

# Une approche synchrone à la conception de systèmes embarqués temps réel

Dumitru Potop-Butucaru

dumitru.potop@inria.fr

cours EPITA, 2022, 4<sup>ème</sup> séance

# Contenu de ce cours

- Ordonnancement en ligne préemptif
- Introduction à
  - IMA – Integrated Modular Avionics
  - ARINC 653
    - RPi653 – une implémentation de ARINC 653 pour cartes Raspberry Pi
- Préparation du TP
  - Temps réel "mou" pour le traitement de signal (son)
  - Début des activités sur carte Raspberry Pi

# Jargon temps réel

- **Tâches** = Fonctions calculant les réactions (ou juste des bouts de ces réactions)
- Moments importants dans la vie d'une tâche :
  - **Arrivée** = date où l'exécution d'une tâche **peut** commencer
  - **Echéance** = date après l'arrivée où la tâche doit être finie.
  - **Démarrage** (start)
  - **Fin** (des calculs)
  - **Interruption** – suspensive ou définitive
  - **Reprise** – avec préservation de l'état (contexte)
- Types de tâches:
  - **Périodiques** – arrivent à des intervalles fixes
  - **Sporadiques** – arrivent avec une distance minimale entre elles
  - Apériodiques – peuvent arriver sans restrictions

# Ordonnancement temps-réel

- Allocation des ressources aux tâches dans le but d'assurer le respect des échéances
  - Démarrage/interruption/reprise de tâches
  - Mais aussi allocation mémoire, allocation de ressources I/O...
- On peut le faire :
  - **En ligne** - lors de l'arrivée des stimuli, suivant des politiques d'ordonnancement généralistes
  - **Hors ligne** - dates de départ choisies avant exécution, politique d'ordonnancement spécifique au système

...mais la limite entre les 2 n'est pas exactement définie:

- Une politique « en ligne » a des paramètres que l'on peut varier avant l'exécution
- Une politique « hors ligne » peut ne pas couvrir des aspects comme l'allocation des bancs mémoire ou même le choix du processeur, qui sont réalisées alors en ligne.

Encore une fois, c'est l'état d'esprit qui compte.

# Ordonnancement en ligne

- **Implantation événementielle**
  - Plusieurs signaux/interruptions peuvent déclencher des calculs.  
Problèmes de synchronisation
- **Algorithmes classiques**
  - RM (rate monotonic), FP (fixed priority), EDF (earliest deadline first), DM (deadline monotonic), etc.
- **Avantages:**
  - Réactions très rapides à des événements prioritaires.
  - Robustesse aux variations temporelles (temps d'exécution, dates d'arrivée des tâches).
- **Problèmes:**
  - Nécessitent souvent des marges importantes avec les critères d'ordonnancabilité classiques (30% pour RM)
  - Non-déterminisme temporel, plus difficile à vérifier/simuler/tester
  - Exécution conditionnelle difficile à exploiter

# Algorithmes d'ordonnancement en ligne

- A lire:
  - **Liu & Layland 1973:** Scheduling algorithms for multiprogramming in a hard real-time environment.
  - **G. Buttazzo 2005:** Hard real-time computing systems
- Modélisation classique:
  - $n$  tâches  $\tau_1, \dots, \tau_n$ . Chaque tâche  $\tau_i$  a:
    - Une période  $T_i$ 
      - Tâches périodiques (avec ou sans gigue et/ou dérive) ou sporadiques
    - Une durée (capacité)  $C_i$
    - Une échéance  $d_i$  – calculée à partir de chaque arrivée
  - Exécution sur un processeur **séquentiel**
  - Comment ordonner l'exécution des tâches pour **garantir** le respect les échéances?
    - Qu'est-ce qui se passe si une échéance est manquée ?

# Algorithmes d'ordonnancement en ligne

- Les algorithmes « équitables »
  - FIFO, Round Robin, Weighted Round Robin, Resource reservation/Real-Time Calculus
  - Pas de prise en compte de l'état de l'application (sauf changements de configuration coûteux)
  - Mais: modularité (pour la réservation de ressources)
- Ordonnancement préemptif à priorités
  - Chaque tâche  $\tau_i$  a une priorité  $prio_i$
  - A chaque instant, la tâche qui s'exécute est une des tâches actives de priorité maximale
    - Une tâche plus prioritaire interrompt les tâches moins prioritaires
  - Permet de donner plus de temps aux tâches qui en ont besoin, en fonction de l'état du système
    - État = activation des tâches

# Ordonnancement préemptif

- Comment choisir les priorités
  - Statiquement (offline) = Priorité fixe (FP)
    - RM = Rate Monotonic:  $T_i > T_j \Rightarrow prio_i < prio_j$
    - DM = Deadline Monotonic:  $d_i > d_j \Rightarrow prio_i < prio_j$
    - ...
  - Dynamiquement (online)
    - EDF = Earliest Deadline First (échéance la plus proche)
    - LLF = Least Laxity First (la tâche avec le moins de "mou")
    - ...
- Règles simples, faciles à implémenter, à l'aide d'interruptions
  - Support matériel fourni par tous les processeurs

# Ordonnancement préemptif

- Est-ce que toutes les tâches vont respecter leurs échéances?
  - Condition nécessaire : chaque tâche, prise séparément, doit pouvoir s'exécuter :
$$C_i \leq T_i \quad \text{ou} \quad \frac{C_i}{T_i} \leq 1$$
    - Jamais une nouvelle instance arrive alors que l'ancienne ne soit finie
  - Notion centrale : **utilisation (charge) du processeur**
    - Exemple : La tâche  $t$  a une période de 50ms et une durée de 10ms. Alors,  $t$  utilise  $10/50 = 20\%$  de la capacité de calcul du processeur.

# Ordonnancement préemptif

- Est-ce que toutes les tâches vont respecter leurs échéances?
  - Notion centrale : **utilisation (charge) du processeur**

$$\sum_{i=1}^n \frac{C_i}{T_i}$$

- On vise à garantir le respect des échéances => utilisation des durées au pire cas  $C_i$
- $\frac{C_i}{T_i}$  = pourcentage du temps CPU utilisé (au pire cas) par  $\tau_i$
- Condition **nécessaire** sur un mono-processeur :
- Condition **nécessaire** sur n processeurs:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n$$

# Ordonnancement préemptif

- Est-ce que toutes les tâches vont respecter leurs échéances?
  - Critère d'ordonnancabilité = condition suffisante
  - Liu & Layland 1973: Si le coût des préemptions est 0, et  $T_i = d_i$ , alors :
    - FP/RM/DM: Condition suffisante :

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n \cdot (2^{1/n} - 1)$$

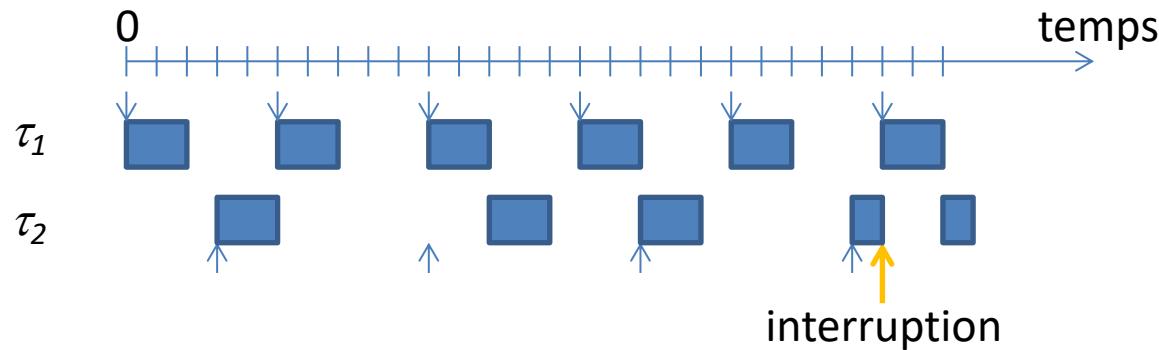
$$\lim_{n \rightarrow \infty} \left( n \cdot (2^{1/n} - 1) \right) = \ln 2 \cong 0,69$$

- EDF: Condition nécessaire et suffisante (optimal):

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

# Ordonnancement préemptif

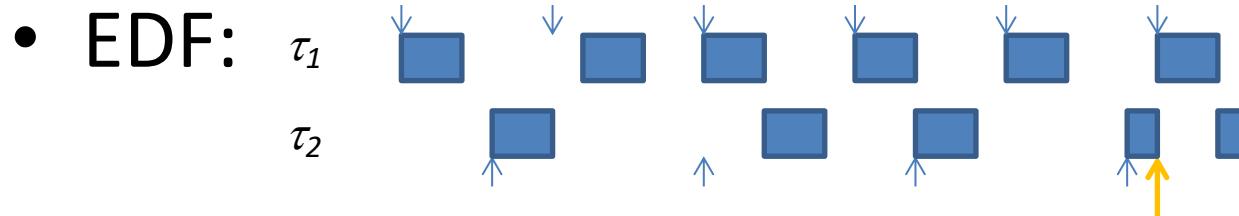
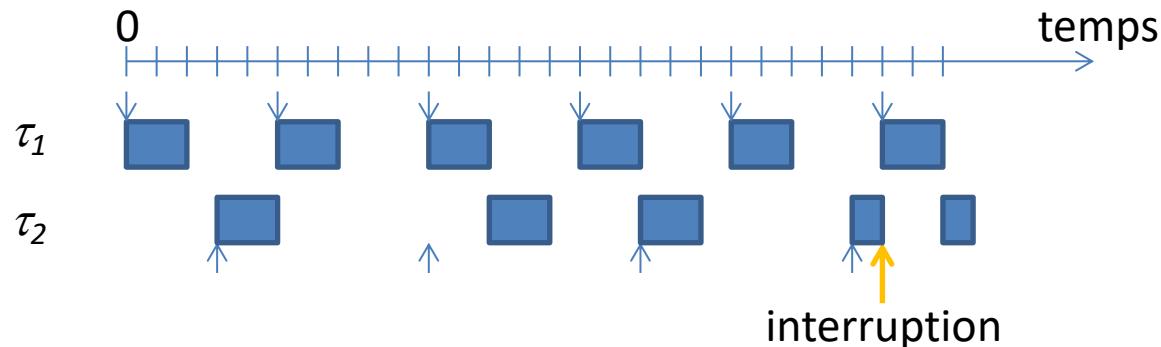
- Exemple:  $n=2, T_1 = 5, C_1 = 2, T_2 = 7, C_2 = 2$
- RM:  $prio_1 = 1, prio_2 = 0$



$$\sum_{i=1}^n \frac{C_i}{T_i} < n \cdot (2^{1/n} - 1) < 1$$

# Ordonnancement préemptif

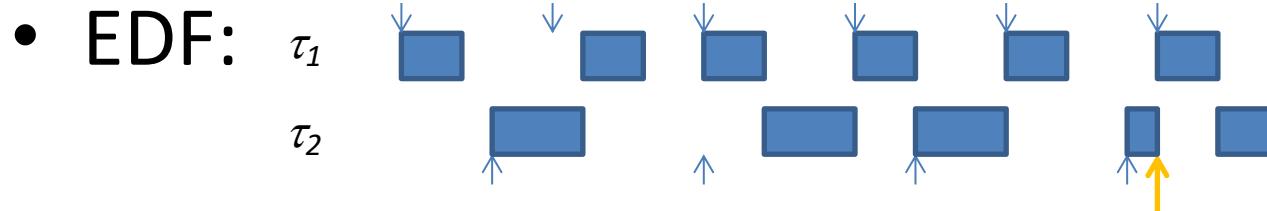
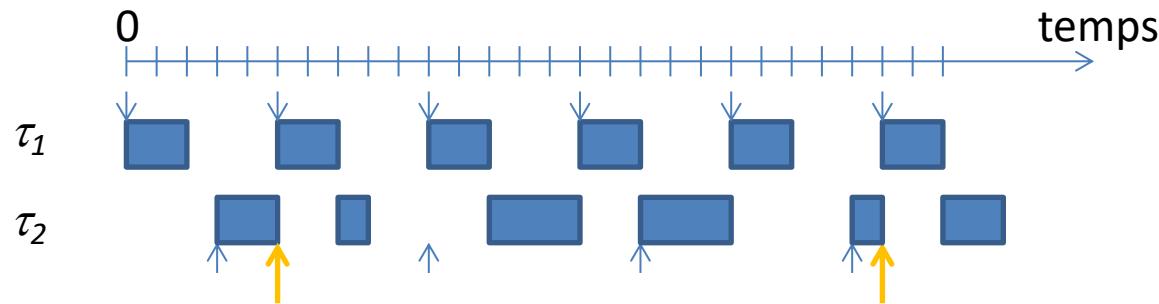
- Exemple:  $n=2, T_1 = 5, C_1 = 2, T_2 = 7, C_2 = 2$
- RM:  $prio_1 = 1, prio_2 = 0$



$$\sum_{i=1}^n \frac{C_i}{T_i} < n \cdot (2^{1/n} - 1) < 1 \quad \text{\~{}0.68\% CPU charge}$$

# Ordonnancement préemptif

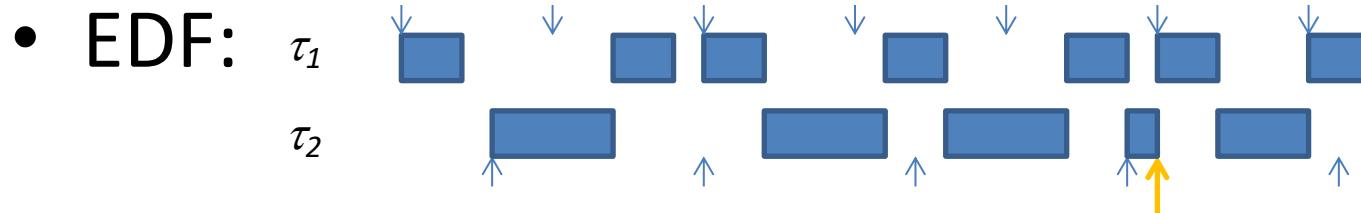
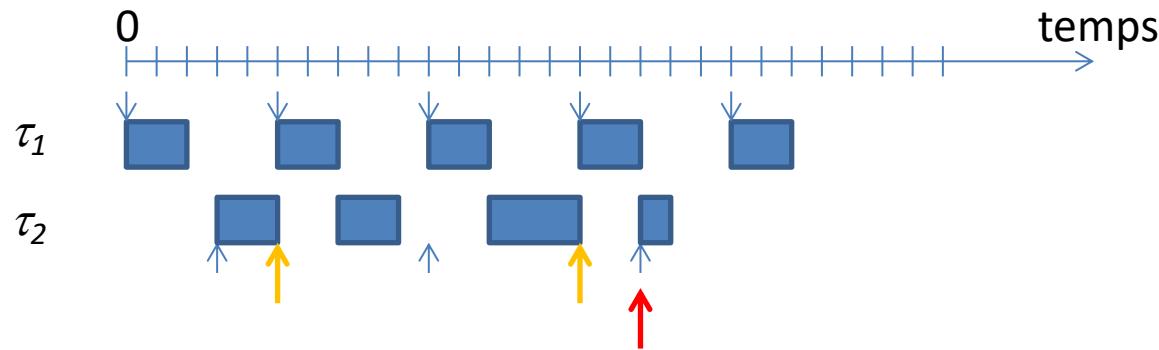
- Exemple:  $n=2, T_1 = 5, C_1 = 2, T_2 = 7, C_2 = 3$
- RM:  $prio_1 = 1, prio_2 = 0$



$$n \cdot (2^{1/n} - 1) < \sum_{i=1}^n \frac{C_i}{T_i} < 1 \quad \text{\~{}0.82\% CPU charge!}$$

# Ordonnancement préemptif

- Exemple:  $n=2, T_1 = 5, C_1 = 2, T_2 = 7, C_2 = 4$
- RM:  $prio_2 = 0, prio_1 = 1$



$$n \cdot (2^{1/n} - 1) < \sum_{i=1}^n \frac{C_i}{T_i} < 1 \quad >0.97\% \text{ CPU charge!}$$

# Ordonnancement préemptif

- Toutes les analyses d'ordonnancabilité font des hypothèses:
  - Durées des opérations:
    - au pire cas (WCET), parfois dans le meilleur des cas (BCET)
  - Coût 0 (ou fixe) pour les préemptions, communications, etc.
- Que se passe-t-il si ces hypothèses ne sont pas respectées ?
  - FP (RM/DM):
    - Les tâches plus prioritaires gardent l'accès aux ressources
      - bien pour les systèmes critiques, lien priorité-criticité
  - EDF:
    - On n'en sait rien (pas bien du tout)

# Ordonnancement préemptif

- Beaucoup de sources d'incertitude dans l'analyse
  - Date de premier départ des tâches
  - Durée effective de chaque tâche
    - Une tâche se terminant plus tôt n'améliore pas toujours les temps de réponse : notion d'**anomalie temporelle**
  - Et en général, c'est pire : gigue, allocation à choisir sur un multiprocesseur, allocation mémoire...
- Incertitude+analyse au pire cas+complexité => coût en précision de l'analyse

# ARINC 653

# Implantation des systèmes embarqués

Graphes  
flot de données  
déterministes

Spécification fonctionnelle

Spécification non-fonctionnelle

**Mapping**  
Dans l'espace (allocation)  
Dans le temps (ordonnancement)  
Calculs et communications

Correction  
fonctionnelle

Implantation  
exécutable

Exigences

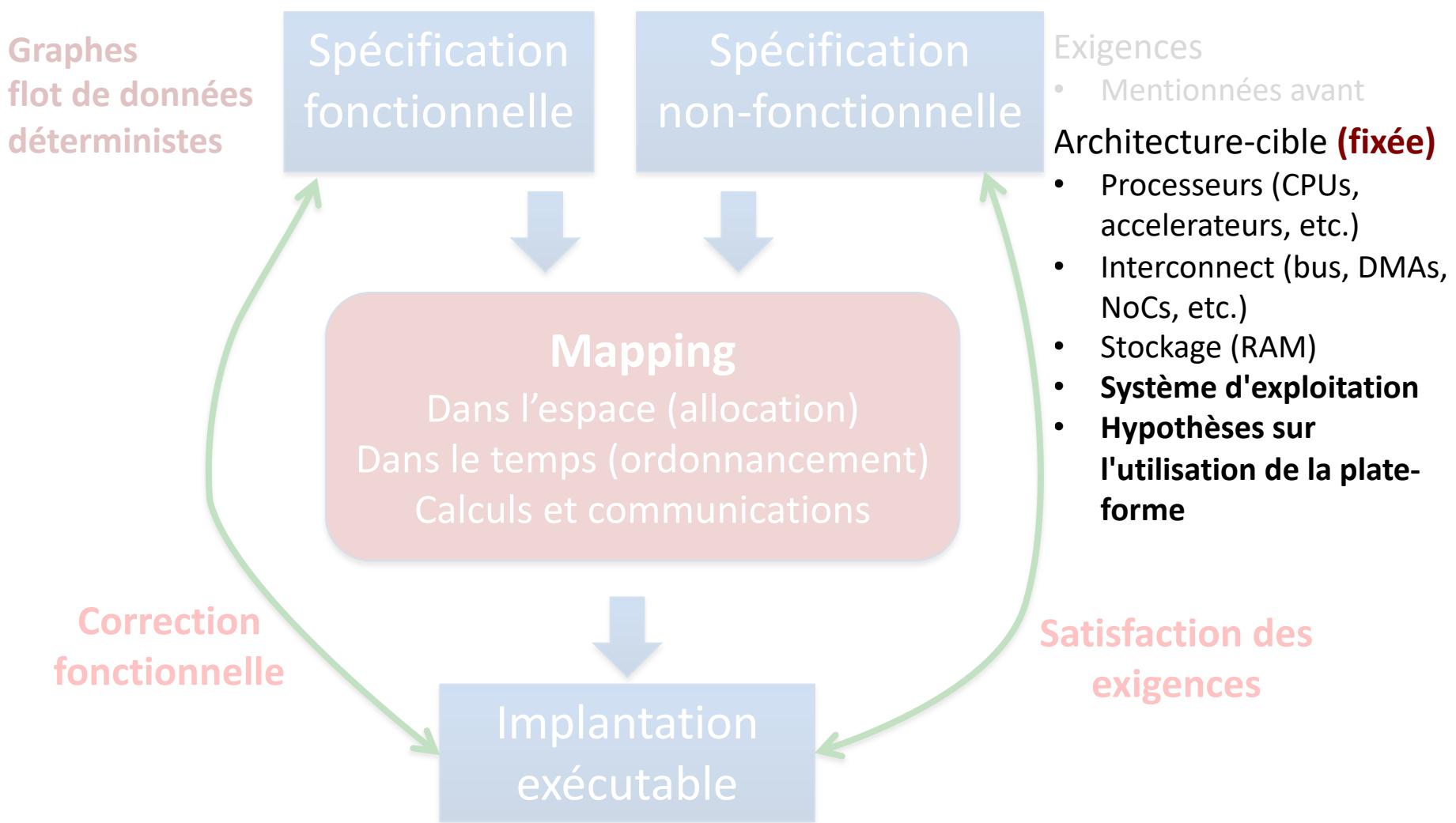
- Mentionnées avant

Architecture-cible (**fixée**)

- Processeurs (CPUs, accélérateurs, etc.)
- Interconnect (bus, DMAs, NoCs, etc.)
- Stockage (RAM)
- Système d'exploitation**
- Hypothèses sur l'utilisation de la plate-forme**

Satisfaction des  
exigences

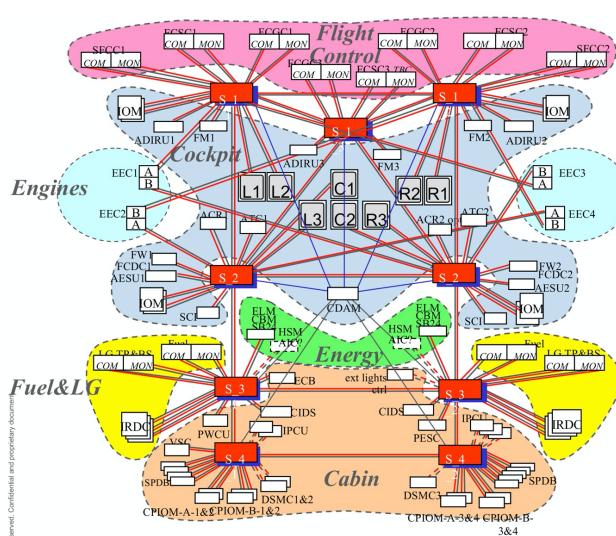
# Implantation des systèmes embarqués



# Systèmes avioniques

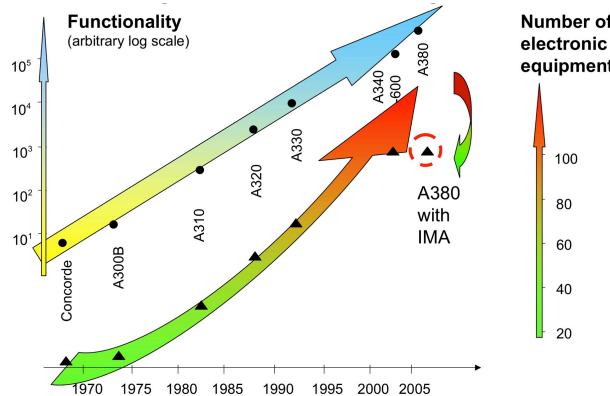
- Les systèmes embarqués avioniques :
  - Complexes, répartis, temps réel, soumis à la certification

# Systèmes avioniques



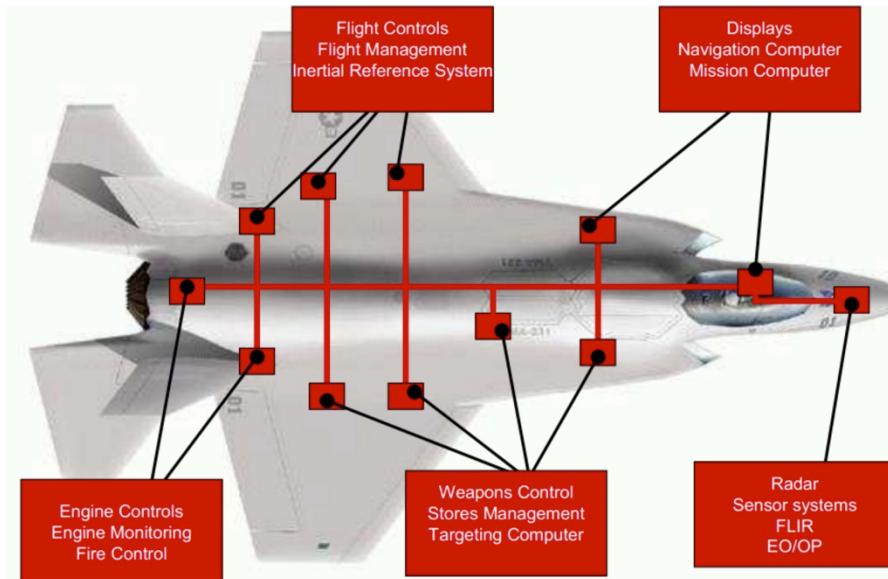
## AFDX Network:

- 100 Mbits
- Redundant Network (A&B) with independent alimentation
- AFDX switches = 2 x 8
- NB of ports (connections) possible on each switch (20 to 24)
- MTBF of the switch is very high (100 000 hours expected)
- Up to 80 AFDX subscriber



# Federated avionics

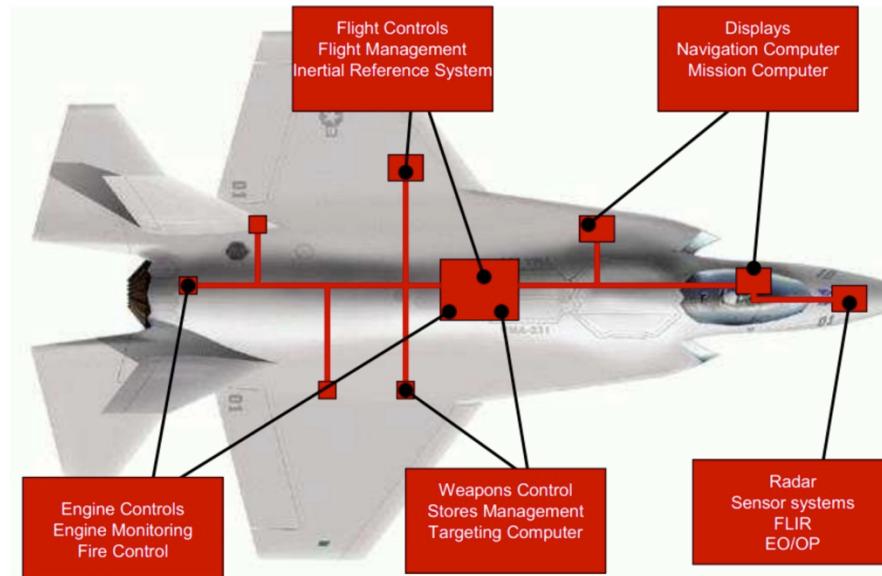
- Les fonctions/sous-systèmes sont fournies sous la forme d'unités fermées (self-contained).
  - LRU/LRM = line-replaceable units/modules
- Pas de partage de ressources entre fonctions/sous-systèmes



<https://docplayer.net/287772-Arinc-653-an-avionics-standard-for-safe-partitioned-systems.html>

# Integrated Avionics

- Architectures intégrées
  - Partage de ressources entre fonctions/sous-systèmes
    - Temps CPU, bande passante réseau, etc.



<https://docplayer.net/287772-Arinc-653-an-avionics-standard-for-safe-partitioned-systems.html>

# Federated vs. Integrated Avionics

- Avantages des architectures intégrées
  - Réduction de volume/poids
    - Moins de calculateurs/alimentations
    - Moins de fils
  - Efficacité énergétique (consommation/dissipation)
  - Meilleur support pour x-by-wire (moins de fils !)
  - Adapté à des systèmes plus complexes
- Désavantages
  - Plus difficile d'assurer sûreté et sécurité lorsque des **ressources sont partagées**
    - Temps réel
- Lecture supplémentaire :
  - New Challenges for Future Avionic Architectures. Bieber et al. 2012

# ARINC 653

- Norme pour le système d'exploitation permettant le partage de ressources de calcul IMA
  - Norme ARINC (Aeronautical Radio, Inc., propriété de Rockwell Collins)
- Titre officiel : "Avionics application software standard interface"
  - 6 documents
    - Part 0 – Introduction
    - Part 1 – Required services
    - Part 2 – Extended services
    - Part 3 – Conformity tests
    - Part 4 – Subset services
    - Part 5 – Core software recommended capabilities

# ARINC 653

- Version de base : exécution sur un seul processeur de plusieurs applications multi-tâches, assurant la non-interférence entre les applications.
  - Extensions déjà définies ou en cours
    - Concurrence restreinte (1 tâche prioritaire + 1 tâche de fond par cœur de calcul)
    - Multi-cœur – "CAST32A"
      - [https://www.sysgo.com/fileadmin/user\\_upload/www.sysgo.com/data/SYSGO\\_Presentation\\_ARINC653RTOS\\_for\\_multi-core\\_certification.pdf](https://www.sysgo.com/fileadmin/user_upload/www.sysgo.com/data/SYSGO_Presentation_ARINC653RTOS_for_multi-core_certification.pdf)
- Toute un vocabulaire spécifique à l'avionique embarquée :
  - Module = processeur utilisé sous ARINC 653
  - Process = tâche avec une forme spécifique

# Concept fondamental : TSP

- **TSP** = Time-Space Partitioning ( = partitionnement spatial et temporel)
  - Partition = concept dual
    - Application (mono- ou multi-tâche) à isoler des autres
    - Ensemble de ressources (temps CPU, RAM, périphériques) alloués à l'exécution d'une application
  - L'exécution d'une application/partition sur ses ressources n'influence pas l'exécution des autres partitions
    - Seule exception : canaux de communication explicitement définis
- Séparation des activités de conception entre entre l'intégrateur système et le concepteur d'applications
  - La définition du partitionnement est du niveau système<sup>38</sup>

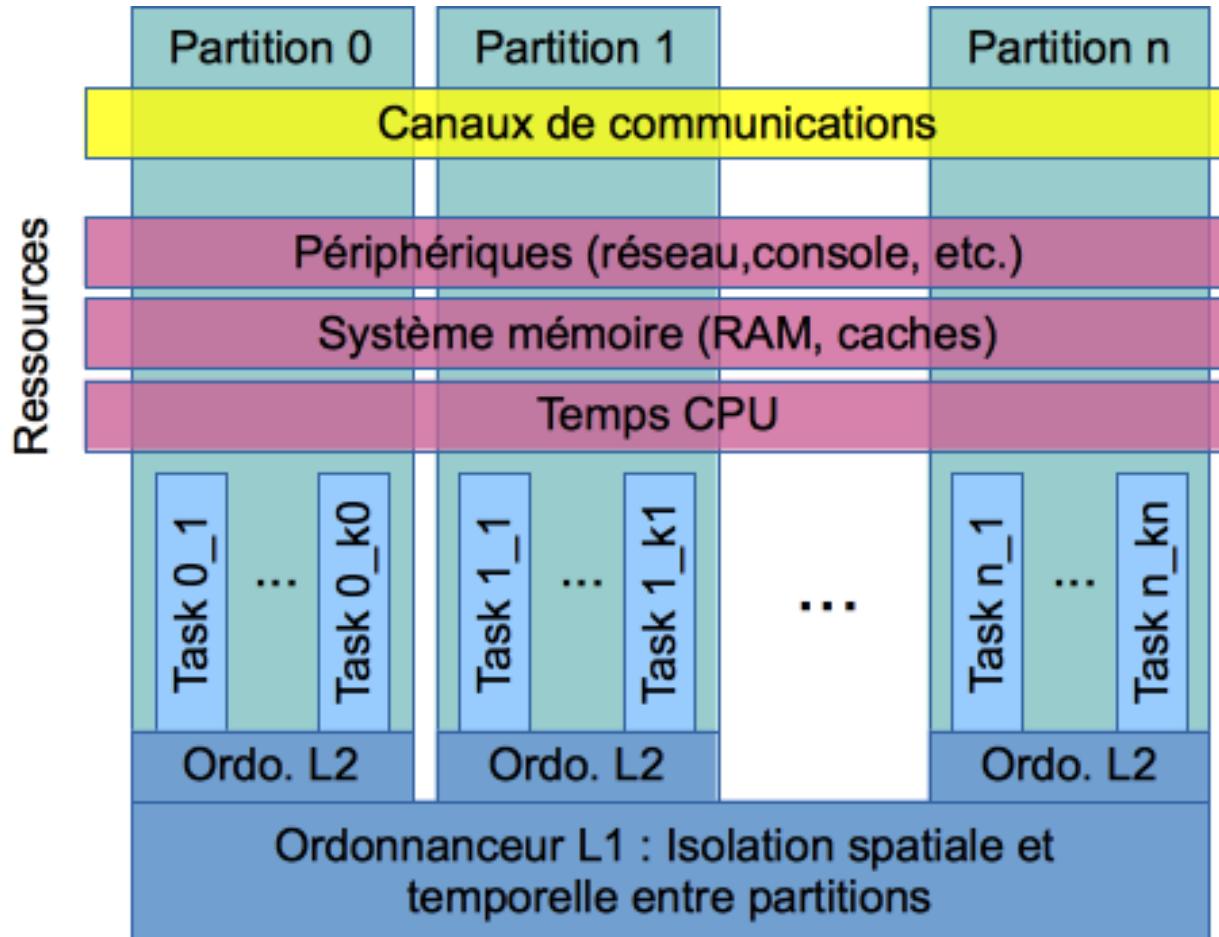
# Partitionnement spatial

- Une partition ne doit pas pouvoir accéder directement aux ressources privées d'une autre partition
  - Toutes les données et le code sont une ressource privée
    - Pas de mémoire partagée
    - Utilisation de la MMU pour isoler les zones mémoire
  - Mais aussi :
    - État des caches
    - État des périphériques partagés (qui contient des données privées)
- Relativement facile à obtenir, en théorie
  - Architectures modernes (MMU)
  - Support des outils de développement existants (code relocable, etc.)
  - Divers bugs la rendent plus difficile (Meltdown, Spectre)

# Partitionnement temporel

- L'exécution d'une partition ne doit pas être influencée temporellement par les autres
  - Extrêmement difficile à obtenir en pratique, car l'efficacité implique l'utilisation de la hiérarchie mémoire
    - Ordonnancement TDM au niveau des partitions
    - Etat des caches
      - Caches partagés ou flush complet nécessaire lors du changement de partition
    - Etat des périphériques en accès partagé
      - Remise à zéro nécessaire lors du changement de partition
    - Etat du pipeline (e.g. spéculation)
      - Laisser le temps pour le vidage
    - Ordonnancement multi-cœur très difficile
      - Grandes marges de sûreté
  - Coût temporel élevé (analyse au pire cas)
  - Souvent relaxé (en fonction de l'application)
    - Pas de flush des caches, parfois même multi-coeurs, etc.

# Organisation d'un système Arinc 653



# Niveau module (L1)

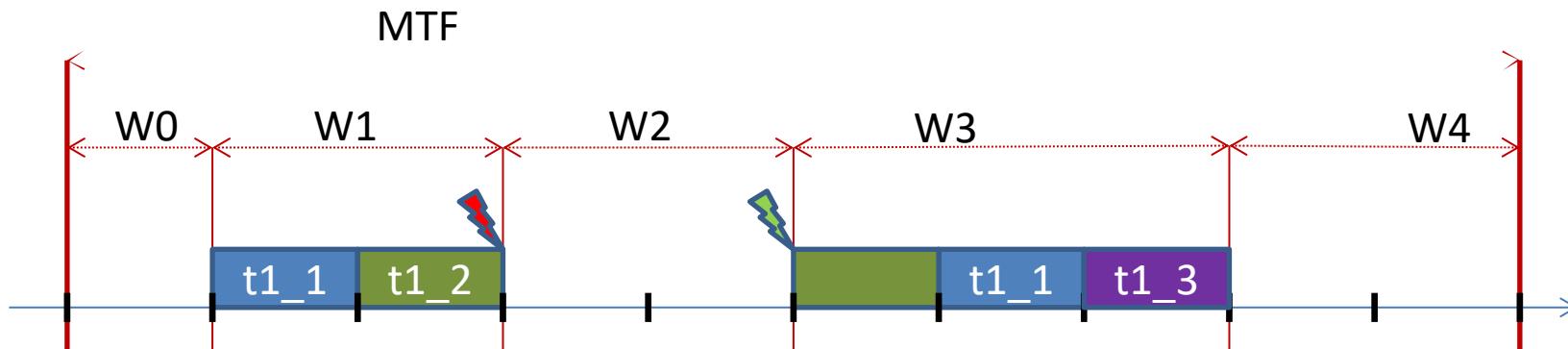
- Fonctions
  - Boot & configuration du système
  - Gestion des interruptions
    - Pilotes de périphériques (en partie)
    - Gestion du temps système
  - Partitionnement spatial
    - Gestion de la MMU et des caches
  - Partitionnement temporel
    - Ordonnancement des partitions
  - Canaux de communications entre partitions
  - Health monitoring and recovery

# Niveau module (L1)

- Ordonnancement TDM de type multiplexage temporel statique
  - Time-triggered = déclenchement par le temps
  - Motif fixé = Major time frame (MTF)
  - MTF divisé en fenêtres temporelles (windows)
  - Chaque fenêtre est statiquement allouée à une partition
    - Chaque partition doit avoir au moins une fenêtre
  - Exécution = répétition infinie du MTF
    - Le code d'une partition s'exécute pendant les fenêtres qui lui sont associées

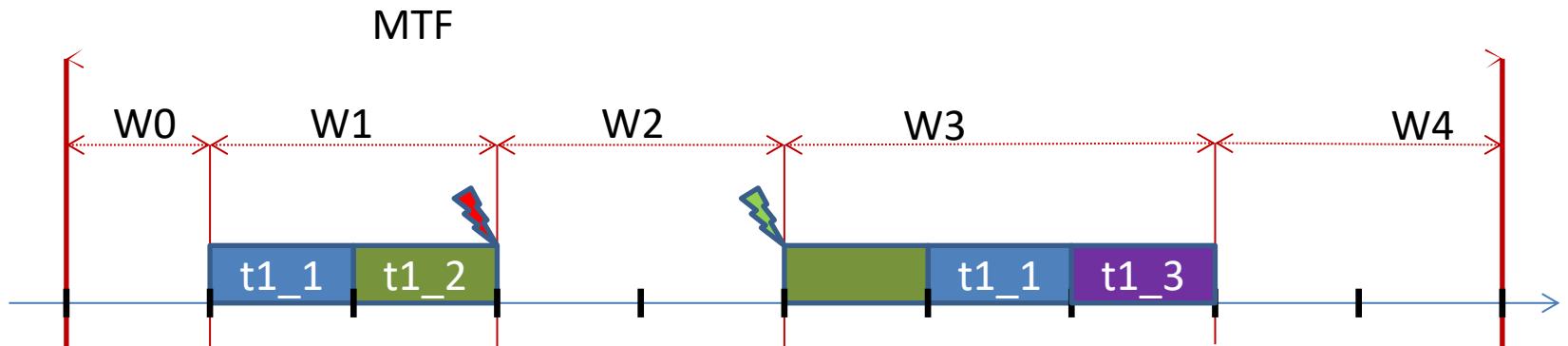
# Niveau module (L1)

- Ordonnancement TDM de type multiplexage temporel statique
  - Souvent le temps est divisé en ticks (quantas de temps de taille fixe).
  - Exemple : tick = 500usec, MTF = 5ms, 5 fenêtres (W0-W4), dont W1, W3 associées à la partition 1
    - 3 tâches de périodes MTF/2, MTF, et MTF respectivement



# Niveau module (L1)

- Ordonnancement TDM de type multiplexage temporel statique
  - En fin de fenêtre, le code d'une partition est interrompu
    - Permettre ou non l'utilisation d'opérations non-interruptibles, en fonction du niveau d'isolation temporelle souhaité
  - Choix de modélisation (abstraction)
    - L'exécution de l'ordonnanceur L1 prend du temps sur le temps des partitions (timer, preemptions, changements de partition)



# Niveau module (L1)

- Canaux de communication
  - Envoi et réception de messages
    - Message = ensemble de données transmises d'une manière atomique
  - Une partition source, une ou plusieurs partitions destination
    - Seul mécanisme de communication pour une partition "applicative"
    - "Pseudo-partitions" ou partitions système permettent de communiquer avec les pilotes de périphériques

# Niveau module (L1)

- Canaux de communication
  - Accès aux canaux par des ports d'entrée et de sortie
    - Sampling ports – periodic sending, regardless of new data arriving or consumption
    - Queuing ports – aperiodic sending and receiving, upon call of API functions
  - Le niveau L1 gère la copie des données entre partitions.
    - Aucun partage de mémoire entre partitions

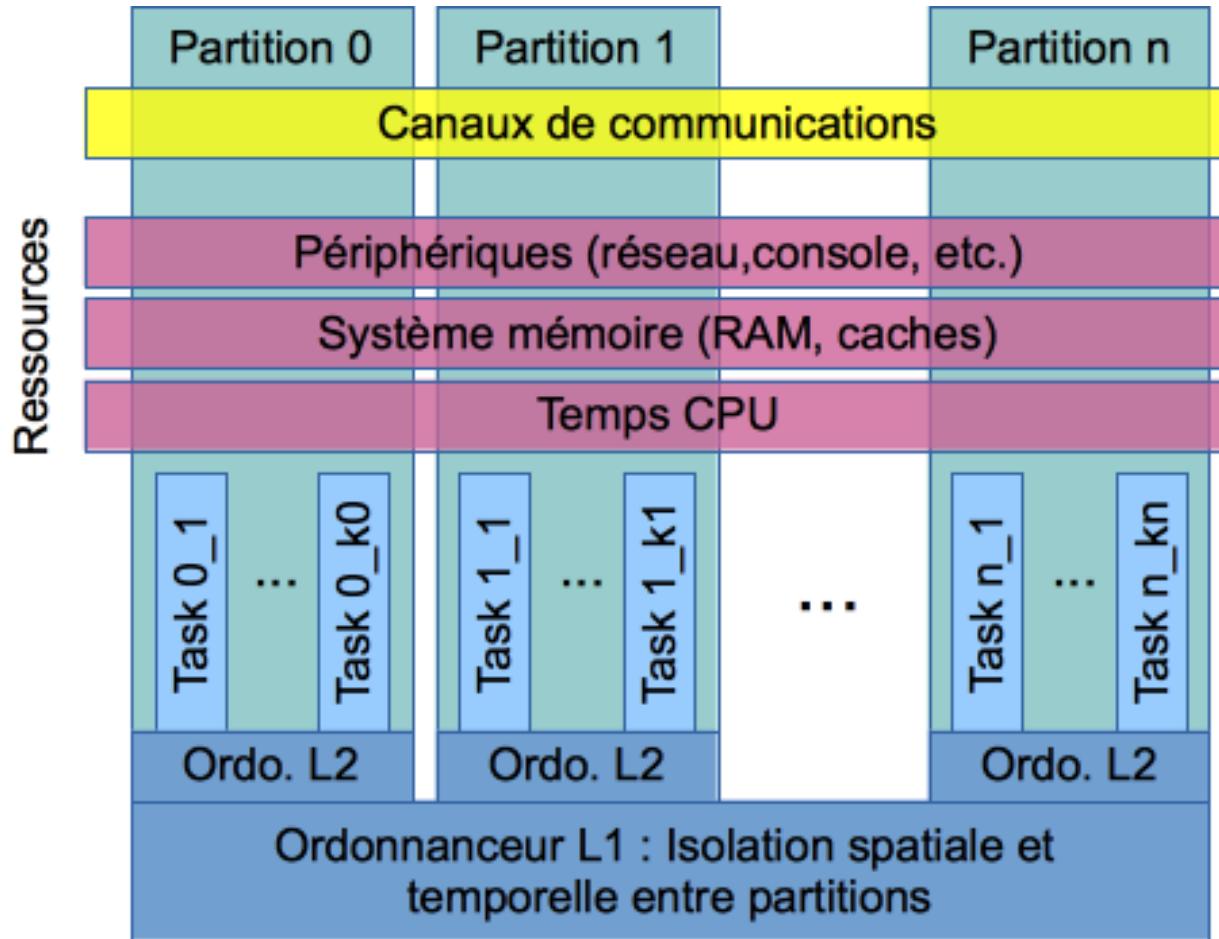
# Niveau module (L1)

- Health monitoring
  - Surveillance des erreurs au niveau du module et des partitions
  - Gestion des erreurs
    - Réactions tabulées
      - **Tout est fait pour faciliter la vérification de la correction globale**

# Niveau module (L1)

- Configuration par fichier XML
  - Définition des partitions
    - Configuration de l'espace mémoire
    - Partitions applicatives, partitions système, pseudo-partitions
  - Configuration du (ou des) MTF
    - Taille
    - Découpage en fenêtres. Pour chaque fenêtre :
      - Taille
      - Allocation
      - Est-elle "partition period start" (**PPS**) ?
  - Configuration des communications entre partitions
    - Ports des partitions
    - Canaux connectant ces ports
  - Configuration du “Health monitoring”

# Organisation d'un système Arinc 653



# Niveau partition (L2)

- Fonctions
  - Interface application/OS – API APEX
    - Gestion de la vie d'une application/partition
      - Changements de mode
  - Ordonnancement des tâches
    - Création de tâches, changement des paramètres d'ordonnancement, etc.
  - Communications inter-partitions
    - Configuration des ports, envoi, réception
  - Communications intra-partition
  - Actions HM
- Interface APEX standardisée pour C et Ada

# Niveau partition (L2)

- Modes d'une partition
  - COLD\_START and WARM\_START
    - Lors du démarrage de ces modes, le contrôle est donné au point d'entrée de la partition. Ce code s'exécute séquentiellement jusqu'au passage en mode NORMAL or IDLE.
    - La création des tâches et des ports de communication doit être réalisée en mode \* \_START (pas de création dynamique)
    - Les tâches créées peuvent être marquées comme démarées, mais leur exécution ne commencera qu'au passage en mode NORMAL
  - NORMAL
    - Lors du démarrage de ce mode, l'exécution concurrente des tâches démarées peut commencer.
  - IDLE
    - Aucun code n'est exécuté. Ce mode ne peut être quitté que par une action au niveau L1 ou par redémarrage du module.

# Niveau partition (L2)

- Changements de mode :
  - Appels à SET\_PARTITION\_MODE

```
RETURN_CODE_TYPE rc ;  
...  
SET_PARTITION_MODE(<new_mode>,&rc) ;
```

- Actions “Health monitoring”

# Niveau partition (L2)

- Point d'entrée de la partition
  - Fonction séquentielle démarrée lors du lancement d'un des modes \* \_START
    - Attention : cela peut se produire plus d'une fois au cours d'une exécution
  - Seul code à s'exécuter dans la partition avant passage en mode NORMAL ou IDLE.
    - Une partition peut rester indéfiniment en mode START
    - Il n'est pas spécifié ce qui se passe avec le point d'entrée lors du passage en mode NORMAL
      - Dépendant de l'implémentation
      - Souvent, son exécution s'arrête lors du changement de mode
  - Exécution séquentielles, mais préemptive !
    - Interrompue par les fins de fenêtres de la partition
  - Création de toutes les tâches
    - Certaines tâches peuvent être marquées comme démarrées
  - Initialisation de tous les ports I/O

# Niveau partition (L2)

- Le mode NORMAL
  - Ordonnancement préemptif de tâches
    - Le standard, à travers l'API APEX, spécifie une exécution dirigée par les priorités
      - À chaque instant, parmi les tâches éligibles pour exécution, une tâche de la priorité la plus haute s'exécute
    - Les priorités des tâches ont une valeur initiale, mais peuvent être modifiées dynamiquement
      - Appels API explicites (EDF difficile à implémenter)

# Niveau partition (L2)

- Caractéristiques des processus (tâches) :
  - Fixes
    - Nom – une chaîne de caractères
    - Point d'entrée – une fonction
    - Taille de la pile – qui sera allouée par l'ordonnanceur
    - Priorité de base – la période au moment du démarrage de la tâche
    - Période – pour les tâches périodiques (ou une valeur spéciale identifiant les tâches apériodiques)
    - Capacité – durée servant à calculer l'échéance
    - Type d'échéance – HARD ou SOFT
  - Variables
    - Priorité courante
    - Echéance courante
    - Etat du processus

# Niveau partition (L2)

- Processus périodiques
  - Arrivées périodiques par rapport à une date de première arrivée
  - Date de première arrivée
    - Date de référence
      - En mode COLD\_START/WARM\_START – la date de début de la première fenêtre de type “partition period start” de la partition après le passage en mode NORMAL
      - En mode NORMAL – la date de début de la première fenêtre de type “partition period start” de la partition après la date courante
    - Démarrage avec START → date de référence
    - Démarrage avec DELAYED\_START → comme pour START, plus un incrément
  - Échéances – par rapport aux dates d’arrivée
    - Possibilité de changer dynamiquement d’échéance

# Niveau partition (L2)

- Processus périodiques
  - Primitive APEX PERIODIC\_WAIT
  - Exemple :

```
void task0() {
    RETURN_CODE_TYPE rc ;/* Utilisé par les appels API */
    int i;
    for(i=0;;i++) {
        debug_printf("task0 instance %d\n",i) ;
        /* Séparateur d'instances. Doit être appelé dans
           chaque période */
        PERIODIC_WAIT(&rc) ;
        console_perror(rc,"mypartition","PERIODIC_WAIT") ;
    }
    /* La fonction ne doit pas se terminer */
}
```

# Niveau partition (L2)

- Processus périodiques
  - Création d'une tâche

```
PROCESS_ATTRIBUTE_TYPE pat;
PROCESS_ID_TYPE pid;
RETURN_CODE_TYPE rc;
...
pat.PERIOD = <value_in_NS> ;
pat.TIME_CAPACITY = <value_in_NS> ;
pat.ENTRY_POINT = (void*)<function>;
pat.STACK_SIZE = <value_in_bytes> ;
pat.BASE_PRIORITY = <integer> ;
pat.DEADLINE = HARD ; /* ou SOFT */
strcpy(pat.NAME,"matache") ; /* Attention à la taille max */
CREATE_PROCESS(&pat,&pid,&rc) ;
/* vérifie si tout s'est bien passé */
console_perror(rc,<part_name>,<message>) ;
...
```

# Niveau partition (L2)

- Processus périodiques
  - Démarrage

```
PROCESS_ID_TYPE pid;
RETURN_CODE_TYPE rc;
...
START(pid,&rc);
or
DELAYED_START(pid,<value_in_NS>,&rc) ;
/* vérifie si tout s'est bien passé */
console_perror(rc,<part_name>,<message>);
...
```

# Préparation du TP

- Interaction avec le matériel
  - Temps réel "mou" sur votre ordinateur portable
    - Traitement de signal
    - Établir un lien entre code Heptagon et le système audio de l'ordinateur
    - Programmation de la FFT/IFFT en flux
    - **Pour la prochaine fois : micro et casque nécessaires**
  - Installation d'un compilateur ARM pour la RPi
    - Exécution de code sur carte Raspberry Pi

# Objectif 0

- Installer SoX sur votre ordinateur
  - Linux (Ubuntu/Debian) : sudo apt-get install sox
    - Directement sur la machine (pas sous VM), si votre machine comporte un OS Linux
      - MaxOSX : sudo port install sox
        - » Gestionnaire de paquets macports à installer

# Objectif 0

- Tester l'installation de SoX :
    - Enregistrer du son :

```
rec -t raw -r 44100 -e signed -b 16 -c 2 snd.raw
```
    - Changer la fréquence d'échantillonage du son :

```
sox -t raw -r 44100 -e signed -b16 -c 2 snd.raw
```

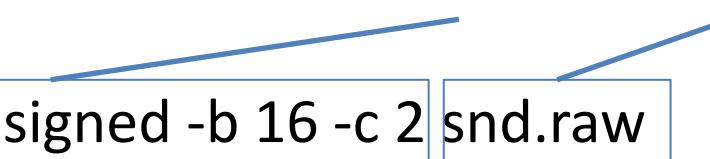
```
-t raw -r 8k -e signed -b16 -c 2 snd1.raw
```
    - Reproduire du son :

```
play -t raw -r 8k -e signed -b 16 -c 2 snd1.raw
```

# Objectif 0

- Tester l'installation de SoX :
  - Enregistrer du son :

```
rec -t raw -r 44100 -e signed -b 16 -c 2 snd.raw
```



Attention : votre matériel peut forcer certaines configurations

– En appellant "rec –t raw myfile.raw" vous aurez la configuration à respecter

– Attention : ne pas faire du copier-coller de ces commandes (les caractères ne sont pas les bons).

# Objectif 0

- Les fichiers d'entrée et de sortie peuvent être remplacés par "-" pour signifier l'utilisation de pipes
  - Possibilité de créer des chaînes de traitement



```
rec -t raw -r 44100 -e signed -b 16 -c 2 - |  
sox -t raw -r 44100 -e signed -b16 -c 2 -  
-t raw -r 8k -e signed -b16 -c 2 - rate 8k |  
play -t raw -r 8k -e signed -b 16 -c 2 -
```

# Objectif 1

- Créer une chaîne de traitement où sox est remplacé par du code synthétisé par nous avec Heptagon
  - Très simple (juste un changement de volume)



```
rec -t raw -r 44100 -e signed -b 16 -c 2 - |  
./myprog |
```

```
play -t raw -r 44100 -e signed -b 16 -c 2 -
```

# Objectif 1

- Application Heptagon : myprog
  - Lecture des échantillons d'entrée, 256 par 256
  - Configuration : "-t raw -r 44100 -e signed -b 16 -c 2"
    - Chaque échantillon est formé de 2 entiers sur 16 bits, signés, en format "little endian" (le même que votre machine)

```
struct sample {  
    int16_t l;  
    int16_t r;  
}
```
  - Le code C de lecture/ecriture est fourni dans sndlib.[ch]
  - Essayez-le !
    - Créez un programme qui, cycliquement, lit 256 échantillons de stdin et les écrit sur stdout

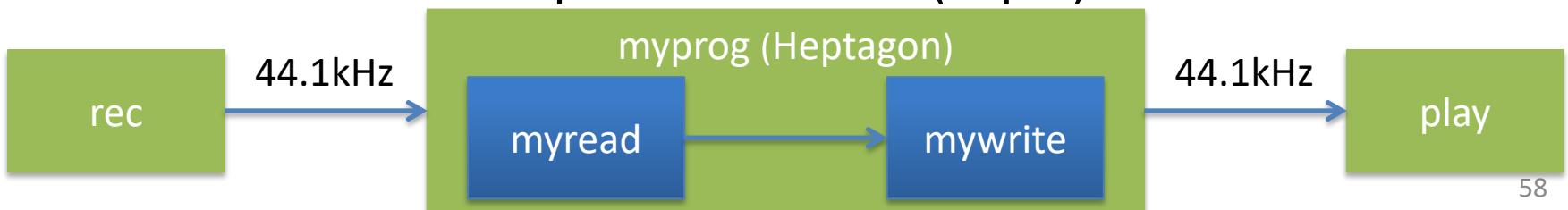
# Objectif 1

- Application Heptagon : myprog
  - Interface Heptagon :

fun myread(size:int) returns (samples:float<sup>^256</sup>)

fun mywrite(size:int;samples:float<sup>^256</sup>) returns ()

- Le paramètre size n'est pas strictement nécessaire, car nous connaissons la taille des données.
  - Cependant, il permet d'écrire un code C qui ne change pas si on change la taille des données
- Avec ces 2 fonctions externes, créer le 1<sup>er</sup> programme flot de données qui traite du son (copie)

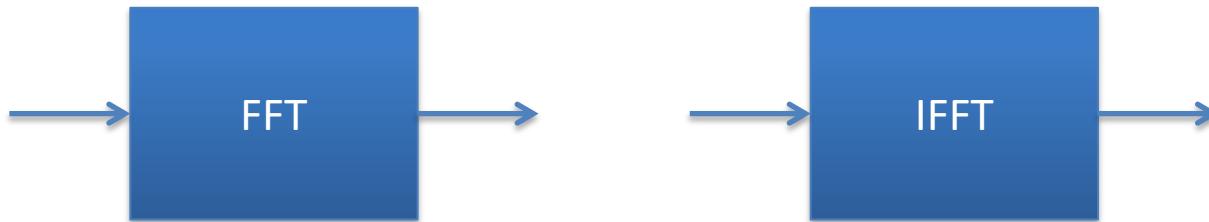


# Objectif 1

- Application Heptagon : myprog
  - Faire le lien entre l'interface Heptagon et le code C
  - Ecrire le programme Heptagon, et compiler l'ensemble
    - Exécutez-le dans une chaîne avec rec et play (effet "echo")
    - Attention : il faut fixer les mêmes paramètres
  - Une fois le programme qui fait la copie finalisé, transformez-le pour changer le volume du son en sortie
    - Diminuer le volume de 50%
      - En divisant chaque sample par 2 – utilisation de map
      - Attention, les données sont de type float (opérateurs \*., +., ...)

# Objectif 2

- TP précédent
  - Programmer la FFT et la IFFT



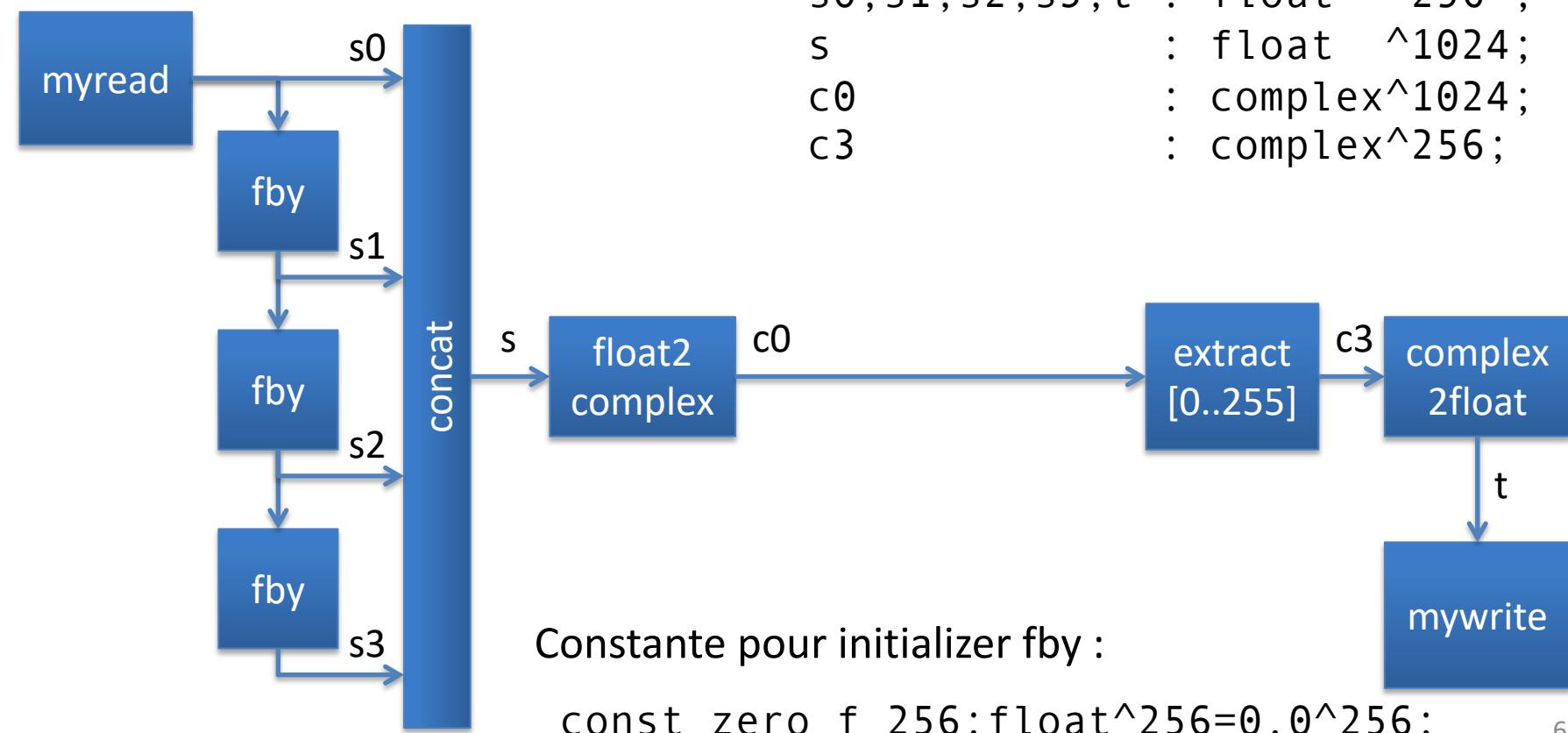
- Nouvelle tâche : Connecter les 2 ensemble

# Objectifs 2 et 3

- Flot de données à réaliser :
  - Passage en domaine fréquentiel et retour
    - Lecture d'échantillons par paquet de 256
    - En faire une fenêtre glissante de 1024 échantillons
      - Attention: dans un vecteur d'échantillons  $v$ ,  $v[0]$  est l'échantillon le plus ancien.
      - Attention lors de la construction de la fenêtre glissante.
        - » Les derniers échantillons arrivent en dernier
    - Application de la FFT, puis de la IFFT
    - Ecriture des 256 échantillons les plus anciens
  - Construction en deux temps

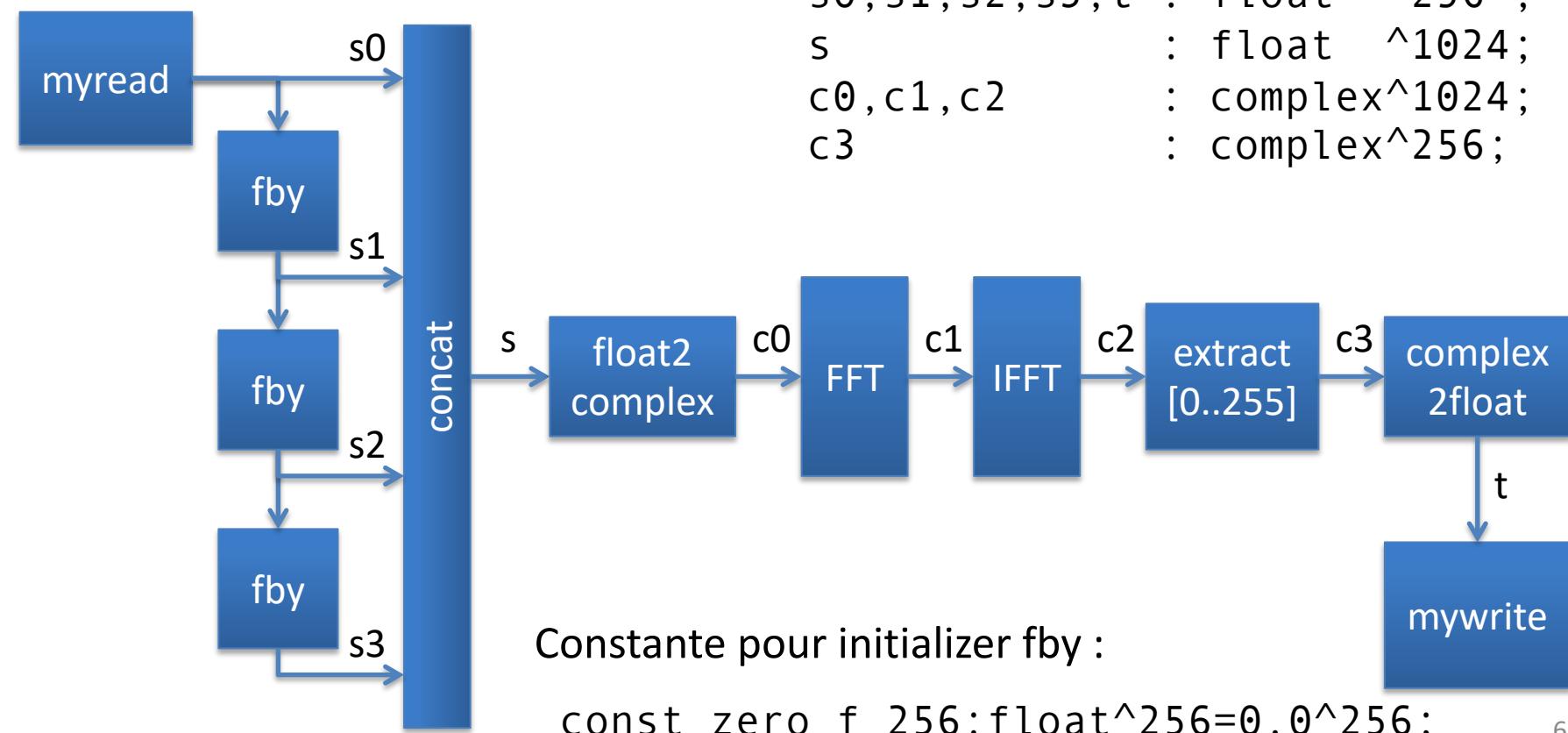
# Objectif 2

- Flot de données à réaliser :



# Objectif 3

- Flot de données à réaliser :



# Objectif 4 - détaillé en fiche séparée

- Installation d'un compilateur pour processeur ARM
  - Cible : Raspberry Pi 1
- Compilation des sources de l'OS RPi653
- Exécution sur carte Raspberry Pi