Rapport projet intégré

Antoine Aupée – Joachim Draps – Nathan Dwek

23 mai 2015

Table des matières

Chapitre 1

Introduction

Le but de ce projet intégré est de doter un robot d'un système de contrôle à distance, basé sur une transmission sonore. La base mécanique du robot est fournie, et celui-ci est déjà muni d'une première carte à microcontrôleur complète. Cette dernière permet d'une part de s'interfacer avec les moteurs et les encodeurs à partir du premier microcontrôleur, et d'autre part d'accéder à certaines pattes de celui-ci ainsi que des alimentations. Les autres blocs nécessaires doivent être conçus et implémentés par les étudiants et le code des microcontrôleurs doit être produit.

Ce rapport décrit la démarche, ainsi que les solutions techniques adoptées pour mener à bien ce projet. Dans un premier temps le cahier des charges doit être analysé pour diviser l'objectif final en différent sous-problèmes bien définis.

1.1 Analyse du cahier des charges

Les différentes fonctionnalités à développer, tirées du cahier des charges, sont les suivantes :

- Le robot doit être capable de se déplacer en ligne droite d'une distance donnée et d'effectuer une rotation d'un angle donné.
- Le robot doit être capable de recevoir et démoduler un signal audio FSK qui contient des instructions, à partir d'un microphone fourni.
- Le robot doit être capable d'interpréter et d'exécuter les instructions (déplacement en ligne droite et rotation) reçues.

Puisque le traitement numérique du signal audio est relativement intensif en calcul, il est suggéré d'utiliser un microcontrôleur supplémentaire pour effectuer celui-ci. Ceci permettra aussi d'illustrer une communication UART qui sera utilisée pour transmettre les instructions entre les deux microcontrôleurs. Dans la suite, nous appellerons le microcontrôleur permettant de s'interfacer avec les roues le microcontrôleur «propulsion» et l'autre le microcontrôleur «communication».

A partir de tout ceci, nous déduisons la liste des tâches à accomplir suivantes, correspondant chacune à un bloc bien délimité du robot :

- Développer une régulation de position et de position angulaire sur le microcontrôleur propulsion ;
- Dimensionner et implémenter une chaîne d'acquisition pour conditionner et numériser le signal audio à l'aide de l'ADC du microcontrôleur communication ;
- Développer un démodulateur numérique sur le microcontrôleur communication ;
- Configurer et développer une transmission UART entre les deux microcontrôleurs. Le niveau exact de traitement du signal avant d'être transféré au microcontrôleur propulsion doit encore être défini.
- Développer le bloc chargé d'interpréter les trames audio reçues et de fixer les paramètres de la régulation afin que celle-ci exécute la commande reçue ;

1.2 Structure de ce rapport

C'est dans cet ordre que ces blocs sont abordés dans ce rapport. Le chapitre 2 couvre la configuration des moteurs et encodeurs ainsi que le développement du régulateur et de ses consignes. Le chapitre 3 présente la chaîne d'acquisition et le traitement numérique du signal audio. Le chapitre 4 couvre la configuration des modules UART et le développement des émetteurs et récepteurs. Enfin, le chapitre 5 fait le lien entre la régulation et la réception des instructions : il présente la manière dont les paramètres utilisés pour générer les consignes de la régulations sont modifiés par une commande reçue. Dans le chapitre 6, nous conclurons en présentant l'aboutissement du projet.

Chapitre 2

Régulation du déplacement

Ce chapitre présente le bloc qui permet au robot d'exploiter les moteurs et les encodeurs afin de se déplacer. Pour cela, ceux-ci doivent être configurés avant d'être utilisés respectivement comme actionneurs et capteurs dans une boucle fermée de régulation, afin d'obtenir une bonne précision et une résistance aux perturbations. Dans la section suivante, les moteurs et encodeurs ainsi que leur configuration sont brièvement présentés ; la régulation sera ensuite couverte. [1]

2.1 Moteurs et encodeurs

2.1.1 Commande en PWM des moteurs

Les moteurs sont commandé en PWM. Comme le montre la figure 2.1, les paramètres importants à ce bloc sont les suivants :

- $3\,\text{ms} \leq T \leq 20\,\text{ms}$
- $1\,\text{ms} \leq T_{on} \leq 2\,\text{ms}$
- V_{mot} croît linéairement de $-V_{alim}$ à $+V_{alim}$ avec T_{on}
- $v_{max} \simeq 60\,\text{cm}\,\text{s}^{-1}$, $a_{max} \simeq 2\,\text{m}\,\text{s}^{-2}$
- Point de fonctionnement suggéré : $v_0 = 40\,\text{cm}\,\text{s}^{-1}$, accélération limitée à $50\,\text{cm}\,\text{s}^{-2}$

Le module output compare du microcontrôleur propulsion suffit à générer cette commande assez simple des moteurs ; il sera donc utilisé pour les contrôler. La configuration est opérée par la fonction `configPWM()`, qui initialise les paramètres suivants :

- Output mapping des ports des output compare sur les pattes physiquement reliées aux moteurs
- Choix du mode de PWM : impulsion haute commençant au début de la période et de longueur variable, fixée par `OCXRS`
- Choix du timer 2 comme valeur de comparaison

1. Ce chapitre couvre les fichiers `wheels.*` et `regul.*` du projet `propulsion.X`

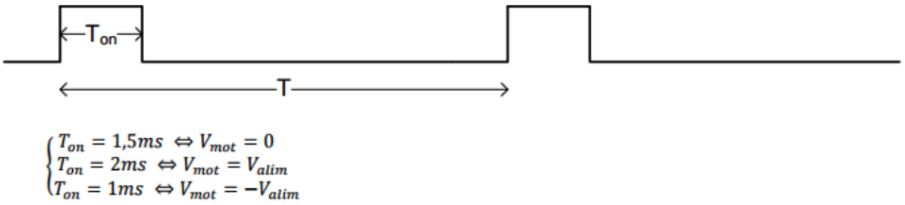

FIGURE 2.1 – Signal de commande des servomoteurs. [Source : Étude du déplacement du robot]

— Choix de $T_2 = 10\,\text{ms}$, et démarrage du timer 2

Une fois T_2 fixé, il est plus facile de parler en terme de rapport cyclique (D) qu'en longueur arbitraire. Puisque $T_2 = 10\,\text{ms}$, on a directement :

$$\begin{aligned} D = 0.1 &\Rightarrow V_{mot} = -V_{alim} \\ D = 0.15 &\Rightarrow V_{mot} = 0 \\ D = 0.2 &\Rightarrow V_{mot} = +V_{alim} \end{aligned}$$

La commande en tension des moteurs est donc finalement imposée par une simple instruction du type `OCXRS = D*PR2`. D sera donc la sortie du régulateur.

2.1.2 Encodeurs en quadrature

La rotation de chaque roue est mesurée par un encodeur en quadrature dont les paramètres importants sont les suivants :

— 90 impulsions montantes par tour de roue ;

— Pas de détection d'un tour de roue complet ;

La sortie des encodeurs est traitée par le module QEI du microcontrôleur propulsion. Le module est configuré comme suit par la fonction `configQEI()` :

— Input mapping des pattes physiquement reliées aux encodeurs sur les ports des QEI ;

— Choix du mode 4X sans index (pas disponible sur les encodeurs utilisé) : 360 impulsions par tour de roue ;

— Choix du maximum des compteurs à 360, et activation de l'interruption lorsqu'un compteur overflow, ce qui permet de reproduire le mécanisme d'index et de compter les tours de roues, afin de n'avoir virtuellement aucune limite sur les distances à mesurer (en cas d'utilisation prolongée, par exemple) ;

Les routines d'overflow sont extrêmement simples : `xSpins` est simplement incrémentée ou décrémentée en fonction de `QEIxCON.UDPN`.

L'interface des encodeurs pour la régulation est la fonction `readDistances` pour chaque roue, la distance parcourue en centimètres est donnée par :

$$\mathtt{d_x} = (\mathtt{xSpins} \cdot \mathtt{MAXxCNT} + \mathtt{POSxCNT}) \cdot \mathtt{CM\_PER\_TICK}$$

`readDistances()` sera donc la sortie du capteur de la régulation.

2.1.3 Validation des moteurs et encodeurs

L'implémentation de `wheels.c` est validée en fixant D à une certaine valeur ($\neq 0.15$) et en vérifiant que le robot se déplace bien et que les mesures de position des roues sont correctes. Les tests sont concluants, les encodeurs sont entièrement satisfaisants, mais les moteurs semblent posséder une zone morte [1] non négligeable qui varie en plus entre les deux roues et avec le temps [2]. Les moteurs sont aussi généralement dissymétriques, c'est à dire qu'à un même D donné correspondent des vitesses gauche et droite différentes. C'est bien sûr le rôle de la régulation de rejeter le mieux possible ces perturbations, comme nous allons le voir dans la section suivante.

2.2 Régulation

Avant d'aborder le cœur du problème, c'est à dire l'implémentation du régulateur, le code qui permet de fermer la boucle et d'utiliser le régulateur est d'abord présenté.

2.2.1 Configuration minimale

Les commandes des moteurs sont mises à jour à intervalles réguliers par la régulation, un timer est donc nécessaire. La configuration de la régulation est opérée par `configRegul()`, qui effectue les actions suivantes :

— Choix de la période du timer 1 ($f_{regul} = f_{OC} = 100\,\text{Hz}$) et démarrage de celui-ci
— Initialisation de certaines variables d'état et des consignes
— Activation de l'interruption du timer 1

C'est dans la routine d'interruption du timer 1 que toute le boucle fermée se déroule et que le microcontrôleur propulsion passe la plupart de son temps d'exécution. Les actions suivantes sont effectuées durant la routine de régulation :

— Lecture des encodeurs à l'aide de `readDistances()`
— Calcul des D par le régulateur
— Mise à jour des commandes par `OCxRS =` D_x`*PR2`

1. Celle-ci se manifeste même lorsque les roues ne sont pas en contact avec le sol, et l'inertie du robot n'est donc pas le seul facteur.
2. Et probablement la charge de la batterie

— Mise à jour des consignes
— Détection de l'arrivée du robot à sa position visée

Nous pouvons maintenant enfin aborder la conception du régulateur, et la génération de ses consignes.

2.2.2 Conception et implémentation du régulateur

Choix du schéma de régulation MIMO

Il est suggéré de mettre en place deux régulateurs proportionnels indépendants agissant sur la position des roues. Deux régulateurs indépendants constituent la solution la plus simple, tandis qu'une régulation de position permet de bénéficier d'un intégrateur et donc d'une erreur statique théoriquement nulle sur la vitesse. Le déplacement en ligne droite serait obtenu en donnant des consignes identiques aux deux roues, et la rotation en donnant des consignes opposées.

Cette régulation est cependant fortement dégradée par les zones mortes variables dont nous avons parlé à la section 2.1.3. Lorsque le robot est muni d'un tel régulateur, ces zones mortes se manifestent par un changement de direction du robot avant de commencer le déplacement en ligne droite. En effet, puisqu'un moteur sort forcément de sa zone morte (bien) avant l'autre, le robot commence par tourner autour de sa roue immobile avant de commencer à avancer. Ceci est l'effet le plus visible, mais on constate aussi que le robot dévie de sa trajectoire. Tout ceci ne peut pas être compensé par la régulation suggérée, qui assure simplement que les deux roues aient parcouru la même distance, mais pas forcément dans le même temps. Il y a plusieurs solutions à ce problème.

La première solution consiste à augmenter le gain de façon à ce que le temps d'établissement soit le plus court possible. En effet, c'est pendant cette durée que les effets de dissymétrie ont lieu (puisqu'après les positions sont par définition calées sur les consignes). Cependant, ceci fait apparaître des oscillations puis des instabilités et est limité par la saturation des moteurs. Même le gain le plus grand possible n'est pas suffisant pour compenser l'effet des dissymétries.

La deuxième solution consiste à essayer de caractériser le mieux possible les zones mortes et les dissymétries entre les deux moteurs, pour par exemple introduire un offset dans les commandes des moteurs et utiliser des gains différents sur les deux régulateurs. Cette alternative est limitée par la précision des mesures et du modèle choisi, et par la dérive des valeurs avec le temps et d'autres facteurs.

La troisième et dernière solution est beaucoup plus fondamentale et constructive : elle consiste à essayer de concevoir un régulateur qui, par design, est supposé être capable de rejeter les dissymétries entre les moteurs. Pour cela le régulateur opère non plus sur les positions séparées des deux

roues, mais sur la distance parcourue et la rotation du robot. Le déplacement en ligne droite est obtenu en donnant une consigne non nulle de distance parcourue et une consigne nulle de rotation, et vice-versa pour la rotation, ce qui a l'avantage d'être aussi un peu plus naturel. Les deux régulateurs sont toujours proportionnels, et on conserve la propriété d'intégration qui assure une erreur statique nulle. C'est cette régulation-ci que nous avons finalement choisi.

Implémentation du régulateur choisi

Pour calculer la distance totale parcourue, nous utilisons les relations suivantes :

$$v_l = \frac{v_{right} + v_{left}}{2}$$
$$\omega_z = \frac{v_{right} - v_{left}}{l_{axis}}$$

L'intégration est immédiate, d_l et θ_z s'expriment simplement à partir des lectures des encodeurs :

$$d_l(t) = \int_0^t v_l dt = \frac{d_{right} + d_{left}}{2}$$
$$\theta_z(t) = \int_0^t \omega_z dt = \frac{d_{right} - d_{left}}{l_{axis}}$$

Ces expressions sont implémentées par `readDistances()`.

L'implémentation du régulateur proportionnel est ensuite extrêmement simple et se trouve dans la fonction `setPWMs(float distance)`. Les gains sont trouvés expérimentalement. Ils sont différents selon que le robot est à l'arrêt, en rotation ou en déplacement en ligne droite. En effet, selon le mouvement en cours, un des k_p sert au suivi de consigne, tandis que l'autre sert à la réjection des perturbations, et est donc logiquement plus important [1].

Génération des consignes

Les consignes de distance parcourue et de rotation sont générées par intégration à partir d'un profil trapézoïdal de vitesse, comme le suggère la figure 2.2, extraite de l'étude du déplacement reçue. La fonction `updateConsignes()` génère les consignes. Il s'agit simplement d'une double intégration de l'accélération (angulaire) puis de la vitesse (angulaire), avec en plus une mise à zéro de l'accélération (angulaire) lorsque la vitesse dépasse la vitesse (angulaire) de croisière.

1. A l'arrêt, les deux gains servent bien sûr à la réjection de perturbations.

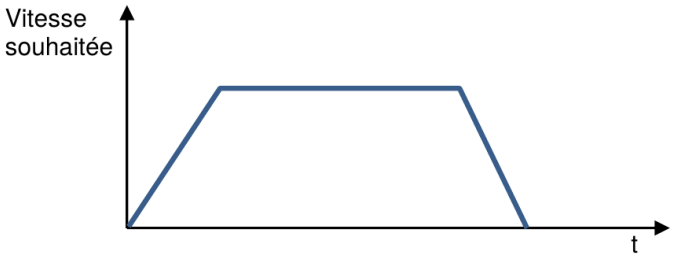

FIGURE 2.2 – Profil de vitesse trapézoïdal.[Source : Étude du déplacement du robot]

Détection de l'arrivée

La fonction `checkTerminalCondition()` est responsable de donner une valeur d'accélération (angulaire) opposée à la vitesse (angulaire) lorsque l'objectif est distant de la distance de décélération, et d'ensuite annuler la vitesse (angulaire) et l'accélération (angulaire) lorsque la vitesse (angulaire) change de signe. Le choix de la distance de décélération est plus détaillé dans la section 5.1.

2.3 Validation du bloc «déplacement»

Chronologiquement, nous avons implémenté les fonctions `straight(char distance)` et `rotate(char angle, char way)` de `decision.c`, présentées dans le chapitre 5, immédiatement après la régulation, notamment pour la tester. Il suffit donc d'appeler une de ces fonctions dans le main pour tester la régulation. Dans l'ensemble, celle-ci est satisfaisante, et permet d'effectuer tous les déplacements demandés et l'arrêt immédiat. On remarque cependant que la précision relative sur des courtes distances et sur les rotations [1] n'est pas très bonne. Ceci est dû à la précision limitée des encodeurs [2] mais surtout aux moteurs et leurs zones mortes, auquel s'ajoute le fait que la régulation garantit théoriquement une erreur statique nulle sur la vitesse, mais pas sur la position.

Nous disposons maintenant d'un bloc déplacement complet et fonctionnel, nous allons maintenant repartir «par l'autre côté» pour aborder la réception des instructions, en commençant donc par la chaîne d'acquisition audio.

1. Celles-ci demandent toujours des courtes distances parcourues par chacune des roues.

2. $\sim \frac{1}{360}$ de tour $= 0.09\,\text{cm}$, donne une borne absolue sur la précision, mais celle-ci n'est de loin jamais atteinte.

Chapitre 3

Réception et traitement du signal audio FSK

Le robot doit être capable de répondre à trois ordres différents : déplacement en ligne droite et rotation dans les deux sens. Cette information est transmise par FSK audio sous forme d'instructions de 10 bits qui contiennent le type de déplacement ainsi que le nombre (signé) de centimètres à parcourir ou l'angle de rotation en degrés. Dans ce chapitre, nous allons développer le chemin que parcourt l'information entre le moment où elle est envoyée sous forme sonore par l'émetteur qui nous a été fourni et la réception par le microcontrôleur. [1]

3.1 Codage de l'ordre

Le message envoyé au robot est codé sur 13 bits. Le premier et le dernier sont des `0` et sont respectivement le start bit et le stop bit : ils permettent de détecter le début et la fin d'une trame. Le bit 11 est un bit de parité pour la détection d'erreur. Les 10 bits restant contiennent l'information utile, les deux premiers donnent l'ordre suivant le codage suivant :

— `0b00` : avance tout droit
— `0b01` : tourne à droite
— `0b10` : tourne à gauche

Les 8 bits suivants représentent simplement le nombre de degrés ou de centimètres à parcourir.

L'information est modulée en FSK, et le signal audio est généré par un script Matlab ou python fourni. Les paramètres importants pour le dimensionnement de la chaîne d'acquisition et la démodulation sont repris

1. Ce chapitre couvre les fichiers `ADC.*`, `filter.*` et `peakDetector.*` du projet `communication.X`

ci-dessous :

$$f_{sym} = 10\,\text{Hz}\ (\tau_{sym} = 100\,\text{ms})$$
$$f_{HI} = 1100\,\text{Hz}$$
$$f_{LO} = 900\,\text{Hz}$$

La chaîne d'acquisition est constituée des différents étages suivants :
— Microphone ;
— Amplification ;
— Filtre de garde ;
— Convertisseur analogique-numérique ;
Le signal doit être ensuite démodulé à l'aide des éléments suivants :
— Deux filtres passe-bande ;
— Deux détecteurs de crête ;

Dans les deux sections suivantes, nous allons donc passer en revue le dimensionnement de la chaîne d'acquisition et le traitement numérique du signal audio, jusqu'à la sortie du comparateur. Le but final de ce bloc est donc de déterminer, à chaque période d'échantillonnage, si le signal comporte du 900 Hz et/ou du 1100 Hz

3.2 Dimensionnement de la chaîne d'acquisition

3.2.1 Montage du microphone

Le microphone fourni sort un signal d'une amplitude maximale de 1 mV. Il doit être polarisé par une résistance de 2.2 kΩ suivant le schéma de la figure 3.1.

3.2.2 Montage amplificateur

Comme nous l'avons dit précédemment, le signal en sortie de notre micro a une amplitude maximale de 1 mV. Le première priorité est donc de l'amplifier afin de préserver le plus possible le SNR pour la suite du traitement.

La plage de tension d'entrée de l'ADC va de 0 V à 3.3 V. Afin de l'utiliser le mieux possible tout en gardant une marge de sécurité par rapport à la saturation de l'ADC, il est suggéré de dimensionner l'amplification de façon à centrer le signal sur 1.65 V et avec une amplitude de 1.5 V. Pour cela nous avons utilisé deux étages amplificateurs, respectivement d'un gain de 68 et 22. Nous avons donc utilisé le montage représenté à la figure 3.2.

Calculons d'abord la tension à la sortie du premier étage (V_{out1}), à l'aide du principe de superposition. On considère d'abord la seule source de tension continue, la capacité est alors un circuit ouvert (3.1) :

$$V_+ = V_- = V_{out1} = 1.65\,\text{V} \tag{3.1}$$

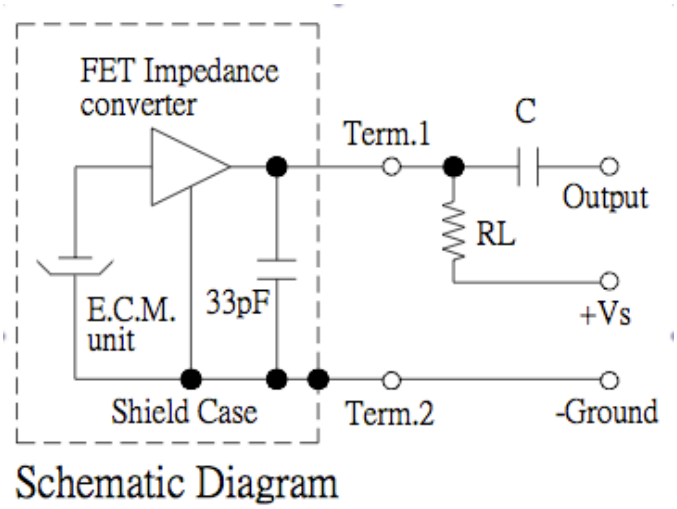

FIGURE 3.1 – Circuit d'utilisation du microphone

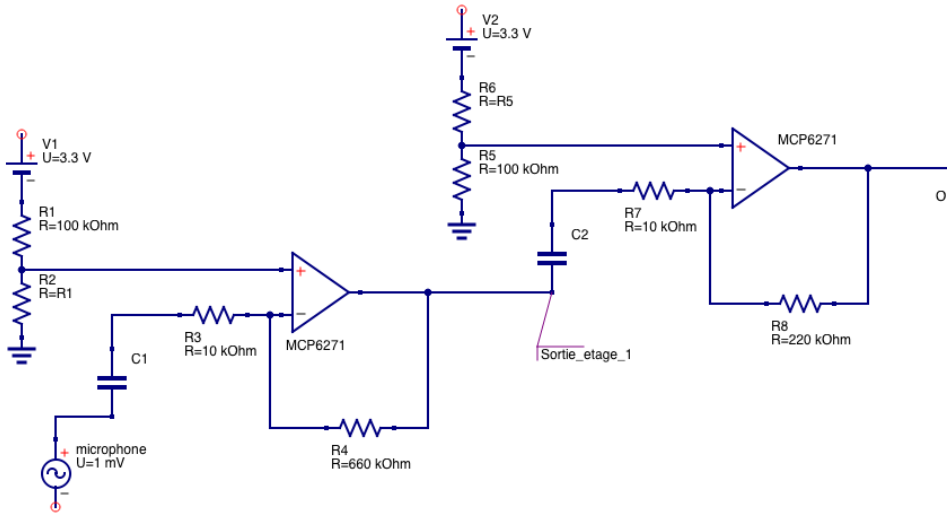

FIGURE 3.2 – Étage amplificateur

Considérons ensuite uniquement la source de tension alternative, le circuit est alors un inverseur classique (3.2) :

$$V_{out1} = \frac{R_2}{R_1} V_{micro} = 68 V_{in} \tag{3.2}$$

On retrouve la tension en sortie du premier étage en sommant ces deux termes pour finalement obtenir (3.3) :

$$V_{out1} = 1.65\,\mathrm{V} + 68 V_{micro} \tag{3.3}$$

La capacité C_2 placée entre nos deux étages d'amplification sert à supprimer la composante continue de la tension, la recentrant autour de 0 V. On évite ainsi de faire saturer notre deuxième ampli-op. Pour le deuxième étage, les calculs sont exactement les mêmes que pour le premier, exception faite du gain du montage inverseur qui vaut 22. On obtient alors notre signal amplifié (3.4) :

$$V_{outfinal} = 1.65\,\mathrm{V} + 1496 V_{micro} \tag{3.4}$$

Pour vérifier le bon fonctionnement de l'étage amplificateur, nous avons effectué des essais sur celui-ci à l'aide du générateur de fonction et du micro. A ce niveau-ci, observer le signal amplifié du micro à l'oscilloscope n'est pas entièrement satisfaisant puisqu'on observe somme toute beaucoup de bruit. Il est cependant possible de vérifier quelques points clés : pas de saturation (plage de sortie du micro conforme à ce qui était attendu), contenu fréquentiel principalement à basse fréquence (FFT ou simplement vérifier le temps caractéristique à l'oscilloscope), et enfin quand même une certaine réaction au son. Les résultats obtenus à l'oscilloscope sont finalement entièrement satisfaisants.

3.2.3 Filtre de garde

Étant donné que nous allons numériser notre signal, il est nécessaire d'utiliser un filtre de garde pour éviter tout repliement spectral. Les spécifications suivantes sont imposées :

- Filtre Butterworth d'ordre 2 ;
- Atténuation maximale des fréquences utiles de 0.99 ;
- Atténuation minimale des fréquences filtrées de 0.05. ;

L'amplitude de la réponse en fréquence d'un filtre de Butterworth est donnée par l'équation 3.5

$$|H(j\omega)| = \sqrt{\frac{1}{1 + (\frac{\omega}{\omega_c})^{2n}}} \tag{3.5}$$

Nous connaissons notre fréquence utile maximale, elle est de 1100 Hz. Si nous injectons cette valeur de ω dans l'équation, nous pouvons remplacer le terme de gauche par 0.99 et n par 2. On peut alors calculer l'inconnue restante, f_c, qui vaut 2972.97 Hz.

Maintenant que nous connaissons la fréquence de coupure de notre filtre, nous pouvons calculer la fréquence à partir de laquelle il fournira une atténuation d'au moins 20. On remplace dans l'équation 3.5 le module de la réponse en fréquence par 0.05, n par 2 et f_c par 2972.97 Hz. On obtient comme valeur de $f_{-26\,\mathrm{dB}}$ 13.287 kHz.

Avec toutes ces informations, nous avons utilisé l'outil «Filter Wizard» d'Analog Devices [1] qui nous a permis de designer notre filtre. La figure 3.3

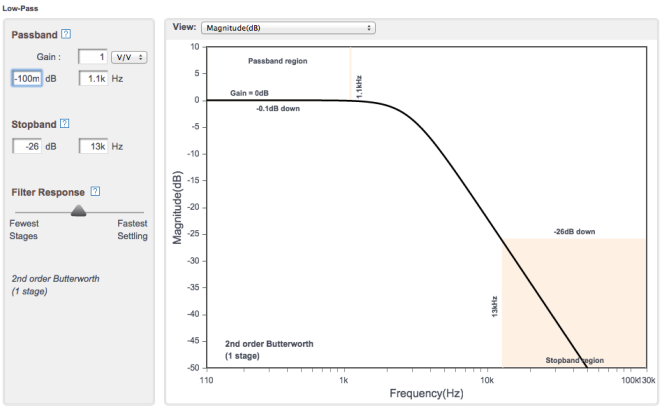

FIGURE 3.3 – L'interface du "Filter Wizard"

montre les spécifications de notre filtre analogique ainsi que l'amplitude de sa réponse en fréquence en fonction de la fréquence. Pour l'implémentation du filtre, Analog suggère une topologie Sallen-Key que nous avons utilisée. La figure 3.4 montre le montage de notre filtre passe-bas. Pour tester notre implémentation, nous lui avons mis en entrée des sinus à différentes fréquences [2] : 900 Hz, 1100 Hz, 13 kHz (première fréquence atténuée avec un facteur 0.05) et à la fréquence de coupure (2972.97 Hz). Les résultats sont exposés aux figures 3.5, 3.6, 3.8 et 3.7. L'entrée est représentée en rouge et la sortie en bleu.

3.2.4 Convertisseur analogique-numérique

Afin que notre microcontrôleur chargé de la communication puisse interpréter les consignes qui lui sont envoyées, il a bien sûr fallu convertir le signal reçu en signal numérique. Pour cela nous avons configuré l'ADC du microcontrôleur communication : c'est la fonction `initADC()` qui contient le code correspondant. Ci-dessous nous listons quelques éléments importants de cette configuration :

1. http://www.analog.com/designtools/en/filterwizard
2. Nous avons d'abord essayé de faire une FFT de la réponse impulsionnelle à l'aide de générateur de PWM et de l'oscilloscope, mais cela n'a pas été concluant car la résolution fréquentielle de l'oscilloscope est trop mauvaise et aussi probablement parce que le pulse le plus étroit que peut fournir le générateur est encore trop large.

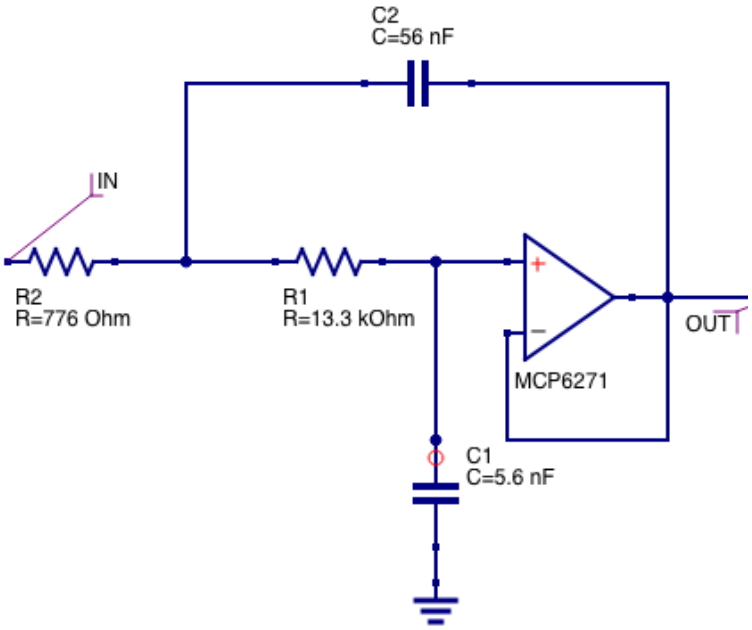

FIGURE 3.4 – Filtre de garde Sallen-Key

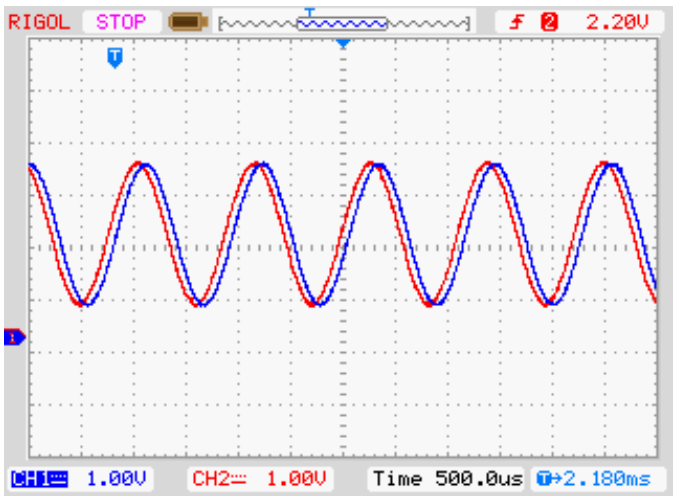

FIGURE 3.5 – Sortie (bleu) du filtre de garde pour une entrée à 900 Hz (rouge)

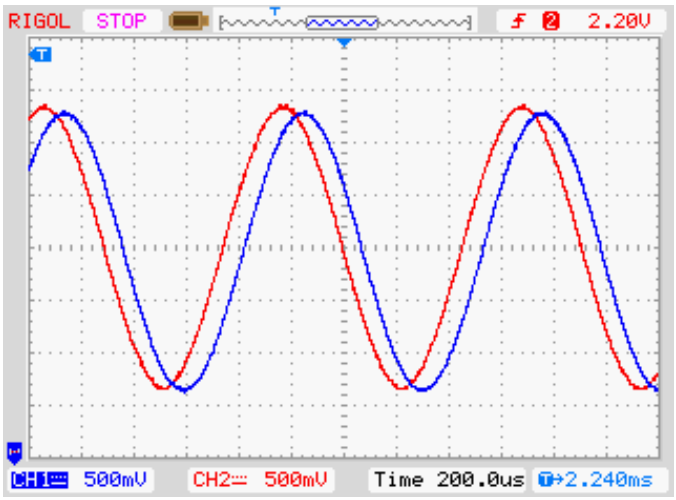

FIGURE 3.6 – Sortie (bleu) du filtre de garde pour une entrée à 1100 Hz (rouge)

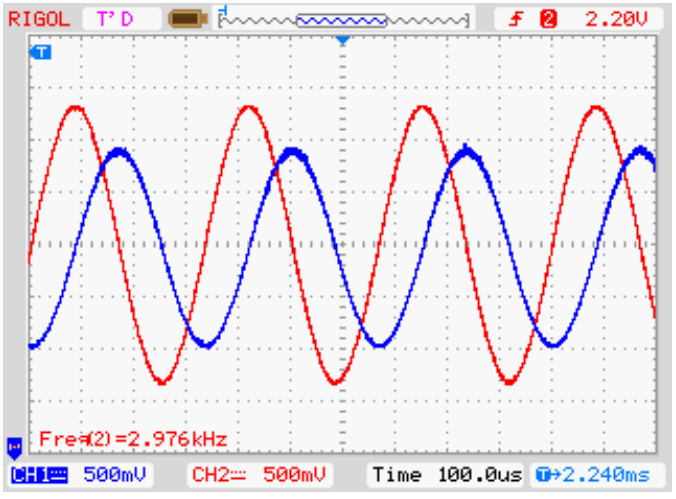

FIGURE 3.7 – Sortie (bleu) du filtre de garde pour une entrée à la fréquence de coupure (rouge)

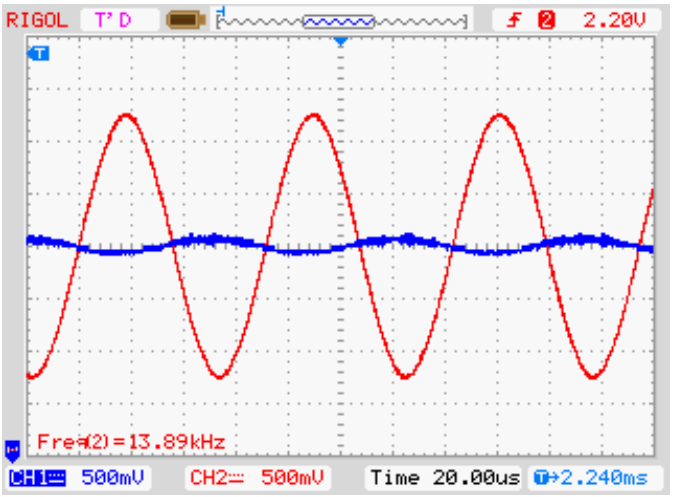

FIGURE 3.8 – Sortie (bleu) du filtre de garde pour une entrée à 13 kHz (rouge)

— L'ADC fonctionne en mode 12 bits, le maximum, afin d'obtenir la meilleure précision possible ;
— La tension de référence pour la conversion est la même que la tension d'alimentation, puisqu'on ne dispose pas de source plus précise ;
— L'ADC commence automatiquement à échantillonner lorsqu'il termine une conversion ;
— L'ADC déclenche une conversion à chaque période du timer 3, dont la période est 50 µs ($f_s = 20\,\text{kHz}$) ;
— Chaque conversion entraîne une interruption ;

Ce dernier point est particulièrement important, car c'est dans la routine de l'ADC que se déroule en fait tout le traitement numérique du signal audio. La trame sortant de l'ADC y est filtrée par deux filtres numériques, et les sorties passent ensuite chacune dans un détecteur de crête. Ceci fait office de démodulation ; la trame est ensuite reconstituée par la fonction `fskDetector()`, qui est déjà fournie. Tout ceci est couvert dans la section suivante, après que la chaîne d'acquisition ait été validée.

3.2.5 Validation de la chaîne d'acquisition

Pour valider la chaîne d'acquisition dans son ensemble, un signal audio est généré à une fréquence connue, dans des conditions réelles[1], et les 100 derniers échantillons sont gardés dans un array qui est ensuite exporté grâce au mode debug et ensuite représenté. Ces tests ont été effectués pour 900, 1100, 3000 (f_c) et 18900 (première fréquence repliée dans le signal utile) Hz

1. Source audio éloignée de 20 – 30 cm, test dans le laboratoire qui est raisonnablement bruité.

et ont tous été concluants (signal suffisamment préservé ou atténué).

3.3 Traitement numérique du signal

Une fois notre signal numérisé, il faut encore en extraire la commande, s'il y en a bien une. Pour cela nous devons détecter la présence de nos deux fréquences de modulation : 900 Hz et 1100 Hz. En pratique, on filtre le signal de façon à ce que toutes les autres fréquences soient atténuées et on vient ensuite vérifier si l'amplitude du signal dépasse encore un certain seuil. Ces deux tâches incombent respectivement aux fonctions `filterNewSample(unsigned int sample, int returnArray[2])` et `char peakDetect(int input[2])`.

3.3.1 Filtres numériques passe-bande

Les filtres sont des filtres récursifs du second ordre. Ils sont décrits par l'équation récursive 3.6 que l'on retrouve dans la fonction `long recurrence(long a1, long a2, long gain, int arrayX[3], long arrayY[3])` ou par le schéma bloc de la figure 3.9.

$$\begin{aligned} Y(z) &= g\frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}} X(z) \\ y[n] &= \frac{1}{a_0}\Big(g(b_0 x[n] + b_1 x[n-1] + b_2 x[n-2]) \\ &\qquad - a_1 y[n-1] - a_2 y[n-2]\Big) \end{aligned} \tag{3.6}$$

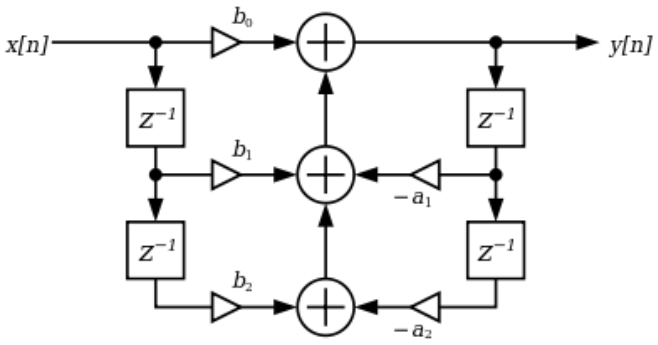

FIGURE 3.9 – Schéma bloc d'un filtre numérique du second ordre

Nous avons utilisé l'outil `fdatool` de Matlab pour trouver les coefficients de nos deux filtres. En donnant comme paramètres au programme les spécifications qui nous étaient imposées, les filtres obtenus étaient composés de quatre sections d'ordre 2. Nous avons commencé par tester ces filtres de manière indépendante du reste du montage et ils fonctionnaient alors

très bien. Cependant quand on les plaçait derrière notre ADC, les sorties des filtres n'avaient plus aucun sens. Après un moment, nous nous sommes aperçu que nos filtres étaient tout simplement trop lents : le temps de calcul nécessaire ralentissait le fonctionnement de l'ADC (nous rappelons que le filtrage a lieu dans l'interruption de l'ADC). Pour nous en rendre compte, nous avons flippé un bit sur une des pattes du microcontrôleur dans cette même interruption de l'ADC et nous avons mesuré la fréquence du signal ainsi obtenu à l'oscilloscope. Si l'ADC arrive à fonctionner à la fréquence d'échantillonnage f_s prévue, la fréquence mesurée doit être égale à $\frac{f_s}{2}$ (le bit est flippé à chaque interruption de l'ADC, une période représente donc deux périodes de l'ADC). Nous avons donc tenté de rendre nos filtres plus rapides en modifiant le code, notamment en remplaçant tous les nombres de type `float` par des `int`, mais aussi en utilisant la connaissance de certains coefficients toujours nuls ou unitaires. Les courbes de Bode du filtre à une seule section centré autour de 900 Hz que nous utilisons pour finir sont données à la figure 3.10.

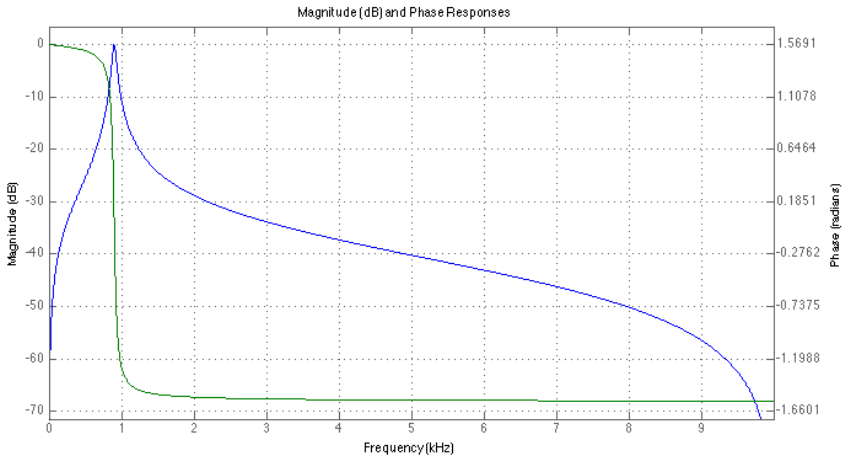

FIGURE 3.10 – Courbes de Bode du filtre numérique centré autour de 900 Hz

3.3.2 Détecteurs de crête

A la sortie des filtres numériques, le signal passe dans deux détecteurs de crête pour détecter la présence de 900 Hz et/ou 1100 Hz. Pour cela, le microcontrôleur garde en mémoire un bloc du signal correspondant à une période de 1/900 s ou 1/1100 s et trouve le maximum sur ce bloc. Si ce maximum dépasse un certain seuil, la fréquence correspondante est considérée présente. Ces informations sont alors transmises à la fonction `fskDetector(int detLow, int detHigh)` qui va mettre en forme la trame détectée.

3.4 Validation du bloc «communication audio»

Chronologiquement, ce bloc a été achevé en dernier, son fonctionnement a donc été prouvé par le fonctionnement du robot dans son ensemble et sa réponse aux ordres reçus. On observe cependant que la portée de la communication est extrêmement limitée (30 – 40 cm maximum dans la direction la plus favorable), et dépend fortement de la direction [1].

1. Cela est aussi dû à la forte directionnalité des enceintes de portable utilisées

Chapitre 4

Transmission des ordres entre les deux microcontrôleurs

Ce chapitre couvre la transmission des trames démodulées par le premier microcontrôleur vers le deuxième microcontrôleur. Dans la section suivante, le format des entrées et sorties de ce bloc vont être détaillées. [1]

4.1 Première analyse

Puisqu'on désire limiter le nombre d'opérations effectuées par le microcontrôleur effectuant le traitement du signal audio, les trames de 10 bits renvoyées par la fonction `fskDetector` sont envoyées telles quelles au deuxième microcontrôleur. L'entrée du bloc transmission est donc une trame de 10 bits. L'uart, implémenté en hardware des deux côtés est utilisé pour effectuer la transmission. Celui-ci utilise des trames de 8 ou 9 bits, et il faudra donc 2 trames d'UART pour transmettre une trame de FSK. L'émetteur et le récepteur seront donc logiquement des machines à états séquentielles. Plutôt que de simplement reconstituer la trame de 10 bits originale, on choisit que le récepteur renvoie directement d'une part les 2 bits de commande et d'autre part les 8 bits du paramètre contenus dans une transmission.

Les parties récepteur et émetteur vont être abordées en parallèle dans la suite, puisqu'elles sont fortement liées. Tout d'abord, la configuration des modules UART est examinée [2], ensuite, l'émetteur et le récepteur vont être construits.

1. Ce chapitre couvre les fichiers `uart.*` des projets `communication.X` et `propulsion.X`
2. La plupart des paramètres doivent forcément être les mêmes des deux côtés de la transmission pour que celle-ci soit possible.

4.2 Configuration des modules UART

4.2.1 Paramètres communs

Pour que la communication soit possible, les deux UART doivent être en accord sur quatre paramètres : le Baud Rate, la polarité des ports `TX` et `RX`, le format de trame, et enfin le protocole d'envoi de trame. Ces paramètres sont fixés dans la fonction `initUart()`, qui est donc en grande partie commune aux deux microcontrôleurs.

Le Baud Rate est fixé à 9600. Cette valeur rend la transmission d'une instruction (deux trames) pratiquement instantanée par rapport aux autres constantes de temps en présence ($f_{regul} = 100\,\text{Hz}$, $f_{symbol} = 10\,\text{Hz}$), tout en étant suffisamment basse pour permettre d'utiliser l'horloge de l'UART en 16X speed mode, ce qui diminue la probabilité d'erreur car chaque bit est alors échantillonné trois fois au lieu d'une.

Les trames sont choisies contenant 8 bits utiles avec un bit de parité et terminées par 1 stop bit. Passer à 9 bits utiles et sans bit de parité ne présente pas d'intérêt puisqu'il faut toujours envoyer deux trames pour transmettre une instruction complète (10 bits). La détection rudimentaire d'erreur fournie par le bit de parité est donc légèrement préférable. La polarité n'a elle aucune importance, du moment qu'elle est identique de part et d'autre d'une ligne TX – RX.

Vu les très faibles contraintes sur l'UART, le risque d'erreur et la probabilité que les FIFO de réception et transmission se remplissent sont pratiquement nuls, on peut donc se contenter du protocole le plus simple avec seulement deux fils RX et TX et pas de flow control physique. Au niveau des pattes, RX est d'une part simplement lié à la patte reprogrammable choisie pour RX dans le registre `RPINR18bits.U1RXR` et celle-ci est mise en input, et d'autre part, la patte reprogrammable choisie pour TX est liée à TX dans le registre `RPORXbits.RPXR`. Enfin, les branchements croisés RX-TX sont effectués. Les paramètres de l'UART et les branchements sont validés dans la section suivante.

4.2.2 Validation de la partie «physique» de l'UART

A partir de cette configuration essentielle, le fonctionnement de l'UART entre les deux microcontrôleurs est vérifié en envoyant une suite de caractères avec un microcontrôleur et en vérifiant que ceux-ci sont bien reçus par l'autre microcontrôleur, et ce à une fréquence suffisamment lente pour pouvoir utiliser le debugger de MPLab. L'émetteur de test utilise l'interruption d'un timer 32 bit [1] pour écrire un caractère dans le registre `U1TXREG` et incrémenter ce caractère. On place un breakpoint dans l'interruption `U1RXREG`

1. Un timer 32 bits est utilisé afin d'envoyer des caractères suffisamment lentement pour qu'une fois le breakpoint déclenché, nous ayons le temps de vérifier la valeur du caractère et de relancer le code côté récepteur avant qu'une nouvelle trame ne soit envoyée.

pour vérifier que la trame est bien reçue et qu'il n'y a pas eu d'erreur de parité ou de formatage.

La communication a été testée dans les deux sens et la partie «physique» de l'UART a ainsi été validée. Il reste maintenant à implémenter le software autour de ce bloc pour transmettre des trames FSK de 10 bits du microcontrôleur communication vers le microcontrôleur propulsion. Tout d'abord, les modes d'interruption pour la réception et l'émission sont adaptés pour chaque microcontrôleur en fonction du rôle que celui-ci joue dans la communication.

4.2.3 Choix des modes d'interruption

La communication est clairement asymétrique : le microcontrôleur communication transmet les ordres au microcontrôleur propulsion, qui ne répond presque jamais, comme il est expliqué dans la suite. Il est donc logique que les modes d'interruptions soient légèrement différents pour les deux microcontrôleurs.

Réception

Le mode d'interruption pour la réception est tout de même identique pour les deux microcontrôleurs. Une interruption est déclenchée dès qu'une nouvelle trame peut être lue. La routine de réception d'une trame d'UART vérifie simplement qu'il n'y a pas d'erreur de formatage ou de parité puis appelle la fonction `handleReceived(char received)`, qui déclenche le traitement de la trame reçue si elle est correcte, ou la sort simplement de la FIFO si elle est incorrecte. Bien évidemment, la fonction `handleReceived(char received)` contient toute la logique et varie entre les deux microcontrôleurs.

Émission

Les deux microcontrôleurs envoient finalement assez peu de trames. Pour cette raison, l'émetteur UART ne déclenche pas par défaut d'interruption liée au statut de sa FIFO. La différence entre les deux microcontrôleurs réside dans le fait que le microcontrôleur propulsion n'envoie jamais de trames successives mais le microcontrôleur communication bien. Pour ce dernier, nous avons choisi d'exploiter une routine d'interruption pour envoyer les trames.

Du côté propulsion, l'interruption d'émission est donc simplement désactivée. La configuration de l'UART du microcontrôleur communication inhibe au départ l'interruption d'émission, mais comme elle sera activée durant le fonctionnement, il faut tout de même configurer le mode d'interruption. On choisit de déclencher une interruption dès qu'une trame peut-être écrite dans la FIFO, ce qui correspond au mode `0b00`. La routine d'émission contient en

réalité la machine d'état de l'émetteur, et est donc présentée dans la section suivante.

4.3 Programmation des émetteurs et récepteurs

Dans cette section, le format de deux trames d'UART pour représenter une trame de FSK va être présenté, et les récepteurs et émetteurs des deux microcontrôleurs vont être construits à partir de ce format.

4.3.1 Division d'une trame de FSK

Pour rappel le format des trames à la sortie du démodulateur FSK est donné à la figure 4.1. C'est cette trame qui doit être divisée en deux trames

Données	
Ordre	Paramètre
2 bits	8 bits

FIGURE 4.1 – Format d'une trame à la sortie de `fskDetector`.

de 8 bits. De plus, le récepteur doit pouvoir faire la différence entre les trames «première partie» et les trames «deuxième partie», afin de pouvoir détecter certaines erreurs (partie manquante ou répétition) et pouvoir correctement reconstituer l'instruction reçue. Dans la suite, les trames «première partie» seront notées T1 et les trames «deuxième partie» T2.

L'ordre ne peut prendre que les valeurs `0b00`, `0b10` ou `0b01`. On choisit donc que la quatrième possibilité, `0b11` constitue les deux premiers bits de T1 afin, de distinguer celle-ci de T2, dont les deux premiers bits sont l'ordre [1]. Les 8 bits de paramètre restants sont répartis entre T1 et T2. Il reste donc deux bits inutilisés dans chaque trame, qu'on met simplement à 0. Tout ceci est résumé dans la figure 4.2.

A partir de ce format de trame, les émetteurs et récepteurs vont maintenant être construits en suivant l'émission d'une commande et en examinant tous les scénarios possibles.

4.3.2 Émetteur communication

L'entrée du bloc UART dans son entier est la fonction `sendCommand(int newCommand)`. Comme des erreurs sont envisageables, le récepteur peut demander que la commande soit répétée. Pour cette raison, `newCommand` est stockée de manière persistante avant d'être envoyée. Comme annoncé plus haut,

1. Voir la section 4.3.3 sur le récepteur propulsion pour voir pourquoi l'ordre est envoyé dans la deuxième trame.

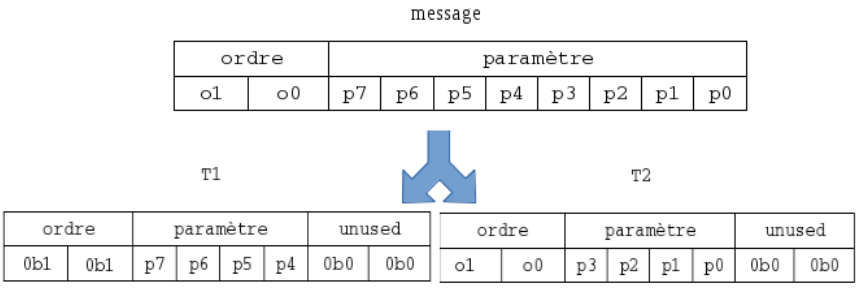

FIGURE 4.2 – Division du message en deux trames UART.

les interruptions d'émissions sont utilisées pour envoyer les deux trames, et l'émetteur est une machine d'état séquentielle.

Lorsqu'une commande doit être envoyée, l'interruption d'émission est activée, et l'état de l'émetteur mis à zéro, ce qui correspond à aucune trame déjà envoyée. Ensuite, dans la routine d'émission, T1 ou T2 est construite selon l'état, puis envoyée, et l'état est incrémenté. Après l'envoi de T2, l'interruption est désactivée jusqu'à ce qu'une nouvelle commande doive être envoyée, et ainsi de suite. Ceci est exécuté par la routine d'émission.

Le récepteur propulsion est calqué sur cet émetteur et est présenté dans la section suivante.

4.3.3 Récepteur propulsion

Le récepteur propulsion est donc lui aussi une machine d'état séquentielle à deux états.

A l'état 0, une T1 est attendue. Si c'est bien une T1 qui est reçue, alors l'état passe à 1 et `p<7-4>` contenus dans la trame sont stockés. Sinon, il y a eu une erreur, l'état reste à 0 et l'UART propulsion demande que le message soit répété.

A l'état 1, une T2 est attendue. Si une T2 est reçue, alors le message a été entièrement reçu, l'état retourne à zéro et le récepteur retourne séparément `o<1-0>` et `p<7-0>` à partir de T2 [1] et de `p<7-4>` en mémoire. Sinon, il y a eu une erreur, l'état retourne aussi à zéro et l'UART propulsion demande que le message entier soit répété. Ceci est exécuté par la fonction `handleReceived()` qui appelle `handleParam1()` ou `handleParam2()` selon le type de trame reçue.

D'autres possibilités d'erreurs ont été vues plus haut : mauvaise parité (détecté par l'UART lui-même) ou mauvais formatage (détecté par l'UART ou le récepteur si LSB et LSB+1 d'une trame sont non nuls). Le mécanisme est systématiquement le même : l'état du récepteur retourne à zéro et la répétition du message est demandée par la fonction `askRepeat()`.

1. On voit donc ici pourquoi il est plus intéressant d'envoyer l'ordre dans T2 : cela permet de ne pas devoir le stocker à l'état 0.

La demande de répétition du message est la seule chose que l'UART propulsion émet. L'émetteur propulsion est donc très simple. Il est présenté dans la section suivante.

4.3.4 Émetteur propulsion

L'émetteur propulsion n'envoie que des messages d'une trame, et ce relativement rarement. On peut donc se contenter de vérifier (pour la forme) que l'on peut écrire dans la FIFO et d'y écrire la trame `0x01` qui demande le renvoi de la commande. Il s'agit donc aussi de la seule trame que le récepteur communication doit gérer. Ce récepteur est donc lui aussi très simple. Il est présenté dans la section suivante.

4.3.5 Récepteur communication

Si l'UART communication reçoit une trame `0x01`, il faut donc renvoyer la commande actuelle. Pour cela `sendCommand(int newCommand)` est légèrement modifiée. Si `newCommand` est `0xFFFF` (pas un message valide), alors la commande gardée en mémoire n'est pas écrasée, et l'envoi déclenché ensuite renvoie donc la commande précédente. Ceci est fait par un simple test au début de la fonction `sendCommand`

Le cas où la trame reçue côté communication n'est pas valide n'est pas géré : la transmission de la commande, s'il y avait bien une commande à transmettre au départ, est abandonnée. Ce cas ne s'est jamais présenté. Tous les scénarios possibles ont donc été examinés et le code présenté implémente donc une communication UART complète et parfaitement fonctionnelle.

Le traitement de l'ordre et du paramètre doit maintenant être abordé. Ce traitement est appelé dans la fonction `handleParam2()` du code source du récepteur propulsion par l'instruction `interpretCommand(command, param);`, et va être présenté dans le chapitre suivant. La transmission de la commande entre les deux microcontrôleurs et l'interprétation de celle-ci seront validées en même temps à la fin du chapitre suivant.

Chapitre 5

Interprétation des ordres

Ce chapitre décrit le bloc qui fait le lien entre les instructions reçues par l'UART propulsion et la régulation des moteurs. Dans le chapitre 2, nous avons vu que les consignes sont générées sur base de trois variables accessibles par les autre blocs : l'accélération, la distance ou l'angle total à parcourir, et la distance ou l'angle à parcourir avant décélération. La régulation doit aussi savoir si le robot est en train de tourner ou d'avancer tout droit pour simplifier le test d'arrivée. Dans la section suivante, ces variables d'état sont rapidement détaillées, et nous verrons ensuite comme celles-ci sont modifiées par les instructions reçues par l'UART. [1]

5.1 Variables d'état externes de la régulation du robot

Les variables sont dédoublées pour les mouvementsa rectilignes et les rotations. Dans la suite, toutes les variables sont citées, mais le rôle ou l'évolution au cours du temps n'est décrit que pour les variables «rectilignes» afin de ne pas alourdir cette section.

L'accélération et l'accélération angulaire sont fixées par `float acceleration` et `float angularAcceleration`. La régulation est déjà responsable, lors de l'exécution d'une commande, de d'abord mettre l'accélération à zéro lorsque la vitesse nominale est atteinte, puis de fixer une décélération lorsque la distance avant décélération est atteinte, et enfin de remettre une dernière fois l'accélération à zéro lorsque la distance totale est atteinte. Le rôle du bloc traité ici se borne donc à fixer une valeur non-nulle lors de la réception d'une commande pour démarrer.

La distance ou l'angle total à parcourir sont fixés par `char goalDistance` et `float goalTheta`. Seul le bloc couvert dans ce chapitre-ci modifie cette valeur. La distance ou l'angle à parcourir avant décélération sont

1. Ce chapitre couvre les fichiers `decision.*` du projet `propulsion.X`

fixés par `char decelerationDistance` et `float decelerationTheta`. Seul ce bloc modifie cette valeur, qui est soit une valeur par défaut si la distance totale à parcourir est suffisante, soit un tiers de la distance totale si $\frac{\texttt{goalDistance}}{3} < \texttt{DFLT\_DECELERATION\_DST}$.

Enfin, la commande en cours d'exécution, est définie par `char goingStraight`, et `char rotating`. Ces variables sont simplement utilisées pour faciliter les tests d'arrivée, et sont uniquement modifiées par ce bloc-ci.

5.2 Interprétation des ordres reçus

L'ordre et le paramètres sont interprétés par la fonction `interpretCommand(unsigned char order, unsigned char param)`. En fonction de la valeur de `order`, `param` est soit interprété comme signé, soit comme non-signé et le signe est ensuite affecté selon que l'ordre soit `0b10` ou `0b01`. Ensuite les valeurs des autres variables listées plus haut sont déterminées. Ce bloc est aussi responsable de fixer la valeur des k_p de la régulation de distance et de rotation. Ces valeurs dépendent simplement de l'ordre en cours d'exécution [1] et ont été déterminées expérimentalement.

`decision.h` fournit aussi `void resetStateVariables()` qui permet de remettre le robot dans un état inactif, et le wrapper `void stop()` qui appelle les fonctions de reset des différents blocs du robot afin d'obtenir un arrêt immédiat et de garantir que l'état du robot soit connu et acceptable avant d'exécuter une nouvelle commande. Ceci permet également d'obtenir un arrêt instantané du robot en envoyant une nouvelle consigne «avance de 0».

5.3 Validation du bloc «transmission - interprétation»

Le fonctionnement complet du robot à partir de la démodulation FSK non incluse est vérifié en envoyant des commandes connues depuis le microcontrôleur communication et en vérifiant que celles-ci sont bien accomplies par le robot. Pour faire apparaître les erreurs et pour mesurer leur fréquence, nous utilisons, comme à la section 4.2.2, un timer 32 bits pour répéter un grand nombre de fois différentes commandes, tout en laissant au robot le temps d'exécuter celles-ci.

Nous n'avons jamais observé d'erreur dans la transmission d'une trame complète : le robot effectue toujours la bonne commande. Comme vu au chapitre 2 sur la régulation, la précision de cette dernière n'est cependant pas meilleure que 5 – 7 cm, et une commande trop petite peut mener le

1. Selon que le robot est en train d'aller tout droit ou d'effectuer une rotation un des deux k_p sert essentiellement au suivi de consigne tandis que l'autre sert à la réjection de perturbation.

robot à ne pas bouger. Le debugger confirme alors bien que l'ordre a été reçu mais qu'il n'est pas suffisant pour que l'erreur maximale suffise à ébranler le robot.

Chapitre 6

Conclusion

Dans l'ensemble ce projet nous a permis d'apprendre beaucoup de choses. Nous avons finalement pu appliquer beaucoup de théorie étudiée dans différentes matières et les appliquer à un projet concret.

Nous avons réussi à respecter le cahier des charges : notre robot est capable de recevoir, traiter, convertir, transmettre et effectuer les différents ordres qu'il reçoit. Les seules difficultés que nous rencontrons encore sont pour les petites distances à parcourir ainsi que les petits angles. Les moteurs n'arrivent pas à être assez précis, leur qualité ne semble pas suffisante pour cela. En cause, la présence de zones mortes assez larges et variables.

Pour le reste, le robot fonctionne bien, il faut que le volume soit élevé et l'enceinte proche du micro pour qu'il soit correctement perçu, mais le signal est bien interprété par notre chaîne d'acquisition. l'ordre est rapidement transmis à notre chaîne de régulation qui met en marche le robot. Les consignes données sont respectées à un degré tout à fait respectable.

Pour conclure, nous dirons que, même si ça n'a pas toujours été évident, nous sommes content d'avoir finalement réussi à terminer notre projet et de tout ce que nous avons pu apprendre pour y parvenir.

Table des figures