

Rapport de laboratoire

Ecole supérieure
Électronique

Laboratoire MINF
Salle R112

PWM et RS232

Réalisé par :

Jérémie Jean-Elie
Kirian Gruber

A l'attention de :

Philippe Bovey
Serge Castoldi

Dates :

Début du laboratoire : 22 septembre 2005
Fin du laboratoire : 28 janvier 2026

Table des matières :

PWM et RS232	1
1 Cahier des charges.....	5
2 Changements dans app.c	5
3 Fonction SendMessage	6
4 ISR de l'UART	8
5 Fonction GetMessage.....	8
6 Mesures	11
6.1 Schémas de mesures	11
6.1.1 Mesure RX.....	11
6.1.2 Mesure TX	11
6.1.3 Mesure trame erronée	12
6.2 Protocole de mesures	12
6.3 Mesure RX.....	13
6.3.1 App C#.....	13
6.3.2 Mesure à l'oscilloscope	14
6.4 Mesure TX	15
6.4.1 Mesure à l'oscillogramme	15
6.4.2 Application C#.....	16
6.5 Mesure trame erronée	17
6.5.1 Application C#.....	17
6.5.2 Mesure à l'oscilloscope	18
7 Conclusion	19

1 Cahier des charges

Voir donnée en annexe

2 Changements dans app.c

```
if(sendCounter == 5)
{
    //Envoi valeurs
    if(CommStatus == false)
    {
        //Gestion du code en local (Comme sur le TP1)
        //Appel de la fonction SendMessage
        SendMessage(&pData);
    }
    else
    {
        /*Gestion du code en Remote*/
        /*Appel de la fonction SendMessage*/
        SendMessage(&pDataToSend);
    }
}
else
{
    sendCounter++;
}
```

Le compteur s'occupe d'appeler la fonction SendMessage tous les 5 cycles. Un test est ensuite effectué pour vérifier que la carte est branchée ou pas.

3 Fonction SendMessage

```
// Compose le message (Start)
TxMess.Start = 0xAA;

// Compose le message (Speed)
TxMess.Speed = pData->SpeedSetting;

// Compose le message (Angle)
TxMess.Angle = pData->AngleSetting;
```

Le début du message est d'abord composé de manière simple avec le bit de start, puis les valeurs de vitesse et d'angle signées. Les différentes valeurs sont enregistrées dans la structure des valeurs à envoyer.

```
//Calcul CRC
uint16_t valCRC16 = 0xFFFF;
valCRC16 = updateCRC16(valCRC16, 0xAA);
valCRC16 = updateCRC16(valCRC16, pData->SpeedSetting);
valCRC16 = updateCRC16(valCRC16, pData->AngleSetting);
```

Ces lignes de codes ont été reprises de la données. Elles permettent de calculer le CRC à partir des 3 premières valeurs de la trame. À savoir, le bit de start, la valeur de vitesse et de l'angle. La fonction « updateCRC16 » est justement appelée pour calculer le CRC.

```
//Séparation de ValCrc16 en 2 bytes
uint8_t valCRC_Byte_1; //Byte MSB (envoyé en premier)
uint8_t valCRC_Byte_2; //Byte LBS (envoyé en dernier)

valCRC_Byte_1 = (uint8_t) (valCRC16 >> 8);
valCRC_Byte_2 = (uint8_t) (valCRC16 & 0x00FF);
```

Étant donné que le CRC est sur 16 bits et qu'il n'est que possible d'envoyer un bit à la fois, il faut donc séparer la valeur du CRC pour la répartir sur 2 variables de 8 bits.

Le premier bit du CRC envoyé est celui qui contient le MSB. Il faut donc décaler 8 fois tous les bits par la droite pour ne garder que la partie utile.

Avant décalage :

MSB															LSB
0	1	0	1	1	1	0	0	1	1	1	1	1	1	1	1

Après décalage :

MSB															LSB
								0	1	0	1	1	1	0	0

De cette manière, il est donc possible de stocker dans une variable 8 bits les 8 bits MSB d'une variable 16 bits.

Pour enregistrer les 8 bits LSB du CRC, il suffit de faire un masquage « AND » de bits pour être sûr de ne prendre que ces 8 bits LSB.

Masquage :

MSB															LSB
0	1	1	0	1	1	1	0	0	1	0	1	1	1	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	0

En vert, la partie du CRC que l'on veut enregistrer dans la variable 8 bits et qui ne sera pas affectée par le masquage. Et en orange, la partie non utile que le n'on souhaite pas garder et qui sera donc affectée par le masquage. En faisant un masquage « AND », tous les bits masqués par un « 0 » seront mis à « 0 ». Et les bits masqués par un « 1 » garderont leur valeur.

```
// Compose le message (MsbCrc)
TxMess.MsbCrc = valCRC_Byte_1;

// Compose le message (LsbCrc)
TxMess.LsbCrc = valCRC_Byte_2;
```

Les deux variables 8 bits sont ensuite enregistrées dans le tableau des valeurs à envoyer.

```
// Dépose le message dans le FIFO (Start)
PutCharInFifo(&descrFifoTX, TxMess.Start);
// Dépose le message dans le FIFO (Speed)
PutCharInFifo(&descrFifoTX, TxMess.Speed);
// Dépose le message dans le FIFO (Angle)
PutCharInFifo(&descrFifoTX, TxMess.Angle);
// Dépose le message dans le FIFO (LSBCRC)
PutCharInFifo(&descrFifoTX, TxMess.MsbCrc);
// Dépose le message dans le FIFO (MSBCRC)
PutCharInFifo(&descrFifoTX, TxMess.LsbCrc);
```

Les valeurs du tableau sont ensuite toutes chargées dans le registre FIFO pour l'envoi.

4 ISR de l'UART

L'écriture de l'ISR s'est basée sur le cours de MINF. Celle-ci a pu être implémentée sans problèmes et sans changer de paramètres. À noter que le code comprenait déjà les fonction de commutations des LEDs qui serviront pour les mesures.

5 Fonction GetMessage

La fonction « **GetMessage** » a été modifiée par rapport à la version fournie dans le structogramme. La version annotée par Mr Bovey du structogramme se trouve en annexe de la version papier de ce rapport.

Le tableau « **valeursMessage[5]** » a été remplacé par la structure « **StruMess** » déjà présente dans le fichier source « **Mc32gest_RS232.c** » fourni par les enseignants.

```
// Structure décrivant le message
typedef struct {
    uint8_t Start;
    int8_t Speed;
    int8_t Angle;
    uint8_t MsbCrc;
    uint8_t LsbCrc;
} StruMess;

// Struct pour émission des messages
StruMess TxMess;
// Struct pour réception des messages
StruMess RxMess;
```

Cette structure nous a aussi permis de nous débarrasser de la variable « **valeurLue** » car nous pouvons utiliser « **RxMess.Start** » pour vérifier si l'octet de début de la trame est le bon.

```
/*Vérification si le caractère de début du message est le bon*/
if(RxMess.Start == AA)
{
```

AA a été défini dans le fichier header « **Mc32gest_RS232.h** ».

```
#define AA 0xAA
```

La fonction « **Lecture1Octet()** » a été remplacé par la fonction « **GetCharFromFifo()** » déjà présente dans le fichier source « **GesFifoTh32.c** » fourni par les enseignants.

Elle nous permet de faire du passage par référence afin d'extraire de la trame le caractère voulu et l'enregistrer dans la structure « **StruMess** » au bon endroit.

```
/*Récupération du caractère de début du message*/
GetCharFromFifo(&descrFifoRX, &RxMess.Start);
/*Vérification si le caractère de début du message est le bon*/
```

Cette fonction nous a aussi permis de remplacer le « **for** », utiliser pour enregistrer les autres octets de la trame dans le tableau « **valeursMessage[5]** », par 4 appels de cette fonction.

```
/*Récupération des caractères de vitesse, angle et
*du CRC (MSB puis LSB)*/
GetCharFromFifo(&descrFifoRX, &RxMess.Speed);
GetCharFromFifo(&descrFifoRX, &RxMess.Angle);
GetCharFromFifo(&descrFifoRX, &RxMess.MsbCrc);
GetCharFromFifo(&descrFifoRX, &RxMess.LsbCrc);
```

La fonction de vérification du CRC « **VerifierCRC()** » a été remplacée par la fonction « **updateCRC16()** » déjà présente dans le fichier source « **Mc32calCrc.c** » fourni par les enseignants.

```
/*Calcul du CRC avec les valeurs les valeurs Start, Vitesse  
et Angle*/  
verifCRC = updateCRC16(verifCRC, RxMess.Start);  
verifCRC = updateCRC16(verifCRC, RxMess.Speed);  
verifCRC = updateCRC16(verifCRC, RxMess.Angle);
```

Les fonctions « **VerifierVitesse()** » et « **VerifierAngle()** » ont été faite sous le nom de « **verifVitesse** » et « **verifAngle()** ».

```
/*Vérification que la valeur de la vitesse ne  
*dépasse pas -99 ou +99*/  
verifVitesse = VerifierVitesse(RxMess.Speed);  
/*Vérification que la valeur de l'angle ne  
*dépasse pas -90 ou +90*/  
verifAngle = VerifierAngle(RxMess.Angle);
```

Le but de ces fonctions est de vérifier si la valeur de la vitesse et de l'angle reçu ne dépasse pas la plage de valeur maximale autorisée (-99 à +99 pour la vitesse et -90 à +90 pour l'angle).

Fonction verifVitesse :

```
//verifvitesse  
bool VerifierVitesse(int8_t valVitesse)  
{  
    uint8_t valAbsAngle = 0;  
    bool valToReturn = false;  
    valAbsAngle = abs(valVitesse);  
    if(valAbsAngle > VITESSE_MAX)  
    {  
        valToReturn = false;  
    }  
    else  
    {  
        valToReturn = true;  
    }  
    return valToReturn;  
}
```

Fonction verifAngle :

```
//Verif Angle  
bool VerifierAngle(int8_t valAngle)  
{  
    uint8_t valAbsAngle = 0;  
    int8_t valAngleLocal = 0;  
    bool valToReturn = false;  
    valAngleLocal = valAngle;  
    valAbsAngle = abs(valAngleLocal);  
    if(valAbsAngle > ANGLE_MAX)  
    {  
        valToReturn = false;  
    }  
    else  
    {  
        valToReturn = true;  
    }  
    return valToReturn;  
}
```

Dans cette fonction, nous avons copié la valeur reçue dans une variable locale (valAngleLocal) pour être sûr de ne pas modifier le signe (positif ou négatif) de « **RxMess.Angle** ».

Nous avons fait cela car lors de nos mesures nous nous sommes rendu compte qu'en mode Remote, le servomoteur prenait la valeur absolue de l'angle pour effectuer ses rotations. Il ne pivotait que de 0° à +90°. Cette modification n'a pas résolu le problème et a été laissée là pour montrer l'une des pistes explorées lors de la tentative de résolution du problème. Nous avons relevé ce point dans la fiche d'autocontrôle.

La dernière modification majeure par rapport au structogramme est le « **SI** » de comparaison du nombre d'erreur.

Nous avons enlevé la remise à 0 de la variable erreur et avons mis un « **SI** » qui force la variable erreur à 10 si la valeur de cette dernière dépasse 20.

De cette manière, on reste en mode local une fois les 10 cycles d'erreur dépasser et on ne revient en mode remote que si une trame est reçue correctement. Cela évite que le programme ne passe en mode local tous les 10 cycles et que pour 1 cycle.

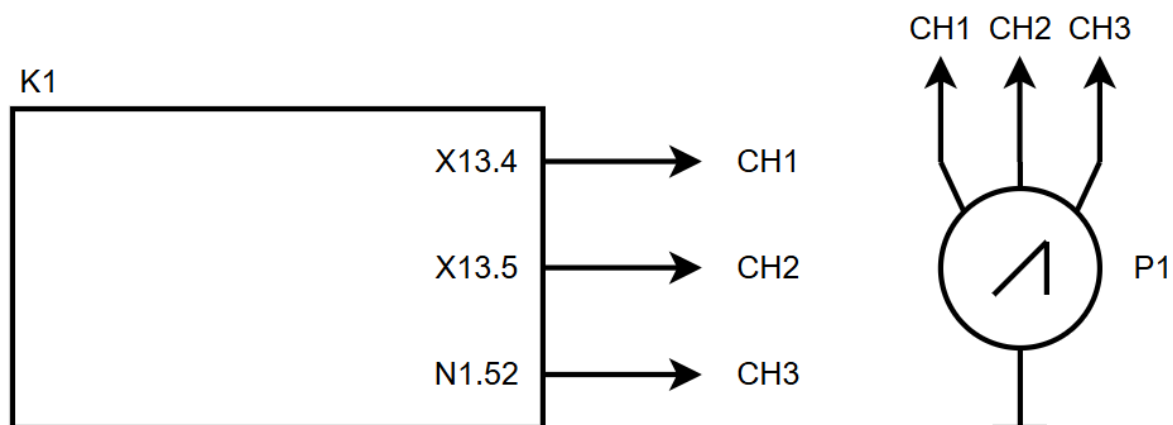
La remise à 10 de la variable erreur évite l'overflow de cette variable, vu que cette dernière est incrémentée tant qu'une trame valide n'est pas reçue.

```
if(erreur >= 10)
{
    commStatus = false;
    if(erreur > 20)
    {
        erreur = 10;
    }
}
else
{
    commStatus = true;
}
```

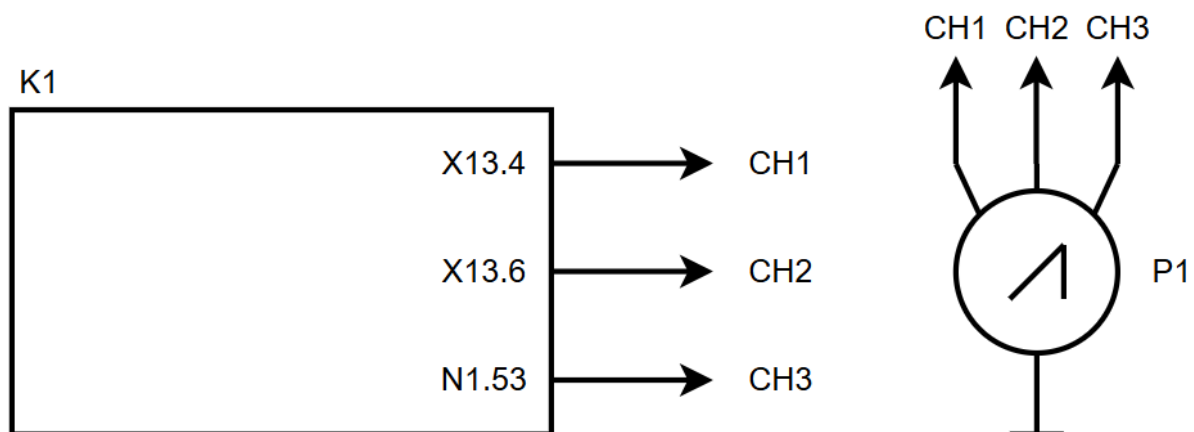
6 Mesures

6.1 Schémas de mesures

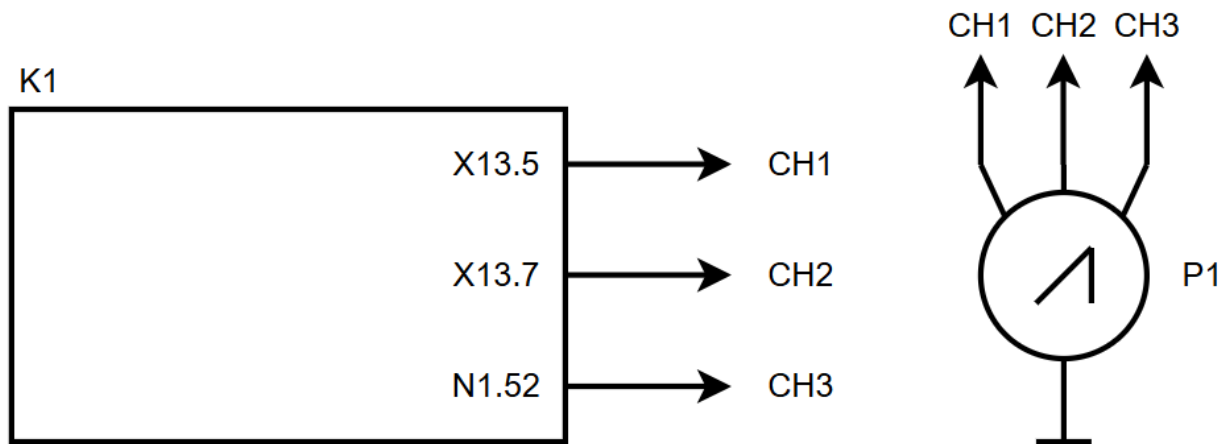
6.1.1 Mesure RX



6.1.2 Mesure TX



6.1.3 Mesure frame erronée



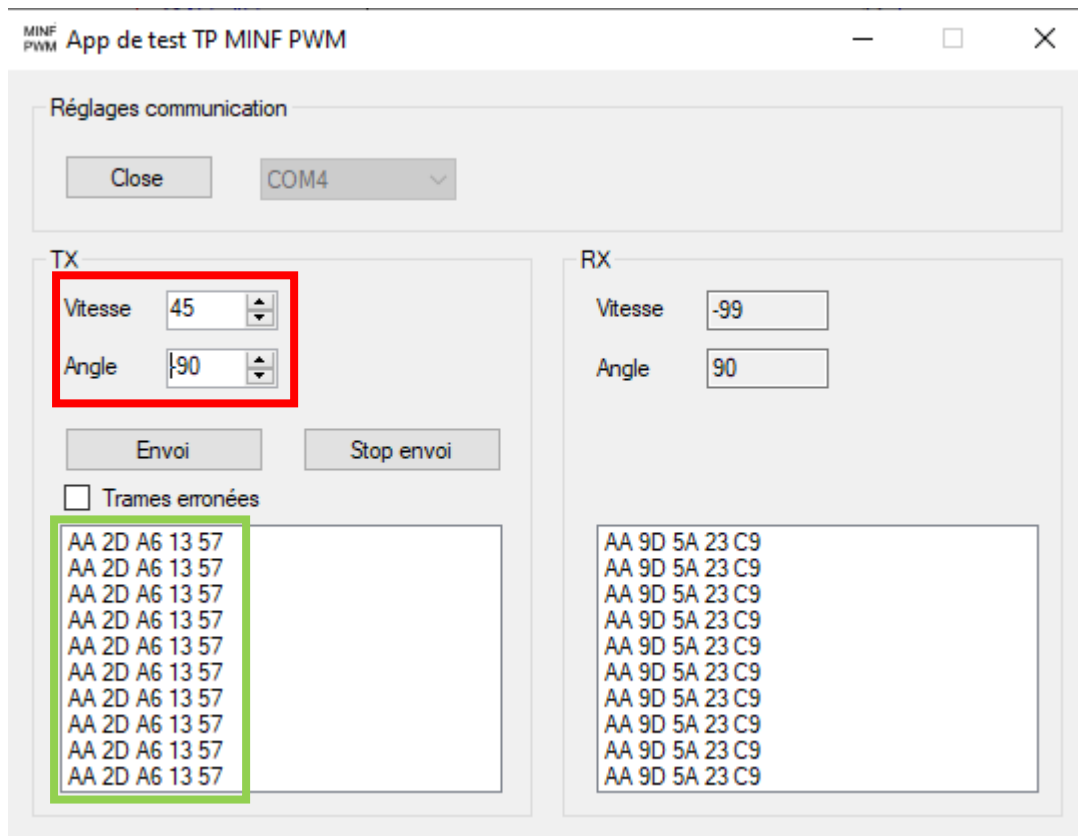
6.2 Protocole de mesures

Pour détecter la trame UART, il faut presser le bouton « Protocol » sur l'oscilloscope et sélectionner le bon type de bus dans « Bus Type ». Dans l'onglet « Trigger Setup », veiller à ce que l'oscilloscope décode bien sur le bon canal (ici, canal3). Définir le bitrate à 57.6kbits et ne pas mettre de parité.

6.3 Mesure RX

6.3.1 App C#

Pour mesurer RX, il faut d'abord envoyer des valeurs depuis l'application C#.

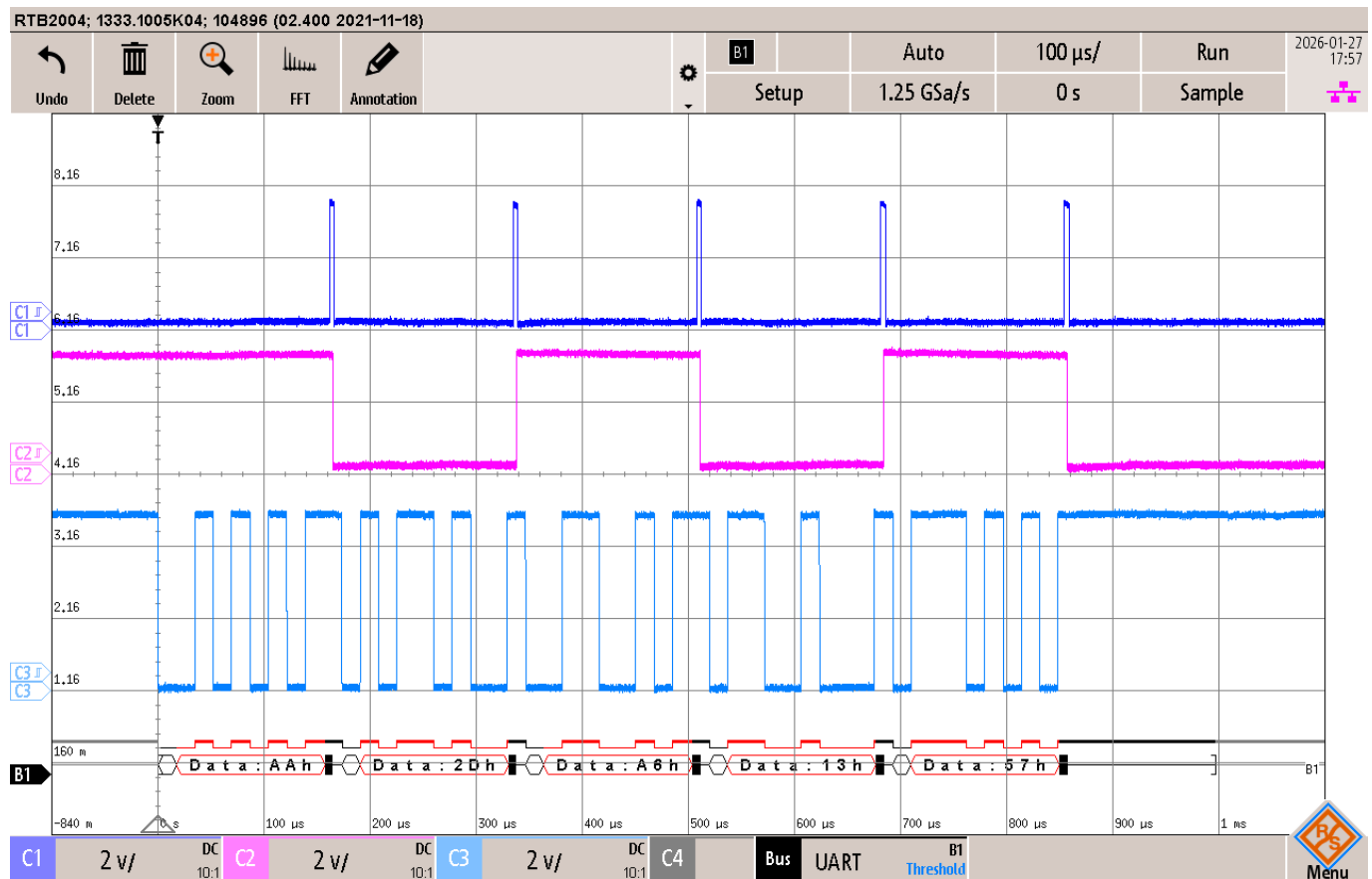


La raison pour laquelle les valeurs envoyées se trouvent dans la zone TX est que les valeurs sont envoyées depuis une app externe, donc par TX. Or, sur la carte programmée, ce canal est connecté au port RX.

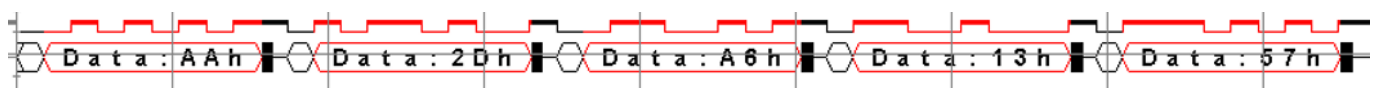
Les valeurs à réceptionner sont donc dans le carré rouge. Noter aussi la trame envoyée (dans le carré vert).

Pour rappel : Start : AA, Vitesse : 2Dh -> 45, Angle : A6 -> -90, CRC : 13 57

6.3.2 Mesure à l'oscilloscope



La LED3 (canal 1), s'éteint lors de la durée de l'interruption de l'UART. Dans l'ISR de l'UART, la LED4 (canal 2) est commutée à chaque fois que l'interruption est d'origine RX. C'est-à-dire qu'à chaque fois que la carte reçoit une trame valide, elle fait commuter la LED4.

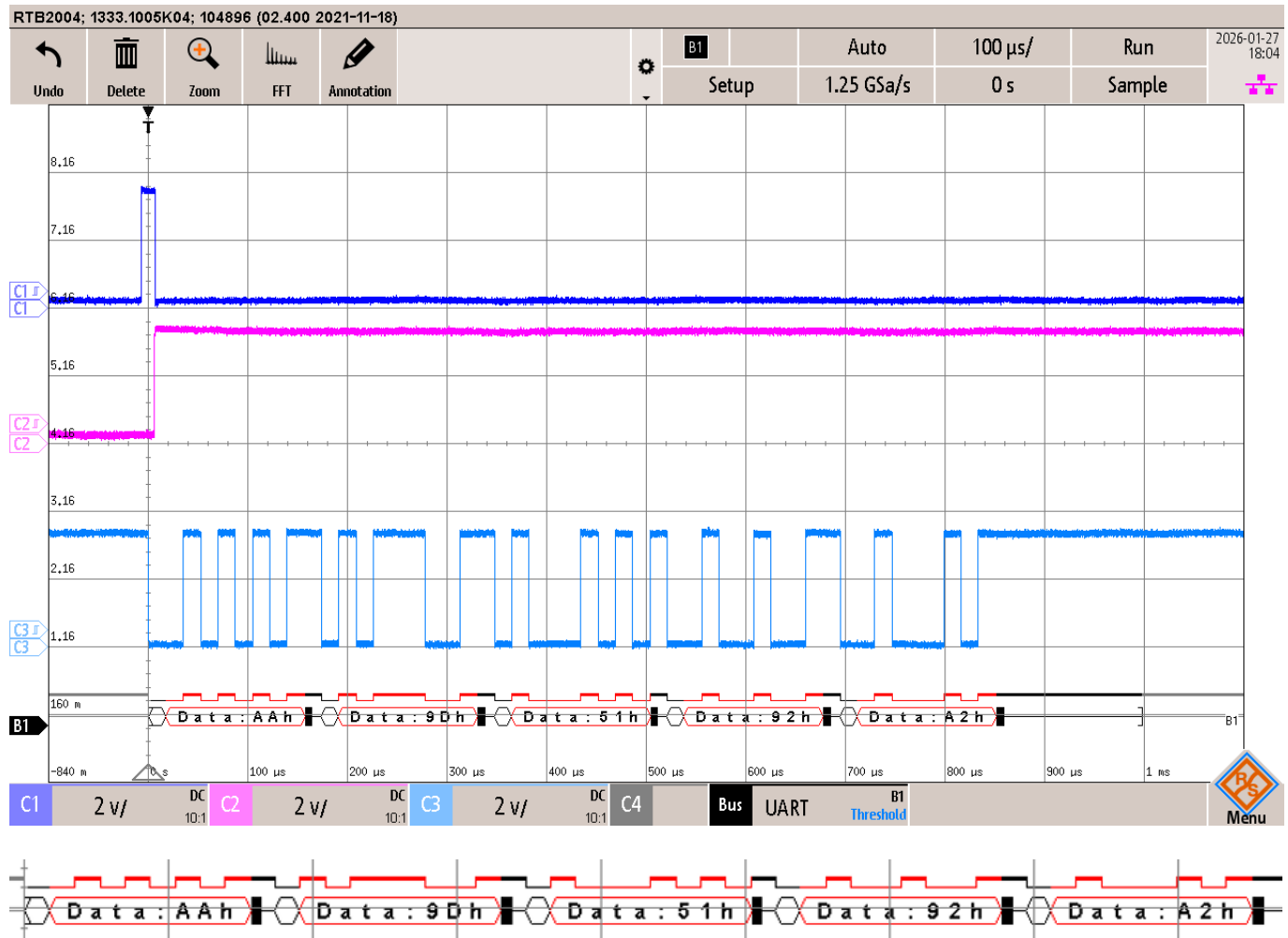


La trame observée correspond bien à la trame envoyée par l'application C#.

6.4 Mesure TX

6.4.1 Mesure à l'oscillogramme

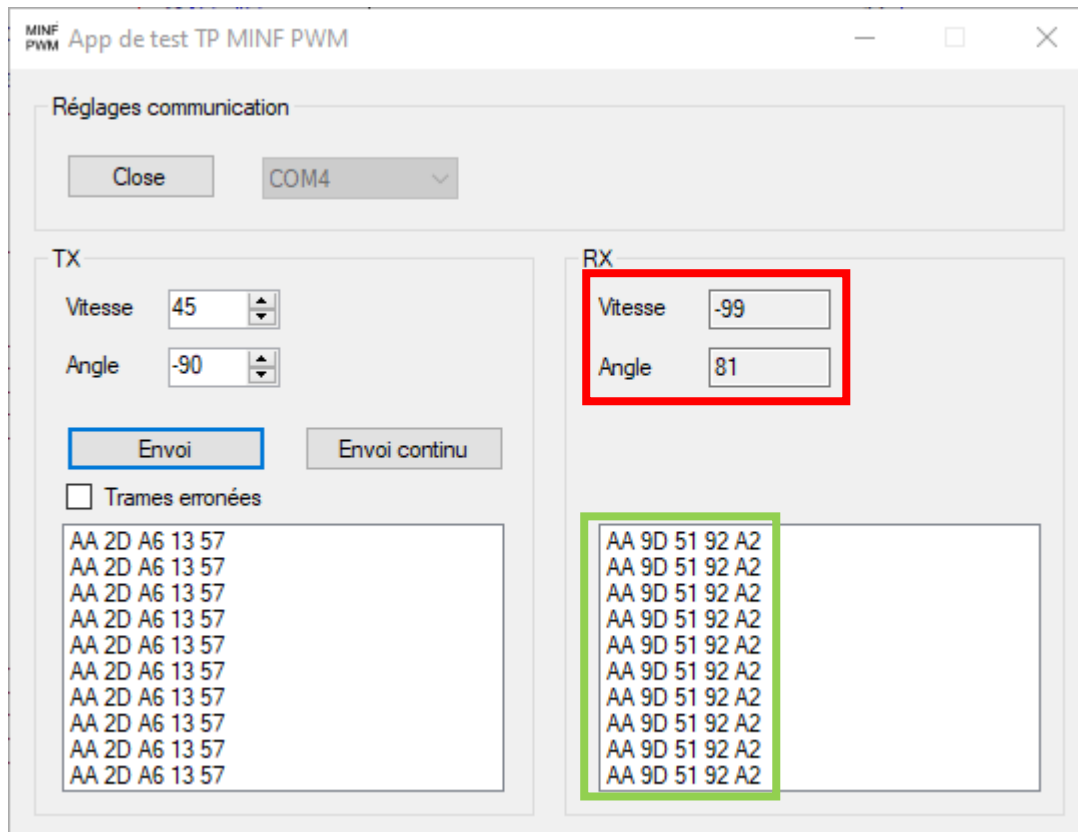
Cette fois-ci, il faut mesurer le canal TX de la carte électronique. Celle-ci envoie sans arrêt ses valeurs de vitesse et d'angle.



La LED3 (canal 1) s'éteint à nouveau durant l'interruption de l'UART. La LED5 commute à chaque envoi de trame.

Les valeurs envoyées ici sont les suivantes : Vitesse à -99 et Angle à 81

6.4.2 Application C#

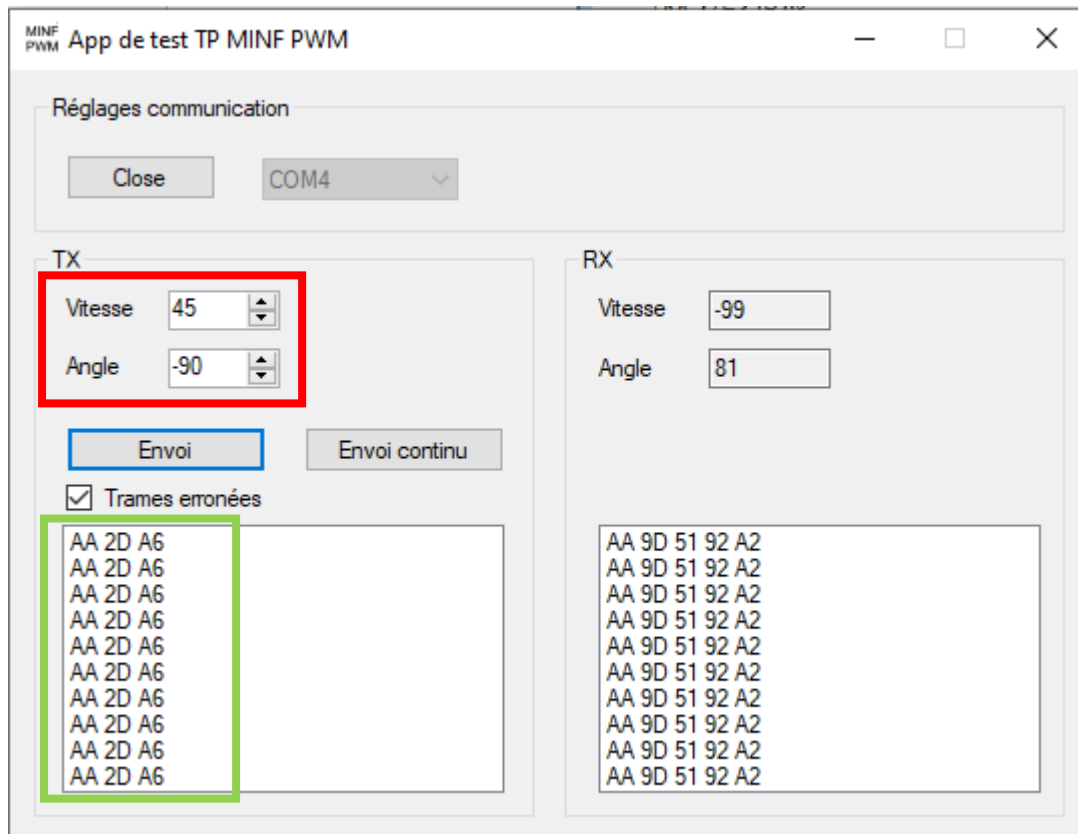


Lorsque l'application réceptionne des données de la carte, celles-ci s'affichent dans la zone RX. Dans le carré rouge, on retrouve les valeurs envoyées par la carte. Il est à nouveau possible de vérifier dans le carré vert que le trame reçue est bien identique à la trame envoyé. Dans ce cas, la trame est bien identique.

6.5 Mesure trame erronée

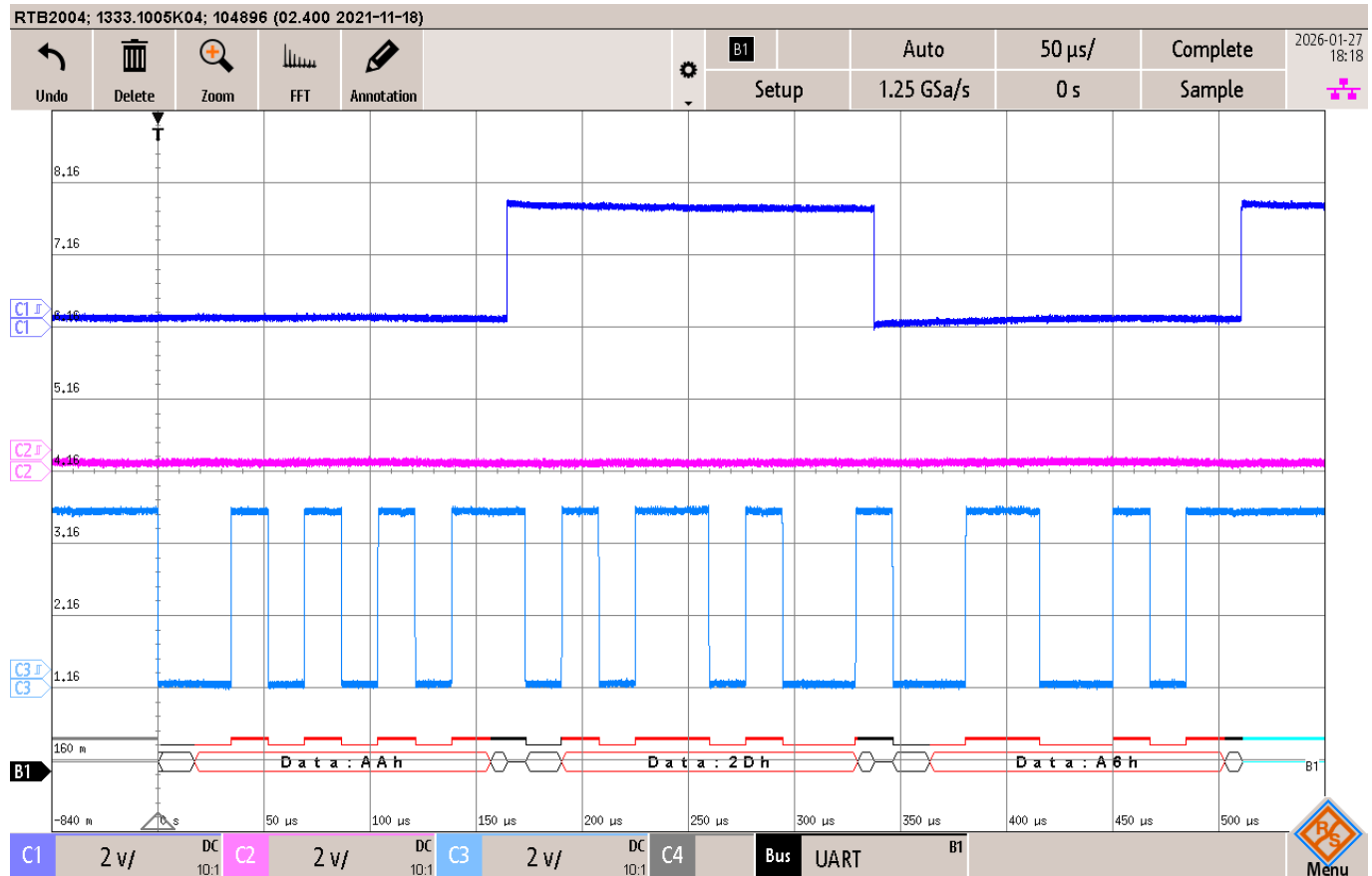
6.5.1 Application C#

Dans l'application C#, il est possible d'envoyer à la carte électronique une trame erronée.



Lorsque l'application envoie des trames erronées, elle n'envoie en fait pas de CRC. La carte électronique devrait donc pas prendre en compte ces trames et devrait aussi allumer la LED6 pour indiquer réception d'une trame fausée.

6.5.2 Mesure à l'oscilloscope



La LED4 commute à nouveau à chaque fois qu'elle reçoit un bit de trame valide. La LED6, quant à elle, s'allume quand la carte reçoit une trame erronée. Il est aussi possible de voir que l'on reçoit bien les valeurs de vitesse et d'angle mais qu'il n'y a pas de CRC ensuite.

7 Conclusion

Le programme fonctionne correctement. Celui-ci passe bien en mode Remote lorsque la carte communique avec l'application. La carte envoie bien les octets de start, de vitesse, d'angle et de CRC et reçoit bien les octets de start, de vitesse, d'angle et de CRC de l'application et les affiche sans erreurs.

Lorsque la carte est en mode Remote, la première ligne du LCD affiche « mode Remote » et lorsque la carte est en mode local, la première ligne du LCD affiche « mode local ».

L'ISR de l'UART a été complétée grâce à la théorie de MINF qui donnait déjà à disposition la totalité du code nécessaire. Il n'a pas fallu changer de paramètres pour que celle-ci fonctionne avec le reste du programme.

Durant l'écriture de la fonction « **SendMessage** », une petite erreur a d'abord été faite sur la composition du CRC. Le bit LSB était envoyé en premier au lieu du bit MSB. Cette mauvaise composition a donc eu pour effet de créer une erreur dans l'application C# qui empêchait la carte de communiquer correctement avec l'application. Après une correction, la carte pouvait correctement communiquer. Lors des mesures, il a été confirmé que les bonnes données étaient transmises à l'application

Lors de la conception de la fonction « **GetMessage()** » nous nous sommes rendu compte que les fichiers sources et headerfiles complémentaires fournis par les enseignants avaient déjà des fonctions et des structures nous permettant de récupérer les octets de la trame et de les « traiter ».

Cela a amené à une modification de la conception de la fonction et nous a permis de gagner du temps, notamment sur la gestion du CRC.

Le problème du servomoteur est apparu lors du remplissage de la fiche d'autocontrôle et nos différentes tentatives pour régler ce problème se sont avérées être des échecs. Nous ne comprenons pas d'où peut venir cette erreur.

Comme indiqué au début de ce chapitre, la fonction « **GetMessage()** » reçoit bien les trames envoyées par l'application et affiche correctement leurs valeurs sur le LCD.

Lausanne, le 27.01.2026

Jérémie Jean-Elie, Kirian Gruber

Annexes :

- Structogramme
- Fiche d'autocontrôle