



Etude de cas E5, Systèmes Embarqués

Par OMARI Nisrine & BENOIT Thomas

Décembre 2018

Partie automatique :

Simulation sous MathLab

Encadrant: A.CELA

1er étape : Contexte de travail & découverte du projet	2
2ème étape : modélisation du pendule sous Matlab	5
3ème étape : Insertion du BUS CAN	9
4ème étape : Fusion des blocs Sensor & Actuator	14
5ème étape : Calcul déporté de la commande avec deux pendules	16
6ème étape : Conception des deux interfaces	20
7ème étape : Tests et définition des limites du système	26
BIBLIOGRAPHIE	30
ANNEXE N°1 : Commandes de True Time et descriptions	31

Ici l'objectif est d'intégrer le réseaux CAN et de déporter les calculs sur un calculateur (nœud) distant s'inspirant de l'exemple Distributed de TrueTime.

1er étape : Contexte de travail & découverte du projet

Nous avons débuté ce projet par de la recherche de documentation sur True Time et avons réussi à comprendre son fonctionnement grâce à plusieurs manuels (voir bibliographie) puis pour mieux comprendre le cheminement du code qui nous a été donné en exemple nous avons listé les différentes fonctions que True Time nous permet d'utiliser (voir annexe 1).

True time a pour fonction de Co-simuler l'exécution des tâches du contrôleur et de la transmission sur le réseau. De plus, il fournit des modèles de noyaux en temps réel et des réseaux sous forme de blocs Simulink. Le code est sous la forme de tâches et de gestionnaires d'interruptions.

True Time offre un block Kernel, deux blocks réseau, un bloc d'interface réseau et un bloc batterie, les fonctions Simulink sont écrites en C++.

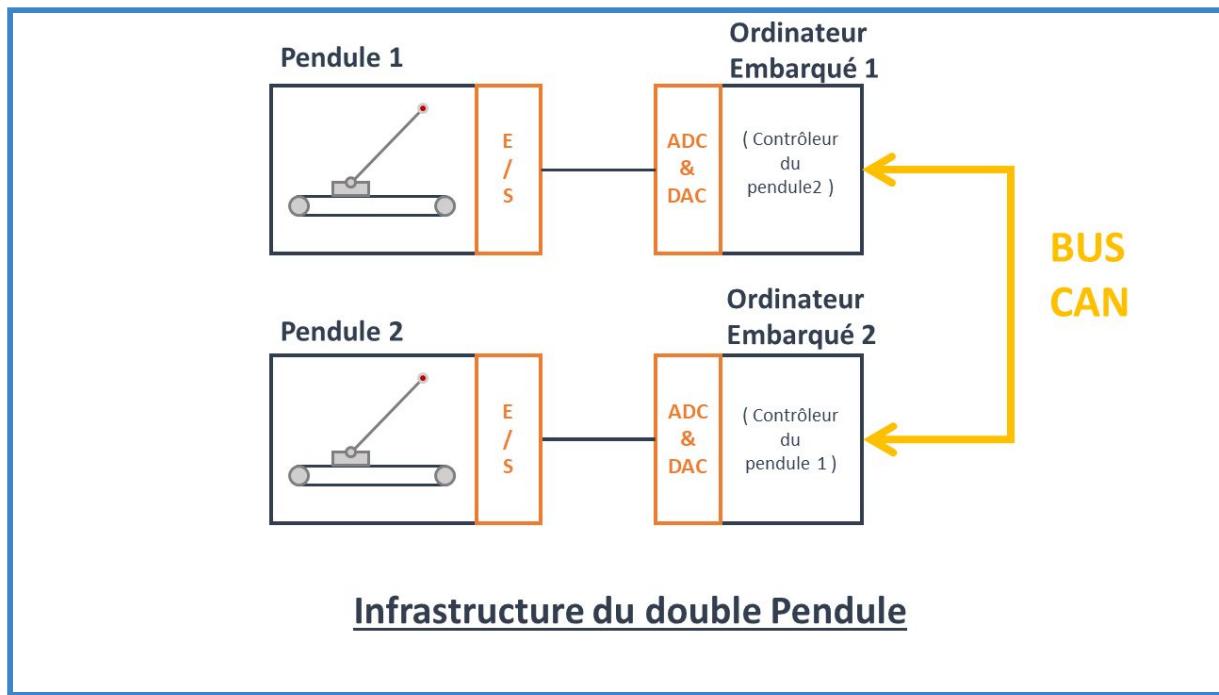
TrueTime implémente un noyau complet en temps réel avec plusieurs files d'attente;

- Une pour les tâches prêtes à l'exécution
- Une pour les tâches en attente de sortie
- Une pour les moniteurs et les événements

Les files d'attente sont manipulées par le noyau ou par des appels au noyau. Ce noyau simulé est idéal (pas de latence d'interruption et pas de temps d'exécution associé aux primitives en temps réel).

La simulation est basée sur les événements obtenues à l'aide de la fonction de passage à zéro de Simulink ce qui garantit que le noyau s'exécute chaque fois qu'un événement se produit.

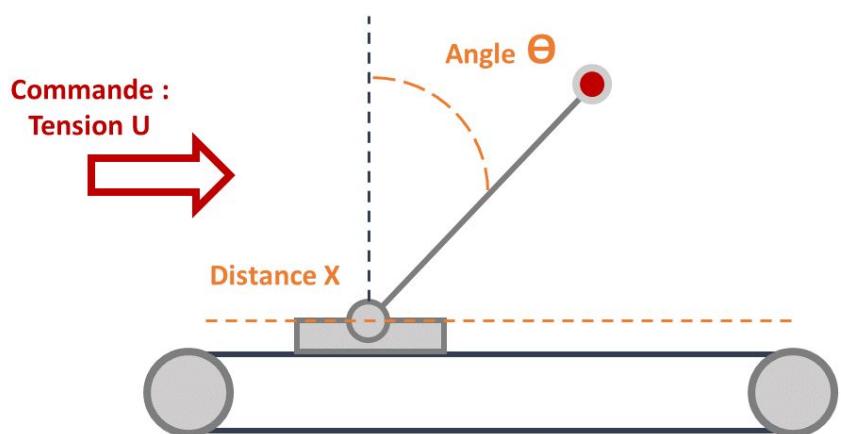
Cette étude de cas porte sur l'étude d'un double pendule inversé. Notre objectifs est de stabiliser les deux systèmes en temps réel. Tout au long de ce projet, nous utilisons l'architecture suivante :



Nous notons que les deux pendules sont chacun reliés à un ordinateur embarqué. Les missions de ces derniers sont :

- l'acquisition des données de leurs pendules associés
- l'échange des informations via bus CAN
- la réalisation d'un calcul déporté de la commande

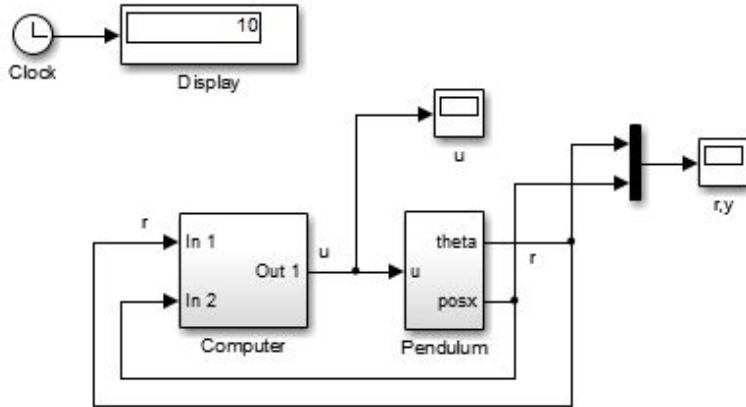
Pour contrôler nous calculons la commande **U**, une tension en Volt, grâce à l'angle Θ de la tige du pendule, et à la position **X** du pendule sur le rail. Le schéma suivant illustre l'utilisation de ces grandeurs :



Grandeurs des pendules du système

Dans la partie Automatique de l'étude de cas, nous allons simuler ces infrastructures son Matlabs et Simulink. Nous pourrons alors définir un asservissement optimal pour notre système.

2ème étape : modélisation du pendule sous Matlab



TrueTime 1.5 PID-control of a DC servo
Copyright (c) 2007
Martin Ohlin, Dan Henriksson and Anton Cervin
Department of Automatic Control, Lund University, Sweden
Please direct questions and bug reports to: trutime@control.lth.se

Schéma du fonctionnement de True Time + pendule

Après avoir étudié le fonctionnement de True Time avec un cerveau moteur, nous avons introduit notre pendule, ce qui a engendré la modification de notre code de base.

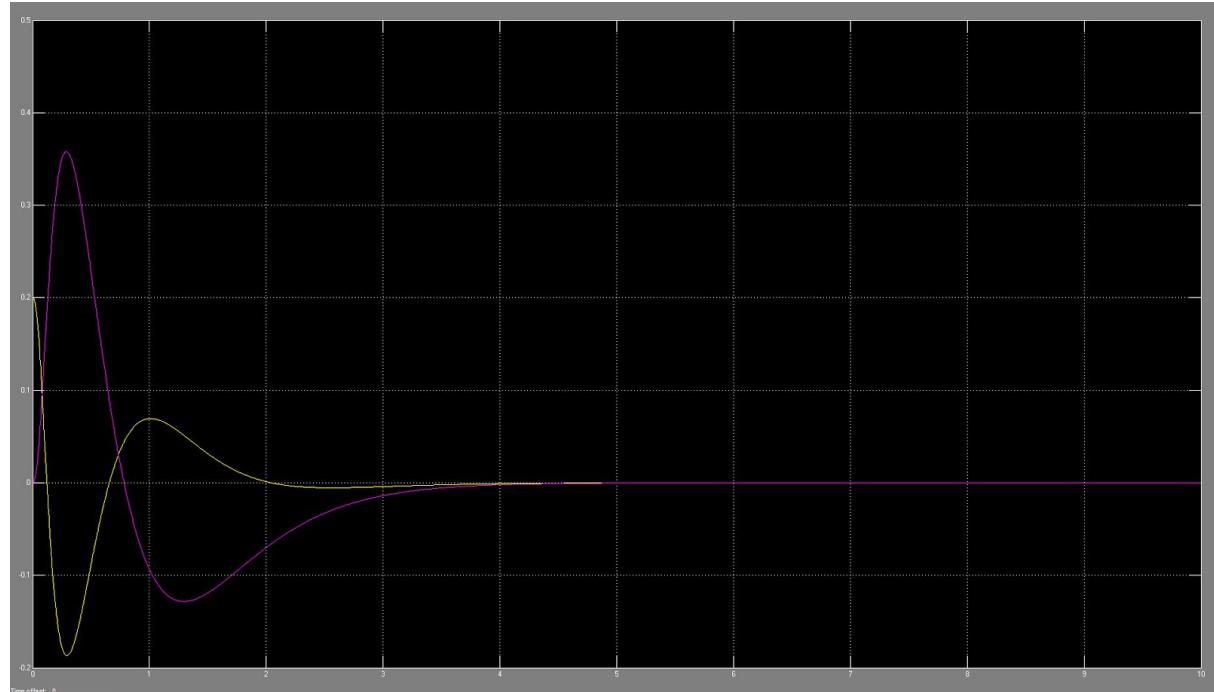
Le true Time est situé à l'intérieur du bloc “Computer”. Nous souhaitons récupérer en sortie ‘posx’ la position du pendule et ‘theta’ l’angle d’inclinaison du pendule.

Nous avons introduit à la partie synthèse la notion de “delay” pour établir la plage d’équilibre du pendule (position verticale haute et basse). Dans le linmode nous précisons le modèle et le point de linéarisation.

Puis pour utiliser True Time nous définissons le chemin “path” dans lequel nous pointons le kernel.

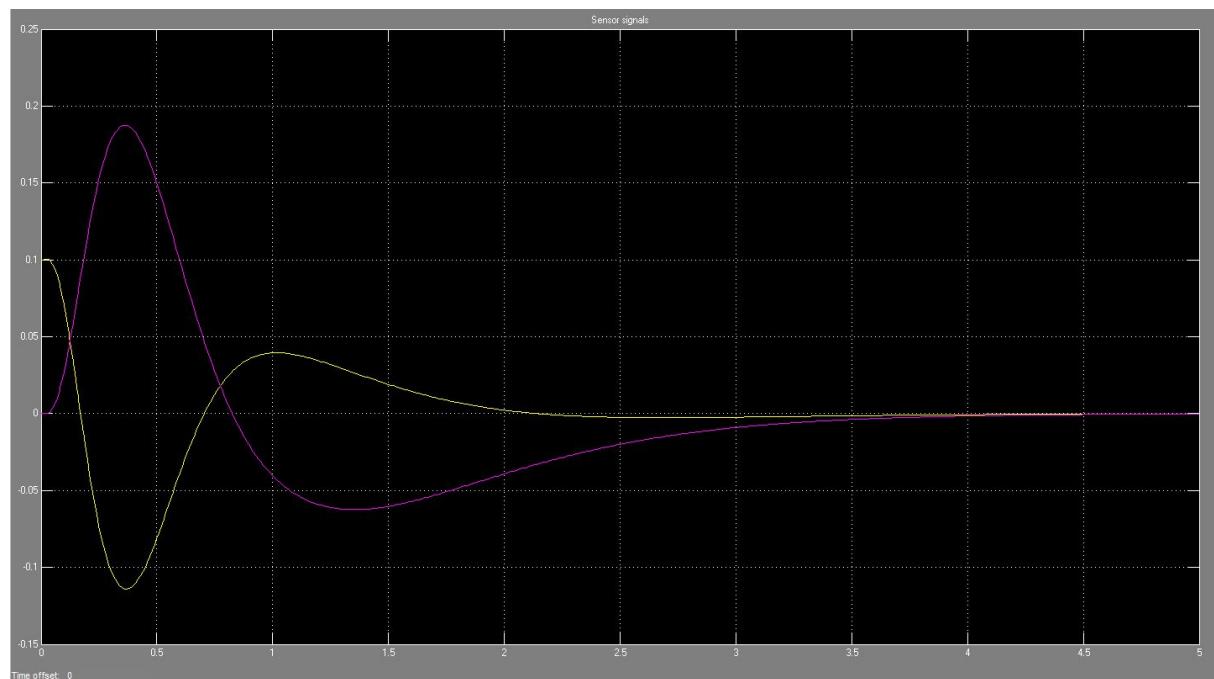
Par la suite nous avons manipulé trois servomoteurs, ce qui nous a permis de comprendre l’ordonnancement des tâches. En effet, le premier servomoteur ne fonctionnait pas car la tâche n’étant prioritaire et la ressource étant partagée ce dernier ne pouvait y accéder. Nous

avons alors changé l'ordonnancement en “PrimoEDF”. Cet ordonnancement est alors plus équitable car la politique d'allocation de la ressource est en fonction de la deadline.

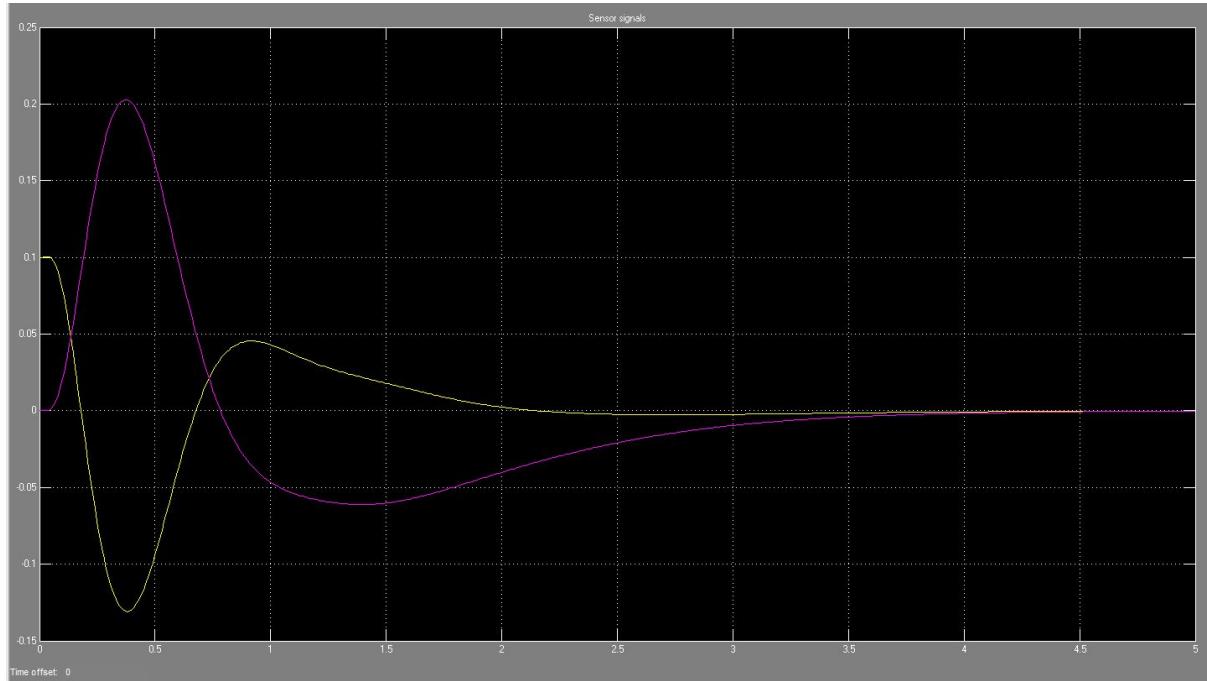


Visualisation du signal de sortie

Après simulation nous observons bien une stabilisation des courbes autour de 0, qui correspond à la position d'équilibre verticale.

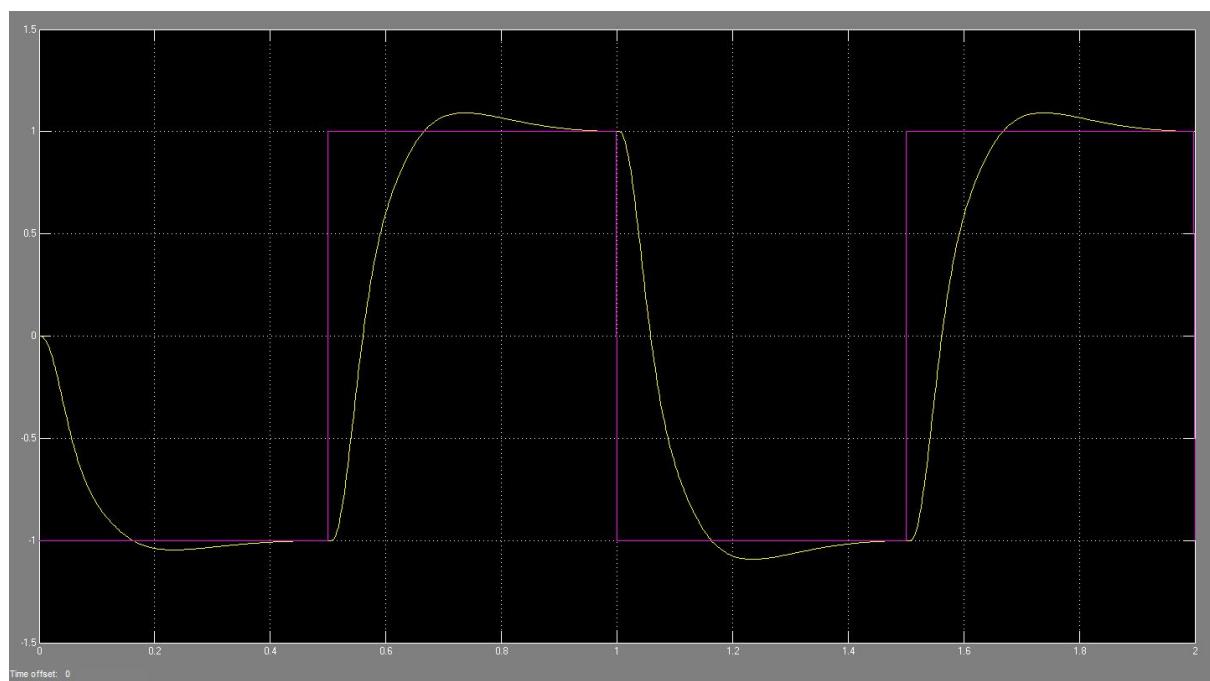


Visualisation du signal de sortie pour $t_delay=0.01$



Visualisation du signal de sortie pour $t_delay=0.02$

Nous déterminons la bande passante en testant différentes valeurs de t_delay .



Visualisation du signal de sortie à la première exécution de True Time

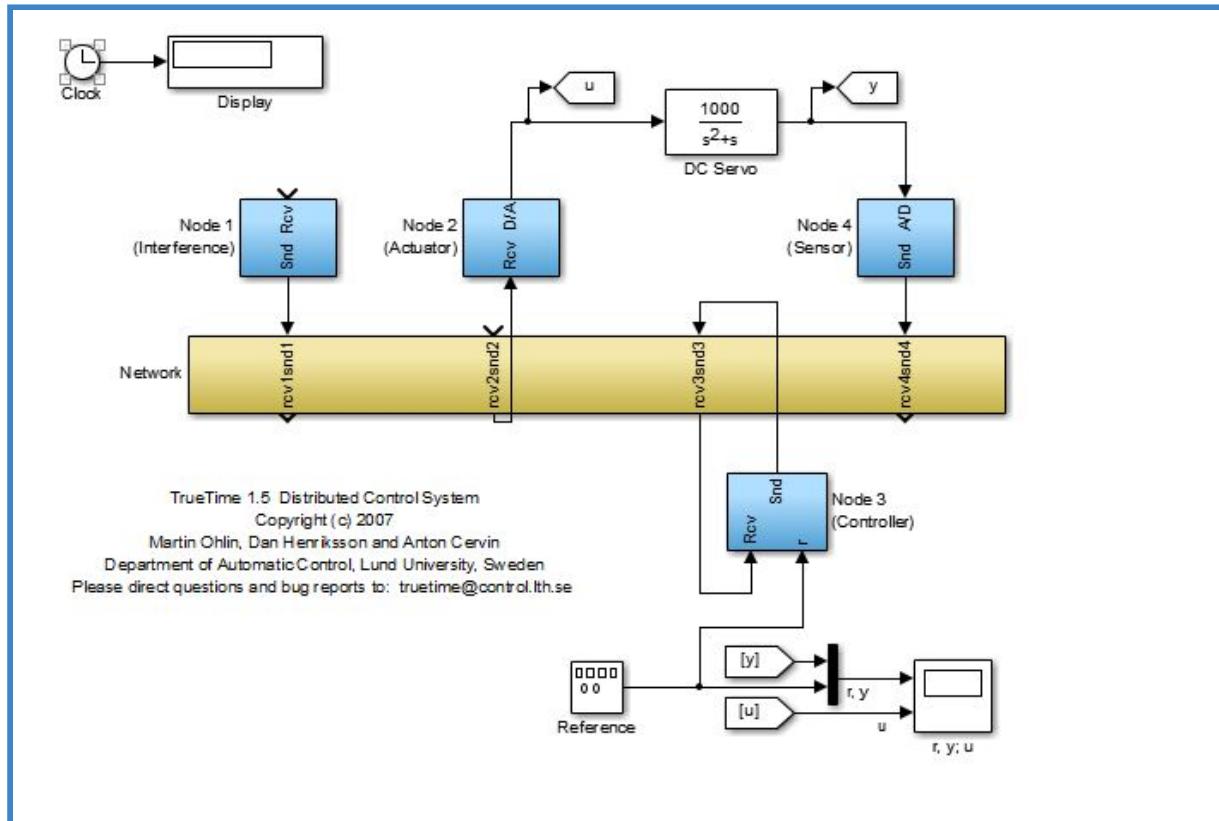
Grâce à la modélisation de cette partie, nous avons pu récupérer les matrices des valeurs initiales de notre pendule.

Nous obtenons alors les coefficients, que nous utiliserons pour la suite du projet :

```
Adc=[0.6300 -0.1206 -0.0008 0.0086  
      -0.0953 0.6935 0.0107 0.0012  
     -0.2896 -1.9184 1.1306 0.2351  
    -3.9680 -1.7733 -0.1546 0.7222];  
Bdc=[0.3658 0.1200  
      0.0993 0.3070  
     1.0887 2.0141  
    3.1377 1.6599];  
Cdc=[-80.3092 -9.6237 -14.1215 -23.6260];  
Ddc=[0 0];
```

3ème étape : Insertion du BUS CAN

Avant d'insérer le bus CAN dans notre architecture du pendule sous Simulink, nous avons étudié un exemple proposé par les archives de TrueTime. En effet, l'exemple “*Distributed*” asservie un cerveau moteur grâce à trois nœuds (*Actuator*, *Sensor*, et *Controller*) reliés par un BUS CAN. Cette architecture est la suivante :



Infrastructure de l'exemple *Distributed*, proposé par TrueTime

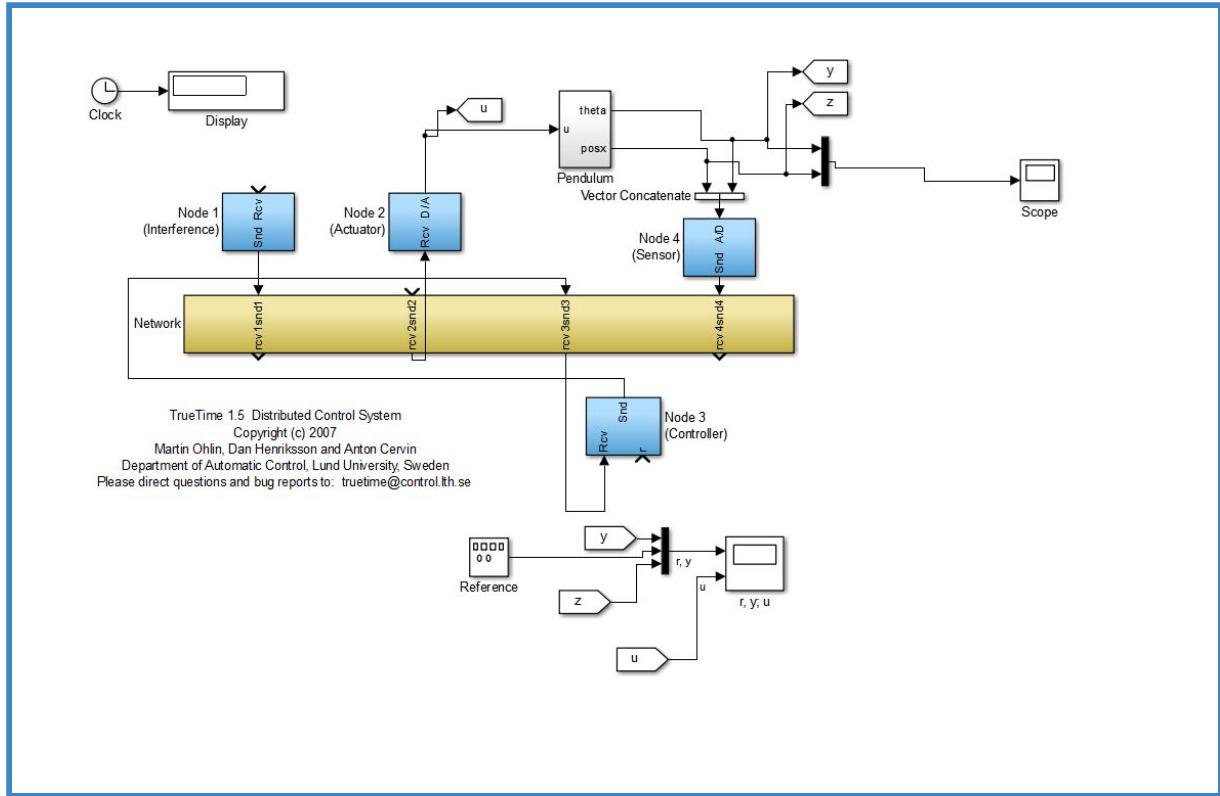
Dans cet exemple, les trois noeuds, associés à un servomoteur, ont les fonctions suivantes :

- Le nœud (node 4) *Sensor* permet l'acquisition des données. Les informations récupérées sont envoyées au noeud *Controller*.
- Le nœud(node 3) *Controller* permet le calcul de la commande . Le résultat du traitement est envoyé au noeud *Actuator*.
- Le nœud (node 2) *Actuator* permet l'application de la commande au servomoteur.

Nous avons alors un exemple de calcul déporté et d'une communication via bus CAN.

Par conséquent, nous pouvons utiliser cet exemple en adaptant l'infrastructure pour notre pendule, modélisé dans la partie 2.

Après modification de l'infrastructure Simulink de l'exemple précédent, nous obtenons le schéma suivant :



Architecture d'un pendule simple avec calculs déportés et bus CAN

Les premières modifications marquantes sont la création d'un vecteur de donné à l'entrée du noeud *Sensor*, et la suppression de l'entrée de référence dans le noeud *Controller*. En effet, le noeud *Sensor* doit réaliser l'acquisition de l'angle Θ et de la position X, nous pouvons alors regrouper ces données sous forme d'un vecteur exploitable par le noeud *Controller*. De plus, l'entrée de référence du noeud *Controller* était utile pour le calcul de la commande du servomoteur. Mais dans notre infrastructure, cette entrée ne peut pas être exploitée pour le cas du pendule.

D'un point de vu programmation, notre noeud *Sensor* pointe sur une fonction *sensor_init-t.m*. Au sein de ce fichier, nous créons une tâche périodique *sens_task*, dont la routine a pour objectifs de récupérer l'angle Θ et de la position X, à partir des deux entrées analogiques du noeud. De suite, dans un second segment de la tâche, notre routine envoie les données au noeud 3, le *Controller*. Voici les extraits des scripts correspondants :

```

1 function sensor_init-t
2
3 % Distributed control system: sensor node
4 %
5 % Samples the plant periodically and sends the samples to the
6 % controller node.
7
8 % Initialize TrueTime kernel
9 ttInitKernel(2, 0, 'prioFP'); % nbrOfInputs, nbrOfOutputs, fixed priority
10
11 % Create sensor task
12 data.y = [0;0];
13 offset = 0;
14 period = 0.010;
15 prio = 1;
16 ttCreatePeriodicTask('sens_task', offset, period, prio, 'senscode-t', data);
17
18 % Initialize network
19 ttCreateInterruptHandler('nw_handler', prio, 'msgRcvSensor');
20 ttInitNetwork(4, 'nw_handler'); % node #4 in the network
21

```

Fonction sensor-t.m . initialisation du noeud Sensor

```

1 function [execetime, data] = senscode-t(seg, data)
2
3 switch seg,
4 case 1,
5   data.y = ttAnalogIn(1);
6   execetime = 0.00005;
7 case 2,
8   ttSendMsg(3, data.y, 80); % Send message (80 bits) to node 3 (controller)
9   execetime = 0.0004;
10 case 3,
11   execetime = -1; % finished
12 end
13

```

Routine senscode-t

Une fois les données acheminées à notre noeud *Controller*, ce dernier réalise le calcul de la commande. Le noeud *Controller* pointe sur une fonction *controllerbis_init.m*, qui initialise le calcul de la même manière que le *bloque computer* du schéma de la partie 2. Une tâche périodique *Obs_Cont_task*, de routine *ctrl_obs*, réalise le calcul de la commande. Une fois la commande **U** calculée, l'information est envoyée au noeud 2, *Actuator*. Voici les extraits des scripts correspondants :

```

1  function controllerbis_init
2  % Control of Inverted Pendulum via TrueTime Real Time Kernel Model
3  %
4  %
5  % * Observer Controller task
6  % * name: Obs_Cont_task
7  % exec code: Obs_Cont
8  % type: Periodic task
9  % function: Calculates the Pendulum states for each sampling period based on its linear model and measurement of pendulum angular position and cart horizontal position
10 %           each T_ech seconds
11 %
12 %
13 T_ech = 0.010; % Sampling period
14 %
15 % Observer-Controller parameter
16 %
17 %
18 Adc=[0.6300 -0.1206 -0.0008 0.0086
19      -0.0953 0.6935 0.0107 0.0012
20      -0.2896 -1.9184 1.1306 0.2351
21      -3.9680 -1.7733 -0.1546 0.7222];
22 Bdc=[0.3658 0.1200
23      0.0993 0.3070
24      1.0887 2.0141
25      3.1377 1.6599];
26 Cdc=[-80.3092 -9.6237 -14.1215 -23.6260];
27 Ddc=[0 0];
28 %
29 %
30 ttInitKernel(2, 1, 'prioFP'); % nbrOfInputs, nbrOfOutputs, fixed priority
31 %
32 % Task attributes
33 % Controller parameters
34 h = 0.010;
35 N = 100000;
36 Td = 0.035;
37 K = 1.5;
38 %
39 %period = 0.01;
40 %deadline = period;
41 %
42 %offset = 0.0;
43 %prio = 1;
44 %
45 % Create task data (local memory)
46 %
47 data.Adcc=Adc;
48 data.Bdc=Bdc;
49 data.Cdc=-Cdc;
50 data.Ddc=Ddc;
51 %
52 data.x=[0 0 0 0]'; % initial state condition of pendulum
53 data.y=[0 0]'; % initial positions/outputs of pendulum
54 data.u=0; % initial control of the pendulum
55 %
56 % Data relative to pendulum interfaced with Computer
57 %
58 data.th = 0;
59 data.d=0;
60 data.ys=[0;0];
61 data.rhChannel=1;
62 data.chanel=2;
63 data.uChanel=1;
64 data.test=[];
65 offset = 0;
66 period = T_ech;
67 deadline=period;
68 prio = 3;
69 %
70 %
71 % Create controller task
72 %deadline = h;
73 %prio = 2;
74 % IMPLEMENTATION : using the built-in support for periodic tasks
75 %
76 ttCreatePeriodicTask('Obs_Cont_task', offset, period, prio, 'ctrl_obs', data);
77 %
78 % Optional disturbance task
79 if arg > 0
80 offset = 0.0002;
81 period = 0.007;
82 prio = 1;
83 %
84 ttCreatePeriodicTask('dummy', offset, period, prio, 'dummicode');
85 end
86 %
87 % Initialize network
88 ttCreateInterruptHandler('nw_handler', prio, 'msgRcvCtrl');
89 ttInitNetwork(3, 'nw_handler'); % node #3 in the network
90 %
91 %
92 %
93

```

Fonction controllerbis_init.m. fonction d'initialisation du noeud Controller

```

1 function [execetime, data] = ctrl_obs(seg, data)
2
3 switch seg,
4
5 case 1,
6 y = ttGetMsg; % Obtain sensor value
7 %r = ttAnalogIn(1); % Read reference value
8 %enlever theta et d , comme ils sont dans y (signal reçu)
9 theta = ttAnalogIn(data.thChanel); % Read pendulum angle signal
10 d = ttAnalogIn(data.dChanel); % Read cart horizontal distance
11 y=[theta;d];
12
13 data = ctrlcode_bis(data,y);
14 execetime = 0.002; % Predicted calculation time of Observer-Controller code
15 case 2,
16 %ttAnalogOut(data.uChanel, data.u);
17 ttSendMsg(2, data.u, 80); % Send 80 bits to node 2 (actuator)% Output control signal application to the input of Inverted Pendulum
18 execetime = -1;
19 end
20

```

Routine ctrl_obs.m

```

1 function data = ctrlcode_bis(data, y)
2
3 data.x=data.Adc*y;
4 data.u=data.Cdc*x; % State space equation of Observer
5 % Controller Equations
6 %
7 %
8 %
9

```

Fonction de traitement ctrlcode_bis.m

Enfin, la commande arrive au nœud 2, *Actuator*, qui applique la commande fraîchement calculé. Ce nœud pointe sur une fonction *actuator_init.m*, qui initialise une tâche périodique *act_task*. La routine de cette tâche est définie dans la fonction *actcode.m*. Sa mission est de simplement assurer la retransmission de la commande **U** sur la seule sortie analogique du nœud *Actuator*. Ainsi, notre pendule réceptionne la commande. Voici les extraits des scripts correspondants :

```

1 function actuator_init
2
3 % Distributed control system: actuator node
4 %
5 % Receives messages from the controller and actuates
6 % the plant.
7
8 % Initialize TrueTime kernel
9 ttInitKernel(0, 1, 'prioFP'); % nbrOfInputs, nbrOfOutputs, fixed priority
10
11 % Create actuator task
12 deadline = 100;
13 prio = 1;
14 ttCreateTask('act_task', deadline, prio, 'actcode');
15
16 % Initialize network
17 ttCreateInterruptHandler('nw_handler', prio, 'msgRcvActuator');
18 ttInitNetwork(2, 'nw_handler'); % node #2 in the network
19
20

```

Fonction actuator_init.m, fonction d'initialisation du noeud actuator

```

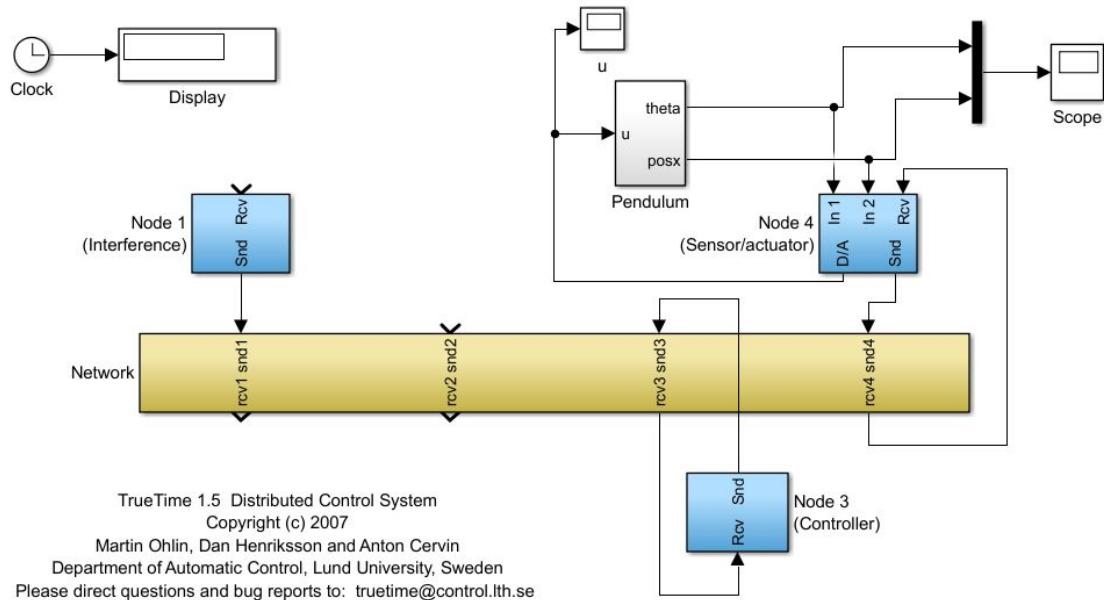
1 function [execetime, data] = actcode(seg, data)
2
3 switch seg,
4
5 case 1,
6 data.u = ttGetMsg;
7 execetime = 0.0005;
8 case 2,
9 ttAnalogOut(1, data.u)
10 execetime = -1; % finished
11 end
12

```

Routine actcode.m

4ème étape : Fusion des blocs Sensor & Actuator

Après avoir réussi l'étape précédente, nous pouvons fusionner les blocs *Sensor* et *Actuator*. Cette étape permettra de réaliser un modèle de calcul déporté de la commande, pour un pendule. Voici le schéma Simulink de cette partie :



Modèle Simulink avec fusion des blocs Actuator et Sensor, pour un pendule

Notons que nous avons :

- Node 1 : le bloc d'interférence
- Node 3 : le bloc *Controller*
- Node 4 : le bloc *Sensor/Actuator*

Pour arriver à ce résultat, nous avons créé la fonction d'initialisation *sensor_actuator_init.m*, qui configure deux tâches : *senscode_task*, tâche périodique pour l'acquisition des données, et *act_task*, interruption pour l'application de la commande. Voici le script de *sensor_actuator_init.m* :

```

1  function sensor_actuator_init(arg)
2  % Control of Inverted Pendulum via TrueTime Real Time Kernel Model
3  %
4
5  % * Observer Controller task
6  % * name: Obs_Cont_task
7  % exec code: Obs_Cont
8  % type: Periodic task
9  % function: Calculates the Pendulum states for each sampling period based on its line
10 %           each T_ech seconds
11 %
12
13 - T_ech = 0.010; % Sampling period
14
15 % Observer-Controller parameter
16 - ttInitKernel(2, 1, 'prioFP'); % nbrOfInputs, nbrOfOutputs, fixed priority
17
18 % Data relative to pendulum interfaced with Computer
19
20 - data.th = 0;
21 - data.d=0;
22 - data.yy=[0;0];
23 - data.thChanel=1;
24 - data.dChanel=2;
25 - data.test=[];
26 - data.donnee=[];
27 - offset = 0;
28 - period = T_ech;
29 - prio = 3;
30 % IMPLEMENTATION : using the built-in support for periodic tasks
31
32 - ttCreatePeriodicTask('senscode_task', offset, period, prio, 'senscode', data);
33 - disp('SENSOR INIT')
34
35
36 % Distributed control system: actuator node
37 %
38 % Receives messages from the controller and actuates
39 % the plant.
40 - data.uChanel=1;
41 % Initialize TrueTime kernel
42
43 % Create actuator task
44 - deadline = 100;
45 - prio = 1;
46 - ttCreateTask('act_task', deadline, prio, 'actcode', data);
47
48 % Initialize network
49 - ttCreateInterruptHandler('nw_handler', prio, 'msgRcvActuator');
50 - ttInitNetwork(4, 'nw_handler'); % node #4 in the network
51 - disp('ACTUATOR INIT')
52
53

```

Script de sensor_actuator_init.m

Les routines des tâches *senscode_task* et *act_task* ont respectivement pour routine *senscode.m* et *actcode.m*. Les scripts de ces routines ont été introduits précédemment dans la troisième étape, *Insertion du BUS CAN*, (*senscode.m* est équivalent à *senscode-t.m*).

Une fois cette étape réalisée, nous pouvons dupliquer cette infrastructure pour deux pendules. Par conséquent, nous pouvons réaliser un calcul déporté de la commande, pour deux pendules sur le réseau CAN.

5ème étape : Calcul déporté de la commande avec deux pendules

Dans cette partie, nous allons effectuer le calcul déporté de la commande. Nous définissons nos deux ordinateurs embarqués, ARCOM_1 et ARCOM_2, de la manière suivante :

ARCOM_1 :

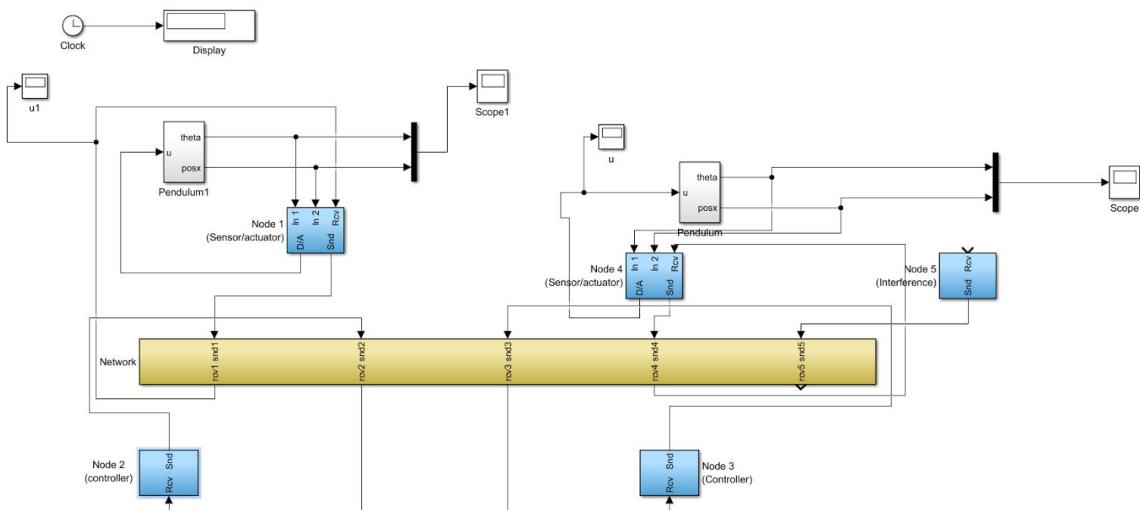
- Un bloc *Sensor et Actuator* pour le pendule 1
- Un bloc *Controller*, afin de calculer la commande du pendule 2

ARCOM_2 :

- Un bloc *Sensor et Actuator* pour le pendule 2
- Un bloc *Controller*, afin de calculer la commande du pendule 1

En d'autre terme, le Controller de l'Arcom_1 calculera la commande à appliquer pour l'Arcom_2, et inversement.

Le modèle Simulink de cette partie est le suivant :



Modèle Simulink avec calcul déporté de la commande, pour deux pendules

Notes :

- Node 1 : Bloc Sensor & Actuator du pendule 2, implémenté sur l'ARCOM 2
- Node 2 : Bloc Controller du pendule 1, implanté sur l'ARCOM 2
- Node 3 : Bloc Controller du pendule 2, implémenté sur l'ARCOM 1
- Node 4 : Bloc Sensor & Actuator du pendule 1, implémenté sur l'ARCOM1
- Node 5 : Bloc d'interférences

Les différents blocs communiquent toujours avec le réseau CAN, et les flux sont les suivants :

Pour le pendule 1 :

- Le *Sensor* de l'ARCOM_1 envoie les données au *Controller* de l'ARCOM_2 (communication du Node 4 vers le Node 2)
- La commande calculée par le *Controller* de l'ARCOM_2 est envoyé à l'*Actuator* de l'ARCOM_1 (communication du Node 2 vers le Node 4)

Pour le pendule 2 :

- Le *Sensor* de l'ARCOM_2 envoie les données au *Controller* de l'ARCOM_1 (communication du Node 1 vers le Node 3)
- La commande calculée par le *Controller* de l'ARCOM_2 est envoyé à l'*Actuator* de l'ARCOM_1 (communication du Node 3 vers le Node 1)

Si nous retranscrivons ce schéma de communications au sein des routines des tâches de chaque ARCOM, nous obtenons les modifications suivantes :

- Pour l'ARCOM_1 :

Nous modifions la routine *senscode.m*, de la tâche *senscode_task* (*lié au pendule 1*), ainsi que la routine *ctrlcode1.m*, de la tâche *Oc_task* (*lié au pendule 2*):

```

1  function [execetime, data] = senscode(seg, data)
2
3  switch seg,
4  case 1,
5    data.th = ttAnalogIn(data.thChanel); % Read pendulum angle signal
6    data.d = ttAnalogIn(data.dChanel);   % Read cart horizontal distance
7    data.donnee = [data.th;data.d];
8    execetime = 0.0003;
9  case 2,
10   ttSendMsg(2, data.donnee, 80); % Send message (80 bits) to node 2 (controller)
11   execetime = 0.0004;
12  case 3,
13   execetime = -1; % finished
14 end
15

```

Routine senscode.m, avec envoi vers l'ARCOM_2

```

1  function [execetime, data] = ctrlcode1(seg, data)
2
3  switch seg,
4  case 1,
5    y = ttGetMsg;           % Obtain sensor value
6    data = obscont(data, y);
7    execetime = 0.0005;
8  case 2,
9    ttSendMsg(1, data.u, 80); % Send 80 bits to node 1 (actuator)
10   execetime = -1; % finished
11 end
12

```

Routine ctrlcode1.m, avec envoi vers l'ARCOM_2

- Pour l'ARCOM_2 :

Nous modifions la routine *senscode2.m*, de la tâche *senscode_task* (lié au pendule 2), ainsi que la routine *ctrlcode2.m*, de la tâche *Oc_task* (lié au pendule 1):

```

1  function [execetime, data] = senscode2(seg, data)
2
3  switch seg,
4  case 1,
5    data.th = ttAnalogIn(data.thChanel); % Read pendulum angle signal
6    data.d = ttAnalogIn(data.dChanel);   % Read cart horizontal distance
7    data.donnee = [data.th;data.d];
8    execetime = 0.0003;
9  case 2,
10   ttSendMsg(3, data.donnee, 80); % Send message (80 bits) to node 3 (controller)
11   execetime = 0.0004;
12  case 3,
13   execetime = -1; % finished
14 end
15

```

Routine senscode2.m, avec envoi vers l'ARCOM_1

```
1 function [execetime, data] = ctrlcode2(seg, data)
2
3 switch seg
4 case 1,
5     y = ttGetMsg;           % Obtain sensor value
6     data = obscont(data, y);
7     execetime = 0.0005;
8 case 2,
9     ttSendMsg(4, data.u, 80);    % Send 80 bits to node 4 (actuator)
10    execetime = -1; % finished
11
12 end
```

Routine ctrlcode2.m, avec envoi vers l'ARCOM_1

Grâce à ce modèle nous pouvons faire une première simulation du calcul déporté. Cependant, bien que nous continuions de nous rapprocher du modèle réel, cette simulation ne prend pas en compte la gestion des tâches *Sensor*, *Controller*, et *Actuator*, sur une même interface. Autrement dit, la modélisation doit encore évoluer pour se rapprocher du modèle réel, un ARCOM qui gère en même temps les tâches Sensor, Controller, et Actuator.

La partie suivante décrit la conception d'une simulation plus proche de ce que le cahier des charges décrit, et montre les résultats obtenus.

6ème étape : Conception des deux interfaces

Cette partie permettra la conception de notre système final, à savoir que chacun de nos deux ordinateurs embarqués gèrent les tâches de *Sensor*, *Controller*, et *Actuator*, sur la même interface.

Afin d'illustrer cela, voici le Simulink de cette partie :

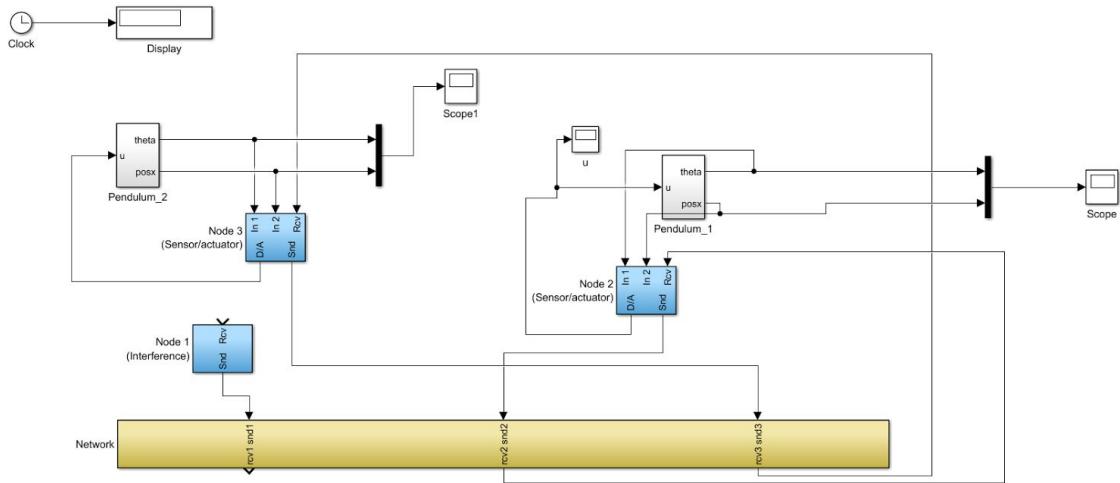


Schéma Simulink de notre système final

Notons que la structure du réseau a changé. Nous avons alors :

- Node 1 : Bloc d'interférence
- Node 2 : ARCOM_1, qui gère le pendule_1
- Node 3 : ARCOM_2, qui gère le pendule_2

Nous retrouvons bien une seule interface, un seul bloc, pour gérer les tâches *Sensor*, *Controller*, et *Actuator* de chaque ARCOM.

Par conséquent, des problèmes de gestion des tâches surgissent, comme leur ordonnancement, par exemple. Pour régler ces problématiques, nous modifions le code des deux initialisations des deux interfaces ARCOMs. Nous n'avons plus alors que deux tâches : "senscode_task.m" tâche périodique pour le capteur (*Sensor*), et "act_task" devient une interruption pour la restitution et le calcul de la commande (*l'Actuator* et le *Controller*).

En modifiant les déclarations de tâches de nos fichiers *sensot_actuator_init.m* et *sensor_actuator_init2.m*, nous obtenons les extraits de scripts suivant :

```

102
103
104 % IMPLEMENTATION : using the built-in support for periodic tasks
105
106 ttCreatePeriodicTask('senscode_task', offset, period, prio, 'senscode', data);
107 disp('SENSOR INIT1')
108
109
110
111 % Distributed control system: actuator node
112 %
113 % Receives messages from the controller and actuates
114 % the plant.
115 data.uChanel=1;
116 % Initialize TrueTime kernel
117
118
119
120 % Create actuator task
121 deadline = 100;
122 prio = 1;
123 %ttCreateTask('act_task', deadline, prio, 'actcode',data);
124 ttCreateTask('act_task', deadline, prio, 'ctrl_act_code2',data);
125

```

extrait de sensor_actuator_init.m modifié

```

102
103 % IMPLEMENTATION : using the built-in support for periodic tasks
104
105 ttCreatePeriodicTask('senscode_task', offset, period, prio, 'senscode2', data);
106 disp('SENSOR INIT2')
107
108
109 % Distributed control system: actuator node
110 %
111 % Receives messages from the controller and actuates
112 % the plant.
113 data.uChanel=1;
114 % Initialize TrueTime kernel
115
116
117
118 % Create actuator task
119 deadline = 100;
120 prio = 1;
121 %ttCreateTask('act_task', deadline, prio, 'actcode2',data);
122 ttCreateTask('act_task', deadline, prio, 'ctrl_act_codel',data);
123
124
125 %prio interruption
126 prio =1 ;
127 % Initialize network
128 ttCreateInterruptHandler('nw_handler', prio, 'msgRcvActuator');
129 ttInitNetwork(3, 'nw_handler'); % node #4 in the network
130 disp('ACTUATOR INIT2')
131

```

Extrait de sensor_actuator_init2.m modifié

Nous constatons bien que nos anciennes tâches « *act_task* » et “*controller_task*” ont fusionné pour une seule tâche d'interruption, qui déclenchera soit une routine de restitution de commande, soit une tâche de calcul déporté de commande.

Si cette tâche était scindée en deux interruptions distinctes, une pour l'*Actuator* et une autre pour le *Controller*, un conflit d'interruption pour arriver et bloquer notre système.

Penchons-nous maintenant sur la routine des tâches d'interruptions. Pour gérer les deux types de routines, *Actuator* et *Controller*, nous avons construit une structure conditionnelle. Nous prendrons l'exemple de L'ARCOM_1, de sa tâche *act_task* de routine *ctrl_act_code2.m*. Le script de *ctrl_act_code2.m* est le suivant :

```
1  function [execetime, data] = ctrl_act_code2(seg, data)
2
3  switch seg,
4  case 1,
5      %%% Notre tableau
6      retreived_data = [1 1 1];
7
8      % Obtain sensor value
9      retreived_data = ttGetMsg;
10
11     % Identificateur du type de Message reçu
12     flag = retreived_data(1);
13
14     if (flag == 0)
15
16         data.y = [retreived_data(2) ;retreived_data(3)];
17         % State space equation of Observer
18         data.x=data.Adc*data.x+data.Bdc*data.y;
19         data.u=data.Cdc*data.x;
20
21         data.message = 'controller';
22
23
24     else % from controller 2
25         data.u = retreived_data(2);
26         data.message = 'actuator';
27
28
29
30     end
31     execetime = 0.0005;
32
33     case 2,
34     |
35     if(strcmp(data.message, 'actuator'))% from Actuator
36         ttAnalogOut(1, data.u)
37
38     else % from controller 2
39         flag = 1; % from controller 1
40         sent_data = [flag ; data.u];
41         ttSendMsg(3, sent_data, length(sent_data)); % Send to node 1 (actuator)
42
43     end
44     execetime = -1; % finished
45
46
```

Script *ctrl_act_code2.m*

Détaillons l'algorithme du script ci-dessous, afin de cerner tous ses aspects. Dans un premier temps, nous récupérons les données sous la forme d'un tableau *retreived_data* de 3 éléments, variant selon l'élément *Flag*.

Cette nouvelle variable *Flag* indique si le message reçu est destiné à la partie *Controller* ou à la partie *Actuator*. Concrètement, Si *Flag* est égale à 0, le message reçu contient des données pour le calcul déporté de la commande (*Controller*), et au contraire, si *Flag* est égale à 1, les données reçues sont destiné à l'application de la commande (*Actuator*).

Les différentes trames de données sont alors les suivantes :

Si *Flag* = 0 :

<i>Trame de donnée, reçue depuis le sensor de l'autre pendule (3 éléments)</i>		
<i>Flag = 0</i>	<i>L'angle Théta</i>	<i>La distance X</i>

Si *Flag* = 1 :

<i>Trame de donnée, reçue depuis le Controller de l'autre pendule (3 éléments)</i>		
<i>Flag = 1</i>	<i>Commande U</i>	<i>X</i>

Constitution des trames de données reçues

Au sein du segment 1, si *Flag* est égale à 0, nous effectuons le calcul déporté de la commande, sinon nous récupérons la commande. Dans tous les cas, nous indiquons à *data.message* le scénario choisi. Les valeurs de *data.message* sont « *actuator* » si l'axe de l'*Actuator* a été choisi, et la valeur « *controller* » si l'axe du *Controller* a été entrepris.

Dans le segment 2, en fonction de *data.message*, nous appliquons la commande au pendule associé à l'ARCOM, ou nous envoyons les données au second pendule.

Bien entendu, pour mettre en place cet algorithme au sein de l'interruption, nous devons adapter les routines *senscode_task* des deux interfaces. En effet, il faut insérer « *Flag = 0* » dans les trames des données émises par les *Sensors*, afin de réaliser les calculs des *Controllers* présentés précédemment.

Prenons la routine *senscode2.m* de la tâche *senscod_task* lié au pendule2 :

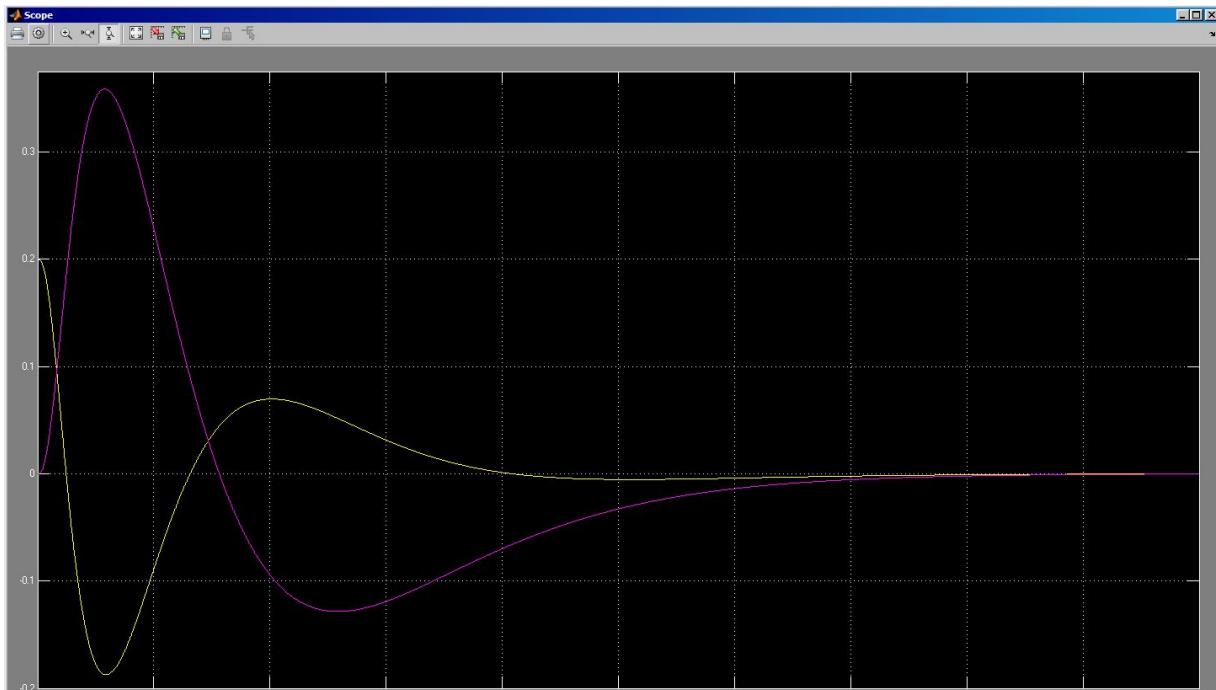
```

1 function [exectime, data] = senscode2(seg, data)
2
3 - switch seg
4 - case 1
5 -     data.th = ttAnalogIn(data.thChanel);      % Read pendulum angle signal
6 -     data.d = ttAnalogIn(data.dChanel);        % Read cart horizontal distance
7 -     data.donnee = [ 0; data.th ;data.d];
8 -     exectime = 0.0003;
9 - case 2
10 -    % Send message (80 bits) to node 2 (controller)
11 -    ttSendMsg(2, data.donnee, length(data.donnee));
12 -    exectime = 0.0004;
13 - case 3
14 -    exectime = -1; % finished
15 - end
16

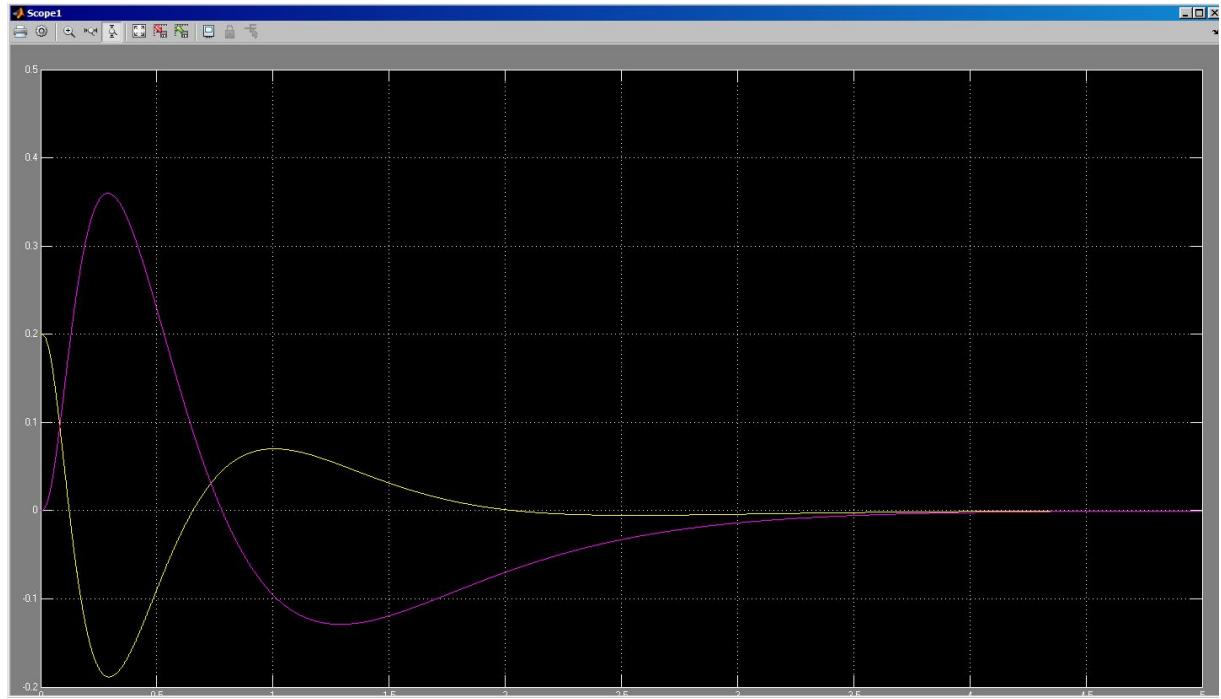
```

Script senscode2 pour l'envoi d'une trame de trois données, dont Flag

Suite à toutes ces modifications, nous pouvons simuler notre modèle. Ainsi nous obtenons les résultats suivants :



Résultats à la sortie de l'Actuator de l'ARCOM du pendule 1



Résultats à la sortie de l'Actuator de l'ARCOM du pendule 2

Nos deux pendules fonctionnent, et notre cahier des charges a été respecté pour la partie simulation sous *MathLab/Simulink*.

7ème étape : Tests et définition des limites du système

Nous avons testé les limites de notre système en modifiant la fonction “sencode”. En effet, nous avons modifié la valeur du delay.

```
function [exectime, data] = senscode(seg, data)

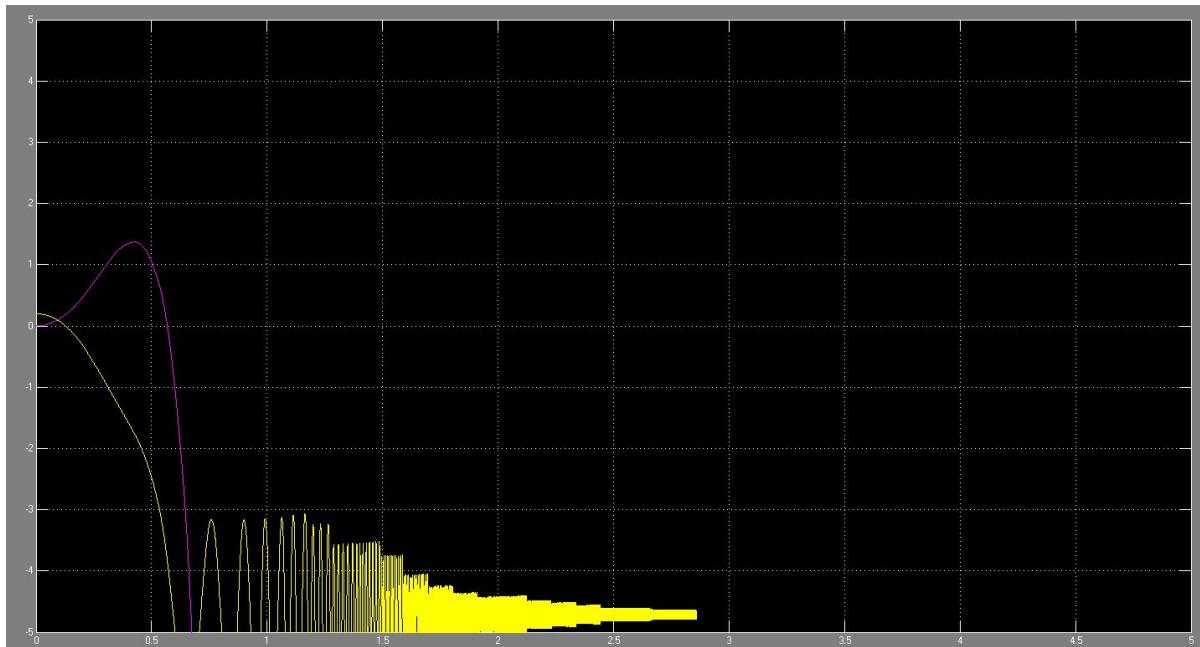
switch seg,
case 1,
    data.th = ttAnalogIn(data.thChanel);
    data.d = ttAnalogIn(data.dChanel);
    data.donnee = [ 0; data.th; data.d];
    exectime = 0.0003;
case 2,
    ttSendMsg(3, data.donnee, length(data.donnee)); % Send message (80 bits) to node 3 (controller)
%exectime = 0.0004;
    exectime = 0.021;

case 3,
    exectime = -1; % finished
end
```

% Read pendulum angle signal
% Read cart horizontal distance

La valeur initial de “exectime” est de 0.0004. A cette valeur, nos deux courbes convergent l'équilibre du pendule est donc atteint.

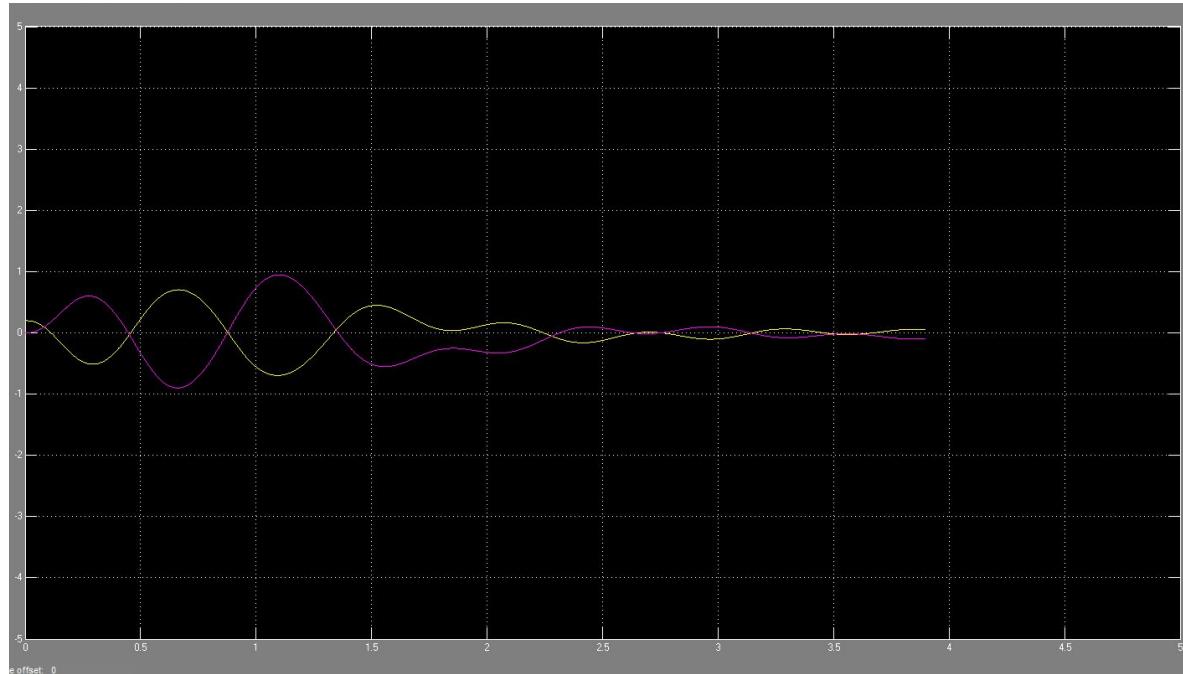
Nous testons d'abord pour une grande valeur de exectime = 0.01 pour observer l'effet du delay sur l'équilibre du pendule.



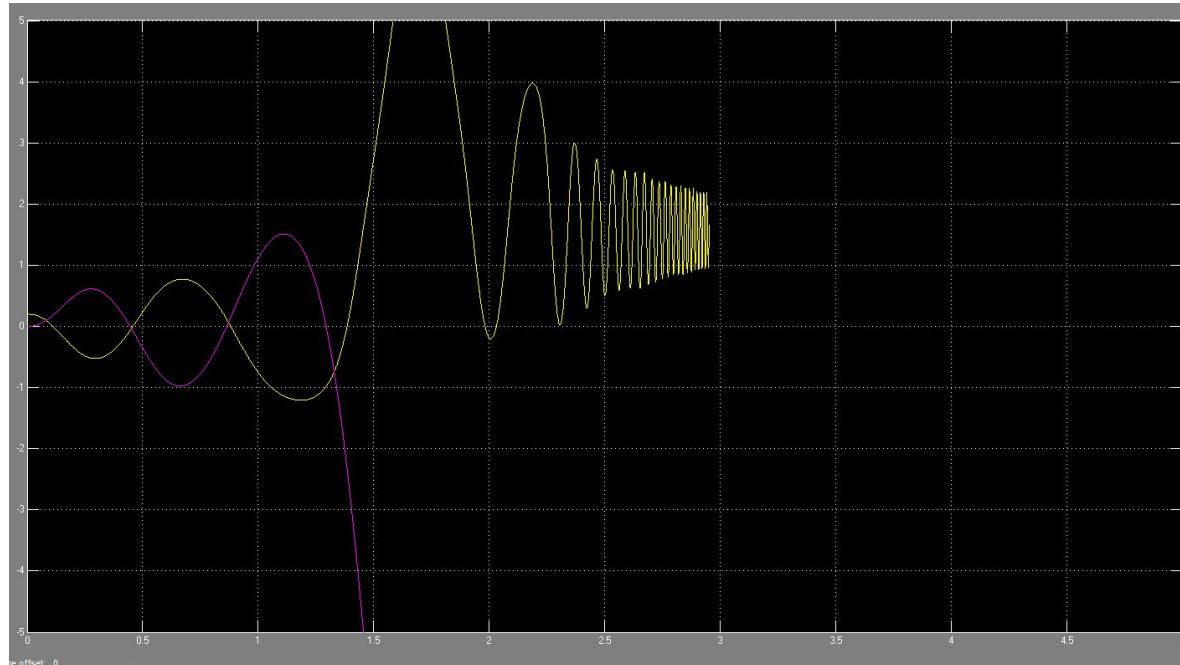
Pour cette valeur les deux courbes divergent et le pendule n'atteint pas sa position d'équilibre.

Après avoir effectué plusieurs test pour déterminer la bande passante pour laquelle notre pendule reste en équilibre, nous avons remarqué que le basculement s'opère autour de la valeur 0.02.

Ainsi, dans la figure suivante nous observons que l'équilibre est encore atteint pour une valeur de 0.021.



Cependant, la figure qui suit nous montre que pour une valeur de 0.025 l'équilibre est rompu et les courbes divergent.



La valeur limite du delay de notre système est donc 0.021.

Conclusion première partie

Dans cette première partie de l'étude de cas nous avons commencé par la modélisation d'un seul pendule sous Matlab. Par la suite nous avons inseré le BUS CAN et fusionner les blocs Sensor & Actuator.

La dernière partie a consisté à intégrer un deuxième pendule et déporter le calcul du premier sur le deuxième et vice versa. Enfin, nous avons entrepris la conception des deux interfaces ARCOM_1 et ARCOM_2, et nous avons analysé les résultats obtenus en testant les limites du système.

Ces différentes étapes de la partie automatique, nous ont permis de mieux comprendre le système. Nous avons alors une meilleure visualisation de celui-ci.

Après avoir fini de développer, tester et d'analyser la partie automatique de cette étude de cas, nous passons à la partie temps réel. Nous sommes alors prêts à tester notre système sur l'ARCOM.

BIBLIOGRAPHIE

TRUETIME 1.4—Reference Manual par Martin Ohlin, Dan Henriksson et Anton Cervin

TrueTime: Real-time Control System Simulation with MATLAB/Simulink par Dan Henriksson, Anton Cervin, Martin Ohlin, Karl-Erik Årzén

Introduction à la commande des systèmes linéaires par A.Cela

ANNEXE N°1 : Commandes de True Time et descriptions

Command	Description
ttInitKernel	Initialize the kernel
ttInitNetwork	Initialize the network interface
ttCreateTask	Create a task
ttCreatePeriodicTask	Create a periodic task
ttCreateInterruptHandler	Create an interrupt handler
ttCreateExternalTrigger	Associate a interrupt handler with an external interrupt channel
ttCreateMonitor	Create a monitor.
ttCreateEvent	Create an event.
ttCreateLog	Create a log structure and specify data to log
ttCreateMailbox	Create a mailbox for inter-task communication
ttCreateSemaphore	Create a counting semaphore
ttNoSchedule	Switch off the schedule generation for a specific task or interrupt handler
ttNonPreemptable	Make a task non-preemptable
ttAttachDLHandler	Attach a deadline overrun handler to a task
ttAttachWCETHandler	Attach a worst-case execution time overrun handler to a task.
ttAttachHook (C++ only)	Attach a run-time hook to a task
ttAbortSimulation	Abort the simulation

Table 1 Commands used to create and initialize TRUETIME objects, and to control the simulation.

Command	Description
ttCreateJob	Create a job of a task.
ttKillJob	Kill the running job of a task.
ttEnterMonitor	Attempt to enter a monitor.
ttExitMonitor	Exit a monitor.
ttWait	Wait for an event.
ttNotify	Notify the highest-priority task waiting for an event.
ttNotifyAll	Notify all tasks waiting for an event.
ttLogNow	Log the current time.
ttLogStart	Start a timing measurement for a log.
ttLogStop	Stop a timing measurement and save in the log.
ttTryPost	Post a message to a mailbox (non-blocking).
ttTryFetch	Fetch a message from a mailbox (non-blocking).
ttPost	Post a message to a mailbox (blocking).
ttFetch	Fetch a message from a mailbox (blocking).
ttRetrieve	Read the actual message fetched from a mailbox.
ttTake	Take a semaphore.
ttCreateTimer	Create a one-shot timer and associate an interrupt handler with the timer.
ttCreatePeriodic	Timer Create a periodic timer and associate an interrupt handler with the timer.
ttSleepUntil	Remove a specific timer.
ttSleep	Sleep until a certain point in time.
ttAnalogIn	Sleep for a certain amount of time.
ttAnalogOut	Read a value from an analog input channel.
ttSetNextSegment	Write a value to an analog output channel.
ttInvokingTask	Set the next segment to be executed in the code function (to implement loops and branches)
ttCallBlockSystem	Get the name of the task that invoked a task overrun handler.
ttSendMsg	Call a Simulink block diagram from within a code function.
ttGetMsg	Send a message over a TRUETIME network.
ttDiscardUnsentMessages	Get a message that has been received over a TRUETIME network.
	Delete any unsent messages.

ttSetNetworkParameter	Set a specific network parameter on a per node basis.
ttSetKernelParameter	Set a specific kernel parameter on a per node basis.

Table 3 Real-time primitives

Partie Temps Réel :

Simulation sous RTAI

Encadrant: R.Kocik
 R.Hamouche

Context de la partie Temps Réel	36
Architectures du systèmes	39
Conception des architectures du systèmes	39
Architecture matérielle	39
Architecture système	40
Architecture logicielle	41
Description du DAC	42
Description de l'ADC	44
Description du module source_sensor.c	47
Description du module source_controller.c	49
Description de la communication CAN	51
Description des FIFO	58
Annexes	62
Pour l'acquisition des données	62
Pour la communication CAN	63

Context de la partie Temps Réel

Pour la seconde partie de notre projet, nous avons débuté par de la recherche de documentation sur les propriétés de notre carte ARCOM grâce à la Datasheet mise à disposition. Après avoir réussi à comprendre le fonctionnement de la carte nous avons suivi les directives pour commencer sa configuration et ensuite la construction de notre code.

L'objectif de cette partie est de compléter le parcours du cycle de développement d'un système embarquable réparti et temps réel, de l'analyse à la réalisation d'un prototype opérationnel.

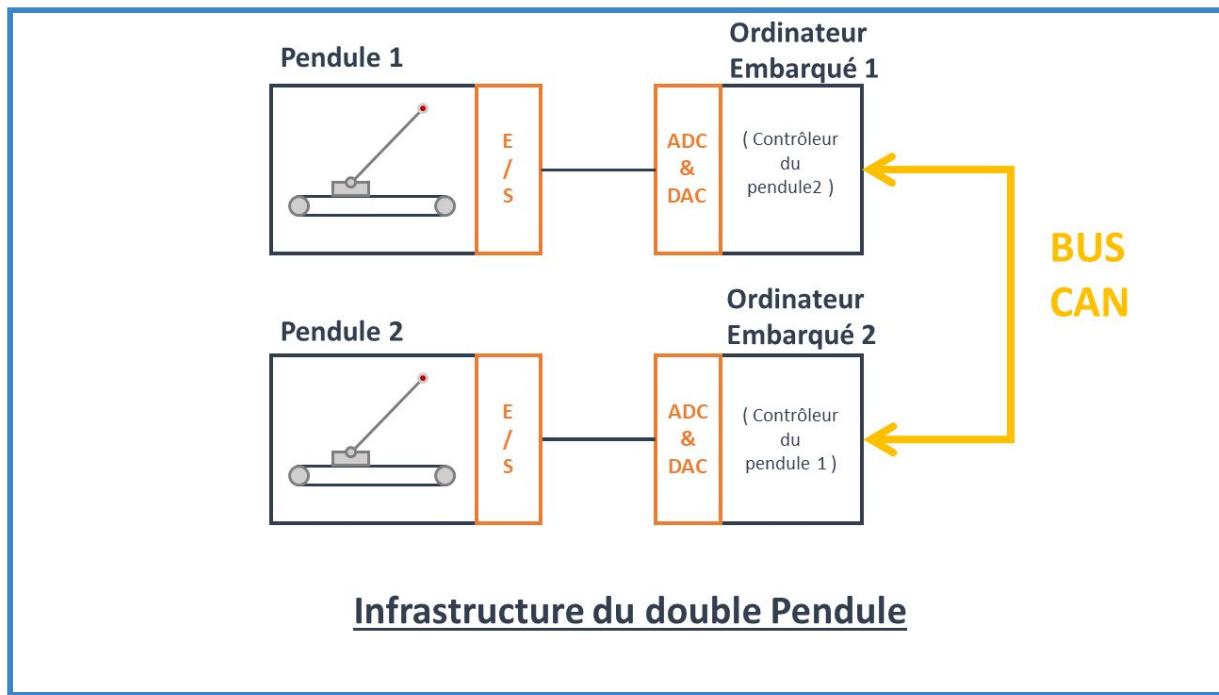
Notre système intègre alors :

- Des procédés réels à commander ainsi que leurs interfaces électroniques
- Des processeurs assurant différentes fonctionnalités (acquisition, conversion de commande, contrôleur ...),
- Un réseaux de communication entre les différents processeurs.

La carte ARCOM dispose d'une connexion Ethernet ainsi que Linux RTAI 3.4 et est connectée à une carte AIM104 CAN.

Une première étape consistera à implémenter le système en mono-processeur. Une seconde étape sera d'implémenter ce système en multi-processeurs et donc d'assurer une communication par bus CAN.

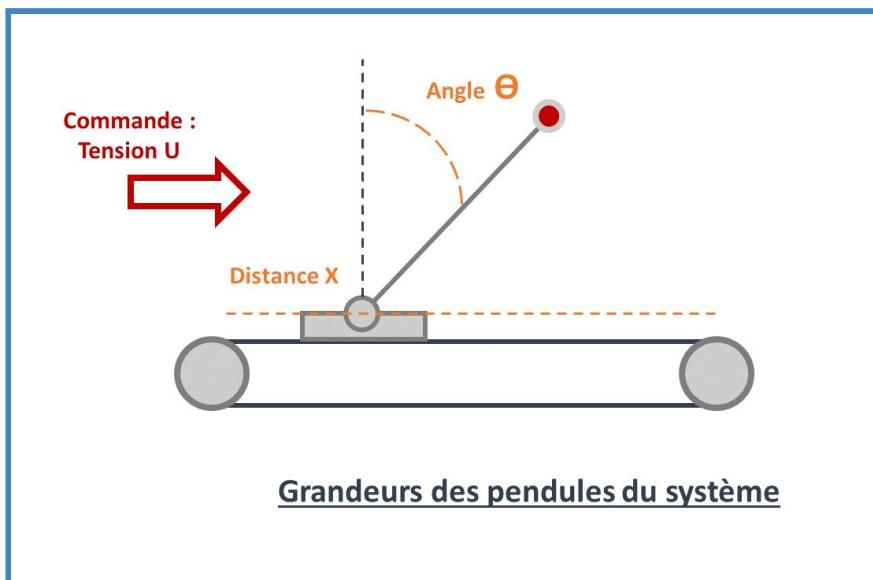
Cette étude de cas porte, donc, sur l'étude d'un double pendule inversé. Notre objectifs est de stabiliser les deux systèmes en temps réel. Tout au long de ce projet, nous utilisons l'architecture suivante :



Nous notons que les deux pendules sont chacun reliés à un ordinateur embarqué. Les missions de ces derniers sont :

- l'acquisition des données de leurs pendules associés
- l'échange des informations via bus CAN
- la réalisation d'un calcul déporté de la commande

Pour contrôler nous calculons la commande **U**, une tension en Volt, grâce à l'angle Θ de la tige du pendule, et à la position **X** du pendule sur le rail. Le schéma suivant illustre l'utilisation de ces grandeurs :



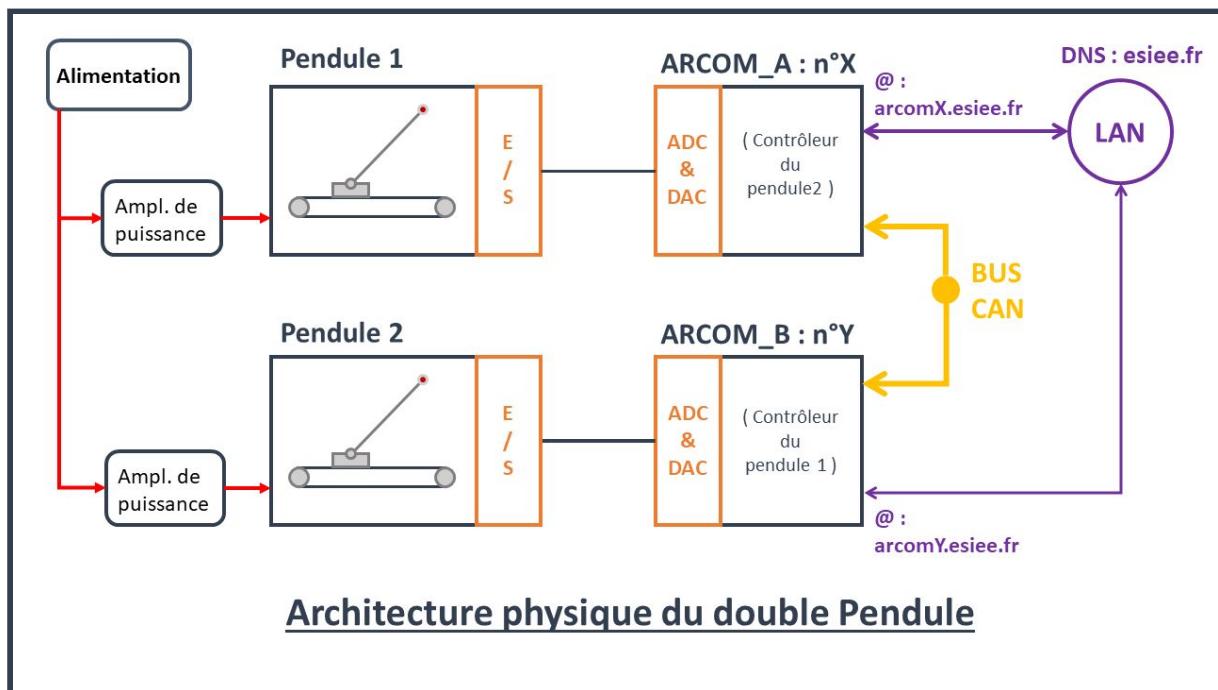
Dans la partie Temps réel de l'étude de cas, nous allons concevoir l'architecture informatique du double pendule inversé. A partir des résultat de la partie automatique, nous pourrons alors appliquer l'asservissement optimal à notre système.

Architectures du systèmes

Conception des architectures du systèmes

Architecture matérielle

Ci-dessous est présenté l'architecture matérielle de la maquette utilisée :



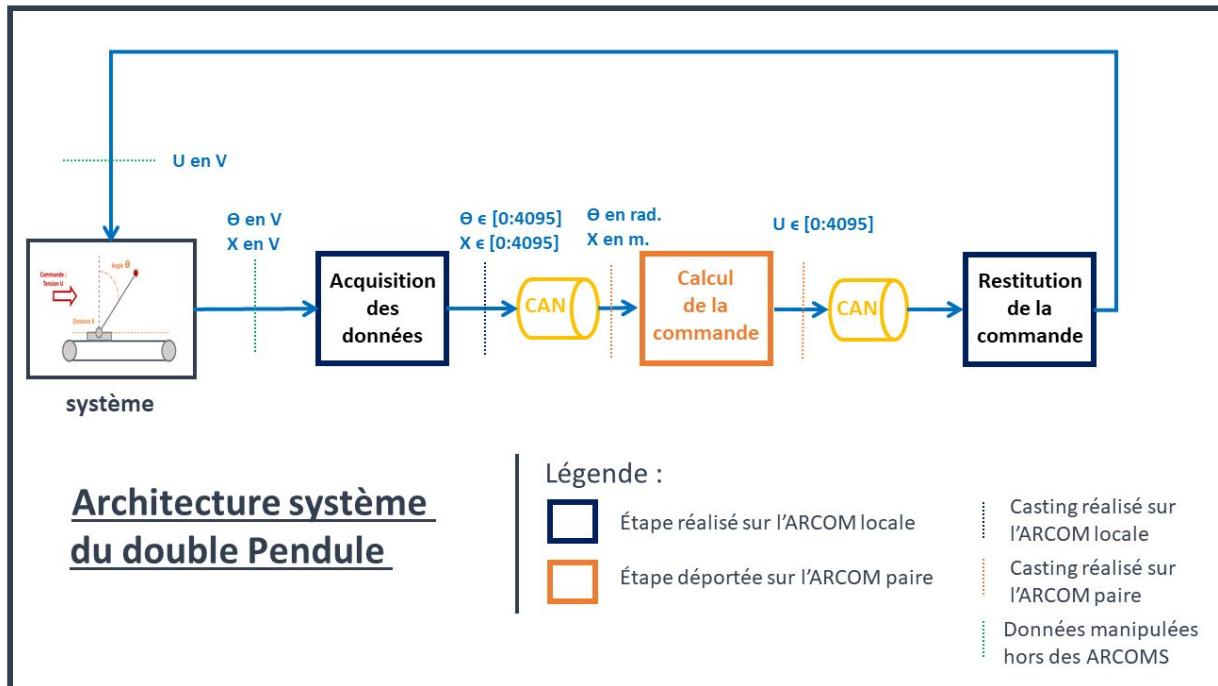
Les pendules sont reliés aux deux ordinateurs embarqués, les ARCOMs, via les interfaces ADC, carte d'acquisition des données, et DAC, carte restitution de la commande.

Les logiciels embarqués sont conçus sur un ordinateur sous environnement linux RTAI, externe aux deux ARCOMS. Les programmes sont transmis aux ordinateurs embarqués via communication SSH.

De plus, les deux ARCOMS présentent chacune une carte SJA-1000 permettant de communiquer via un bus CAN. L'ARCOM_A et l'ARCOM_B peuvent ainsi s'échanger des informations via une communication intégrant le temps réel, grâce au bus CAN.

Architecture système

Dans cette section, nous allons illustrer le cheminement des données, de leurs acquisitions jusqu'à la restitution de la commande. Le schéma suivant montre cette vision de l'étude de cas :



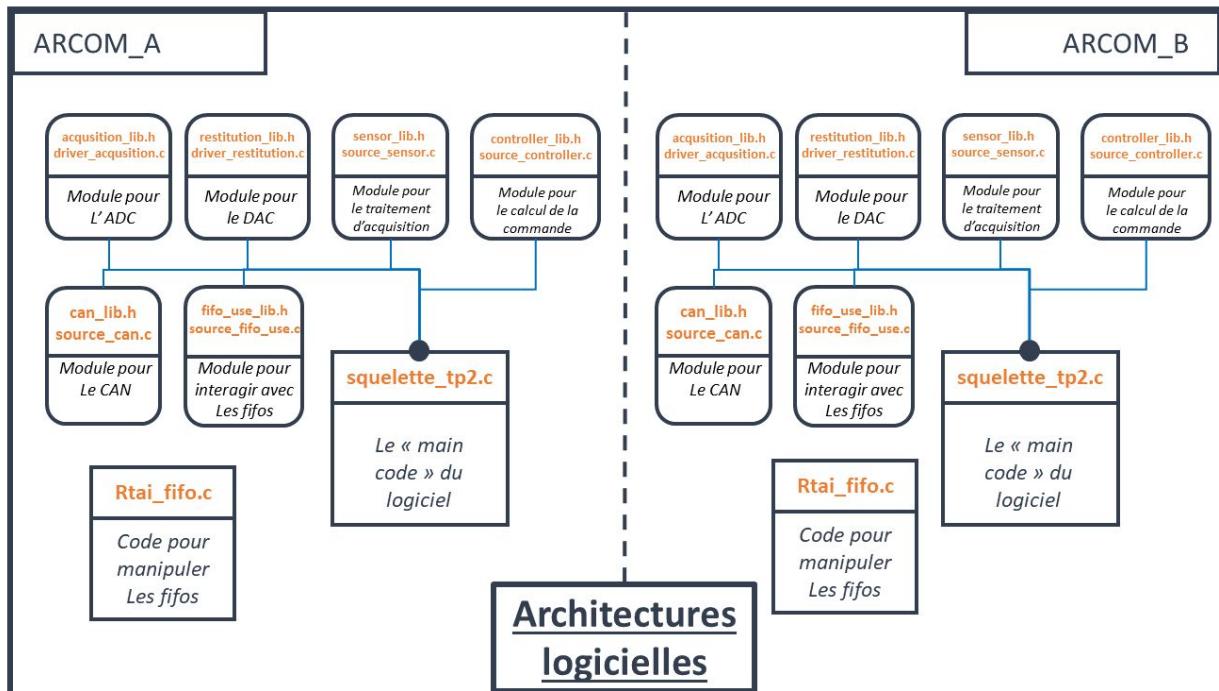
Nous pouvons comprendre les traitements suivant :

- l'angle théta et la position x d'un pendule sont captés par l'ADC.
- l'ARCOM associé au pendule, formatera les informations pour les envoyer via CAN à l'ARCOM paire.
- L'ARCOM paire réceptionne les informations, puis calcul la commande à appliquer.
- La commande est formatée pour son envoi sur la première ARCOM
- L'ARCOM hôte du pendule reçoit la commande, puis transmet l'information au DAC qui s'appliquera.

Nous remarquons que les données prennent différents types de format, en fonction des étapes de la chaîne de traitement. Après les avoir notés, nous étudierons les raisons de ces formats au fil de nos analyses.

Architecture logicielle

Enfin voici une architecture logicielle de nos programmes, les liens entre les différents modules que nous avons conçus :



Nous remarquons que pour nos deux ARCOMs, nous retrouvons :

- Un code **squelette_tp2.c** qui exécute toutes les fonctions nécessaires à la chaîne de traitement.
- Des modules de gestion de l'ADC et du DAC, pour piloter ces deux interfaces
- Un module de communication CAN, pour dresser les fonctions essentielles à aux échanges de messages sur le bus CAN
- Un module sensor (source_sensor.c + sensor_lib.h), qui enveloppe les fonctions pour piloter les traitements d'acquisition de des données.
- Un module controller (source_controller.c + controller_lib.h) pour gérer le calcul de la commande
- Un module fifo_use (source_fifo_use.c + fifo_use.h) afin de fournir des fonctions de manipulation des fifos, adaptées aux données à analyser.
- Un code Rtaififo.c qui permet d'exploiter les fichiers fifo (utile pour la supervision du système)

Ainsi, toutes ces composantes permettent de réaliser la chaîne de traitement, la gestion des communications, et la supervision du système.

Description du DAC

Afin d'envoyer au moteur la commande calculée qui permet de le contrôler nous avons utilisé la carte d'acquisition ADVANTECH PCM-3712 PC104. Cette carte convertit une valeur numérique en tension en ses sorties.

Pour manipuler l'ADVANTECH PCM-3712 PC104, nous avons créé un driver de restitution, driver_restitution.c, et sa librairie, restitution_lib.h.

Dans un premier temps, nous devons initialiser la carte et configurer le canal de sortie. Nous implémentons ensuite une fonction SetDA() qui positionne la valeur à envoyer dans le CNA de la sortie channel de la carte.

Configuration de la carte :

- La carte est configurée avec l'adresse de base 0x300
- Jumpers
 - Base Address Switch Setting SW1
 - 1 2 3 4 5 6 7 8
 - on X X X X X X
 - off X X
 - JP11: Asynchronous mode
 - Channel 1 (JP1,JP3,JP5) : bipolar +-10v
 - Channel 2 (JP2,JP4,JP10) : bipolar +-10V

Ce module fournit donc les fonctions suivantes:

- *int init3712(void)* : initialisation de la carte d'acquisition. Cette fonction sera appelée dans l'init_module. Retourne un entier nul lorsque l'initialisation est correcte.

```
22
23 int init3712(void){
24     outb(0x80,Output);
25     return 0;
26 }
27
```

- void *SetDA(int channel, int value)* : Cette fonction positionne la valeur *value* dans le CNA de la sortie *channel* de la carte

```

29
30 void SetDA(unsigned char channel, int value){
31
32
33     if (channel==0x01){
34         data_L= value & 0xFF;
35         data_H= (value>> 8) & 0x0F;
36         outb(data_L,LowByteDataC1 );
37         outb(data_H,HighByteDataC1);
38         outb(0xFF,Synchro);
39
40     }else if (channel == 0x00){
41         data_L= value & 0xFF;
42         data_H= (value>> 8) & 0x0F;
43         outb(data_L,LowByteDataC0);
44         outb(data_H,HighByteDataC0);
45         outb(0xFF,Synchro);
46     }
47 #ifdef DEBUG
48     printk("data_L %x \n", (unsigned char) data_L);
49     printk("data_H %x \n", (unsigned char) data_H);
50 #endif
51 }
52

```

Selon le chanel choisi, on coupe l'octet de la valeur et on assigne cette valeur à deux registres différents.

On récupère les bits de poids faible dans le registre de “*LowByteData*” en appliquant le masque “0xFF”, puis les bits de poids forts dans le registre de “*HighByteData*” en appliquant un décalage de 8 vers la droite puis le masque “0x0F”.

Afin de tester le bon fonctionnement de notre programme de contrôle moteur, nous avons utilisé la fonction escalier pour voir l'évolution de la tension au borne du moteur.

Nous avons, aussi, essayé d'envoyer une commande avec une valeur numérique supérieur à 4096, nous observons alors à l'oscilloscope un retour à la valeur minimale (-10V) puis une nouvelle incrémentation.

Description de l'ADC

Afin de récupérer les informations de position, angulaire Θ et spatiale D, nous avons utilisé la carte d'acquisition *ADVANTECH 3718HG PC104*. Les capteurs situés sur la maquette du pendule nous délivrent Θ et D sous forme de tension. Par la suite, nous traduisons dans un autre module le voltage en radian pour Θ , et en mètres pour D.

Pour manipuler l'*ADVANTECH 3718HG PC104*, nous avons créé un driver d'acquisition, *driver_acquisition.c*, et sa librairie, *acquisition_lib.h*. Avant de décrire les fonctions que nous employons, penchons-nous sur la procédure d'acquisition à suivre.

Dans un premier temps, nous devons initialiser la carte et configurer le range des canaux d'entrée. En effet, la position Θ est lue sur un premier canal, et la position D sur un second canal distinct. Nous devons définir le « range », la plage de voltage exploitable pour chaque canal d'acquisition.

Nous prenons un range de [-10 V ; 10 V] pour le canal 0 et le canal 1, canaux respectifs de la lecture de Θ et la lecture de D. De plus, nous n'utiliserons ni de génération d'interruption, ni de transfert DMA, et n'utiliserons pas le *Pacer*. Le déclenchement de l'acquisition sera Logiciel. Nous obtenons alors la fonction d'initialisation suivante :

```
40 | int initADC(void){  
41 |     outb(0x00,Control); //  
42 |     outb(0x01,Parser); //Desactivation du Pacer  
43 |  
44 |     // Range des canaux 0 et 1 |  
45 |     ADRRangeSelect(0x00,8);  
46 |     ADRRangeSelect(0x01,8);  
47 |  
48 |     // lecture du canal 0 par defaut  
49 |     setChannel(0);  
50 |  
51 |     return 0;  
52 | }
```

Fonction d'initialisation du DAC, du driver *driver_acquisition.c*

Pour maintenant réaliser l'acquisition, nous devons :

1. Sélectionner le canal à lire.
2. Déclencher l'acquisition en écrivant une valeur quelconque dans le registre *Low Byte* (*BASE+0*).
3. Vérifier que la conversion a pris fin, en lisant le bit INT du registre de statut (*BASE+8*).
4. Lire les informations dans les registres des poids faibles et de poids fort des données, respectivement (*BASE+0*) et (*BASE+1*).
5. Convertir les données en valeur entière.

Pour suivre cette procédure, nous utilisons une fonction du module *source_sensor.c* du nom de *data_acquisition*, qui utilise le driver *driver_acquisition.c*.

```
52 ▼ void data_acquisition(int canal_debut, int canal_fin, u16 table[],int num_pendule){  
53     int plage = canal_fin - canal_debut+1;  
54  
55     while (plage > 0){  
56         setChannel(plage-1);  
57  
58         // rt_task_wait_period();  
59  
60         read_channel();  
61         table[plage-1] = read_adc();  
62         // data[plage-1] = voltage_to_valeurs( conversion_adc(table[plage-1]),plage-1 ,num_pendule);  
63         plage --;  
64         // rt_task_wait_period();  
65     }  
66 }  
67 }  
68 }
```

Fonction *data_acquisition* , du module *source_sensor.c*

Initialement, la conversion des données en valeur entière étaient effectué dans cette fonction. L'utilisation du CAN a modifié le comportement de *data_acquisition*, afin de garder que l'acquisition (expliqué dans la partie CAN).

Pour mieux cerner les fonctions utiles à l'acquisition, mettons à part la conversion des données en valeurs entières pour le moment, nous la développerons dans la partie consacrée au module *source_sensor.c*.

La fonction *set_Channel*, correspond à l'**étape 1**, et elle permet de positionner le canal à lire :

```
13 ▼ void setChannel(int in_chanel){  
14     unsigned char canaux =(in_chanel<<4)+in_chanel; //unsigned int ?  
15     if(in_chanel>3)  
16         return -1;  
17     outb(canaux,MuxScan);  
18 }  
19 }
```

Set_Channel, driver *driver_acquisition.c*

Son objectif est de modifier le registre de MuxScan, qui permet de déterminer la plage de canal à lire. Les 4 bits de poids forts correspondent au premier canal à lire, et les 4 bits de poids faibles au dernier canal. Si on met un canal de début et un canal de fin identiques, un seul canal sera lu. Et c'est l'objectif de cette fonction, à son appel, *setChannel* permet de choisir le canal à lire.

Les fonctions *read_channel* et *read_adc* sont conçus au tour des mêmes mécanismes, une écriture dans le registre de poids faible des données (**étape 2**), une phase où l'on attend que le bite INT, du registre de statu, passe à 1 (**étape 3**), et enfin, la lecture des registres des poids faibles et de poids fort des données (**étape 4**).

Voici les fonctions *read_channel* et *read_adc* :

```

60 ▼ u16 read_channel(void) {
61     outb(0xFF, LowByteData);
62     while((inb(Status) & 0x90) != 0x10);
63     return (u16)(inb(LowByteData) & (0x0F));
64 }
65

```

Fonction read_channel, dans le driver driver_acquisition.c

```

53 ▼ u16 read_adc(void) {
54     outb(0x0F, LowByteData);
55     while((inb(Status) & 0x90) != 0x10);
56     return ((u16)((inb(LowByteData) >> 4) | (inb(HighByteData) << 4)));
57 }
58

```

Fonction read_adc, dans le driver driver_acquisition.c

Les deux fonctions renvoient des données en *U16*, soit les valeurs hexadécimales des registres des poids faibles et de poids fort des données.

Notons que la valeur du canal est stockée dans les quatre bits de poids faibles du registre de poids faible des données.

Il reste une dernière fonction présente dans ce driver, la fonction *conversion_adc*. Sa mission est de convertir la valeur fraîchement acquise, comprise entre 0 et 4096, en une tension. Cette fonction correspond à l'**étape 5**, décrite précédemment. Son corps est le suivant :

```

66 ▼ float conversion_adc(u16 param) {
67
68     return ((float)(param) - 2048.0) / 205.0 ;
69 }

```

Fonction conversion_adc, dans le driver driver_acquisition.c

Pour obtenir cette équation de conversion, nous devons prendre en compte plusieurs paramètres :

- Les valeurs restituées par l'ADC *ADVANTECH 3718HG PC104* sont comprises entre 0 et 4096.
- Si notre range est de [-10 V ; 10 V], -10 V correspond à la valeur 0, 10 V correspond à la valeur 4096, et 0 V à la valeur 2048.
- Comme il existe une loi proportionnelle entre l'échelle de valeur de l'ADC et leurs correspondance en volt, nous avons calculé un coefficient proportionnel, et nous avons trouvé *205.0*.
- Maintenant, nous avons compris la conception du driver d'acquisition, nous pourrons aborder le module *source_sensor.c*, correspondant au bloc *Sensor* de la partie simulation.

Description du module source_sensor.c

Le module `source_sensor.c` a pour librairie `sensor_lib.h`, et sa mission est de transmettre les données Θ , en radian, et D, en mètre, aux calculateurs.

Précédemment, nous avons analysé le fonctionnement du driver d'acquisition, `driver_acquisition.c`, et nous avons constaté que les valeurs acquises finales ne pouvaient être manipulées qu'en voltage. Comme le driver d'acquisition ne devrait servir qu'à piloter la carte ADC, nous ne pouvons pas effectuer la conversion du voltage vers les radians ou les mètres. Il est alors nécessaire de créer un module capable de calculer les correspondances.

Notons que la correspondance voltage vers radian/mètre dépend du pendule manipulé. En effet, chaque pendule a subi de l'usure, et ne possède pas les mêmes intervalles de tension, que ce soit pour Θ , ou pour D. Nous utilisons un `define NUM_PENDULUM` dans `squelette_tp2.c` pour indiquer à nos fonctions le pendule utilisé.

```
42  /*  
43  
44     Numéro du pendule utilisé  
45     ARCOM 12 -> NUM_PENDULUM 1  
46     ARCOM 21 -> NUM_PENDULUM 2  
47 */  
48 #define NUM_PENDULE 1
```

Définition de NUM_PENDULUM, Extrait de squelette_tp2.c

Au sein du module `source_sensor.c`, nous avons la fonction `voltage_to_valeur`

```
44 float voltage_to_valeurs(float voltage, int canal, int num_pendule){  
45     if (canal == 0){  
46         return voltage_to_theta(voltage,num_pendule);  
47     } else {  
48         return voltage_to_distance(voltage,num_pendule);  
49     }  
50 }  
51 }
```

Fonction voltage_to_valeurs, dans le module source_sensor.c

Cette fonction calcule l'angle Θ ou la distance D, en fonction des paramètres procurés, grâce à une fonction de calcul de Θ , `voltage_to_theta`, et à une fonction de calcul de D, `voltage_to_distance`.

```

31 ▼ float voltage_to_theta(float voltage, int num_pendule){
32 ▼     switch(num_pendule){
33         case 1:
34             //return (int)( ( (voltage + 0.615)/31.983 )*1000 );
35             return ( (voltage + 0.587)/22.329 );
36             break;
37         case 2: //à calculer
38             return (voltage - 0.14)/20.94;
39             break;
40     }
41
42 }
```

Fonction voltage_to_theta , dans le module source_sensor.c

```

18 ▼ float voltage_to_distance(float voltage, int num_pendule){
19 ▼     switch(num_pendule){
20         case 1:
21             //return ( (voltage + 1.8)/26.4 );
22             return (voltage + 1.16)/16.846 ;
23             break;
24         case 2: // à calculer
25             return (voltage - 0.76)/16.6 ;
26             break;
27     }
28
29 }
```

Fonction voltage_to_distance , dans le module source_sensor.c

Dans ces deux fonctions nous prenons en compte le pendule étudié, qui possède sa propre équation de correspondance.

Nous rappelons que la routine d'acquisition de données est présente dans ce module, comme avant l'implantation du CAN, la correspondance était effectuée dans *data_acquisition* :

```

52 ▼ void data_acquisition(int canal_debut, int canal_fin, u16 table[],int num_pendule){
53     int plage = canal_fin - canal_debut+1;
54
55     while (plage > 0){
56         setChannel(plage-1);
57
58         // rt_task_wait_period();
59
60         read_channel();
61         table[plage-1] = read_adc();
62         // data[plage-1] = voltage_to_valeurs( conversion_adc(table[plage-1]),plage-1 ,num_pendule);
63         plage --;
64         // rt_task_wait_period();
65
66     }
67
68 }
```

Fonction data_acquisition , du module source_sensor.c

Description du module source_controller.c

Le module *source_controller.c*, de librairie *controller.lib.h*, permet le calcul de la commande. Ce module est similaire au bloc Controller de notre *partie simulation* sous *Mathlab*.

Source_controller.c est composée de la fonction *obs_count*, et de fonctions de bornes logicielles que nous détaillerons ultérieurement. La fonction *obs_count* est similaire à la fonction *obs_count* du bloc *Controller* de notre *partie simulation*, son but est d'obtenir la commande via des calculs matriciels. Son corps de fonction est le suivant :

```
36 ▼ float obs_count(float theta, float distance){  
37     float Xc_1=0.0, Xc_2=0.0, Xc_3=0.0, Xc_4=0.0, U=0.0;  
38  
39     Xc_1 = ( x_1*0.6300 + x_2*-0.1206 + x_3*-0.0008 + x_4*0.0086 + theta*0.3658 + distance*0.1200 ) ;  
40     Xc_2 = ( x_1*-0.0953 + x_2*0.6935 + x_3*0.0107 + x_4*0.0012 + theta*0.0993 + distance*0.3070 ) ;  
41     Xc_3 = ( x_1*-0.2896 + x_2*-1.9184 + x_3*1.1306 + x_4*0.2351 + theta*1.0887 + distance*2.0141 ) ;  
42     Xc_4 = ( x_1*-3.9680 + x_2*-1.7733 + x_3*-0.1546 + x_4*0.7222 + theta*3.1377 + distance*1.6599 ) ;  
43  
44     U= (Xc_1*-80.3092 + Xc_2*-9.6237 + Xc_3*-14.1215 + Xc_4*-23.6260 );  
45  
46     x_1 = Xc_1 ;  
47     x_2 = Xc_2 ;  
48     x_3 = Xc_3 ;  
49     x_4 = Xc_4 ;  
50  
51     return U;  
52  
53 }  
54 }
```

Fonction obs_count, dans le module controller.c

Notons que les valeurs *Xc_1*, *Xc_2*, *Xc_3*, et *Xc_4*, correspondent aux éléments du vecteur *Xn+1* de l'équation :

$$\{X_{n+1} = Adc * Xn + Bdc * E\} Y = X_{n+1} * Cdc$$

Rappelons les valeurs trouvées en simulation pour les matrices A, B, et C :

```
Adc=[0.6300 -0.1206 -0.0008 0.0086  
      -0.0953 0.6935 0.0107 0.0012  
      -0.2896 -1.9184 1.1306 0.2351  
      -3.9680 -1.7733 -0.1546 0.7222];  
Bdc=[0.3658 0.1200  
      0.0993 0.3070  
      1.0887 2.0141  
      3.1377 1.6599];  
Cdc=[-80.3092 -9.6237 -14.1215 -23.6260];  
Ddc=[0 0];
```

Valeurs des matrices trouvées en simulation sous Mathlab

De plus le vecteur E, correspond aux entrées de notre système :

$$E = \Theta D$$

La valeur Y correspond à la commande trouvée, soit l'équivalent de la variable U du script précédent.

Ainsi, nous pouvons calculer la commande à appliquer à un pendule via la fonction *obs_count*, du module *source_controller.c* .

Description de la communication CAN

Pour réaliser un calcul déporté de la commande, nos deux ARCOMs doivent communiquer via BUS CAN. Nous avons alors créé un module pour la communication CAN, à savoir *soucre_can.c* avec sa librairie *can.lib.h*.

De plus, notons que même si nos deux ARCOMs réalisent le calcul de la commande de leur pair, nous avons créé deux répertoires : *ARCOM_A* et *ARCOM_B*. Les programmes sont adaptés aux besoins de chacune de nos deux ARCOMs. Cette différenciation est plus adaptée quand nous effectuons du calcul déporté, sur deux pendules configurés différemment.

Dans le module *source_can.c*, nous retrouvons une fonction d'initialisation de la communication CAN, une fonction d'envoi de message, une fonction de réception de message.

Ces fonctions ont été développées durant l'unité d'E4 spécifique au CAN, mais nous rappelons dans cette partie leurs corps de fonction :

```
22 ▼ int initCan(){
23
24
25     outb(0x01,CAN_CONTROL); //reset mode
26     outb(0xFF,CAN_A_CODE_REG); //Filtrage
27     outb(0xFF,CAN_A_MASK_REG); //Pas de filtre
28     outb(0x03,CAN_BUS_TIMING_REG0); //Vitesse CAN
29     outb(0x1C,CAN_BUS_TIMING_REG1); //Vitesse CAN
30     outb(0xFA,CAN_OUTPUT_CONTROL_REG); //OUtput control register
31     outb(0x02,CAN_CONTROL); //Mode avec interruption
32
33
34     return 0;
35 }
36 }
```

Fonction d'initialisation de la communication CAN, dans le module source_can.c

Remarquons que nous utiliserons une communication CAN avec interruption au sein de ce projet. De plus, comme nous n'avons que deux nœuds dans notre communication CAN (nos deux ARCOMs), nous ne cherchons pas alors à filtrer les messages reçus.

```

39 | /* Dans le cas de l'utilisation des interruptions */
40 |
41 | int receiveCan(unsigned int* buffer)
42 | {
43 |     printk("CAN status = %x \n",inb(CAN_STATUS));
44 |     char descriptor_0;
45 |     char descriptor_1;
46 |     int id;
47 |     int i, taille;
48 |     descriptor_0 = inb(CAN_RECEIVE_DESCRIPTOR_0);
49 |     descriptor_1 = inb(CAN_RECEIVE_DESCRIPTOR_1);
50 |
51 |     id = (descriptor_0<<3) | (descriptor_1>>5);
52 |     printk(" id = %d \n",id);
53 |     taille = descriptor_1&0x0F;
54 |     if(taille != 0)
55 |     {
56 |         for(i=0; i<taille; i++)
57 |         {
58 |             buffer[i] = inb(CAN_RECEIVE_DATA+i);
59 |             printk("le message recu = %x \n",buffer[i]);
60 |         }
61 |         outb(0x04, CAN_COMMAND);
62 |         return id;
63 |     }
64 |     else
65 |     {
66 |         return 0;
67 |     }
68 | }

```

Fonction de réception, dans le module source can.c

```

99 | void sendCan(int id, int taille, unsigned int* message)
100 | {
101 |     printk("on est dans la fonction \n");
102 |     unsigned char descriptor_0, descriptor_1;
103 |     int i;
104 |     if(id>2047 || id<0 || taille>8 || taille<1)
105 |     {
106 |         return -1;
107 |     }
108 |     else
109 |     {
110 |         printk("CAN status = %x \n",inb(CAN_STATUS));
111 |         if((inb(CAN_STATUS)&0x04) == 0x04) //registre libre
112 |         {
113 |             /* descriptor_0 = (id>>3)&0x00FF;
114 |              descriptor_1 = ((id&0x0007)<<5)+(taille&0x001F);*/
115 |
116 |             descriptor_0 = (id>>3);
117 |             descriptor_1 = ( (id&0x0F)<<5 )|(taille);
118 |             printk(" id_1 = %x et id_2 =%x \n\n",descriptor_0,descriptor_1);
119 |             outb(descriptor_0, CAN_SEND_DESCRIPTOR_0);
120 |             outb(descriptor_1, CAN_SEND_DESCRIPTOR_1);
121 |             for(i=0; i<taille; i++)
122 |             {
123 |                 printk("%d : le message envoye est %x \n",i,message[i]);
124 |                 outb(message[i], CAN_SEND_DATA+i);
125 |             }
126 |         }
127 |         else
128 |         {
129 |             return -1;
130 |         }
131 |     }
132 |

```

Fonction d'envoi, dans le module source can.c

Comme rappelé dans la fonction d'initialisation de la communication CAN, notre programme génère des interruptions à chaque réception de message. Nous avons fait ce choix afin de limiter les temps de latence procurés par un calcul déporté de la commande.

Dans le fichier `squelette_tp2.c`, nous avons initialisé l'interruption et créé une routine d'interruption, `handle_receive`. Voici ces éléments :

```
203 ▼ static int tpcan_init(void) {
204
205     int ierr;
206     int ierr2;
207     RTIME now;
208
209
210     /* Initialisation du handler d'interruption*/
211     rt_request_global_irq(irq_7841, handle_receive);           //mise en place du handler
212     rt_startup_irq(irq_7841);                                //activation de la ligne d'interruption
213     rt_global_sti();                                         //re-activation des int
214
215     //rtf_create(fifo,1000);
216
217
218     /* creation tache periodiques*/
219     rt_set_oneshot_mode();
220     ierr = rt_task_init(&tache_horloge,routine_recup_data ,0,STACK_SIZE, PRIORITE, 1, 0);
221     start_rt_timer(nano2count(TICK_PERIOD));
222     now = rt_get_time();
223
224     rt_task_make_periodic(&tache_horloge, now, nano2count(PERIODE_CONTROL));
225
226     return(0);
227 }
228
```

Initialisation de l'interruption, dans squelette_tp2.c

```
229 ▼ static void tpcan_exit(void) {
230     stop_rt_timer();
231     rt_task_delete(&tache_horloge);
232
233     //rtf_destroy(fifo);
234
235     rt_shutdown_irq(irq_7841);
236     rt_free_global_irq(irq_7841);
237
238 }
239
```

Sortie de l'interruption, dans squelette_tp2.c

```

154  /*** Routine d'interruption ****/
155  void handle_receive(void){
156
157  if((inb(CAN_INTERRUPT)&0x01) == 1){
158      #ifdef DEBUG
159          printk("\n");
160          printk("\n");
161          printk("test RCPT \n");
162      #endif
163      id_recu = receiveCan(messageR);
164      #ifdef DEBUG
165          printk("RCPT OK \n ");
166      #endif
167
168  if (id_recu == 1){
169
170      theta_recu     = messageR[0] | (messageR[1]<<8 );
171      distance_recu = messageR[2] | (messageR[3]<<8 );
172
173      data_ready[0] = voltage_to_valeurs( conversion_adc(theta_recu),0,NUM_PENDULE);
174      data_ready[1] = voltage_to_valeurs( conversion_adc(distance_recu),1,NUM_PENDULE);
175
176      U = obs_count( data_ready[0], data_ready[1] )*coeff_model ;
177
178      value_U= (int)(U*1000.0);
179      send_U[0]= value_U & 0xFF;
180      send_U[1]= (value_U & 0xFF00)>>8;
181
182      sendCan(2, 2, send_U);
183  }else if (id_recu == 2){
184
185      #ifdef DEBUG
186          U_recu     = (float)( messageR[0] | messageR[1]<<8 )/1000.0 ;
187          #endif
188          printk("U = %d \n", (int)(U_recu*1000.0));
189          printk(" ----- Restitution ----- \n");
190          value_test = (unsigned int)( (U_recu/10.0)*2048.0+2048.0 );
191          SetDA(0,value_test);
192          printk(" ----- FIN ----- \n");
193          rt_ack_irq(irq_7841);
194          return 1;
195
196  }else {
197
198      rt_pend_linux_irq(irq_7841);
199
200
201 }

```

Routine d'interruption handle_receive, dans squelette_tp2.c

Nous remarquons la présence de deux fonctions : *rt_ack_irq(irq_7841)* et *rt_pend_linux_irq(irq_7841)*. Ces deux fonctions réalisent respectivement : l'acquittement l'interruption, et ajoute l'interruption au mode IRQ de linux.

Pour comprendre le contenu de la routine d'interruption *handle_receive*, nous devons analyser les tâches réalisées par la partie d'acquisition/restitution, et les tâches réalisées par la partie de calcul de la commande.

Sur nos deux ARCOMS, nous avons une tâche périodique, de routine *routine_recup_data*, qui réalise l'acquisition des données, et les envoie à son ARCOM pair. La routine de récupération des données est la suivante :

```

129 ▼ void routine_recup_data(long arg){
130
131 ▼ while(1){
132         printk("\n");
133         printk("-----Debut de conversion----- \n");
134         data_acquisition(0,1,table,NUM_PENDULE);
135     /*
136     rt_task_wait_period();*/
137
138         printk(" theta = %d, distance = %d \n", table[0], table[1] );
139
140         data_send[0]= table[0] & 0xFF;
141         data_send[1]= (table[0] & 0xFF00)>>8;
142         data_send[2]= table[1] & 0xFF;
143         data_send[3]= (table[1] & 0xFF00)>>8;
144
145         sendCan(1, 4, data_send);
146         printk("envoi realise\n");
147         rt_task_wait_period();
148
149     }
150
151 }

```

Routine de la tâche périodique tache_horloge, dans squelette_tp2.c

Avant d'être envoyées, les données sont segmentées octets par octets, puis ces segments sont envoyés un à un.

Notons que comme nous avons deux données, Θ et D , de deux octets, à chaque exécution de cette routine, nous envoyons quatre octets. Nous envoyons alors Θ et D sont leur valeur d'acquisition, comprise entre [0 ; 4096]. Les valeurs en float étant sur quatre octets, et les valeurs d'acquisition sur deux octets, une conversion sur le CAN pair permet de réduire la taille des trames CAN.

Les messages avec des données d'acquisition ont un ID égale à 1.

Les données d'acquisition sont traitées sur l'ARCOM pair. La réception s'effectue dans la routine d'interruption handle_receive. Dès à présent, notons que notre routine d'interruption est capable d'identifier deux types de messages : pour ID égale à 1, et ID égale à 2. Nous pouvons l'illustrer de cette manière :

```

154 /**
155 ▼ void handle_receive(void){
...
168 ▼     if (id_recu == 1){
...
184 ▼     }else if (id_recu == 2){
...
204 }

```

Extrait de la fonction handle_receive, dans squelette_tp2.c

En effet, nous avons défini la messagerie suivante :

IDs	Tailles des données (DLC)	Types des données	Types de messages	Réponses
1	4 octets	Int	Messages comportant des données d'acquisition (Θ et D).	Calcul de la commande pour l'ARCOM pair.
2	2 octets	Int	Messages comportant la commande à appliquer au pendule de l'ARCOM pair.	Application de la commande au pendule de l'ARCOM local.

Messagerie des deux ARCOMS, par rapport à la communication CAN

Si nous nous penchons sur les corps, des réponses aux différents messages, nous obtenons les extraits suivants pour l'ID reçu égale à 1 et l'ID reçu égale à 2 :

```

154  /*** Routine d'ibterruption ****/
155  void handle_receive(void){

168  ...
169
170      if (id_recu == 1) {
171
172          theta_recu     = messageR[0] | (messageR[1]<<8) ;
173          distance_recu = messageR[2] | (messageR[3]<<8) ;
174
175          data_ready[0] = voltage_to_valeurs( conversion_adc(theta_recu),0,NUM_PENDULE);
176          data_ready[1] = voltage_to_valeurs( conversion_adc(distance_recu),1,NUM_PENDULE
177
177          U = obs_count( data_ready[0], data_ready[1] )*coeff_model ;
178
179          value_U= (int)(U*1000.0);
180          send_U[0]= value_U & 0xFF;
181          send_U[1]= (value_U & 0xFF00)>>8;
182
183          sendCan(2, 2, send_U);

184  ...
185
186  ...
187
188  ...
189
190  ...
191
192  ...
193
194  ...
195
196  ...
197
198  ...
199
200  ...
201
202  ...
203
204  }

```

Pour ID reçu égale à 1, extrait de handle_receive, dans squelette_tp2.c

```

154  /*** Routine d'ibterruption ****/
155  void handle_receive(void){

...

```

```

184     }else if (id_recu == 2){
185
186         U_recu      = (float)( messageR[0] | messageR[1]<<8 )/1000.0 ;
187 #ifdef DEBUG
188         printk("U = %d \n", (int)(U_recu*1000.0));
189 #endif
190         printk(" ----- Restitution ----- \n");
191         value_test = (unsigned int)( (U_recu/10.0)*2048.0+2048.0 );
192         SetDA(0,value_test);
193         printk(" ----- FIN ----- \n");
194
195     }
196
197     ...
198
199
200 }

```

Pour ID reçu égale à 2, extrait de handle_receive, dans squelette_tp2.c

Notes pour ID égale à 1 :

La conversion des valeurs donnée par la carte d'acquisition en radian ou en mètre, s'effectue bien dans ce bloc.

La commande est traduite en *int* avec une précision décimal de 10⁻³, afin d'alléger le trafic et de garantir une application fluide des commandes.

Notes pour ID égale à 2 :

La commande est traduite en *float*, convertit en échelle de valeur propre à la carte de restitution, puis appliquer au pendule de l'ARCOM local.

Grâce à cette nouvelle infrastructure, nous arrivons à réaliser le calcul déporté des deux commandes de nos deux ARCOMs.

Rappelons que pour obtenir ces résultats, nous sommes passés par plusieurs étapes comme la réception par scrutation, et le calcul déporté de la commande d'un seul ARCOM. Cette approche incrémentale nous a permis de réaliser l'infrastructure actuelle, une communication CAN utilisant la réception par interruption, et permettant les calculs simultanés des deux commandes de nos deux ARCOMs, de manière simultanée.

Description des FIFO

Afin de déterminer les raisons pour lesquelles nos pendules ne se stabilisent pas, nous avons décidé d'utiliser la méthode FIFO pour récupérer les valeurs de l'angle, la position et la commande reçues durant nos tests.

Nous récupérons alors dans trois fichiers text deux colonnes : une avec le temps pour lequel la variable à été récupérée et dans la deuxième colonne la variable récupérée (angle, position ou commande).

Nous utiliserons par la suite ces fichiers pour construire des graphs de nos variables en fonction du temps grâce à l'outil XGRAPH.

Nous avons commencé par créer trois modules “*rtai_fifo_angle*”, “*rtai_fifo_mesure*”, “*rtai_fifo_commande*” qui ont pour but de lire les variables mises dans les fifos et de les écrire dans trois fichiers text différents.

Les trois fichiers sont décrits de la manière suivante :

```
9 int main(){
10
11 char buf[10];
12 int v;
13 const char *fifo = "/dev/rtf1";
14 const char *path = "/home/arcom/text.txt";
15
16 //#####
17
18 int size_not_read = 1 ;
19
20 //#####
21
22 int opf= open(fifo, O_RDONLY);
23 int opp= open(path, O_WRONLY | O_CREAT, 0644);
24
25
26 do{
27
28     size_not_read = read(opf,&v, sizeof(char));
29     sprintf(buf,"%d",v\0);
30
31
32     //write(opp,buf, strlen(buf));
33
34     write(opp,&v, strlen(buf));
35 }while(size_not_read >0 );
36
37
38 close(opf);
39 close(opp);
```

On commence par ouvrir en lecture simple le fichier fifo de l'ARCOM. On recopie son contenu dans un buffer. Puis on ouvre un fichier text et on écrit dedans le contenu de notre buffer.

Ensuite, nous avons crée le module “*source_fifo_use*” dans lequel nous avons mis les fonctions qui servent à convertir puis insérer nos variables dans les fifos.

La première fonction “*convertAscii*”, convertit les valeurs, qui vont être récupérés dans les fifos en char. Les valeurs finales seront donc en ASCII pour être copier dans le fichier text.

```

14
15 void convertAscii(int param, int values[]){
16
17     if (param < 0 ){
18         param = -1*param;
19         int moins = 45;
20         values[0] = (param/10000)+48;
21         values[1] =((param/1000)*1000 - (values[0]-48)*10000)/1000 +48;
22         values[2] =((param/100)*100 - (values[0]-48)*10000 - (values[1]-48)*1000 )/100 +48;
23         values[3] =((param/10)*10 - (values[0]-48)*10000 - (values[1]-48)*1000 - (values[2]-48)*100 )/10 +48;
24         values[4] = (param - (values[0]-48)*10000 - (values[1]-48)*1000 - (values[2]-48)*100 -(values[3]-48)*10) + 48;
25         values[5] = -1;
26
27     }else {
28         values[0] =(param/10000)+48;
29         values[1] =((param/1000)*1000 - (values[0]-48)*10000)/1000 +48;
30         values[2] =((param/100)*100 - (values[0]-48)*10000 - (values[1]-48)*1000 )/100 +48;
31         values[3] =((param/10)*10 - (values[0]-48)*10000 - (values[1]-48)*1000 - (values[2]-48)*100 )/10 +48;
32         values[4] =(param - (values[0]-48)*10000 - (values[1]-48)*1000 - (values[2]-48)*100 -(values[3]-48)*10) + 48;
33         values[5] = 0;
34     }
35
36 }
```

On prend chaque caractère, puis on le convertit et l'insère dans un tableau pour construire notre valeur finale.

Puis, la fonction “*insert_fifo*” qui prend en paramètre la valeur de la variable (angle, position ou commande) à insérer dans le fifo et le numéro du fifo.

```

38
39 void insert_fifo(int values[], int num_fifo){
40     int i = 0;
41     int moins = 45;
42
43
44     if (values[5] < 0){
45         rtf_put( num_fifo, &moins, 1);
46     }
47     while (i< 5){
48         rtf_put(num_fifo, &values[i], 1);
49         i++;
50     }
51     rtf_put(num_fifo, " \n ", 3);
52 }
53 }
```

La fonction “*insert_time*” est décrite comme la fonction “*insert_fifo*”, elle nous sert à insérer le temps dans le fifo.

```
54 void insert_time(int values[], int num_fifo){  
55     int i = 0;  
56     int moins = 45;  
57  
58     if (values[5] < 0){  
59         rtf_put( num_fifo, &moins, 1);  
60     }  
61     while (i< 5){  
62         rtf_put(num_fifo, &values[i], 1);  
63         i++;  
64     }  
65     rtf_put(num_fifo, " ", 1);  
66 }  
67 }
```

Enfin, la fonction “*add_to_fifo*”, prend en paramètre le temps, la variable à insérer et le numéro du fifo. Elle appelle les deux fonctions précédemment décrites pour récupérer les valeurs converties et les insérer dans le fifo correspondant.

```
68  
69 void add_to_fifo(int time, int param, int num_fifo){  
70     int values[6];  
71     int date[5];  
72  
73     convertAscii(param, values);  
74     convertAscii(time, date);  
75  
76     insert_time(date, num_fifo);  
77     insert_fifo(values, num_fifo);  
78 }
```

Cette fonction va être appelée dans le “*squelette_tp2*” après la réception de l’angle et de la position et le calcul de la commande.

Nous avons créé un fichier “*bash_app*” dans lequel on écrit les commandes pour compiler les trois fichiers .c et lancer le programme qui permet de copier le contenu des fifos dans des fichiers text.

```
4 /opt/x86_64/gcc/gcc-3.4.4/bin/gcc -m32 -march=i386 rtai_fifo_angle.c -o rtai_fifo_angle  
5 /opt/x86_64/gcc/gcc-3.4.4/bin/gcc -m32 -march=i386 rtai_fifo_mesure.c -o rtai_fifo_mesure  
6 /opt/x86_64/gcc/gcc-3.4.4/bin/gcc -m32 -march=i386 rtai_fifo_commande.c -o rtai_fifo_commande
```

En parallèle de l'envoi de notre code sur l'ARCOM avec la commande "make", nous compilons nos trois fichiers "*rtai_fifo_angle*", "*rtai_fifo_mesure*", "*rtai_fifo_commande*" simultanément avec la commande suivante :

```
./rtai_fifo_angle & ./rtai_fifo_mesure & ./rtai_fifo_commande
```

Puis on ouvre un autre terminal pour arrêter le processus et récupérer les fichiers .txt avec les commandes suivantes :

```
ps -aux pour recuperer le pid
```

```
kill pid
```

Une fois les fichiers texte générés, nous souhaitons les récupérer de l'ARCOM vers notre ordinateur. On ouvre dans un nouveau terminal dans lequel on tape la commande suivante :

```
scp arcom@arcom21.esiee.fr:/home/arcom/mesure.txt ~/Desktop/
```

Exemple pour lequel nous récupérons le fichier "mesure.txt" de l'ARCOM 21 vers notre Desktop.

Nous réitérons ensuite la commande pour les deux autres fichiers "angle.txt" et "mesure.txt". Ensuite nous utilisons la commande suivante pour générer le graphique : **xgraph mesure.txt**.

Retour sur expérience et piste d'amélioration

Lors de ce projet nous sommes passés par toutes les phases de la mise en place d'un système embarqué, chose que nous n'avions expérimenté que partiellement lors de précédentes unités. Cette vision globale nous a permis de bien saisir les enjeux et les difficultés de la conception et l'implémentation d'un tel système. D'autant plus que nous étions en autonomie quasi-complète, ce qui nous a poussé à gérer nos difficultés.

Nous pouvons dire que ce projet permet d'appréhender le quotidien et les enjeux du métier d'ingénieur. Nous pensons que cette expérience a été très bénéfique pour notre parcours et que nous sommes satisfaits de terminer notre cursus par un projet qui nous a certes demandé beaucoup d'énergie mais qui a été très formateur.

Nous suggérons d'intégrer dans toutes les étapes du cursus système embarqués de tels projets.

Une de nos difficultés majeur a été de comprendre les raisons pour lesquelles la commande envoyé au pendule ne permettait pas de le stabiliser. Pour pallier cette difficulté nous avons manipulé le fonctionnement FIFO. Nous avons aussi eu beaucoup de mal lors de ce développement ce qui nous a permis d'acquérir des compétences tant en autonomie que en savoir technique. Nous sommes, cependant, satisfait d'être arrivé au bout de ce développement.

Annexes

Dans cette partie, nous retrouvons les différends registre des cartes utilisées.

Pour l'acquisition des données

Carte ADC ADVANTECH 3718HG PC104	
BASE_ADC	0x320
LowByteData	(BASE_ADC+0)
HighByteData	(BASE_ADC+1)
MuxScan	(BASE_ADC+2)
DIOlowByte	(BASE_ADC+3)
Status	(BASE_ADC+8)
Control	(BASE_ADC+9)
Parser	(BASE_ADC+10)
DIOHighByte	(BASE_ADC+11)
Count0	(BASE_ADC+12)
Count1	(BASE_ADC+13)
Count2	(BASE_ADC+14)

Pour la communication CAN

Carte pour la communication CAN SJA1000	
CAN_BASE	0x180
CAN_CONTROL	CAN_BASE
CAN_COMMAND	CAN_BASE+1
CAN_STATUS	CAN_BASE+2
CAN_INTERRUPT	CAN_BASE+3
CAN_A_CODE_REG	CAN_BASE+4
CAN_A_MASK_REG	CAN_BASE+5
CAN_BUS_TIMING_REG0	CAN_BASE+6
CAN_BUS_TIMING_REG1	CAN_BASE+7
CAN_OUTPUT_CONTROL_REG	CAN_BASE+8
CAN_SEND_DESCRIPTOR_0	CAN_BASE+10
CAN_SEND_DESCRIPTOR_1	CAN_BASE+11
CAN_SEND_DATA	CAN_BASE+12
CAN_RECEIVE_DESCRIPTOR_0	CAN_BASE+20
CAN_RECEIVE_DESCRIPTOR_1	CAN_BASE+21
CAN_RECEIVE_DATA	CAN_BASE+22