# Introduction

Le fil rouge des TP de la filière IS du second semestre est un gyropode. Dans le cadre de l’UF “Programmation Orientée Objet et Temps Réel”, ce fil rouge est décliné à travers la conception et la programmation de la commande et de la supervision en temps réel du prototype de gyropode. Ce dernier a été conçu, réalisé et maintenu par une équipe d’enseignants, de techniciens et d’étudiants du GEI et du GM.

Pour vous guider, ce document illustre sur le système “gyropode” la démarche d’analyse que vous aurez à reproduire en TD et en TP sur le sous-système de commande et supervision ; elle est basée sur celle adoptée dans l’UF “Processus en Ingénierie Système”. La démarche procède en 3 grandes étapes : définir les exigences du système, définir les fonctions et leurs interactions (architecture logique qui ne sera pas présentée dans ce document), définir une architecture organique reposant sur des sous-systèmes ou composants assurant les fonctions.

Ce document commence par présenter une vue général du système gyropode, puis une réalisation des trois étapes de la démarche d’analyse à un haut niveau et enfin le travail attendu en TD.

# I. Présentation du prototype INSA de gyropode

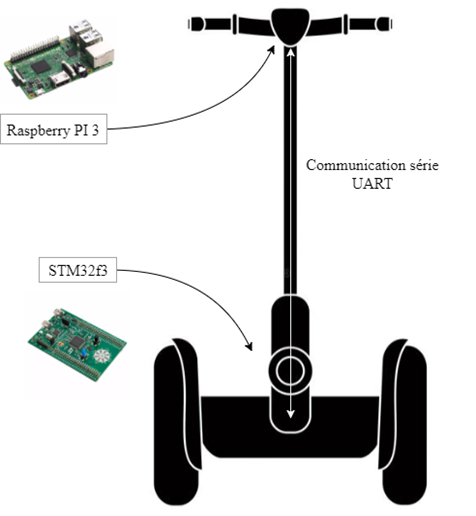
Pour concevoir et réaliser un prototype de gyropode répondant au fonctionnement attendu, plusieurs choix ont été faits ; ils sont exposés ici.

|  |  |
| --- | --- |
| Trois batteries de 12 V en série de capacité 12 Ah servent de source d’énergie. Ces batteries alimentent les deux moteurs à courant continu de 500 W qui permettent de mettre en mouvement chacun une roue. En effet, pour que le gyropode puisse tourner, il faut que la vitesse de chaque roue soit gérée indépendamment l’une de l’autre. Par exemple, pour tourner à droite, la roue gauche doit tourner plus vite que la roue droite. La vitesse des moteurs étant trop importante, un réducteur sous forme de pignon chaîne est utilisé en sortie de moteur. Les roues sont de diamètre 25 cm. Tous ces composants se trouvent sur la partie basse du gyropode, au niveau du plateau sur lequel se tient l’utilisateur.  Le système peut supporter une charge utile de 100 kg mais il n’est pas équipé de capteur de poids.  Le guidon permet de piloter le gyropode. Il est en liaison pivot par rapport au plateau, ce qui permet de le faire pivoter de quelques degrés autour de la direction d’avancée du système.  Au niveau du guidon, une tablette sert à l’affichage de diverses informations : évolution de l’angle θ et la vitesse angulaire du guidon par rapport à la verticale, évolution de l’angle β, niveau de batterie, couple moteur, présence de l’utilisateur, arrêt d’urgence, état de la communication entre les deux cartes électroniques, et des informations supplémentaires éventuelles. | *Prototype du gyropode INSA* |

Deux boutons de part et d’autre du guidon doivent être maintenus enfoncés pour que le gyropode puisse fonctionner. Lorsque les boutons sont relâchés, le gyropode s’immobilise et ne peut être utilisé. Ce sont donc des boutons de sécurité qui permettent de s’assurer de la présence du pilote sur le plateau. Ils assurent la sécurité du conducteur lorsqu’il descend subitement du gyropode. NB: il suffit qu’un seul bouton soit enfoncé pour que le gyropode considère que le pilote est présent sur le plateau.

Pour arrêter rapidement le gyropode, le sous-système de commande et supervision envoie une consigne de couple nulle. Le gyropode décélère alors pour atteindre une vitesse linéaire nulle.

Afin de commander (envoyer des consignes) et superviser (vérifier que le système est en fonctionnement nominal) le gyropode, deux cartes électroniques sont reliées entre elles par une liaison série. La première est une carte STM32 (bas niveau) située au-dessus des moteurs et la deuxième est une carte Raspberry Pi 3 (haut niveau) située au niveau du guidon.

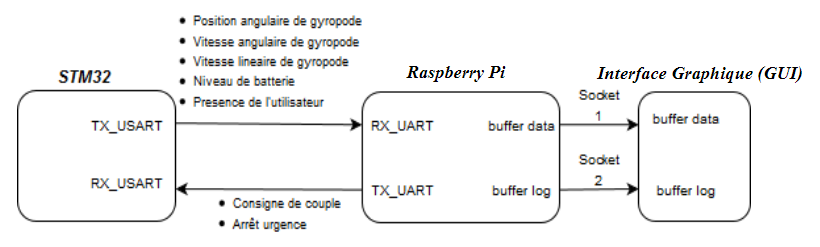


*Figure 1 : Schéma de positionnement des cartes électroniques sur le châssis*

La carte STM32 reçoit les informations de tous les capteurs présents sur la maquette comme l’accéléromètre et le gyroscope, puis les transfère à la carte Raspberry. Aussi, elle effectue la boucle de courant à partir de la consigne de couple envoyée par la carte Raspberry. Puis, à l’aide d’un correcteur PI, elle contrôle les moteurs par MLI (Modulation à Largeur d’Impulsions).

La carte Raspberry Pi 3 est programmée en temps réel pour assurer la surveillance du niveau de batterie et la gestion de l’arrêt d’urgence. Elle transmet à la carte STM32 la consigne de couple après l’avoir calculée grâce aux valeurs d’angle et de vitesse de rotation du gyropode envoyées par la carte STM32. C’est la boucle d’asservissement en angle.

La communication entre les deux cartes s’effectue à travers une liaison série de type UART. Les données sont encapsulées dans des trames 35 bits. Les données envoyées de la carte STM32 à la carte Raspberry Pi 3 sont toutes les données des capteurs de courant dans les moteurs, de position angulaire, de vitesse angulaire, de niveau de batterie, … Les données transmises dans l’autre sens sont la consigne de couple et celle d’arrêt d’urgence.

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

# II. Exemple de la démarche d’analyse sur le système

## II.1. Besoin et exigences

Objectif du système: Déplacer une charge utile de 100 kg maximum sur un chemin goudronné, de pente maximale 5 %, à une vitesse maximale de 7 km/h en toute sécurité.

La vitesse est limitée à 7 km/h car le prototype n’est pas conçu pour aller plus vite et la charge utile n’est pas mesurée car le Segway ne dispose pas de capteur de poids.

Exigences : Le système doit donc satisfaire plusieurs exigences. Les exigences marquées d'une \* sont les exigences qui doivent être remplies par le programme de supervision. Certaines sont fonctionnelles (E1, E2, E3, E4, E5) et d'autres non fonctionnelles (EF6, EF7, EF8, EF9, EF10, EF11) :

E1 : Le gyropode doit permettre le déplacement de l'utilisateur (accélération, freinage, arrêt, direction) à une vitesse < 7 km/h.

E2\* : Le gyropode doit pouvoir limiter la vitesse de l'utilisateur si celle-ci dépasse 7 km/h.

E3\* : Le gyropode doit surveiller le niveau de la batterie régulièrement (toutes les secondes). Il doit afficher une alarme lorsque le niveau de batterie devient inférieur à 25% et déclencher l'arrêt du système lorsque le niveau de batterie est inférieur à 15% en moins de 2 secondes.

E4\* : Le gyropode doit détecter en moins de 500 ms si l'utilisateur n'est plus présent sur le système et déclencher le système d'arrêt d’urgence.

E5\* : A des fins de contrôle et de debug du système, le gyropode doit permettre à l’utilisateur de connaître sa vitesse angulaire, l’angle de la barre du gyropode, l’angle ẞ, la consigne de couple appliquée, le niveau de batterie, l'état de la variable *arret d’urgence*, l’état de la variable *prensence\_user* et l’état de la communication à travers l'affichage de ces valeurs ainsi que l'affichage d'informations textuelles. Ces informations doivent être affichées sur l’interface graphique en 500 ms.

EF6 : Le gyropode doit protéger physiquement l'utilisateur en cas de choc frontal à 10 km/h.

EF7 : Le gyropode doit empêcher son utilisation aux personnes non autorisées.

EF8 : Le gyropode doit être équipé d’un système de signalisation.

EF9 : Le gyropode ne doit pas dépasser 1 m 80 de hauteur, 90 cm de largeur et 50 cm de profondeur.

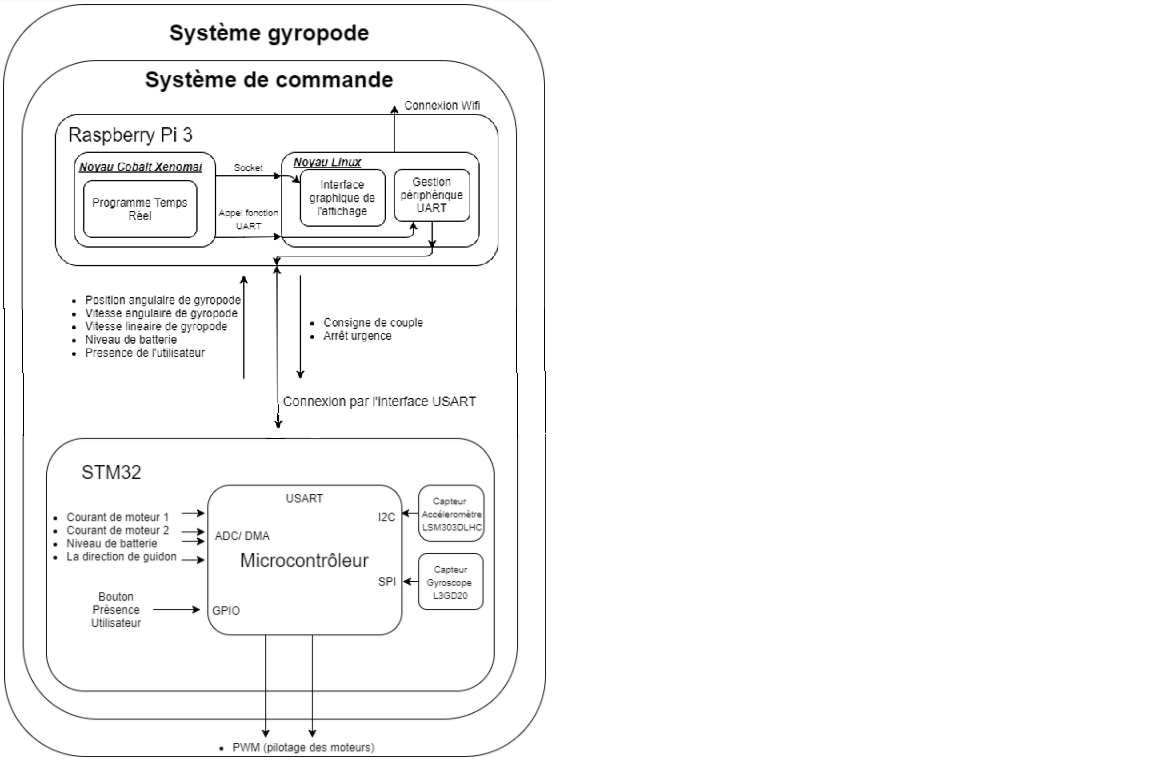
EF10 : Le gyropode doit être protégé de la pluie et de l'humidité.

EF11 : Le gyropode doit être facilement transportable.

## II.3) Architecture organique

On peut structurer le système en deux sous-systèmes : un sous-système de motorisation, comprenant le moteur, les chaînes, la carte électronique du pont en H, et les batteries ; et un sous-système de commande et supervision avec la carte STM32, la Raspberry Pi 3 ainsi que tous les capteurs du système. Ce dernier sous système de commande et supervision peut être éclaté en deux sous-systèmes : un sous-système bas niveau comprenant les capteurs et la carte STM32, et un sous-système haut niveau, ou de *supervision*, avec la carte Raspberry Pi 3, qui exécute le programme de supervision que vous allez concevoir, ainsi que l'interface graphique.

Nous vous présentons ici l’architecture organique du sous-système de commande :



*Figure 3 : Flux de données pour le sous-système de « commande et supervision »*

# III. Sujet de TD

Le but du TD est d’appliquer cette démarche au niveau logiciel c’est-à-dire au niveau des cartes Raspberry et STM32. Vous allez donc devoir :

* En déduire les fonctions associées. proposer un diagramme (format SA-RT like) en liant ces différentes fonctions par leurs entrées et sorties
* Proposer un découpage en tâches et un diagramme d’architecture (format AADL) de l’application (tâches et paramètres des tâches, activations, synchronisation, communication)

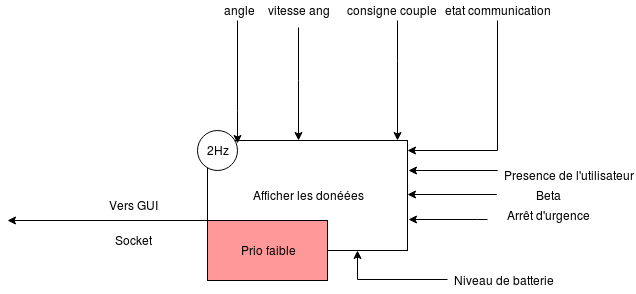
**Exemple pour la tâche Affichage :**

Affichage

Le thread *Affichage* se charge de communiquer à l’utilisateur toutes les informations importantes récoltées par le STM32 ainsi que les variables du programme de temps réel comme l'état de la communication ou le déclenchement de l'arrêt d'urgence. Lors de sa première exécution, il initialise le socket qui sera utilisé pour communiquer les informations au processus de l'interface graphique. Ensuite à chaque exécution, il consulte certaines variables partagées et envoie des trames sur un socket pour que le programme GUI puisse les afficher. Pour bien pouvoir notifier la perte de communication, la priorité de ce thread doit être supérieure à celle du thread *Communication*.

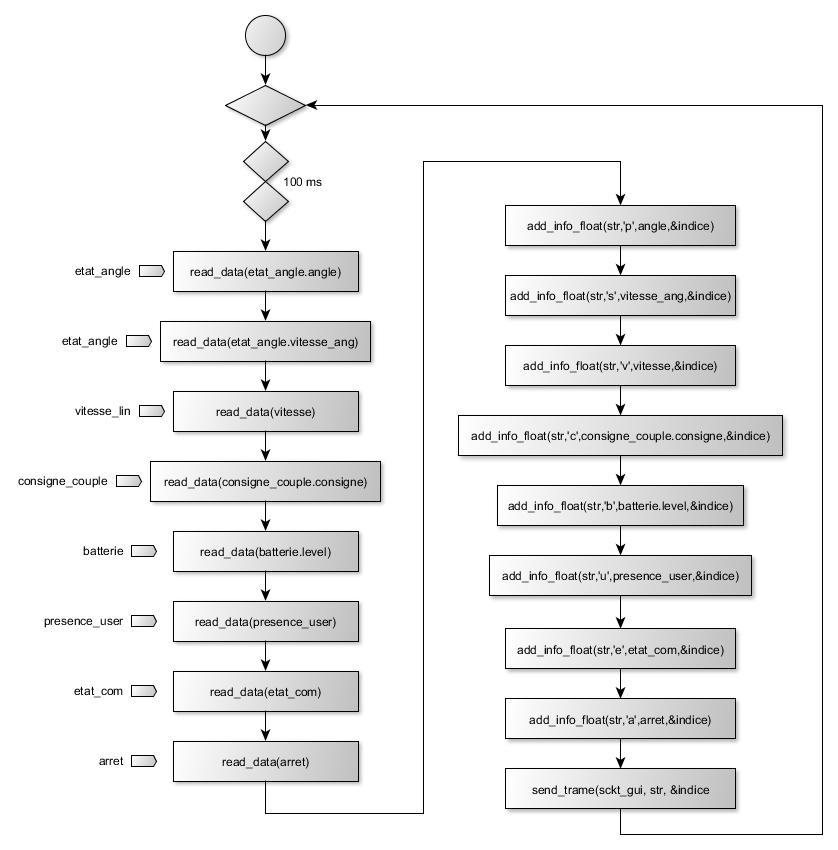
**Besoin** : Afficher toutes les données nécessaires à l’utilisateur sur l’interface graphique (voir E5\*).

**Architecture logique qui en découle est celle-ci :**

****

*Figure 4 : SA-RT de la thread Affichage*

**Et nous pouvons la traduire en diagramme d’activité comme suit:**



*Figure 5 : Diagramme d’activités de la thread Affichage*

La tâche Affichage est entièrement codée dans le programme que vous aurez en TP et vous servira de support et d’exemple pour coder les autres fonctions.

**Exercice :**  
Reproduisez ce schéma de réflexion pour les tâches énumérées ci-dessous en définissant pour chacunes d’elles une architecture logique et un diagramme d’activité. Ensuite, regroupez les architectures logiques de chaque tâche dans une même architecture.

1) Arrêt d'urgence

Le thread *Arrêt Urgence* est déclenché par le sémaphore *arret* (qui peut être déclenché par le thread Surveillance Batterie ou Présence User). Nous avons choisi un fonctionnement apériodique car cette tâche a seulement besoin d'être exécutée lorsqu'il est nécessaire d'arrêter le système, un fonctionnement périodique aurait fait consommer des ressources inutilement. Lorsque la tâche est déclenchée, la variable partagée arrêt est mise à *true*,et un message de type arrêt (de label "a"), avec la valeur égale à "1" est envoyée à la file de message.

2) Présence User

Le thread *Présence User* vérifie de manière périodique à 10 Hz la présence de l’utilisateur en consultant la variable partagée *presence\_user*. Si cette variable est à l’état *false*, la variable arrêt est mis à l’état *true*, afin que l’arrêt d’urgence soit déclenché. Puisque le STM32 effectue lui-même un comptage de 500 ms, il n'est pas nécessaire de le faire dans le système de temps réel. *Cependant dans la version simulateur, utilisée par l'étudiant, il n'y a pas de compteur de 500 ms au niveau du STM32, ce qui signifie que l'étudiant devra gérer cette contrainte de temps (qui vise à éviter qu'une absence de l'utilisateur soit détectée suite à un faux contact).* De la même manière, le thread consulte la variable état communication, et grâce à un autre compteur, il vérifie la durée du problème de communication et demande l’arrêt du système si ce compteur est égal à 2 (200 ms de perte de communication).

3) Surveillance batterie

Le thread *Surveillance Batterie* se charge de vérifier le bon fonctionnement des moteurs. Il consulte périodiquement, à une fréquence de 1 Hz, les informations contenues dans la variable partagée batterie, tant que la variable état communication est à l’état *true*. Si le niveau de batterie est faible, la variable *batterie* est mise à jour afin que l’utilisateur en soit informé par le thread *Affichage*. Enfin, si le niveau de batterie atteint un niveau critique, la variable *arrêt* est mise à jour pour déclencher l’arrêt d’urgence. La fréquence de ce thread n'a pas besoin d'être plus élevée, le niveau des batteries évoluant de manière relativement lente. Afin de pouvoir notifier la remise en marche du système après un arrêt d'urgence, il est nécessaire de remettre la variable partagée *arret* à 0. C'est cette tâche qui a été choisie pour cette fonction car il n'est pas nécessaire de faire la vérification du bon fonctionnement du système à une fréquence supérieure à 1 Hz. Cette tâche consulte alors régulièrement la valeur de la variable *presence\_user* et remet à 0 la valeur de la variable *arret* si *presence\_user* est à l'état *true*.

4) Envoyer

Le thread *Envoyer* est chargé d’envoyer au STM32 les messages qui sont stockés dans la file de messages. Etant donné que le thread Asservissement écrit un message dans la file à 50 Hz, et que le thread *Arrêt* peut également ajouter un message dans la file, cette tâche doit être périodique et deux fois plus rapide que la tâche Asservissement, elle fonctionne donc à 100 Hz.

5) Communication

Le thread *Communication* se charge de récupérer périodiquement, à une fréquence de 94Hz, les informations reçues du STM32 à l’aide de la liaison série. La fréquence a été choisie pour correspondre à celle de l'envoi des données par le STM32. La tâche déchiffre les trames et met à jour les variables partagées utilisées par les autres threads. Elle doit d’abord vérifier que la communication UART est bien établie. Si ce n’est pas le cas, elle va réessayer de connecter. Le thread se met ensuite en attente d’une trame sur la liaison série. A la réception de celle-ci, les données sont extraites et les variables partagées sont mises à jour en fonction y compris la variable état communication qui doit être utilisée pour les autres threads. Ce thread fonctionne à une fréquence de 50 Hz.

6) Asservissement

Le thread *Asservissement* effectue les calculs nécessaires au contrôle des déplacements du gyropode. Si la variable état communication est égale à *true*, il récupère les informations d’angle, et après son calcul, écrit les résultats dans la variable partagée de consigne courant et ajoute ces valeurs à la file de message. La fréquence de 50 Hz correspond à la fréquence d'échantillonnage calculée lors de la conception des lois d'automatique. La sauvegarde de la valeur de consigne de couple sert au thread *Affichage* qui relaie cette information au programme GUI.