

## Séance 2

# Modèles des systèmes embarqués



Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons  
Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

# Rappels

- Exploitation et développement de **systèmes embarqués**
  - Définition, environnement et caractéristiques
  - Design d'OS pour système embarqué from scratch ou general
- Développement d'un OS **general-purpose**

*Le cas de Linux embarqué et ses caractéristiques*
- Développement d'un **OS spécifique** pour systèmes embarqués

*Le cas de TinyOS et réseau de senseurs sans fil*

# Objectifs

- **Modèles de structure de programme pour systèmes embarqués**
  - Le modèle super-loop
  - Le modèle orienté évènements et interruptions
  - Le modèle basé sur les processus
- **Méthodes de design** du programme d'un système embarqué
  - Modèle des machines à états finis (FSM) et statecharts*

# Système embarqué

- Système embarqué d'abord designé pour **application spécifique**
  - Système embarqué construit autour d'un microcontrôleur
  - Monitoring de senseurs, génération de signaux de contrôle
- Premier **programme de contrôle** par nature très simples
  - Structure de type super-loop ou orienté événements
  - Rapidement limité suite à augmentation de fonctionnalités
  - Possible aujourd'hui d'exécuter des OS à part entière



Modèle super-loop

# Super-loop (1)

- Structure du programme composée d'**une boucle infinie**

*Toutes les tâches sont contenues dans la boucle*

```
1 main()
2 {
3     initialisation();           // initialisation du système
4
5     while (1) {
6         check_device();         // examination de l'état du périphérique
7
8         process_data();        // traitement des données
9
10        output_response();    // production d'une sortie
11    }
12}
```

# Super-loop (2)

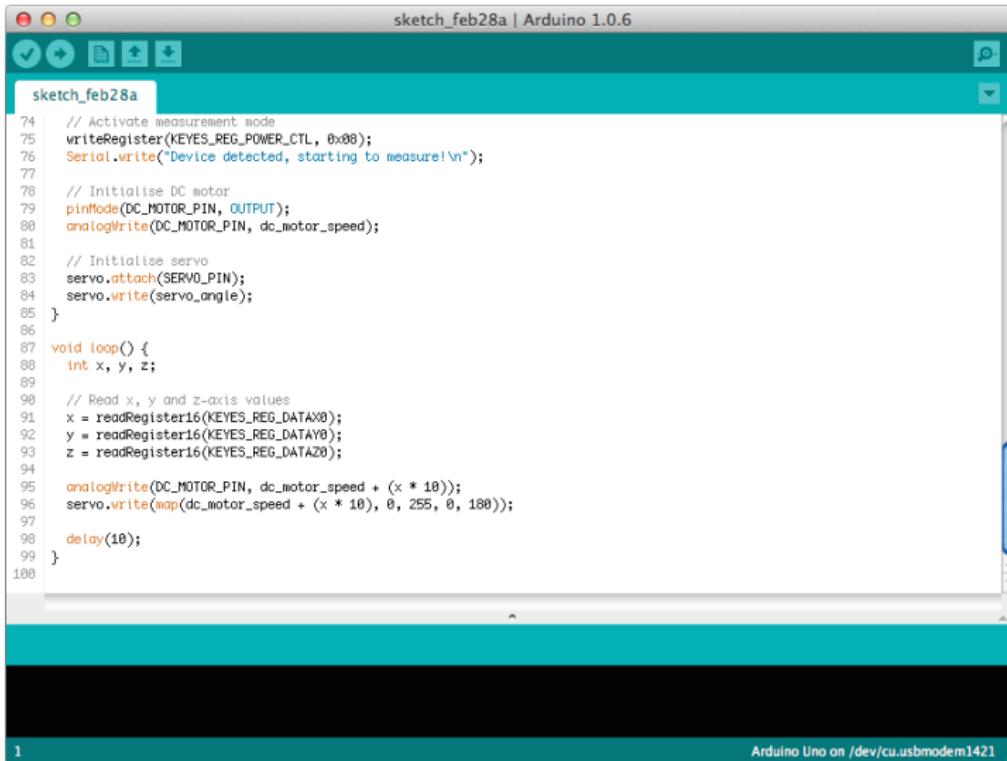
- **Initialisation** du système à son démarrage

*Se produit une seule fois pour préparer le système pour son travail*

- Exécution d'une **boucle infinie** avec trois étapes

- 1 Vérification du statut de composants du système
- 2 Récolte et traitement des données prêtes venant du composant
- 3 Génération d'une réponse en guise de sortie

# Programmation Arduino



The screenshot shows the Arduino IDE interface with a sketch titled "sketch\_feb28a" open. The code is written in C++ and performs the following tasks:

- Activates measurement mode by writing to the KEYES\_REG\_POWER\_CTL register.
- Prints a message to the Serial port indicating the device has detected and is starting to measure.
- Initializes a DC motor connected to pin DC\_MOTOR\_PIN with an output mode.
- Sets the analog write value for the DC motor based on the measured values.
- Initializes a servo connected to pin SERVO\_PIN.
- Attaches the servo to the specified pin.
- Writes the servo angle based on the measured values.
- Enters a loop where it reads x, y, and z-axis values from registers KEYES\_REG\_DATAx0, KEYES\_REG\_DATAy0, and KEYES\_REG\_DATAz0 respectively.
- Adjusts the DC motor speed by adding a multiplier of 10 to the x-axis value.
- Maps the adjusted DC motor speed and the x-axis value to a range of 0 to 255.
- Writes the mapped value to the servo.
- Delays the process for 10 milliseconds.

```
sketch_feb28a | Arduino 1.0.6

sketch_feb28a

74 // Activate measurement mode
75 writeRegister(KEYES_REG_POWER_CTL, 0x08);
76 Serial.write("Device detected, starting to measure!\n");
77
78 // Initialise DC motor
79 pinMode(DC_MOTOR_PIN, OUTPUT);
80 analogWrite(DC_MOTOR_PIN, dc_motor_speed);
81
82 // Initialise servo
83 servo.attach(SERVO_PIN);
84 servo.write(servo_angle);
85 }
86
87 void loop() {
88     int x, y, z;
89
90     // Read x, y and z-axis values
91     x = readRegister16(KEYES_REG_DATAx0);
92     y = readRegister16(KEYES_REG_DATAy0);
93     z = readRegister16(KEYES_REG_DATAz0);
94
95     analogWrite(DC_MOTOR_PIN, dc_motor_speed + (x * 10));
96     servo.write(map(dc_motor_speed + (x * 10), 0, 255, 0, 180));
97
98     delay(10);
99 }

1
Arduino Uno on /dev/cu.usbmodem1421
```

# Contrôle port UART

- Contrôle d'un **port UART** pour une entrée/sortie
  - Vérification permanente de la présence d'une entrée sur le port
  - Récupération donnée et transformation en majuscule

```
1 #define UDR 0x00
2 #define UFR 0x18
3 char *ubase;
4
5 int main()
6 {
7     char c;
8     ubase = (char *) 0x101F1000;           // 1. Initialiser adresse base UART0
9     while (1) {
10         if (*(ubase + UFR) & 0x40) {        // 2. Vérifier RxFull sur UART0
11             c = *(ubase + UDR);            // 3. Récupérer input UART0
12             if (c >= 'a' && c <= 'z')      // si c'est une minuscule
13                 c += ('A' - 'a');          // convertir en majuscule
14             *(ubase + UDR) = c;           // 4. Générer la sortie
15             if (c == '\r')
16                 *(ubase + UDR) = '\n';
17         }
18     }
19 }
```

# Désavantage

- Vérification continue de l'état de tous les périphériques
  - Utilisation inutile du CPU dans grande proportion de temps
  - Consommation énergétique excessive
- Préférable d'attendre que le périphérique soit prêt

```
while (device_has_no_data);
```
- Attente active d'un périphérique ne solutionne rien
  - Minimisation consommation vs maximisation utilisation CPU
  - Empêche la réponse à d'autres périphériques qui seraient prêts

# Modèle orienté évènements



# Évènement

- Système embarqué designé pour être **orienté évènements**
  - Un évènement est quelque chose produit par une source
  - Ce dernier est ensuite reconnu par un destinataire
  - L'évènement est traité par ce dernier qui prend une action
- Deux types d'**occurrences d'évènements**
  - **Synchrone** si ils se produisent de manière prédictible  
*Évènements produits par un timer*
  - **Asynchrone** si n'importe quand et dans n'importe quel ordre  
*Évènements produits par un utilisateur (clavier, souris...)*

# Super-loop vs évènement

- Super-loop pas adaptée pour les évènements asynchrones  
*À cause de la grande imprédicibilité des occurrences*
- En orienté évènements, attente dans une **boucle ou en idle**
  - Un gestionnaire reconnaît l'évènement particulier produit
  - Exécution d'une action appropriée selon l'évènement

# Modèle orienté interruptions

- Association d'évènements à des **interruptions hardware**

*On parle alors de modèle orienté interruptions*

- **Deux possibilités** de structure pour réagir à des interruptions
  - Réaction à des interruptions périodiques (timer...)
  - Interruptions non périodiques (UART, clavier...)

# Affichage de l'heure (1)

- Affichage de l'heure sur un écran LCD à partir d'un timer
  - Le timer produit une interruption 60 fois par seconde
  - Mise à jour du LCD doit se produire toutes les secondes
  - Affichage d'un texte particulier toutes les cinq secondes

```
1 void IRQ_handler()
2 {
3     // ...
4     if (vicstatus & (1<<4))
5         timer_handler();
6 }
7
8 int main()
9 {
10    // ... initialisation ...
11    while (1) {
12        asm("MOV r0, #0; MCR p15,0,R0,c7,c0,4");      // mettre CPU en état WFI
13    }
14 }
```

# Affichage de l'heure (2)

- Gestionnaire d'interruption dans une fonction dédiée
  - Mets à jour un compteur du nombre de ticks et du temps
  - Affichage de l'heure (/1s) et du texte particulier (/5s)

```
1 void timer_handler()
2 {
3     TIMER *t = &timer;
4     t->tick++;
5     if (t->tick == 60) {
6         // ... mettre à jour le temps ...
7     }
8
9     if (t->tick == 0) {
10        // ... afficher l'heure ...
11    }
12
13    if ((t->ss % 5) == 0) {
14        // ... afficher texte particulier ...
15    }
16
17    timer_clearInterrupt();
18 }
```

# Wait-For-Interrupt (WFI)

- CPU placé dans un état d'**économie d'énergie**

*Ne sera réveillé que lors de l'occurrence d'une interruption*

- Existence dans la plupart des **ARM Cortex-5**

*Écriture vers le co-processeur CP15*

# Interruption imbriquée

- Nécessité d'avoir des gestionnaires d'interruption **courts**  
*Temps d'exécution inférieur au timer tick*
- Pas de soucis si possibilité d'avoir **interruption imbriquée**  
*Interruption alors qu'un gestionnaire est en cours d'exécution*

# Affichage de l'heure (3)

- Utilisation du CPU au lieu de juste dormir
  - Utilisation de flags marquant les secondes et cinq secondes
  - Réduction drastique du temps d'exécution du gestionnaire

```
1 volatile int one_second = 0, five_seconds = 0;
2
3 void timer_handler()
4 {
5     // ...
6
7     if (t->tick == 0)
8         one_second = 1;
9     if ((t->ss % 5) == 0)
10        five_seconds = 1;
11
12     timer_clearInterrupt();
13 }
```

# Affichage de l'heure (4)

- Utilisation du CPU au lieu de juste dormir
  - Boucle principale vérifie s'il y a du boulot à chaque réveil
  - Ancien gestionnaire devient une « *simple* » fonction

```
1 void wall_clock(TIMER *t)
2 {
3     // ...
4 }
5
6 int main()
7 {
8     // ... initialisation ...
9     while (1) {
10         if (one_second) {
11             wall_clock(&timer);
12             one_second = 0;
13         }
14         if (five_seconds) {
15             // ... afficher texte particulier ...
16             five_seconds = 0;
17         }
18         asm("MOV r0, #0; MCR p15,0,R0,c7,c0,4");           // mettre CPU en état WFI
19     }
20 }
```

# Contrôle port UART et clavier (1)

- Récupération et écho de texte depuis **port UART ou clavier**
  - Utilisation de flags globaux vérifiés de manière répétée
  - Flags mis à jour par gestionnaire d'interruptions

```
1  volatile int uline = 0, kline = 0;
2
3  void uart_handler(UART *up)
4  {
5      u8 mis = *(up->base_MIS);
6      if (mis & 0x10) do_rx(up);
7      else             do_tx(up);
8  }
9
10 int do_rx(UART *up)
11 {
12     if (c == '\r')
13         uline = 1;
14 }
15
16 // ...
17
18 void kbd_handler()
19 {
20     if (c == '\r')
21         kline = 1;
22 }
```

# Contrôle port UART et clavier (2)

- Récupération et écho de texte depuis **port UART ou clavier**
  - Gestionnaire d'interruption echo et place dans buffer
  - Lorsque ENTER, `main()` est notifiée par flag

```
1 int main()
2 {
3     // ... initialisation ...
4     while (1) {
5         if (uline) {
6             ugets(line);
7             uprintf("UART: line=%s\n", line);
8             lock(); uline = 0; unlock();
9         }
10        if (kline) {
11            gets(line);
12            color = GREEN;
13            printf("KBD: line=%s\n", line);
14            lock(); kline = 0; unlock();
15        }
16        asm("MOV r0, #0; MCR p15,0,R0,c7,c0,4");           // mettre CPU en état WFI
17    }
18 }
```

# Condition de course

- Mise à jour concurrente de plusieurs flags partagés
  - Section critique à protéger pour éviter condition de course
  - Désactivation des interruptions à l'aide d'un verrou
- Important à cause de la nature asynchrone des évènements

*Concurrence entre `main()` et gestionnaire d'interruption*

# Priorité

- Attribuer une **plus haute priorité** à un évènement urgent
  - Priorité liée à l'importance et à l'urgence d'un évènement
  - Évènements doivent être gérés en accord avec leur priorité
- **Deux façons** principales de gérer les priorités
  - Contrôleur d'interruptions donne priorités utilisées par `main()`
  - Gestionnaire évènements implémenté comme processus/tâche



Modèle processus

# Processus

- Système embarqué avec plusieurs **processus concurrents**
  - Entité d'exécution qui peut être ordonnancée
  - Choisi, suspendu (avec libération CPU) et remis pour exécution
- **Unité d'exécution** indépendante réalisant une tâche spécifique
- Plusieurs **sous-modèles** selon environnement d'exécution
  - Nombre de processeurs
  - Espace d'adresses
  - Création statique ou dynamique de processus
  - Préemptions des processus

# Processeur

- Système **uni-processeur** (UP) ne possède qu'un seul CPU
  - Plusieurs processus exécutés de manière concurrente sur CPU
  - Utilisation de la multiprogrammation pour le multitasking
- Système **multiprocesseurs** (MP) possède plusieurs processeurs
  - Plusieurs processeurs séparés (SMP) ou cœurs (on-chip)
  - Exécution parallèle de plusieurs processus
  - Combinaison de parallélisme et de multiprogrammation

# Espace d'adresses

- Modèle avec **espace d'adresses réel**
  - Contraintes temporelles fortes
  - Pas équipé ou d'utilisation de hardware de gestion de mémoire
  - Unique espace d'adresses qui est le même que celui du kernel
  - Pas de protection de la mémoire, mais simplicité et efficacité
- Modèle avec **espace d'adresses virtuel**
  - Mapping d'adresses calculé via le hardware
  - Chaque processus reçoit un espace d'adresses virtuel
  - Mode user (espace dédié) ou en mode kernel (même espace)

# Création de processus

- **Création statique** de processus au démarrage du système
  - Restent de manière permanente dans le système
  - Chaque processus de type périodique ou orienté évènements
  - Ordonnancement statique avec priorité sans préemption
- Processus peuvent être créés de manière **dynamique**
  - Exécution de tâches spécifiques à la demande
  - Libération de toutes les ressources après terminaison

# Préemption

- Modèle basique est **non préemptif** par rapport aux processus
  - Processus s'exécute jusqu'à libération volontaire du CPU
  - Sleep, suspension de soi-même, yield à un autre processus
- Possibilité pour le CPU d'être pris par **préemption**
  - CPU peut être retiré d'un processus et donné à un autre
  - Mécanisme de sauvegarde et restauration de l'état d'exécution

# Type de modèles (1)

- Possibilité de faire un mix de **différents modèles** de processus
  - Précédents modèles ne sont pas mutuellement exclusifs
  - Existence de quatre principaux types de systèmes embarqués
- Modèle **kernel** en uni-processeur (UP)
  - Un seul CPU et pas de hardware de gestion de la mémoire
  - Processus dans le même espace d'adresses que le kernel
  - Pas de préemption, création statique/dynamique de processus

# Type de modèles (2)

- Modèle **système d'exploitation** uni-processeur (UP)
  - Extension du modèle kernel UP avec gestion mémoire hardware
  - Exécution d'un processus en **mode kernel**
    - Partage du même espace d'adresses que le kernel
    - Utilisation du CPU sans préemption sauf libération volontaire
  - Exécution normale d'un processus en **mode utilisateur**
    - Espace d'adresses dédié protégé et privé au processus
    - Préemption d'un processus pour libérer le CPU pour un autre

# Type de modèles (3)

- Modèle de systèmes **multiprocesseurs** (MP)
  - Plusieurs CPU ou cœurs partageant la même mémoire physique
  - Exécution parallèle possible sur plusieurs processeurs
  - Techniques avancées de synchronisation et de protection
- Modèle de systèmes **temps réel** (RT)
  - Généralement contraintes de temps pour systèmes embarqués
  - Réponse rapide à interruptions et temps de gestion court
  - Garanties et respect obligatoire des contraintes temps réel

# Méthode de design



# Méthode de design

- Méthode ad-hoc de développement n'est plus adaptée

*Suite à augmentation de la complexité des systèmes embarqués*

- Plusieurs méthodes de design formelles existent

- Support direct dans langages de haut niveau (Java, C++...)
- Machine à états finis (FSM)
- Statecharts

# Langage de haut niveau

- Utilisation d'un **langage de haut niveau** comme Java et C++
  - Exploitation de constructions comme évènements et exceptions
  - Modèle pour développer des programmes orienté évènements
- Utilisation des **outils** de haut niveau du développeur
  - Environnement de développement intégré (IDE) pour écrire
  - Outils de debugging et d'analyse de performances
  - Prototypage rapide et certifications sur les algorithmes

# Machine à états finis (1)

- Système embarqué comme une **Finite State Machine** (FSM)

$$\mathcal{F} = \langle S, X, O, f \rangle$$

où

- $S$  est un ensemble fini d'états
  - $X$  est un ensemble fini d'entrées
  - $O$  est un ensemble fini de sorties
  - $f$  est une fonction de transition d'états
- 
- Fonction de **transition d'états** contient l'aspect dynamique

$$\begin{aligned}f : S \times I &\rightarrow S \times O \\(s, x) &\mapsto (s', o)\end{aligned}$$

# Machine à états finis (2)

- Plusieurs **caractéristiques** aux FSMs
  - Complètement spécifiée si  $f(s, x)$  définie pour toute paire  $(x, s)$
  - Déterministe si  $f(s, x)$  produit toujours le même résultat
- **Deux types** de FSMs principaux
  - Machine de Mealy si output dépend d'input
  - Machine de Moore si output ne dépend que de l'état

# FSM pour système embarqué

- Machine de Mealy complètement spécifiée et déterministe
  - Permet l'utilisation de vérification formelle
  - Traduction automatique de FSM vers du code
- Suppression des commentaires d'un code source C

*Commence par // et se termine en bout de ligne*

# Étape 1 : Construction tableau d'états (1)

- Identification de **cinq états** pour la FSM dont initial et final
  - $S_1$  on n'a pas encore rencontré un seul /
  - $S_2$  on a vu le symbole / une première fois
  - $S_3$  on a vu deux symboles / adjacents
- **Quatre classes de caractères** qu'il est possible de rencontrer
  - $x_1$  est un slash /
  - $x_2$  est un retour à la ligne \n
  - $x_3$  est tout sauf /, \n ou EOF
  - $x_4$  est EOF (*end-of-file*)

# Étape 1 : Construction tableau d'états (2)

- Dans chaque état, chaque input provoque une **transition**

*Production d'un output en se déplaçant vers l'état destination*

État	$x_1$	$x_2$	$x_3$	$x_4$
$S_0$	$S_2/-$	$S_1/"x_2"$	$S_1/"x_3"$	$S_4/-$
$S_1$	$S_2/-$	$S_1/"x_2"$	$S_1/"x_3"$	$S_4/-$
$S_2$	$S_3/-$	$S_1://"x_2"$	$S_1://"x_3"$	$S_4/-$
$S_3$	$S_3/-$	$S_1/"x_2"$	$S_3/-$	$S_4/-$
$S_4$	$S_4/-$	$S_4/-$	$S_4/-$	$S_4/-$

## Étape 2 : Minimisation tableau d'états (1)

- Table initiale peut contenir **plus d'états que nécessaires**

*La table peut ne pas être minimale, à cause de redondances*
- Identification d'**états équivalents** à fusionner

*Pour les mêmes inputs, produit le même résultat*
- Une **relation d'équivalence**  $R$  est une relation binaire
  - **Réflexive**  $\forall x \in R : xRx$
  - **Symétrique**  $\forall x, y \in R : xRy \implies yRx$
  - **Transitive**  $\forall x, y, z \in R : xRy \wedge yRz \implies xRz$

## Étape 2 : Minimisation tableau d'états (2)

- Une **classe d'équivalence** est un ensemble d'éléments
  - Tous les éléments d'une classe sont équivalents
  - Permet de créer une partition d'un ensemble d'éléments
  - Chaque classe d'équivalence peut se réduire à un représentant
- Deux **états équivalents** même comportement pour tout input  $x$ 
  - Les outputs sont exactement les mêmes
  - Les états atteints par leurs transitions sont équivalents

## Étape 2 : Minimisation tableau d'états (3)

- Construction d'un **tableau d'implications**

*Preuve par contradiction sachant que  $A \implies B \equiv \neg B \implies \neg A$*

- Algorithme d'identification de paires d'**états non équivalents**

- 1 Avec les outputs, barrer toutes les paires  $(S_i, S_j)$  impossibles
- 2 Examiner les  $(S'_i, S'_j)$  atteignable et barrer si pas équivalents
- 3 Répéter tant qu'il reste des choses à barrer

# Étape 3 : Traduction tableau d'états en code

- Traduction **quasi immédiate** vers un programme en C
  - Utiliser une variable state pour indiquer l'état
  - Utilisation de l'instruction switch pour brancher

```
1 int main()
2 {
3     int c;
4     int state = 1;
5     FILE *fp = fopen("cprogram.c", "r");
6
7     while ((c = fgetc(fp)) != EOF) {
8         switch (state) {
9             case 1:
10                 switch(c) {
11                     case '/': state = 2; break;
12                     case '\n': state = 1; printf("%c", c); break;
13                     default : state = 1; printf("%c", c); break;
14                 }
15                 break;
16             // ... autre cas ...
17         }
18     }
19 }
```

# StateCharts

- Modèle étendu des FSM avec de nouvelles constructions
  - Ajout de la concurrence et de la communication
  - Possibilité de modélisation de plus haut niveau
- Possibilité de transformer un statechart en FSM
  - Problème de l'explosion de la taille de l'espace d'états
  - Compromis facilité de modélisation et complexité de calcul

# Crédits

- <https://www.flickr.com/photos/uwehermann/2995807588>
- <https://www.flickr.com/photos/drownedman/4205166796>
- <https://www.flickr.com/photos/thomashawk/9779035382>
- <https://www.flickr.com/photos/junaidrao/34773734383>