void breakpoint();
```

Debug

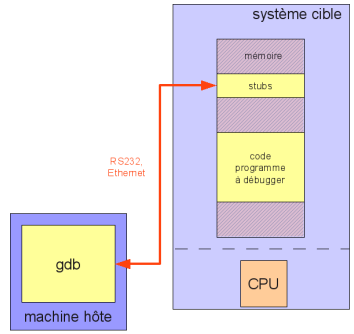
- **Stubs : exemple d'utilisation**




```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    // setup stubs
    set_debug_traps();
    breakpoint();

    // do real work
    printf("Hello world!\n");

    return EXIT_SUCCESS;
}
```

```
CFLAGS += -g

hello : stubs.o hello.o
```

Debug



- **Perdu dans GDB ?**
  - GDB quick reference card
    - à récupérer sur le site de GDB uniquement !
    - https://sourceware.org/gdb/download/onlinedocs/refcard.pdf.gz
  - Documentation officielle de GDB
    - https://sourceware.org/gdb/download/onlinedocs/gdb/index.html
  - Documentation officielle du mode TUI de GDB
    - https://sourceware.org/gdb/download/onlinedocs/gdb/TUI.html
  - Manuel de survie GDB
    - https://sen.enst.fr/elec223/guide-de-survie-gdb

Licence de droits d'usage

Contexte académique } sans modification

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après, et à l'exclusion de toute utilisation commerciale.

Le droit d'usage défini par la licence autorise un usage dans un cadre académique, par un utilisateur donnant des cours dans un établissement d'enseignement secondaire ou supérieur et à l'exclusion expresse des formations commerciales et notamment de formation continue. Ce droit comprend :

- le droit de reproduire tout ou partie du document sur support informatique ou papier,
- le droit de diffuser tout ou partie du document à destination des élèves ou étudiants.

Aucune modification du document dans son contenu, sa forme ou sa présentation n'est autorisé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel, non exclusif et non transmissible.

Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur :

alexis.polti@telecom-paristech.fr