Chaîne de développement : Compilation croisée

- Principe d'une chaîne de développement
- Dépendances entre processus de « production » et architecture opérationnelle
- Compilation croisée, MDE et génération de code
- Déploiement et analyse

Objectifs du cours

- **Chaîne de développement et position du compilateur dans la chaîne**
- **Architecture en couche d'un Système Embarqué** (bibliothèques de programmation et support d'exécution)
- **Production du système et compilation croisée**
- **Test, Validation et Vérification du système déployé**
- **Problématiques de déploiement**
- **Bilan**

Chaîne de développement et Compilation

Processus de développement et outils ...

- **Rôle des outils :**
  Supporter le processus de développement
- **Les tâches du processus :**
  - Spécifier / vérifier
  - Développer / mettre au point
  - Intégrer / tester & valider
  - Déployer / maintenir et observer.

- **Remarque : la nature et le déroulement des tâches dépend du cycle de développement retenu**

Processus de développement pour applications « bureautiques »

- **On développe souvent la où l'on exécute !**
- **Quasiment aucun problème de dimensionnement**

  (Mythe des ressources infinies)

  => spécification fonctionnelle == documentation

  => déploiement à coût ~ 0

  => Validation du produit par des outils tiers lancés en parallèle sur l'OS (gdb, valgrind, …)
- Ce qui compte en embarqué temps réel :
  - Dimensionnement strict (e.g. temporel)
  - Il faut développer et compiler sur A et exécuter sur B

La structure usuelle d'un environnement de développement

- **5 composants élémentaires :**
  - Un éditeur pour la rédaction de la spécification
  - Un éditeur pour le développement
  - Une chaîne de compilation
  - Un outil de test et/ou déboggage
  - Un outil de déploiement
- **En pratique : Modularité & configuration**
  - Des front-end uniformes (IDE, ou M2M processor) (Eclipse, emacs .... Rapid™)
  - Des back-end spécifiques au matériel (ld, ar, binutils)

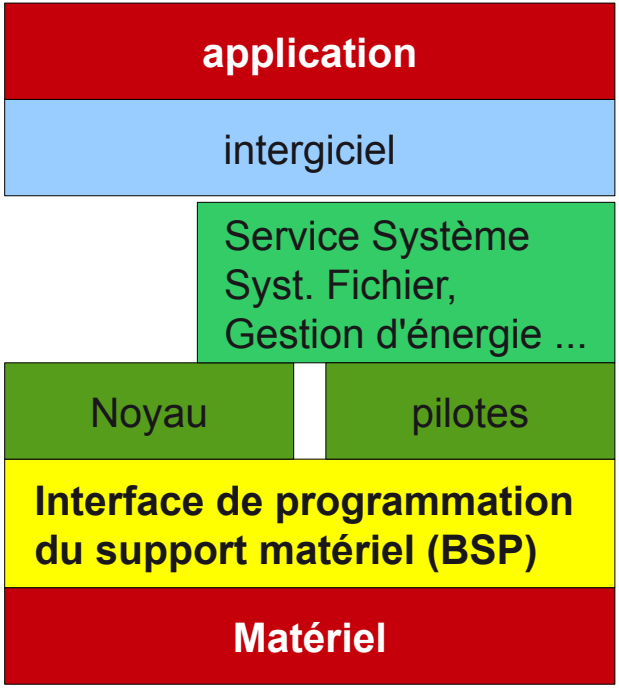
Architecture des Systèmes Embarqués et Chaîne de production

Vue macroscopique (rappel)

optionnel

obligatoire

A chacun son API et ses modules !!



Commande / contrôle d'un véhicule … logique de calcul

RT-corba

sshd, ftp, file systems

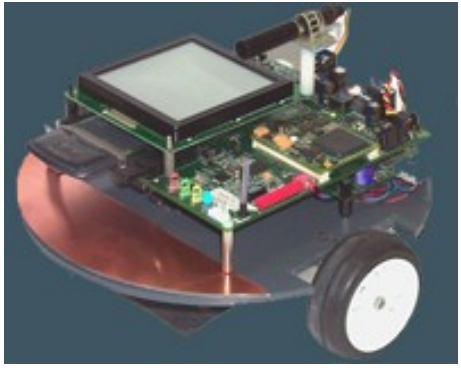
Interface % services « coeur » OS (API RTEMS, API POSIX, « device »)

Interface % services matériels (Changement de contexte, IT...)

MPC680

Un exemple de système embarqué …

- **Calculateur pour SCADA (supervisory control and data acquisition) : SPIF (Telecom ParisTech)**
  - Entrées/Sorties :
    9 types différents
    mais pas de clavier ....
  - CPU :
    MPC860, 64Mb RAM
  - Stockage :
    8 Mb ROM
  - Utilisation : Contrôle d'un robot mobile
  - Programmation : RTEMS & Linux API



Rq : ROM plus petite que la RAM, fait exprès pour les gros calculs, traitement du signal, traitement d'image pour les robots, etc...

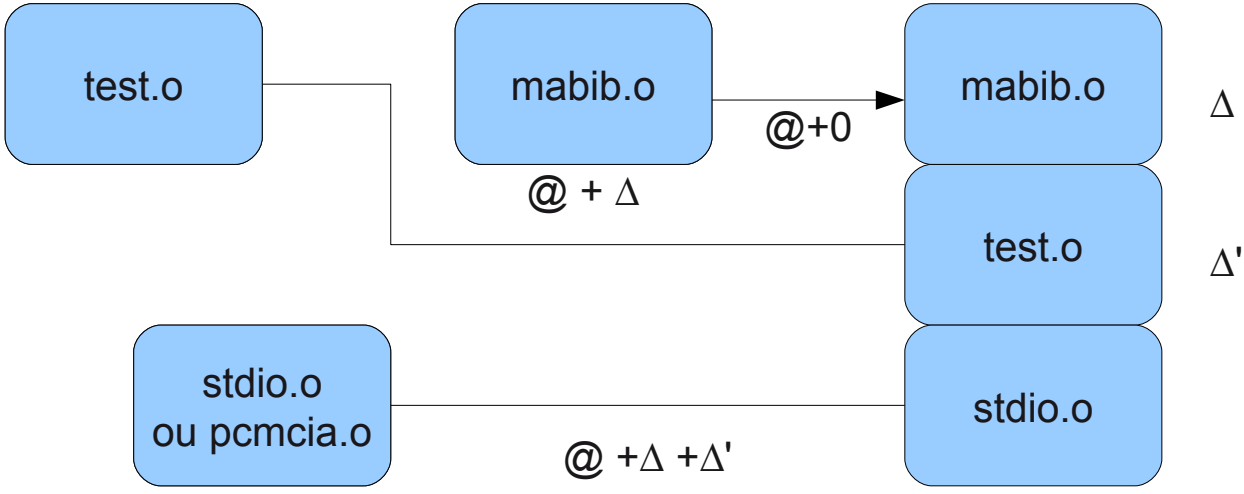
Fonction et déroulement de la compilation

Etapes du processus de compilation

pmap : function qui permet de connaitre le schema d'exécution pour un binaire

Rappel sur l'édition de lien et la compilation séparée

- **Permet d'assembler une application par modules:**
  (interface de prog. du matériel, de l'OS, du middleware)



Le principe reste le même : l'exécutable est un patchwork de bloc **MAIS** il faut qu'ils soient tous exécutables sur la cible....

Edition de liens – bilan

- **Contenu des .o = code binaire**
  - Dépend du processeur
  - Dépend du compilateur (plusieurs formats e.g. ELF)
- **Les modules de la compilation séparée**
  - Modules applicatifs (conception modulaire)
  - Les bibliothèques de programmation (Matériel, OS, middleware).
- **Ce qui est fait :** Juxtaposition des différents modules et transformations des adresses de références
- **Le résultat :** un fichier contenant un binaire complet exécutable

Interfaces de programmation et environnement de compilation

- **L'environnement de compilation**
  => le compilateur (code source & support d'exec)
  => **l'accès** aux modules implémentant les interfaces
- **Les interfaces de programmation imposent un format de fichiers objets (binaires linkables)**
  - API d'interaction avec le matériel (drivers)
  - API du système d'exploitation (appels systèmes)
  - API de services de + haut niveau

  => L'intégration de ces éléments dépend du matériel
- **La chaîne de développement doit aider le processus d'intégration de l'application sur le support d'exécution**

La vision MDE, Modèle de composants & langages dédiés

- **Idée : Décrire l'implémentation avec des modèles**
  - AADL
  - Signal, Lustre
- **Valider le comportement à travers des raisonnements logiques explicites**
  - Modèles de tâches et Ordonnancement
  - Flots de données et simulation numériques
- **Outils associés aux langages dédiés**
  - Mathlab/Simulink
  - Sigali Polychroni (SIGNAL)

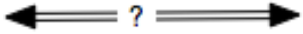
Lien entre modéles validés et le produit final

- **Les modèles == entrée des chaines de production**
- **Architectures à base de composants avec interfaces explicites**

  pb : il faut obtenir le code implémentant le comportement des modèles
- **Chaîne de production fiable == production automatique de l'application finale à partir des modèles**
  - AADL → Ada Ravenscar via Ocarina. (intégration)
  - LUSTRE → C via LUSTRE V4 + LESAR tool box (comportement complet)

La compilation croisée et BSP (lib)

Développer sur A, exécuter sur B ?



Développement
Compilation
Pilotage de tests

Exécution opérationnelle
Déboggage
Exécution de tests

Compilation croisée : le compilateur

- **Ses contraintes :**
  - S'exécute sur A et Génère du langage machine pour B
  - Doit identifier l'emplacement des bibliothèques pour B
  - Doit construire une « image » compatible aux capacités de B (communication, stockage, structure ...)
- **En pratique**
  - Pour la production du binaire: le type de processeur et la méthode d'encodage comptent énormément.
  - Le résultat de la compilation est indépendant de A
  - La manipulation des fichiers binaires est spécifique à B

Compilation croisée : les bibliothèques

C'EST LE COEUR DU PROBLEME

- **Dépendances & contraintes multiples :**
  - Dépendances fonctionnelles API ↔ API
  - Dépendances au matériel (assembleur, ressources)
  - Pb de dimensionnement
- **Conseils :**
  - Privilégier le code source pour les couches hautes
  - Limiter l'usage de l'assembleur spécifique dans les couches les + basses (e.g. Le BSP)
  - Garder la trace de la cible et des compilateurs utilisés

Contenu et rôle du BSP (board support package)

- **Objectif : fournir une interface entre OS et matériel**
  - Permettre à l'OS d'utiliser les ressources matérielles (interruptions, registres, MMU, ports matériels...)
  - Concentrer la majorité du code ultra spécifique dépendant du matériel utilisé (code assembleur, accès aux registres, à l'horloge)
- **Intégration OS / BSP : Hardware Abstract Layer**

  interface visant à uniformiser l'accès au matériel dans un Noyau
  - L'OS définit son propre HAL « à remplir »
  - Implémenter un HAL standard (utilisé par l'OS).

BSP et RTEMS

- **RTEMS a son propre HAL pour lequel il faut remplir les squelettes**
  - Chaque fonction a une spécification informelle avec une signature
  - Des exemples d'implémentation sont disponibles pour différents matériels (ex. Guide RTEMS-BSP)
- **HAL modulaire % architecture matérielle classique**
  - Changement de contexte, timers, I/O (manipulation des registres, et interruptions)
  - Port série, réseau ...

A retenir Chapitre I, II, III

- **La compilation dans 1 processus de production**
- **Des outils réutilisables mais complexes**
  => notion d'environnement de compilation (= conf)
  => Approche MDE et génération de code

- **Etapes de constitution de l'environnement :**
  1) obtenir la chaîne de production pour le processeur
  2) S'interfacer avec le système pour déployer le logiciel
  COMMENT ?

Déploiement, Test et Mise au point

Chargement et interfaces

- **Interfaces de chargement**
  - Réseaux
  - Mécanique (intégration stockage de masse)
- **Pb : Quand ces interfaces sont elles actives ?**
  - Activation matérielle
  - Activation logicielle

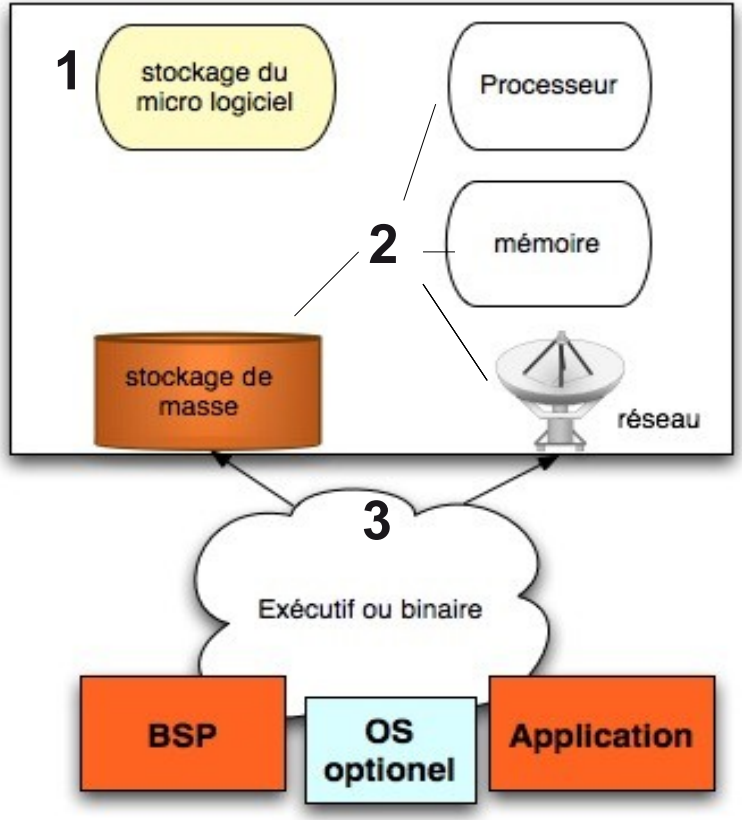
IL FAUT COMPRENDRE COMMENT UN SYSTEME SE LANCE

Mode interactif vs automatique

- **Chargement hors-ligne vs en-ligne :**
  - Ex 1 : carte flash physiquement déplacée
  - Ex 2 : connexion réseau + protocole de chargement (TP) avec attente
  - Type de la mémoire du support de stockage (ROM/RAM/stockage de masse)
- **Les deux doivent être supportés car cela correspond aux deux phases du développement**
  - Mise au point
  - Vie opérationnelle

Comment un calculateur démarre ? (et justification du terme « bootstrap »)

- **Exécution du micro-logiciel et moniteur**
  - Test
  - Mise en service des périphériques
  - Amorçage depuis le stockage de masse (SM) ou réseau (R)
- **Exécution de l'amorce**
  - Configuration des couches du Syst. Logiciel
  - Chargement du noyau / de l'applcationi



4 : ça marche

Comment un système démarre : le moniteur

- **Rôle :**

  1) Diagnostic et initialisation des fonctions du matériel

  2) interface basique de commande du matériel

  dont fonctions de chargement et exécution de code (placé en mémoire)

- **Implémentation (dépend du matériel)**
  http://penguinppc.org/embedded/howto/rom-monitor.html
- **Les fonctions usuelles en plus :**
  - **Protocole de chargement réseau ou série**
  - **Mécanismes d'observation / debug et audit**
- **Contraintes : doit tenir sur la ROM du matériel.**

Bootloader

- **Rôle : assure le passage entre le micrologiciel embarqué et l'OS**
- **Etapes :**
  - Chargement de systèmes de fichier + élaborés
  - Chargement de fonctions d'audit (par hyperviseur)
  - Configuration dynamique et initialisation de l'OS (laisse la main à l'OS à partir de là)
- **Principe : on y met tout ce que l'on ne peut mettre dans le micrologiciel et qui sert à lancer l'OS.**
- **En pratique:**
  fusion du micrologiciel et du bootloader (ex PPCBOOT)

Exemple RTEMS + MPC860 (PPCBOOT)

- **1 Distribution RTEMS**
- **1 BSP pour MPC860 standard**
- **1 rack de cartes MPC860 (accessible depuis Xuri)**
- **Méthode de chargement : réseau**
  liaison série-ethernet
- **Protocole de communication Hôte / cible :**
  synchrone-asynchrone

Dimensionnement et réflexion globale

Dimensionner, c'est quoi ?

- **Comprendre les contraintes matérielles :**
  - Mémoire /CPU
  - Temps de réponse
  - Autres .... horloges, bus de communication..
- **Définir à partir des exigences fonctionnelles, des exigences de ressources**
  - L'asservissement d'un moteur par une application doit s'exécuter à une période de 10ms
    => une tache périodique à 10 ms
  - PB: il faut s'assurer que c'est faisable en fonction du traitement

Ce qui influence le WCET ?

- **Double dépendance :** architecture et du code binaire
- **Le code binaire**
  - Définit le chemin d'exécution (implémentation des fonctions, complexité effective des traitements)
  - Définit le modèle d'exécution (parallélisme, tâches)
- **L'architecture matérielle**
  - Latence d'accès à la mémoire, pipeline, et pre-fetch
  - Fréquence d'horloge
- **WCET : mesures ou estimations analytique ?**

  Pas de solution miracle

WCET : vision analytique et vision empirique

- **Analytique :**
  - Call graph et ETU:
  - Animation d'un automate pour générer une séquence d'exécution
  - Interprétation de la séquence pour calcul du WCET
- **Empirique :**
  - Contrôle de la plateforme d'exécution pour activation des pires scénarios d'exécution
  - Mesure : (oscillo ou logicielle …)

Conclusion

Pour faire le point

- **La chaine de compilation croisée == le coeur de la chaîne de production et déploiement du logiciel**
- **Le matériel == un processeur et des PI chacun influence l'environnement de compilation**
  - Processeur → instruction et assembleur spécifique
  - La carte → périphériques et services matériels (MMU, réseau, interruption, DMA ....)=> API dédiée
- **Le dimensionnement phase clé de la conception**

  (Pas de solution exacte totalement fonctionnelle)