# Projet Informatique Industrielle INSA 5$^{\text{ième}}$ année
# Commande d'un système robotique

Laurent Barbé

22 novembre 2022

# Chapitre 1

# Préambule

La maquette proposée est un système électromécanique entièrement réalisée à l'INSA : conception mécanique, fabrication des pièces, assemblage et programmation informatique. Nous avons pris le parti de ne pas utiliser une maquette pédagogique commerciale afin de vous proposer un système qui soit le plus pertinent par rapport au cours "Informatique Industrielle". Bien que nous ayons apporté un grand soin à chaque sous-systèmes, il se peut que certains problèmes persistent. C'est grâce à vos retours que nous aboutirons à une maquette pleinement fonctionnelle.

Le système d'exploitation Linux temps-réel est déjà installé ainsi que les outils permettant de communiquer avec le matériel de la maquette.

Il y a quatre maquettes disponibles, il faut donc former des groupes équilibrés (3 ou 4 personnes maximum par maquette). Il y a suffisamment de travail pour chaque membre d'un groupe. Voici les grandes lignes des travaux demandés, certaines étapes peuvent être menées en parallèle :

- **Mise en marche des maquettes** : Vérifier que tout est opérationnel.
- **Prise en main de l'environnement de développement** par la programmation d'une tâche temps-réel périodique, puis de mettre en œuvre la communication avec le matériel EtherCAT ;
- **Présentation du système et modélisation** pour appréhender le comportement du système. Établir les modèles géométriques direct et inverse,en prenant le paramétrage proposé, puis valider ces modèles sous Matlab en tenant compte du fait que ces modèles puissent être programmés en C.
- **Programmation d'une boucle d'asservissement** dans l'espace articulaire et validation du modèle géométrique direct.

La dernière séance sera consacrée à l'évaluation de vos connaissances sur le dispositif. Il vous sera demandé de faire une courte présentation des différents points abordés au cours des séances, puis de répondre à une série de questions en relation avec les maquettes et le cours d'informatique industrielle.

# Chapitre 2

# Le système robotique

## 2.1 Présentation de la maquette

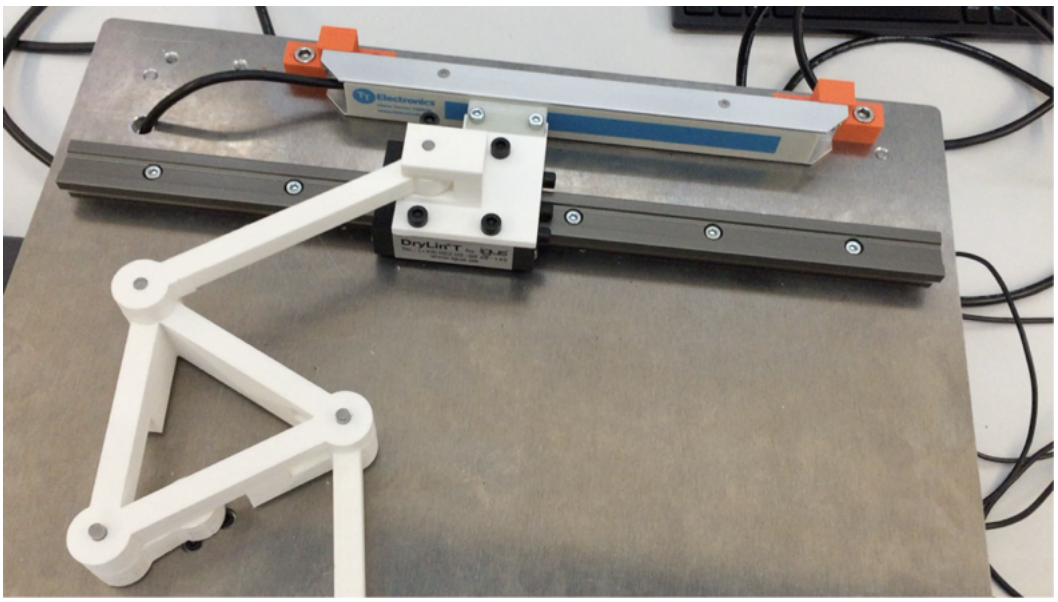


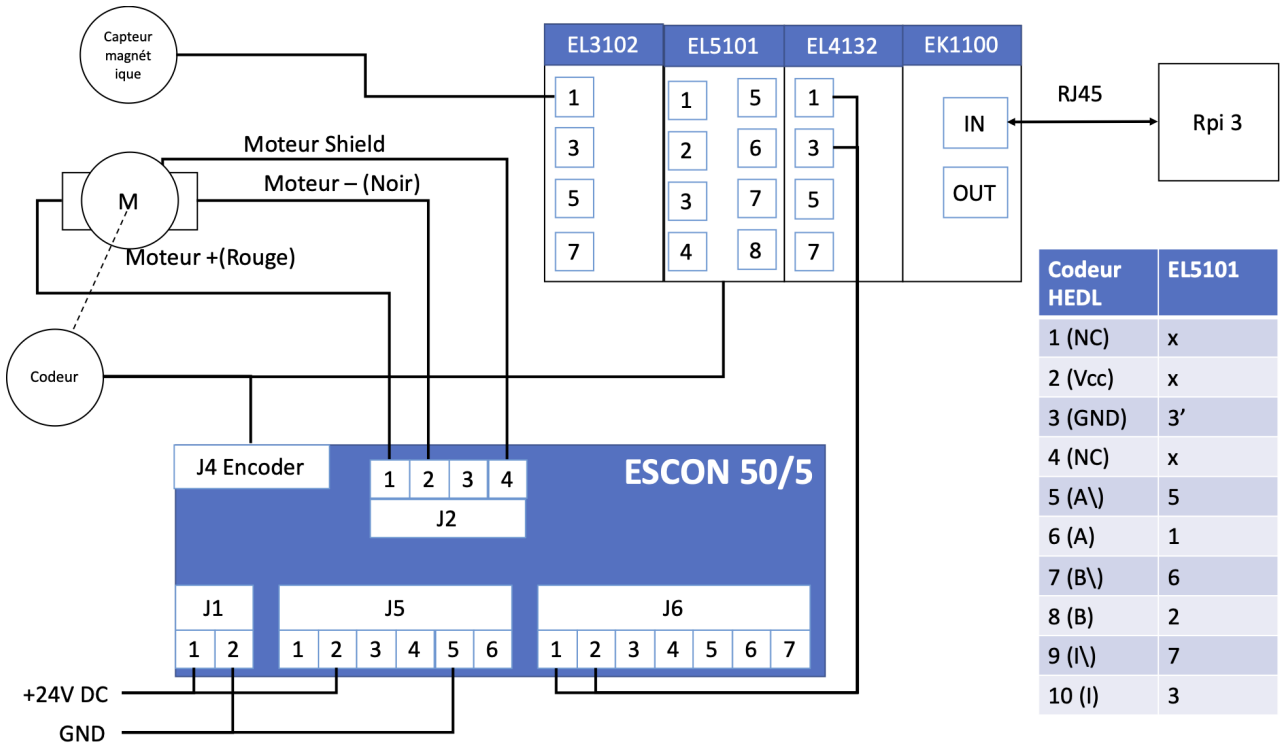
FIGURE 2.1 – Photo de la structure mécanique de la maquette.

La figure 2.1 présente le système robotique et le schéma cinématique est présenté figure 2.2. La structure mécanique est un système 4 barres accouplé à une glissière. La plupart des pièces de la structure ont été obtenues par impression 3D (donc elles sont relativement fragiles). La maquette est équipée :

- d'un moteur à courant continu, situé à l'articulation $A$, est de type DCX 26L 24V développé par la société Maxon Motor. Les spécifications principales du moteur sont :
  - Tension nominale : 24 $V$
  - Vitesse nominale : 4600 $tr/min$
  - Couple nominal : $52,3mNm$
  - Courant nominal : $1,25$ $A$
  - Constante de couple : $42,9$ $mNm/A$
- d'un réducteur GPX 26A avec un rapport de réduction de 83 : 1
- d'un variateur analogique ESCON configuré en vitesse. La tension envoyée au variateur $+/-$ $10V$ est l'image de la vitesse que l'on souhaite imposer à la sortie de l'axe moteur ;
- d'un capteur de position angulaire de type codeur incrémental HEDL-500 (500 $pts/tr$ sur quatre quadrants) situé en $A$ qui mesure $\theta_1$. Le capteur est monté directement sur

l'axe moteur ;
- d'un capteur de position linéaire de type potentiomètre magnétique situé en $F$ qui mesure $x$. Ce capteur mesure de $0 - 150\ mm$ sur une plage $0,25\ V$ à $4,75\ V$ ;
- un Raspberry Pi 3 avec Linux Xenomai
- un bus EtherCAT connecté au Raspberry avec 4 modules esclaves :
  - Module coupleur EK1100 : qui permet de connecter le port RJ45 du Raspberry aux autres modules ;
  - Module comptage EL5101 : qui permet de compter les impulsions venant du codeur incrémental ;
  - Module Convertisseur Numérique- Analogique EL413x : (avec 2 ou 4 sorties) qui permet de convertir une commande numérique en un signal analogique pour commander le moteur (16 bits, +/- 10 V)
  - Module Convertisseur Analogique-Numérique EL310x : (avec 2 ou 4 entrées) qui permet d'acquérir des signaux analogiques (différentiels, 16 bits, +/-10 V) pour les convertir en signaux numériques (la sortie du capteur magnétique est connectée sur ce module)
- d'une alimentation 24 V DC ;
- un clavier, une souris et un câble HDMI pour se connecter à un écran.

La figure 2.1 est un schéma de principe du câblage de la maquette.




| Codeur HEDL | EL5101 |
|---|---|
| 1 (NC) | x |
| 2 (Vcc) | x |
| 3 (GND) | 3' |
| 4 (NC) | x |
| 5 (A\) | 5 |
| 6 (A) | 1 |
| 7 (B\) | 6 |
| 8 (B) | 2 |
| 9 (I\) | 7 |
| 10 (I) | 3 |

FIGURE 2.2 – Schéma de câblage de la maquette.

Sur la face avant du boitier de commande (ou armoire de contrôle - qui sert également de support au mécanisme), il y a deux boutons : le bouton à gauche permet de mettre le système sous tension et le bouton plus au centre permet d'activer ou non le variateur (cette permet d'activer la prise en compte la commande).

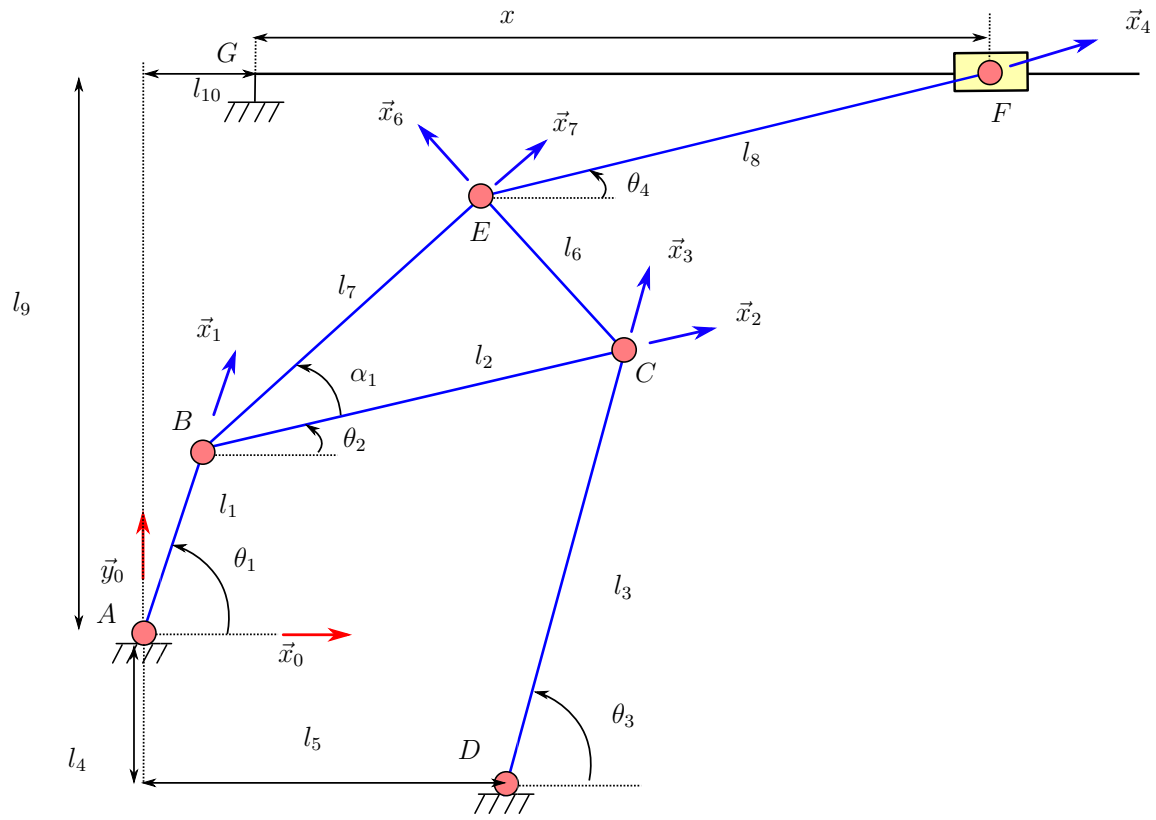



FIGURE 2.3 – Schéma cinématique du mécanisme et paramétrage.

## 2.2 Paramétrage et modélisation

Pour modéliser le mécanisme vous devez adopter le paramétrage de la figure 2.2. Nous avons choisi le repère de base $\mathcal{R}_0 = \{A; \vec{x}_0, \vec{y}_0, \vec{z}_0\}$, associé à l'articulation active $A$. Les coordonnées articulaires sont repérées par les angles $\theta_i = (\vec{x}_0, \vec{x}_i)$. L'articulation $A$ est la seule articulation active du mécanisme. La seule mesure angulaire disponible est l'angle $\theta_1$. La position de l'organe terminal $F$ est mesurée à l'aide d'un capteur linéaire magnétique, qui mesure la distance $x = \overrightarrow{GF} \cdot \vec{x}_0$.

Les paramètres géométriques de la structure sont définis de la manière suivante : $\alpha_1 = (\vec{x}_2, \vec{x}_7)$, $\overrightarrow{AB} = l_1\vec{x}_1$, $\overrightarrow{BE} = l_7\vec{x}_7$, $\overrightarrow{BC} = l_2\vec{x}_2$, $\overrightarrow{DA} = l_3\vec{x}_3$, $\overrightarrow{CE} = l_6\vec{x}_6$, $\overrightarrow{EF} = l_8\vec{x}_4$, $\overrightarrow{AG} = l_9\vec{y}_0 + l_{10}\vec{x}_0$ et $\overrightarrow{AD} = l_5\vec{x}_0 - l_4\vec{y}_0$. Les données numériques sont les suivantes : $l_1 = 30\ mm$, $l_2 = 90\ mm$, $l_3 = 90\ mm$, $l_4 = 56\ mm$, $l_5 = 69\ mm$, $l_6 = 90\ mm$, $l_7 = 90\ mm$, $l_8 = 120\ mm$, $l_9 = 140\ mm$, $l_{10} = 68\ mm$ et $\alpha_1 = 60°$. (dimensions à vérifier)

## 2.3 Questions

1. Établir le modèle géométrique direct permettant de relier l'entrée du système $\theta_1$ et la position de l'organe terminal $F$ dans le repère $\mathcal{R}_0$ ;
2. Proposer une méthode numérique qui permette de calculer facilement les positions articulaires $\theta_i$, $i \in [1, 4]$ à partir de la mesure de l'organe terminal $x$ (Attention la mesure $x$ est référencée par rapport à $G$) ;
3. Valider ces modèles à l'aide d'un outil de simulation (Matlab par exemple) ;

4. Implémenter le modèle géométrique direct en C afin de vérifier que la position mesurée est égale à la position calculée. Quelles conclusions en tirez-vous ?
5. Pour étalonner le système robotique, vous aurez également besoin du MGI. Sans implémenter directement la méthode en C, proposez un moyen pour initialiser la position articulaire initiale. Pour cela il faudra attendre le programme en C final TPEtherCAT.

# Chapitre 3

# Programmation temps-réel

Un thread est une séquence d'exécution dans un programme. Il permet au programme d'effectuer plusieurs opérations ou tâches en même temps (ou du moins virtuellement). Dans cette section, nous présentons quelques fonctions permettant de programmer des thread temps-réel. Il est donc recommandé d'aller sur le site de Xenomai pour avoir une vision plus précise des possibilités offertes par l'API Xenomai (voici l'URL : `https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__alchemy__task.html` il s'agit de la version 3.0.8 qui diffère légèrement de la version installée sur les Raspberry Pi, qui est la version 3.0.3).

**ATTENTION** des instabilités du système d'exploitation patché temps-réel ont été constaté. Il faut impérativement sauvegarder régulièrement vos données et programmes.

## 3.1 Gestion d'une tâche temps-réel

La création d'une tâche ou d'un thread temps-réel sous Xenomai nécessite trois éléments :
— la priorité de 1 à 99 (99 étant la priorité la plus élevée) ;
— un nom qui l'identifie ;
— un pointeur sur une fonction associée au thread.

Ensuite il faut créer puis exécuter le thread depuis le processus principal en utilisant les fonctions de la **skin** *native*. Voici la démarche à suivre :

1. Définir le processus principal :

```
int main(int argc, char *argv[]);
```

2. Déclarer la variable permettant d'identifier le thread :

```
RT_TASK main_task;
```

3. Créer la tâche temps-réel :

```
int rt_task_create(
        RT_TASK* task, /*Adresse du descripteur de la tâche*/
        const char* name, /*Nom de la tâche en caractères ASCII*/
        int stksize, /*Taille allouée pour la pile de la tâche*/
        int prio,   /*Priorité de base de la nouvelle tâche(1 à 99)*/
        int mode /* Mode associé à la tâche au moment de sa création.*/
)
```

4. Exécuter la tâche temps-réel :

```
int rt_task_start(
        RT_TASK* task, /*Adresse du descripteur de la tâche*/
```

```
        void(*)(void *cookie) entry, /*Pointeur sur une fonction*/
        void* cookie /*arguments de la fonction*/
)
```

5. Détruire la tâche temps-réel :

```
int rt_task_delete(
        RT_TASK* task, /*Adresse du descripteur de la tâche*/
)
```

6. Attendre de la fin de l'exécution de la tâche temps réel est réalisée grâce à :

```
int rt_task_join(
        RT_TASK* task, /*Adresse du descripteur de la tâche*/
)
```

   Attention cette fonction doit être utilisée avec précaution car elle bascule automatiquement le système en mode non temps-réel jusqu'à ce que la tâche soit terminée. Il faut également que la tâche soit créée avec le mode **T_JOINABLE**.

**Exercice 1** Programmer une tâche temps-réel dont l'objectif est d'incrémenter une variable locale jusqu'à 100.
Pour réaliser ce programme vous devez créer l'arborescence suivante :

```
workspace/
└─ TPExo1/
   ├─ src/
   │  └─ main.c
   ├─ build/
   └─ CMakeLists.txt
```

Le fichier source **main.c** est le suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <alchemy/task.h>
#define TASK_PRIO 99
#define TASK_MODE 0
#define TASK_STKSZ 0

RT_TASK tA;

void compute_task_func (void *arg) {
        int i;
        for (i=0;i< 100 ;i++)
                printf("%d\n",i);
}

int main (int argc, char *argv[]) {
        int ret;
        mlockall(MCL_CURRENT|MCL_FUTURE);
```

```
        /*Création de la tâche*/
        ret = rt_task_create(&tA, "computeTask", TASK_STKSZ, TASK_PRIO, TASK_MODE);
        if( ret )
                perror("Impossible de créer la tâche ");

        /*Démarrage de la tâche*/
        ret = rt_task_start(&tA, &compute_task_func, NULL);
        if (ret) {
                perror("Démarrage de la tâche ");
                rt_task_delete(&tA);
                exit(1);
        }
        printf("Appuyer sur une touche pour terminer ....\n");
        getchar();
        /*Destruction de la tâche*/
        rt_task_delete(&tA);
        return 0;
}
```

Ensuite pour compiler ce programme, voici le fichier **CMakeLists.txt**

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(INSAProject)
set(XENO_DIR /usr/xenomai)
set(XENO_INCLUDE_DIR ${XENO_DIR}/include)
set(XENO_BIN_DIR ${XENO_DIR}/bin)
set(XENO_SKIN native)
execute_process(COMMAND ${XENO_BIN_DIR}/xeno-config --skin=${XENO_SKIN}
         --cflags OUTPUT_VARIABLE XENO_CFLAGS OUTPUT_STRIP_TRAILING_WHITESPACE)
execute_process(COMMAND ${XENO_BIN_DIR}/xeno-config --skin=${XENO_SKIN}
        --ldflags OUTPUT_VARIABLE XENO_LDFLAGS OUTPUT_STRIP_TRAILING_WHITESPACE)
include_directories(${XENO_INCLUDE_DIR})
file(GLOB_RECURSE
        src_files
        src/*)
set(EXTRA_LIBS ${XENO_LDFLAGS})
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${XENO_CFLAGS}")
add_executable(my_exe ${src_files})
target_link_libraries(my_exe ${EXTRA_LIBS})
```

Pour compiler et exécuter le programme suivre les instructions suivantes :

1. dans le répertoire **build**, créer la chaîne de compilation :

   ```
   cmake ..
   ```

2. dans le répertoire **build**, compiler le code source :

   ```
   make
   ```

3. dans le répertoire **build**, exécuter le programme en mode administrateur

   ```
   sudo ./my_exe
   ```

## 3.2 Gestion d'une tâche temps-réel périodique

Lorsque l'on souhaite avoir une tâche qui s'exécute périodiquement à une fréquence donnée (comme c'est le cas pour une boucle d'asservissement), il est nécessaire de prendre certaines précautions. Si on créée une tâche temps-réel et que la fonction qui lui est associée boucle à l'infini sans restituer la main au système d'exploitation Linux, alors on se retrouve dans une situation de blocage. Il est donc important de spécifier à la tâche en question que nous souhaitons la basculer dans un mode particulier pour lui permettre d'être périodique tout en assurant l'intégrité du système. Pour cela il existe plusieurs mécanismes dans Xenomai, en particulier les fonctions disponibles dans *Timer management services* de la skin native, pour gérer le temps. Pour utiliser ce service, vous devez inclure la bibliothèque timer de la manière suivante :

```
#include <alchemy/timer.h>
```

Ensuite pour créer une tâche périodique, il suffit de créer une tâche temps-réel, puis d'indiquer que le corps de cette tâche sera exécuter pendant un instant déterminer (ou à l'infini) mais à des instants précis.

1. Déclarer et programmer une tâche temps-réel comme précédemment ;
2. Définir la période d'activation de la séquence d'exécution de votre tâche. Généralement, il est recommandé d'utiliser une variable de type **long long** pour définir la période qui est utilisée en $ns$ ;
3. Créer l'horloge qui va déclencher la périodicité de votre exécution ;

   ```
   int rt_task_set_periodic (
           RT_TASK *task, /*Identifiant de la tâche à basculer en périodique*/
           RTIME idate,  /*Instant qui va déterminer le démarrage réel*/
           RTIME period /*Période en clock ticks*/
   )
   ```

4. Attendre la prochaine période, ce qui permet de restituer la main à Linux pour que d'autre tâches puissent être exécuter ;

   ```
   int rt_task_wait_period (
           unsigned long *overruns_r
   )
   ```

**Exercice 2** Le second exercice vise à mettre en œuvre une tâche temps-réel périodique. L'objectif est d'incrémenter une variable toute les 100 ms.

Créer l'arborescence suivante :

```
workspace/
└── TPExo2/
    ├── src/
    │   └── main.c
    ├── build/
    └── CMakeLists.txt
```

Programmer le code source **main.c** suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <alchemy/task.h>
#include <alchemy/timer.h>
#define TASK_PRIO 99
#define TASK_MODE 0
#define TASK_STKSZ 0

RT_TASK periodic_task;

/*periode de la tâche en ns (ici 100 ms)*/
long long period_ns = 100*1000*1000LL;
/*fonction appelée périodiquement associée à la tâche "periodic task"*/
void periodic_task_func (void *arg) {
        int i;
        RTIME start_ns; /* Instant de démarrage de la tâche périodique*/
        /*Lecture du timer courant et ajout 1 ms*/
        start_ns = rt_timer_read()+1000000;
        /*Bascule la tâche courante en tâche périodique de période spécifiée*/
        rt_task_set_periodic(NULL, start_ns, period_ns);
        while(1)
        {
                /*Attente du prochaine top horloge
                cadencé à 100 ms (restitue la main à Linux)*/
                rt_task_wait_period(NULL);
                i++;
                printf("%d\n",i);
        }
}

int main (int argc, char *argv[]) {
        int ret;
        mlockall(MCL_CURRENT|MCL_FUTURE);
        ret = rt_task_create(&tA, "computeTask", TASK_STKSZ, TASK_PRIO, TASK_MODE);
        if( ret )
                perror("Impossible de créer la tâche ");
        ret = rt_task_start(&tA, &periodic_task_func, NULL);
        if (ret) {
                perror("Démarrage de la tâche ");
                rt_task_delete(&tA);
                exit(1);
        }
        getchar();
        rt_task_delete(&tA);
        return 0;
}
```

Le fichier **CMakeLists.txt** est identique au précédent. Pour compiler et exécuter le programme suivre les instructions suivantes :

1. dans le répertoire **build**, créer la chaîne de compilation :

   ```
   cmake ..
   ```

2. dans le répertoire **build**, compiler le code source :

   ```
   make
   ```

3. dans le répertoire **build**, exécuter le programme en mode administrateur

   ```
   sudo ./my_exe
   ```

***Question :*** Que pouvez vous dire de la stabilité de la période d'échantillonnage. En vous aidant de la fonction **rt_timer_read()** disponible dans l'API de Xenomai, proposer une adaptation du programme précédent permettant de connaître à chaque pas la période de la tâche temps-réel et le différentiel entre chaque période.

## 3.3 Gestion du protocol EtherCAT

EtherCAT (Ethernet for Control Automation Technology) est un bus de terrain temps-réel développé par Beckhoff Automation depuis 2003. Un bus de terrain est un réseau industriel qui permet de mettre en connexion différents équipements localement sur un système électromécanique ou à distance sur l'ensemble d'un site industriel. Le bus EtherCAT se présente sous la forme d'un réseau de type maître-esclave qui se basent sur les trames Ethernet pour communiquer avec le matériel. Contrairement aux protocoles TCP/IP ou UDP, le protocole EtherCAT n'utilise qu'une partie de la trame Ethernet, ce qui permet d'avoir des taux de transmissions bien plus important et donc aboutir à une solution compatible avec les contraintes temps-réel. Les trames Ethernet ne sont plus reçues, interprétées et stockées au niveau de chaque équipement. Les esclaves EtherCAT lisent et écrivent les données durant le passage de la trame à l'intérieur du nœud. De plus, ils ne lisent que les données qui leur sont spécifiquement adressées. Les trames subissent alors un délais de quelques nanosecondes. Sans entrer dans le détail du protocole EtherCAT, il est important de savoir que le protocole EtherCAT est encapsulé dans les trames Ethernet grâce à un champ Ethertype spécifiques.

Les caractéristiques principales de ce protocole sont :

— disponible publiquement ;
— normalisé CEI 61158 ;
— maintenu par le consortium EtherCAT Technology Group depuis 2006 ;
— compatible avec le temps-réel dur et mou, garantissant des temps de cycles de $\approx 10\ \mu s$ avec un jitter inférieur à $1\mu s$ ;
— utilisation avec des câbles et des connecteurs standard Ethernet ;
— assure la synchronisation grâce au mécanisme de *Distributed Clock (DC)* ;

Attention, EtherCAT n'est pas une communication TCP/IP standard. Pour utiliser le communication TCP/IP sur un bus EtherCAT il faut utiliser EoE (Ethernet over EtherCAT qui sont des modules addtionnels). Bien que le protocole EtherCAT utilise le matériel standard des cartes réseaux Ethernet, il est nécessaire de changer le driver ou pilote du matériel pour pouvoir bénéficier de la communication temps-réel et ainsi avoir le système maître qui gère l'ensemble des esclaves sur le bus. Il existe plusieurs solutions commerciales permettant de prendre en charge le bus EtherCAT (pile ACONTIS, suite logicielle TWINCAT, etc.). Il existe également des solutions gratuites open-sources, parmi lesquelles la suite logicielle EtherLab ou la pile SOEM (intégrée dans OROCOS)

**Utilisation de EtherCAT Master Igh** Pour utiliser le protocole EtherCAT, nous utiliserons la pile EtherCAT disponible dans la suite logiciel EtherLab (`https://etherlab.org/en/what.php`). Cette pile EtherCAT est open-source et compatible avec Linux Xenomai. EtherCAT Master est fournie avec une API qui contient des fonctions de base qui permettent d'ouvrir et fermer la communication avec les modules esclaves du bus EtherCAT, d'écrire et de lire les données pour les modules esclave. La pile EtherCAT Master est également fournie avec des programmes ou scripts qui permettent de connaîtr el'état du réseau, la topologie du réseau, identifier les modules esclaves présent sur le réseau, configurer les modules. **ATTENTION** la pile EtherCAT Master Igh n'a pas été développé spécifiquement pour le matériel du Raspberry Pi 3. En effet, le port Ethernet natif sur la carte n'est qu'une émulation d'un port Ethernet sur un hub USB, ce qui a pour conséquences de rendre la communication un peu capricieuse. Sauvegarder régulièrement votre travail.

**Préambule** Configuration du port Ethernet pour la pile EtherCAT

La pile EtherCAT Master est pré-installée sur votre Raspberry Pi. Cependant le protocole EtherCAT utilise le port Ethernet de votre Raspberry, et comme la "carte réseau" à une adresse MAC différente pour chaque ordinateur, il faut s'assurer que la pile EtherCAT Master est bien configurée pour votre système. Pour cela, il suffit de suivre les instructions suivantes, depuis un terminal :

1. Récupérer l'adresse MAC de votre carte réseau (HWaddr correspondant à la ligne eth0)

   ```
   sudo ifconfig
   ```

2. Ouvrir le fichier de configuration de la pile EtherCAT en administrateur

   ```
   sudo emacs /etc/sysconfig/
   ```

   Sur la ligne **MASTER0_DEVICE=""** vous devez avoir l'adresse MAC de votre carte réseau, si ce n'est pas le cas, fêtes les modifications.
3. Vérifier dans le fichier que **DEVICE_MODULES="generic"**
4. Enregistrer le fichier

**Exercice 1** Pour vous familiariser avec l'API fournie avec la pile EtherCAT Master et pour découvrir la puissance du protocole de communication EtherCAT, nous vous proposons de réaliser les instructions suivantes (tout se passe dans un terminal) :

1. **Chargement du module Ethercat** pour remplacer le module noyau de Linux qui gère la carte réseau Ethernet par un module spécifique EtherCAT, ce qui revient à définir le **master**.

   ```
   sudo /etc/init.d/ethercat start
   ```

2. Vérifier que les modules **ec_master** (module associé au **master** de la pile EtherCAT) et **ec_generic** (module associé à la gestion de la carte réseau pour la rendre compatible avec EtherCAT) sont bien chargés, grâce à la commande :

   ```
   lsmod
   ```

3. Vérification de l'état du **master** :

   ```
   sudo /etc/init.d/ethercat status
   ```

4. Arrêt du **master**. Cette action décharge les modules **ec_master** et **ec_generic** :

   ```
   sudo /etc/init.d/ethercat stop
   ```

Il est également possible de redémarrer le **master** de la pile EtherCAT à l'aide de la fonction :

```
sudo /etc/init.d/ethercat restart
```

**Exercice 3** Vérifier la topologie du réseau.

Pour réaliser cette étape vous avez besoin de charger la pile EtherCAT et vous devez utiliser le programme suivant :

```
sudo /opt/etherlab/bin/ethercat <COMMANDE> [OPTIONS] [ARGUMENTS]
```

Sans option, le programme affiche une aide des options disponibles. Utiliser les bonnes commandes et les bons arguments pour obtenir les informations sur la topologie du bus EtherCAT (regardez l'aide de la commande ethercat) :

1. afficher les modules esclaves présents sur le bus ;
2. générer les structures de données PDO(Process Data Object) pour tout les modules esclaves.

A l'issue de cette étape vous devriez voir 4 modules esclaves (pas forcement dans cet ordre) :

- EK1100 : Coupleur EtherCAT
- EL5101 : Module de comptage pour le codeur incrémental
- EL3102 ou EL3104 : Module Convertisseur Analogique Numérique 16 bits, entrées différentielles+/- 10 V (2 entrées ou 4 entrées analogiques)
- EL4132 ou EL4134 : Module Convertisseur Numérique Analogique 16 bits, 2 ou 4 sorties +/- 10 V.

Vous devriez également avoir 3 structures de données de la forme suivante (les structures de données sont associées aux trois modules esclaves) qui définissent les configurations PDO nécessaires à la communication EtherCAT.

```
/* Master 0, Slave 1, "EL4134"
 * Vendor ID:       0x00000002
 * Product code:    0x10263052
 * Revision number: 0x03fa0000
 */

ec_pdo_entry_info_t slave_1_pdo_entries[] = {
 {0x7000, 0x01, 16}, /* Analog Output */
 {0x7010, 0x01, 16}, /* Analog Output */
 {0x7020, 0x01, 16}, /* Analog Output */
 {0x7030, 0x01, 16}, /* Analog Output */
};

ec_pdo_info_t slave_1_pdos[] = {
 {0x1600, 1, slave_1_pdo_entries + 0}, /* A0 RxPDO-Map OutputsCh.1*/
 {0x1601, 1, slave_1_pdo_entries + 1}, /* A0 RxPDO-Map OutputsCh.2 */
 {0x1602, 1, slave_1_pdo_entries + 2}, /* A0 RxPDO-Map OutputsCh.3 */
 {0x1603, 1, slave_1_pdo_entries + 3}, /* A0 RxPDO-Map OutputsCh.4 */
};

ec_sync_info_t slave_1_syncs[] = {
 {0, EC_DIR_OUTPUT, 0, NULL, EC_WD_DISABLE},
 {1, EC_DIR_INPUT, 0, NULL, EC_WD_DISABLE},
 {2, EC_DIR_OUTPUT, 4, slave_1_pdos+0, EC_WD_DISABLE},
 {3, EC_DIR_INPUT, 0, NULL , EC_WD_DISABLE},
 {0xff}
};
```

*Remarques :* Si vous vous attardez un peu sur ces structures de données, beaucoup d'informations sont disponibles pour comprendre le fonctionnement du protocole EtherCAT.

— Dans la zone commentaires, on retrouve : le nom du module, l'identifiant du constructeur, le code produit ainsi que la version du firmware
— On peut voir en particulier, la première structure de données qui renseigne les quatre sorties analogiques du module (parce que c'est un module de type EL4134). Cette structure de données contient plusieurs champs comme l'index et le sub-index qui définissent l'emplacement mémoire des registres relatifs aux sorties, et la taille en octets de la valeur.
— Les autres structures sont nécessaires à la synchronisation avec les autres modules et à l'encapsulation des données dans la trame Ethernet.

**Programmation communication EtherCAT** Lorsque le module de gestion du bus EtherCAT est correctement chargé dans le noyau, il est alors possible de développer un programme qui communique directement avec les modules esclaves. La structure de base du code source pour communiquer avec les modules esclaves via le bus EtherCAT peut se résumer de la manière suivante (il s'agit d'une proposition non complète du programme pour communiquer avec les modules esclaves) :

1. inclure les fichiers d'en-tête de Linux, de Xenomai et de la pile EtherCAT
2. déclaration des variables spécifiques à la communication par le protocole EtherCAT
3. définition de la topologie du bus EtherCAT (obtenue avec le programme **ethercat slaves** et **ethercat cstruct**)
4. déclaration et définition des variables spécifiques à la tâche temps-réel périodique
5. définition de la procédure périodique associée à la tâche temps-réel principale. Cette fonction permettra de communiquer avec les modules esclaves sur le bus EtherCAT, elle sera appelée périodiquement.
6. définition du programme principal :
   — gestion de interruptions logicielles pour arrêter le programme
   — verrouillage de la mémoire pour éviter le swapping en temps-réel
   — récupération des informations du master EtherCAT
   — configuration des modules esclaves et construction de la trame Ethernet pour la communication via une structure Domain
   — activation du master. A partir de ce moment, il est possible d'utiliser le bus EtherCAT
   — récupération du pointeur sur la structure domain pour mettre en forme les données qui transiteront sur le bus
   — création et exécution de la tâche temps-réel
   — attente que l'utilisateur kill le processus par $Ctrl + C$
   — désactivation du master EtherCAT

Il s'agit d'une structure simplifiée du code de base qui permet la communication avec le bus EtherCAT. Vous trouverez dans le répertoire **workspace** une archive **TPEtherCAT.tgz** qui reprend ce squelette, pour vous aider à développer cette communication avec le bus EtherCAT. Les différentes étapes sont résumées ici, mais vous trouverez toutes les informations dans le code source. Les zones où vous devez écrire votre code sont indiquées par des commentaires **A FAIRE** (avec des informations supplémentaire) avec une numéro qui indique l'ordre dans lequel vous devez faire le travail. Pour chaque étape vérifier la compilation. Les autres zones ne doivent être sous aucun prétexte modifiées. Elles sont fonctionnelles. Vous pouvez les lire, les analyser, vous en inspirer pour réaliser les travaux que vous avez à faire

— Étape 1 : explorer le code pour retrouver les différentes parties du squelette (attention des fonctions supplémentaires ont été ajouté pour alléger le code) ;

— Étape 2 : intégrer les éléments de votre système (modules esclaves Beckhoff) dans ce code pour que le "domain" correspond à la topologie de votre bus ;
— Étape 3 : créer la tâche temps-réel qui appelle périodiquement une fonction qui interrogera le bus EtherCAT
— Étape 4 : implémenter la fonction $rtController\_doPositionTacking\_Proc$ qui est appelée lorsque le contrôleur bascule en mode $POSITION\_TRACKING\_MODE$. Nous souhaitons implémenter une commande articulaire pour que la position $q_1$ suive une trajectoire sinusoïdale (toutes les informations nécessaires et les paramètres de la sinusoïde sont donnés dans le code). Ensuite implémenter le MGD dans une nouvelle fonction pour comparer la position calculée et la position réelle.

Pour avoir accès au code source contenu dans **TPEtherCAT.tgz**, il vous suffit décompresser l'archive dans le répertoire **workspace**. Pour celles et ceux qui ne connaissent pas la commande, depuis un terminal faire :

```
cd workspace
tar xfvz TPEtherCAT.tgz
```

Après cette étape vous devriez avoir l'arborescence suivante :

```
workspace/
└─ TPEtherCAT/
   ├─ src/
   │  ├─ main.c
   │  ├─ main.h
   │  └─ CMakeLists.txt
   ├─ build/
   ├─ cmake/
   │  └─ modules/
   │     ├─ FindEtherCAT.cmake
   │     └─ FindXenomai.cmake
   └─ CMakeLists.txt
```

Le code source à modifier est dans le répertoire **src**. Le répertoire **cmake** contient les modules permettant de configurer le Makefile pour EtherCAT et Xenomai (vous ne devez pas y toucher). Les deux CMakeLists.txt sont récurrents car ce code est extrait d'un logiciel plus important. Lorsque vous allez compiler le code dans le repertoire **build**, l'exécutable sera dans le répertoire **build/src** et s'appelle **rtController**. Pour exécuter le code il suffit alors de suivre la séquence suivante (mais attention en l'état la compilation du code source n'aboutira pas !)

```
cd build
sudo /etc/init.d/ethercat start
sudo ./src/rtController
```

Pour terminer l'exécution du programme, il suffit d'invoquer la commande **Ctrl+C**, ce qui aura pour conséquence d'envoyer une interruption logicielle au programme pour le tuer *kill*. Puis faire

```
sudo /etc/init.d/ethercat stop
```