DSPIC30F3011

Liaison série asynchrone

COUCHE PHYSIQUE

Les liaisons au format série asynchrone sont très courantes. Par exemple :

- en courtes distances (qq mètres) :
  - liaisons PC ↔ PC
  - liaisons PC ↔ périphériques lents (imprimantes, modem, …)
- en longues distances par fil : on utilise le réseau téléphonique ce qui nécessite un modem (V21, V23, etc.)
- en courtes distances par infrarouge (télécommandes)
- en courtes et longues distances par radio (modem nécessaire).

Avantages :

– Facilité de mise en œuvre
– Bonne immunité au bruit

Inconvénients :

– Vitesse très inférieure au format série synchrone
– Protocole assez flou (couches hautes de l'ISO) comparé à celui du bus USB qui a été pensé à l'origine "Plug and Play"

1. Le format série asynchrone

Les messages à transmettre sont constitués d'un ensemble de bits structurés de façon très précise (fichier de données informatiques par exemple). Cette structure est accompagnée de règles complémentaires nécessaires au transfert entre l'émetteur et le récepteur. Bien entendu, les mêmes règles doivent être utilisées des 2 côtés du canal de transmission.
Pour éviter une certaine anarchie, des organisations internationales (ETSI, CCITT, …) ont établi des règles rigoureuses décrites dans des documents appelés "standards" ou "recommandations".

Pour faciliter le traitement du message, les bits constitutifs sont regroupés en mots de 5 bits (code BAUDOT), 7 bits (code Ascii) ou par octets. Ces éléments de base sont nommés **symboles**.

Dans toute liaison série (synchrone ou asynchrone), le récepteur doit être capable de reconstituer le message dans son entier alors que les bits lui arrivent les uns après les autres.
Dans le format synchrone, cette opération nécessite d'ajouter au message utile des informations complémentaires dont le traitement dans le récepteur est assez complexe.
Dans le cas du **format asynchrone**, la reconstitution du message est facilitée car chaque symbole est accompagné de bits complémentaires de synchronisation et de contrôle :

– bit **Start** : indicateur du début de la transmission d'un symbole,
– bit **Stop** : indicateur de fin de transmission d'un symbole (dans certains cas la durée de cet indicateur est rallongée : on place 1,5 ou 2 bits Stop),
– bit de **Parité** (optionnel) : précède le bit Stop. Permet la détection d'une erreur de transmission.

L'ensemble des bits : Start + Données + Parité + Stop est appelé : **trame**.

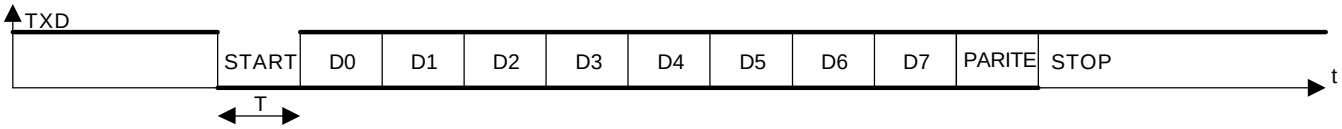
1.1 Chronogrammes typiques

On suppose que les symboles ont une longueur de 8 bits (cas le plus fréquent).
Identification des signaux caractéristiques :

- TXD : signal binaire transmis par l'émetteur
- TXC : horloge de transmission de l'émetteur (signal non transmis)
- RXD : signal binaire reçu par le récepteur. En principe identique à TXD s'il n'y a pas d'erreur
- RXC : horloge de réception reconstituée dans le récepteur

1.1.1 Transmission d'un symbole isolé

Un symbole est dit "isolé" quand il n'est ni précédé, ni suivi de la transmission d'un autre symbole.



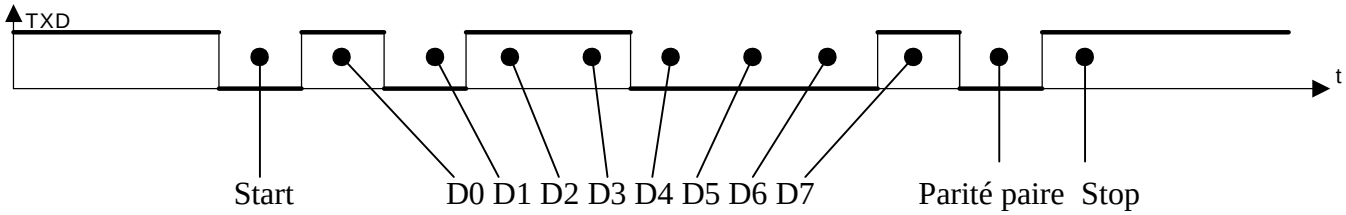
Le signal TXD est à "1" au repos, c'est à dire en l'absence de toute transmission (signal MARK).
Le bit "Start" est toujours à "0" et le bit "Stop" toujours à "1".
Les bits du symbole sont D0 à D7. D0 est toujours transmis en premier
Bit de parité :

- si on choisit une parité paire : état du bit de parité tel que le nombre d'états "1" (sur D0 à D7, y compris le bit de parité) est pair
- si on choisit une parité impaire : état du bit de parité tel que le nombre d'états "1" (sur D0 à D7, y compris le bit de parité) est impair

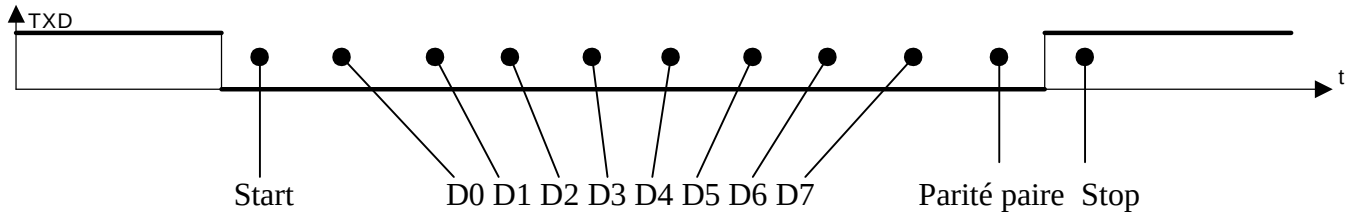
La durée T définit le débit de moments (nombre maximum de changements d'états par seconde) :

$$R = \frac{1}{T} \quad \text{(en Bauds)}$$

Exemples de trames



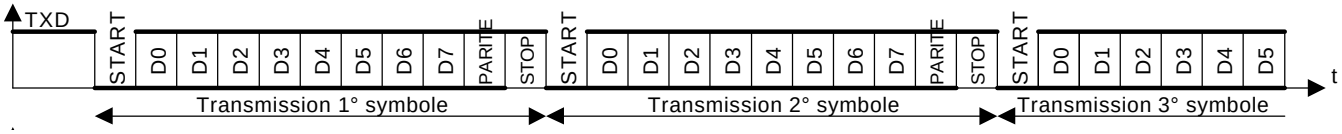
Le symbole transmis vaut ici 10001101b = 8Dh (poids fort à gauche)
Le nombre de "1" (y compris le bit de parité) est pair : CQFD



Le symbole transmis vaut ici 00000000b = 00h
Le nombre de "1" (y compris le bit de parité) est pair (il est nul) : CQFD

Note : ce dernier chronogramme est remarquable : à l'exception des bits Start et Stop, le signal TXD ne présente aucun changement d'état ! Le récepteur doit être capable d'identifier les 8 bits du symbole malgré cela.

1.1.2 Transmission de plusieurs symboles sans temps mort



On comprend, avec ce chronogramme, la présence nécessaire du bit Stop. Grâce à lui, il y a toujours un flanc descendant sur TXD au début de chaque bit Start. Si le bit Stop était absent, le bit de parité, ou D7 quand on ne transmet pas la parité, serait le dernier bit d'une trame. Celui-ci peut être à "0" et le bit Start ne crée alors pas de changement d'état et ne peut alors être identifié.
Le flanc descendant de TXD entre les bits Stop et Start est exploité dans le récepteur pour reconstituer l'horloge RXC (voir plus loin).

Exemple

Exercice : Identifier les bits de chaque symbole
Déterminer les valeurs des symboles transmis

Note : si le récepteur n'est pas adapté au format de transmission (par exemple : 8 bits, parité paire et 1 bit Stop), il va très rapidement faire des confusions. Par exemple, si le récepteur s'attend à recevoir des données au format : 7 bits, pas de parité et 1 bit Stop, les 7 premiers bits du premier symbole seront correctement identifiés et traités, mais il va interpréter le bit D7 comme un bit Stop et le bit de parité comme un éventuel bit Start. La suite de la transmission est perturbée !

Note : dans ce cas (pas de temps morts entre 2 transmissions), le débit binaire "utile" est maximum. Pour l'exemple, il vaut : $\frac{1}{T} \times \frac{8}{11}$ bits/S

1.2 Formats standards

Bien que standards, les formats sont nombreux. En effet, on peut choisir :

- Le nombre de bits du symbole : 5 , 7 ou 8 (et même 9 dans certains cas)
- La présence et le type de bit de parité : absent, parité paire, parité impaire
- La durée minimum du bit Stop (il n'y a pas de maximum !) : 1T, 1,5T ou 2T
- Le débit de moments: 110, 330, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 bauds

2. RS232/EIA232

RS signifie 'Recommanded Standard' soit en français : standard recommandé.

Dans les années 1960, un comité, actuellement nommé '**E**lectronic **I**ndustries **A**ssociation' (d'où le EIA232 des années 1990), a développé un standard d'interface de transmission de données en série entre équipements. A l'époque, il était prévu essentiellement pour des communications entre ordinateur et modem. Par la suite, il a été utilisé à d'autres fins comme la transmission de données entre des ordinateurs, entre un ordinateur et ses périphériques (imprimante, table traçante, ...), entre un ordinateur et d'autres systèmes équipés de processeur tel les commandes numériques de machines outils, ...

Ce standard défini les niveaux de tensions correspondant aux 1 et aux 0, le brochage des connecteurs, la fonction de chacun des signaux et un protocole d'échange des informations. Il permet des communications bi- directionnelles (les 2 équipements peuvent émettre en même temps, full duplex, ou l'un après l'autre, half duplex).

2.1 Caractéristiques électriques

Les tensions représentant les 1 et 0 sont relatives à la masse (0V) commune entre les 2 équipements.
Elles sont définies comme suit :

Tension	**État**
- Vmax à - 3V	1 logique, marque (mark), en attente
- 3V à 3V	zone interdite, afin éliminer les problèmes dus aux bruits sur la ligne
3V à Vmax	0 logique, espace (space), actif
Version de la norme	**Vmax**
RS232	48V
RS232A	25V
RS232B	12V
RS232C	5V

2.2 Longueur et type de câbles :

La longueur maximum théorique du câble est de 15 mètres. Dans la pratique, on se rend compte qu'avec un câble de bonne qualité, on peut largement dépasser cette longueur. Il suffit de prendre des câbles blindés (général, ou mieux par pair) pour pouvoir porter cette longueur à 25- 30 mètres, voire plus.

2.2 Connecteur 9 broches (modem ou DCE)

Broche	Signal	Type	Utilisation
1	CD	Sortie	Carrier Detect (détection porteuse du modem)
2	TD	Sortie	Transmitted Data : donnée émise. Etat de repos : "1"
3	RD	Entrée	Received Data : donnée reçue
4	DTR	Entrée	Data Terminal Ready : terminal prêt à recevoir
5	SG	Signal	Ground : masse de référence des signaux (0V)
6	DSR	Sortie	Data Set Ready : modem prêt à transmettre
7	CTS	Entrée	Clear To Send : ordre de transmission
8	RTS	Sortie	Request To Send : demande de transmission
9	RI	Sortie	Ring Indicator : indicateur d'appel

Les modem ciblés par la norme ne sont plus utilisés de nos jours. Les liaisons séries "RS232" se contentent des signaux TD et RD (en ajoutant quelquefois RTS et CTS) en connexions croisées.

FONCTIONS DE LA COUCHE LIAISON POUR MICROCONTROLEURS DSPIC

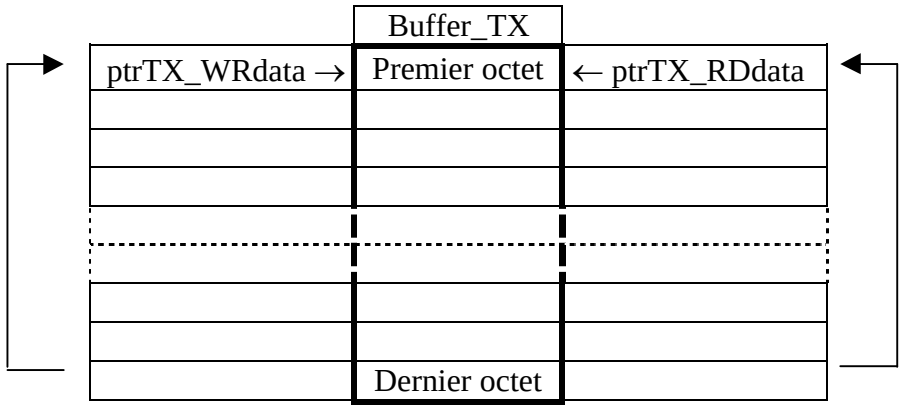
1. Principe

Les fonctions de transmission et de réception décrites n'utilisent aucun protocole de communication matériel (signaux RTS, CTS, DSR, DTR, etc.), seuls TXD et RXD sont utilisés. On ne vérifie pas que l'équipement distant est prêt à recevoir les caractères transmis.
Les **interruptions sont utilisées** pour éviter toute boucle d'attente (utilisation efficace du temps CPU) et limiter le risque de perte de caractères en réception.

Par ailleurs, les fonctions utilisent des **tampons mémoire circulaires** pour :

- éviter d'attendre la fin de la transmission en cours à la transmission d'un nouveau caractère,
- de ne pas perdre de caractères en réception car l'exploitation des données n'arrive pas toujours à suivre le rythme de réception.

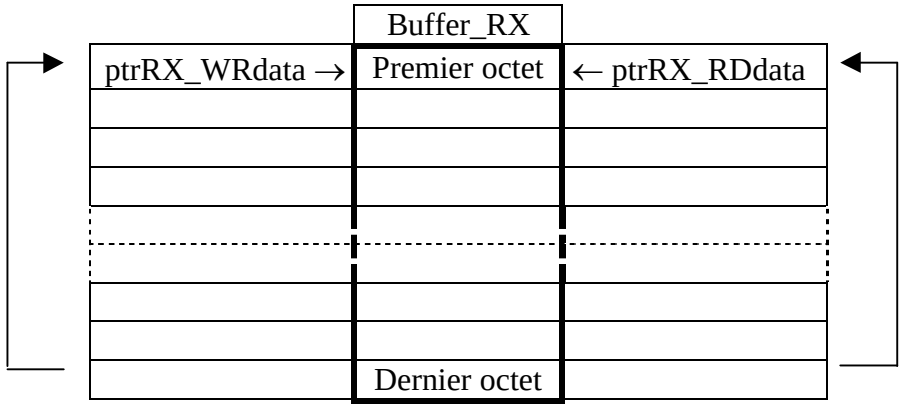
1.1 Tampon circulaire en transmission



Le "**Buffer_TXD**" est un bloc d'octets consécutifs en RAM du microcontroleur. Sa taille dépend de l'application (quelques octets à plusieurs koctets).
Les variables "**ptrTX_WRdata**" et "**ptrTX_RDdata**" sont des pointeurs (taille = 16 bits pour un dsPIC), leur contenu est une adresse qui *pointe* une des cases mémoires du buffer.
Fonctionnement :

- Au "reset" du microcontroleur, les pointeurs "ptrTX_WRdata" et "ptrTX_RDdata" sont initialisés avec l'adresse du 1° octet du buffer "Buffer_TX" (comme illustré sur la figure ci-dessus).
- Une application a des données à transmettre. Elle utilise le pointeur d'écriture "ptrTX_WRdata" pour remplir le buffer. Pour chaque octet à transmettre :
  - L'octet est rangé dans le buffer à l'adresse contenue dans le pointeur "ptrTX_WRdata".
  - Le contenu du pointeur est alors incrémenté pour pointer la prochaine case libre du buffer.
  - Si le contenu du pointeur dépasse la taille du buffer, il est affecté par l'adresse de la 1° case (d'où l'expression "circulaire").
  - S'il n'y a pas de transmission en cours (cas du premier octet par exemple), on provoque la première interruption pour lancer la fonction d'interruption de transmission.
- Une fonction d'interruption se charge de transmettre les octets placés du buffer. Cette fonction est activée quand on place le premier octet dans le buffer encore vide ou automatiquement à la fin de la transmission précédente. La fonction effectue alors les opérations suivantes :
  - Ne rien faire si le buffer circulaire est vide ("ptrTX_WRdata"=="ptrTX_RDdata").
  - Lecture de l'octet pointée par "ptrTX_RDdata" dans le buffer (prochain caractère à transmettre).
  - Déclenchement de la transmission de cet octet via le coupleur UART du dsPIC.
  - Incrémentation du contenu du pointeur "ptrTX_RDdata". Si le contenu du pointeur dépasse la taille du buffer, il est affecté par l'adresse de la 1° case (pointeur "circulaire").
  - Acquittement de l'interruption.

1.2 Tampon circulaire en réception



Le "**Buffer_RXD**" est un bloc d'octets consécutifs en RAM du microcontroleur. Sa taille dépend de l'application (quelques octets à plusieurs koctets).
Les variables "**ptrRX_WRdata**" et "**ptrRX_RDdata**" sont des pointeurs (taille = 16 bits pour un dsPIC), leur contenu est une adresse qui *pointe* une des cases mémoires du buffer.

Fonctionnement :
- Au "reset" du microcontroleur, les pointeurs "ptrRX_WRdata" et "ptrRX_RDdata" sont initialisés avec l'adresse du 1° octet du buffer "Buffer_RX" (comme illustré sur la figure ci-dessus).
- Un équipement externe transmet des données vers le coupleur UART du microcontroleur. Ce coupleur est configuré au format de transmission utilisé et provoque une **interruption** à chaque octet reçu.
- La fonction d'interruption associée se charge de ranger les octets reçus dans le buffer. Pour cela elle effectue les opérations suivantes :
  - Acquittement de l'interruption.
  - Placer l'octet reçu dans le buffer "Buffer_RXD" à l'adresse pointée par "ptrRX_WRdata".
  - Incrémentation du contenu du pointeur "ptrRX_WRdata". Si le contenu du pointeur dépasse la taille du buffer, il est affecté par l'adresse de la 1° case (pointeur "circulaire").
- Une fonction doit lire les données stockées dans le buffer sous peine de le saturer et de perdre les premiers octets reçus. Cette fonction lit le buffer en respectant l'algorithme suivant :
  - Indiquer que le buffer est vide si les pointeurs "ptrRX_WRdata" et "ptrRX_RDdata" pointent la même case (sont égaux).
  - Dans le cas contraire :
    - Lire le prochain octet pointé par "ptrRX_RDdata".
    - Incrémenter du contenu du pointeur "ptrRX_RDdata". Si le contenu du pointeur dépasse la taille du buffer, il est affecté par l'adresse de la 1° case.

2. Sources des fonctions

2.1 Déclarations des constantes et variables associées

```
// Buffers RX et TX pour l'UART
#define RX_BufSize 16         // Taille du Buffer_RX
char Buffer_RX[RX_BufSize];   // Buffer circulaire de réception
char *ptrRX_WRdata=Buffer_RX; // Pointeur d'écriture ds Buffer_RX
char *ptrRX_RDdata=Buffer_RX; // Pointeur de lecture ds Buffer_RX
#define TX_BufSize 64         // Taille du Buffer_TX
char Buffer_TX[TX_BufSize];   // Buffer circulaire de transmission
char *ptrTX_WRdata=Buffer_TX; // Pointeur d'écriture ds Buffer_TX
char *ptrTX_RDdata=Buffer_TX; // Pointeur de lecture ds Buffer_TX
```

Les tailles des buffer peuvent être modifiées en cas de besoin. Il faut veiller évidemment à ne pas dépasser la taille de la RAM du microcontroleur.

2.2 Initialisation du coupleur UART

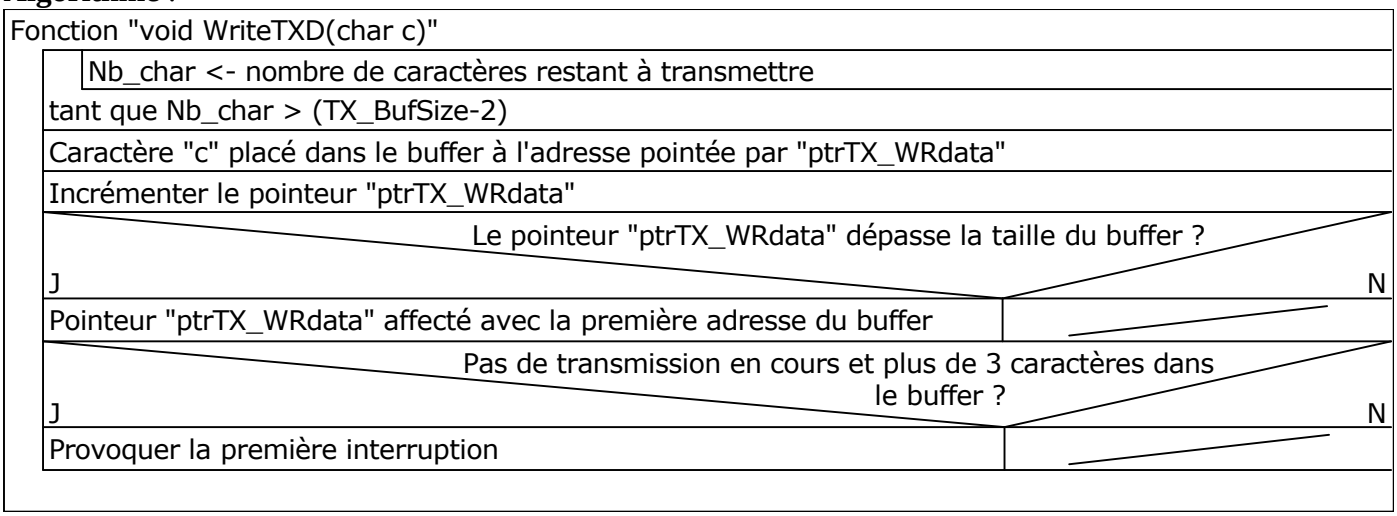
```
/*********************************************************************
  Function:        void InitUART(void)
  Description:     Initialisation de l'UART
                     - 38400 bauds, 8 bits, pas de parité, 1 bit Stop
                     - interruption sur TX si buffer UART vide
                     - interruption sur RX à chaque caractère
*********************************************************************/
void InitUART(void)
{
 U2MODE=0x8000; // UART validé, 8 bits, pas de parité, 1 bit stop
 U2STA =0x8400; // Interruption TX si buffer UART vide
                // UART TX validé
                // Interruption RX à chaque caractère
 U2BRG =(FCY/(16*38400))-1; // 38400 bauds (erreur +0,16% avec FCY=16MHz)
}
```

On peut évidemment choisir un autre format. Celui-ci est adapté au projet VAE.
Note : le dsPIC3011 utilisé dans l'application VAE comporte 2 coupleurs UART. Les fonctions proposées exploitent le coupleur numéro 2, mais les modifications sont simples pour utiliser le numéro 1 (changer le chiffre "2" en "1" !)

2.3 Fonction de transmission d'un octet

```
/*********************************************************************
  Function:        void WriteTXD(char c)
  Description:     Ecriture du caractère "c" dans le buffer d'émission :
                     - attendre libération du buffer
                     - envoi du caractère reçu
                     - pointeur circulaire ptrTX_WRdata incrémenté
*********************************************************************/
void WriteTXD(char c)
{
 int Nb_char;
 // Attendre libération d'une partie du buffer
 do
  {// Calcul du nb de caractères restant à transmettre
   Nb_char=ptrTX_WRdata-ptrTX_RDdata;
   if (Nb_char < 0) Nb_char=Nb_char+TX_BufSize;
  }
 while (Nb_char > (TX_BufSize-2));
 // Caractère placé dans le buffer circulaire
 *ptrTX_WRdata++=c;
 if (ptrTX_WRdata==Buffer_TX+TX_BufSize) ptrTX_WRdata=Buffer_TX;
 // Provoquer la 1° interruption si aucune transmission en cours
 // Le buffer de l'UART fait 4 octets (Nb_char=3)
 if ((U2STAbits.TRMT)&&(Nb_char>=3)) IFS1bits.U2TXIF=1;
}
```

Algorithme :



Note : La fonction comporte une boucle d'attente qui vérifie la disponibilité du buffer. Cette boucle d'attente peut être inadmissible dans certaines applications. Dans ce cas, la seule solution pour l'éliminer est d'augmenter la taille du buffer pour absorber tout le bloc de données à transmettre.

2.4 Fonction d'interruption en transmission

```
/*****************************************************************
  Function    :    void _ISR _U2TXInterrupt (void)
  Description :    Programme d'interruption déclenché quand le buffer
                   de transmission de l'UART (4 octets) est vide
*****************************************************************/
void _ISR _U2TXInterrupt (void)
{
 if (ptrTX_RDdata!=ptrTX_WRdata) // Y a-t-il des caractères à transmettre ?
 {
  do
  {
   U2TXREG=*ptrTX_RDdata++;//Transm. du caractère et incrément. du pointeur
   // Buffer circulaire
   if (ptrTX_RDdata==Buffer_TX+TX_BufSize) ptrTX_RDdata=Buffer_TX;
  }
  while ((ptrTX_RDdata!=ptrTX_WRdata)&&(!U2STAbits.UTXBF));
 }
 IFS1bits.U2TXIF=0; // Acquittement de l'interruption
}
```

Attention : il ne faut pas confondre le buffer "Buffer_TXD" déclaré dans le programme et le buffer du coupleur UART du dsPIC. En effet, l'UART d'un dsPIC comporte son propre tampon circulaire de 4 octets en "hardware". Il permet, pour des applications simples (peu de transmission), de se passer d'un buffer circulaire géré par le logiciel.

Note : on exploite le buffer de l'UART pour limiter le nombre de demandes d'interruption.

2.5 Fonction de réception d'un octet

```
/*********************************************************************
  Function   : int ReadRXD(void)
  Description: Lecture d'un octet dans le buffer de réception
   - renvoi de 0 (false) si aucun caractère reçu
   - renvoi de 1 (true) si caractère reçu: - caractère placé dans "c"
                                           - point. circ. ptrRX_RDdata incrém.
*********************************************************************/
int ReadRXD(char *c)
{
 if (ptrRX_RDdata==ptrRX_WRdata) return(0); // Pas de caractère reçu
 else
 {
  *c=*ptrRX_RDdata++;// Affecter "c" avec le caractère reçu puis incrémenter
                     // le pointeur ptrRX_RDdata
  // Pointeur circulaire
  if (ptrRX_RDdata==Buffer_RX+RX_BufSize) ptrRX_RDdata=Buffer_RX;
  return(1);
 }
}
```

Commentaires :

- Aucun nouveau caractère n'est reçu si les pointeurs d'écriture ("ptrRX_WRdata") et de lecture ("ptrRX_WRdata") sont égaux.
- Le paramètre de la fonction est l'adresse (ou pointeur) du caractère "c" (*c) et non sa valeur pour que la fonction puisse affecter son contenu (c'est à dire la valeur de "c").

2.6 Fonction d'interruption en réception

```
/*********************************************************************
  Function:        void _ISR _U2RXInterrupt (void)
  Description :    Programme d'interruption déclenché à la réception
                   d'un caractère
*********************************************************************/
void _ISR _U2RXInterrupt (void)
{
 IFS1bits.U2RXIF=0;      // Clear interrupt flag
 while (U2STAbits.URXDA) // Lire tous les caractères du buffer de l'UART
 {
  *ptrRX_WRdata++=U2RXREG;// Lecture du caractère suivant et incrémentation
                          // du pointeur circulaire de lecture
  if (ptrRX_WRdata==Buffer_RX+RX_BufSize) ptrRX_WRdata=Buffer_RX;
 }
}
```

Commentaires :

- La boucle itérative "while (U2STAbits.URXDA) {…} " permet de vider le buffer de réception de l'UART (à ne pas confondre avec "Buffer_RXD") au cas, peu probable, où plusieurs caractères seraient en attente.

3. Programmes de test

3.1 Transmission

Il s'agit de valider les fonctions "WriteTXD" et "_U2TXInterrupt".
Pour montrer l'intérêt d'utiliser les interruptions, on écrit une fonction chargée de transmettre un "paquet" de données, en l'occurrence, une chaîne de caractères. Cela simplifie le contrôle sur un PC avec le logiciel "hyperterminal".

3.1.1 Fonction "WriteTXD_String" et "WriteLnTXD_String"

```
/*****************************************************************
  Function:        void WriteTXD_String(char *String)
  Description:     Ecriture d'une chaîne de caractères dans le buffer TX
*****************************************************************/
void WriteTXD_String(char *String)
{
 while (*String) // Répéter jusqu'à caractère final (0)
 {
  WriteTXD(*String++); // Transmission du caractère suivant et inc. pointeur
 }
}
```

Fonctionnement :

- La fonction reçoit l'adresse d'une chaîne de caractère comme paramètre.
- Chaque caractère de cette chaîne est transmis en le donnant comme paramètre à la fonction WriteTXD
- En "C" le caractère terminal d'une chaîne est "NULL" (valeur 0) : cette caractéristique est utilisée pour terminer la boucle itérative "while". Le caractère "NULL" n'est pas transmis.

```
/*****************************************************************
  Function:        void WriteLnTXD_String(char *String)
  Description:     Ecriture d'une chaîne de caractères dans le buffer TX
                   "Retour chariot" (CR+LF) ajouté à la fin
*****************************************************************/
#define CR 13 // Retour chariot
#define LF 10 // Saut de ligne
void WriteLnTXD_String(char *String)
{
 TEST1=1;
 WriteTXD_String(String);   // Transmission de la chaîne
 WriteTXD(CR); WriteTXD(LF);// Transmission du "retour chariot"
 TEST1=0;
}
```

Fonctionnement :

- La fonction fait appel à "WriteTXD_String" pour transmettre la chaîne
- La transmission de la chaîne est suivie des caractères "CR" et "LF" pour opérer un retour de ligne sur le terminal du PC.
- Les instructions "TEST1=1" et "TEST1=0" affectent à "1" et respectivement à "0" le port RF1 (broche 29 du dsPIC3011). Cela permet d'observer à l'oscilloscope la durée d'exécution de cette fonction.

3.1.2 Programme de test

```
                        /*****************************
                        *    Programme principal     *
                        *****************************/

/*********************************************************************
  Function:        void main(void)
  Description:     Appelée au reset
*********************************************************************/
int main(void)
{
 U2MODE=0;    // Inhiber l'UART à cause du bootloader
 TEST0_Dir=0; // Signal TEST0 en sortie
 TEST1_Dir=0; // Signal TEST1 en sortie
 // Initialisations
 InitUART();  // 38400 bauds, 8 bits, pas de parité, 1 bit stop
 // Raz indicateurs interruptions
 IFS1bits.U2TXIF=0; // Clear interrupt flag TX
 // Validations interruptions UART en transmission
 IEC1bits.U2TXIE=1; // Validation interruption TX

 WriteLnTXD_String("Test de dialogue controleur VAE-PC.");// 37 caractères
 while (1) {}
}
```

3.1.3 Fonction d'interruption en transmission "_U2TXInterrupt" de test

La fonction décrite plus haut est légèrement modifiée en insérant 2 signaux de test pour observer l'activation de la fonction à l'oscilloscope.

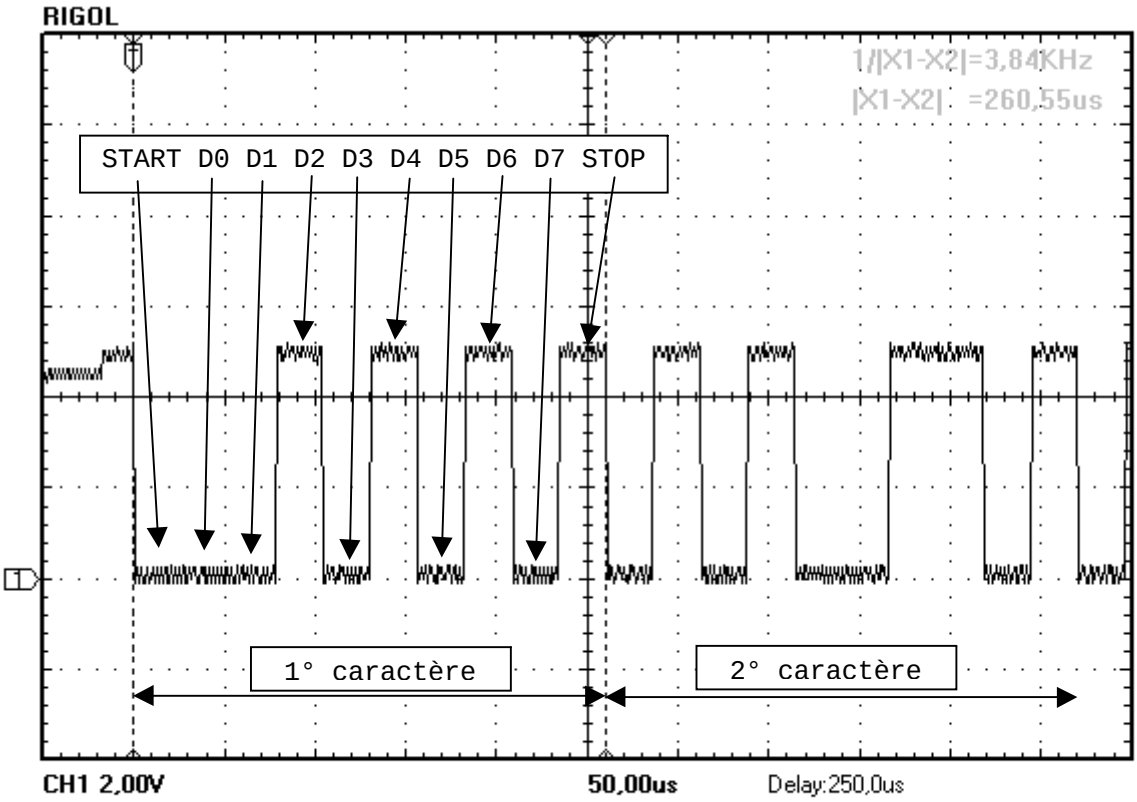
```
/*********************************************************************
  Function:        void _ISR _U2TXInterrupt (void)
  Description :    Programme d'interruption déclenché quand le buffer
                   de transmission de l'UART (4 octets) est vide
*********************************************************************/
void _ISR _U2TXInterrupt (void)
{
 TEST0=1;
 if (ptrTX_RDdata!=ptrTX_WRdata) // Y a-t-il des caractères à transmettre ?
 {
  do
  {
   TEST2=1;
   U2TXREG=*ptrTX_RDdata++;//Transm. du caractère et incrém. du pointeur
   // Buffer circulaire
   if (ptrTX_RDdata==Buffer_TX+TX_BufSize) ptrTX_RDdata=Buffer_TX;
   TEST2=0;
  }
  while ((ptrTX_RDdata!=ptrTX_WRdata)&&(!U2STAbits.UTXBF));
 }
 IFS1bits.U2TXIF=0; // Acquitement de l'interruption
 TEST0=0;
}
```

- TEST0 : mis à "1" dès le début de la fonction et remis à "0" à sa fin. Ce signal de test indique donc que la fonction est en cours d'exécution.
- TEST2 : impulsion de courte durée à chaque écriture dans le registre U2TXREG

3.2 Relevés

3.2.1 Vérification du format

On relève le premier caractère transmis juste après un reset.



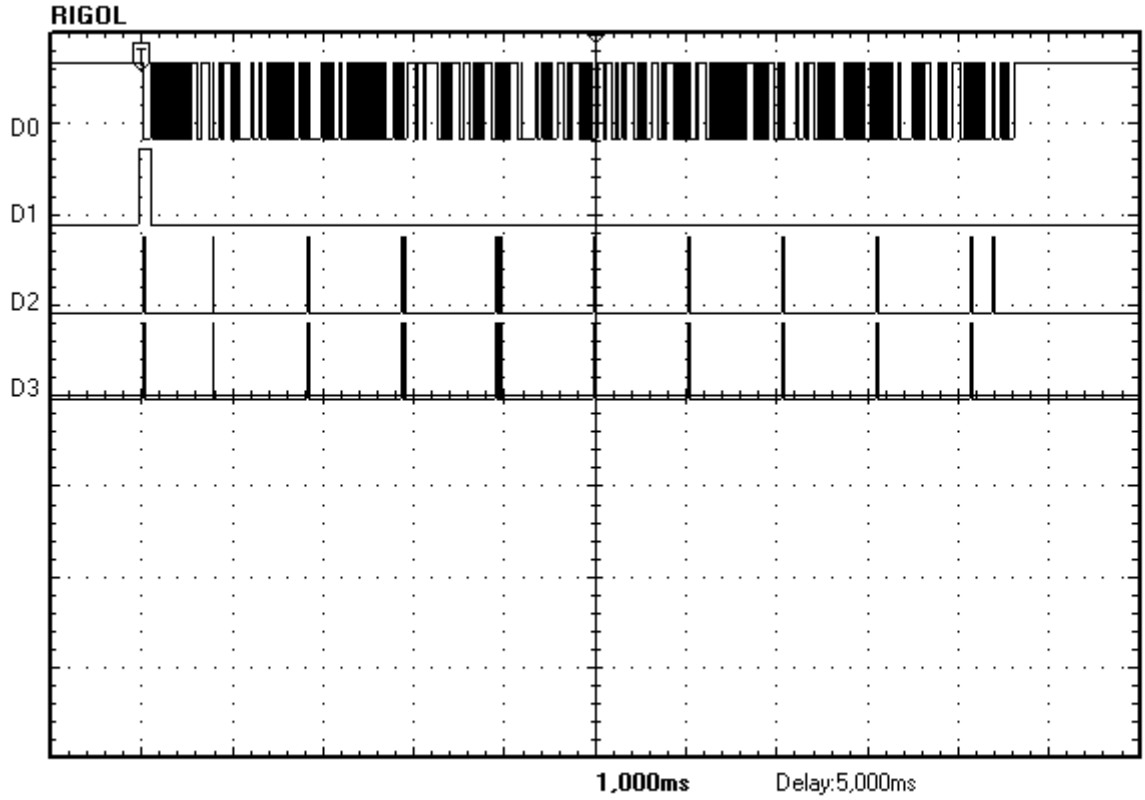
TXD

Pour identifier la trame série du premier caractère, on règle l'intervalle entre les curseurs de l'oscilloscope à la valeur nominale de la durée de la trame : 10/38400 = 260µS. Le curseur de gauche est alors placé au début du bit START qui est facile à identifier.
On peut alors situer chacun des bits de la trame et déterminer la valeur de l'octet transmis.
On vérifie ainsi :

- la vitesse de transmission (38400 bauds)
- l'octet transmis = 01010100 en binaire, soit 0x54 qui est bien le code Ascii de la lettre "T" (première lettre de la chaîne de caractères).

On constate aussi qu'aucun temps mort supplémentaire ne sépare 2 caractères consécutifs.

3.2.2 Trame complète avec TX_BufSize = 64

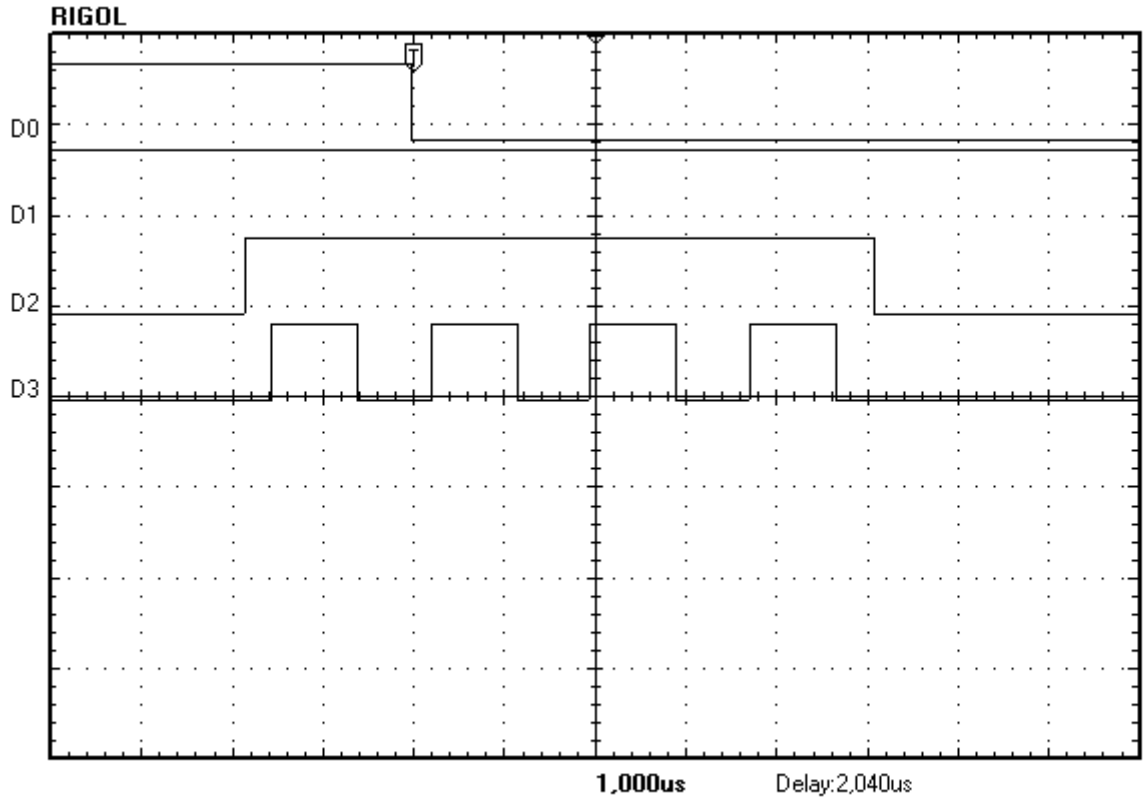


TEST1 : phase d'exécution de la fonction "WriteTXD_String"
TEST0 : phases d'exécution de la fonction d'interruption "U2TXInterrupt"
TEST2 : impulsions indiquant l'écriture dans le registre U2TXREG

Commentaires :

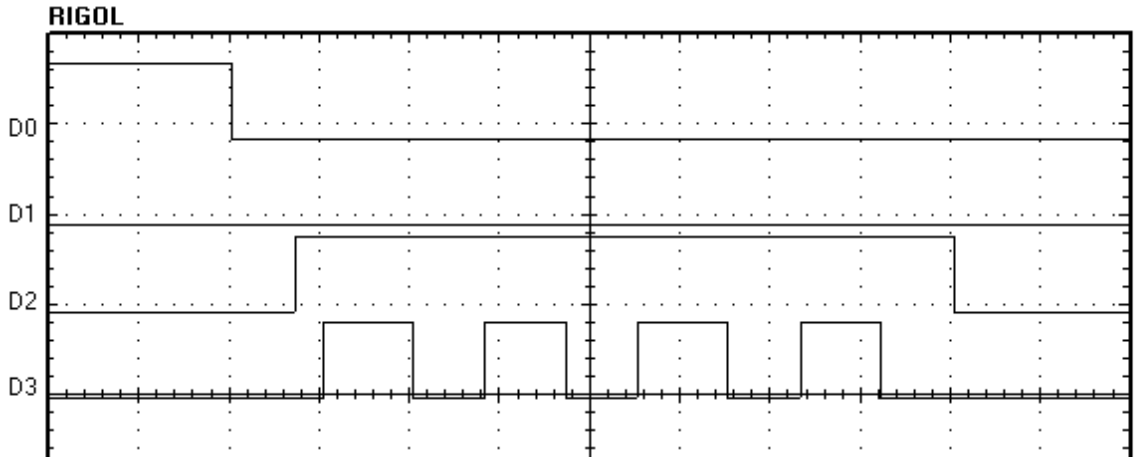
- La chaîne de caractères à transmettre comporte 37 caractères (voir §3.1.2). Le buffer "Buffer_TX" (64 octets) peut donc tous les absorber sans attendre leur transmission. On constate donc que la durée d'exécution de la fonction "WriteTXD_String" **est très courte : soit 61µS** (en "zoomant" sur TEST1) car elle n'attend jamais la libération du buffer.

– Première interruption TXD (zoom sur la 1° impulsion de TEST0) :



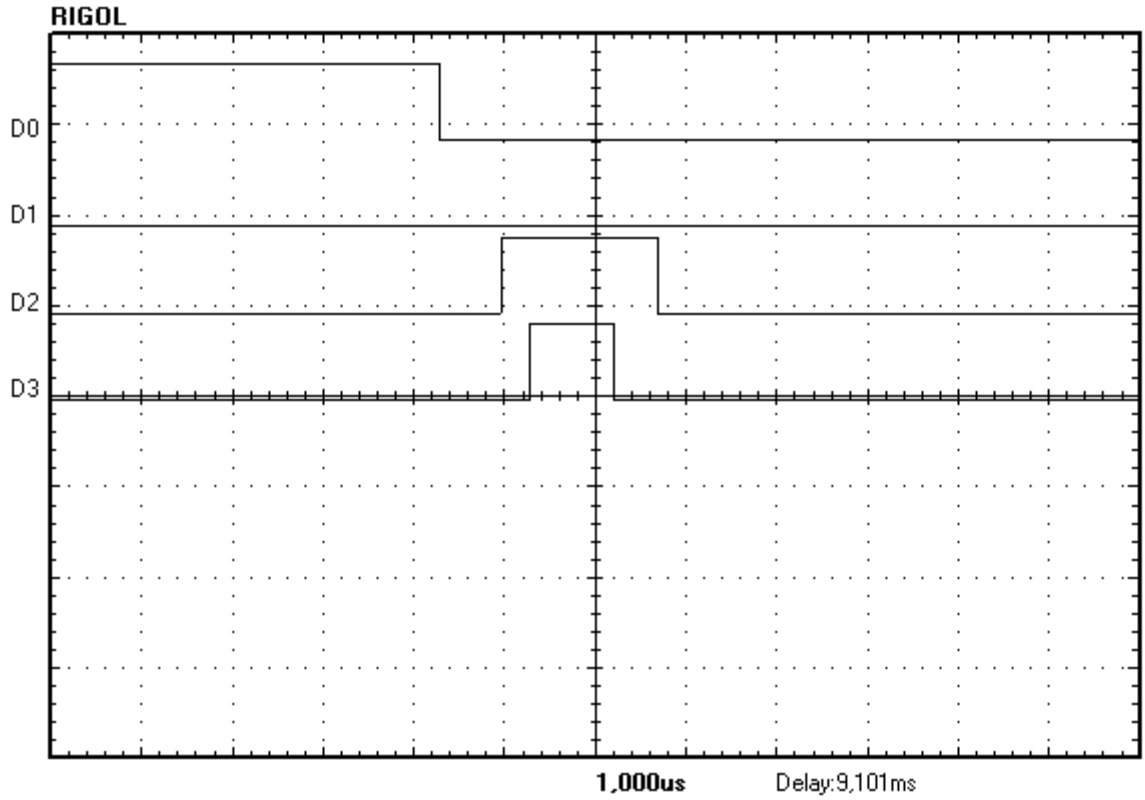
  – La fonction "WriteTXD" provoque la première demande d'interruption dès que le buffer "Buffer_TX" contient 4 caractères (voir §2.3). Son activation est visualisée par le signal TEST0. Durée d'exécution = 7µS
  – La fonction d'interruption "_U2TXInterrupt" transfère les caractères du buffer "Buffer_TX" dans le registre FIFO de l'UART du dsPIC jusqu'à ce que "Buffer_TX" soit vide ou le buffer FIFO plein (4 octets). Dans le cas représenté, le "Buffer_TX" comporte 4 caractères et le registre FIFO est vide à l'appel de la fonction. Elle transfère donc ces 4 octets sans attendre la fin de leur transmission effective. Ce transfert est visualisé par les 4 impulsions de TEST2.

– Deuxième interruption TXD (zoom sur la 2° impulsion de TEST0) :



On observe le même comportement car à ce moment, tous les caractères de la chaîne à transmettre se trouvent dans le tampon "Buffer_TX" ("WriteTXD_String" a fini son traitement car TEST1 est à "0") et il reste encore 33 caractères à transmettre (37 – 4 déjà transmis).
La fonction "_U2TXInterrupt" se termine ici quand le registre FIFO de l'UART est plein.

– Avant dernière interruption TXD (zoom sur l'avant-dernière impulsion de TEST0) :



TXD

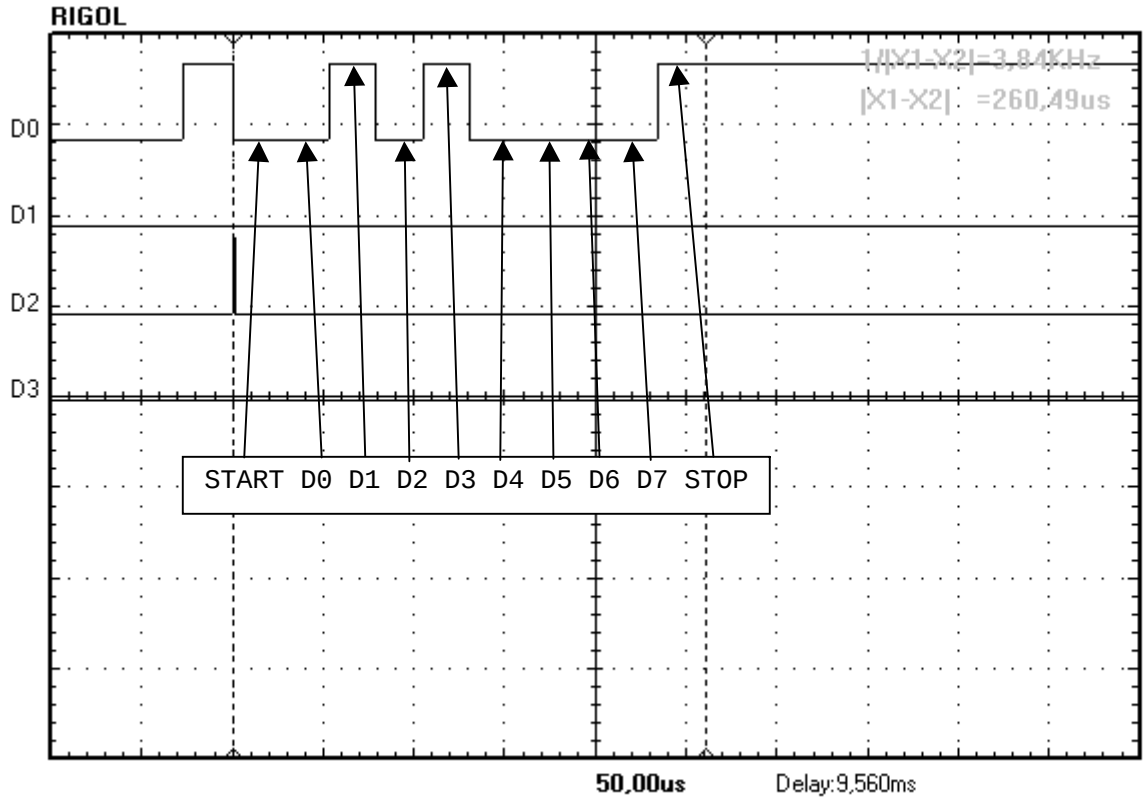
TEST1

TEST0

TEST2

– Il s'agit ici de la 10° activation de la fonction d'interruption "_U2TXInterrupt". A chacune des 9 précédentes, 4 caractères ont été transmis au registre FIFO de l'UART, soit 9x4=36 caractères. Il reste donc encore 1 caractère en attente dans le tampon "Buffer_TX".
– Ceci est confirmé car on observe une seule impulsion sur TEST2 qui correspond à l'écriture de ce dernier caractère dans le registre U2TXREG.
– La fonction "_U2TXInterrupt" se termine ici quand le tampon " Buffer_TX " est vide.

– Dernière interruption TXD (zoom sur la dernière impulsion de TEST0 et la fin de la trame) :



TXD

TEST1

TEST0

TEST2

  – On observe l'absence d'impulsion sur TEST2, ce qui signifie que le registre "U2TXREG" n'est pas affecté. Ceci est normal car tous les caractères ont déjà été transférés (les pointeurs "ptrTX_RDdata" et "ptrTX_WRdata" sont égaux).
  – Ce chronogramme permet aussi de lire la valeur du dernier octet : 00001010=0x0A, soit le caractère LF. C'est conforme.

Conclusions :

– Les fonctions "WriteTXD" et "_U2TXInterrupt" sont validées.
– Occupation CPU :
  – la durée de la trame complète est de **9,63mS** (37 caractères à 38400 bauds sans temps mort)
  – temps CPU = 61µS ("WriteLnTXD_String") + 9x7µS ("_U2TXInterrupt" avec 4 écritures) + 1x1µS ("_U2TXInterrupt" avec 1 écriture) + 400nS (dernier appel de "_U2TXInterrupt") = **125,4µS**
  – la transmission de la trame n'occupe donc que **1,3% du temps CPU** pendant sa transmission.

3.3 Réception

Il s'agit de valider les fonctions "ReadRXD" et "_U2RXInterrupt" et de montrer l'intérêt d'utiliser les interruptions.
Dans l'application "VAE", l'essentiel des octets reçus sont des commandes en provenance du logiciel de pilotage sur PC. La fonction "Detect_Cmd_RXD" de décodage de ces commandes sera également validée dans ce paragraphe.

3.3.1 Codage des commandes

Les trames des commandes sont toujours constituées de 5 octets :

Synchro	Code	D_H	D_L	Synchro

Les 2 octets "Synchro" (valeur "Z" pour le contrôleur du VAE) encadrent la commande effective : la fonction "Detect_Cmd_RXD" peut alors facilement identifier le début et la fin de chaque trame.
Pour réduire au minimum le risque d'une mauvaise identification d'une trame, on évitera d'utiliser un code de commande égal au caractère "Synchro".
L'octet "Code" est le code de la commande. Ses paramètres sont placés dans les 2 octets suivants "D_H" et "D_L".

Liste des commandes (non exhaustive) :

Code	Signification
"B"	Avance ou retard de phase = (D_H*256+D_L)*28,6µ°
"C"	Type de consigne : si D_H = "P" → poignée ou pédales si D_H = "T" → fournie par la commande "T"
"T"	Consigne "tension" du PC = (D_H*256+D_L)
"P"	Liaison avec le PC : si D_H = O → le contrôleur fonctionne normalement si $D_H \neq$ O → le contrôleur est piloté par le PC

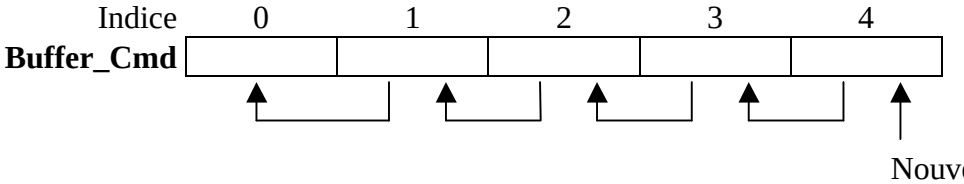
3.3.2 Fonction "Detect_Cmd_RXD"

```
/*****************************************************************
  Function    : void Detect_Cmd_RXD(void)
  Description : Détection et exécution d'une éventuelle commande reçue
                Les commandes ont toujours une longueur de 5 octets :
                Synchro, code, Dh, Dl, Synchro
                   - code : identifie la commande
                   - Dh, Dl : paramètre de la commande
*****************************************************************/
void Detect_Cmd_RXD(void)
{
 int i;
 char c;
 TEST1=1; // Pour identifier l'activation de la fonction à l'oscilloscope
 if (!ReadRXD(&c)) return;
 for (i=1; i<5; i++) Buffer_Cmd[i-1]=Buffer_Cmd[i];
 Buffer_Cmd[4]=c;
 if ((Buffer_Cmd[0]==Synchro) && (Buffer_Cmd[4]==Synchro))
 {
  switch(Buffer_Cmd[1]) // Code de la commande
  {
   case 'B' : // Avance/retard de phase (à multiplier par 16)
   {
    WriteTXD_Param("Avance ou retard de phase :",Buffer_Cmd[2],Buffer_Cmd[3]);
    break;
   }
   case 'C' : // Type de consigne
   {
    switch(Buffer_Cmd[2])
    {
     case 'P' : // Poignée
     {
      WriteLnTXD_String("Consigne poignée de gaz");
```

```
      break;
     }
     case 'T' :
     {
      WriteLnTXD_String("Consigne PC");
      break;
     }
     default : break;
    }
    break;
   }
   case 'T' : // Consigne "Tension" du PC
   {
    WriteTXD_Param("Consigne tension du PC :",Buffer_Cmd[2],Buffer_Cmd[3]);
    break;
   }
   case 'P' : // Liaison avec PC
   {
    if (Buffer_Cmd[2]=='O')
    {
     Flags.PC_Connected=1;
     WriteLnTXD_String("Liaison avec PC établie");
    }
    else
     {
      Flags.PC_Connected=0;
      WriteLnTXD_String("Liaison avec PC coupee");
     }
    break;
   }
   default : break;
  }
 }
 TEST1=0; // Pour identifier l'arrêt de la fonction à l'oscilloscope
}
```

Commentaires :

- Aucun traitement n'est réalisé (retour immédiat) si la fonction "ReadRXD" renvoie la valeur 0 (aucun caractère dans le buffer "Buffer_RX").
- Chaque nouveau caractère reçu est placé dans un petit buffer de 5 octets ("Buffer_Cmd") utilisé pour identifier la commande suivant le principe suivant :



  - La fonction commence par décaler d'un cran à gauche le contenu des 4 dernières cases (taille = 1 octet) de "Buffer_Cmd". Le caractère de la case "0" est perdu.
  - Le dernier caractère reçu est alors rangé dans la case "4". Ainsi, les 5 derniers caractères reçus se trouvent dans "Buffer_Cmd" à la réception de chaque caractère.
  - La fonction "Detect_Cmd_RXD" identifie une trame de commande si les cases "0" et "4" contiennent la valeur "Synchro".
  - Dans ce cas :
    - le code de la commande est dans la case "1" (Buffer_Cmd[1]),
    - le paramètre D_H est dans la case "2" (Buffer_Cmd[2]),
    - le paramètre D_L est dans la case "3" (Buffer_Cmd[3]).
- L'instruction "switch(Buffer_Cmd[1])" permet de rendre le décodage de chaque commande très lisible.

3.3.3 Fonction "WriteTXD_Param"

Cette fonction est utilisée pour transmettre une réponse via l'UART et vérifier ainsi sur le PC la bonne interprétation de chaque commande.

```
/*****************************************************************
  Function   : void WriteTXD_Param(char *Phrase, unsigned char P1, unsigned char P2)
  Description: transmission via l'UART :
                - de la chaîne de caractère "Phrase"
                - des 2 paramètres "P1" et "P2" convertis en Ascii
*****************************************************************/
void WriteTXD_Param(char *Phrase, unsigned char P1, unsigned char P2)
{
 WriteTXD_String(Phrase);     // Transmission de la phrase sans RC
 pString=String_4;            // On utilise la chaîne de caractères "String_4"
 Conv_Bin8_String(pString,P1);// pour convertir le 1° paramètre en Ascii
 WriteTXD_String(pString);    // Transmission du 1° paramètre (octet) en Ascii
 WriteTXD(' ');               // Transmission d'un espace séparateur
 Conv_Bin8_String(pString,P2);// Conversion Ascii du 2° paramètre ds "String_4"
 WriteLnTXD_String(pString);  // Transmission du 2° paramètre (octet) en Ascii + RC
}
```

Commentaires :

- La chaîne de caractères "Phrase" est transmise à la fonction par *adresse* pour gagner du temps et limiter la taille des variables locales
- La fonction utilise les fonctions "WriteTXD_String", "WriteLnTXD_String" (voir §3.1.1) et "Conv_Bin8_String" décrite ci-dessous.

3.3.4 Fonction "Conv_Bin8_String"

```
/*****************************************************************
  Nom             :  Conv_Bin8_String
  Description     :  Conversion d'une valeur codée 8 bits en binaire
                     -> chaîne de caractères
  Arguments       :  x : nombre binaire à convertir
                     *String : pointeur vers la chaîne de destination
  Valeur renvoyée : aucune
*****************************************************************/
void Conv_Bin8_String(char *String, unsigned char x)
{
 String[0]=x/100;                              // Centaines
 String[1]=(x-(100*String[0]))/10;             // Dizaines
 String[2]=x-(100*String[0])-10*String[1]+'0'; // Unités en Ascii
 String[0]+='0'; // Conversion BCD -> Ascii des centaines
 String[1]+='0'; // Conversion BCD -> Ascii des dizaines
 String[3]=0;    // Terminateur de chaîne
}
```

Commentaires :

- La fonction utilise l'opérateur division, ce qui n'est pas optimal pour la vitesse. Mais il ne s'agit qu'une fonction de test qu'il est inutile d'optimiser.
- Comme pour les autres fonctions, la chaîne de caractères "String" est transmise par *adresse* pour gagner du temps et limiter la taille des variables locales

3.3.5 Programme de test

```
                        /*****************************
                        *    Programme principal     *
                        *****************************/

/*******************************************************************
  Function:        void main(void)
  Description:     Appelée au reset
*******************************************************************/
int main(void)
{
 U2MODE=0;    //Inhiber l'UART à cause du bootloader
 TEST0_Dir=0; // Signal TEST0 en sortie
 TEST1_Dir=0; // Signal TEST1 en sortie
 TEST2_Dir=0; // Signal TEST2 en sortie
 // Initialisations
 InitUART();  // 38400 bauds, 8 bits, pas de parité, 1 bit stop
 // Raz indicateurs interruptions
 IFS1bits.U2TXIF=0; // Clear interrupt flag TX
 IFS1bits.U2RXIF=0; // Clear interrupt flag RX
 // Validations interruptions UART
 IEC1bits.U2TXIE=1; // Validation interruption TX
 IEC1bits.U2RXIE=1; // Validation interruption RX
 while (1)
 {
  Detect_Cmd_RXD(); //Détection et exécution éventuelle d'une commande sur RXD
 }
}
```

Commentaires :

- La fonction "main" réalise d'abord toutes les initialisations nécessaires
- Le dsPIC exécute alors une boucle sans fin qui appelle à chaque fois la fonction "Detect_Cmd_RXD" de détection des éventuelles commandes reçues.

3.3.6 Fonction d'interruption en réception " _U2RXInterrupt" de test

La fonction décrite plus haut est légèrement modifiée en insérant 2 signaux de tests pour observer le déroulement de la fonction à l'oscilloscope.

```
/*******************************************************************
  Function:        void _ISR _U2RXInterrupt (void)
  Description :    Programme d'interruption déclenché à la réception
                   d'un caractère
*******************************************************************/
void _ISR _U2RXInterrupt (void)
{
 char c;
 TEST0=1;
 IFS1bits.U2RXIF=0;      // Clear interrupt flag
 while (U2STAbits.URXDA) // Boucle pour lire ts les car. ds le buf. de l'UART
 {
  c=U2RXREG;
  if (c==Synchro) TEST2=1;
  *ptrRX_WRdata++=c; // Lecture du caractère suivant et incrémentation
                     // du pointeur circulaire de lecture
  if (ptrRX_WRdata==Buffer_RX+RX_BufSize) ptrRX_WRdata=Buffer_RX;
  TEST2=0;
 }
 TEST0=0;
}
```

Commentaires :

- La variable locale "c" évite de lire 2 fois le registre "U2RXREG". En effet, le registre FIFO de l'UART est décalé d'un cran à chaque lecture; ainsi 2 lectures consécutives ne donnent pas le même résultat et peuvent même provoquer la perte d'un caractère.
- TEST0 indique l'activité de la fonction "_U2RXInterrupt"
- Une impulsion sur TEST2 indique la reconnaissance du caractère "Synchro" (ici "Z").

3.4 Relevés

Les tests sont effectués avec l'utilitaire "hyper-terminal" de Windows configuré au format de l'application (38400 8-N-1).

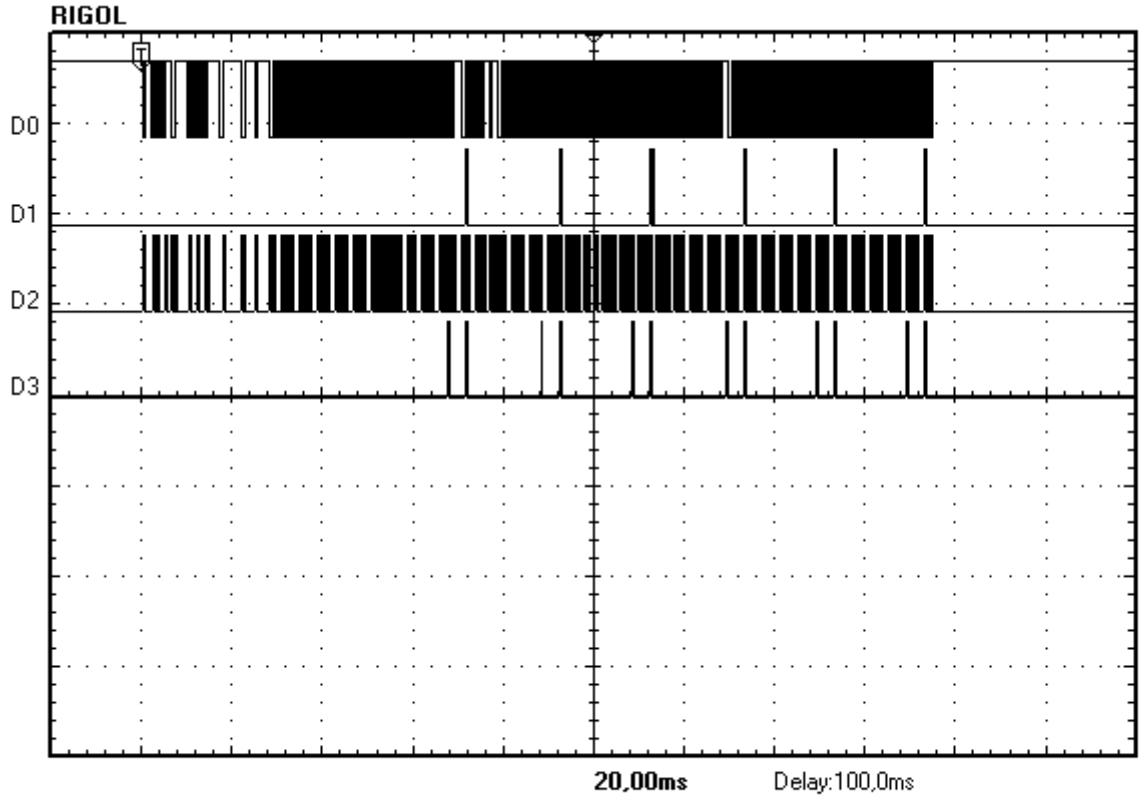
3.4.1 Fichier "texte" de commandes

Pour être plus réaliste et éviter de taper les commandes à la main, on édite un fichier texte et on le transfère sur le port COM utilisé avec une commande de l'hyper-terminal :

```
Fichier de test pour decodage commandes
Commande 1 : ZB12Z
Commande 2 : ZCPXZ
Commande 3 : ZCTYZ
Commande 4 : ZT89Z
Commande 5 : ZPOWZ
Commande 6 : ZPNWZ
```

La fonction "Detect_Cmd_RXD" devra reconnaître les séquences "ZxxxZ" ("Z" étant le caractère "Synchro") parmi tous les caractères reçus et réagir en conséquence.

3.4.2 Trame RXD du fichier "texte" complet



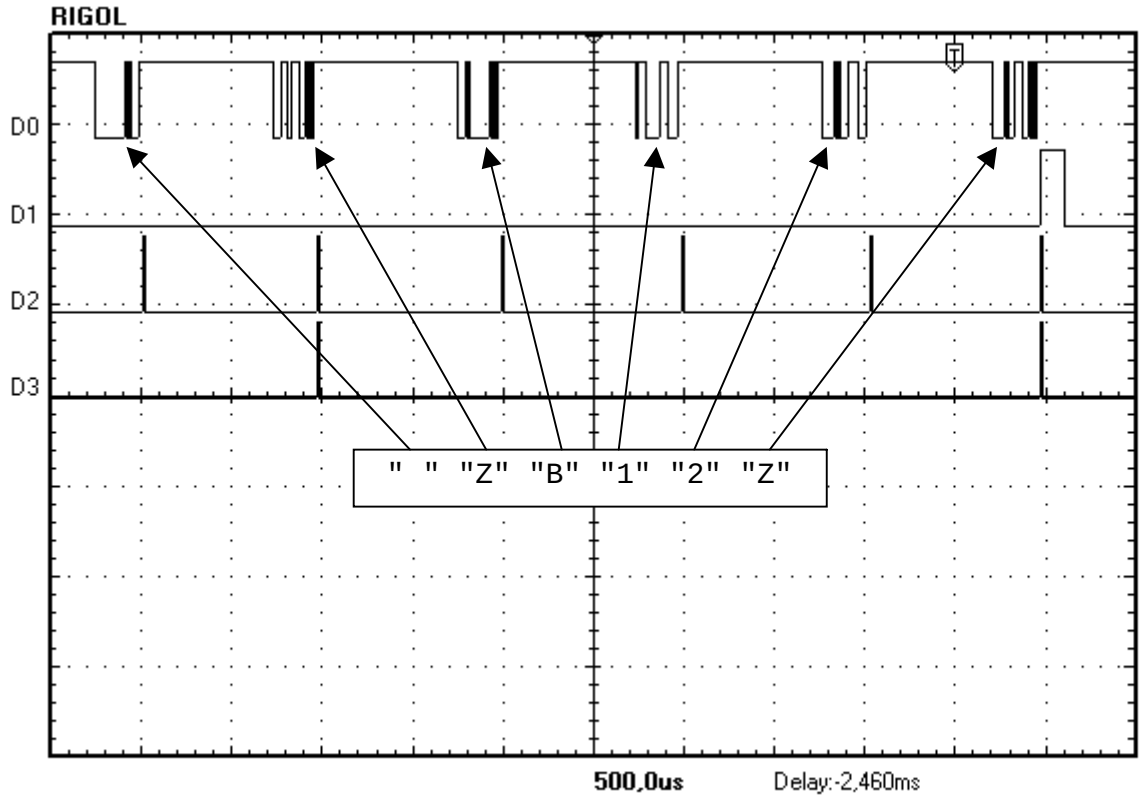
- **Rappels** :
  - TEST0 : à "1" pendant l'exécution de "_U2RXInterrupt"
  - TEST2 : impulsion à "1" à la reconnaissance des caractères "Z" dans "_U2RXInterrupt"
  - TEST1 : à "1" à la détection d'une commande (5 caractères dont le 1° et le dernier sont "Z") pendant l'exécution de "Detect_Cmd_RXD".

- **Commentaires** :
  - Le fichier transféré comporte 6 commandes. On constate bien les 6 paires d'impulsions sur TEST2 et les 6 impulsions sur TEST0 qui montrent que la fonction "Detect_Cmd_RXD" a bien identifié les 6 commandes.
  - Réponses sur l'hyper-terminal :

```
Avance ou retard de phase :049 050
Consigne poignée de gaz
Consigne PC
Consigne tension du PC :056 057
Liaison avec PC etablie
Liaison avec PC coupee
```

  On constate que les 6 commandes sont bien reconnues et interprétées.

3.4.3 Première commande : "ZB12Z"



RXD

TEST1

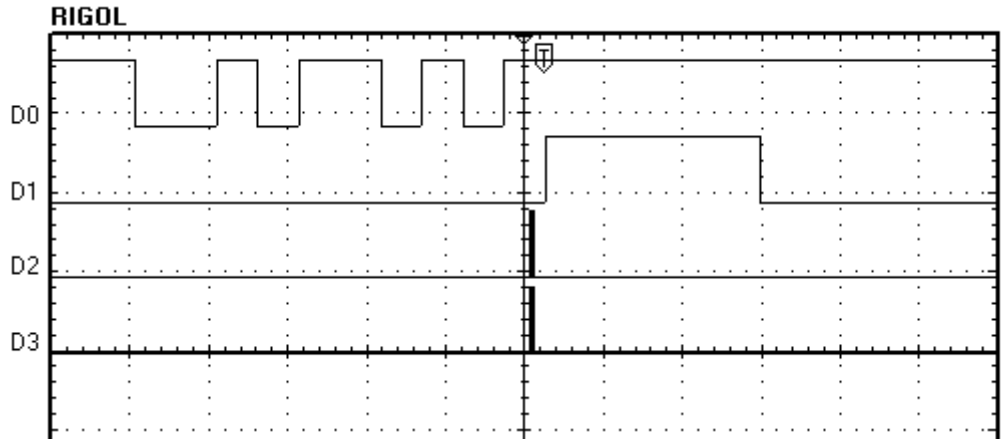
TEST0

TEST2

Commentaires :

- Les trames transmises par le PC ne sont pas jointives. Ce n'est pas gênant, mais la transmission est ralentie.
- L'espace (" ") qui précède la commande "ZB12Z" est le dernier caractère de la phrase "`Commande 1 : `" (voir contenu du fichier texte de commandes au §3.4.1)
- Les impulsions sur TEST0 montrent que la fonction d'interruption "_U2RXInterrupt" est bien activée à la fin de chaque caractère ce qui confirme sa réception.
- Les impulsions sur TEST2 suivent la réception des caractères "Z" : CQFD.
- L'impulsion sur TEST1 à la réception du 2° caractère "Z" confirme la détection de la commande par la fonction "Detect_Cmd_RXD"

3.4.4 Zooms sur le 2° "Z"



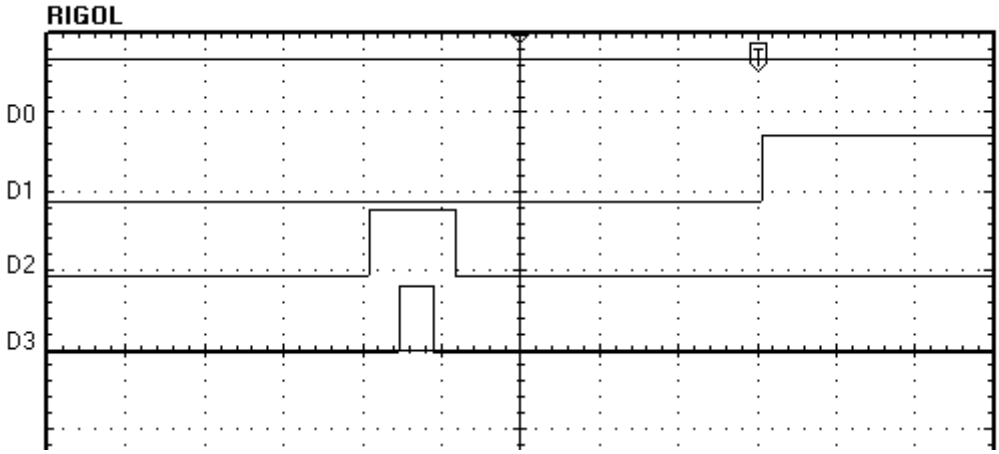
RXD

TEST1

TEST0

TEST2

Échelle = 50µS/div



RXD

TEST1

TEST0

TEST2

Échelle = 2µS/div

Commentaires :

- La fonction d'interruption est activée 16µS après la transmission du bit D7, soit avant la fin du bit "Stop" (26µS plus tard) ! L'interruption "RX" est donc provoquée dès l'échantillonnage du bit "Stop", soit 13µS après la fin de D7.
- L'impulsion TEST2 est encadrée par TEST0 car elle est produite par la fonction "_U2RXInterrupt"
- Occupation CPU pour la réception : 2µS par caractère, soit 2µS/(10x26µS) = **0,77%** du temps de transmission.

3.4.5 Conclusion

Dans ce programme simple, seules 2 sources d'interruptions sont utilisées. Dans l'application VAE, elles seront beaucoup plus nombreuses et quelques unes plus prioritaires.
Le délai d'activation de "_U2RXInterrupt" pourra donc être fortement rallongé. Le buffer RX de l'UART (4 caractères) est mis à contribution pour limiter le risque de perte de caractères : délai max = 4x10x26µS=1mS environ.