Rapport projet intégré

Antoine Aupée – Joachim Draps – Nathan Dwek

20 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. 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 adoptée, 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-problème bien définis.

1.1 Analyse du cahier des charges

Le robot doit

Chapitre 2

Communication

Pour ce projet, le robot doit être capable de répondre à trois ordres différents : déplacement en ligne droite et rotation dans les deux sens. Le signal audio qui est envoyé doit donc contenir cet ordre ainsi qu'un paramètre : le nombre (signé) de centimètres à parcourir ou l'angle de rotation en degré. 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.

2.1 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 : il permette 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 la convention suivante :

— `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 dimensionnement de la chaîne d'acquisition et la démodulation sont repris 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 :

— Le micro
— L'amplification
— Le filtre de garde
— Le convertisseur analogique-numérique

Le signal doit ensuite est ensuite démodulé à l'aide des éléments suivants :

— Les filtres passe-bande
— Les comparateurs

Dans les deux section 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 et/ou du 1100 Hz

2.2 Dimensionnement et implémentation de la chaîne d'acquisition

2.2.1 Le micro

Notre microphone fournit 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 2.1.

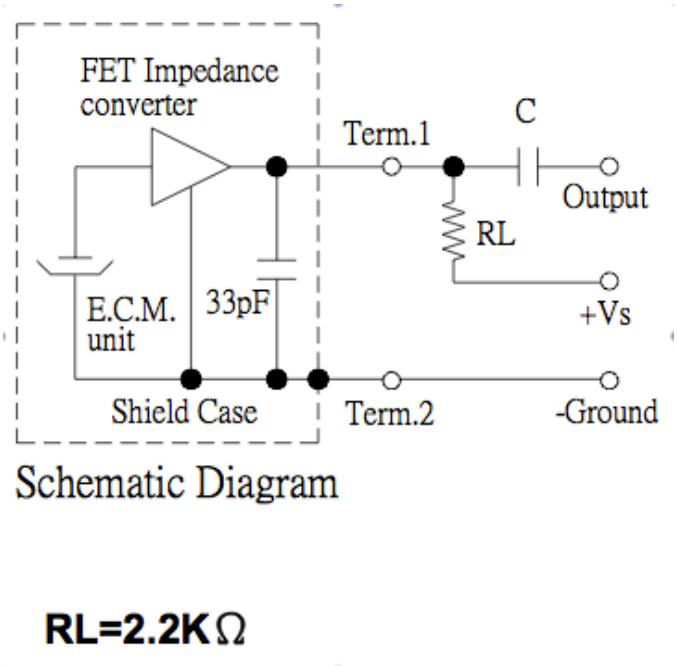

FIGURE 2.1 – Circuit du microphone

Le principe de fonctionnement du micro est que sa membrane extérieure

mobile constitue l'armature d'un condensateur chargé en permanence, la deuxième armature étant fixe. Le mouvement de cette membrane, dû aux variations de pression, va modifier la capacité du condensateur, et donc la tension à ses bornes. Le microphone est dit piézoélectrique. Dans notre cas, le microphone est aussi dit "à électret", ce qui signifie qu'il ne nécessite pas d'alimentation pour maintenir le condensateur chargé : le matériau constituant la membrane présente la propriété de conserver une charge électrostatique. Cependant, une alimentation est nécessaire pour polariser le transistor de l'étage de sortie du microphone, au travers de la résistance R_L.

2.2.2 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 d'occuper cette plage le plus largement 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 2.2.

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 (2.1) :

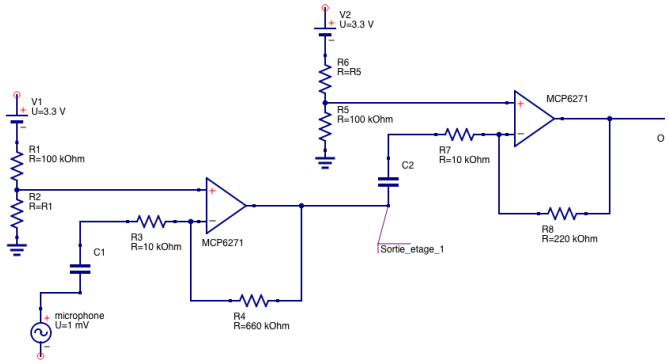

FIGURE 2.2 – Étage amplificateur

$$V_+ = V_- = V_{out1} = 1.65\,\text{V} \tag{2.1}$$

Considérons ensuite uniquement la source de tension continue, le circuit est alors un inverseur classique (2.2) :

$$V_{out1} = \frac{R_2}{R_1} V_{micro} = 68 V_{in} \tag{2.2}$$

On retrouve la tension en sortie du premier étage en sommant ces deux termes pour finalement obtenir (2.3) :

$$V_{out1} = 1.65\,\mathrm{V} + 68V_{micro} \tag{2.3}$$

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 (2.4) :

$$V_{outfinal} = 1.65\,\mathrm{V} + 1496V_{micro} \tag{2.4}$$

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 point 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.

2.2.3 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
— L'atténuation maximale des fréquences utiles doit être de 0.99
— L'atténuation minimale des fréquences filtrées doit être de 0.05.

L'amplitude de la réponse en fréquence d'un filtre de Butterworth est donné par l'équation 2.5

$$|H(j\omega)| = \sqrt{\frac{1}{1 + (\frac{\omega}{\omega_c})^{2n}}} \tag{2.5}$$

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 2.5 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 2.3

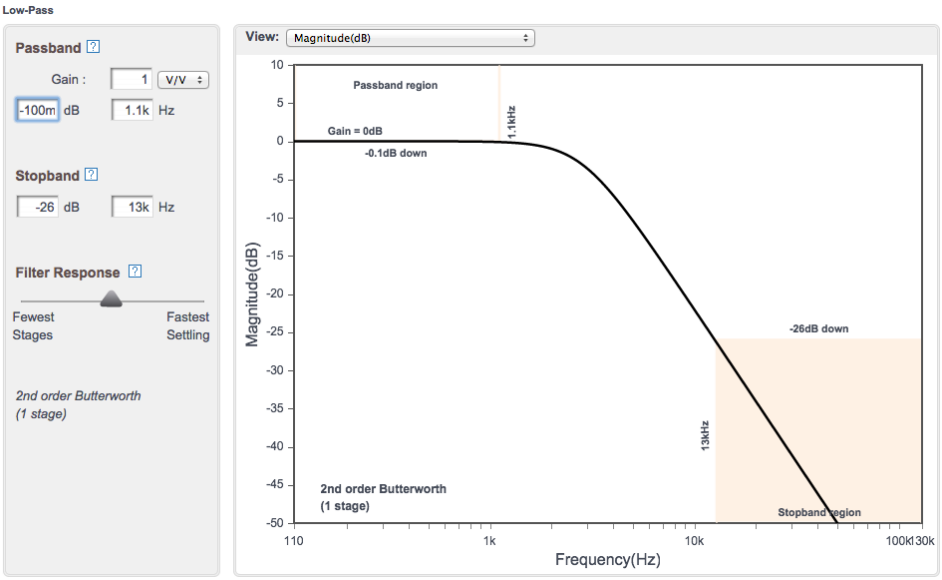

FIGURE 2.3 – L'interface du "Filter Wizard"

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 nous conseillait un étage Sallen-Key, ce que nous avons fait. La figure 2.4 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 2.5, 2.6, 2.7 et 2.8. L'entrée est représentée en rouge et la sortie en bleu.

2.2.4 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.

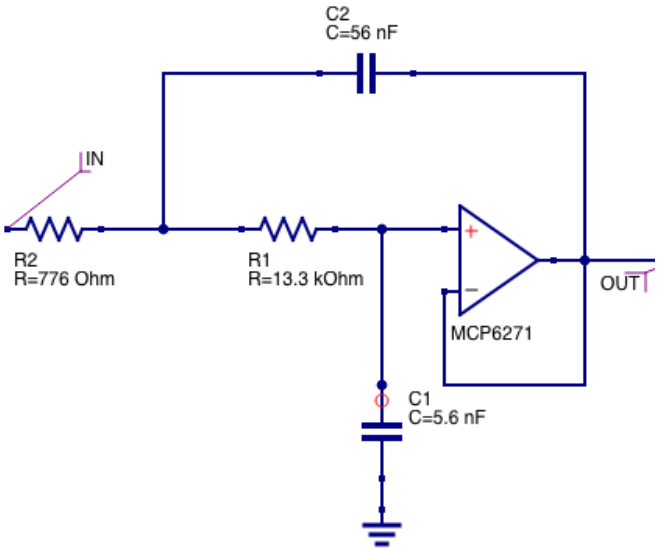

FIGURE 2.4 – Montage de notre filtre analogique

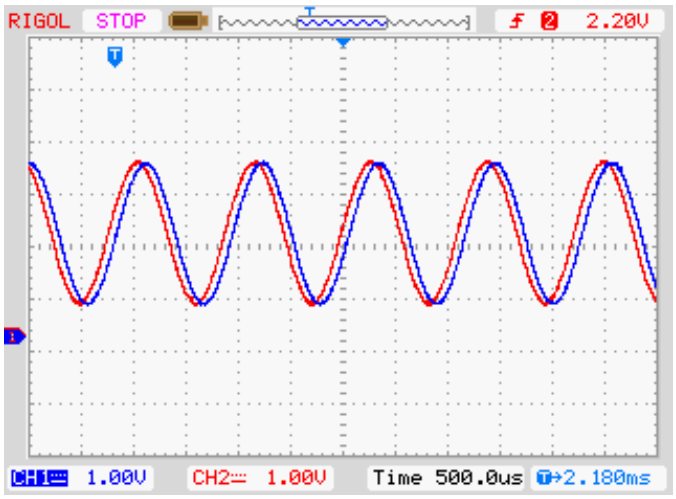

FIGURE 2.5 – 900 Hz

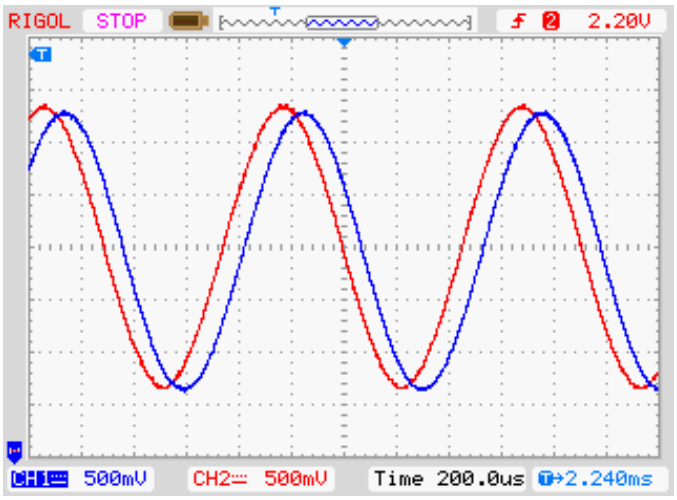

FIGURE 2.6 – 1100 Hz

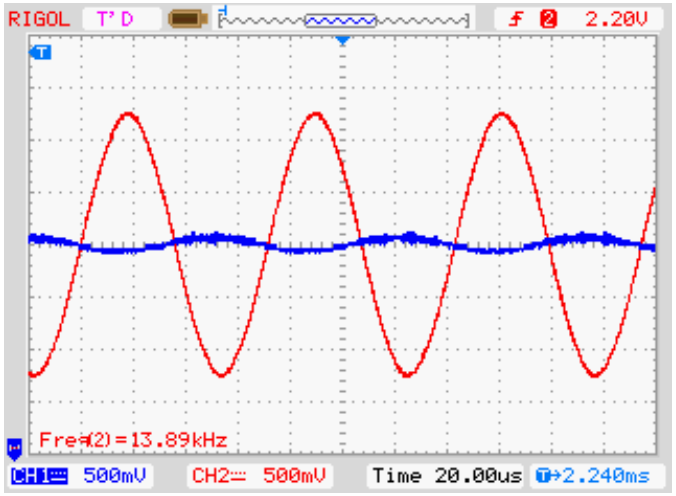

FIGURE 2.7 – 13 kHz

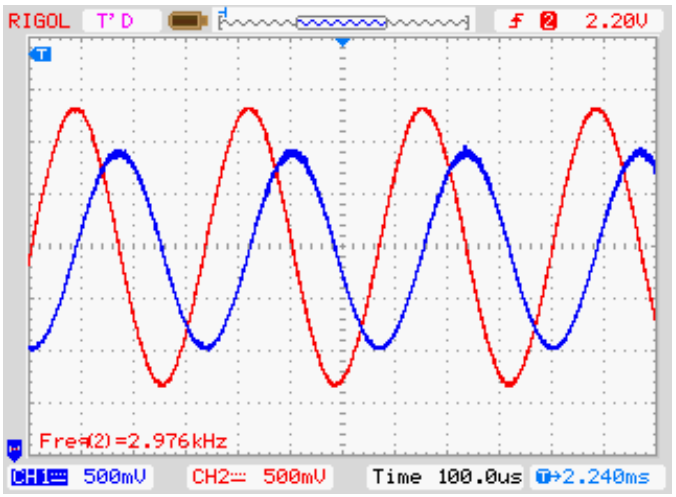

FIGURE 2.8 – Fréquence de coupure

— L'ADC fonctionne en mode 12 bits, ce qui est son maximum
— La tension de référence pour la conversion est la même que la tension d'alimentation
— 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 cette interruption 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.

2.3 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 `peakDetect(int input[2])`.

2.3.1 Filtres numériques passe-bande

Les filtres sont des filtres récursifs du second ordre. Ils sont décrits par l'équation récursive 2.6 que l'on retrouve dans la fonction `recurrence(long a1, long a2, long gain,int arrayX[3], long arrayY[3])` ou par le schéma bloc de la figure 2.9.

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) - a_1y(n-1) - a_2y(n-2) \quad (2.6)$$

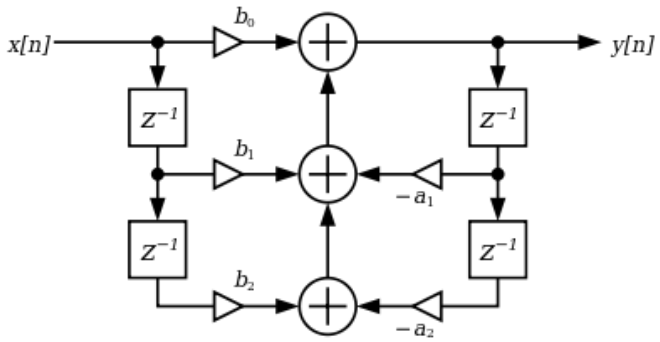

FIGURE 2.9 – Schéma bloc d'un filtre numérique du second ordre

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 2.10.

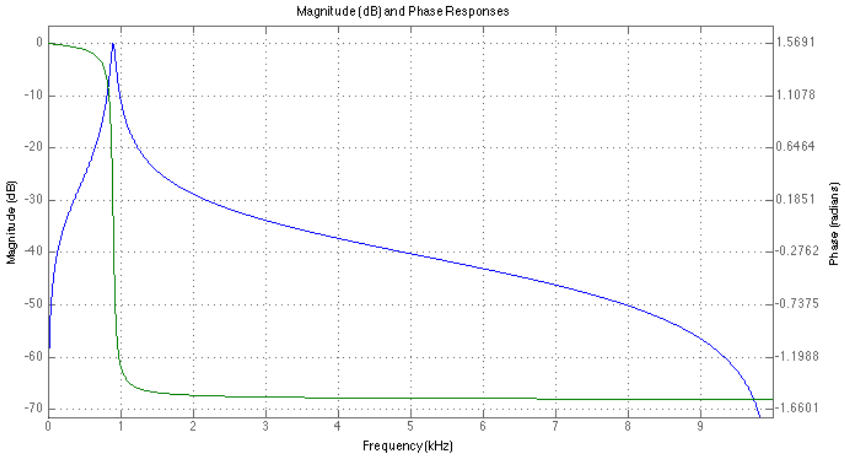

FIGURE 2.10 – Courbes de Bode du filtre numérique centré autour de 900 Hz

2.3.2 Détecteurs de crête

Derrière les filtres numériques, le signal passe dans deux comparateurs chargés de détecter la présence de 900 Hz 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.

Chapitre 3

Régulation

Dans cette partie

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.

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 sera donc logiquement des machines à état 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 d'arguments 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[1], ensuite, l'émetteur et le récepteur vont être construits.

1. 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 bites 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(μC_1)-RX(μC_2).

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 branchement 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 caractère 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`, 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` 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 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.

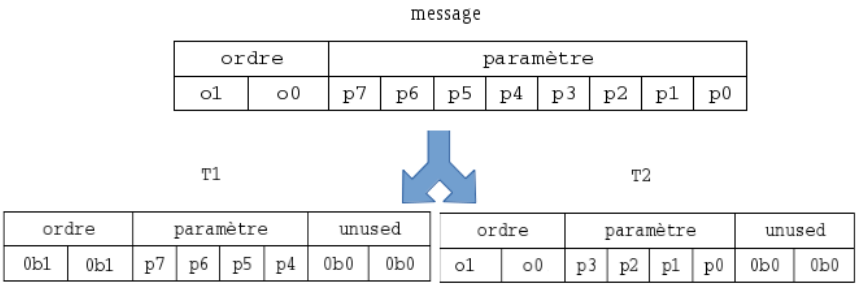

FIGURE 4.2 – Division du message en deux trames UART.

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 de 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 peut 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. Ceci va être validé dans la section suivante.

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 validée 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 3, 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.

5.1 Variables d'état externes de la régulation du robot

Les variables sont dédoublées pour les mouvement 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 fixés par `char decelerationDistance` et `float decelerationTheta`. Seul ce bloc modifie cette valeur, qui est soit une valeur par défaut si la dis-

tance 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 `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, 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 leurs fréquence, nous utilisons, comme à la section 4.2.2, un timer 32 bits pour répéter un grand nombre de fois différentes commandes.

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 3 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 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.

1. Selon que le robot soit 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.

Table des figures