Rapport projet intégré

Antoine Aupée – Joachim Draps – Nathan Dwek

11 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 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, notre robot doit être capable de répondre à trois ordres différents : avance tout droit, tourne à droite, tourne à gauche. Le signal audio que nous lui envoyons doit donc contenir cet ordre ainsi qu'une quantité associée, le nombre de centimètres à parcourir ou le nombre de degré qu'il doit tourner. 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

2.2 Notre chaîne d'acquisition

Chapitre 3

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.

3.1 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.

3.2 Configuration des modules UART

3.2.1 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 para-

```
/*Extrait de initUart*/
void initUart(void){
    //Config Générale
    U1MODEbits.IREN = 0;//IRDA off.
    U1MODEbits.UEN = 0b00;//Seuls les ports U1TX et U1RX sont utilisés.
                          //=>il ne faut pas config l'hardware flow-control
    U1MODEbits.LPBACK = 0;//0 :inter uC. 1 : test uC vers lui même.
    U1MODEbits.ABAUD = 0;//Auto Baud off.
    U1MODEbits.BRGH = 1;//16coups de clock par bit envoyé
    //Plus robuste (3 samples par bit) et de toute façon on a un petit baudrate.
    U1BRG = BRGVAL;//Fixe le baud rate par la longueur du timer lié
    //BRGVAL = f_μC / (4×f_Baud) − 1
    U1MODEbits.PDSEL = 0b01;//8bit data, bit de parité (paire)
    U1MODEbits.STSEL = 0;//1 stop bit.

    //Polarité
    U1STAbits.UTXINV = 1;
    U1MODEbits.URXINV = 1;//tout actif à l'état haut

    //Routage des ports TX et RX vers les pattes utilisées : propre à chaque μC

    //Start uart et ses composants
    U1MODEbits.UARTEN = 1;//Active l'uart 1
    U1STAbits.UTXEN = 1;//UART prend le controle des ports
}
```

Code source 3.1 : Configuration commune aux deux UART.

mètres sont fixés dans le code source 3.1, qui est donc commun 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.

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

A partir de la configuration 3.1, 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 est donné en 3.2, et le récepteur de test en 3.3. Un timer sur 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.

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.

3.2.3 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

```c
unsigned char toSend = 'a';

int main(void){
    init();
    initUart();
    T2CONbits.T32 = 1;
    PR2 = 65000;
    PR3 = 5000;//Période très lente
    IEC0bits.T3IE = 1;
    T2CONbits.TON = 1;
    while(1){
    }
}

void _ISR _T3Interrupt(void){
    IFS0bits.T3IF = 0;
    if (!U1STAbits.UTXBF){
        U1TXREG = toSend;
        toSend++;
        if(toSend > 255){
            toSend = 0;
        }
    }
}
```

Code source 3.2 : Émetteur test UART.

```c
int main(void){
    init();
    initUart();
    U1STAbits.URXISEL = 0b00;//Déclenche une interruption à chaque trame reçue
    IEC0bits.U1RXIE = 1;//Enable UART RX interrupt
    while(1){
    }
}
void _ISR _U1RXInterrupt(void){
    IFS0bits.U1RXIF = 0;
    if ((U1STAbits.PERR || U1STAbits.FERR )== 0 ){
    //Check erreur de parité ou de formatage
        char received = U1RXREG;//Breakpoint ici pour vérifier les caractères reçus
    }else{
        char garbage = U1RXREG;//Breakpoint ici pour vérifier qu'on peu (pas) d'erreur
    }
}
```

Code source 3.3 : Récepteur test UART.

nouvelle trame peut être lue. Ceci est configuré par le code 3.4.

```
/*Extrait de initUart*/
void initUart(void){
    U1STAbits.URXISEL = 0b00;//Déclenche une interruption à chaque trame reçue
    IEC0bits.U1RXIE = 1;//Enable UART RX interrupt
}

void _ISR _U1RXInterrupt(void){
    IFS0bits.U1RXIF = 0;
    if ((U1STAbits.PERR || U1STAbits.FERR )== 0 ){
    //Check erreur de parité ou de formatage
        char received = U1RXREG;
        handleReceived(received);
    }else{
        char garbage = U1RXREG;
    }
}
```

Code source 3.4 : Configuration de l'interruption RX des deux UART.

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 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` 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 est un peu plus longue et donnée dans le code 3.5. La routine d'émission contient en réalité la machine d'état de l'émetteur, et est donc présentée dans la section suivante.

3.3 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

```
/*Extrait de initUart du uC communication*/
void initUart(void){
    //Modes d'interruption :
    IEC0bits.U1TXIE = 0;//Disable UART TX interrupt
    //Mais elle activée temporairement quand une trame FSK doit être envoyée
    //Donc il faut config la suite :
    U1STAbits.UTXISEL0 = 0;
    U1STAbits.UTXISEL1 = 0;//Déclenche une interruption dès qu'il est possible
                           //d'écrire dans le registre d'envoi (SI U1TXIE==1)
}
```

Code source 3.5 : Configuration de l'interruption TX de l'UART audio.

microcontrôleurs vont être construits à partir de ce format.

3.3.1 Division d'une trame de FSK

Pour rappel le format des trames à la sortie du démodulateur FSK est donné à la figure 3.1. C'est cette trame qui doit être divisée en deux trames

Données	
Ordre	Paramètre
2 bits	8 bits

FIG. 3.1 : 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 3.2.

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.

[1] Voir la section 3.3.3 sur le récepteur propulsion pour voir pourquoi l'ordre est envoyé dans la deuxième trame.

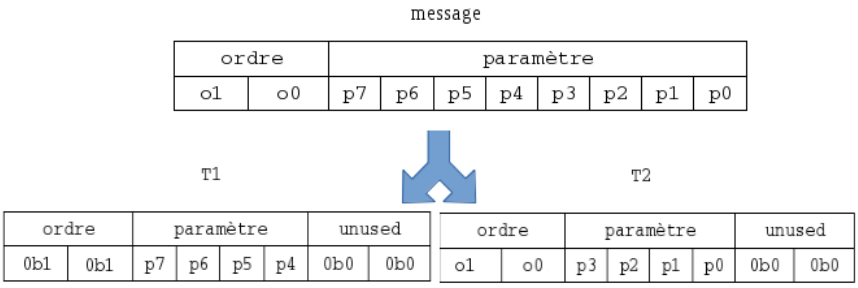

FIG. 3.2 : Division du message en deux trames UART.

3.3.2 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, 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 le code 3.6.

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

3.3.3 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 le code 3.7.

[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.

```
/*Extrait de communication.X/uart.c*/
int command;
char senderState;

void sendCommand(int newCommand){
    command = newCommand
    IEC0bits.U1TXIE = 1;//Demande  à être interrompu dès
                        //qu'une transmission est possible
    if (U1STAbits.UTXBF == 0){//Si on peut déjà commencer à écrire
        //Envoie la première partie
        char part1 = (command & 0b0000000011110000)/4 + 0b11000000;
        //Construit la première partie
        U1TXREG = part1;//Envoie la première partie
        senderState = 1;//Etat passe à 1 : première partie envoyée
    else{
        senderState = 0;//Etat=0 Il reste tout à envoyer
    }
}

void _ISR _U1TXInterrupt(void){
    //On peut envoyer une partie de commande
    IFS0bits.U1TXIF = 0;

    //Envoyer la partie 1 ou 2 selon l'état, et switcher l'état
    if (senderState == 0){
        char part1 = (command & 0b0000000011110000)/4 + 0b11000000;
        U1TXREG = part1;
        senderState = 1;
    }else{
        char part2 = (command & 0b0000000000001111)*4
                 +(command & 0b0000001100000000)/4;
        U1TXREG = part2;
        IEC0bits.U1TXIE = 0;
        //Les deux parties ont été envoyées, plus besoin d'être interrompu
        //jusqu'à la prochaine commande à envoyer
    }
}
```

Code source 3.6 : Émetteur communication – machine à état séquentielle.

```
/*Extrait de propulsion.X/uart.c*/
char receiverState;
//0 :Attend le début d'une nouvelle commande (0b11----00)
//1 :Attend la fin d'une commande en cours (0b..----00)
unsigned char param;

/*Déjà présentée plus haut, mais cette fois-ci
traitement de l'erreur.*/
void _ISR _U1RXInterrupt(void){
    IFS0bits.U1RXIF = 0;
    if ((U1STAbits.PERR || U1STAbits.FERR )== 0 ){
    //Check erreur de parité ou de formattage
        char received = U1RXREG;
        handleReceived(received);
    }else{
        char garbage = U1RXREG;
        //Nouveau :traitement de l'erreur :
        //état=0 et demande la répétition du message
        receiverState = askRepeat();
    }
}

void handleReceived(char received){
    if ((received & 0b00000011) == 0b00000000){
    //Deux derniers bits toujours nuls dans le cas d'une commande correcte.
        if ((received & 0b11000000) == 0b11000000){
        //Deux premiers bits sont 0b11 si première partie de commande
            receiverState = handleParam1(received);
        }else{
        //0b00,01,10 :Deuxième partie de commande
            receiverState = handleParam2(received);
        }
    }else{
    //Erreur
        receiverState = askRepeat();
    }
}
```

```
char handleParam1(char received){
//Le char reçu semble contenir la première partie du paramètre.
    if (receiverState == 0){
    //OK ça colle avec l'état du receiver.
        param = (received & 0b00111100)*4;
        //Les 4 bits du milieu => Les 4 MSBs du param
        return 1;
        //Passe à l'état 1 puisqu'on a reçu la première partie d'une commande.
    }else{
    //Erreur, on s'attendait à la fin d'une commande.
        return askRepeat();
    }
}

char handleParam2(char received){
//Le char reçu semble contenir la deuxième partie du paramètre et la commande.
    if (receiverState == 0){
    //Erreur, on a jamais reçu de première partie
        return askRepeat();
    }else{
    //OK, ça colle avec l'état du receiver.
        param = param + (received & 0b00111100)/4;
        //Les 4 bits du milieu => Les 4 LSBs du param
        unsigned char command = (received & 0b11000000)/64;
        //Les 2 bits du début => les 2 bits de commande
        interpretCommand(command, param);
        //Commande traitée.
        return 0;
        //Retourne à l'état 0 pour recevoir une nouvelle commande.
    }
}
```

Code source 3.7 : Récepteur propulsion – machine à état séquentielle.

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.

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.

3.3.4 Émetteur propulsion

L'émetteur propulsion n'envoie que des messages de une trame, et ce relativement rarement. On peut donc se contenter du code 3.8. Une trame `0x01` demande la répétition du message. Il s'agit donc aussi de la seule trame que le récepteur communication peut doit gérer. Ce récepteur est donc lui

```
/*Extrait de propulsion.X/uart.c*/
char askRepeat(){
//Il y a eu une erreur. Demande de répéter une commande entière et état=>0.
    if (U1STAbits.UTXBF == 0){
        U1TXREG = 0b00000001;
    }else{
        //Doesn't ever happen
    }
    return 0;
}
```

Code source 3.8 : Émetteur propulsion.

aussi très simple. Il est présenté dans la section suivante.

3.3.5 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 dans le code 3.9. 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 3.7 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.

```
/*Extrait de communication.X/uart.c*/
void _ISR _U1RXInterrupt(void){
    IFS0bits.U1RXIF = 0;
    if ((U1STAbits.PERR || U1STAbits.FERR )== 0 ){
        char received = U1RXREG;
        handleReceived(received);
    }else{//askRepeat ici aussi?
     //On a jamais eu de souci avec l'uart (2 fils sur 5 cm aussi) donc l'error
     //catching est assez rudimentaire
    }
}

void handleReceived(char received){
    if (received == 1){
    //askRepeat effectué côté propulsion
        sendCommand(0xFF);
    }//nothing else so far
}

//déjà présenté plus haut. Maintenant avec possibilité
//de renvoyer la commande précédente
void sendCommand(int newCommand){
    /*Renvoie la commande en cours si newCommand = 0xFF, sinon remplace la
     *commande est cours et envoie la nouvelle commande*/
    if (newCommand != 0xFF){
        command = newCommand;
    }
    //Le reste est inchangé
    //...
}
```

Code source 3.9 : Récepteur communication.

Chapitre 4

Interprétation des ordres

Table des figures

Table des codes source