INSA TOULOUSE

Rapport projet tutoré Segway

Projet tutoré du S8 de 4IR-I

Alexandre Benazech, Vincent Pera et Romain Rivière

Table des matières

Introduction

Ce projet consiste à mettre en place un système de pilotage embarqué sur un segway afin de piloter ce dernier par les informations remontées par un STM32. L'objectif principal de ce projet tutoré est qu'il devienne un sujet de Travaux Pratiques en 4ème année option Ingénierie Systèmes.

Ce projet a été réalisé sous la tutelle de Madame Claude BARON et Monsieur Ion HAZIUK. Il a de plus été réalisé avec la forte collaboration de Monsieur José MARTIN, Monsieur Emmanuel LOMBARD et Monsieur Lucien SENANEUCH.

Présentation du système

Nous avons commencé le projet avec une approche ingénierie système afin de formaliser la finalité et la mission que doit accomplir notre système, les exigences auxquelles notre système est soumis, ainsi que le contexte dans lequel le système va se trouver. Le but est de balayer, de la manière la plus exhaustive possible, le contexte et le fonctionnement du système que nous devions développer.

Finalité, Mission, Objectifs

Finalité

La finalité du système SP est de contrôler le déplacement et la stabilité d'un gyropode en fonction de la demande du conducteur. Il doit aussi assurer la sécurité du conducteur et du système.

Mission

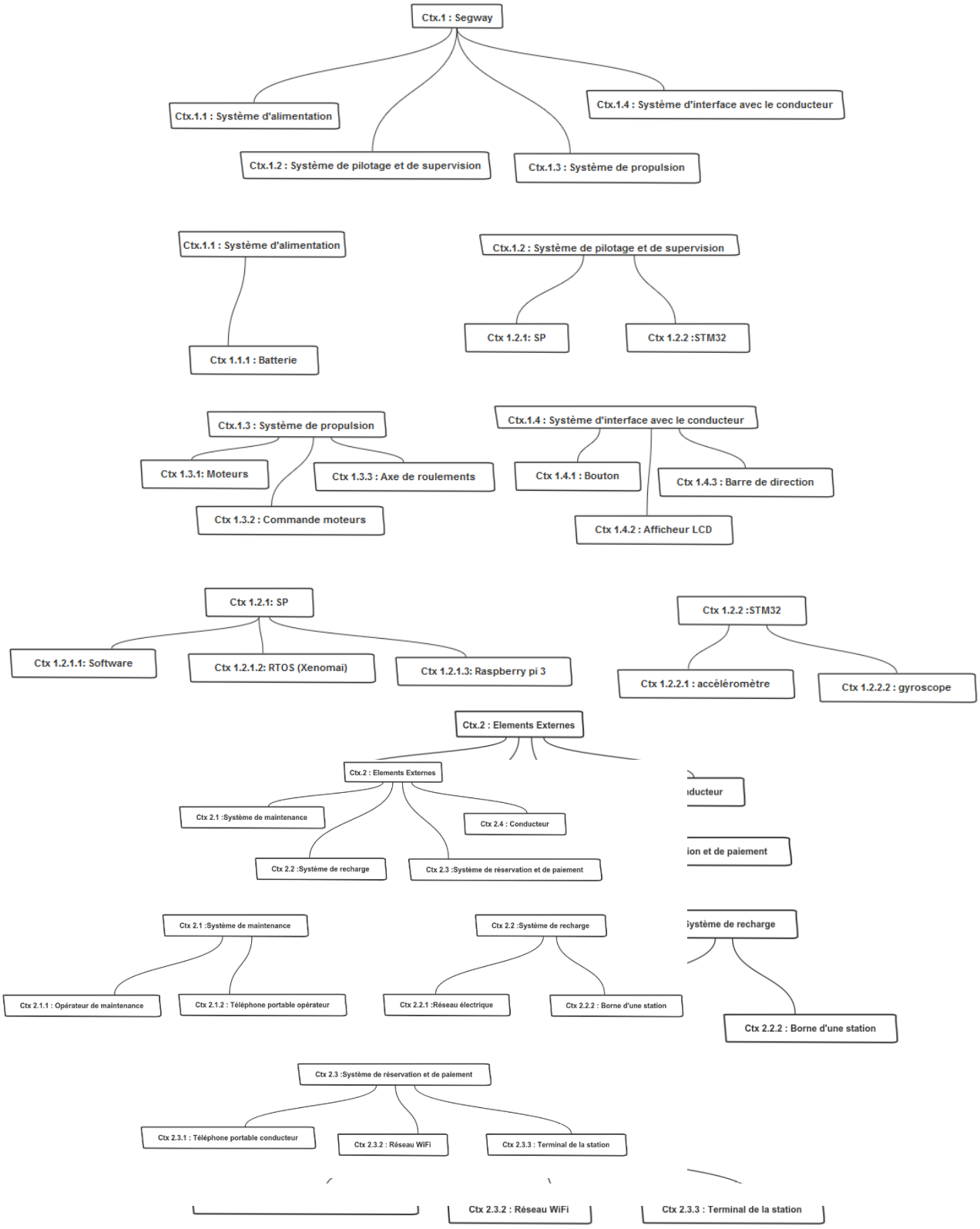
Le système SP génère la commande à partir des données physiques propres au segway (angle et accélération du segway) et envoie une consigne de deux couples vers les deux moteurs par un processeur intermédiaire. Il assure également la sécurité en stabilisant le segway et supervisant son état et son fonctionnement.

Objectifs

- Limiter le couple développé par les moteurs
- S'assurer que le système respecte un temps de réponse convenable
- Limiter « l'angle de rotation » du segway
- Limiter la vitesse du segway
- Offrir une interface intuitive à l'utilisateur
- Permettre la connexion au système via un réseau sans fil

Contexte d'utilisation du système

Contexte organique



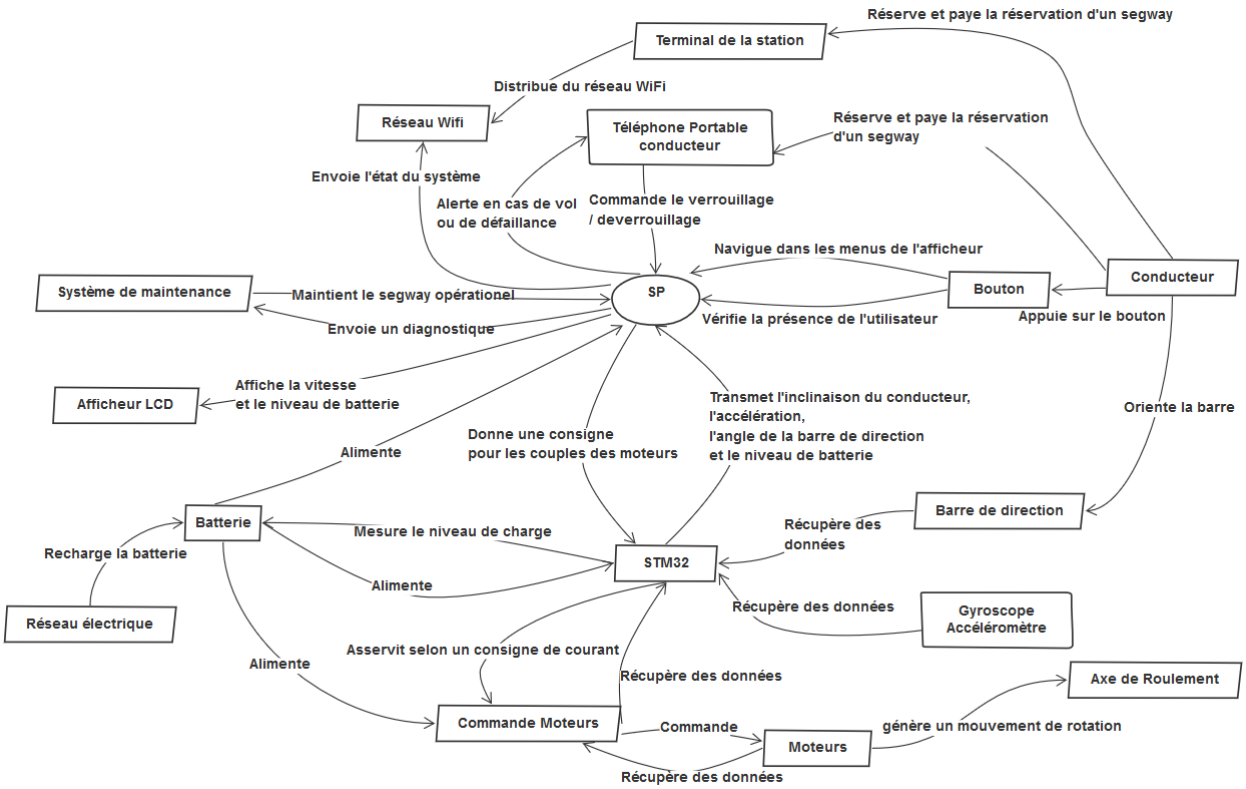
Arborescence Organique du contexte, niveau 2

Liste des constituants du contexte :

Identificateur	Constituant	Description
Ctx. 1.1	Système d'alimentation	Système alimentant l'ensemble des constituants du segway
Ctx. 1.1.1	Batterie	Élément central du système d'alimentation permettant d'assurer son fonctionnement
Ctx. 1.2	Système de pilotage et de supervision	Système permettant au segway d'assurer sa fonction principale tout en supervisant son état
Ctx. 1.2.1	SP	Système permettant de contrôler le déplacement du segway ainsi que sa stabilité en fonction des commandes de l'utilisateur
Ctx.1.2.1.1	Software	Logiciel qui implémente l'ensemble des fonctions que doit remplir SP
Ctx. 1.2.1.2	RTOS (Xenomai)	OS sous lequel fonctionne SP
Ctx. 1.2.1.3	Raspberry pi 3	Carte représentant la partie matérielle de SP
Ctx. 1.2.2	STM32	Système permettant de faire l'asservissement de la consigne de courant pour les moteurs et de remonter vers SP les informations concernant le segway
Ctx. 1.2.2.1	Accéléromètre	Capteur retournant la valeur de l'accélération du segway
Ctx. 1.2.2.2	Gyroscope	Capteur retournant la valeur du mouvement angulaire du segway
Ctx. 1.3	Système de propulsion	Système permettant de donner du mouvement au segway
Ctx. 1.3.1	Moteurs	Moteurs courant continu permettant de générer un mouvement de rotation différent pour chacune des roues
Ctx. 1.3.2	Commande Moteurs	Consigne de couple donnée aux moteurs
Ctx. 1.3.3	Axe de Roulement	Les axes de roulement permettent la transmission du mouvement de rotation

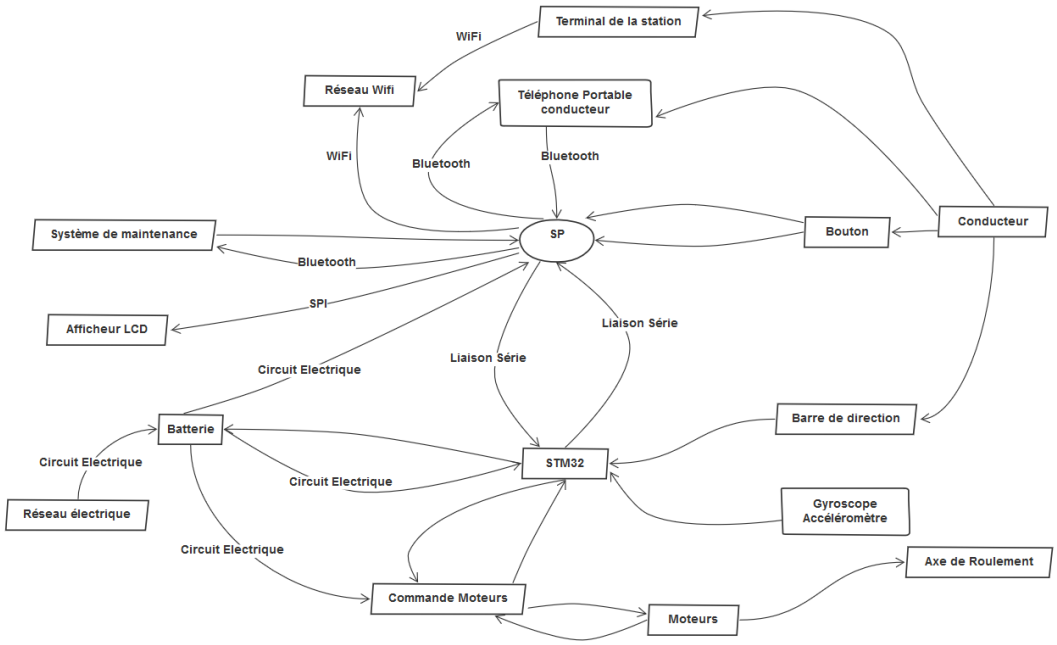
		généré par un moteur à la roue qu'il lui est associée
Ctx. 1.4	Système d'interface avec le conducteur	Ensemble des interfaces permettant au conducteur d'interagir avec le segway
Ctx. 1.4.1	Bouton	Bouton maintenu par le conducteur afin de prouver sa présence lors de la conduite du segway
Ctx. 1.4.2	Afficheur LCD	Affichage des éléments importants (vitesse, niveau de batterie) à destination du conducteur
Ctx. 1.4.3	Barre de direction	Barre permettant au conducteur de transmettre ses consignes de direction
Ctx. 2.1	Système de maintenance	Système permettant l'entretien périodique, les réglages et la réparation du segway
Ctx. 2.1.1	Opérateur de maintenance	Personne en charge du système de maintenance
Ctx. 2.1.2	Téléphone portable opérateur	Outil pour que l'opérateur puisse interagir avec le segway
Ctx. 2.2	Système de recharge	Système permettant la recharge de la batterie du Segway
Ctx. 2.2.1	Réseau électrique	Distribue l'énergie nécessaire pour la charge de la batterie
Ctx. 2.2.2	Borne d'une station	Permet de connecter un segway au réseau électrique et de lancer sa recharge
Ctx. 2.3	Système de réservation et de paiement	Permet à un utilisateur de réserver un segway et de payer cette réservation
Ctx. 2.3.1	Téléphone portable du conducteur	Permet la réservation et le paiement via une application, peut être lié au segway réservé et commande le verrouillage de ce dernier
Ctx. 2.3.2	Réseau WiFi	Permet d'effectuer la réservation et son paiement en ligne
Ctx. 2.3.3	Terminal de la station	Permet d'effectuer la réservation et son paiement (en CB) au niveau de la station

Contexte Fonctionnel



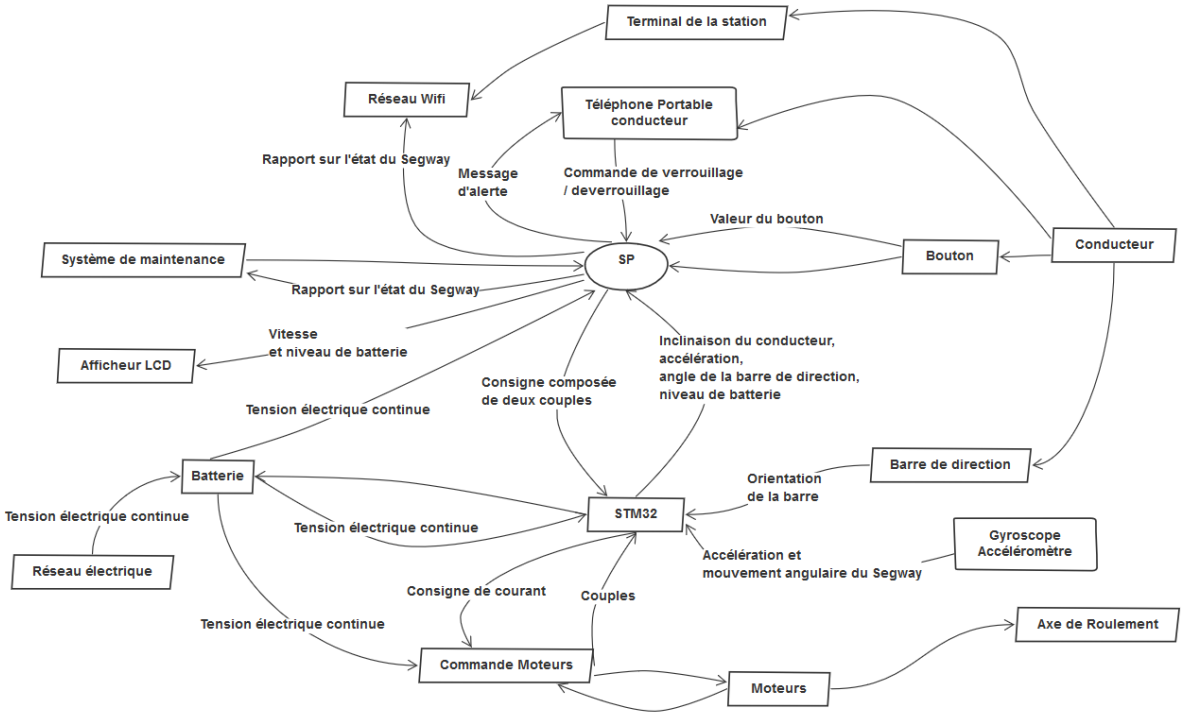
Organigramme du contexte fonctionnel

Contexte des Interfaces



Organigramme des interfaces

Contexte des données



Organigramme des données

Liste des parties prenantes

Identificateur	Partie Prenante	Commentaire
PP.1	Fabricant du segway	Intègre la solution SP dans ses segways
PP.2	Opérateur de Maintenance	Entretient, règle et répare le segway
PP.3	Usager du segway	Pilote le segway
PP.4	Autres usagers de la route	Partage l'environnement avec le segway
PP.5	Opérateur de la sécurité routière	Respecter la réglementation
PP.6	Société exploitant le segway	Propriétaire du segway
PP.7	Gestionnaire du réseau électrique	Pour la recharge des batteries
PP.8	Gestionnaire du réseau Bluetooth	
PP.9	Concepteurs de SP	Equipe s'occupant du sous-système SP
PP.10	Individu malveillant	Personne qui peut voler, ou déplacer le segway sans autorisation
PP.11	Autorités de régulation des réseaux sans-fil	

Exigences techniques :

Exigences fonctionnelles :

Identificateur	Exigence
E.F.0	SP génère la commande à partir des données physiques du segway pour pouvoir le piloter
E.F.1	Envoie la consigne obtenue des deux couples vers les deux moteurs
E.F.2	Assure la sécurité de l'utilisateur
E.F.2.1	Vérifie le maintien du bouton placé sur la barre
E.F.2.2	Limite la vitesse du segway
E.F.2.3	Limite l'angle de rotation du segway
E.F.3	Vérifie le bon état de marche du segway
E.F.3.1	Vérifie en continue le niveau de charge de la batterie
E.F.3.2	Vérifie le bon fonctionnement des deux moteurs
E.F.4	Envoie un message d'alerte au système gestionnaire
E.F.5	Répond aux ordres envoyés par l'utilisateur via le Bluetooth
E.F.5.1	Répond à l'ordre de verrouillage et vérifie que le système répond correctement
E.F.5.2	Répond à l'ordre de déverrouillage et vérifie que le système répond correctement
E.F.5.3	Vérifie que l'utilisateur qui se connecte au Bluetooth est bien celui qui a commandé le segway

Exigence de performance :

Identificateur	**Exigence**
E.P.1	Génère la commande pour les deux moteurs toutes les 20 ms (cadence égale à 50Hz)
E.P.2	Vérifie la présence de l'utilisateur à une cadence de xHz
E.P.3	Vérifie le niveau de batterie à une cadence de 1Hz
E.P.4	Vérifie la présence du bon utilisateur via le Bluetooth à une cadence de xHz

Exigences d'interface fonctionnelle

Identificateur	**Exigence**
E.IF.1	Interfaces fonctionnelles de données
E.IF.1.1	SP envoie à la STM32 une consigne pour les couples des moteurs gauche et droit
E.IF.1.2	SP reçoit, de la part du STM32, l'inclinaison du segway, l'accélération et le niveau de la batterie
E.IF.1.3	SP envoie une alerte sur le téléphone portable de l'utilisateur en cas de vol ou de défaillance
E.IF.1.4	SP reçoit du téléphone de l'utilisateur la commande de verrouillage ou de déverrouillage du segway
E.IF.1.5	SP envoie un diagnostic du segway au système de maintenance par le Bluetooth

E.IF.1.6	SP reçoit du système de maintenance, par Bluetooth, l'autorisation ou non d'être utilisé par un potentiel utilisateur
E.IF.1.7	SP envoie au réseau wifi local l'état du système actuel
E.IF.1.8	SP reçoit l'information par un appui continu sur les boutons l'information de présence de l'utilisateur
E.IF.1.9	SP reçoit l'information par un appui répété sur le bouton droit ou gauche, tout en gardant l'autre appuyé, l'information de changement dans le menu de l'afficheur
E.IF.1.10	SP envoie sur l'afficheur LCD via un câble SPI la vitesse et le niveau de batterie
E.IF.2	Interfaces fonctionnelles électriques
E.IF.2.1	SP est alimenté avec une tension électrique continue fournie par la batterie principale

Exigences d'interface physique

Identificateur	**Exigence**
E.IP.1	Interfaces physiques des données
E.IP.1.1	SP est relié au STM32 via un bus de données
E.IP.1.2	SP est relié au téléphone portable de l'utilisateur via un échange de données Bluetooth
E.IP.1.3	SP est relié au système de maintenance par un échange de données Bluetooth
E.IP.1.4	SP est relié au réseau wifi local des bornes via un échange de données Wifi
E.IP.1.5	Le SP est relié aux boutons du guidon par un câble de transport de signaux électriques
E.IP.1.6	SP est connecté à l'afficheur LCD par un câble SPI
E.IP.2	Interfaces physiques électriques
E.IP.2.1	SP est connecté à la batterie principale par un câble

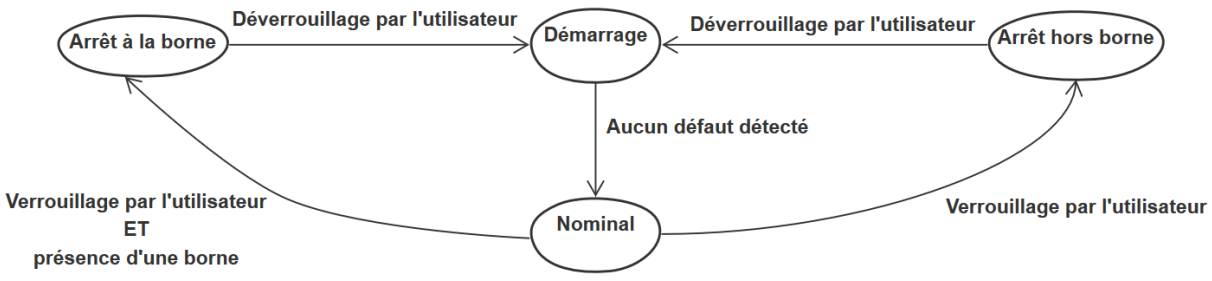
Exigences d'environnement

Identificateur	**Exigence**
E.Env.1	SP fonctionne sur la plage de température suivante 5°C-30°C
E.Env.2	SP fonctionne sur la plage d'humidité relative suivante 0% - 70%
E.Env.3	SP fonctionne seulement dans des conditions ensoleillées (ne résiste pas à l'eau)
E.Env.4	Sp fonctionne suite à des chocs mineurs contre le segway (petit trottoir, graviers, branches)

Exigences d'utilisation :

Identificateur	Mode	Commentaires
M.1	En arrêt	Le système de pilotage ne gère plus les liaisons avec les moteurs, il ne communique qu'avec la STM32 pour obtenir les informations des gyroscopes ainsi que du niveau de la batterie
M.1.1	En arrêt à la borne	Ecoute en permanence sur le Bluetooth pour acquérir les ordres provenant du smartphone de l'utilisateur et veille sur la recharge de la batterie suivant son niveau
M.1.2	En arrêt hors borne	Le système communique avec le STM32 afin de détecter un mouvement suspect du segway et envoie un message d'alerte au système gestionnaire pour prévenir du problème. Ecoute en permanence sur le Bluetooth pour acquérir les ordres provenant du smartphone de l'utilisateur
M.2	En fonctionnement	Le système donne une commande aux moteurs en fonction des consignes données par l'utilisateur, ce qui permet le déplacement du segway. Il assure la sécurité de l'utilisateur en limitant la vitesse du segway et son angle de rotation. Il vérifie la présence de l'utilisateur (grâce au bouton placé sur la barre). Il vérifie également le niveau de la batterie ainsi que les pannes potentielles, et si détection d'une défaillance, envoie un message d'alerte au système gestionnaire pour prévenir du problème. Ecoute le Bluetooth en permanence afin de contrôler si l'utilisateur est le bon.
M.3	Démarrage	Le système vérifie qu'il est relié au bon utilisateur Bluetooth, il vérifie aussi le bon état de fonctionnement du segway afin de détecter d'éventuelles pannes qui seraient survenues lors de son arrêt.

Ces dernières exigences de fonctionnement nous donnent 4 modes dont le passage peut être décrit avec la machine à état suivante :



Machine à état des modes de fonctionnement

Par défaut, le système commence dans l'état Arrêt à la borne. Puis, lorsqu'il reçoit une demande de déverrouillage, il passe dans l'état de Démarrage. Dans cet état de contrôle, si aucun défaut n'est

détecté, le système passe automatiquement dans l'état de fonctionnement nominal. Lorsque le conducteur arrête le segway et le verrouille, le système repasse dans un des deux états d'arrêts.

Un dernier état dit état KO n'est pas mentionné sur cette machine à état. Dès lors qu'un défaut est détecté dans n'importe quel des autres états, le système se retrouve dans l'état KO, dans lequel le segway va se retrouver complément bloqué.

Configuration

Installation de xenomai

L'étape d'installation de xenomai fut cruciale dans le déroulement de ce projet. En effet nous voulions mettre en place un Xénomai 3.0.3 Cobalt (en architecture co-noyau) sur un Raspberry Pi 3, et nous ne disposions pas d'image prête à l'emploi à flasher sur une carte SD. Nous avons dû nous-même créer cette image disque.

Il nous a donc fallu tout d'abord compiler un noyau linux patché avec Xenomai Cobalt pour Rasberry Pi 3. Pour cela deux choix s'offraient à nous :

- Cross-compiler le noyau depuis un pc ayant comme OS un environnement linux. Cette méthode peut sembler intéressante mais présente certains inconvénients. Il est nécessaire de disposer d'un toolchain de cross-compilation installé et bien configuré sur la machine effectuant la cross compilation. Ce toolchain permet de compiler des programmes binaires pour une architecture différente. Ce toolchain peut varier d'une machine à une autre et est assez compliqué à configurer.
- Compiler le noyau directement depuis un Raspberry Pi 3. L'avantage est que comme tout se fait depuis le Raspberry Pi 3. Nous ne sommes pas dépendants de la machine utilisée lors de la configuration du noyau, le même tutoriel peut donc s'appliquer à un utilisateur sous n'importe quel OS. Cependant comme la compilation se fait directement sur le Raspberry Pi 3, le temps de calcul est considérablement plus long que sur un PC fixe.

Après plusieurs essais infructueux de la première méthode nous nous sommes tournés vers la seconde solution afin de compiler notre noyau. Nous avons suivi le tutoriel de Christophe Blaess disponible ici (https://www.blaess.fr/christophe/2017/03/20/xenomai-sur-raspberry-pi-3-bilan-mitige/). Nous allons résumer les grandes lignes de la méthode à appliquer mais un tutoriel complet est disponible en annexe.

Le point de départ de l'installation de xenomai est l'usage d'une image raspbian. Raspbian est une version de Debian qui est optimisée pour le hardware du Raspberry Pi. Nous utilisons la version Lite, qui est la version minimale et la plus légère de raspbian. Cette image, une fois copiée sur une carte microSD est directement bootable par un Raspberry Pi.

On se connecte en SSH au Raspberry Pi 3, on y télécharge la version 3.0.3 de Xenomai ainsi que le patch ipipe. On applique ce patch à Xenomai ainsi qu'un second patch fourni par Christophe Blaess. On compile ensuite le noyau directement depuis le Raspberry Pi 3. Le Raspberry n'ayant pas une puissance de calcul importante, cela prend un certain temps. Même en utilisant les 4 cœurs durant la compilation, ce qui est peu recommandé si on ne dispose pas d'un système de refroidissement performant, la compilation a pris plus d'une heure.

On redémarre, on compile les bibliothèques Xenomai et si tout s'est bien passé le système est opérationnel et peut faire tourner du code en temps réel dur sur le noyau Cobalt.

La liaison série

Initialement il était prévu d'utiliser un bus SPI entre le Raspberry et le STM32. Cependant nous avons été heurtés au problème suivant : il est compliqué de faire du vrai full-duplex sur un bus SPI.

Le bus SPI repose sur une communication maitre esclave. Le maitre active le périphérique auquel il veut parler, génère une horloge et envoie son message. L'esclave doit répondre immédiatement après que le maitre ait fini de parler. Pour que l'esclave puisse répondre le maitre doit continuer à générer l'horloge. Cela nécessite que le maitre envoie des données qui ne sont pas interprétées par l'esclave pendant ce temps. Cela implique aussi que l'esclave ne peut pas prendre la parole de manière autonome.

Pour ces différentes raisons le bus SPI a été écarté au profit d'une liaison série.

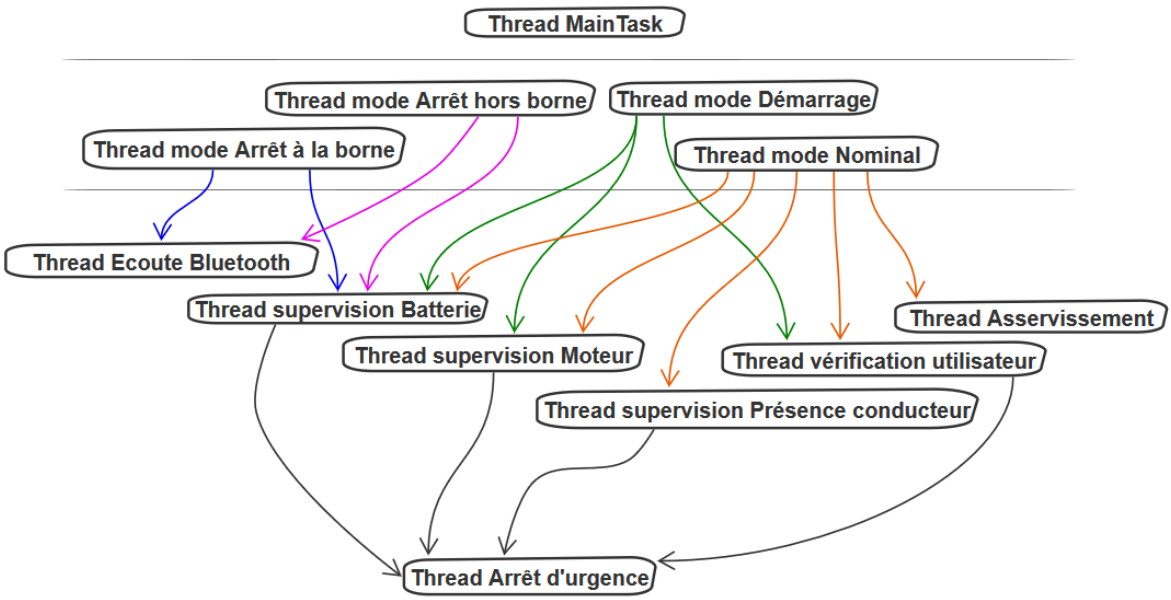
Le problème rencontré ensuite est que le driver de la liaison série en temps réel (le driver 16550A), n'est pas encore fonctionnel sur le Raspberry Pi 3. Après plusieurs essais d'installation, avant la compilation dans le fichier de compilation ou après la compilation du noyau nous nous sommes résignés à utiliser les services de linux pour accéder à la liaison série.

Cela implique que notre tâche de communication avec le STM32 passera systématiquement en secondary mode et n'est donc plus prioritaire du tout. Au niveau des performances ce n'est pas très grave car la charge de travail de notre processeur est assez faible. Cela veut dire que la tâche d'envoi de données via le port série a toujours le temps de s'exécuter dans des délais très raisonnables. On observe très peu de variations de latence liées à ce passage en secondary mode, mais ils ne perturbent pas la stabilité du système.

Architecture logicielle développée

Pour assurer les fonctionnalités de SP, nous avons développé une architecture logicielle qui est embarquée sur la Raspberry PI 3 et qui utilise les primitives offertes par Xenomai.

L'architecture logicielle choisie



Arborescence de l'architecture développée

Nous avons donc développé une architecture à 3 niveaux.

Le niveau 3 qui correspond au niveau le plus bas est constitué de threads qui réalisent des tâches très précises. Ils ont donc vocation à être en concurrence les uns avec les autres. Parmi tous ces threads de bas niveaux, il faut noter que les threads de supervision (Batterie, Moteur, Présence Utilisateur et Contrôle de l'utilisateur) peuvent, si une défaillance est détectée, appeler le thread d'arrêt d'urgence.

Le niveau 2 est constitué de 4 threads, chacun correspondant à un état de la machine à état précédente. Ils n'ont pas vocation à se retrouver en concurrence. Ces threads se contentent de lancer les threads de niveaux inférieurs.

Le niveau 1 est constitué du thread Main qui permet de naviguer entre les modes et donc de clôturer et lancer les threads des modes en fonctions de certaines conditions.

Voici un tableau récapitulatif pour tous ces threads :

Nom du thread	Priorité	Commentaires
Main Task	0	
Arret hors borne	99	
Arret à la borne	99	
Démarrage	99	
Nominal	99	
Ecoute Bluetooth	30	Écoute le Bluetooth afin de récupérer des demandes de l'utilisateur
Supervision Batterie	30	Contrôle le niveau de batterie
Supervision Moteur	30	Contrôle les défauts moteurs
Présence Conducteur	30	Contrôle la présence du conducteur quand le segway est en mouvement
Vérification Utilisateur	30	Contrôle que ce soit le bon utilisateur sur le Segway
Asservissement	50	
Arret d'urgence	70	

Le code applicatif prévu initialement

Dans un premier temps, les détails de l'architecture complète prévue lors de la phase de conception sera présentée.

Avant toute chose, il est nécessaire de présenter les variables qui seront stockées dans la mémoire partagée. Evidemment, à chaque accès (lecture ou écriture) sur une variable placée en mémoire partagée, il faut demander le mutex correspondant puis le relâcher une fois la variable utilisée.

Liste des variables :

- Int mode : représente le mode de fonctionnement (0 = arrêt total, 1 = arrêt à la borne, 2 = démarrage, 3 = en fonctionnement, 4 = Arrêt hors borne).
- Int arret_def : signifie un arrêt total du segway quand sa valeur est 1
- Liste des adresses Bluetooth autorisées pour la communication
- Variable condition change_mode : Permet de notifier au Main le changement de mode
- Et diverses variables pour les informations du système (niveau batterie, angle, accélération, couples, …)

Nous allons présenter maintenant les différents threads avec leurs contenus.

Thread Main Task :

- Phase d'initialisation : Initialise la liste des adresses Bluetooth autorisées, change_mode=0, mode=0 (Arrêt total), arrêt_def = 0.
- Affectation de la valeur 1 à mode (passage dans le mode Arrêt à la borne), et lancement du Thread « Arrêt à la borne ».
- Boucle tant que arrêt_def = 0, faire :
    - Sauvegarde du mode actuel dans une variable locale (save_mode).
    - Attente sur la variable condition change_mode.

- Quand nous sommes débloqués (change_mode vaut alors 1), ferme le thread correspondant au mode précédent (connu grâce à save_mode). Puis, lancement du thread correspondant au mode qui suit, connu grâce à la variable mode (qui entre temps est mise à jour dans les autres threads).
- Repasse change_mod à 0 puis reboucle.

Thread Arrêt à la borne :

- Lancement du thread d'écoute Bluetooth (tâche asynchrone qui ne se termine pas).
- Passe le thread de supervision de la batterie en mode périodique (fréquence 1Hz).
- Lancement du thread pour la supervision de la batterie.

Thread Démarrage :

- Vérifie que le Bluetooth de l'utilisateur qui a loué le segway soit bien visible dans l'environnement proche.
- Lance le thread de supervision de batterie (de manière asynchrone, pour que la vérification se fasse une seule fois).
- Fait la même chose pour le thread de supervision du moteur
- SI aucun arrêt d'urgence n'est déclenché, cela veut dire que le mode nominal peut être enclenché. On affecte donc la valeur 3 à la variable mode et la variable condition change_mod passe à 1 pour que le changement soit effectif.

Thread mode Nominal (mode en fonctionnement) :

- Passe la tâche Asservissement en périodique (f=50Hz).
- Passe la tâche supervision Batterie en périodique (f=1Hz).
- Passe la tâche supervision Moteur en périodique (f=10Hz).
- Passe la tâche supervision présence Conducteur en périodique (f=10Hz).
- Passe la tâche vérification Utilisateur en périodique (f=1Hz).
- Lancer l'ensemble des threads dans l'ordre suivant : Asservissement, supervision présence, supervision Batterie, supervision Moteur, vérification Utilisateur.

Thread Arrêt hors borne :

- Lancement du thread d'écoute Bluetooth (tâche asynchrone qui ne se termine pas).
- Passe en périodique le thread supervision batterie (f=1Hz).
- Lancement du thread supervision batterie.

Thread Asservissement :

Ce thread est bloquant sur la réception d'une trame provenant du STM32 et contenant les données nécessaires au calcul. Une fois reçue, la trame est ensuite décomposée puis les données sont stockées dans des zones de mémoire partagées.

Le thread va ensuite effectuer les traitements nécessaires pour calculer les commandes de couple.

Les nouveaux couples calculés sont stockés dans la mémoire partagée. On construit enfin une nouvelle trame qui est envoyée vers le STM32.

Thread Ecoute Bluetooth :

Ecoute en continue sur le « port » Bluetooth de la Raspberry.

Lorsqu'un message entrant est récupéré, il est traité.

On s'intéresse à l'adresse expéditrice du message. Si l'adresse n'est pas dans la liste des adresses autorisées, le message est alors ignoré.

Sinon, un traitement adéquat est effectué. Il faut noter que lorsque l'utilisateur loue le segway, son adresse Bluetooth est ajoutée à la liste des adresses autorisées pour ce segway. Cet ajout doit rester néanmoins temporaire et prendre fin à la fin de la location.

Hormis le cas du gestionnaire principal qui permet justement cet ajout, 2 autres cas peuvent se présenter.

- Réception d'un message de l'utilisateur pour déverrouiller le segway =>

  Passage au mode « en fonctionnement » (affectation de la valeur 2 à la variable mode) puis passage de la variable condition à 1 afin que le changement soit effectif.

- Réception d'un message de l'utilisateur pour le verrouillage du segway.

  Vérifier si le segway est en contact avec une borne ou non.

  Si oui => passage en mode « Arrêt à une borne » (affectation de la valeur 1 à la variable mode).

  Si non => passage en mode « Arrêt hors borne » (affectation de la valeur 4 à la variable mode).

  Dans les deux cas, la variable condition change_mod passe à 1 pour que le changement soit effectif.

Thread supervision Batterie :

Le raspberry envoie un message au STM32 pour lui demander les données sur la batterie. Puis attend la réception de la réponse.

Le message est alors décomposé puis la valeur de la batterie est placée dans une zone de la mémoire partagée.

Si le niveau de batterie reçue est en dessous de 5%, appel au thread Arrêt d'urgence.

Thread supervision Moteur :

Accède à certaines données placées en mémoire partagée, puis peut conclure en fonction des valeurs observées sur la présence d'un disfonctionnement moteur. Dans ce cas-là, appel du thread réalisant la procédure d'arrêt d'urgence.

Thread supervision Présence Conducteur :

Les boutons de présence sont directement câblés sur la Raspberry, le thread va récupérer leurs états dans deux variables locales.

Si l'un des deux boutons n'est pas à son état haut et que la valeur enregistrée précédemment d'accélération du segway est non nulle (donc segway en mouvement), appeler le thread réalisant la procédure d'arrêt d'urgence.

Thread Vérification Utilisateur :

Vérifier que le Bluetooth de l'utilisateur qui loue le segway est visible dans l'environnement proche. Si ce n'est pas le cas, appel du thread d'arrêt d'urgence.

Thread arrêt d'urgence :

Envoie au STM32 des consignes de couples calculées avec des valeurs négatives d'angle afin de freiner rapidement le segway. Dès que l'accélération est nulle, la variable change_mod passe à 1, la variable mode passe à 0 et arret_def passe à 1.

Le code applicatif développé

Concernant le code applicatif développé, il est extrêmement proche du code applicatif prévu à l'origine à l'exception que nous ne possédons malheureusement pas de segway physique pour faire communiquer notre système de pilotage. Tous les threads de supervision et de vérification décris au-dessus sont actuellement créés dans le code fourni. N'ayant pas de segway physique, leur contenu affiche dans le Log et dans l'invite de commandes ce qu'ils sont censés faire.

Les threads de mode (démarrage, arrêt à la borne, arrêt hors borne et nominal) sont implémentés et lancent les threads de supervision et d'action nécessaires. Le thread main est aussi implémenté et permet de changer de mode lorsque le changement est demandé par l'un des modes, ce changement s'effectue en changeant de valeur une variable locale protégé par un mutex et en libérant un sémaphore bloquant situé dans le main. Etant donné que nous ne pouvions pas simuler les vérifications sur le segway, nous avons décidé arbitrairement que nous changerons de mode au bout de 10 secondes afin de poursuivre dans un déroulement d'utilisation typique du segway lors de la séquence actuellement en place.

Le simulateur

Durant la fin du projet nous ne disposions pas d'un segway fonctionnel sur lequel tester nous aurions pu tester notre Raspberry Pi. Nous avons donc décidé de créer un simulateur sur un microcontrôleur qui remplacerait le STM32 et le reste du Segway.

Le modèle simulé

On nous a fourni le modèle physique linéarisé du segway. Ce modèle n'est valable qu'entre une inclinaison de -20° à +20° mais cela est suffisant pour nos tests. Ce modèle est donc basé autours de deux matrices A et B.

$$A = \begin{pmatrix} 1.003 & 0.02002 \\ 0.3166 & 1.003 \end{pmatrix}, B = \begin{pmatrix} -0.0002483 & 0.00312 \\ -0.02485 & 0.3122 \end{pmatrix}$$

Ces valeurs ont été calculées pour une fréquence d'échantillonnage de 50Hz. On souhaite donc calculer le vecteur x qu'on va envoyer au Raspberry avec la formule

$$\boldsymbol{x_{k+1}} = \boldsymbol{Ax_k} + \boldsymbol{Bu_k}$$

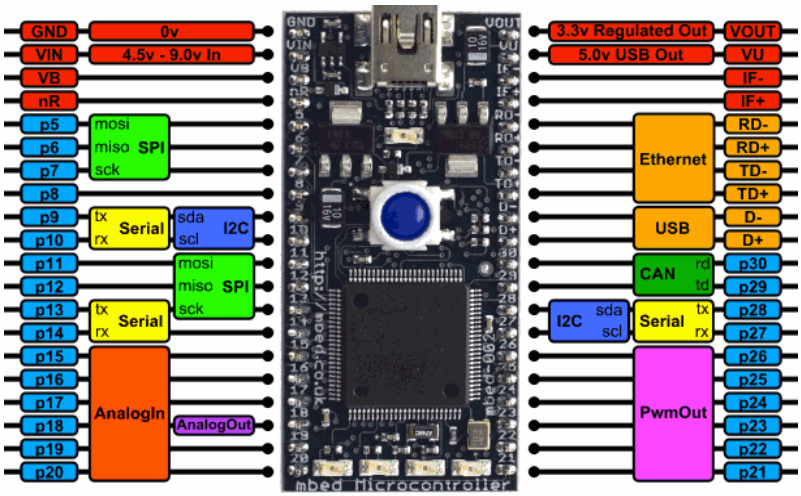
Sachant que $\boldsymbol{x} = \begin{bmatrix} \theta \\ \Omega \end{bmatrix}$ et $\boldsymbol{u} = \begin{bmatrix} \Gamma \\ \beta \end{bmatrix}$.

On a Θ l'angle du segway, Ω la vitesse angulaire, Γ le couple appliqué aux moteurs et β l'angle de l'utilisateur par rapport au guidon.

On envoie donc la nouvelle valeur de x au Raspberry qui doit calculer le couple $\Gamma = \boldsymbol{Kx}$ et le renvoyer au simulateur.

Le MBED

Comme les délais pour finir le projet étaient assez cours nous n'avons pas utilisé de STM32 car son utilisation des périphériques est complexe. Nous nous sommes donc tournés vers un MBED LPC1768 qui a une connectique très complète, qui est puissant (processeur ARM cortex M3 32 bits à une cadence de 96MHz), qui abrite un OS temps réel et qui dispose d'un environnement de développement haut niveau en C++.



Le MBED a deux rôles

- Simuler le comportement physique du segway grâce à un modèle linéarisé.
- Envoyer via une liaison série toute les 20ms l'angle et la vitesse angulaire du segway et attendre la réponse du Raspberry Pi 3.

Pour communiquer avec le Raspberry Pi on utilise les pins 9 et 10. Le pin 9 (tx) du MBED est connecté au pin 10 (rx) du Raspberry, le pin 10 (rx) du MBED est connecté au pin 8 (tx) du Raspberry. Enfin on connecte le pin 1 du MBED au pin 6 du Raspberry Pi pour avoir la masse commune.

On peut connecter un potentiomètre au MBED pour contrôler la différence d'angle entre la personne et le guidon du Segway. Pour cela on connecte la borne reliée au curseur au pin 20 et on connecte les deux autres pins à la masse et au +3.3V.

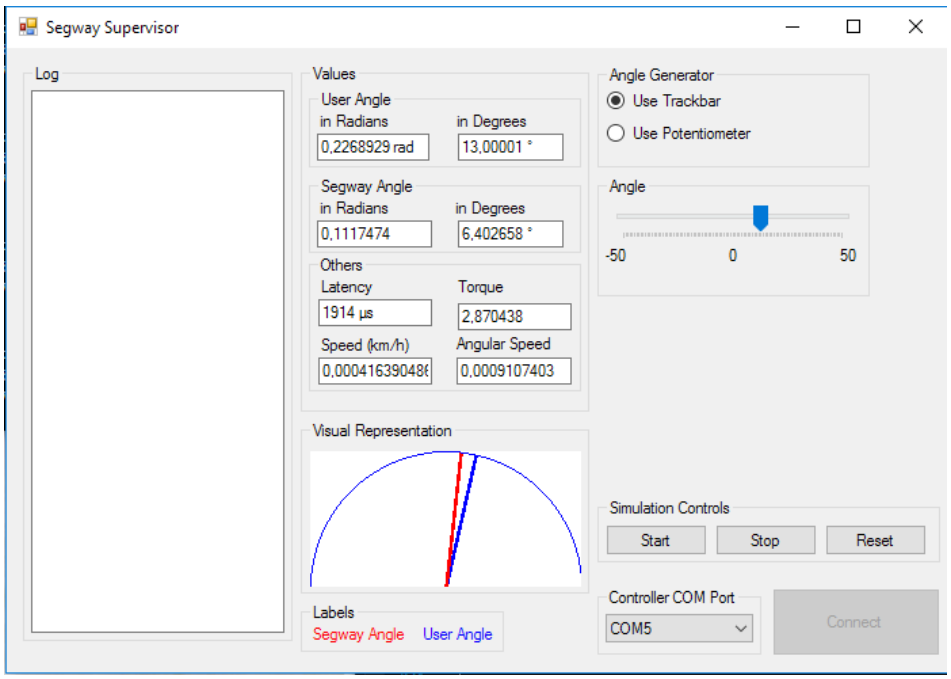
Le programme de supervision

Le MBED nous permet d'utiliser son port USB comme une liaison série avec un PC. Du point de vue du PC la liaison est vue comme un port COM. Cela permet le dialogue entre le PC et le MBED, et on lire et écrire vers le MBED grâce à une console série telle que PuTTY ou Tera Term par exemple.

Nous avons voulu une interface moins austère qu'une simple console pour superviser notre simulateur, c'est pourquoi nous avons développé une interface graphique de supervision. Nous avons utilisé le framework .NET et sa librairie Winform afin de créer une interface graphique. L'application a été créée en utilisant l'IDE Visual Studio.

Cette interface nous permet de nous connecter au MBED, de lancer et d'arrêter la simulation, de suivre des paramètres physiques du segway ainsi que de contrôler l'angle qu'a l'utilisateur avec le segway. On y visualise aussi la latence avec le Raspberry, c'est-à-dire le temps qui s'écoule entre l'envoi de l'angle et de la vitesse angulaire par le MBED et la réception de la nouvelle commande de couple envoyé par le Raspberry.

On dispose aussi d'une petite représentation graphique de l'angle du segway ainsi que l'angle de son utilisateur.



Communication entre les systèmes

Le Raspberry, le MBED ou STM32 et le PC communiquent tous via des liaisons séries. Les données échangées étant des nombres flottants nous avons eu un choix à faire vis-à-vis du format à utiliser pour les transférer d'un acteur à un autre.

- On peut les envoyer les valeurs en caractères ascii. Les trames seront directement lisibles mais seront longues et complexes à décoder. De plus une trame longue met plus de temps à se transférer qu'une trame courte et a donc un impact néfaste et non négligeable sur la latence du système.
- On peut diviser le nombre flottant en 4 octets et les envoyer sur la liaison série. Le décodage demande peu de calcul. Le problème est que ces octets peuvent prendre n'importe quelle valeur, y compris des caractères de contrôle. Il est donc nécessaire d'entourer octets du flottant par des caractères de contrôle.

C'est la deuxième solution qui a été gardée et nous avons donc mis en place un format de trames sur 7 octets commun à tout le système.

Une trame commence donc obligatoirement par le caractère '<'. Elle est ensuite suivie d'un caractère appelé le label. Son rôle est d'identifier la nature de la donnée qui suit. Par exemple si la valeur du label est *'t'* on sait que la valeur qui suit est le couple (torque en anglais). Après le label on trouve la charge utile du message. Généralement ce sera un nombre flottant mais il peut y avoir des exceptions. Enfin le dernier octet de la trame est toujours le retour chariot '\n'.

Cela peut nous donner une trame comme la suivante

'<' 'p' 'float sur 4 octets' '\n' qui véhicule la valeur de la position angulaire (label 'p')

Nous allons maintenant lister tous les messages qui peuvent circuler entre les différents éléments du système. On peut avoir jusqu'à 256 labels donc autant de types de messages. En pratique on en dispose d'un peu moins si on se limite aux caractères affichables de l'ascii étendu. Ce format a été pensé afin qu'il soit simple d'ajouter des types de messages comme par exemple la valeur de la batterie ou encore la vitesse linéaire du segway. Ils n'ont pas été implémentés car nous ne disposions pas de ces valeurs durant le projet. Néanmoins certains sont ajouté en grisé dans le tableau afin de fournir des exemples.

Dans les tableaux suivants sont indiqués en noir les messages qui ont été implémentés dans le système. On trouve en gris des messages qui n'ont pas été implémentés mais qui pourraient exister qui sont donnés comme exemples.

Communication entre le Raspberry Pi et le STM32/MBED

Envoi du STM32/MBED vers Raspberry Pi

Label	Type de message	Valeur transmise
p	Position angulaire	float
s	Vitesse angulaire	float
b	Niveau de batterie (en %)	float
v	Vitesse linéaire	float
f	Dysfonctionnement critique du système	Octet 0 : code d'erreur (char) Octets 1 à 3 vides

Envoi du Raspberry Pi vers STM32/MBED

Label	Type de message	Valeur transmise
t	Couple à appliquer	float

Communication entre le PC et le STM32/MBED

Envoi du PC vers le STM32/MBED

Label	Type de message	Valeur transmise
c	Message de contrôle	Octet 0 : 's' => démarrer simulation 'e' => arrêter simulation 'r' => redémarer simulation Octets 1 à 3 ignorés.
p	Utiliser le potentiomètre pour angle d'utilisateur	vide
t	Impose angle utilisateur (rad)	Valeur de l'angle (float)

Envoi du MBED/STM32 vers le PC

Label	Type de message	Valeur transmise
f	Position angulaire du Segway (rad)	float
s	Vitesse angulaire du segway (rad/s)	float
p	Angle entre l'utilisateur et le guidon du segway (rad)	float
t	Couple du segway	float
l	Latence du système (ms)	float

Bibliothèques développées

Au cours de ce projet nous avons développé deux bibliothèques qui pourront être utiles au développement d'applications sous Xenomai. La première est une bibliothèque qui permet l'utilisation du port série. La seconde est une bibliothèque permettant de générer une trace d'exécution d'un programme Xenomai dans un fichier.

Ces deux bibliothèques sont stockées dans le dossier lib du projet.

Nous avons fait le choix d'écrire ces bibliothèques en C alors que tout le projet est en C++ car nous voulons qu'elles soient réutilisables dans n'importe quel projet. Afin de les rendre compilable en C++ nous avons ajouté des macros dans les .h ajoutant l'attribut extern « C » avant les fonctions pour qu'elles soient compilées correctement.

Bibliothèque Serial

L'objectif de cette bibliothèque est d'ajouter une couche d'abstraction à l'utilisation du port série du Raspberry Pi 3. Les intérêts sont multiples. Tout d'abord elle permet de limiter les interactions de l'utilisateur avec le port série en utilisant des fonctions spécifiques. Grâce à cela on peut s'assurer que les messages transmis au STM32 sont au bon format. Cela permet à l'utilisateur de ne pas avoir à se soucier du format des messages sur la trame. En effet, on envoie des nombres flottants sur un bus qui ne peut transmettre qu'octet par octet. L'utilisateur final, qui ne sera peut-être pas un utilisateur

avancé du C/C++ n'aura peut-être pas les compétences nécessaires pour encoder et décoder correctement les messages, c'est pour cela que la bibliothèque le fait pour lui.

L'autre avantage majeur est qu'on masque la technologie utilisée lors de l'envoi de message via le port série. Pour l'instant nous avons utilisé, faute de mieux, les primitives de linux pour accéder au port série. Cependant, lorsque le driver temps réel pour la liaison série sera disponible pour le Raspberry Pi 3 il sera possible de changer le contenu de la librairie sans altérer le code de l'utilisateur pour faire la transition vers un usage du port série purement temps réel.

La bibliothèque met à disposition de l'utilisateur les fonctions suivantes :

- **int init_serial()** : initialise le port série afin de permettre l'émission et la réception de messages
- **message_serial read_from_serial** : fonction bloquante qui attend un message reçu sur le port série. Quand un message est reçu il est retourné dans une structure message_serial.
- **Int send_float_to_serial(float fl_value, char tag)** : permet d'envoyer des messages sur le port série. On passe la valeur, un nombre flottant et le tag associé à cette valeur et la fonction s'occupe de construire la trame et de l'envoyer.
- **Int close_serial()** : permet de fermer la liaison série.

La librairie introduit aussi la structure message_serial qui est composée d'un caractère label et d'un float value.

La bibliothèque Monitor

L'objectif de cette bibliothèque est d'écrire dans un fichier les différents évènements qui se produisent lors de l'exécution du programme xenomai. Elle a pour but de faciliter le débug. Tous les évènements ainsi que leur heure est écrite dans le fichier log.xenolog.

Elle propose les fonctions suivantes à l'utilisateur

- void log_task_deleted(RT_TASK * task)
- void log_task_entered()
- void log_task_ended()
- void log_sem_entered()
- void log_sem_waiting()
- void log_sem_signaled()
- void log_mutex_entered()
- void log_mutex_waiting()
- void log_mutex_released()

On dispose aussi des fonctions int init_recording() et void stop_recording() qui permettent de lancer et d'arrêter la production de la trace.

Bien que l'écriture dans un fichier ne puisse pas se faire en temps réel, l'utilisation de cette bibliothèque ne fait pas passer en secondary mode les tâches qui appellent ces fonctions. Quand une tâche appelle une des fonctions de la librairie, un message de log est placer dans un pipe. Une tâche

dédiée et très peu prioritaire va toute les secondes lire le contenu du pipe et écrire les évènements correspondants dans un fichier.

Si on n'a pas initialisé l'enregistrement des évènements avec init_recording() l'appel des fonctions de log est sans effet.

Il est envisageable d'améliorer cette bibliothèque en ajoutant de nouveaux évènements à logger. La syntaxe du fichier log.xenolog a été pensée pour ce que ce soit un fichier simple à parser, ainsi il serait possible d'écrire une application qui représenterait de façon graphique et plus lisible l'exécution du programme.

Conclusion

Nous tirons un bilan positif de ce projet. D'une part, nous avons vu de nouvelles choses, que ce soit l'approche système, ou le travail dans un environnement temps réel. D'autre part, malgré les difficultés rencontrées notamment pour passer Xenomai sur Raspberry et maitriser les technologies pour la liaison entre le STM32 et la Raspberry, nous avons réussi à mettre en œuvre nos idées pour faire fonctionner du code temps réel qui nous a permis de valider 2 points principaux de notre conception : l'Architecture avec le passage d'un mode à un autre et plus précisément la manière d'effectuer la génération de la consigne et son envoi. De plus nous avons appris à mener un projet en équipe, d'une part au sein de notre trinôme, mais aussi lorsque nous nous colaborions avec M. Martin, M. Lombard et M. Seunaneuch.