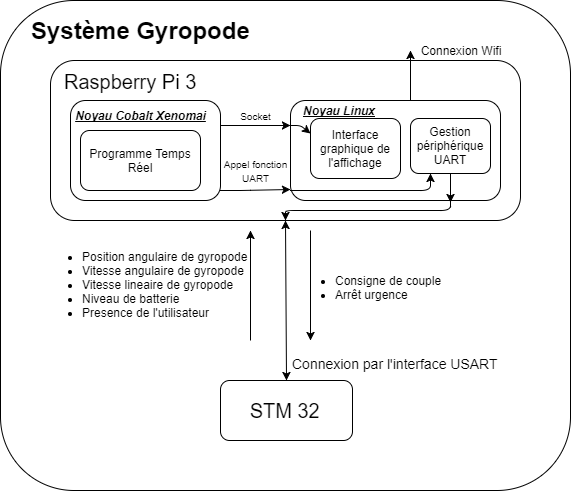
Sujet de TP: Temps réel

# 

# 1 - Partie Superviseur : Raspberry Pi

Le Raspberry Pi est un nano-ordinateur mono-carte à processeur ARM. La carte joue le rôle de superviseur et elle effectue les opérations temps réel, en interaction avec la carte STM32. Le système temps réel de Raspberry Pi 3 est réalisé dans Xenomai.

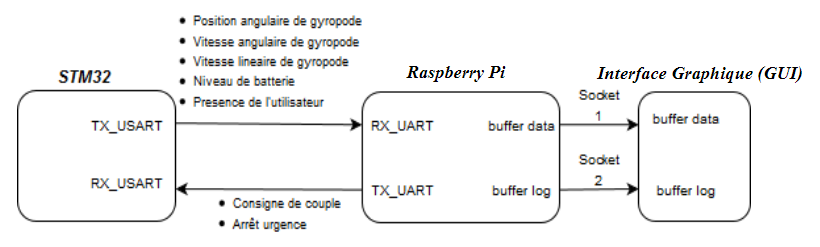
Dans la carte Raspberry Pi, nous avons deux types d’opérations. La première sur le système temps réel de gyropode, et le deuxième sur des tâches de système Linux. Avec sa caractéristique de Co-noyau, nous pouvons effectuer ces deux opérations différentes sans problème.



*Figure 1 : Diagramme de contexte*

## 1.1 - Présentation du superviseur

Le superviseur est le programme de temps réel qui tourne dans le noyau de cobalt. Dans ce programme, nous avons aussi appelé des fonctions non temps réel (côté linux) pour permettre l'affichage de données et également la communication avec la partie de commande des moteurs (STM32).

*Figure 2 : Diagramme de contexte avec Entrée-Sortie*

• Partie Temps Réel

La fonction principale de ce programme est de faire l’asservissement du gyropode ainsi que le traitement de données et la surveillance du fonctionnement du système. Toutes ces opérations sont inscrites dans l’historique des événements (log) mis à disposition pour le programmeur.

• Partie Linux

La carte est connectée avec l’ordinateur, qui sert de moniteur. Il y a une interface graphique qui affiche des données du système et des historiques des événements en temps réel. Ce programme est exécuté dans le noyau Linux et reçoit les données du programme de temps réel grâce à deux sockets : un pour la réception de données et l'autre pour recevoir les chaînes de caractères du log à afficher. En plus de cette interface, étant donné que la communication entre la carte Raspberry Pi 3 et la carte STM32 se fait par le protocole série UART (non synchrone), qui n’est pas encore dans la librairie de Xenomai, cette communication est gérée par le noyau Linux.

## 1.2 - La communication via l’interface UART

Dans la carte Raspberry Pi 3, il n’y a pas d’interface de liaison série USART. Il est donc obligé d’utiliser le protocole UART, qui est asynchrone.

### 

### 1.2.1 - Envoi de données de Raspberry Pi sur liaison série

Dans notre programme, l’envoi de données (les consignes de courant et le message d’arrêt d’urgence) est effectué par le **thread Envoyer**. Le **thread Asservissement** va envoyer toutes les 20 ms des données à la file de message et le thread Envoyer va envoyer ces messages avec le protocole FIFO. Il y a d’autres threads de surveillance qui peuvent envoyer le message d’arrêt d’urgence à la file d’attente, comme le **thread Arrêt Urgence**.

### 1.2.2 - Variables échangées sur liaison série

Voici les informations envoyées par le STM32 à la Raspberry PI:

|  |  |  |  |
| --- | --- | --- | --- |
| Donnée | Type | Unité | Label |
| position angulaire | float | rad | ’p’ |
| vitesse angulaire | float | rad /s | ’s’ |
| niveau batterie | integer | % | ’b’ |
| vitesse linéaire | float | m/s | ’v’ |
| présence utilisateur | integer | 1 si présent, 0 sinon | ’u’ |

Voici les informations envoyées par la Raspberry Pi au STM32 :

|  |  |  |  |
| --- | --- | --- | --- |
| Donnée | Type | Unité | Label |
| Consigne de couple | float | N.m | ’c’ |
| Arrêt | int | 1 si arrêt d’urgence, 0 sinon | ’a’ |

## 

## 

## 

## 

## 

## 

## 

## 1.3 - L’interface graphique sur Raspberry Pi

L’interface graphique qui a été conçue sur la carte Raspberry PI a deux objectifs :

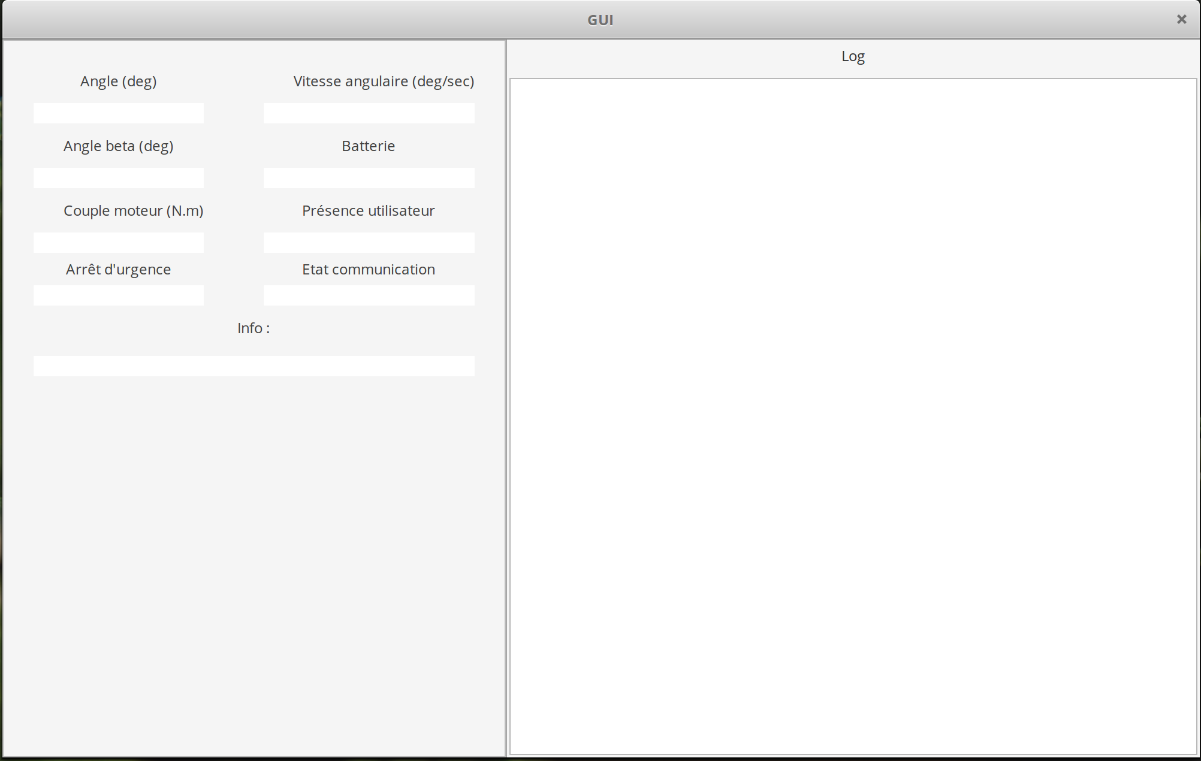
• Permettre à l’utilisateur du gyropode de consulter certaines informations sur le système, comme sa vitesse ou le niveau de batterie. Les informations sont disponibles avec leur valeur numérique mais aussi grâce à des messages d’information qui préviennent l’utilisateur de certains dangers (batterie faible, vitesse trop élevée...)

• Permettre au programmeur du code de temps réel de vérifier l’état et le comportement du système pendant l’exécution de son programme, suite à des modifications de certaines variables. De plus, l’interface graphique affiche également le log du programme de temps réel, qui contient des informations sur l’exécution des tâches, ainsi que la manipulation des mutex et sémaphores.

L’exécution de programme se fait dans un terminal de Linux, qui n’est pas une tâche de temps réel. Cela a pour but de ne pas consommer de ressources temps réel et de laisser au programmeur des libertés quant à l’architecture de son programme.

En effet, l’interface graphique est exécutée par le noyau Linux, en même temps que le programme de temps réel. En revanche, toutes les tâches du noyau Xenomai sont prioritaires, ce qui implique que les instructions du programme de l’interface graphique ne s’exécutent que lorsque le CPU n’est pas occupé par les tâches temps réel.

La transmission de données du processus temps réel à non temps réel est faite grâce à l’utilisation d’un socket. Les fonctions suivantes sont disponibles : *add\_info* et *send\_to\_socket*, elles permettent au programmeur d’envoyer les données qu’il souhaite au socket (et donc de les afficher sur l’interface graphique) à partir de n’importe quelle tâche de son programme de temps réel.



*Figure 3 : Capture d'écran de GUI*

## 

## 1.4 - Présentation du log du programme de supervision

Pour mieux comprendre l'exécution des tâches dans le programme de supervision, une librairie *lib\_monitor* permet d'écrire des informations d'exécution dans le document *log.xenolog*. Il s’agit d'une tâche de priorité faible qui tourne en fond, en même temps que le programme de supervision. Une file de message permet de stocker les évènements afin de les traiter lorsque le programme de temps réel ne consomme pas de ressources du processeur. Pour écrire des évènements dans le fichier de log, il faut utiliser des fonctions de la librairie dédiée. Voici des fonctions à utiliser dans le programme de temps réel avec ses descriptions:

|  |  |
| --- | --- |
| Noms de fonctions | Description de fonction |
| int init\_recording(); | Initialisation de l'enregistrement des évènements par le log (à lancer au début du programme principal) |
| int stop\_recording(); | Arrêt de l'enregistrement des événements par le log (à lancer à la fin du programme principal) |
| void log\_wait\_for\_mutex(RT\_MUTEX \* mut); | Ecrire le log lorsque la tâche attend un mutex, cette fonction doit être écrite avant la fonction de *rt\_mutex\_acquire()* |
| void log\_mutex\_acquired(RT\_MUTEX \* mut); | Ecrire le log lorsque la tâche acquiert un mutex, cette fonction doit être écrite après la fonction de *rt\_mutex\_acquire*() |
| void log\_mutex\_released(RT\_MUTEX \* mut); | Ecrire le log lorsque la tâche relâche un mutex, cette fonction doit être écrite après la fonction de *rt\_mutex\_release()* |
| void log\_sem\_waiting(RT\_SEM \* sem); | Ecrire le log lorsque la tâche attend un sémaphore, cette fonction doit être écrite avant la fonction de *rt\_sem\_p()* |
| void log\_sem\_entered(RT\_SEM \* sem); | Ecrire le log lorsque la tâche attend un sémaphore, cette fonction doit être écrite après la fonction de *rt\_sem\_p()* |
| void log\_sem\_signaled(RT\_SEM \* sem); | Ecrire le log lorsque la tâche signale un sémaphore, cette fonction doit être écrite avant la fonction de *rt\_sem\_v()* |
| void log\_task\_entered(); | Ecrire le log lorsque la tâche commence son exécution, cette fonction doit être écrite au début de la fonction |
| void log\_task\_new\_iteration(); | Ecrire le log lorsque la tâche commence une nouvelle itération, cette fonction doit être écrite au début de la boucle de fonction |
| void log\_task\_ended(); | Ecrire le log lorsque la tâche se termine, cette fonction doit être écrite à la fin de la fonction |
| void log\_task\_deleted(RT\_TASK \* task); | Ecrire le log lorsqu'une tâche est supprimée, cette fonction doit être écrite à la fin de la fonction de *deleteTasks()* du programme principal |

# 

# 2 - Simulateur du système gyropode

Le simulateur du système gyropode est composé de la même carte de STM32 que celle utilisée dans la maquette réelle, et toutes les interruptions et fréquences des tâches sont similaires au code de la maquette réelle afin de retranscrire au mieux le comportement du gyropode réel.

Le code dédié à la communication des données avec la carte Raspberry PI est également identique au code de la maquette réelle.

Seule deux parties de simulation ont été rajoutées. Une partie est dédiée à la simulation physique du gyropode, tandis que l’autre sert à simuler quelques paramètres tels que le niveau de batterie, ou la présence de l’utilisateur.

### 2.1 - Simulation physique du gyropode

La simulation du comportement physique du gyropode est effectuée grâce au modèle du système qui nous a été fourni. En effet, il s’agit du modèle physique linéarisé du gyropode qui n’est valable qu’entre une inclinaison de -20° à +20°. Mais cela est suffisant pour simuler le comportement du système.

Il serait intéressant de connaître la vitesse linéaire, cependant, pour la calculer il nous faut la tension fourni aux moteurs, qui n’est pas disponible dans le simulateur. C’est pourquoi dans le simulateur, il n’y a pas de variable de vitesse linéaire contrairement à la vraie maquette

### 2.2 - Simulation des variables du système

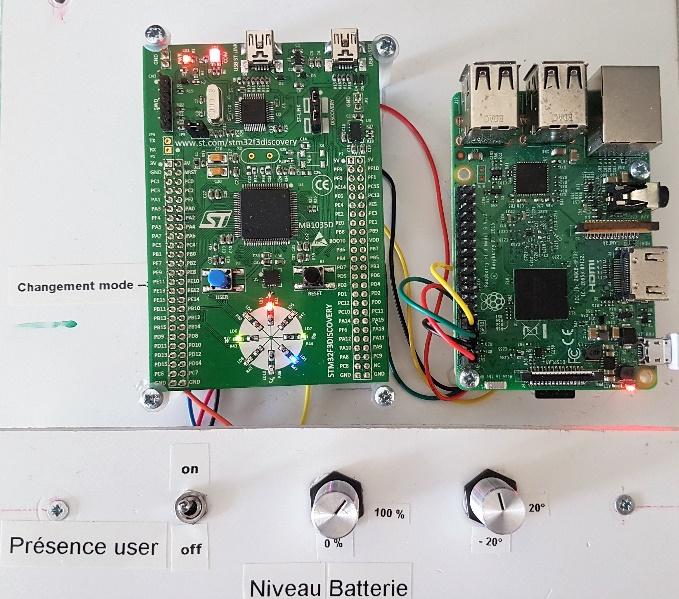
Dans le but de tester le code de temps réel, il peut être utile de modifier des variables du système comme le niveau de batterie ou la présence de l’utilisateur. Vous pouvez le faire de deux façons présentées ci-dessous:

La carte du simulateur est connectée à deux potentiomètres et un bouton switch. La modification des variables se fait directement grâce à ces périphériques. Les configurations sont alors identiques à la maquette réelle. Les variations des variables sont instantanées et peuvent être continues, ce qui se rapproche plus de la réalité.

* 1. Potentiomètres :

Toutes les lectures des valeurs de potentiomètre sont effectuées par l’ADC du STM32.

* + 1. Un potentiomètre sert à simuler le niveau de batterie
    2. Un autre potentiomètre sert à simuler la valeur de l'angle utilisateur () entre -20 degrés et +20 degrés.
  1. Boutons :
     1. Le switch sert à simuler le bouton vérifiant la présence de l’utilisateur, présent sur le guidon de la maquette.
     2. Le bouton bleu sur la carte de simulateur (STM32) sert à changer le mode de modification de l'angle utilisateur (), soit ce dernier est indiqué par le potentiomètre, soit il est simulé par des capteurs intégrés dans la carte STM32 (fusion d’angle de gyroscope et accéléromètre).



*Figure 4 : Photo de Simulateur*

Pour faciliter la compréhension des opérations dans le STM32, les LEDs servent à indiquer des états de la carte.

|  |  |
| --- | --- |
| Numéro de LED | Action |
| LED 3 | Clignote à la réception de la trame de consigne |
| LED 4 | Mode d’entrée de l’angle relatif entre l’utilisateur et le guidon  => ON : Mode accéléromètre => OFF: Mode potentiomètre |
| LED 6 | Etat de présence de l’utilisateur  => ON : Présence => OFF: Absence |
| LED 8 | Etat d’arrêt d’urgence  => ON : arrêt déclenché |
| LED 5 | Clignote à la réception de données par la Raspberry via USART |
| LED 7 | Clignote quand il n’y a pas de connexion USART avec la Raspberry |
| LED 9 | Clignote à l’envoi de données via USART à la Raspberry |

# 

# 3 - Consigne du TP:

* Aller sur le lien suivant: [https://github.com/leolaudouard/TP-Temps-reel](https://l.facebook.com/l.php?u=https%3A%2F%2Fgithub.com%2Fleolaudouard%2FTP-Temps-reel&h=ATNdgYmvG8TUJ7kCYDhjAALDi1fqk_s0D8M92dbBCI7q2yrpe4UetRnNVpOmHEFpcJIg4JDdXSAwvhIrEWK122QxKzaq_YjRB3OjGlYN_CbbLw)
* Récupérez le dossier et enregistrez le sur votre bureau (le chemin le plus court sera la mieux).
* Dans le dossier, Code → Temps\_reel lisez le “readme” afin de comprendre l’organisation du code.

Les fichiers “fonction.cpp”, “fonction.h” et “main.c” doivent être complétés. Vous n’avez pas besoin de modifier les autres fichiers mais vous pouvez les consulter.

Comme la communication entre le programme temps réel et l’interface graphique se fait à travers des sockets, il vous faut renseigner l’adresse IP pour l’initialisation des sockets.

Les étapes sont les suivantes :

Pour lancer l’interface graphique et lancer le main :

1. Dans un 1er terminal :

* exécuter la commande : ssh -X pi@*adresseIP Raspberry (écrite sur le simulateur)*
* aller dans le dossier GUI : cd Desktop/Temps\_Reel\_Trou/GUI
* lancer l’interface GUI : ./GUI

1. Dans un 2ème terminal :

* exécuter la même commande : ssh -X pi@*adresseIP Raspberry*
* aller dans le dossier TR : cd Desktop/Temps\_Reel\_Trou/TR
* ouvrir le fichier includes.h : nano includes.h
* changer l’adresse IP au début du code par *localhost*
* quitter le fichier : ctrl+x puis faites Entrée pour sauvegarder
* lancer le main : sudo ./main
* Penser à bien nommer les éléments du code : les mutex sont notés *var\_mutex\_xxxx* et les sémaphores *var\_sem\_xxxx,* les tâches ont souvent des noms qui décrivent leur fonction principale (voir “global.cpp”).
* La programmation orientée objet a été utilisée pour certaines variables partagées, comme c'est le cas pour *consigne\_couple*. Les diagrammes de classes ainsi que la documentation des librairies disponibles sont en annexes.
* Le fichier de log peut être utilisé pour vérifier le respect de certaines contraintes de temps fixées par les exigences, il est disponible dans le dossier principal du programme de temps réel et est nommé *log.xenolog.* Les exigences fonctionnelles peuvent être testées grâce à l'interface d'affichage.

# 4 - Gestion des anomalies de fonctionnement

#### 

#### 4.1 - Rupture de communication

Lors d’une rupture de communication entre le STM32 et la Raspberry PI, le système doit s’arrêter, car les nouvelles valeurs de consigne de couple envoyées par la Raspberry n’arrivent plus jusqu’au STM32. Il faut alors éviter que le microcontrôleur continue d’appliquer les anciennes consignes de courant aux moteurs, ce serait dangereux pour l’utilisateur. La détection de l’anomalie doit être réalisée lors de la réception de donnée car c’est la seule façon d’observer une rupture de communication.

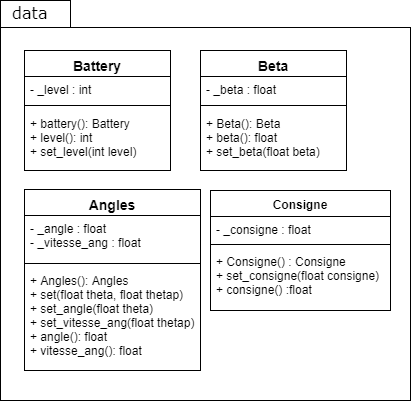
Pour éviter cela, un système de Watchdog est en place. Un compteur est incrémenté continuellement dans la boucle de 50 Hz. Lorsqu’un message est reçu, il est remis à zéro. Ainsi, si aucun message n’est reçu pendant 500 ms, les moteurs vont s’arrêter.

#### 4.2 - Réinitialisation de l’état d’arrêt d'urgence

Lorsque l’arrêt d'urgence est déclenché par la Raspberry Pi, le système s’arrête. Pour pouvoir redémarrer le système, la carte STM32 doit être capable de réinitialiser l’état de la variable d’arrêt d'urgence. L’arrêt d'urgence du système peut être déclenché par la Raspberry Pi pour des raisons de sécurité (absence de l’utilisateur ou niveau de batterie trop faible). La Raspberry Pi envoie sans arrêt des trames d’arrêt d’urgence si une des deux conditions n’est pas remplie. Nous avons un compteur dans la partie réception de données pour vérifier si le STM32 reçoit toujours des trames d’arrêt d’urgence. Dès que le STM32 reçoit une trame d’arrêt d’urgence, le compteur est remis à zéro. Si le STM32 reçoit des trames différentes de l’arrêt, le compteur est incrémenté. Si le compteur atteint 300 (environ 3 secondes si la fréquence est 100 Hz dans l’envoi de données de la Raspberry Pi), il considère que les deux conditions de sécurité sont remplies et réinitialise l’état d'arrêt d'urgence.

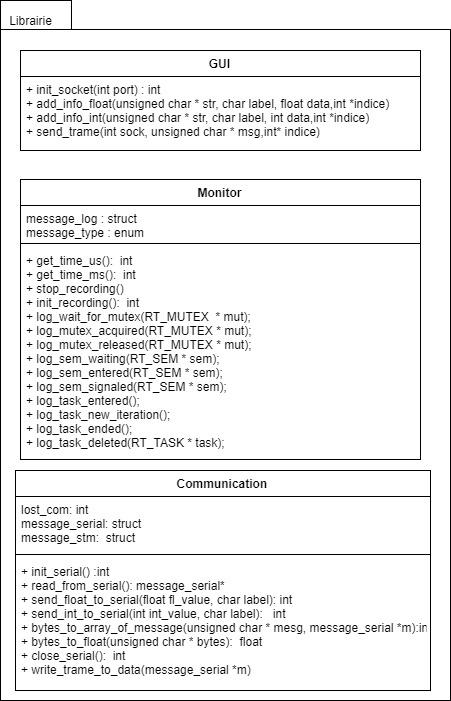
# 

# ANNEXES:



*Annexe 1 : Diagramme de classes des données et méthodes échangées*

## 



*Annexe 2 : Diagramme de classes des commandes temps réel*