

GYROPODE

SUJET DE TD & TP

**DÉPARTEMENT DE GÉNIE
ÉLECTRIQUE ET INFORMATIQUE**

Février 2019

Enseignant : Claude BARON



1. INTRODUCTION ET VUE D'ENSEMBLE

Dans le cadre de l'UF “Programmation Orientée Objet et Temps Réel”, nous abordons la conception et la programmation de la commande et de la supervision en temps réel du prototype de gyropode 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 fonctionnelle), définir une architecture organique reposant sur des sous-systèmes ou composants physiques assurant les fonctions.

La partie électronique du gyropode comporte de 2 cartes. La carte STM32 est en charge de la gestion des capteurs et actionneurs. Elle est aussi en charge des boucles d’asservissement de bas niveau, proches du matériel, ainsi que du filtrage des capteurs. Elle communique l’état du système (position angulaire, vitesses angulaire et linéaire, niveau de batterie, présence de l’utilisateur sur le plateau) à une carte Raspberry Pi et reçoit les ordres (consignes de couple, arrêt d’urgence...) de la Raspberry Pi via une liaison série asynchrone. La carte Raspberry Pi est en charge de la surveillance du système (exécution d’un programme de supervision), de la loi de commande de haut niveau et de la communication réseau (Wi-Fi) avec un poste de travail (PC).

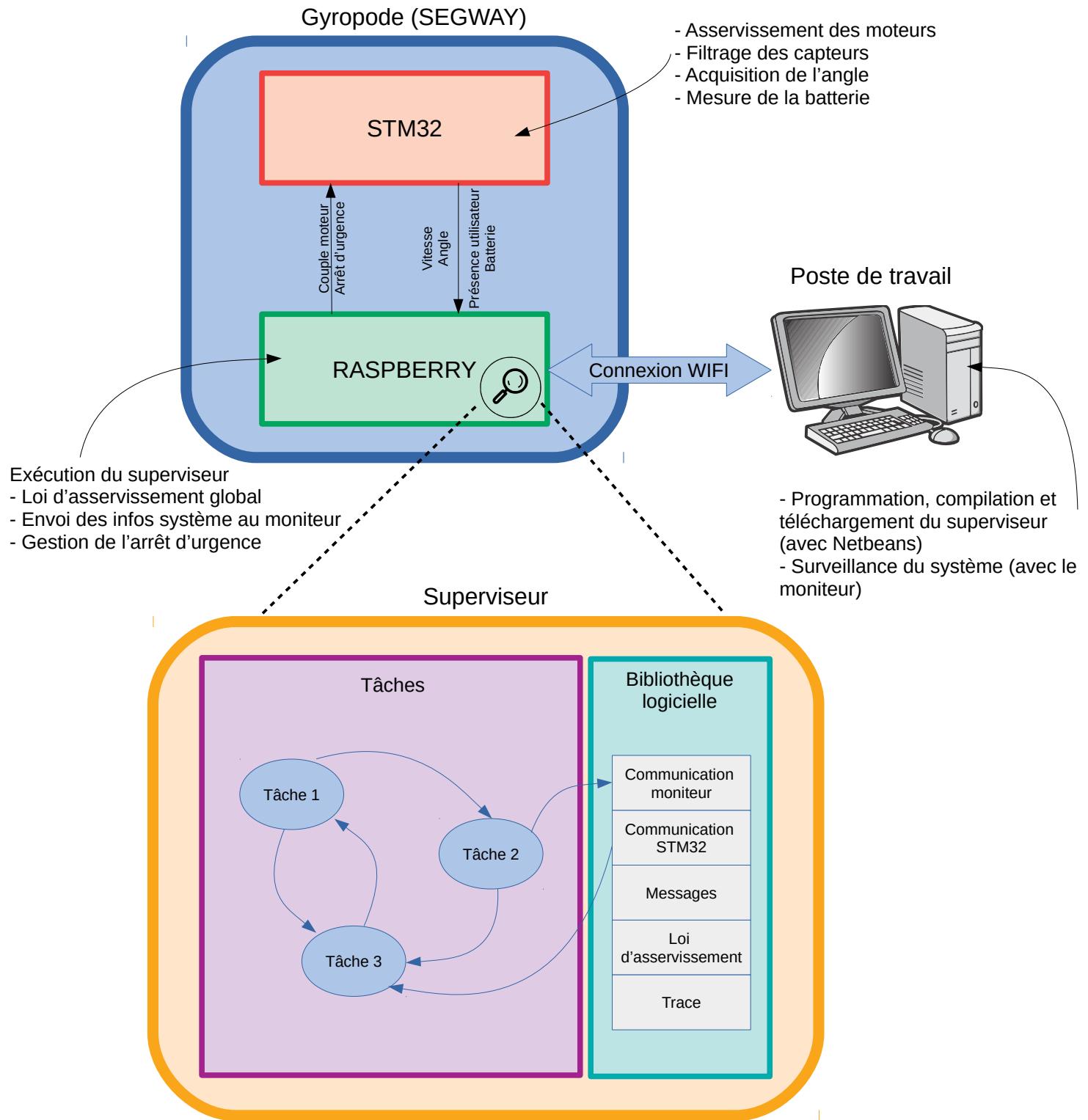
Dans le cadre de ce projet, votre rôle sera de concevoir, coder et tester le programme de supervision s'exécutant sur la carte Raspberry.

2. LEXIQUE

Ce projet utilise un certain nombre de termes et de concepts qu'il convient de préciser avant d'entrer plus en détail dans le projet. Avant d'aller plus loin, voici une liste de mot couramment utilisé dans ce document avec le sens utilisé dans ce projet :

- Dépôt (git) : En informatique, un dépôt correspond à l'ensemble des documents informatiques d'un projet géré en gestion de version. Cela se traduit par un archivage en réseau qui une fois téléchargé sur l'ordinateur correspond au contenu d'un répertoire donné.
- Git : Git est un gestionnaire de version couramment utilisé en informatique. Son rôle est de gérer les évolutions faites dans les fichiers d'un dépôt et assurer la synchronisation avec archivage réseau, dans notre cas Github.
- Moniteur : Dans le cadre de ce projet, le moniteur correspond au logiciel s'exécutant sur votre poste informatique et dont le rôle est d'afficher l'état du sSegway (vitesse, angle, présence de l'utilisateur...). Il est fourni dans le dépôt de ce TP.
- Superviseur : Le superviseur correspond au logiciel s'exécutant sur la Raspberry du Segway. C'est ce programme que vous aurez à concevoir et modifier pour qu'il réponde aux exigences.
- Netbeans : Un environnement de développement (IDE) qui vous servira à écrire, compiler et télécharger le programme du superviseur dans la raspberry du Segway.
- Raspberry : Un ordinateur minimaliste et embarquable, équivalent en puissance d'un smartphone d'entrée de gamme.
- STM32 : Une puce programmable (microcontrôleur) en charge du contrôle des capteurs et actionneurs du Segway.
- Terminal : Une fenêtre (généralement noire), ou l'on peut taper des commandes exécutées par le système Ubuntu

3. VUE GLOBALE DU SYSTÈME



4. LE PROTOTYPE INSA DE GYROPODE

Pour concevoir et réaliser un prototype de gyropode, plusieurs choix ont été faits par les équipes. 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 (il n'est cependant pas encore équipé de capteur de poids pour s'en assurer).

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.



Illustration 1: Le Segway de l'INSA

Deux boutons de part et d'autre du guidon doivent être maintenus enfouis 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: Si au moins un bouton est relâché, le pilote est considéré comme absent (chute).

Si le système est en mouvement, 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.

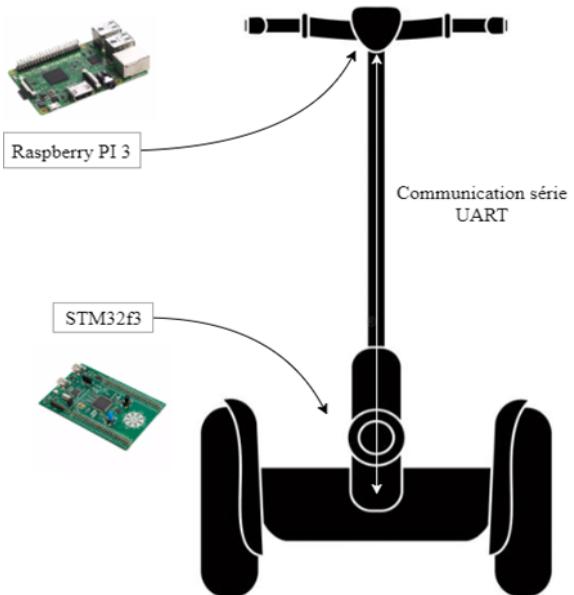


Illustration 2: Emplacement des composants sur le Segway réel.

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. Puis, à l'aide d'un correcteur PI, elle contrôle les moteurs par MLI (Modulation à Largeur d'Impulsions).

La carte Raspberry Pi 3 s'occupe de la supervision et de la loi de commande générale. Elle assure 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 envoyées de la carte STM32 à la carte Raspberry Pi 3 sont l'angle Bêta (entre l'utilisateur et la plateforme), la position angulaire de la plateforme, sa vitesse angulaire, le niveau de batterie, la présence de l'utilisateur. Les données transmises dans l'autre sens sont la consigne de couple et celle d'arrêt d'urgence.

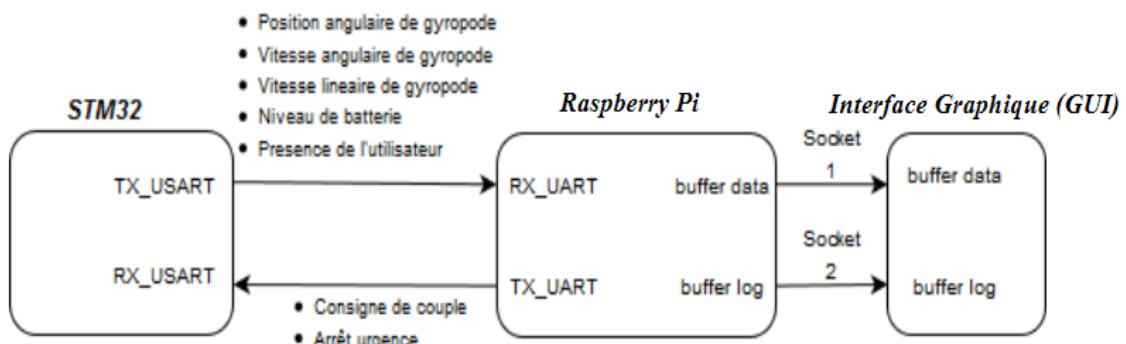


Illustration 3: Communication entre les différents systèmes

En ce qui concerne l'angle Bêta, voici un schéma expliquant la relation entre l'utilisateur, la position de la plateforme et l'angle Bêta.

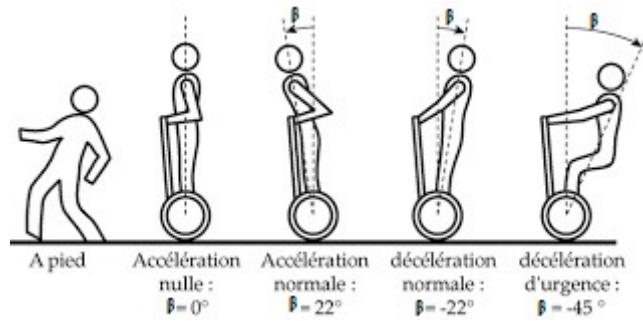


Illustration 4: Valeurs de Bêta en fonction de la position de l'utilisateur

5. ILLUSTRATION DE LA DÉMARCHE D'ANALYSE SUR LE SYSTÈME GYROPODE

5.1 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 (NB : La vitesse est limitée à 7 km/h car le prototype n'est pas conçu pour aller plus vite).

Exigences : Le système doit donc satisfaire plusieurs exigences (NB : les exigences marquées d'une * sont les exigences sont celles relatives à la supervision ; certaines sont fonctionnelles (E1, E2, E3, E4, E5) et d'autres non fonctionnelles (EF6, EF7, EF8, EF9, EF10, EF11)). La liste ci-dessous n'est pas exhaustive.

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, le gyropode doit permettre à l'utilisateur de connaître sa vitesse angulaire, l'angle de la barre du gyropode, l'angle β , la consigne de couple appliquée, le niveau de batterie, l'état de l'arrêt d'urgence, la présence de l'utilisateur et l'état de la communication. 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.

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.

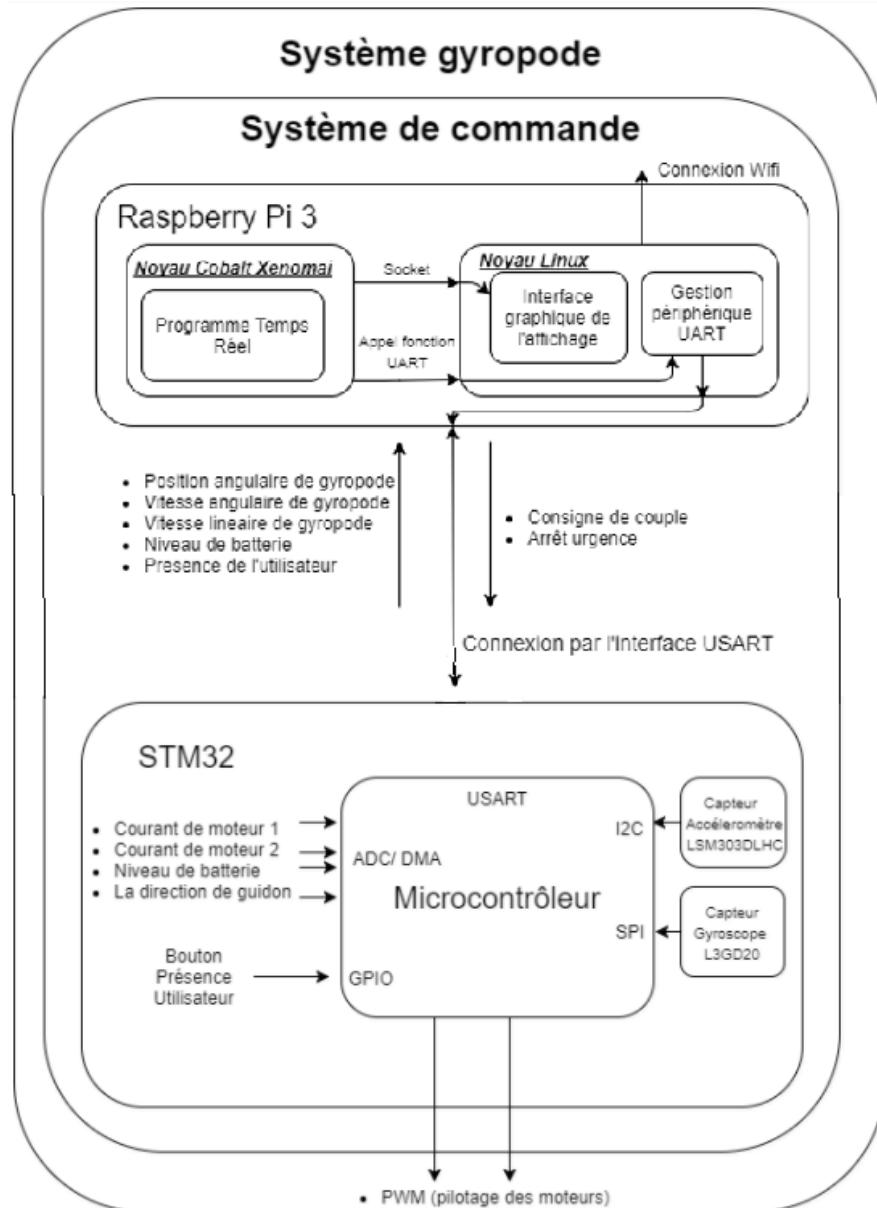
5.2 Architecture logique

A revoir, je ne sais pas trop quoi mettre

5.3 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 à son tour être décomposé en deux sous-systèmes : un sous-système de commande de bas niveau, comprenant les capteurs et la carte STM32, et un sous-système supervision de haut niveau, comprenant avec la carte Raspberry Pi 3, qui exécutera le programme de supervision que vous allez concevoir, ainsi que l'interface graphique.

Voici l'architecture organique du sous-système de commande et supervision:



6. LE PROGRAMME DE SUPERVISION

La carte Raspberry Pi est un nano-ordinateur mono-carte à processeur ARM, sur lequel s'exécute une version adaptée du système d'exploitation Linux. Le noyau de base a été modifié pour lui ajouter un co-noyau temps réel, Xenomai. La carte exécute le programme temps réel de supervision. L'interaction avec ce système Linux ne se fait qu'en ligne de commande (pas d'IHM) et, virtuellement tout ce que l'on peut faire sur un PC standard peut être réalisé sur la Raspberry.

Pour les opérations qui s'effectuent en temps contraint (gestion de l'état du système et calcul de la loi de commande), le programme de supervision utilise l'exécutif temps réel Xenomai. Pour les entrées/sorties (série ou réseau), il utilise le noyau Linux. Le routage des opérations vers l'un ou l'autre des noyaux se fait de façon transparente pour le programme, à partir de l'interface de programmation de Xenomai, du fait de son architecture en co-noyaux.

Le programme de supervision que vous devez développer sera écrit en C++ (programmation objet). Une première version de ce programme, compilable, vous est fournie ; il faudra la compléter pour répondre au cahier des charges et coder toutes les fonctionnalités attendues. Cette version est livrée avec un ensemble de classes déjà écrites, notamment en ce qui concerne la gestion de la liaison série et la mise en place d'un serveur TCP pour remonter l'état du système (variables) vers un moniteur (interface graphique) s'exécutant sur le poste de travail. L'objectif est d'illustrer comment utiliser les classes, mais aussi de vous montrer qu'elle sera l'architecture générale du programme (en threads) et comment créer et utiliser les différents outils de communication et de synchronisation offerts par Xenomai (sémaphore, mutex, file de message ...).

6.1 Variables échangées entre le STM32 et le superviseur

Informations envoyées par le STM32 vers le superviseur

Donnée	Type	Unité
Position angulaire	float	rad
Vitesse angulaire	float	rad /s
Niveau batterie	integer	%
Vitesse linéaire	float	m/s
Présence utilisateur	integer	1 si présent, 0 sinon

Informations envoyées par le superviseur au STM32

Donnée	Type	Unité
Consigne de couple	float	N.m
Arrêt	int	1 si arrêt d'urgence, 0 sinon

Plus d'information sur la structure des messages et leur format peut être trouvée en [Annexe C – Communication série (STM32)]

6.2 Trace du programme de supervision

Pour mieux comprendre l'exécution des tâches dans le programme de supervision, une classe Trace permet de générer des messages relatifs à l'état d'élément de synchronisation, qui peuvent être alors affichés sur la console (terminal) du poste de travail ou dans la fenêtre de log du moniteur. Plus d'information se trouvent en [Annexe D – Mise en place d'une trace]

6.3 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 l'une des deux conditions n'est pas remplie. Côté STM32, un compteur vérifie 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.

7. LE SIMULATEUR DE GYROPODE

Plutôt que de travailler directement sur le gyropode, avec tous les risques que comporte la mise au point du programme, un simulateur a été développé, sur lequel vous allez travailler. Le simulateur du gyropode est composé de la même carte de STM32 que celle utilisée dans le prototype physique ; toutes les interruptions et fréquences des tâches sont similaires à celles du prototype afin de retranscrire au mieux le comportement du gyropode réel. Le code s'exécutant sur la carte STM32 est une version un peu modifiée afin de simuler la présence des roues et le comportement de la plateforme mais la partie dédiée à la communication des données avec le superviseur est identique au prototype.

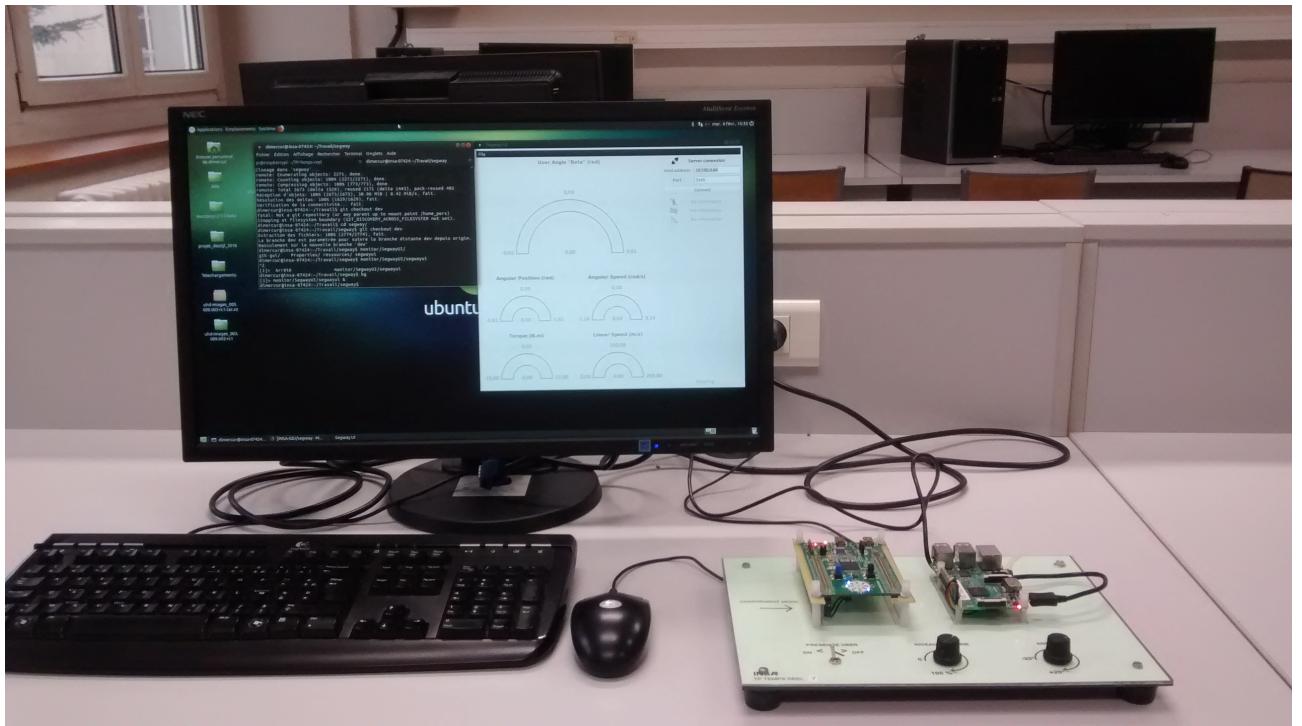


Illustration 5: Le poste de travail avec son simulateur

Deux parties spécifiques à la simulation ont été rajoutées. Une partie dédiée à la simulation physique du gyropode (angle d'inclinaison, simulé par un bouton rotatif), tandis que l'autre sert à simuler certains paramètres tels que le niveau de batterie (bouton rotatif) ou la présence de l'utilisateur (interrupteur).

7.1 Simulation physique du gyropode

La simulation du comportement physique du gyropode est effectuée grâce au modèle du système qui nous a été fourni. Il s'agit d'un modèle physique linéarisé du gyropode qui n'est valable qu'entre une inclinaison de -20° à $+20^\circ$. Mais cela est suffisant pour simuler le comportement du système.

Il aurait été intéressant de connaître la vitesse linéaire ; cependant, pour la calculer, il nous faudrait la tension fournie aux moteurs, qui n'est pas disponible dans le simulateur. C'est pourquoi dans le simulateur, la vitesse linéaire est toujours à 0 contrairement au prototype.

7.2 Simulation des variables du système

Dans le but de tester le code du logiciel de supervision, vous allez être amenés à faire varier l'état des variables du système comme le niveau de batterie ou la présence de l'utilisateur.

Le simulateur possède deux potentiomètres (niveau batterie et inclinaison) et un bouton (présence user).

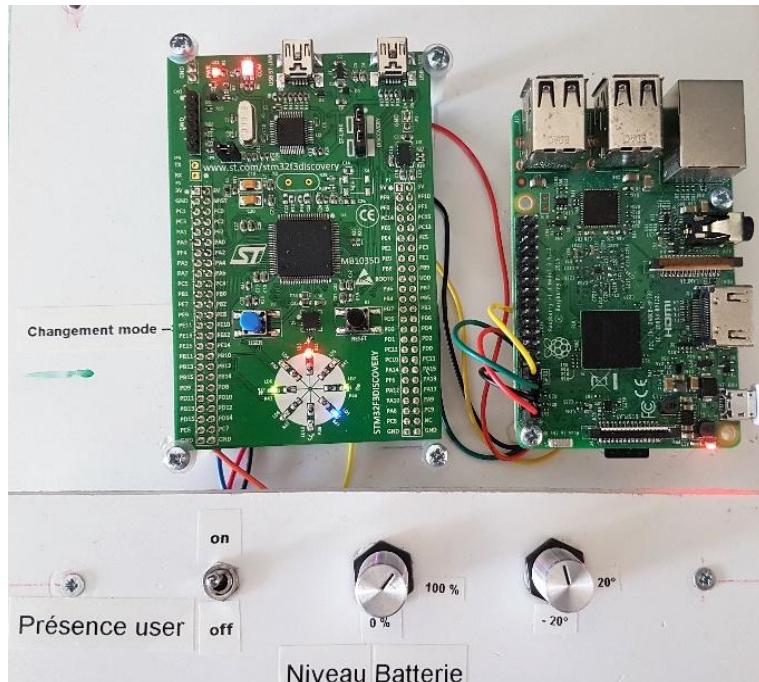


Illustration 6: Emplacement des boutons, potentiomètres et led sur le simulateur

La modification des variables se fait directement grâce à ces éléments. Les variations des variables sont instantanées et peuvent être continues, ce qui se rapproche de la réalité.

- Potentiomètres

Toutes les lectures des valeurs de potentiomètre sont effectuées par l'ADC (Convertisseur Analogique vers Digital) du STM32. Un potentiomètre sert à simuler le niveau de batterie, un autre potentiomètre sert à simuler la valeur de l'angle Béta entre -20 degrés et +20 degrés.

- Boutons

Le bouton sert à simuler le bouton vérifiant la présence de l'utilisateur, présent sur le guidon de la maquette.

RQ :Le bouton bleu sur la carte STM32 (notée « Changement mode » sur le simulateur) sert à changer le mode de modification de l'angle utilisateur : soit ce dernier est fixé par le potentiomètre, soit il est simulé par accéléromètre présent dans la carte STM32.

Pour faciliter la compréhension des opérations effectuées dans le STM32, des LEDs (visibles sur l'image, en bas de la carte de gauche, autour d'un cercle blanc) servent à indiquer différents états du simulateur.

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	État de présence de l'utilisateur => ON : Présence => OFF: Absence
LED 8	État 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

8. LE MONITEUR

L'interface graphique du moniteur s'exécute sur le poste de travail de la salle de TP. Lors du démarrage initial, l'interface a l'apparence suivante :

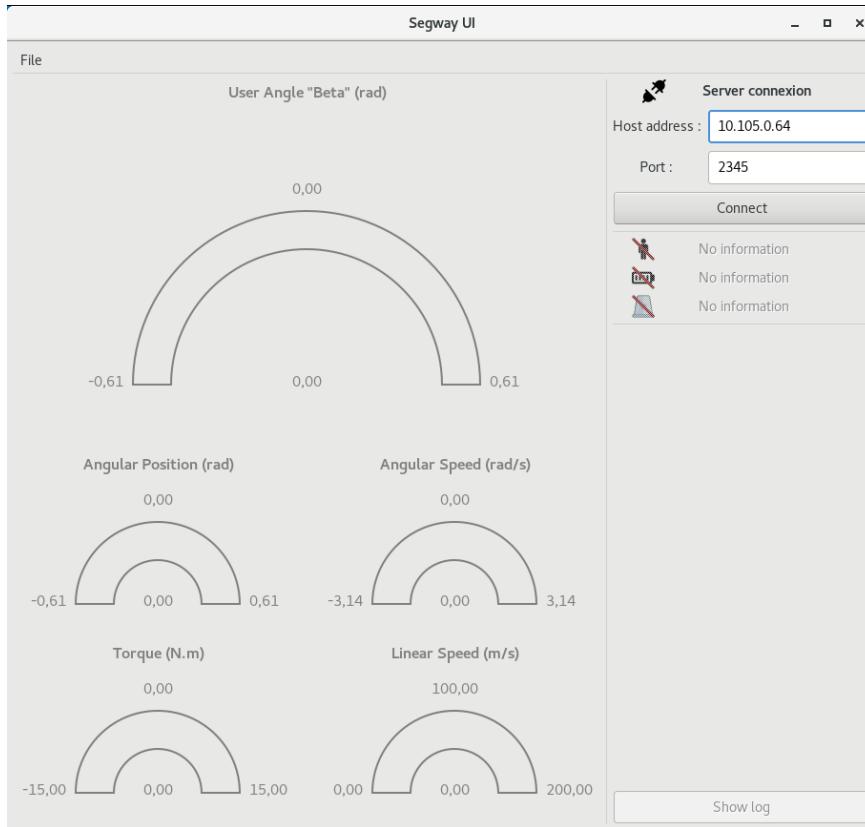


Illustration 7: Vue de l'interface du moniteur avant d'être connecté au superviseur.

Les différents paramètres sont initialement grisés, jusqu'à ce que le moniteur soit connecté au superviseur. Ils le resteront encore jusqu'à recevoir les premières données en provenance du superviseur, via la classe ComGui.

La classe ComGui offre une méthode Write permettant d'envoyer un message vers le moniteur, message contenant soit un paramètre (angle, batterie, présence utilisateur, ...), soit une information sur l'état d'un mutex, sémaphore ou tâche.

Les messages donnant des informations sur les paramètres sont affichés directement sur l'IHM, les messages de trace ne s'affichent que dans la fenêtre de log. Pour information, si la loi d'asservissement à réaliser dans le superviseur n'est pas faite (ou mal), les valeurs de position angulaire et vitesse angulaire varient sans cesse.

Les différents widgets redeviennent gris lorsque l'on se déconnecte du superviseur.

9. SUJET DE TD

Le but du TD est d'appliquer la démarche d'analyse illustrée précédemment au niveau logiciel c'est-à-dire au niveau des cartes Raspberry et STM32. Vous allez donc devoir :

- Identifier les fonctions que le sous-système XXX devra assurer et proposer un diagramme d'architecture logique (format SA-RT like) montrant l'organisation de ces différentes fonctions (caractérisation des fonctions et de leurs interfaces, flux de données par les échanges d'entrées et sorties, flux de contrôles (activations, synchronisations))
- Identifier un ensemble de tâches permettant l'exécution de ces fonctions sur la plateforme logicielle choisie (Xenomai) et proposer un diagramme d'architecture (format AADL) de l'application (caractérisation des tâches, des activations, choix et caractérisation des moyens de synchronisation et des moyens de communication, description de l'activité globale de l'application logicielle (entrelacement des flux d'exécutions des différentes tâches dans le temps) et de l'activité interne des tâches)

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 :

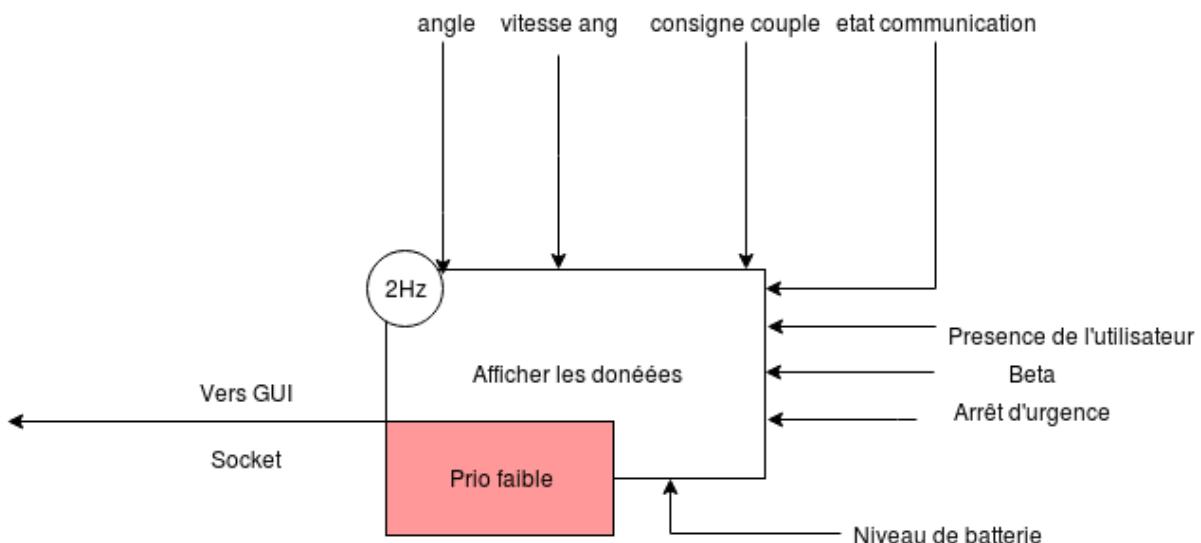


Figure 4 : SA-RT de la thread Affichage

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

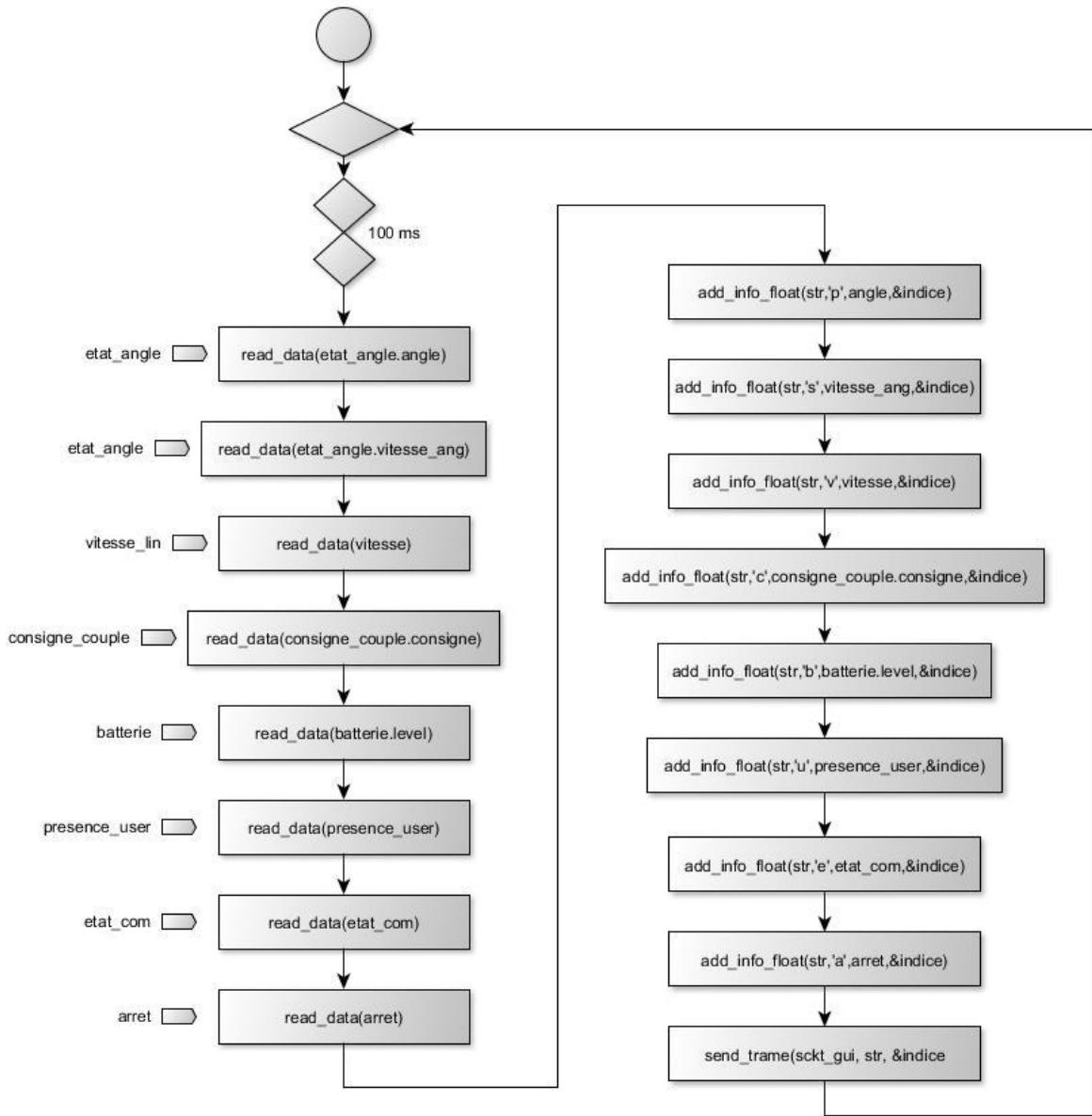


Figure 5 : Diagramme d'activités de la thread *Affichage*

Le thread *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 chacune d'elles une architecture logique et un diagramme d'activité. Ensuite, regroupez les architectures logiques de chaque tâche dans une même architecture.

9.1 Arrêt d'urgence

Le thread *Arrêt Urgence* est déclenché par le sémaphore `arrêt` (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.

9.2 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).

9.3 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*.

9.4 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.

9.5 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.

9.6 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.

10. CONSIGNES DE TP

La première chose à faire est de récupérer le dépôt Git contenant le projet de base. Pour cela, connectez vous sous Ubuntu et dans un terminal, créez un répertoire « gyropode » dans votre espace personnel :

```
mkdir gyropode  
cd gyropode
```

A partir de là, il faut récupérer le dépôt git :

```
git clone https://github.com/INSA-GEI/segway.git  
git checkout stable
```

Vous obtenez un répertoire « segway » contenant 4 répertoires :

- docs : contient la doc technique ainsi que les sujet de TP et TD
- monitor : contient le projet de l'IHM. L'exécutable se nomme segwayui et se trouve dans le répertoire SegwayUI, avec l'ensemble du code de l'IHM
- raspberry : contient le code du superviseur, sous forme d'un projet NetBeans, à compiler et exécuter sur la carte Raspberry

Pour lancer l'interface graphique, exéutez la commande suivante (dans le terminal sur le PC) :

```
./segway/monitor/segwayui &
```

Le « & » à la fin de la commande est important ! Ensuite, lancez NetBeans pour ouvrir le projet superviseur (toujours dans un terminal) :

```
netbeans &
```

Referez-vous à [Annexe E – Prise en main de Netbeans] pour voir comment ouvrir un projet, ajoutez une cible de compilation à distance (la carte Raspberry du simulateur) et compilez votre code.

Ouvrez le projet se trouvant dans « segway/raspberry ». C'est votre projet de base, se contentant de transférer les paramètres remontés par le STM32 directement sur l'IHM. Les fichiers « tasks.cpp » et « tasks.h » doivent être complétés. Vous n'avez pas besoin de modifier les autres fichiers mais vous pouvez les consulter. Accessoirement, vous pouvez modifier « parametres.cpp » et « parametres.h » mais ce n'est pas nécessaire.

Si l'édition du code et sa compilation se font bien à travers NetBeans, l'exécution du superviseur doit se faire à la main, la faute aux droits d'administration nécessaire à l'exécution d'un programme Xenomai. Pour cela, dans un autre terminal, connectez-vous à votre simulateur (pensez à mettre l'adresse IP de votre cible (qui se trouve indiquée sur la carte de droite. Elle est de la forme 10.105.0.XX)

```
ssh pi@<adresse IP de la cible>
```

Les informations de connexion sur le simulateur sont :

Login : pi

Mot de passe : raspberry

Une fois connecté, votre programme se trouve au bout d'une arborescence ayant la forme suivante :

```
cd .netbeans/remote/10.105.0.64/insa-<Nom de votre machine>-  
Linux-x86_64/<Chemin vers le depot  
git>/segway/raspberry/dist/Debug_Raspberry/GNU-Linux/
```

où :

- <Nom de votre machine> : nom de la machine de TP, sous la forme insa-xxxx, xxxx étant le numéro de la machine.
- <Chemin vers le dépôt git> : chemin complet (à partir de /) où se trouve le dépôt git sur votre compte.

Une fois que vous êtes entré dans ce répertoire se trouve le programme « segway_supervisor ». Pour l'exécuter avec les droits administrateur (nécessaire pour Xenomai), il suffit de taper la commande :

```
sudo ./segway_supervisor
```

Dernier point : l'ensemble du code fourni utilise la convention de nommage du C#, à savoir un mélange de camelCase et de PascalCase. Le camelCase consiste à mettre en majuscule les premières lettres de chaque mot d'un nom, sauf le premier mot. Le PascalCase met en majuscule toutes les premières lettres de chaque mot. Ceci permet de rapidement différentier une variable et une méthode.

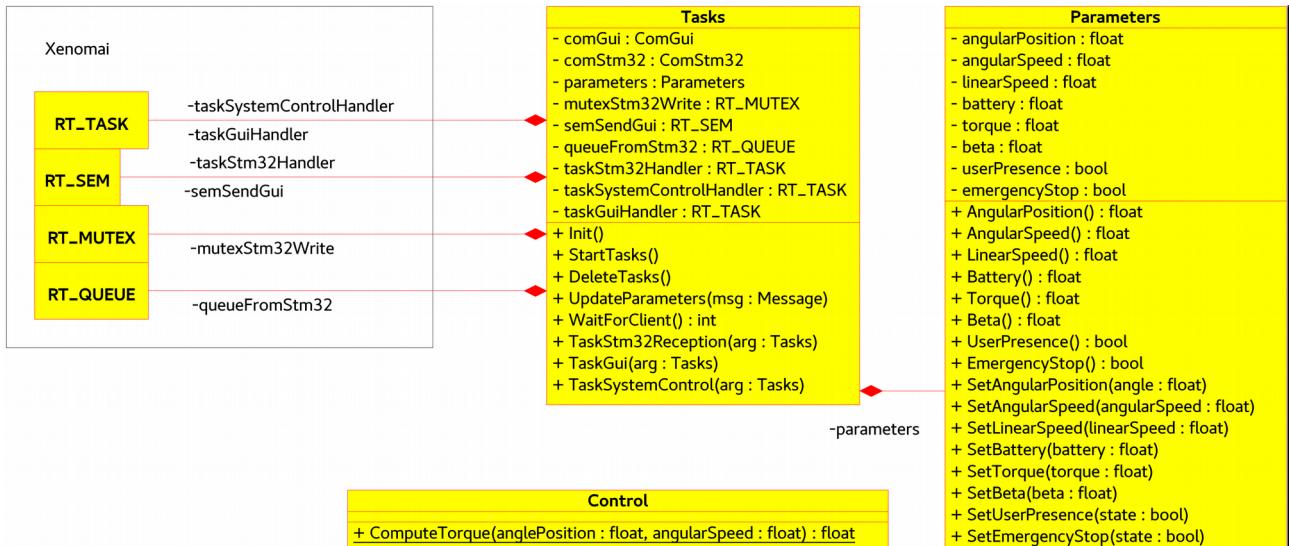
Pour info, voici un résumé rapide des différentes formes d'écriture rencontrées dans le code :

Nom de l'objet	Notation	Exemple
Nom de classe	PascalCase	MaClasse
Constructeur/Destructeur	PascalCase	MaClasse();

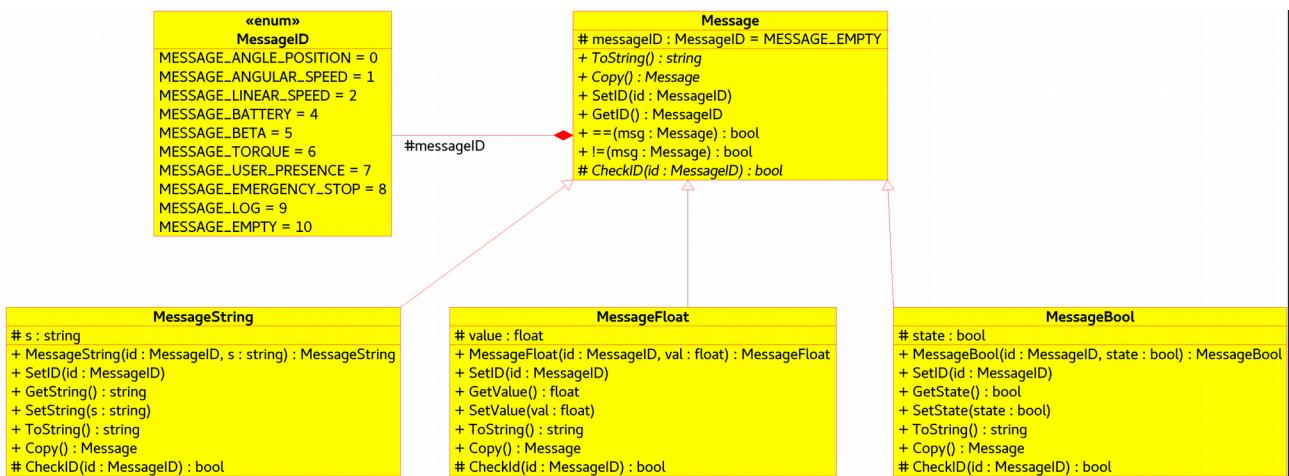
Méthode	PascalCase	void MaMethode();
Arguments de méthode	camelCase	void MaMethode(int unParametre);
Attributs, variable locale	camelCase	int compteurDeVitesse;
Constantes	MAJUSCULE	#define VALEUR_INITIALE 0
Type énuméré	PascalCase	typedef enum {...} MonEnumeration;

11. ANNEXE A – DIAGRAMMES DE CLASSE

11.1 Classes principales (temps réelles)



11.2 Classes de messages



11.3 Classes de communication et de trace

ComGui # socketFD : int = -1 # clientFD : int = -1 + Open(port : int) : int + Close() + AcceptClient() : int + Write(msg : Message) + Write_Pre() + Write_Post()	ComStm32 # fd : int + Open() : int + Close() : int + GetComState() : bool + Read() : Message + Write(msg : Message) : int + Read_Pre() + Read_Post() + Write_Pre() + Write_Post() # CharToFloat(bytes : unsigned char) : float # CharToBool(bytes : unsigned char) : bool # CharToInt(bytes : unsigned char) : unsigned int # CharToMessage(bytes : unsigned char) : Message # MessageToChar(msg : Message, inout buffer : unsigned char)
Trace - beginTime : int + GetTimeUs() : int + GetTimeMs() : int + WaitForMutex(mut : RT_MUTEX) : Message + MutexAcquired(mut : RT_MUTEX) : Message + MutexReleased(mut : RT_MUTEX) : RT_MUTEX + WaitForSem(sem : RT_SEM) : Message + SemEntered(sem : RT_SEM) : Message + SemSignaled(sem : RT_SEM) : RT_SEM + TaskEntered() : Message + TaskNewIteration() : Message + TaskEnded() : Message + TaskDeleted(task : RT_TASK) : Message - MutexGeneric(mut : RT_MUTEX, event : string) : Message - SemGeneric(sem : RT_SEM, event : string) : Message - TaskGeneric(task : RT_TASK, event : string) : Message	

12. ANNEXE B – RAPPELS DE C++ ET PROGRAMMATION OBJET

Le C++ est un langage de programmation, dérivé du langage C, permettant une programmation orientée objet, basé sur le principe de classe (d'autres types de programmation objet existent). Les spécifications du langage continuent aujourd'hui encore d'évoluer pour lui rajouter des fonctionnalités.

En ce qui concerne les langages objets basés sur des classes, l'élément principal est la classe, sorte de patron (template) servant à décrire l'architecture des objets. De ce fait, on dit qu'un objet est une instance d'une classe, ou instancie une classe, autrement dit qu'il donne une existence réelle à une classe, notamment en lui affectant une zone en mémoire.

Ainsi, quand on écrit :

```
class MaClasse {           // Définition de la classe MaClasse
public:
    int var;           // var est un attribut de MaClasse (variable)
    MaClasse() {var=0;} // Constructeur, appelé lors de la création d'un
                        // objet

    void Add(int x) {var=var+x;} // Méthode (fonction)
};

void main(void) {
    MaClasse maClasse; // Création de l'objet maClasse à partir de la
                        // classe MaClasse (instanciation) => var vaut 0

    maClasse.Add(2);    // Ajoute 2 à maClasse.var => var vaut 2
}
```

maClasse est un objet basé sur (de type) MaClasse. maClasse est donc une instance de la classe MaClasse. A partir de là, on peut accéder à ses attributs (variables) et méthodes (fonctions) publiques.

Il y a 2 façons de créer un objet et, du coup, 2 façons d'accéder à ses attributs et méthodes : soit déclarer directement l'objet, soit déclarer un pointeur sur l'objet. Dans le premier cas, on accédera aux éléments de l'objet par un point '.', dans l'autre cas, il faudra, avant de l'utiliser, lui allouer de la mémoire (via le mot clef 'new') et on accédera à ses éléments par le biais d'un flèche ' \rightarrow '

```
void main() {
    MaClasse monObjet;      // Déclaration d'un objet de type MaClasse
    MaClasse *ptrSurObjet; // Déclaration d'un pointeur sur un objet de
                          // type MaClasse

    monObjet.var=0;         // Accès à l'attribut var de monObjet
    ptrSurObjet = new MaClasse(); // Création d'un objet de type
                                // MaClasse et affectation du
                                // pointeur ptrSurObjet

    ptrSurObjet->var=0;     // Accès à l'attribut var de l'objet
                            // pointé par ptrSurObjet
}
```

12.1 Définition et déclaration

Une classe se compose d'une partie déclaration (qui se trouve dans un fichier .h) et d'une partie définition (qui se trouve dans un fichier .cpp).

La déclaration utilise le mot clef ‘class’. Les éléments constitutifs de la classe sont rangés selon leur accessibilité : publique, privée ou protégée. La forme classique d'une déclaration est la suivante :

```
class MaClasse {  
public:  
    MaClasse();  
    int UneMethode(int i);  
  
private:  
    int unAttributPrivee;  
};
```

Pour les méthodes les plus triviales, on peut directement ajouter le corps de la méthode (sa définition) lors de sa déclaration dans le .h (par exemple ici pour la méthode UneMethode)

```
class MaClasse {  
public:  
    MaClasse();  
    int UneMethode(int i) { unAttributPrivee=unAttributPrivee +i; }  
  
private:  
    int unAttributPrivee;  
};
```

Pour les cas plus complexe, on sépare bien la déclaration et la définition. La définition des méthodes se trouve du coup dans le fichier .cpp et prend la forme suivante :

- dans le fichier maclasse.h

```
class MaClasse {  
public:  
    MaClasse();  
    int UneMethode(int i);  
  
private:  
    int unAttributPrivee;  
};
```

- dans le fichier maclasse.cpp

```
#include <>maclasse.h>  
  
MaClasse::MaClasse() {  
    unAttributPrivee=0;  
}  
  
MaClasse::UneMethode(int i) {
```

```

        unAttributPrivee=unAttributPrivee +i;
    }
}

```

Notez dans ce cas que l'on indique à quelle classe appartient la méthode (ou le constructeur) en rajoutant le nom de la classe avant le nom de la méthode séparé ‘::’

12.2 Visibilité

Avec les langages objets, on peut choisir si les éléments d'un objet sont visibles par l'appelant ou pas. Dans le cas du C++, on a le choix entre 3 niveaux : publique, privé ou protégé. Un élément public est toujours directement accessible (via ‘.’ ou ‘->’) par l'appelant. Un élément privé est toujours directement inaccessible par l'appelant, seules les méthodes de l'objet y ont accès.

Le cas de protégé (‘protected’) est plus subtil. Par rapport à l'appelant, c'est comme un élément privé, inaccessible directement. Par contre, dans le cas d'une classe dérivée (héritage) d'une autre, les éléments protégés sont accessibles, tandis que les éléments privés restent privés, même pour une classe dérivée. La différence entre privé et protégé prend son sens dans l'héritage de classe.

Le tableau ci dessous résume les visibilités de chaque mot-clefs

	Public Members	Protected Members	Private Members
Public Inheritance	public	protected	private
Protected inheritance	protected	protected	private
Private inheritance	private	private	private

12.3 Méthodes statiques

Une méthode d'une classe peut être déclarée statique (mot-clef ‘static’). Dans ce cas, plutôt que d'être une méthode appartenant à l'objet instancié (et n'existant qu'à ce moment là), la méthode est utilisable telle qu'elle, sans avoir besoin préalablement d'instancier un objet pour y accéder. Et inversement, la méthode ne fait pas partie des méthodes appartenant à un objet. Elle a une existence propre, indépendante des objets.

```

class MaClasse {
public:
    MaClasse();
    int UneMethode(int i);
    static int UneMethodeStatique(int i);

private:
    int unAttributPrivee;
};

void main() {
    MaClasse monObjet;

    monObjet.UneMethode(5);           // Ok, ca marche
}

```

```

monObjet.UneMethodeStatique(3); // Ne compile pas: UneMethodeStatique
// n'appartient pas à l'objet monObjet
MaClasse::UneMethodeStatique(3); // Ok, ca marche
}

```

Étant donné que les méthodes statiques appartiennent à une classe mais pas aux objets qui l’instancient, les méthodes statiques ne peuvent pas accéder aux éléments non statiques d’une classe (méthodes ou attributs). Elles doivent se contenter des paramètres qui leur sont passé.

12.4 Méthodes virtuelles et polymorphisme

Une méthode peut être déclarée virtuelle ('virtual'). Elle existe belle et bien, mais l'intérêt de ce qualificateur est de permettre le polymorphisme lors de l'exécution du programme. Pour cela, il faut que les objets soient manipulés via un pointeur ou une référence sur le type de base.

Lorsqu'une classe dérive d'une classe de base, les méthodes virtuelles qui seront surchargées par cette classe fille pourront, à l'exécution, être correctement appelées, même si le type du pointeur qui pointe sur l'objet est du type de la classe de base.

Concrètement si l'on considère les classes ClasseDeBase et ClasseDerive décrites ci dessous

```

#include <iostream>

using namespace std;

class ClasseDeBase {
public:
    ClasseDeBase() { compteur=0; }
    void Add(int i) { compteur+=i; }
    virtual void ToString() { cout<<"ClasseDeBase: "<<compteur<<endl; }

protected:
    int compteur;
};

class ClasseDerive : public ClasseDeBase {
public:
    ClasseDerive() {compteurSpecifique=0;}
    void Add(int i) { compteurSpecifique+=i; }
    void ToString() {
        cout<<"ClasseDerive: "<<compteurSpecifique<<" Compteur de base=" <<
            compteur<<endl; }

protected:
    int compteurSpecifique;
};

int main(int argc, char** argv) {
    ClasseDeBase base;
    ClasseDeBase *ptrBase;
    ClasseDerive derive;

    ptrBase = new ClasseDerive();

    base.Add(1);
    derive.Add(2);
    ptrBase->Add(3);

    base.ToString();
}

```

```

    derive.ToString();
    ptrBase->ToString();

    return 0;
}

```

On voit que les deux classes possèdent une méthode Add et une méthode ToString. ClasseDerive surcharge les deux méthodes, mais ToString est une méthode virtuelle dans la classe de base, pas Add.

Dans le main, on affecte à ptrBase (qui est de type pointeur sur un objet de type ClasseDeBase) un pointeur de type ClasseDerive. On a le droit vu que ClasseDerive possède ClasseDeBase comme classe de base.

Lorsque l'on appelle les méthodes Add, vu qu'elles ne sont pas virtuelles, ce sont les méthodes Add de leurs classes respective qui sont appelées (dans le cas de ptrBase, c'est la méthode Add de la classe ClasseDeBase qui est appelé, ClasseDebase étant le type du pointeur). Comme Add n'est pas virtuelle, on ne remonte pas à la classe à l'origine du pointeur.

Lorsque l'on appelle les méthodes ToString, qui sont virtuelles, c'est bien la méthode de la classe à l'origine du pointeur qui sera appelé, peu importe le type du pointeur. Du coup, dans le cas de ptrBase->ToString(), c'est bien la méthode de la classe ClasseDerive qui sera appelé, pas celle de ClasseDeBase, pourtant type du pointeur.

Le résultat à l'écran sera celui-ci :

```

ClasseDeBase: 1           ← résultat de base.ToString();
ClasseDerive: 2 Compteur de base=0 ← résultat de derive.ToString();
ClasseDerive: 0 Compteur de base=3 ← résultat de ptrBase->ToString();

```

Ce mécanisme est très largement utilisé dans la classe Message et ses dérivées.

13. ANNEXE C – COMMUNICATION SÉRIE (STM32)

13.1 Envoi de données

L'envoi de données (par le STM32) est effectué dans la gestion d'interruption externe de l'accéléromètre, qui déclenche à 94 Hz. Afin de simplifier les envois, toutes les données sont envoyées dans une seule trame, qui mesure 37 octets en incluant les caractères de contrôle.

L'envoi de données de STM32 se fait grâce à l'envoi de trames par la méthode de division du nombre flottant en 4 octets sur le port série. Le décodage demande moins de calcul que pour des trames en ASCII. Pour éviter que les octets puissent prendre n'importe quelle valeur, les octets des données sont entourés par des caractères de contrôle.

Ces trames sont composées des champs suivants:

- début de trame: contient le caractère 'R', permet de reconnaître le début d'une trame
- fin de trame: contient le caractère 'X', permet de reconnaître le fin d'une trame
- paquet de donnée:
 - *début paquet*: contient le caractère '<' qui indique le début d'un paquet
 - *label*: contient un caractère qui permet d'identifier la grandeur associée aux informations du champ data
 - *data* : contient l'information envoyée
 - *fin paquet*: contient le caractère '\n' qui indique la fin d'un paquet
 - *fin de trame*: contient le caractère 'X', permet de reconnaître la fin de la trame

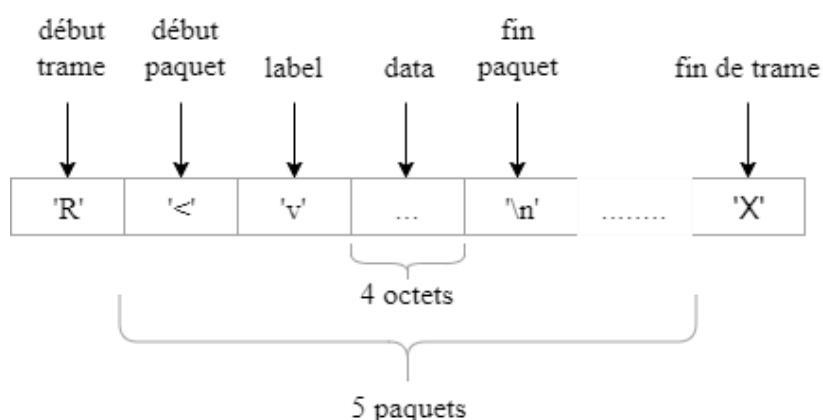


Figure 1 Exemple d'une trame

13.2 Réception de données

Les données sont reçues sous le format d'un paquet (7 octets), ce qui diffère du cas précédent. La détection d'un paquet complet se fait au début et à la fin de paquet (caractères '<' et '\n'). Ce choix vient du fait que l'envoi de consignes de Raspberry Pi se fait à la vitesse de 100 Hz, le temps

de traitement des messages est donc largement suffisamment. De plus, comme il existe d'autres threads de temps réel qui peuvent envoyer une trame d'urgence pour arrêter le gyropode, il est nécessaire de recevoir les informations paquet par paquet. Au niveau du STM32, la consigne de couple reçue du Raspberry Pi est convertie directement en courant (A). La conversion est faite en divisant par la valeur 0.80435, qui est le produit de K_m (le constant de couple de moteur DC(Nm/A)) et le rapport de réduction de moteur à la roue K_g .

13.3 Liste des labels utilisés dans les trames

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'
Consigne de couple	float	N.m	'c'
Arrêt	int	1 si arrêt d'urgence, 0 sinon	'a'

14. ANNEXE D – MISE EN PLACE D'UNE TRACE

Le projet propose une classe Trace pour permettre le suivi des éléments de synchronisation (mutex, sémaphore, taches) du programme. La classe fourni des méthodes permettant d'étiqueter une attente, ou la prise (d'un sémaphore, d'un mutex) et de tagger cette étiquette avec l'heure depuis le lancement de la trace.

Les méthodes renvoie toute un message, qui doit être ensuite envoyé vers l'interface graphique. Peut importe si ces messages ne sont pas envoyés immédiatement, ou par salve, le fait qu'ils soit horodaté suffit.

Ces messages n'apparaissent pas directement sur l'IHM, mais se rajoutent dans la fenêtre de log, pour une analyse après coup.

La classe Trace fourni les méthodes suivantes :

Nom de la méthode	Rôle	A utiliser
Message* WaitForMutex(RT_MUTEX* mut);	Indique que l'on attend la prise du mutex mut	Avant la prise du mutex
Message* MutexAcquired(RT_MUTEX* mut);	Indique que l'on a passé la prise du mutex mut	Après la prise du mutex
Message* MutexReleased(RT_MUTEX* mut);	Indique que l'on vient de libérer le mutex mut	Après la libération du mutex
Message* WaitForSem(RT_SEM* sem);	Indique que l'on attend la prise du sémaphore sem	Avant la prise du sémaphore
Message* SemEntered(RT_SEM* sem);	Indique que l'on a passé la prise du sémaphore sem	Après la prise du sémaphore
Message* SemSignaled(RT_SEM* sem);	Indique que l'on vient de libérer le sémaphore sem	Après la libération du sémaphore
Message* TaskEntered();	Indique que l'on vient de lancer la tâche courante	Au début de la tâche
Message* TaskNewIteration();	Indique que la tâche périodique vient de réitérer	Au début de la boucle de tâche
Message* TaskEnded();	Indique que la tâche courante va se terminer	A la fin de la tâche
Message* TaskDeleted(RT_TASK* task);	Permet de savoir si la tâche task est détruite	N'importe où, ailleurs que dans la tâche task

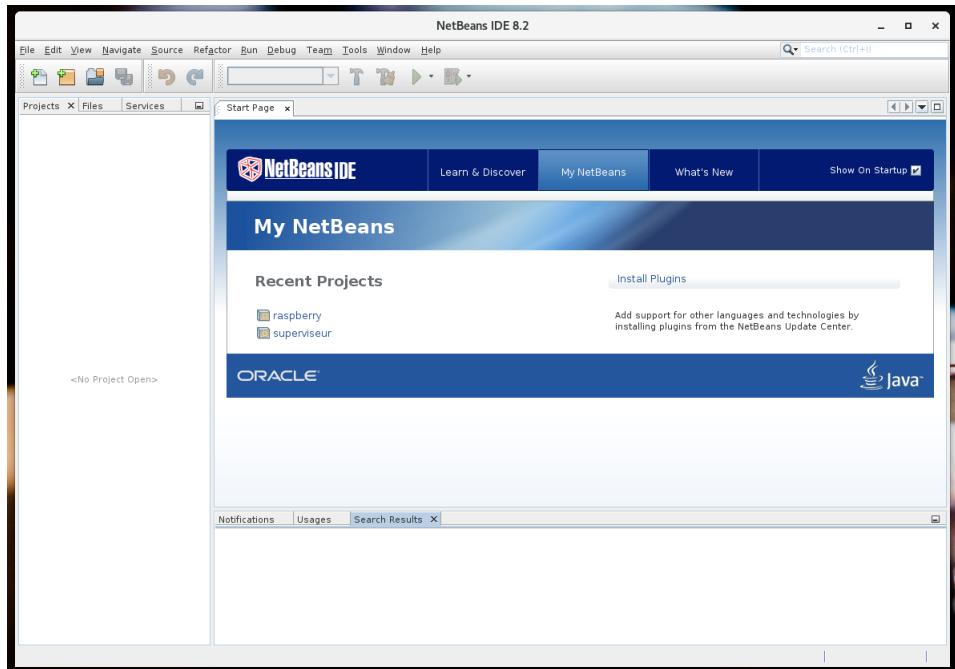
15. ANNEXE E – PRISE EN MAIN DE NETBEANS

Netbeans est un environnement de développement intégré (IDE), conçu initialement pour développer des programmes écrits en Java. Mais il se prête bien à la programmation en C++ et surtout, possède une fonction de compilation à distance permettant, dans notre cas, de compiler le code directement sur la raspberry.

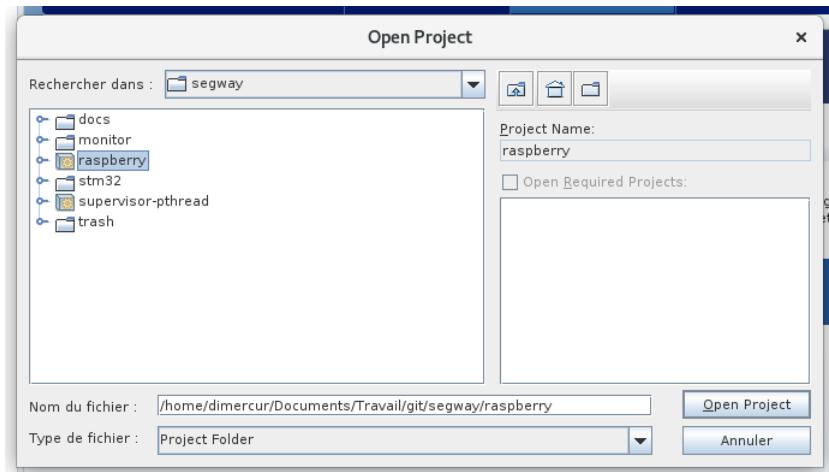
Le lancement de l'environnement se fait en ligne de commande (dans un terminal) en tapant :

```
netbeans
```

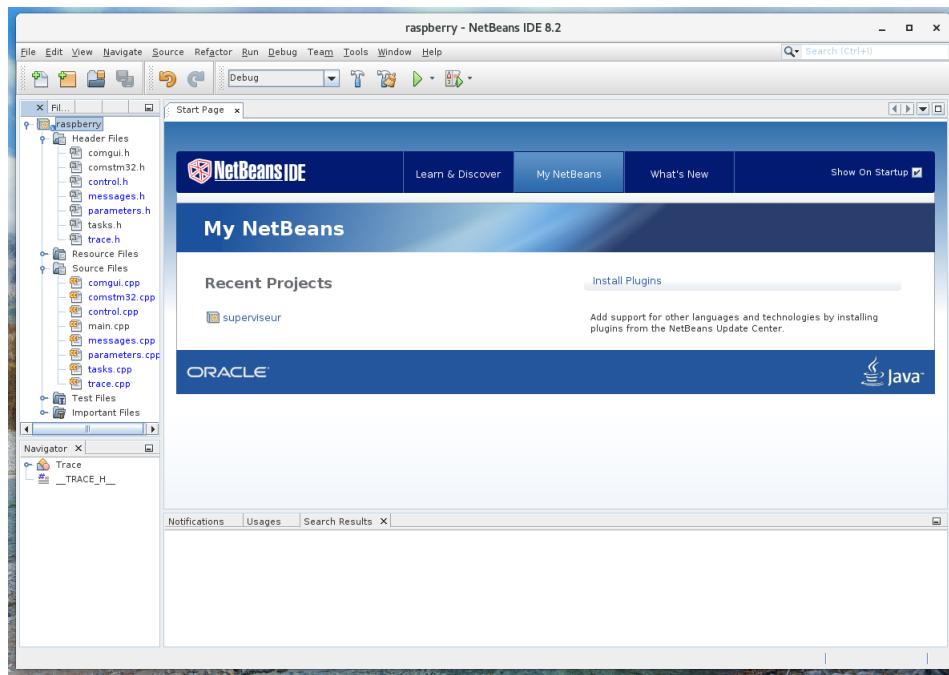
L'IDE apparaît alors comme suit :



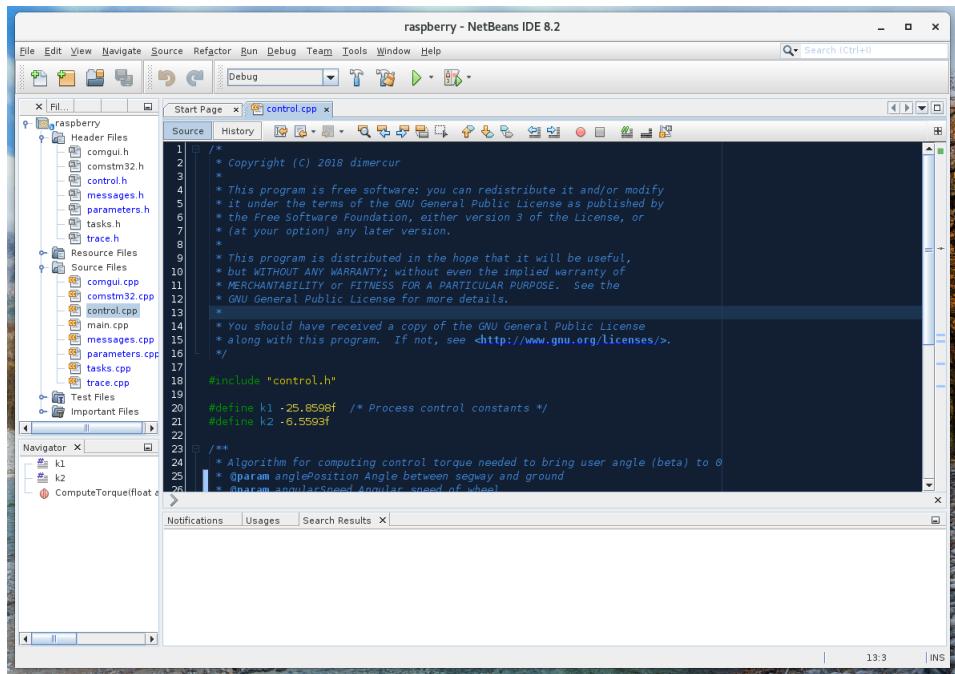
Pour ouvrir un projet, allez dans « File / Open Project » et sélectionnez le projet de votre choix (dans notre cas, il se trouve dans le dépôt git dans segway/raspberry). La fenêtre suivante s'ouvre :



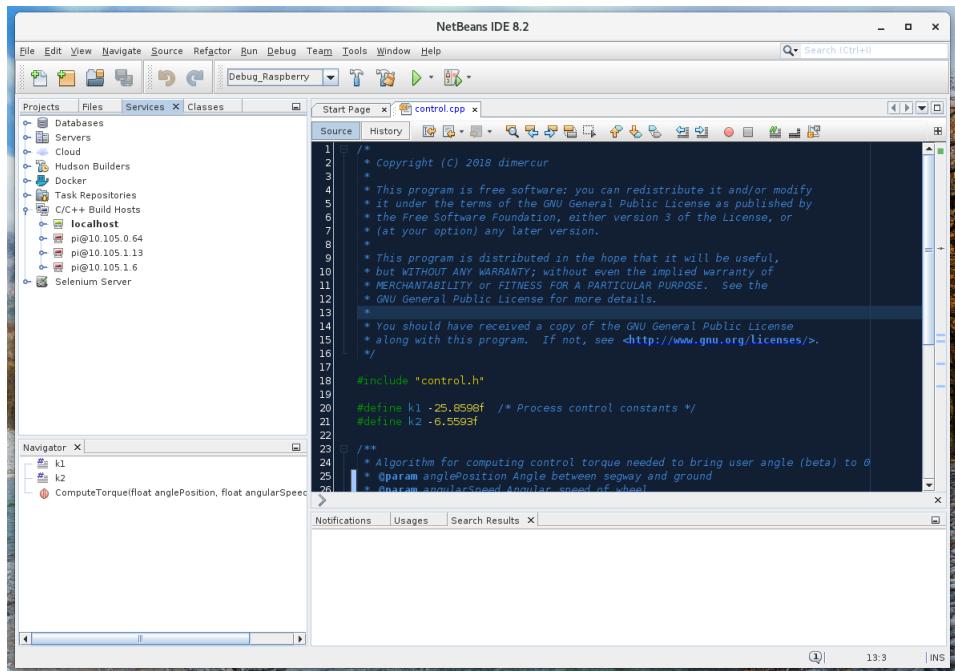
Cliquez sur « OpenProject » et l'environnement se charge avec le projet :



En cliquant (comme fait ici) en face de « Header Files » et de « Source Files », les fichiers constituant votre projet sont accessibles. Double cliquez dessus pour les ouvrir dans l'environnement.

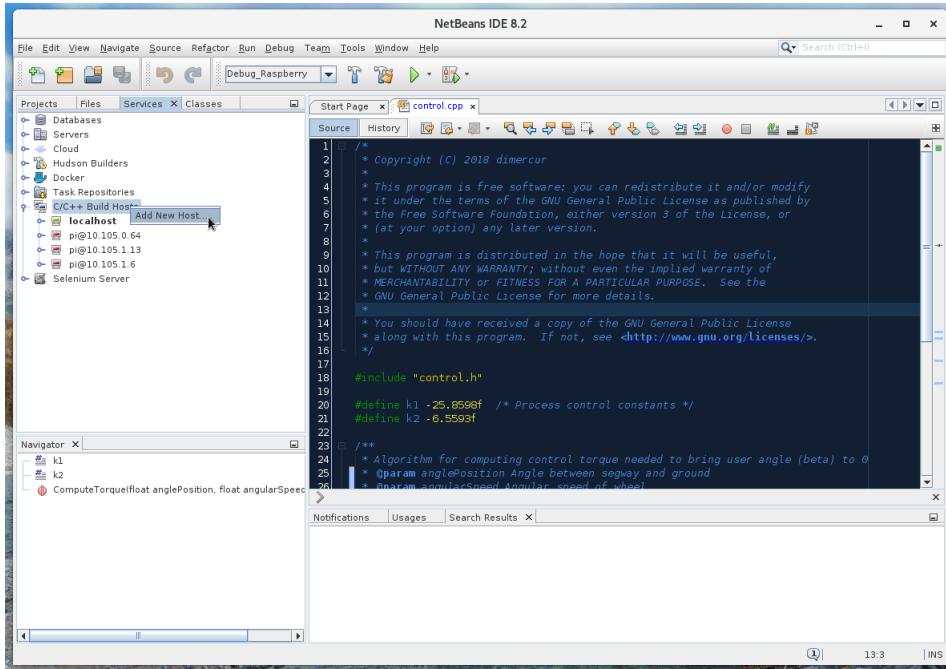


Pour pouvoir compiler sur raspberry, il faut lui rajouter une cible de compilation. Dans le bandeau de gauche (ou se trouve l'arborescence de votre projet) se trouve en haut des onglets. Le troisième se nomme « Services » (agrandissez le bandeau) et contient un champ nommé « C/C++ Build Hosts ». Ouvrez cette entrée :

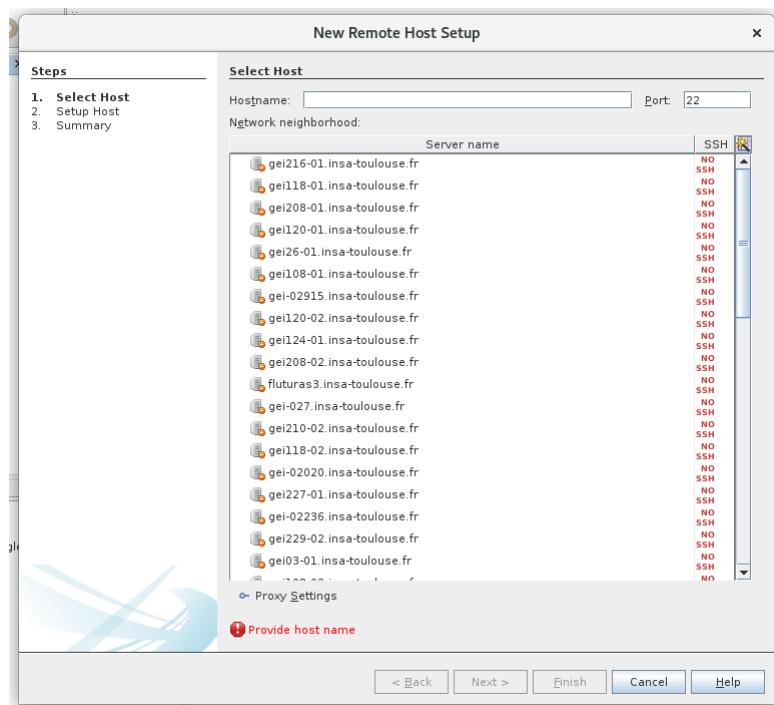


« localhost » correspond à votre machine de TP : c'est la cible par défaut . Les autres présentes dans cet exemple sont des raspberry déjà rajoutées. Pour en rajouter une de plus (votre

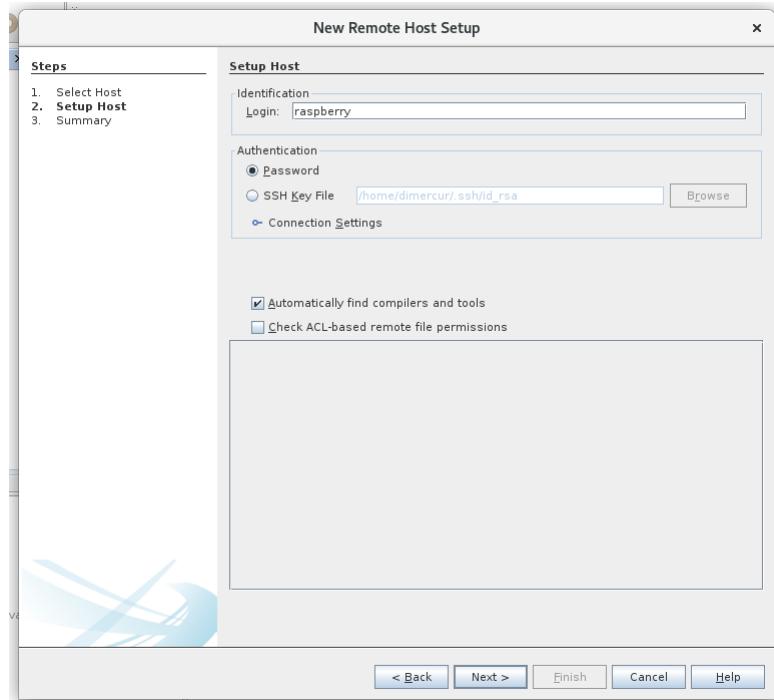
simulateur), cliquez sur « C/C++ Build Hosts » avec le bouton droit : le menu « Add new host » apparaît :



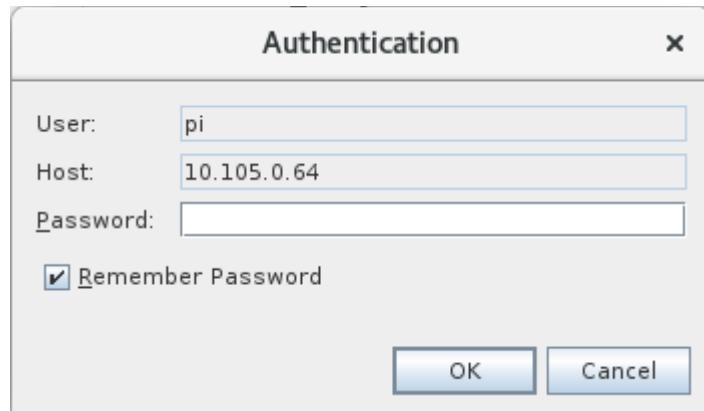
Cliquez dessus (bouton gauche) et la fenêtre suivante s'ouvre :



Dans le champ « Hostname » (en dessous de « Select Host ») saisissez l'adresse IP de votre simulateur (étiquette collée sur la raspberry) puis cliquez sur le bouton « Next » en bas de la fenêtre. Si la cible est démarrée et accessible, vous devriez obtenir ceci :

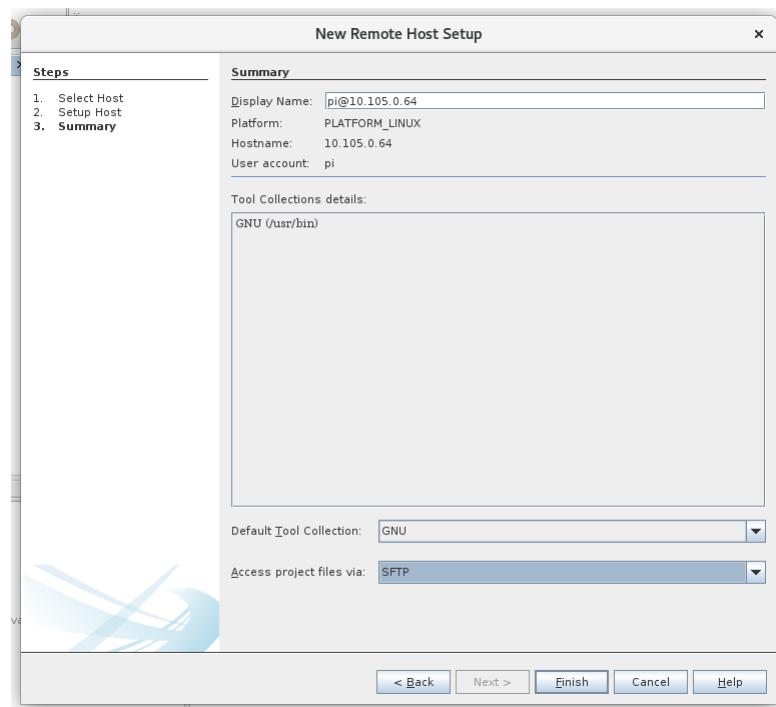


Changez le login par « pi » dans le champ identification. Netbeans demandera l'ouverture d'un portefeuille kdewallet pour mémoriser le mot de passe de connections : indiquez ici le mot de passe de votre compte INSA. Une fenêtre d'authentification sur la cible distante s'ouvre alors . Elle a cette forme là :



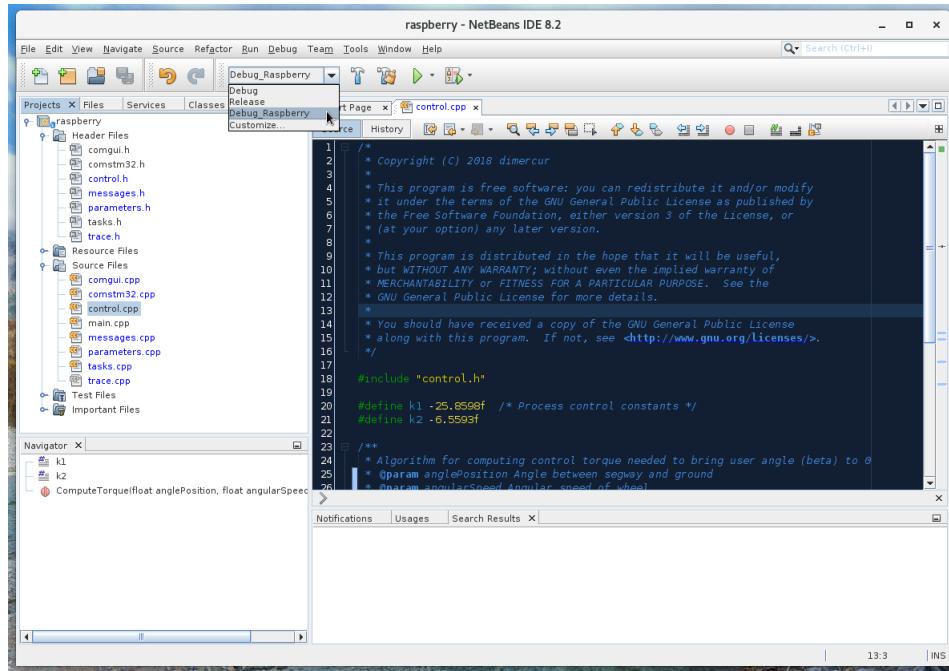
Le mot de passe est « raspberry ». Cliquez ensuite sur OK.

La fenêtre de configuration de la cible distante cherche alors le compilateur et change comme ceci :

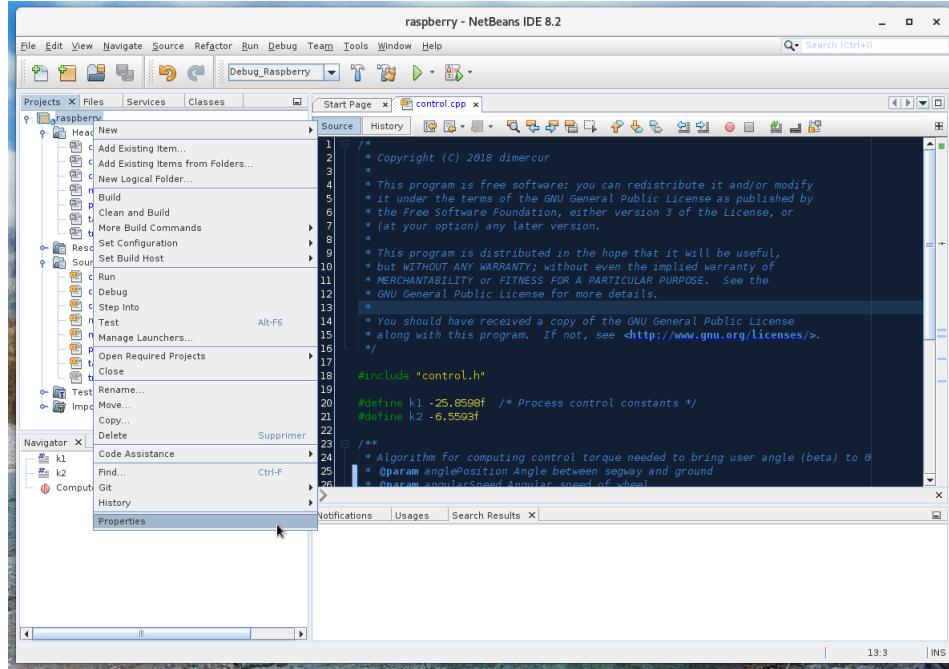


Sélectionnez alors « SFTP » dans le champ « Access project file via : » puis cliquez sur le bouton « Finish ». Votre cible est ajoutée !

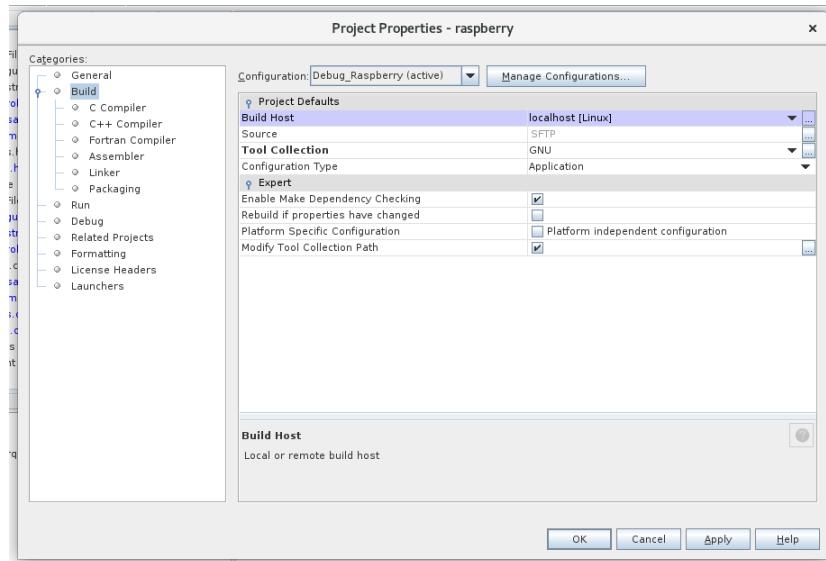
Il faut ensuite configurer la cible « Debug_Raspberry » de votre projet. Pour cela, cliquez sur la liste déroulante (en dessous des menus « Run » et « Debug » et à côté du marteau bleu) et sélectionnez « Debug_Raspberry ».



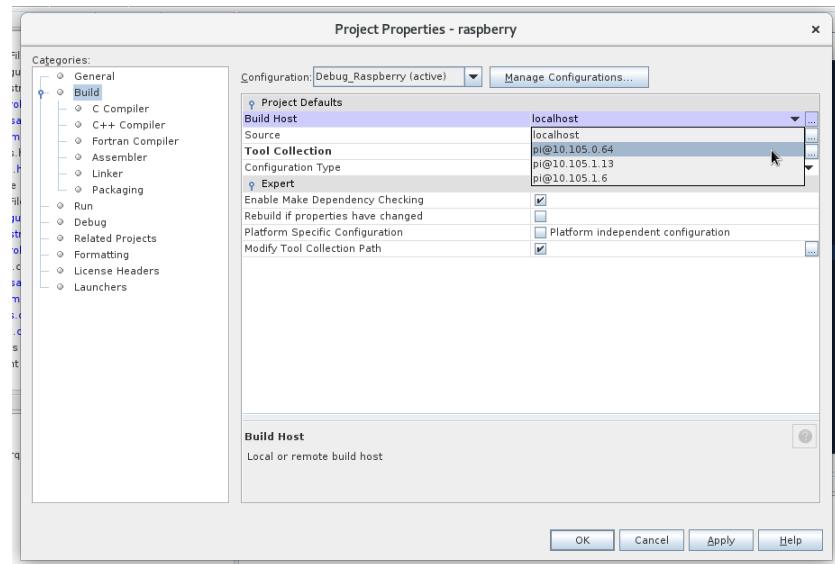
Ensuite, cliquez (bouton droit) sur « raspberry » dans l'arborescence de votre projet (tout en haut) et dans le menu, choisissez « Properties ».



La fenêtre de propriétés du projet s'ouvre. Dans l'arborescence de gauche cliquez sur « Build ».



Sélectionnez la ligne (en violet ici) « Build Host » et déroulez la liste déroulante à droite (contenant initialement « localhost »). Choisissez votre cible distante et cliquez sur le bouton « Ok » en bas de la fenêtre.



Voilà, votre projet peut maintenant compiler sur votre cible distante.

Table des matières

1. Introduction et vue d'ensemble.....	2
2. Lexique.....	3
3. Vue globale du système.....	4
4. Le prototype INSA de gyropode.....	5
5. Illustration de la démarche d'analyse sur le système gyropode.....	8
5.1 Besoin et exigences.....	8
5.2 Architecture logique.....	8
5.3 Architecture organique.....	9
	9
6. Le programme de supervision.....	10
6.1 Variables échangées entre le STM32 et le superviseur.....	10
6.2 Trace du programme de supervision.....	11
6.3 Réinitialisation de l'état d'arrêt d'urgence.....	11
7. Le simulateur de gyropode.....	12
7.1 Simulation physique du gyropode.....	12
7.2 Simulation des variables du système.....	13
8. Le moniteur.....	15
9. Sujet de TD.....	16
9.1 Arrêt d'urgence.....	17
9.2 Présence User.....	18
9.3 Surveillance batterie.....	18
9.4 Envoyer.....	18
9.5 Communication.....	18
9.6 Asservissement.....	19
10. Consignes de TP.....	20
11. Annexe A – Diagrammes de classe.....	23
11.1 Classes principales (temps réelles).....	23
11.2 Classes de messages.....	23
11.3 Classes de communication et de trace.....	24
12. Annexe B – Rappels de C++ et programmation objet.....	25
12.1 Définition et déclaration.....	26
12.2 Visibilité.....	27
12.3 Méthodes statiques.....	27
12.4 Méthodes virtuelles et polymorphisme.....	28
13. Annexe C – Communication série (STM32).....	30
13.1 Envoi de données.....	30
13.2 Réception de données.....	30
13.3 Liste des labels utilisés dans les trames.....	31
14. Annexe D – Mise en place d'une trace.....	32
15. Annexe E – Prise en main de Netbeans.....	33