# Les systèmes embarqués. Mise au point

email : kadionik@enseirb-matmeca.fr
web : http://kadionik.vvv.enseirb-matmeca.fr/

**Patrice KADIONIK**
**ENSEIRB-MATMECA**



## HISTORIQUE

- V1.0 09/17 : Création du document. Récupération depuis le cours IT363.

# PARTIE 1 :
# INTRODUCTION



## INTRODUCTION

- La conception et la mise au point d'un système embarqué est un processus difficile qui demande une certaine expertise.
- Cette partie se propose de donner quelques pistes et quelques règles de bon sens dans l'art de mettre au point un système embarqué :
  - Conception : mise au point du matériel.
  - Tests : mise au point du logiciel avec éventuellement modification du matériel.
- L'expérience acquise lors de la mise au point de précédents systèmes est un capital inestimable...

# INTRODUCTION

- La mise au point (*debug*) d'un système est pénible, long et consommateur d'argent.
- **Dans la conception matérielle** d'un circuit SoC, cette étape de tests/*debug* (avec des *patterns* bien choisis) consomme 60 % du temps du projet.



# INTRODUCTION

- **Dans la conception logicielle**, on estime qu'il reste encore 5 % de *bugs* après compilation statique d'un code source.
- Sur un programme de 10000 lignes, cela représente 500 *bugs* avant inspection du code et tests.
- L'inspection du code permet d'en éliminer 70 à 80 %, ce qui en laisse encore 100.
- Après tests, on estime en avoir éliminé 50 %, ce qui en laisse 50 dans le produit fini (d'après J. Ganssle) !

# INTRODUCTION

- Le calcul précédent n'est pas acceptable bien sûr (à moins de vouloir vendre des *releases* à tout coup) et l'on a donc besoin d'outils très performants et chers pour faire tendre ce chiffre vers 0.

- Toutes les techniques de management et de méthodologie (*extreme programming*) n'élimineront jamais le besoin de tester et de *debugger*…



# PARTIE 2 :
# LES OUTILS POUR LA MISE AU POINT D'UN SYSTEME EMBARQUE

# INTRODUCTION



## INTRODUCTION

- Dans le processus de mise au point du système électronique, il est primordial de bien connaître la palette d'outils à disposition.
- Mettre au point un système est d'autant plus compliqué que l'on travaille souvent avec un matériel non encore testé...
- A chaque problème, il existe l'outil approprié. Il convient donc de le présenter...

# L'OSCILLOSCOPE



## INTRODUCTION

- L'oscilloscope est le premier outil de *debug hard*.
- Il permet de visualiser réellement un signal électrique :
  - Analogique.
  - Numérique (sans niveaux logiques comme avec l'analyseur logique).

# AVANTAGES ET INCONVENIENTS

- L'oscilloscope permet de mettre en évidence parasites, *glitchs*…
- Il est limité à 4 sondes au plus soit 4 signaux.
- La logique de synchronisation (*trigger*) est très limitée.
- Il peut être intrusif en rajoutant une charge :
  - Un circuit logique marche quand on branche une sonde sur une broche puis ne marche plus quand on la débranche (charge capacitive).
- Il est important d'avoir de bonnes sondes calibrées avec une grande bande passante pour ne pas déformer le signal.



# L'ANALYSEUR LOGIQUE

# INTRODUCTION

- L'analyseur logique espionne des signaux logiques : bus d'adresses et de données, signaux de contrôle...
- On peut collecter les accès du processeur en mémoire et ensuite désassembler le code exécuté.
- On voit des adresses physiques externes donc si le cache d'instructions ou de données est activé, on ne voit pas tout…



# AVANTAGES

- L'analyseur logique est un outil universel dans la mise au point de systèmes numériques. Il n'est pas lié à un processeur particulier.
- L'analyseur logique permet de mettre en place des *triggers* de déclenchement de l'acquisition.
- Il est important de garder à l'esprit (idem pour l'émulateur ICE) que dans 90 % des cas, on choisit des conditions de *trigger* simples (booléen) par rapport aux *triggers* complexes tant vantés par les constructeurs, ce qui en augmente le prix !

# INCONVENIENTS

- Un analyseur logique est cher.
- Il y a le problème de la connectique et du nombre important de signaux à capturer.
- Pour désassembler le code exécuté, on est obligé de capturer les bus d'adresses et de données ainsi que les signaux de contrôle soit un nombre très important de signaux logiques.
- Si l'on a validé l'optimisation du code (compilé ou assemblé), il peut être difficile de remonter au fichier source par désassemblage.



# INCONVENIENTS

- Quand on branche l'analyseur logique, on introduit une charge différente sur un bus. On peut ainsi masquer ou créer des problèmes.
- La charge peut réduire le bruit induit et l'ensemble marche dans un environnement bruité. On enlève l'analyseur et plus rien ne marche. Cela peut être aussi le contraire.
- L'analyseur logique est donc intrusif.
- L'analyseur logique est passif : on n'a pas de contrôle sur le processeur.

# INCONVENIENTS

- Il faudra faire attention aux caches validés.
- L'analyseur logique fait un échantillonnage. Il convient donc d'avoir une fréquence d'acquisition suffisante et une taille mémoire d'acquisition conséquente...



# LE MONITEUR ROM

# INTRODUCTION

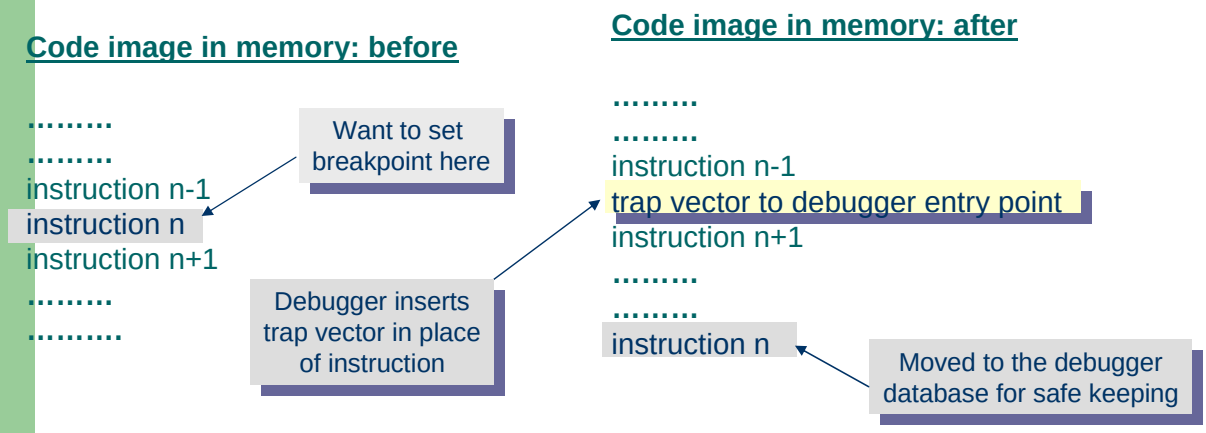
- Le moniteur ROM est un programme de *debug* accessible depuis un port de communication (liaison série).
- Il dialogue avec une application *debugger* pour lui envoyer la valeur courante d'un registre, d'une adresse mémoire et placer un point d'arrêt  simple à une adresse donnée ou télécharger en mémoire du code.



# POINT D'ARRET

- Un point d'arrêt (*breakpoint*) permet de dérouter l'exécution normale du code de l'application du système pour en examiner l'état (du processeur, de la mémoire, des E/S...) ou le modifier.
- L'instruction pointée par le point d'arrêt (son adresse) est remplacée par une routine spécifique (ou un *trap*) :
  - Sauvegarde des registres du processeur.
  - Examen de l'état du processeur.
  - Restauration des registres du processeur.

# POINT D'ARRET






# AVANTAGES

- Le moniteur ROM est bon marché.
- Il n'y a pas de problèmes de connectique.
- Il n'y a pas de problèmes de rapidité (c'est du code exécuté).

# INCONVENIENTS

- Le moniteur ROM demande un port de communication exclusif à prévoir dans le *design*.
- Il consomme des ressources du système cible : RAM, ROM, interruptions, ce qui peut être gênant pour un tout petit système.
- Il y a toujours des problèmes de configuration lors de la prise en main.
- Il ne marche pas s'il y a des *bugs* matériels.
- Il ne possède pas/peu de possibilités de *debug* en Temps Réel.



# L'EMULATEUR ROM

# INTRODUCTION

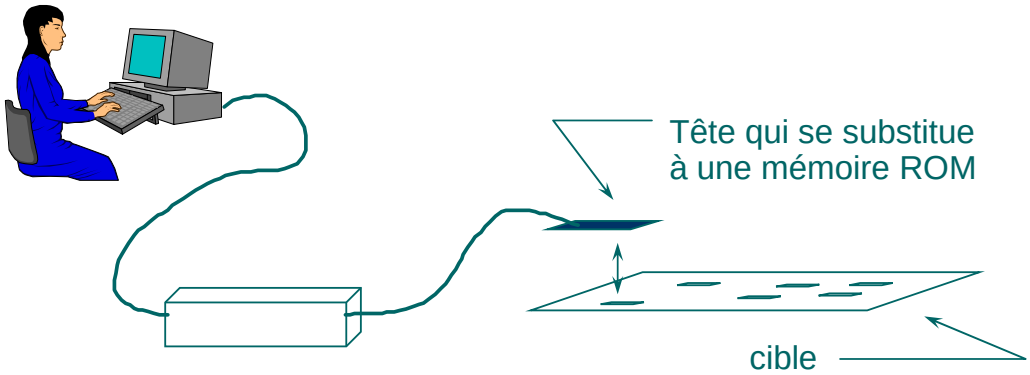
- L'émulateur ROM est un équipement qui se branche sur un support de mémoire ROM du système cible mais contient de la RAM au lieu de la ROM (*Overlay RAM*).

- Il possède :
  - Un câble de liaison en concordance avec les spécificités du support ROM.
  - De la RAM rapide.
  - Un port de communication avec le système hôte pour le *debug*.

- C'est un moyen simple et rapide de télécharger du code dans la cible en lieu et place de la mémoire ROM initiale.



# INTRODUCTION



# AVANTAGES

- L'émulateur ROM est bon marché.
- Il est compatible avec différents types de mémoires et n'est donc pas lié à un type de processeur comme l'émulateur ICE.
- Il permet d'ajouter un port de communication temporaire au système.
- Le téléchargement de code est rapide.
- On peut tracer l'activité ROM en Temps Réel.



# AVANTAGES

- On peut mettre un point d'arrêt en ROM.
- Il peut être utilisé avec d'autres outils de *debug*.

# INCONVENIENTS

- L'émulateur ROM a besoin d'être de plus en plus rapide (comme l'émulateur ICE) avec l'évolution des processeurs.
- Il ne fonctionne qu'avec une interface ROM : problème si l'on a un microcontrôleur avec de la ROM interne.
- Beaucoup de systèmes font une recopie ROM vers RAM pour des soucis de performance donc l'émulateur ROM est hors jeu.
- Il est inefficace en cas de problèmes matériels sur le système.
- Il est intrusif.



# L'EMULATEUR ICE

# INTRODUCTION

- ICE est l'acronyme de *In Circuit Emulator*.

- Pourquoi utiliser un émulateur ICE ?
  - Si le *hardware* du système n'est pas parfait.
  - C'est le meilleur outil pour détecter des problèmes matériels mais aussi logiciels.
  - Il n'utilise pas de ressources du système cible.
  - Il supporte un *debug* en Temps Réel avec possibilités de trace, *triggers*…

- C'est l'outil parfait pour une intégration conjointe matérielle et logicielle.



# INTRODUCTION

- L'émulateur ICE combine à la fois un *debugger*, un émulateur ROM et un analyseur logique.

- Il se substitue complètement au processeur qu'il va émuler.

# INTRODUCTION

- L'émulateur ICE permet de contrôler le système :
  - Lecture/écriture en mémoire.
  - Modification des registres du processeur.
  - Pas à pas.
  - Points d'arrêt.
  - Désassemblage de code.
  - Téléchargement de code.



# INTRODUCTION

- Point d'arrêt matériel : stoppe l'exécution du programme sans affecter le contexte du processeur.

- Point d'arrêt logiciel : insertion dans le code d'un appel vers une routine de *debug*. Il y a un problème si le code est en mémoire externe ROM car pas d'écriture possible.

# TRACES TEMPS REEL

- L'émulateur ICE permet de tracer l'activité du processeur (comme l'analyseur logique).
- Il admet des *triggers* complexes pour le lancement d'une acquisition (trace pour la surveillance d'une interruption) et désassemble le code capturé stocké dans une mémoire tampon circulaire.
- Si l'on possède la table des symboles, on peut remonter au fichier source directement.



# TRACES TEMPS REEL

**Example of Real Time Trace from HP64700 emulator**

```
Trace List                        Offset=0      More data off screen (ctrl-F, ctrl-G)
Label:  Address    Data                      Opcode or Status                 time count
Base:     hex      hex                           mnemonic                     absolute

after    004FFA     2700     2700   supr data rd word                  ------------
+001     004FFC     0000     0000   supr data rd word                  +  520      nS
+002     004FFE     2000     2000   supr data rd word                  +    1.0    uS
+003     002000     2479   MOVEA.L  0001000,A2                         +    1.5    uS
+004     002002     0000     0000   supr prog                          +    2.0    uS
+005     002004     1000     1000   supr prog                          +    2.5    uS
+006     002006     2679   MOVEA.L  0001004,A3                         +    3.0    uS
+007     001000     0000     0000   supr data rd word                  +    3.5    uS
+008     001002     3000     3000   supr data rd word                  +    4.00   uS
+009     002008     0000     0000   supr prog                          +    4.52   uS
+010     00200A     1004     1004   supr prog                          +    5.00   uS
+011     00200C     14BC   MOVE.B   #000,[A2]                          +    5.52   uS
+012     001004     0000     0000   supr data rd word                  +    6.00   uS
+013     001006     4000     4000   supr data rd word                  +    6.52   uS
+014     00200E     0000     0000   supr prog                          +    7.00   uS
```

# TRACES TEMPS REEL

- The user interface software has access to the linker symbol table
- Can intersperse C or C++ source lines with assembly language execution

```
                                    set source on
                                    display trace mnemonic


Trace List                        Offset=0      More data off screen (ctrl-F, ctrl-G)
Label:       Address        Data          Opcode or Status w/ Source Lines       time
Base:        symbols        hex                 mnemonic w/symbols               rel
+009    sysstack:+003FC2      0738     0738  supr data wr word                     520
+010    sysstack:+003FC0      0006     0006  supr data wr word                     480
+011    main:main+00000A      01AA     01AA  supr prog                             520
       ##########main.c - line    104 ##########################################
            initialize_system();
+012    main:main+00000C      4EB9   JSR     |_initialize_sys                     480
+013    main:main+00000E      0000     0000  supr prog                             520
+014    main:main+000010      114A     114A  supr prog                             480
       ##########initSystem.c - line      1 thru     38 ############################
       void refresh_menu_window();

       void
       initialize_system()
       {
+015     |_initialize_sys      4E56   LINK    A6,#00000                           520

STATUS:    M68000--Running user program    Emulation trace complete_____........
```



# AVANTAGES

- L'émulateur ICE est l'outil roi pour la mise au point matérielle et logicielle.
- Il fournit toutes les fonctionnalités de *debug* nécessaires.
- Le contrôle du processeur est garanti quel que soit l'état matériel du système cible.
- Il connaît les instructions en cache.

# INCONVENIENTS

- L'émulateur ICE coûte très cher.
- Il est fabriqué pour un type de processeur donné. Si l'on utilise un nouveau processeur, il faudra mettre à jour l'émulateur ICE.
- Problème de la connectique, du prix et de la fragilité de la tête : adaptation à pourvoir si processeur CMS.
- Il ne supporte pas toujours toutes les variantes d'un processeur donné.
- Les gens pensent que l'émulateur ICE est difficile à utiliser.



# INCONVENIENTS

- Le système cible doit être conçu pour supporter la tête de l'émulateur ICE.
- Un émulateur ICE ne fonctionne pas toujours à la fréquence maximale du processeur.

# INCONVENIENTS

- Quand on branche l'émulateur ICE, on introduit une charge différente de celle du processeur. On peut masquer ou créer des problèmes.
- La charge introduite peut réduire le bruit induit et l'ensemble marche dans un environnement bruité. On enlève l'émulateur ICE et plus rien ne marche. Cela peut être le contraire.
- L'émulateur ICE est donc intrusif.



# JTAG

# INTRODUCTION

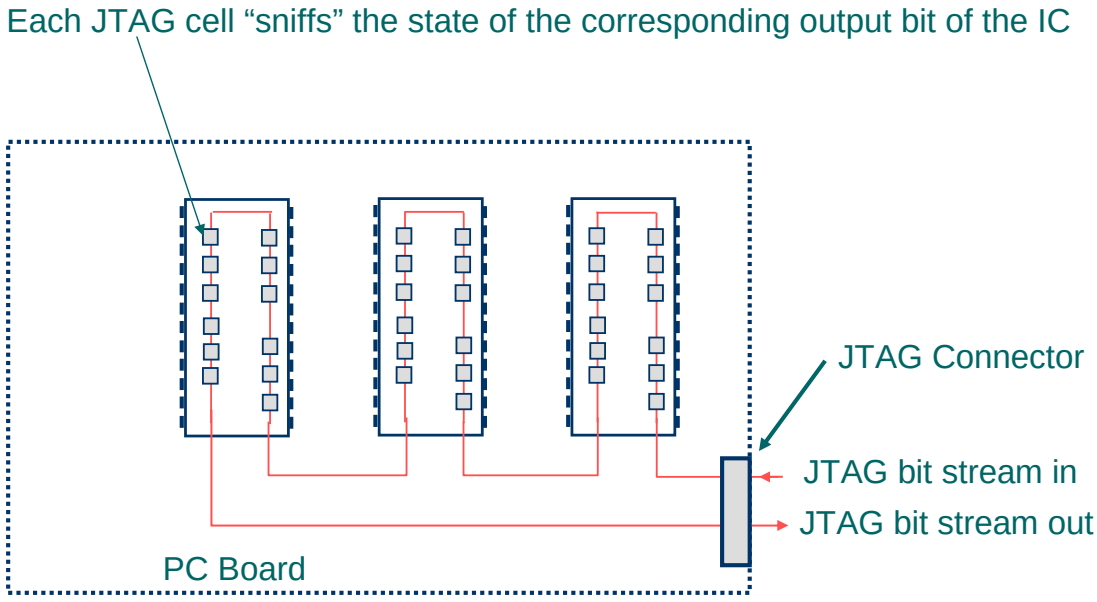
- Le JTAG (*Joint Test Action Group*) a été originellement créé pour le BIST (*Built In Self Test*) : autotests de circuits électroniques.
- C'est un standard IEEE (1149.1) qui permet originellement de tester des circuits électroniques pour l'industrie des cartes PC.
- Le JTAG possède 3 entités :
  - Le contrôleur TAP (*Test Access Port*). Machine à 16 états.
  - Un registre à décalage d'instructions.
  - Un registre à décalage de données.



# INTRODUCTION

- Le JTAG forme une boucle série interconnectant les broches d'E/S des circuits à surveiller/analyser (un point d'entrée, un point de sortie).
- 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 (*Boundary Scan*).
- Chaque bit du registre à décalage de données du JTAG correspond à la valeur instantanée d'une broche du circuit. On peut lire sa valeur courante ou lui affecter une nouvelle valeur (écriture).

# INTRODUCTION



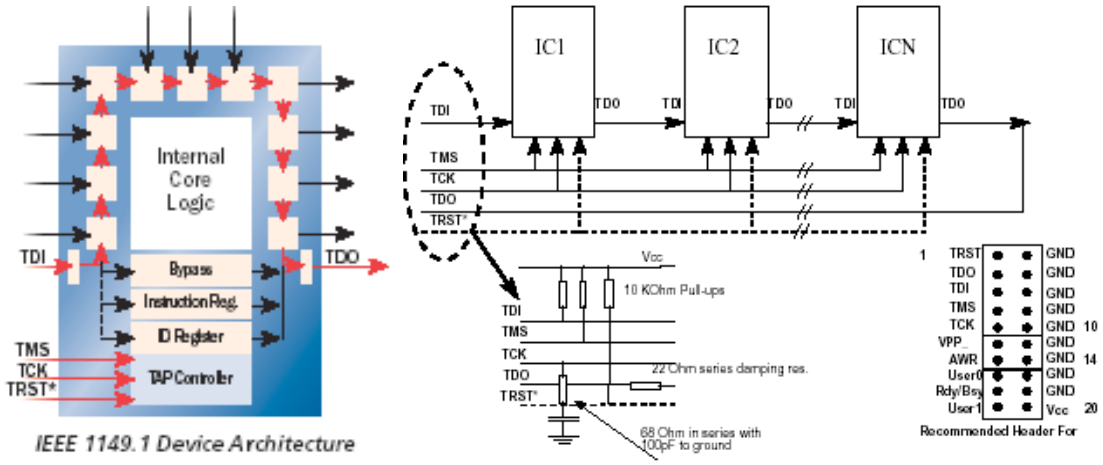

Bit stream forms one long shift-register



# INTRODUCTION

- Le contrôleur TAP possède 4 broches :
  - TCK : horloge (*Test ClocK*).
  - TMS : mode test de contrôle du TAP (*Test Mode Select*).
  - TDI : entrée donnée série (*Test Data In*).
  - TDO : sortie donnée série (*Test Data Out*).

# INTRODUCTION



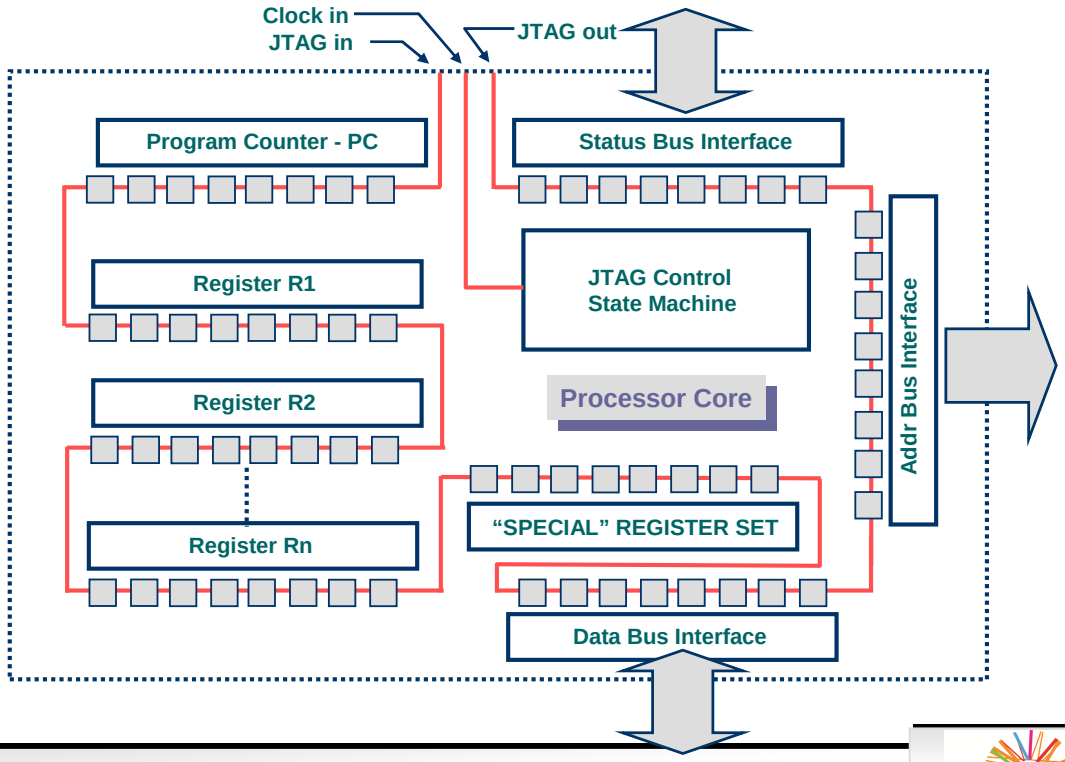

*IEEE 1149.1 Device Architecture*



# DEBUG PAR LE JTAG

- Il est possible de mettre au point un système par le JTAG : la norme JTAG a été étendue par certains fondeurs pour cela.
- On lie tous les registres internes et les blocs fonctionnels importants par une boucle JTAG. On a donc un *debugger* simple sur silicium OCD (*On Chip Debugger*).
- On autorise la lecture comme l'écriture dans les registres.
- On peut rajouter des registres de *debug* spécifiques accessibles par le JTAG.

# DEBUG PAR LE JTAG






# AVANTAGES

- Le *debugger* JTAG est donc un *debugger on chip*. On peut mettre des points d'arrêt, faire du pas à pas, regarder la mémoire, changer une valeur en mémoire, télécharger du code…
- Le JTAG est un standard ouvert.
- Il est adapté aux composants montés en surface.
- Il est bon marché.

# AVANTAGES

- Il n'utilise pas beaucoup de broches car il met en œuvre un protocole série.
- Il permet d'observer des entrées et positionner des sorties sans utiliser la logique du système.
- Il permet de réduire les points de tests sur le système.
- Il ne marche qu'avec un processeur le supportant. Si un processeur supporte le JTAG pour le *debug*, rajouter toujours le connecteur JTAG sur le PCB (on ne sait jamais) !



# AVANTAGES

- Il peut être utilisé aussi pour la programmation de mémoire Flash embarquée sur le système.
- Il supporte éventuellement des point d'arrêt matériels simples. Une broche spéciale suspend l'activité du CPU dans l'extension JTAG.

# INCONVENIENTS

- Il peut être lent : chaque bit doit être décalé dans le registre à décalage.
- Il n'y a pas de points d'arrêt complexes comme avec l'émulateur ICE.



# SIMULATEUR

# SIMULATEUR

- Le simulateur ISS (*Instruction Set Simulator*) est un logiciel exécuté par le système hôte de développement pour simuler un processeur cible particulier.

- La principale difficulté est de simuler les périphériques réels du système cible.

- Exemples de simulateur :
  - MPLAB pour le simulateur du microcontrôleur PIC Microchip.
  - *qemu* pour simuler des cartes Linux embarqué (SBC2440).



# LE CHOIX DES OUTILS

# LE CHOIX DES OUTILS

- Le bon sens doit avant guider l'ingénieur dans le choix des outils de *debug* à utiliser.

- On part généralement des outils les plus simples aux outils les plus sophistiqués :
  - L'oscilloscope : vérification des horloges, signaux critiques…
  - L'analyseur logique : *reset*, vérification des bus d'adresses et de données, accès mémoire.
  - L'émulateur ROM, ICE : couplage avec le logiciel. *Debug* au niveau source.
  - On peut ensuite embarquer un moniteur en ROM pour le lancement de l'application finale ou faire des tests en cours de production.



# LE CHOIX DES OUTILS

| **Fonctionnalité** | **Emulateur ICE** | **BDM** | **ROM Moniteur** | **Analyseur logique** | **ROM Emulateur** |
|---|---|---|---|---|---|
| Event triggers | Oui | Certains | Non | Oui | Non |
| Overlay RAM | Oui | Non | Non | Non | Oui |
| Points d'arrêt hard | Oui | Certains | Non | Non | Certains |
| Points d'arrêt complexes | Oui | Non | Non | Oui | Non |
| Time stamps | Oui | Non | Non | Oui | Non |
| Accès non intrusif | Oui | Oui | Non | Oui | Non |
| Coût | Très cher | Bon marché | Bon marché | Cher | Bon marché |

# LE CHOIX DES OUTILS

| Fonctionnalité | Emulateur ICE | BDM | ROM Moniteur | Analyseur logique | ROM Emulateur |
|---|---|---|---|---|---|
| **Debug au niveau source** | Oui | Oui | Oui | Certains | Oui |
| **Téléchargement code** | Oui | Oui | Oui | Non | Oui |
| **Pas à pas** | Oui | Oui | Oui | Non | Oui |
| **Points d'arrêt** | Oui | Oui | Oui | Non | Oui |
| **Display/modifs registres** | Oui | Oui | Oui | Oui | Oui |
| **Regarder variables** | Oui | Oui | Oui | Oui | Oui |
| **Traces TR** | Oui | Certains | Non | Oui | Non |



# PARTIE 3 :
# LA MISE AU POINT MATERIELLE D'UN SYSTEME EMBARQUE

# ENVIRONNEMENT DE DEVELOPPEMENT

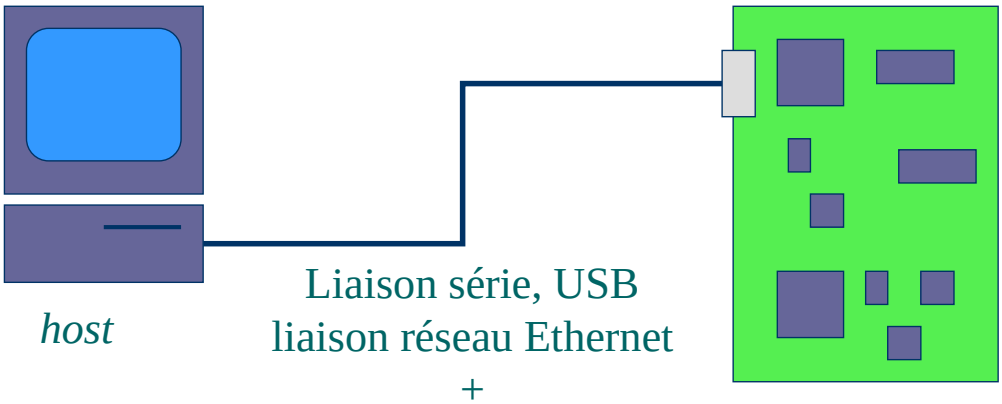


## INTRODUCTION

- Développer et mettre au point une application pour un système embarqué est un art difficile à maîtriser.
- On a généralement un environnement de développement croisé avec :
  - Une machine hôte (*host*) pour le développement et la mise au point.
  - Un système cible (*target*) dans lequel on va télécharger l'application puis l'exécuter dans la phase de mise au point.
- La mise au point fera appel aux outils précédents : émulateurs ICE ou ROM, JTAG, moniteur.

# ENVIRONNEMENT DE DEVELOPPEMENT

On a un environnement de développement croisé :

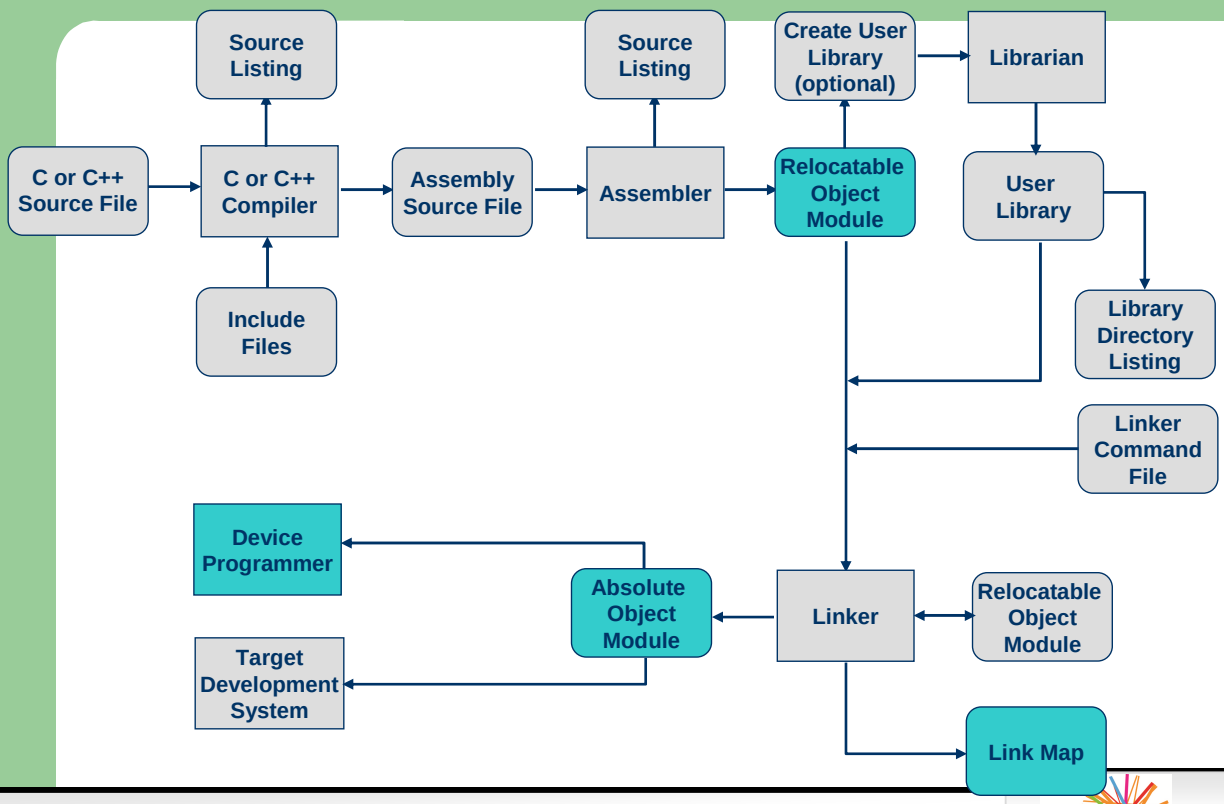





# ENVIRONNEMENT DE DEVELOPPEMENT

- On va développer et tester l'application sur une plateforme standard (PC) avec des outils logiciels conviviaux afin de faciliter le *debug*.
- On aura un compilateur croisé sur l'hôte pour avoir un code objet exécutable par le processeur de la cible.
- On aura un *debugger* croisé sur l'hôte pour mettre au point l'application exécutée par le processeur de la cible.

# ENVIRONNEMENT DE DEVELOPPEMENT






# ENVIRONNEMENT DE DEVELOPPEMENT

- Si l'on travaille en langage évolué (langage C), on teste au niveau du source en langage évolué (*source level debugging*).
- Le *debugger* croisé (au niveau source) sera couplé avec un équipement de *debug* (ICE, émulateur ROM, JTAG...).
- Il doit supporter le mode trace en Temps Réel pour capturer le traitement d'une interruption (ISR) sans ralentir le système.

# TRUCS ET ASTUCES POUR UNE BONNE MISE AU POINT. AVANT DE COMMENCER



## INTRODUCTION

- Avant de commencer quoi que ce soit, il convient d'abord de préparer le terrain quand on doit développer le *firmware*.
- Il est important de savoir de quels outils on a besoin et s'assurer qu'ils sont compatibles avec le système à *debugger*.

# INTRODUCTION

- Les points à ne pas négliger :
  - Etre impliqué dans la conception du système.
  - Maîtriser le *hardware* du système.
  - Avoir une copie de tous les documents.
  - Commencer doucement (mais sûrement).
  - Regarder ce que l'on vient de concevoir.



# ETRE IMPLIQUE DANS LA CONCEPTION

- Vérifier que le composant sur lequel démarre (*boot*) le système est reprogrammable : pas de désoudage à faire, interface JTAG pour le reprogrammer…

- Le système doit inclure un mécanisme de communication entre le logiciel de *boot* (*firmware*) et l'humain : leds, port série, connecteur spécial non équipé dans la version finale pour limiter le coût du produit fini, JTAG...

# MAITRISER LE HARDWARE

- Etablir une bonne relation entre les gens du matériel et du logiciel, ne pas se jeter la balle :
  - « c'est pas le soft qui plante, c'est le *hard*…non c'est pas le *hard* qui plante, c'est le *soft* ».
- Passer un peu de temps avec le gens du *hardware*, poser des questions.
- Avoir une copie des schémas électroniques.



# AVOIR UNE COPIE DES DOCUMENTS

- On doit connaître plus que les schémas.
- Avoir les *datasheets* de tous les circuits utilisés dans le système (travailler sur papier).
- Vérifier s'il n'existe pas d'*errata* pour chaque *datasheet* surtout quand un circuit récent est utilisé. Il est bien sûr possible alors de découvrir des *bugs*.

## ETRE SUR DU MATERIEL

- Si le système est neuf, s'assurer que les tests électriques élémentaires ont été faits :
  - Court-circuits.
  - Tests de continuité.
  - Pas de faux contacts.
  - Soudures froides.

- Savoir comment brancher l'alimentation.



## COMMENCER DOUCEMENT

- Faire de petits pas. Ne pas se lancer dans un énorme programme mais procéder par de petits programmes de tests pour tester un fonctionnalité bien particulière.

- Points à vérifier :
  - Est-ce que le programme est bien « mappé » en mémoire ?
  - Est-ce que la procédure de reset du CPU est bien comprise ?
  - Est-ce que le fichier objet est dans le bon format ?
  - Est-ce que le programmateur est bien configuré ?
  - Est-ce que le partitionnement de la mémoire de *boot* adresses paires/impaires est bien réalisé ?
  - Est-ce que le matériel marche réellement ?

## COMMENCER DOUCEMENT

- Rechercher des exemples de code sur le site du fondeur du processeur.
- S'abonner aux forums adéquats.
- Rechercher des informations sur Internet.



## REGARDER CE QUE L'ON A CONCU

- Consulter le fichier de *mapping* mémoire.
- Vérifier le *mapping* mémoire :
  - Vérifier les sections de code : .text
  - Vérifier les sections de données initialisées : .data
  - Vérifier les sections de données non initialisées : .bss (*Bloc Starting by Symbol*)

# TRUCS ET ASTUCES POUR UNE BONNE MISE AU POINT. DANS L'ACTION



## INTRODUCTION

- Après le préambule précédent, il est possible de « toucher » au matériel.
- En cours d'écriture du logiciel (*firmware*), il y a néanmoins des tests matériels à faire quelquefois quand on découvre des *bugs* matériels ou un *firmware* qui ne fait pas ce que l'on veut.
- Il est important de faire alors appel à l'équipe *hardware* pour confirmation et correction.
- Les procédures suivantes sont importantes à valider.

# POINTS A VERIFIER

- Vérifier les tensions d'alimentation :
  - Utiliser un voltmètre.
  - Utiliser un oscilloscope. Présence de bruit sur les alimentations ?
- Vérifier la présence d'une horloge valide :
  - Utiliser un oscilloscope : présence d'une horloge sur le processeur, bonne fréquence ?
- Vérifier la bonne assertion des *Chip Select* au *boot* (*reset*).



# POINTS A VERIFIER

- Vérifier la bonne soudure des broches : pas de soudures froides, faux contacts.
- Faire attention aux décharges ESD. Utiliser un bracelet anti-statique.
- Ecrire une simple boucle installée au vecteur de reset pour vérifier la bonne assertion du *Chip Select* de la mémoire ROM (68000, à installer au vecteur 0).
- S'aider au besoin d'un analyseur logique :

```
LOOP
        JUMP        LOOP
```

## POINTS A VERIFIER

- Utiliser une led comme test (avec le programme à tester) si l'on n'a pas d'analyseur logique ou d'oscilloscope (ce qui serait surprenant !). Jouer sur la rapidité du clignotement, led allumée ou éteinte. Une fois ce point acquis, écrire 3 routines :
  - *led_on()*
  - *led_blink_slow()*
  - *led_blink_fast()*

  et s'en servir comme routines de *debug*.

- Si l'on a plus de leds, on peut indiquer un état de progression dans la phase de *boot* (3 leds soit 8 états).



## POINTS A VERIFIER

- Tester la SRAM externe (patterns $55 et $AA comme tests grossiers).

- Tester la liaison série (programme d'écho).

- Initialiser le contrôleur de DRAM.

- Et si tout va bien alors : programmer en langage évolué de type C. Utiliser un compilateur croisé C qui offre les routines C ANSI d'E/S et modifier le *getchar()* et *putchar()* pour coller à son *hardware*. On a accès alors à son *printf()* préféré pour un *debug* simple...

# POINTS A VERIFIER

- Faire attention aux données C initialisées au *reset* du système. Elles doivent donc être mise en mémoire non volatile. On ne peut donc pas les modifier en cours d'exécution d'un programme.

- Regarder la documentation du compilateur croisé et notamment la section chargeur (*loader*) en analysant plus particulièrement le fichier assembleur *crts.s* (*C Run Time Startup*).



# PARTIE 4 :
# LA MISE AU POINT LOGICIELLE D'UN SYSTEME EMBARQUE

# INTRODUCTION

- Il ne suffit pas savoir programmer, il faut savoir bien programmer !
- Un système embarqué doit être robuste et son code bien écrit.
- Le langage de prédilection pour le développement logiciel est le langage C.
- Le langage C est un langage de haut niveau mais proche du matériel. Cette partie met le doigt sur certains points de la programmation en langage C auxquels il faudra faire très attention.



# INTRODUCTION

- Se méfier des options d'optimisation du compilateur.
- Bien traiter les interruptions utilisées par le système en renseignant les vecteurs concernés dans la table des vecteurs du processeur. Mise en place de la routine d'interruption ISR (*Interrupt Sub Routine*).
- Renseigner toute la table des vecteurs même pour les interruptions non utilisées (ISR *dummy*…).

# CLASSES DE STOCKAGE

- Pour  un système embarqué, on est concerné par la façon dont le compilateur C va stocker les variables.
- On doit garder le contrôle du *mapping* mémoire.



# CLASSES DE STOCKAGE

- La classe de stockage d'une variable peut être :
  - `auto :` `auto int a;`
    - Stockage sur la pile.
    - Variable détruite en fin d'exécution de la fonction.

# CLASSES DE STOCKAGE

- La classe de stockage d'une variable peut être :
  - `register :  register int a;`
    - Stockage de la variable dans un registre du processeur.
    - Accès à la variable très rapide.
    - Limitation par le nombre de registres de données.



# CLASSES DE STOCKAGE

- La classe de stockage d'une variable peut être :
  - `static :  static int a;`
    - Contraire de auto.
    - Si c'est une variable globale, elle existe durant toute la durée d'exécution du programme.

# CLASSES DE STOCKAGE

- La classe de stockage d'une variable peut être :
  - `extern :` `extern int a;`
    - Variable définie dans un autre fichier source.
    - Variable importée.
    - Utilisé durant la phase d'édition de liens pour résoudre les références croisées.



# MODIFICATEURS D'ACCES

- Il existe 2 modificateurs d'accès à une variable :
  - `const :` `const int a=10;`
    - La valeur de la variable ne peut pas être changée par le programme.
    - A placer en mémoire ROM.
  - `volatile :` `volatile char a;`
    - La valeur de la variable peut changer d'elle-même en cours d'exécution du programme.
    - Registre d'E/S mappé en mémoire.
    - Zone mémoire servant de buffer par un périphérique externe (contrôleur DMA).

# MODIFICATEUR D'ACCES VOLATILE

- Le modificateur d'accès *volatile* est à utiliser obligatoirement :
  - Pour l'accès à des registres de périphériques mappés en mémoire.
  - Zone mémoire accédée par DMA.
  - Variables globales modifiées par une ISR.
  - Variables globales dans une application multi-tâche.



# VOLATILE : MAUVAISE PROGRAMMATION

- Exemple : scrutation active d'un registre d'état d'un périphérique 8 bits.
- Le registre d'état 8 bits est à l'adresse 0x1234.
- Scrutation active du registre d'état jusqu'à ce qu'il soit non nul.

# VOLATILE : MAUVAISE PROGRAMMATION

```
unsigned char  * ptr = (unsigned char *) 0x1234;

// Wait for register to become non-zero
while (*ptr == 0)
   ;
// Do something else
```

Cela risque de ne pas marcher dès que le compilateur optimise le code. Le code généré pourrait être :

```
      mov    ptr, #0x1234
      mov    a, @ptr
loop  bz     loop
```



# VOLATILE : MAUVAISE PROGRAMMATION

- Le compilateur C voit qu'il a déjà la valeur courante de la variable.
- Il n'y a aucun besoin d'aller la relire puisque c'est la toujours même valeur (la valeur d'une case mémoire ne change pas d'elle même !).
- Il génère alors une boucle infinie.
- On ne sort donc pas de le boucle !
- D'où un BUG en *run time* !

# VOLATILE : BONNE PROGRAMMATION

```
volatile unsigned char  * ptr = (volatile unsigned char *)
  0x1234;
```

Le code assembleur généré est alors :

```
              mov    ptr, #0x1234
loop          mov    a, @ptr
              bz     loop
```

On a alors le comportement attendu !



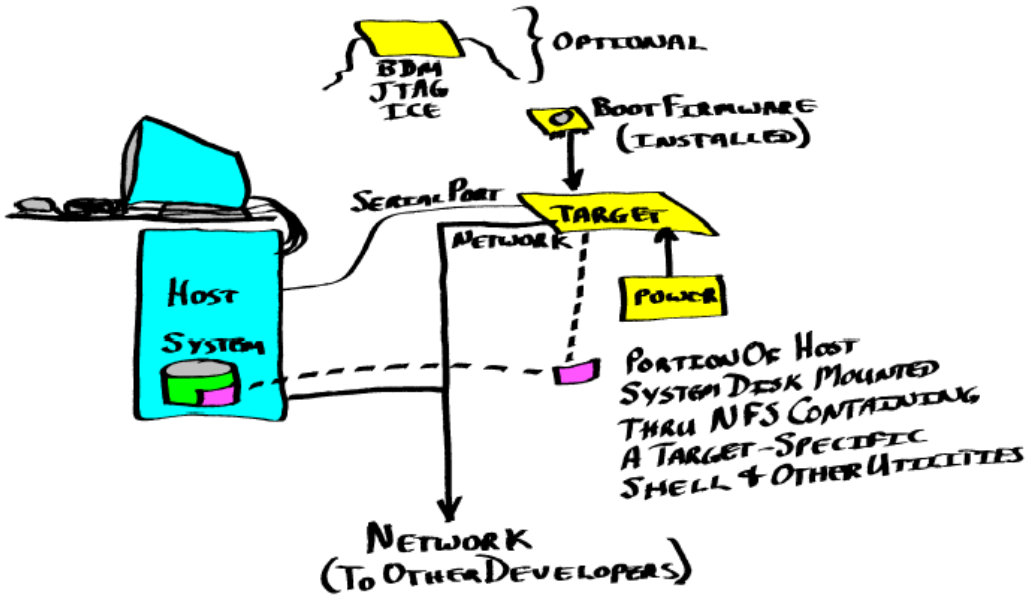
# PARTIE 5 :
# EXEMPLE ILLUSTRE : MISE AU POINT LOGICIELLE D'UN SYSTEME LINUX EMBARQUE

# LE SYSTEME HOTE OU HOST

- Utilisation d'un PC ou une station de travail comme *host* sous Linux en général.
- Connexion par liaison série (ou par réseau Ethernet maintenant) avec le système embarqué cible ou *target* : *minicom*…
- Partage de fichiers avec la cible par montage NFS si la cible a une interface réseau Ethernet.
- Développement croisé sur l'hôte avec tous les composants logiciels nécessaires.



# L'ENVIRONNEMENT



# LE SYSTEME CIBLE OU TARGET

- Utilisation d'une carte CPU qui exécute Linux.
- On a besoin d'y intégrer différents composants logiciels :
  - Une *boot* ROM contenant un *bootloader* chargeant le noyau Linux en mémoire et puis l'exécutant.
  - Un noyau Linux.
  - Une image disque d'un système de fichiers Linux contenant les programmes applicatifs, bibliothèques partagées et libc, modules Linux, fichiers de script...



# TELECHARGEMENT DANS LA CIBLE

- Utilisation d'un équipement particulier pour le téléchargement dans la cible en phase de développement (en final, tout sera « romé ») :
  - ICE, JTAG…
- On flashera en premier en mémoire Flash le *bootloader*.
- Le *bootloader* pourra ensuite télécharger le noyau et l'image du système de fichiers par la liaison série ou par le réseau (TFTP) (et le mettre en mémoire Flash au final…).
- Le système de fichiers (*root File System*) peut être éventuellement monté par NFS au lancement du noyau Linux de la cible...

# ENVIRONNEMENT DE DEVELOPPEMENT CROISE

- Environnement de développement croisé sous Linux. On utilisera les outils GNU :
  - gcc…
  - ld, as, nm, obcopy, strip, ar…
- Ces outils permettent de créer une image binaire du noyau et du système de fichiers (*root File System*) sur l'hôte.
- On pourra utiliser un atelier de génie logiciel IDE (*Integrated Design Environment*) : Eclipse, NetBeans, KDevelop...



# ENVIRONNEMENT DE DEVELOPPEMENT CROISE

- Utilisation du *debugger* GNU GDB.
- *Debug* par la liaison série ou par le réseau. On a sur la cible un programme serveur *gdbserver* qui attend les ordres d'une application cliente exécutée par l'hôte. On bénéficie alors d'un interface graphique conviviale sur l'hôte qui permet de *debugger* au niveau source :
  - Utilisation d'un *front end* à gdb comme DDD (*Data Display Debugger*).
- On pourra aussi mettre à profit les outils de base Linux sur l'hôte pour la gestion du projet :
  - make, svn, git...

# DEVELOPPEMENT VS PRODUCTION

- En phase de développement, on a à disposition des outils :
  - Qui permettent un téléchargement rapide.
  - Qui permettent de télécharger le noyau Linux.
  - Qui permettent de télécharger l'application par NFS.
  - Qui permettent de développer, de tester sur l'hôte puis de porter sur la cible.

- En phase de production, le *bootloader*, le noyau et son *root File System* sont romés en mémoire Flash :
  - Taille totale ?
  - Comment mettre à jour (correction de *bugs*…) ?



# QUOI DEVELOPPER ?

- On doit développer :
  - Le *bootloader*.
  - Le portage du noyau.
  - Les pilotes de périphériques.
  - L'application.
  - L'intégration du tout.

# DEVELOPPEMENT DU BOOTLOADER

- Développement du *bootloader* :
  - En utilisant les sources éventuels fournis avec la carte si on l'a achetée.
  - En utilisant un *bootloader open source* : u-boot…
  - Se l'écrire soi-même.

- Le *bootloader* est en mémoire Flash. Il prend en charge l'initialisation de base du système, les autotests, la décompression de l'image et du *root FS* stockés en mémoire Flash ou téléchargé (liaison série ou réseau).

- Il peut avoir des fonctions de *debugger* (moniteur).



# PORTAGE DU NOYAU

- Il est important de comprendre comment marche le noyau et notamment la procédure de *boot*.

- Il faut choisir le portage correspondant au type de processeur de la cible.

- Il faut modifier éventuellement les sources du noyau (fichiers assembleur) pour coller au *hardware* de la carte cible.

# DEVELOPPEMENT DES DRIVERS

- Il est souhaitable de choisir des composants qui sont supportés par Linux lors de la phase de conception du système.
- Lors de la phase de configuration du noyau, on choisira les *drivers* appropriés. On pourra éventuellement les modifier pour coller au *mapping* mémoire de la cible.
- Si l'on doit écrire un nouveau *driver* pour un matériel spécifique, il est préférable de partir d'un *driver* existant similaire.
- Il faudra faire le choix entre développer un *driver* statique lié au noyau ou un *driver* dynamique (module Linux) chargé en mémoire par le noyau à la demande.



# DEVELOPPEMENT DE L'APPLICATION

- Il est important de voir s'il n'existe pas déjà une application existante collant à son besoin. Dans le cas éventuel, la porter sur sa cible. Dans le meilleur des cas, cela se traduira par une simple recompilation croisée.
- Choisir de préférence des applications *open source*.
- Développer une interface graphique si besoin :
  - *Frame buffer*.
  - Nano/X.
  - Qt embedded.

# INTEGRATION SYSTEME

- Il faudra veiller à :
  - Evaluer la taille mémoire consommée (RAM, Flash).
  - Prendre en compte la faible empreinte mémoire :
    - Chargement du noyau en RAM ou XIP (*eXecute In Place*).
    - *Root File System* en RAM ou Flash.
    - Type du système de fichiers : ext3, JFFS2 pour un système embarqué.
    - Bibliothèques : statiques ou dynamiques.
    - Bibliothèques libc (µClibc, dietlib, newlib…).



# CONCLUSION

# CONCLUSION

- Dans le monde de l'embarqué, le logiciel libre et le matériel libre sont de plus en plus utilisés. Le noyau Linux pour l'embarqué ou plus simplement Linux embarqué est un allié de poids.

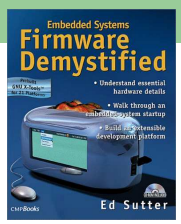
- La mise au point d'un système embarqué est maintenant complexe et il est important de connaître les outils pour cette mise au point.



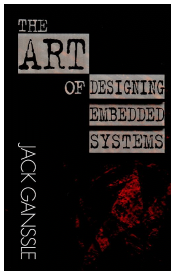
# BIBLIOGRAPHIE

# REFERENCES BIBLIOGRAPHIQUES

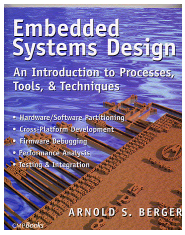
- Embedded Systems. Firmware Demystified. E. Sutter. Editions CMP Books




- The Art of Designing embedded Systems. J. Ganssle. Editions Butterworth-Heinemann



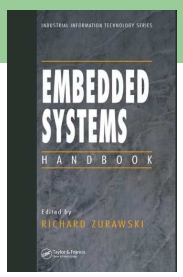

- Embedded Systems Design. A S. Berger. Editions CMP Books






# REFERENCES BIBLIOGRAPHIQUES

- Embedded Systems Handbook. R. Zurawski and al. Editions CRC Press




- Designing Embedded Hardware. J. Catsoulis. Editions O'Reilly


- Embedded Systems Architecture. T. Noergaard. Editions Newnes