

# **LE SYSTEME DOMOTIQUE DOMOCAN**

**PAR BIGONOFF**



**INTERFACE CAN/RS232**

**Révision beta1**



<b>1.CARACTÉRISTIQUES .....</b>	<b>5</b>
1.1 PRÉSENTATION GÉNÉRALE .....	5
1.2 <i>Caractéristiques techniques</i> .....	5
<b>2. RÉALISATION PRATIQUE.....</b>	<b>6</b>
2.1 LE SCHÉMA .....	6
2.2 LE TYPON ET L'IMPLANTATION DES COMPOSANTS .....	7
2.3 RÉALISATION PRATIQUE.....	9
2.4 INSTALLATION .....	10
<b>3. COMMUNICATIONS.....</b>	<b>13</b>
3.1 LES TRAMES CÔTÉ DOMOCAN .....	13
3.2 TRAMES RS232.....	13
3.2. STRUCTURE D'UNE TRAME RS232 .....	13
3.3 COMMANDES RS232 SUPPORTÉES .....	15
3.3.1 <i>La commande 0x50</i> .....	15
3.3.2 <i>La commande 0x51</i> .....	15
3.3.3 <i>La commande 0x52</i> .....	15
3.3.4 LA COMMANDE 0x54 .....	16
3.3.5 <i>La commande 0x60</i> .....	17
3.3.6 <i>La commande 0x70</i> .....	18
<b>4. ANALYSE DU LOGICIEL PIC .....</b>	<b>21</b>
4.1 LE FICHIER CAN-RS232.ASM .....	21
<b>5. MISE EN SERVICE .....</b>	<b>55</b>
5.1 CONNEXION .....	55
5.2 PREMIER LANCEMENT DE DOMOGEST .....	55
5.3 VÉRIFICATION DE L'INTERFACE .....	57
5.4 RAPPELS SUR LES FILTRES ET LES MASQUES.....	60
<b>6. UTILISATION DU PRÉSENT DOCUMENT .....</b>	<b>65</b>



# 1.Caractéristiques

## 1.1 Présentation générale

La carte d'interface CAN/RS232 n'est pas à proprement parler une carte du réseau DOMOCAN. En effet, elle est transparente pour le système, n'est pas reconnue en tant que telle, et n'utilise pas les fichiers de base (domodef.inc et domoboot.inc) utilisés pour les autres cartes.

Sa structure n'est pas non plus identique, puisque c'est la seule carte qui ne reçoit pas ses commandes à partir du bus CAN, mais à partir du port RS232 du PC sur lequel elle est connectée.

Cette carte se « limite » donc à permettre l'échange de données entre un périphérique RS232 et le bus CAN de notre système domotique.

Bien évidemment, ces échanges s'effectuent par des trames structurées de façon conventionnelle, je vais en reparler.

La carte RS232 travaille à un débit maximum possible inférieur au débit CAN autorisé. On pourrait donc se dire qu'on risque, en cas de trafic maximum sur le bus, de saturer l'interface dans le sens CAN vers RS232. C'est effectivement vrai en théorie, beaucoup moins en pratique. De plus, les masques et filtres sont entièrement paramétrables sur cette carte, on peut donc ne laisser filtrer que les commandes qui nous intéressent, ce qui permet de diminuer en conséquence le débit des informations.

Dans l'autre sens, le problème ne se pose pas, les informations étant limitées à la vitesse maximale du port RS232 par définition.

## 1.2 Caractéristiques techniques

<b>Débit du bus CAN</b>	: 500 Kbits/s
<b>Débit du port RS232</b>	: 115200 bauds
<b>Caractéristiques RS232</b>	: 1 start-bit, 8 bits de data, 1 stop-bit, pas de parité
<b>Contrôle du flux RS232</b>	: Software
<b>Trames CAN acceptées</b>	: étendues de type data, conformes à la norme CAN 2.0b
<b>Buffer d'entrée CAN</b>	: 256 octets
<b>Filtres et masques</b>	: 6 filtres et 2 masques paramétrables
<b>Type de carte</b>	: à microcontrôleur PIC 18F248 à 40Mhz
<b>Driver physique CAN</b>	: MCP2551 ou compatible
<b>Driver physique RS232</b>	: Max232 ou compatible
<b>Alimentation</b>	: +12V, prise sur le bus DOMOCAN
<b>Signalisations</b>	: 1 Led trame RS232 entrante, 1 LED trame RS232 sortante

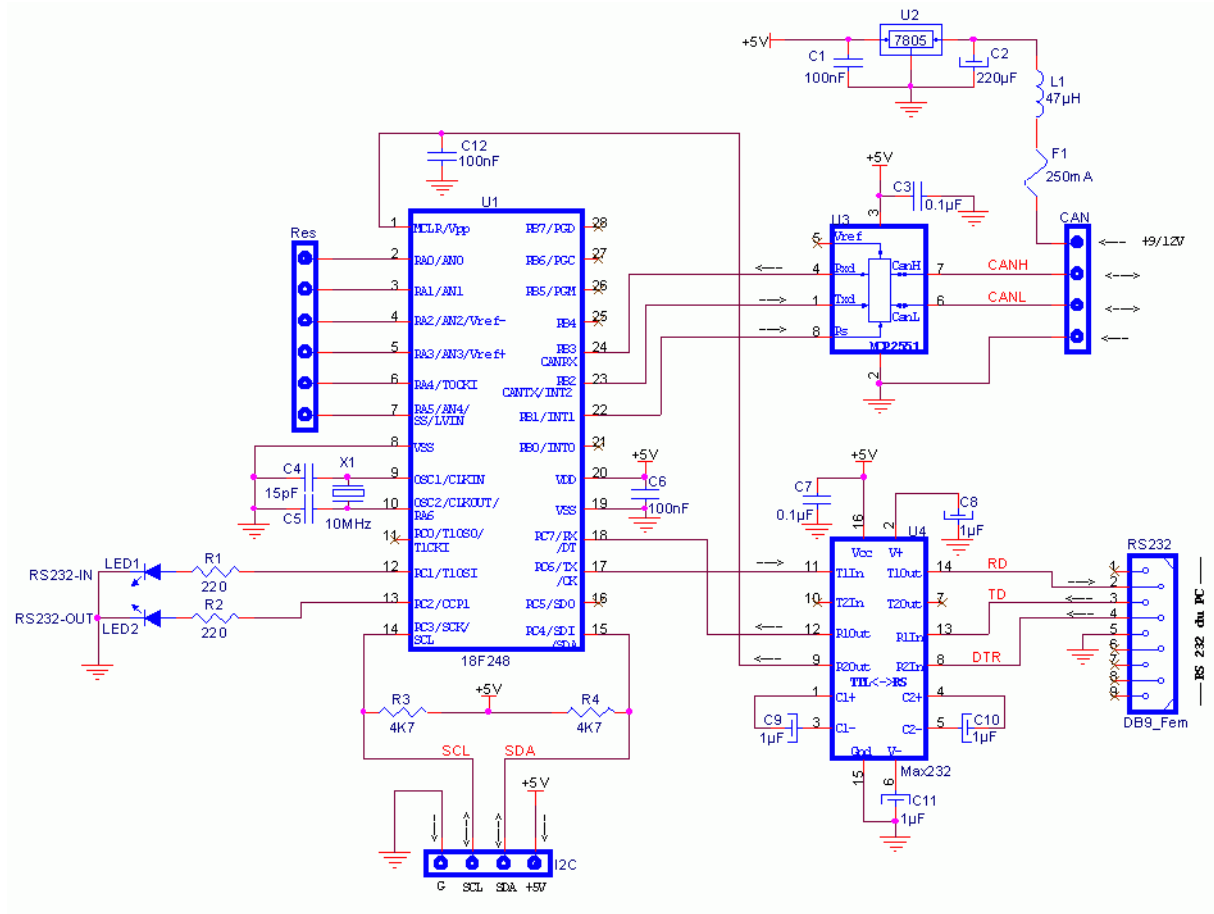
Les deux leds présentes sur la carte permettent donc de visualiser les trames au niveau du port RS232. Ceci signifie que seules seront signalées les trames CAN acceptées par les filtres et masques de la carte.

Par défaut, la carte à la mise sous tension n'accepte aucune trame CAN.

## 2. Réalisation pratique

### 2.1 Le schéma

Tout d'abord, voyons le schéma général :



Vous avez le schéma en plus grand dans les fichiers annexes. Que pouvons-nous en dire ?

En fait, cette interface est très simple : Nous avons le PIC au centre, cadencé à 40Mhz grâce à un quartz à 10 Mhz (boucle PLL en service). Pour rappel (voir cours-part5), il est possible de faire fonctionner le PIC à une vitesse quadruple de celle du quartz.

Cette méthode a l'avantage de limiter la fréquence du signal d'horloge, et donc de diminuer les interférences. Puisque cette possibilité existe, pourquoi s'en priver ?

Le PIC est connecté au driver de bus CAN par ses lignes CanTx et CanRx, et par la pin RB1 qui pilote l'entrée RS du MCP2551. Le datasheet de ce circuit est disponible sur le site de Microchip. Retenez simplement que la mise à la masse de cette pin permet le fonctionnement du MCP2551 à sa vitesse maximale. En réalité, on joue sur la raideur des signaux.

A l'autre extrémité du MCP2551, nous retrouvons nos lignes CANH et CANL. Le connecteur présent amène également l'alimentation de la platine, via deux des quatre fils du bus DOMOCAN.

Notez à ce niveau que l'alimentation +12V reçue est stabilisée à +5V via un simple 7805. Rien d'autre sur cette portion de schéma, exceptés les habituels condensateurs. La self série est là pour participer au blocage d'éventuels parasites amenés sur la ligne d'alimentation.

Revenons au PIC. Les lignes TX et RX sont reliées de façon classique à un max232 équipé de ses habituels condensateurs.

**A ce propos, je réponds une fois pour toute au nombreux courrier que m'a valu ce circuit à propos des interfaces BigoPic : Non, le condensateur C8 n'est pas incorrectement connecté. On peut tout aussi bien connecter ce condensateur entre la pin 2 et la masse qu'entre la pin 2 et le +5V. En fait, pour un signal alternatif, une alimentation continue se comporte comme un court-circuit. Je trouve pour ma part plus logique de référencer le condensateur à la masse.**

Si vous n'êtes pas convaincus, allez sur les sites des constructeurs Maxim et Texas Instruments et téléchargez le datasheet du MAX232. Chez Maxim le condensateur est référencé au +5V, alors que chez Texas il est référencé à la masse. Il s'agit pourtant bien du même circuit.

Ceci étant dit, à l'autre extrémité du MAX232, on retrouve notre habituel connecteur DB9. Le contrôle de flux s'effectue par reconnaissance des trames et par dépassement de temps. Il n'y a pas de ligne de contrôle hardware de flux.

La ligne DTR du port RS232 sera utilisée pour permettre un reset à distance de notre carte d'interface, à partir du PC.

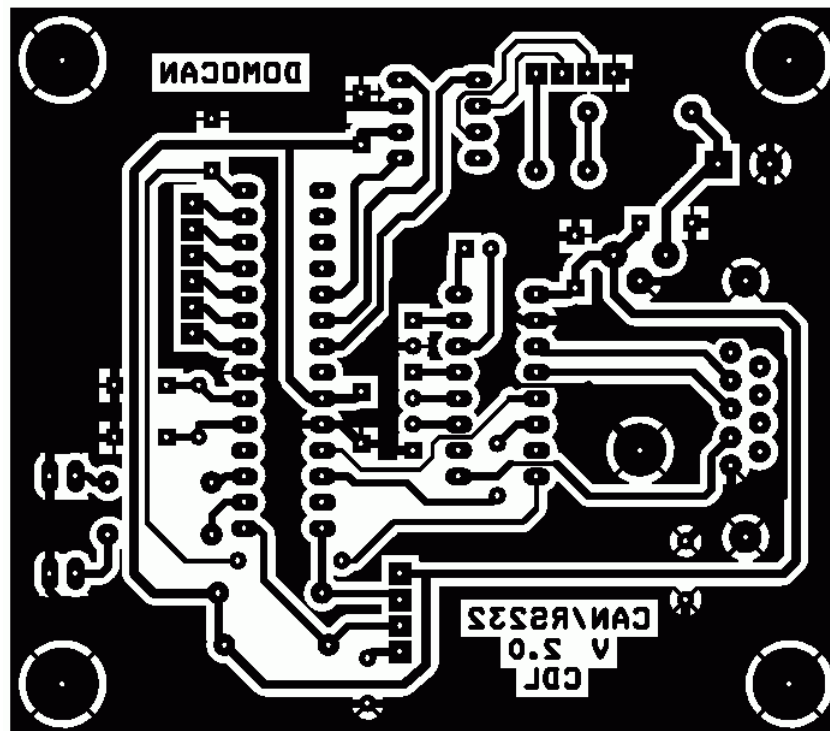
Le pic pilote également deux LEDs, chargées de visualiser le trafic RS232 entrant et sortant, du classique.

J'ai ajouté 1 connecteur et 2 résistances pour gérer un éventuel futur composant I<sup>2</sup>C. Si vous voulez faire des économies, il ne sert pas pour l'instant. Avec une modification du logiciel, l'interface pourrait également servir d'interface RS232/I<sup>2</sup>C.

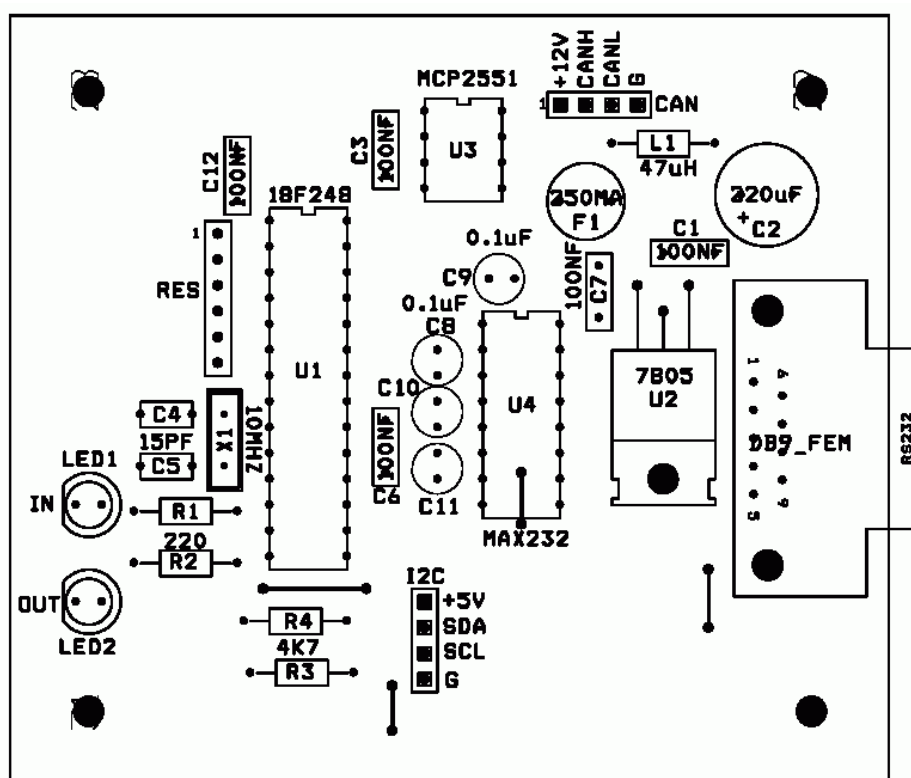
Un dernier connecteur donne accès à quelques pins de réserve, au cas où... De nouveau, vous n'êtes pas obligé de placer ce connecteur. Les dits connecteurs sont du reste de simples barres de contact sécables.

## **2.2 Le typon et l'implantation des composants**

Le typon est fourni à la bonne échelle dans un fichier séparé, se trouve simplement ici une représentation pas à l'échelle pour que ce document soit complet.



Voici l'implantation des composants :

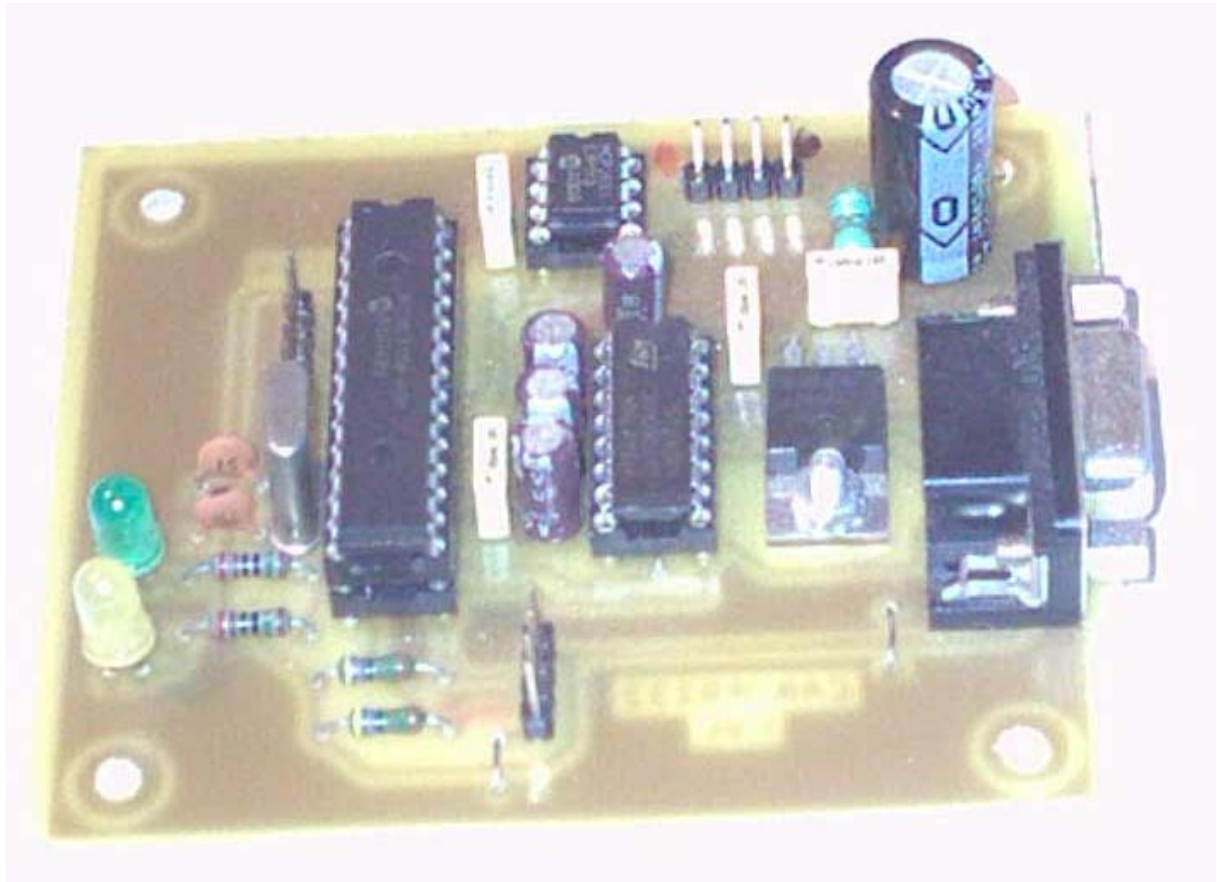


### **2.3 Réalisation pratique**

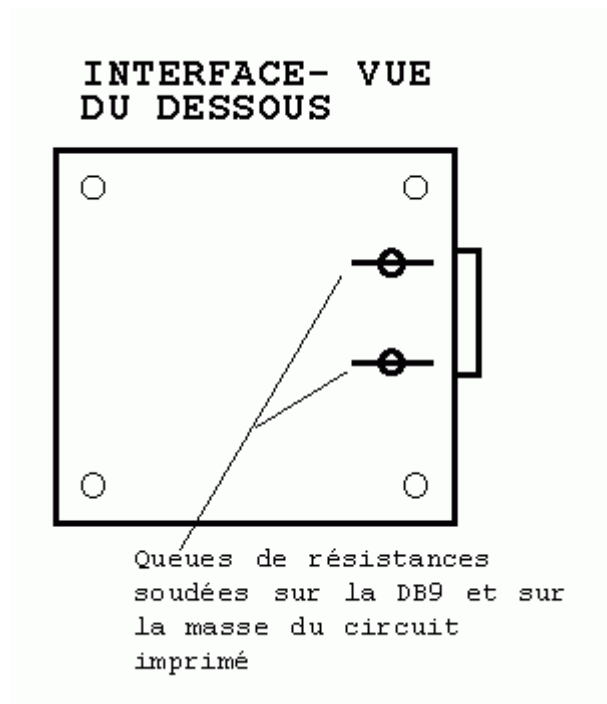
La réalisation pratique amène peu de commentaires. Soudez dans l'ordre habituel des composants. Le 7805 sera monté couché et vissé sur le circuit, il est inutile de lui prévoir un radiateur.

Forez à 3mm pour le trou du 7805 et pour les pattes de fixation de la DB9 femelle.

Voici une photo de la carte terminée. Les plus attentifs dénoteront de légères différences avec le schéma d'implantation, il s'agit ici en fait de la carte version 1.0 que j'ai utilisée pour le développement, et qui ne contenait pas la ligne de reset à distance.



Lors de la réalisation du circuit imprimé de la carte, veillez bien à ce que la masse de la fiche DB9 soit connectée correctement avec la masse du circuit imprimé. Pour ma part, pour assurer cette liaison de façon certaine, j'ai soudé une queue de résistance au travers des pattes de maintien de la fiche, et j'ai soudé ces queues à la fois aux pattes de maintien et à la masse du circuit imprimé.



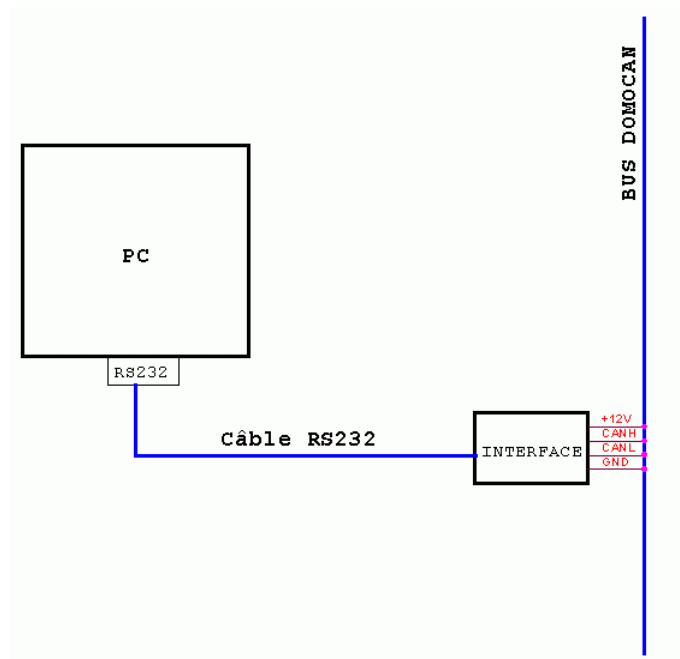
En procédant de la sorte, les pattes de maintien à clipser sont maintenant soudées, ce qui assure une meilleure mise à la masse et une meilleure rigidité mécanique dans le temps.

## **2.4 Installation**

N'oubliez pas de suivre les prescriptions du réseau CAN lors de la mise en œuvre de cette carte, et notamment que la liaison entre la carte et le réseau CAN doit être courte, de l'ordre de quelques cm. Par contre, votre câble RS232 pourra être un peu plus long, tout en respectant les limites de la norme RS232.

Prévoyez donc plusieurs points d'accès à votre réseau CAN, plutôt que d'utiliser de longs câbles, surtout à ces vitesses. Je vous conseille de placer votre circuit d'interface dans un boîtier à l'extrémité de votre câble RS232, afin de limiter la longueur de la liaison CAN.

Voici un exemple d'application correcte :



Notez que si vous désirez placer votre interface dans un petit boîtier, et c'est bien compréhensible, et que vous désirez également placer un connecteur à demeure sur votre câble CAN pour y connecter votre interface, ce qui est également logique, je vous conseille alors d'adopter les connecteurs standards.

Voici les recommandations CIA DR-303-1 en la matière (recommandations qui n'ont rien d'obligatoires, ce sont des conventions) :

Ces connecteurs pourront simplement être des connecteurs de type DB9, vous utiliserez par exemple un DB9 femelle pour votre boîtier mural, connecté au CAN, et un DB9 mâle sur le boîtier de votre interface. C'est le meilleur choix.

- Pin 1 : N.C. (non connectée)
- Pin 2 : CANL
- Pin 3 : Masse
- Pin 4 : N.C.
- Pin 5 : Blindage du câble CAN (s'il existe)
- Pin 6 : Masse
- Pin 7 : CANH
- Pin 8 : N.C.
- Pin 9 : + Alimentation

Si vous décidez de respecter ce brochage, vous relierez ensemble les pins 3, 5, et 6, puisque nous n'utilisons pas d'optocoupleurs d'entrée, les masses sont donc communes.

N'oubliez pas de toujours interconnecter les masses chaque fois que c'est possible. N'oubliez surtout pas que votre PC, s'il dispose d'une alimentation secteur, devra être impérativement relié au minimum au conducteur de protection de votre installation (appelé à tort : prise de terre).

Vous pouvez également utiliser un connecteur de type RJ45, avec le brochage suivant :

Pin 1 : CANH  
Pin 2 : CANL  
Pin 3 : Masse  
Pin 4 : N.C.  
Pin 5 : N.C.  
Pin 6 : Blindage du câble CAN (s'il existe)  
Pin 7 : Masse  
Pin 8 : + Alimentation

Même remarque dans ce cas, interconnectez les pins 3, 6 et 7.

Vous pouvez aussi utiliser des connecteurs DIN, RJ10 etc. Adoptez soit un brochage standard (cherchez la norme CIA DR-303-1 sur Internet) ou votre propre brochage.

### **3. Communications**

#### **3.1 Les trames côté DOMOCAN**

A ce niveau, c'est très simple : sont acceptées en entrée les trames étendues de type data compatibles avec le cahier des charges DOMOCAN.

La vitesse est par défaut de 500Kbits/s, mais peut être commutée par logiciel à 100Kbits/s. Attention, en cas d'utilisation à 100Kbits/s, la commutation doit être effectuée avant la connexion de l'interface sur le bus CAN.

En effet, sous peine de problèmes au niveau du bus, il est interdit de connecter des périphériques sur le même bus travaillant avec des vitesses différentes.

Sont émises des trames CAN de même nature compatibles avec le bus DOMOCAN.

#### **3.2 Trames RS232**

Au niveau de la partie RS232, les choses sont un peu plus compliquées. En fait, étant donné l'absence de lignes de contrôle, il faut que l'interface sache ce qu'on lui envoie, et quelle est sa longueur. Idem lorsqu'il s'agit d'envoyer vers le PC des trames CAN en provenance du réseau domocan.

Pour résoudre ce problème, les trames CAN sont encapsulées dans une trame particulière. Chaque communication avec le PC s'effectue donc à l'aide de trames spécifiques qui indiquent la nature de leur contenu.

De nouveau, nous nous retrouvons face à la nécessité d'utiliser des conventions. Voici celles que j'ai choisies :

#### **3.2. Structure d'une trame RS232**

Rassurez-vous, les trames du côté RS232 sont beaucoup plus simples que les trames CAN. En fait, une trame est, par convention, toujours constituée de :

- 1 octet qui spécifie la commande RS232 (à ne pas confondre avec la commande CAN).
- 1 octet qui indique le nombre total d'octets de data
- Un certain nombre d'octets de data, selon la valeur de l'octet précédent

Une trame commence au premier start-bit rencontré, et se termine après l'absence de réception d'un octet durant 205µs. A 10 bits par octet, et à 115200 bauds, chaque octet a une durée de 86µs. La durée de 205µs correspond à l'absence de réception durant un temps correspondant à 2,4 octets.

Vous pourriez penser que ce temps mort diminue le débit, mais en fait ce temps mort n'intervient que dans le sens PC-PIC, et c'est le sens où il y a le moins de trafic. Dans le sens

inverse, la détection des trames s'effectue sur la commande et la longueur de la trame, il n'y a donc pas de temps mort.

Une fois le temps atteint, on pourra effectuer une vérification de la longueur de la trame, le second octet donnant le nombre de data qu'on aurait du recevoir, aucune trame ne pouvant comporter moins de 2 octets par définition. Ainsi, une coupure de liaison n'est pas considérée comme la réception d'une trame valide.

Au niveau de la liaison PC vers PIC, le contrôle de flux s'effectue comme suit :

- Le PC envoie sa trame sur le port RS232, et attend la confirmation du pic à la commande
- Le PIC reçoit les octets, puis un silence de 205µs
- Le PIC traite la trame, et envoie la confirmation à la commande
- Le PC reçoit la confirmation et est prêt à émettre une nouvelle trame.

L'envoi des commandes est donc sécurisé, car on vérifie l'intégrité de la trame d'après le nombre d'octets effectifs reçus. Ceci se paye par une chute de vitesse dans ce sens, mais ça n'a pas d'importance, le logiciel PC n'étant qu'un outil de paramétrage et de configuration, il n'a pas besoin d'exploiter l'intégralité du débit de 115200 bauds.

De plus, lors du traitement de la trame, des vérifications supplémentaires sont effectuées pour les trames critiques.

L'expérience m'a montré qu'avec une liaison correcte, il n'y a pas besoin de contrôle supplémentaire genre CRC, les données arrivent toujours de façon intègre, et un problème n'aurait de toutes façons aucune conséquence néfaste.

Dans le sens PIC vers PC, le contrôle s'effectue comme suit :

- Le PIC envoie une trame vers le PC
- Le PIC envoie une trame vers le PC
- ...
- Le PC reçoit des données en vrac. Il analyse le second octet reçu, et sépare la première trame en fonction de l'octet de longueur reçu.
- Le PC traite la trame
- Le PC sépare la seconde trame en fonction du second octet de ce qui reste
- Le PC traite la trame

Si le PC détecte une incohérence (commande inexistante), ou un temps mort trop important entre la réception du premier octet et du dernier, il vide son buffer d'entrée et se resynchronise pour la réception d'une nouvelle trame.

Le sens PIC vers PC travaille donc à vitesse maximale, sans aucun contrôle de flux supplémentaire, la vitesse d'analyse des trames CAN est donc maximale.

Le PC que j'ai utilisé supporte sans problème une arrivée des données aux vitesses utilisées. Pour information, j'ai utilisé un PC équipé d'un Athlon 1500+ sous WinMe.

### **3.3 Commandes RS232 supportées**

Puisque nous avons des trames comportant un octet de commande, alors nous avons forcément plusieurs sortes de commandes possibles. Nous allons les étudier. Je n'ai pas donné de nom aux commandes, elles ne sont utilisées qu'au niveau des liaisons PC/interface, leur traitement est unique dans le programme PC, il était donc inutile de les nommer, puisque vous n'aurez jamais besoin de les utiliser (sauf si vous écrivez votre propre programme de gestion).

J'explique donc les commandes par rapport à leur valeur hexadécimale.

#### **3.3.1 La commande 0x50**

Cette commande reçue par l'interface n'est accompagnée d'aucun octet de data. La trame se suffit donc à elle-même. Fort logiquement, la trame sera donc composée de 2 octets : 0x50, 0x00. Autrement dit : commande 0x50, 0 octet de data.

Son rôle est de faire passer la partie CAN de l'interface à la vitesse de 500Kbits/s, ce qui est également la valeur par défaut.

Le pic répondra au PC par la même trame, à savoir 0x50 (écho).

#### **3.3.2 La commande 0x51**

Exactement identique en fonctionnement à la précédente, cette trame RS232 reçue par l'interface commute la partie CAN en 100Kbits/s. Comme il n'y a pas non plus d'octets de data, la commande sera composée de 2 octets : 0x51, 0x00.

Une fois la carte passée à la nouvelle vitesse, le pic renverra la trame vers le PC : 0x51, 0x00.

#### **3.3.3 La commande 0x52**

Le PC envoie cette commande vers l'interface pour lui demander de renvoyer le contenu de ses filtres et masques (par défaut, tous les bits à 1). Dans le sens PC vers interface, cette trame se suffit à elle-même, et ne contient donc aucun octet de data : 0x52, 0x00.

Par contre, cette fois, le PIC va répondre au PC en envoyant le contenu des registres demandés. La réponse s'effectuera donc avec la même commande, mais accompagnée de 32 octets de data.

La réponse sera donc constitué de la trame suivante :

0x52 : commande  
0x20 : 32 octets de data  
data1 : filtre 0 / bits 28 à 21  
data2 : filtre 0 / bits 20 à 16  
data3 : filtre 0 / bits 15 à 8  
data4 : filtre 0 / bits 7 à 0

data5 : filtre 1 / bits 28 à 21  
 data6 : filtre 1 / bits 20 à 16  
 data7 : filtre 1 / bits 15 à 8  
 data8 : filtre 1 / bits 7 à 0  
 data9 : filtre 2 / bits 28 à 21  
 data10 : filtre 2 / bits 20 à 16  
 data11 : filtre 2 / bits 15 à 8  
 data12 : filtre 2 / bits 7 à 0  
 data13 : filtre 3 / bits 28 à 21  
 data14 : filtre 3 / bits 20 à 16  
 data15 : filtre 3 / bits 15 à 8  
 data16 : filtre 3 / bits 7 à 0  
 data17 : filtre 4 / bits 28 à 21  
 data18 : filtre 4 / bits 20 à 16  
 data19 : filtre 4 / bits 15 à 8  
 data20 : filtre 4 / bits 7 à 0  
 data21 : filtre 5 / bits 28 à 21  
 data21 : filtre 5 / bits 20 à 16  
 data23 : filtre 5 / bits 15 à 8  
 data24 : filtre 5 / bits 7 à 0  
 data25 : masque 0 / bits 28 à 21  
 data26 : masque 0 / bits 20 à 16  
 data27 : masque 0 / bits 15 à 8  
 data28 : masque 0 / bits 7 à 0  
 data29 : masque 1 / bits 28 à 21  
 data30 : masque 1 / bits 20 à 16  
 data31 : masque 1 / bits 15 à 8  
 data32 : masque 1 / bits 7 à 0

Ce sont les filtres et masques qui autorisent la réception des trames CAN. Le masque0 et les filtres 0 et 1 concernent le buffer0, tandis que les autres concernent le buffer1. Le buffer0 est prioritaire par rapport au buffer1, mais ceci ne devrait avoir d'importance que dans de rares cas.

### **3.3.4 La commande 0x54**

Cette commande est envoyée du PC vers le PIC pour signaler qu'on désire annuler la transmission des trames CAN en attente d'émission. Aucun octet de donnée n'accompagne la commande : 0x54, 0x00.

Le PIC répondra au PC en renvoyant après l'arrêt l'état de deux de ses registres internes : TXB0CON et COMSTAT. Ces registres donnent une indication de l'état de la liaison CAN, en rapportant notamment les erreurs d'émission et de réception et l'état des compteurs internes. Je vous renvoie au cours-part5 pour plus de détails sur ces registres.

Le PIC répond donc par la même commande accompagnée de 2 octets de données, soit :

0x54 : commande  
 0x02 : 2 octets de data

data1 : valeur du registre TXB0CON  
data2 : valeur du registre COMSTAT

### **3.3.5 La commande 0x60**

Cette commande est très importante, elle est émise par le PC vers la carte d'interface pour lui dire d'envoyer la trame CAN dont il est question vers le bus CAN. Si vous avez bien suivi, vous avez compris que la trame CAN en question est en réalité dans les octets de data de la trame RS232.

Comme les trames CAN ont une longueur comprise entre 5 octets (ID + longueur) et 13 octets (ID + longueur + 8 octets de data), on pourrait penser que le nombre d'octets de données pourrait varier de 5 à 13 (décimal). En fait, étant donné qu'on a déjà envoyé le nombre total d'octets de data RS232, il est inutile de renvoyer le nombre d'octets de données de la trame CAN (puisque les data RS232 sont la trame CAN). On économise donc un octet.

Si je m'exprime correctement, vous comprenez qu'une trame CAN présente la structure suivante :

ID sur 4 octets / nombre d'octets de data CAN/ 0 à 8 octets de data, soit 5 à 13 octets

La trame RS232 correspondante aura la structure suivante :

0x60 /nombre d'octets de data de la trame RS232 /ID sur 4 octets / 0 à 8 octets de data.

Le nombre d'octets de data CAN pourra être calculé aisément par une simple soustraction, puisque le nombre total d'octets de data RS232 est par définition égal à la longueur de la trame CAN. Or, l'ID a une longueur fixe de 4 octets.

Sous forme d'une formule : Nombre d'octets de data CAN = nombre d'octets de data RS232 - 4

La trame est donc de la forme :

0x60 : commande  
nbre data : de 0x04 à 0x0C  
data1 : ID CAN / bits 28 à 21 = commande CAN  
data2 : ID CAN / bits 20 à 16 + type de trame  
: b7 : 1 si trame remote, 0 si trame data  
: b6/b5 : toujours 0  
: b4/b0 : ID CAN / bits 20 à 16 = extension de commande  
data3 : ID CAN / bits 15 à 8 (paramètre 1)  
data4 : ID CAN / bits 7 à 0 (paramètre 2)  
data5 : facultatif : octet 0 de data CAN  
data6 : facultatif : octet 1 de data CAN  
data7 : facultatif : octet 2 de data CAN  
data8 : facultatif : octet 3 de data CAN  
data9 : facultatif : octet 4 de data CAN  
data10 : facultatif : octet 5 de data CAN

data11 : facultatif : octet 6 de data CAN  
data12 : facultatif : octet 7 de data CAN

Le pic pourra répondre à l'aide d'une des deux trames RS232 suivantes :

0x60, 0x00 : tout s'est bien passé  
0x6F, 0x00 : la longueur de la trame reçue est incorrecte

Notez une chose importante : vous savez que le système DOMOCAN ne gère que les trames CAN de type « data » et non les trames de type « remote ». Si vous regardez plus haut, vous voyez qu'en plaçant le bit 7 de l'extension de commande (donc en ajoutant 0x80), vous pouvez demander l'envoi d'une trame de type « remote » sur le bus CAN.

Ceci est inutilisé sur le DOMOCAN, mais rend cette interface un peu plus universelle, et vous permet également de créer des cartes qui gèreraient les trames « remote ». Cette façon de faire ne perturbera aucune carte DOMOCAN présente, votre carte restera donc compatible à condition d'adapter le logiciel PC en conséquence.

### **3.3.6 La commande 0x70**

Toutes les précédentes commandes étaient émises du PC vers l'interface, cette dernière se contentant de répondre à la commande reçue.

La commande 0x70 est la seule qui est générée par la carte d'interface elle-même. En fait, c'est le pendant de la commande 0x60. Cette commande permet d'envoyer vers le PC une trame CAN reçue sur le bus CAN.

Pour faire simple, la commande 0x60 est envoyée par le PC pour placer une trame sur le bus CAN, alors que la commande 0x70 est envoyée par l'interface pour lui fournir la trame CAN qui vient d'être reçue. Exprimé plus simplement : 0x60 = émission, 0x70 = réception (vu côté PC).

Imaginons que vous vouliez envoyer une commande Cmd\_RSoftW à une carte DOMOCAN, à partir du PC.

Vous envoyez donc une commande RS232 0x60 :

0x60, 0x04, commande CAN, commande type « carte », adresse carte, numéro réseau

L'interface répond par :

0x60, 0x00

La trame suivante est envoyée sur le bus CAN :

Cmd\_RSoftW, commande type « carte », adresse carte, numéro réseau, 0 octet de data

La carte DOMOCAN ciblée va répondre :

Cmd\_SoftW, commande de type « carte », adresse carte, numéro réseau, 7 octets de data, data0/6 = valeurs renvoyées par la carte.

La trame CAN arrive sur le bus CAN, entre dans la carte interface, qui va renvoyer au PC (si les masques sont correctement configurés) sur le port RS232 :

0x70, 0x0B, Cmd\_SoftW, commande « carte », adresse carte, numéro réseau, data0/6 = valeurs renvoyées par la carte.

Vous voyez donc que la trame renvoyée est « encapsulée », c'est-à-dire dans le cas présent, précédée de 2 octets qui permettent d'identifier qu'il s'agit d'une trame CAN et qui détermine sa longueur. Le nombre d'octets de data CAN est supprimé, puisqu'il peut être retrouvé en effectuant  $0x0B - 0x04 = 0x07$ .

La structure de la trame RS232 est, comme vous venez de le voir, strictement identique à celle de la commande 0x60.

**Notes :**

## 4. Analyse du logiciel PIC

### 4.1 Le fichier CAN-RS232.asm

Je vais maintenant, selon mon habitude, vous détailler le fichier source. Cette carte, je le rappelle, est la seule qui n'utilise pas les fichiers « domodef.inc » et « domoboot.inc ». Cette carte n'est donc pas bootloadable, c'est logique puisque la carte est reliée au PC par son interface RS232 et non par son interface CAN.

Commençons l'étude par l'en-tête du fichier. Dans les chapitres qui précèdent, j'ai déjà donné les explications qui apparaissent ici, je n'y reviens donc pas.

```
;*****
; Description sommaire                                     *
; -----                                                *
;                                                         *
; Interface RS232 pour réseau domotique can "DOMOCAN".    *
; Permet l'interaction avec l'ensemble des cartes présentes *
; Permet le bootloading des cartes distantes              *
;                                                         *
; Communication RS232 en 115200 bauds                     *
; Communication CAN en 500 Kbauds                         *
;                                                         *
;*****
;                                                         *
; NOM : Interface RS232 pour bus CAN                      *
; Date création : 31/05/2003                             *
; Date modification : 31/10/2003                         *
; Version : 2.0                                           *
; Circuit : Interface CAN/RS232                          *
; Auteur : Bigonoff                                       *
;                                                         *
;*****
;                                                         *
; Fichier requis: P18F248                                 *
;                                                         *
; Fréquence de travail : 40 MHz                           *
; Fréquence du quartz : 10 MHz                           *
;                                                         *
;*****
; Historique                                              *
; -----                                                *
;                                                         *
; V1.0 : le 10/06/2003 : Première version opérationnelle testée *
; V2.0 : le 31/10/2003 : Ajout d'un fusible et d'un reset par PC *
;                                     Refonte totale, ajout d'un buffer CAN software *
;                                     Première version en ligne *
;                                                         *
;*****
; Pins utilisées :                                       *
; -----                                                *
;                                                         *
; OSC1/CLKI : Quartz 10 MHz                               *
; OSC2/CLKO/RA6 : Quartz 10 MHz                           *
;                                                         *
; RC1/T1OSI : Sortie LED1 (trame RS232 reçue)            *
; RC2/CCP1 : Sortie LED2 (trame RS232 émise)             *
```

```

; RC3/SCK/SCL      : SCL connecteur I2C (utilisation future)      *
; RC4/SDI/SDA      : SDA connecteur I2C (utilisation future)      *
; RC6/TX/CK        : Emission RS232                               *
; RC7/RX/DT        : Réception RS232                             *
;                                                           *
; RB1/INT1         : Sortie pour commande RS du MCP2551 (0 = High speed) *
; RB2/CANTX/INT2   : Emission CAN                                 *
; RB3/CANRX        : Réception CAN                               *
;                                                           *
; *****
;                                                           *
; Trames RS232 entre PIC et PC                                   *
; -----                                                       *
; Octet 1 : Commande                                             *
; Octet 2 : nombre d'octets de données                           *
; Octets x: octets de données                                    *
;                                                           *
; La fin de la réception s'effectue par la présence d'un temps-mort. *
; On vérifie l'intégrité par le nombre d'octets reçus           *
;                                                           *
; Commandes supportées PC->PIC                                   *
; -----                                                       *
;                                                           *
; 0x50 : Passage du CAN en mode 500Kbits/s                       *
;       : octets de données : aucun                               *
;       : réponse du pic : 0x50 0x00                             *
;                                                           *
; 0x51 : Passage du CAN en mode 100Kbits/s                       *
;       : octets de données : aucun                               *
;       : réponse du pic : 0x51 0x00                             *
;                                                           *
; 0x52 : Demande du contenu des filtres et masques CAN courants dans *
;       l'interface.                                             *
;       : octets de données : aucun                               *
;       : réponse du pic : 0x52 0x20 (32 octets de données), suivis par : *
;       filtre 0 : bits 28 à 21 (commande)                       *
;       filtre 0 : bits 20 à 16 (extension de commande)         *
;       filtre 0 : bits 15 à 8 (paramètre 1 = EIDH)              *
;       filtre 0 : bits 7 à 0 (paramètre 2 = EIDH)               *
;       filtre 1 : 4 octets idem                                  *
;       filtre 2 : 4 octets idem                                  *
;       filtre 3 : 4 octets idem                                  *
;       filtre 4 : 4 octets idem                                  *
;       filtre 5 : 4 octets idem                                  *
;       masque 0 : 4 octets idem                                  *
;       masque 1 : 4 octets idem                                  *
;                                                           *
; 0x53 : réception des filtres et masques CAN en provenance du PC *
;       Le pic passe alors en mode configuration, puis configure ses *
;       registres.                                               *
;       : octets de données : 32                                  *
;       filtre 0 : bits 28 à 21 (commande)                       *
;       filtre 0 : bits 20 à 16 (extension de commande)         *
;       filtre 0 : bits 15 à 8 (paramètre 1 = EIDH)              *
;       filtre 0 : bits 7 à 0 (paramètre 2 = EIDH)               *
;       filtre 1 : 4 octets idem                                  *
;       filtre 2 : 4 octets idem                                  *
;       filtre 3 : 4 octets idem                                  *
;       filtre 4 : 4 octets idem                                  *
;       filtre 5 : 4 octets idem                                  *
;       masque 0 : 4 octets idem                                  *

```

```

;          masque 1 : 4 octets idem *
;          : réponse du pic : 0x53 0x00 *
; *
; 0x54 : Abandonner arrêt de la transmission en cours *
;          : octet de donnée : aucun *
;          : réponse du pic : 0x54,0x02,TXB0CON,COMSTAT *
; *
; 0x60 : Réception d'une trame CAN en provenance du PC pour envoi sur le *
;          bus CAN *
;          : octets de données : de 4 à 12 *
;          Commande = identificateur bits 28 à 21 *
;          Extended = extension de commande : *
;                  b7 : 1 si trame remote, 0 pour trame data *
;                  b6/b5 : 0 *
;                  b4/b0 : ID bits 20 à 16 *
;          Paramètre1 = EIDH *
;          Paramètre2 = EIDL *
;          data 0 : facultatif : donnée octet 0 *
;          data 1 : facultatif : donnée octet 1 *
;          data 2 : facultatif : donnée octet 2 *
;          data 3 : facultatif : donnée octet 3 *
;          data 4 : facultatif : donnée octet 4 *
;          data 5 : facultatif : donnée octet 5 *
;          data 6 : facultatif : donnée octet 6 *
;          data 7 : facultatif : donnée octet 7 *
;          : réponse du pic : 0x60 0x00 : trame envoyée *
;                  0x6F 0x00 : longueur de commande incorrecte *
; *
; 0x70 : Envoi d'une trame CAN reçue du bus vers le PC *
;          : octets de données : de 4 à 12 *
;          Commande = identificateur bits 28 à 21 *
;          Extended = extension de commande : *
;                  b7 : 1 si trame remote, 0 pour trame data *
;                  b6/b5 : 0 *
;                  b4/b0 : ID bits 20 à 16 *
;          Paramètre1 = EIDH *
;          Paramètre2 = EIDL *
;          data 0 : facultatif : donnée octet 0 *
;          data 1 : facultatif : donnée octet 1 *
;          data 2 : facultatif : donnée octet 2 *
;          data 3 : facultatif : donnée octet 3 *
;          data 4 : facultatif : donnée octet 4 *
;          data 5 : facultatif : donnée octet 5 *
;          data 6 : facultatif : donnée octet 6 *
;          data 7 : facultatif : donnée octet 7 *
; *
; *****

```

En lisant ces commentaires, vous retrouvez les assignations des pins, le fonctionnement des trames etc., selon mon habitude.

Ensuite, nous trouvons la déclaration de notre type de PIC :

```

TYPEPIC = 0x01          ; 0x01 = 18F248
                        ; 0x02 = 18F258

```

Vous voyez que je vous ai laissé le choix entre deux possibilités, soit le 18F248, soit le 18F258. Utilisez celui que vous voulez, à condition de ne pas oublier de changer dans votre fichier source.

Notez que le fichier .hex produit fonctionnera dans ce cas indépendamment dans les deux modèles de pic.

Le numéro de type de PIC renseigné ici n'est pas interrogeable par le PC (par définition), ce numéro sert exclusivement à choisir les bonnes options dans les bits de configuration (moins nombreux sur les 18Fx48).

```
IF TYPEPIC == 0x01
    LIST      p=18F248      ; Définition de processeur
    #include <p18F248.inc>    ; fichier include
ELSE
    LIST      p=18F258      ; Définition de processeur
    #include <p18F258.inc>    ; fichier include
ENDIF
```

Cet assemblage conditionnel précise le nom du fichier include en fonction du type de PIC choisi.

Voyons nos définitions et assignations :

```
;=====
;                                     DEFINES ET ASSIGNS                               =
;=====

#define LEDR   PORTC,1      ; LED réception de trame PC->PIC
#define LEDE   PORTC,2      ; LED d'émission de trame -> PC
#define OUTRS  PORTB,1      ; ligne RS MCP2551 (doit être à 0)

TRISBVAL EQUB'11111001'    ; direction PORTB
TRISCVAL EQUB'11111001'    ; direction PORTC
```

Pas grand chose à dire ici. On précise les valeurs de TRISB et TRISC en fonction de l'électronique, on indique que la sortie vers la ligne RS du MCP2551 sera sur RB1, et que les leds seront sur les pins RC1 et RC2.

Voyons les configurations :

```
;=====
;                                     CONFIGURATIONS                               =
;=====
;-----
;                                     Type d'oscillateur                               -
;-----

__CONFIG __CONFIG1H, _OSCS_OFF_1H & _HSPLL_OSC_1H

;_OSCS_OFF_1H      ; oscillateur principal toujours actif
;_HSPLL_OSC_1H     ; oscillateur haute vitesse + PLL (*4)

;-----
;                                     Protections en lecture                               -
;-----
```

```

IF TYPEPIC == 0x01
__CONFIG __CONFIG5L, _CP0_OFF_5L & _CP1_OFF_5L
ELSE
__CONFIG __CONFIG5L, _CP0_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L & _CP3_OFF_5L
ENDIF

__CONFIG __CONFIG5H, _CPB_OFF_5H & _CPD_OFF_5H

;_CP0_OFF_5L      ; bloc 0 non protégé
;_CP1_OFF_5L      ; bloc 1 non protégé
;_CP2_OFF_5L      ; bloc 2 non protégé
;_CP3_OFF_5L      ; bloc 3 non protégé

;_CPB_OFF_5H      ; bloc boot non protégé
;_CPD_OFF_5H      ; mémoire eeprom non protégée

;-----
;
;                      Protections en écriture
;-----

IF TYPEPIC == 0x01
__CONFIG __CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L
ELSE
__CONFIG __CONFIG6L, _WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L
ENDIF

__CONFIG __CONFIG6H, _WRTB_OFF_6H & _WRTC_OFF_6H & _WRTD_OFF_6H

;_WRT0_OFF_6L      ; bloc 0 non protégé en écriture
;_WRT1_OFF_6L      ; bloc 1 non protégé
;_WRT2_OFF_6L      ; bloc 2 non protégé
;_WRT3_OFF_6L      ; bloc 3 non protégé

;_WRTB_OFF_6H      ; bloc boot non protégé
;_WRTC_ON_6H        ; registres de configuration protégés en écriture
;_WRTD_OFF_6H      ; mémoire eeprom non protégée

;-----
;
;                      Protections contre la lecture de tables
;-----

IF TYPEPIC == 0x01
__CONFIG __CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L
ELSE
__CONFIG __CONFIG7L, _EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L &
_EBTR3_OFF_7L
ENDIF

__CONFIG __CONFIG7H, _EBTRB_OFF_7H

;_EBTR0_OFF_7L      ; bloc 0 non protégé
;_EBTR1_OFF_7L      ; bloc 1 non protégé
;_EBTR2_OFF_7L      ; bloc 2 non protégé
;_EBTR3_OFF_7L      ; bloc 3 non protégé

;_EBTRB_OFF_7H      ; bloc boot non protégé

;-----
;
;                      Réactions à la tension d'alimentation
;-----

```

```

    __CONFIG    __CONFIG2L, _BOR_ON_2L & _BORV_42_2L & _PWRT_ON_2L

; _BOR_ON_2L          ; reset sur chute de tension en service
; _BORV_42_2L         ; reset si tension < 4.2V
; _PWRT_ON_2L         ; retard de démarrage sur mise sous tension en service

;-----
;                               Paramètres du watchdog                      -
;-----

    __CONFIG    __CONFIG2H, _WDT_ON_2H & _WDTPS_4_2H

; _WDT_ON_2H          ; watch dog en service
; _WDTPS_4_2H         ; postdiviseur = .4

;-----
;                               Modes spéciaux                          -
;-----

    __CONFIG    __CONFIG4L, _DEBUG_OFF_4L & _LVP_OFF_4L & _STVR_ON_4L

; _DEBUG_OFF_4L       ; mode debugger hors-service
; _LVP_OFF_4L         ; programmation basse tension hors-service
; _STVR_ON_4L         ; reset sur erreur de pile en service

```

Les configurations sont nombreuses, mais ne prêtent pas à beaucoup de commentaires. Notez qu'on travaille avec un oscillateur avec PLL, que pratiquement aucune zone n'est protégée en lecture et en écriture (logique pour un programme distribué gratuitement), et que tous les resets de sécurité sont activés (watchdog, débordement de pile, tension d'alimentation).

Analysons nos quelques macros :

```

SENDRS macro                ; lancer l'émission RS232
    bsf    LEDE              ; allumer LED émission
    clrf   nbdje             ; aucun octet encore traité
    lfsr   FSR1, tramesend    ; pointer sur trame à envoyer
    bsf    PIE1, TXIE         ; autoriser interruptions émission USART
endm

```

Cette macro lance les interruptions d'émission USART. Si le buffer d'émission est vide, ceci va générer automatiquement une interruption, le remplissage du buffer se fera par cette interruption. La Led d'émission aura été préalablement allumée. Une variable qui sert de compteurs d'octets déjà envoyés est également initialisée.

Le pointeur indirect FSR1 pointe sur le premier octet de la trame à envoyer.

```

SETCONFIG macro              ; passer le can en mode configuration
    bsf    CANCON, REQOP2    ; demander le passage en mode configuration
    clrwdt                ; effacer watchdog
    btfss  CANSTAT, OPMODE2   ; tester si passage effectué
    bra    $-(2*2)           ; non, attendre passage effectif
endm

SETNORMAL macro              ; passer le can en mode normal
    bcf    OUTRS             ; MCP2551 en mode full speed
    clrf   CANCON            ; requête de passage en mode normal
    clrwdt                ; effacer watchdog

```

```

movf  CANSTAT,w          ; charger status actuel
andlw 0xE0                ; conserver mode en cours
btfss STATUS,Z           ; tester si mode normal
bra    $-(4*2)            ; non, attendre passage effectif
endm

```

Nous avons déjà analysé ces deux trames dans le document « présentation du DOMOCAN ». Souvenez-vous que ces trames permettent de placer le module CAN, soit en mode configuration (ce qui est indispensable pour modifier certains registres), soit en mode de fonctionnement normal (ce qui permet de communiquer sur le bus).

Voyons maintenant nos variables :

```

;=====
;                               VARIABLES ACCESS RAM                               =
;=====
; zone de 96 octets
; -----
CBLOCK  0x00                ; zone access ram de la banque 0

    tramesend : .34          ; trame RS232 à envoyer
    tramerec  : .34          ; trame RS232 reçue (laisser sous tramesend)
    nbrec     : 1            ; nombre d'octets RS232 reçus
    nbsend    : 1            ; nombre d'octets restant à envoyer sur RS232
    nbdje     : 1            ; nombre d'octets déjà envoyés

    ptrcr     : 1            ; ptr CAN sur la prochaine trame à recevoir
    ptrct     : 1            ; ptr buf 1 sur la prochaine trame à traiter

    local01   : 1            ; variable locale
    local02   : 1            ; variable locale

    flags     : 1            ; divers flags
                                ;   b0 : trame RS232 reçue

ENDC

#define TRAMEREC flags,0      ; trame RS232 reçue

```

Une trame RS232 fait une longueur maximale de 34 octets, et ce, dans le sens PC vers PIC ou dans le sens inverse. Ceci justifie la taille des deux premières variables.

Vous notez la présence de deux pointeurs, ptrcr et ptrct. Il s'agit de deux pointeurs cycliques qui fonctionnent comme ceci :

Une trame CAN est reçue dans le buffer d'entrée, on l'écrit à l'emplacement pointé par ptrct, puis on positionne le pointeur ptrcr sur l'emplacement suivant en effectuant une simple addition. Si on sort de la zone du buffer, on re pointe sur le début de la zone du buffer.

Le programme principal vérifie si ptrcr est égal à ptrct. Si oui, on n'a aucune trame dans le buffer, si non, alors on traite la trame dont le début est pointé par ptrct, et ensuite on effectue également une addition de même nature sur ptrct. Ainsi, on peut recevoir plusieurs trames CAN avant d'avoir eu le temps de toutes les traiter.

Bien évidemment, si la situation persiste, des trames seront perdues. Ceci résout cependant le cas classique d'un débit entrant dans la carte interface suffisant pour être traité,

mais dont les trames sont reçues par « à-coup », c'est-à-dire, quelques trames à la fois très rapprochées, puis un temps mort.

Si vraiment la carte interface ne suit pas (débit supérieur à 115200 bauds en entrée), il vous suffit de paramétrer les masques et filtres pour n'accepter que les commandes qui vous sont utiles. Nous verrons comment procéder.

La variable « flag » contient en réalité 8 flags utilisables individuellement. Dans cette version du logiciel, un seul est utilisé, sur le bit 0. Ce bit signale qu'on a reçu une trame, et qu'on doit donc l'analyser. Un « define » sur ce bit est effectué tout en bas, pour utiliser une représentation symbolique, selon mon habitude. Voyons la suite :

```

;=====
;                               VARIABLES BANQUE 0                               =
;=====
; zone de 160 octets, suite de l'access RAM
; -----
CBLOCK 0x60
    wreg_temp : 1          ; sauvegarde de W
    status_temp : 1        ; sauvegarde de STATUS
    fsr1l_tl : 1           ; sauvegarde de FSR1L interruptions L.P.
    fsr1h_tl : 1           ; sauvegarde de FSR1H interruptions L.P.
    fsr1l_th : 1           ; sauvegarde de FSR1L interruptions H.P.
    fsr1h_th : 1           ; sauvegarde de FSR1H interruptions H.P.

ENDC

```

Il s'agit de nos variables de sauvegarde pour nos registres. J'aurais pu les mettre en access bank, il y avait la place, mais étant donné qu'on n'utilise ces variables qu'à l'aide de « movff », autant prendre de bonnes habitudes.

Vous notez que FSR1 est sauvegardé deux fois. En effet, il est modifié aussi bien par la routine d'interruption haute que basse priorité. La première pouvant interrompre la seconde, il nous faut bel et bien deux emplacements de sauvegarde différents. Souvenez-vous que FSR1 est composé de deux registres : FSR1L et FSR1H, ce qui vous donne 4 variables de stockage.

```

;=====
;                               VARIABLES BANQUE 1                               =
;=====
; zone de 256 octets
; -----
CBLOCK 0x100
    bufincan : .256        ; buffer de réception CAN

ENDC

```

Dans cette zone, nous stockerons les trames CAN entrantes. Il s'agit donc de notre buffer d'entrée. Le reste de la mémoire RAM est inutilisé pour l'instant.

Ceci termine la zone des variables. Voyons maintenant le début du programme :

```

;=====
;                               PROGRAMME                               =
;=====

```

```

; vecteur de reset
; -----
ORG0x00
bra    init          ; sauter initialisation

; vecteur d'interruption haute priorité
; -----
ORG0x08          ; vecteur d'interruption haute priorité
braint    ; sauter pour éviter de recouvrir 0x18

```

Nous trouvons ici nos deux adresses de saut. Je dénomme (à tort) les adresses d'interruption comme étant les vecteurs d'interruption. En fait, les vecteurs, ce sont les adresses qui sont figées dans l'hardware du PIC. Mais bon, vu que c'est la dénomination que Microchip adopte, autant vous y habituer.

A titre d'exemple, si vous prenez un microprocesseur de la famille 680x0, vous avez une zone qui contient des emplacements dans lesquels vous inscrivez les adresses où se branchent les différentes interruptions (pas un « goto », mais uniquement une adresse). Ca, ce sont des vecteurs d'interruptions : ils indiquent où le processeur se branche en cas d'interruption. Ici, ces vecteurs sont figés par le hardware. 0x00 pour un reset, 0x08 pour une interruption haute priorité, et 0x18 pour une interruption basse priorité.

Notez ici ce dont je vous parlais dans le document de présentation : nous sommes obligé d'effectuer un branchement pour traiter les interruptions haute priorité, alors qu'il n'y en a pas besoin pour les interruptions basse priorité. J'aurais trouvé le contraire plus logique, les interruptions haute priorité ont plus de risques de devoir être critiques en temps que les autres.

Mais bon, à part réclamer chez Microchip pour qu'ils modifient leurs pics, il faudra bien nous contenter de cette manœuvre. Je doute du reste qu'ils vous prennent au sérieux. Si quelqu'un découvre une bonne raison pour justifier le sens actuel, qu'il me le fasse savoir.

Continuons maintenant avec l'étude de nos interruptions basse priorité :

```

;=====
;                                INTERRUPTIONS BASSE PRIORITE                                =
;=====
ORG0x18          ; adresse d'interruption basse priorité

; sauvegarde des registres
; -----
movff STATUS,status_temp ; sauver manuellement STATUS
movff WREG,wreg_temp      ; ainsi que WREG
movff FSR1L,fsr1l_tl      ; sauver FSR1L
movff FSR1H,fsr1h_tl      ; sauver FSR1H

```

Puisque nous utilisons les interruptions prioritaires, nous n'avons pas droit au mode « FAST » pour le retour des interruptions basse priorité. Autrement dit, il nous faudra sauver et restaurer manuellement les registres WREG (W) et STATUS.

Et BSR, me direz-vous ? Et bien, en fait, nous ne modifions pas BSR durant l'exécution du programme, une fois les interruptions en service, il n'y a donc aucune raison d'en effectuer la sauvegarde.

Nous modifions également FSR1, donc nous le sauvegardons. Si vous êtes pointilleux, vous remarquerez que nous n'utilisons nulle part FSR1 dans le programme principal, on pourrait donc se passer de cette sauvegarde. Mais, vu que nous ne manquons ni de temps ni d'espace, cette précaution permet de faciliter les futurs ajouts toujours possibles.

Voyons la suite :

```

; test interruption TX USART
; -----
btfss PIE1,TXIE      ; tester si interruption TX USART en service
bra    intl1         ; non, sauter
btfsc PIR1,TXIF      ; oui, tester si interruption TX USART
rcall  inttx         ; oui, traiter interrupt TX USART

```

Nous allons donc tester de quelle interruption il s'agit. Nous avons 4 interruptions programmées en basse priorité : l'interruption de transmission USART, l'interruption du timer0, chargé de valider la présence d'un temps mort, et les réception CAN sur les buffers 0 et 1.

```

; test débordement timer0
; -----
intl1
btfss INTCON,TMR0IE  ; tester si interrupts timer0 en service
bra    intl2         ; non, sauter
btfsc INTCON,TMR0IF  ; oui, tester si interrupt timer0
rcall  inttmr0       ; oui, traiter

; test des interruptions CAN
; -----
intl2
btfsc PIR3,RXB1IF    ; tester si réception CAN buffer1
rcall  intcanlrec     ; oui, traiter réception buffer 1
btfsc PIR3,RXB0IF    ; tester si réception CAN buffer0
rcall  intcan0rec     ; oui, traiter réception buffer 0

```

Ces tests n'amènent aucun commentaire particulier. Notez simplement que les flags d'interruption ne sont pas resettés dans la routine de test, ils devront donc l'être dans la partie traitement de l'interruption concernée.

Il ne nous reste plus qu'à restaurer les registres sauvés :

```

; restauration des registres
; -----
intlrest
movff fsr1l_tl,FSR1L ; restaurer FSR1L
movff fsr1h_tl,FSR1H ; et FSR1H
movff wreg_temp,WREG ; restaurer WREG
movff status_temp,STATUS ; et STATUS
retfie                ; puis, retour d'interruption

```

ce qui est effectué de nouveau aisément à l'aide d'instructions « movff », qui, je le rappelle, ont l'avantage dans ce cas de ne pas modifier les bits du registre « STATUS ».

Voyons maintenant en quoi consiste notre routine d'interruption d'émission USART.

```

; *****

```

```

;                                INTERRUPTION EMISSION USART (L.P.)                                *
;*****
;-----
; nbsend contient le nombre d'octets restant à envoyer
; nbdje contient le nombre d'octets déjà envoyés
; tramesend contient la trame à envoyer
;-----

```

Notez que dans le titre, j'indique « L.P. » pour « Low Priority » (basse priorité) et H.P. pour... je vous laisse deviner. . Habituez-vous-y car je procède toujours de cette façon.

On travaille avec deux variables : « nbsend » indique le nombre d'octets restant à envoyer, et « nbdje » le nombre d'octets déjà envoyés. La trame complète à envoyer commence à l'adresse « tramesend ».

Cette interruption intervient dès que le registre TXREG est vide, ce qui permet d'y placer l'octet suivant.

Examinons le contenu du traitement de cette interruption :

```

inttx
                                ; pointer sur le bon octet
                                ; -----
lfsr   FSR1, tramesend          ; pointer sur premier octet de la trame
movf   nbdje, w                 ; charger nbre d'octets déjà envoyés

                                ; envoyer l'octet suivant
                                ; -----
movff  PLUSW1, TXREG            ; envoyer l'octet en RS232
incf   nbdje, f                ; pointer sur octet suivant

                                ; tester nombre d'octets restants
                                ; -----
decfsz nbsend, f               ; décrémente nombre d'octets restants
return                                ; pas dernier, fin du traitement
bcf    PIE1, TXIE              ; si, fin des interruptions émission USART
return                                ; fin d'interruption

```

La routine est très simple : FSR1 est initialisé pour pointer sur le début de la trame, et on charge le nombre d'octets déjà envoyés, qui servira d'index de déplacement.

Il suffit de placer l'octet pointé dans le registre d'émission USART, et d'indiquer qu'on a traité un octet supplémentaire.

Ensuite, on décrémente le nombre d'octets restants. S'il en reste encore ( $nbsend > 0$ ), on termine le traitement, s'il n'en reste plus, on stoppe l'autorisation d'interruption de la routine d'émission USART. Cette autorisation sera remise en service lors de l'envoi de la prochaine trame.

Notez que pour l'interruption réception USART, il n'y a pas de flag à resetter explicitement. L'effacement du flag s'effectue automatiquement par l'écriture dans le buffer d'émission USART (voir cours-PART2).

C'est tout pour la routine d'émission USART, difficile de faire plus simple : voyons notre seconde routine d'interruption basse priorité :

```

;*****
;                               INTERRUPTION TIMER 0 (L.P.)                               *
;*****
;-----
; Intervient si aucun octet n'est reçu en 205µs depuis la RS232
; Indique la fin de la réception de la trame courante
;-----
inttmr0
                ; arrêt du timer
                ; -----
        bcf INTCON,TMR0IE,0        ; fin des interruptions timer

```

Je rappelle que cette interruption intervient lorsqu'un octet n'a pas été reçu depuis 205 µs, ce qui indique la fin de la trame courante.

Pour cette interruption, on devrait resetter le flag d'interruption, ce qu'on ne fait pas ici. Pourquoi ? Simplement parce que cette interruption ne doit intervenir qu'une seule fois par trame reçue. On ne resette donc pas le flag d'interruption, ce qui est inutile, puisqu'en réalité on interdit toute nouvelle interruption du timer0. La remise en service de l'interruption (et le reset du flag) s'effectuera lors du début de la réception d'une nouvelle trameRS232.

```

                ; vérifier si réception pas bloquée
                ; -----
        btfss RCSTA,OERR            ; tester si overflow réception
        bra   inttmr02             ; non, ok
        bcf   RCSTA,CREN            ; oui, couper réception, effacer OERR
        bsf   RCSTA,CREN            ; relancer réception
        clrf  nbrec                 ; aucun octet valide reçu

```

Pour le cas où un problème engendrerait une surcharge du buffer de réception, la communication serait définitivement interrompue (voir cours-PART2). Pour éviter ceci, on vérifie dans cette interruption si une erreur d'overflow est survenue. Si oui, l'annuler consiste à couper et à remettre en service la réception USART, ce qui est fait ci-dessus.

Puisque la trame qui avait commencé à se charger est incorrecte, du fait de l'overflow, on signale qu'aucun octet reçu n'est valide. Remarquez que normalement cette portion de code ne sera jamais exécutée, d'autant plus que la réception USART s'effectue par interruption de haute priorité. Mais bon, c'est une règle de bonne pratique pour un matériel destiné à être utilisé en conditions réelles, et puis on a à la fois la place et le temps.

```

                ; traiter fin de réception
                ; -----
inttmr02
        bcf   LEDR                  ; éteindre LED réception
        bsf   TRAMEREC              ; signaler trame reçue
        bcf   RCSTA,CREN            ; et fin de réception
        return                      ; fin d'interruption timer

```

Cette partie est exécutée lorsqu'une trame a été reçue, et que le temps mort est donc écoulé. Il nous suffit alors de signaler au programme principal qu'une trame est à traiter, via le flag TRAMEREC. Celui-ci traitera la trame, qui ne dispose pas d'un buffer d'entrée.

Durant le traitement de la trame, on n'accepte aucune nouvelle trame en entrée RS232, ce qui est logique, car le protocole de communication prévoit que le PC ne peut envoyer la trame suivante avant d'avoir reçu un accusé de réception de sa précédente commande.

Il s'agit ici d'une procédure de sécurité : si votre PC envoie une commande avant d'avoir reçu confirmation de la précédente, les octets ne seront pas reçus. Le mécanisme de sécurité, basé sur l'identification de la commande et sur la longueur de la trame se chargera d'éliminer un fragment de trame éventuel, reçu après envoi de la confirmation. Ceci termine notre routine d'interruption du timer0.

Voyons nos routines d'interruption réception CAN :

```

;=====
;                                INTERRUPTION RECEPTION CAN BUFFER 1  (LP)                                =
;=====
;-----
; Réception d'une trame CAN dans le buffer 1
; On copie la trame dans le buffer CAN pour traitement ultérieur
; on réserve 13 octets par trame CAN, soit 19 trames mémorisées
; la gestion est assurée par 2 pointeurs circulaires ptrcr (ptr reçu) et
; ptrct (ptr traité). Si les 2 pointeurs pointent sur la même valeur, alors
; c'est que toutes les commandes sont traitées
; la structure est la suivante : SIDH,SIDL,EIDH,EIDL,DLC,DATA
;-----
intcanrec
                                ; initialiser pointeur
                                ; -----
lfsr   FSR1,bufincan           ; pointer sur zone RAM du buffer
movff  ptrcr,FSR1L             ; pointer sur emplacement actuel de sauvegarde

                                ; mettre la trame dans le buffer
                                ; -----
movff  RXB1SIDH,POSTINC1       ; sauvegarder SIDH (commande)
movff  RXB1SIDL,POSTINC1       ; sauvegarder SIDL (extension de commande)
movff  RXB1EIDH,POSTINC1       ; sauver paramètre 1 (EIDH)
movff  RXB1EIDL,POSTINC1       ; sauver paramètre 2 (EIDL)
movff  RXB1DLC,POSTINC1        ; sauvegarder RXB1DLC (request/nombre de data)
movff  RXB1D0,POSTINC1         ; sauver data 0
movff  RXB1D1,POSTINC1         ; sauver data 1
movff  RXB1D2,POSTINC1         ; sauver data 2
movff  RXB1D3,POSTINC1         ; sauver data 3
movff  RXB1D4,POSTINC1         ; sauver data 4
movff  RXB1D5,POSTINC1         ; sauver data 5
movff  RXB1D6,POSTINC1         ; sauver data 6
movff  RXB1D7,POSTINC1         ; sauver data 7

                                ; gérer le pointeur
                                ; -----
movlw  0x0D                    ; charger incrément pointeur
addwf  ptrcr,f                 ; pointer sur suivant
btfsc  STATUS,C                ; tester si débordement
clrf   ptrcr                   ; oui, retour au début

                                ; opérations finales
                                ; -----
bcf    PIR3,RXB1IF             ; effacer flag réception buffer 1
bcf    RXB1CON,RXFUL           ; libérer le buffer de réception
return                                ; et retour

```

```

;=====
;                                INTERRUPTION RECEPTION CAN BUFFER 0 (LP)                                =
;=====
;-----
; Réception d'une trame CAN dans le buffer 0
; On copie la trame dans le buffer CAN pour traitement ultérieur
; on réserve 13 octets par trame CAN, soit 19 trames mémorisées
; la gestion est assurée par 2 pointeurs circulaires ptrcr (ptr reçu) et
; ptrct (ptr traité). Si les 2 pointeurs pointent sur la même valeur, alors
; c'est que toutes les commandes sont traitées
; la structure est la suivante : SIDH,SIDL,EIDH,EIDL,DLC,DATA
;-----
intcan0rec
        ; initialiser pointeur
        ; -----
        lfsr   FSR1,bufincan      ; pointer sur zone RAM du buffer
        movff  ptrcr,FSR1L        ; pointer sur emplacement actuel de sauvegarde

        ; mettre la trame dans le buffer
        ; -----
        movff  RXB0SIDH,POSTINC1  ; sauvegarder SIDH (commande)
        movff  RXB0SIDL,POSTINC1  ; sauvegarder SIDL (extension de commande)
        movff  RXB0EIDH,POSTINC1  ; sauver paramètre 1 (EIDH)
        movff  RXB0EIDL,POSTINC1  ; sauver paramètre 2 (EIDL)
        movff  RXB0DLC,POSTINC1   ; sauvegarder RXB1DLC (request/nombre de data)
        movff  RXB0D0,POSTINC1    ; sauver data 0
        movff  RXB0D1,POSTINC1    ; sauver data 1
        movff  RXB0D2,POSTINC1    ; sauver data 2
        movff  RXB0D3,POSTINC1    ; sauver data 3
        movff  RXB0D4,POSTINC1    ; sauver data 4
        movff  RXB0D5,POSTINC1    ; sauver data 5
        movff  RXB0D6,POSTINC1    ; sauver data 6
        movff  RXB0D7,POSTINC1    ; sauver data 7

        ; gérer le pointeur
        ; -----
        movlw  0x0D               ; charger incrément pointeur
        addwf  ptrcr,f             ; pointer sur suivant
        btfsc  STATUS,C           ; tester si débordement
        clrf   ptrcr              ; oui, retour au début

        ; opérations finales
        ; -----
        bcf    PIR3,RXB0IF        ; effacer flag réception buffer 0
        bcf    RXB1CON,RXFUL      ; libérer le buffer de réception
        return                    ; et retour

```

Ces routines sont pratiquement identiques à celles que nous avons vues dans le fichier « base.asm » dans le document « présentation ». Je n’y reviendrai pas. Notez que la seule différence de taille est que, quelque soit le buffer de réception sur lequel la trame arrive, on envoie cette trame sur le même buffer d’entrée software.

Ceci s’explique aisément : la carte d’interface n’a aucun besoin de traiter le contenu de la trame CAN, elle se borne à la réenvoyer vers le PC. Dès lors, qu’elle arrive par l’un ou l’autre buffer n’a strictement aucune importance.

Nous en arrivons à notre unique interruption haute-priorité, à savoir la routine de réception USART :

```

;=====
;                                INTERRUPTIONS HAUTE PRIORITE                                =
;=====
;*****
;                                INTERRUPTION RECEPTION USART (H.P.)                                *
;*****
;-----
; on reçoit un octet en provenance du PC
; on sauve l'octet dans tramerec
; le nombre d'octets déjà reçus est dans nbrec
; La fin de réception USART est déterminée par un temps mort de 205µs, détecté
; par le timer0.
;-----

```

Vous voyez que cette interruption intervient lors de la réception d'un octet en provenance du PC. La variable « nbrec » permet de compter les octets reçus, tandis que la trame reçue est stockée dans tramerec.

Nous allons modifier le pointeur FSR1. Or, ce pointeur est également utilisé dans les routines basse priorité. Fort logiquement, nous allons donc sauvegarder ce pointeur, constitué, je vous le rappelle, de deux registres, FSR1H et FSR1L :

```

inth
movff FSR1L,fsr1l_temp    ; sauver FSR1L
movff FSR1H,fsr1h_temp    ; sauver FSR1H

```

Puis nous devons resetter le timer0. En effet, à chaque octet reçu, on resette le timer pour l'empêcher de déborder. C'est donc bien une absence de réception d'un octet par l'USART qui valide la fin de la trame :

```

; opérations préliminaires
; -----
clrf TMR0L                ; reset timer 0

```

Remarquez que le timer 0 sera utilisé en mode 8 bits. En effet, contrairement aux 16Fxxx, ce timer peut maintenant fonctionner sur 8 ou sur 16 bits. Fort logiquement, le timer est alors composé de 2 registres, TMR0L et TMR0H. En mode 8 bits, nous nous contentons de remettre TMR0L à 0. Voyons la suite :

```

movf nbrec,w              ; charger nombre d'octets déjà reçus
bnz inth1                 ; si ce n'est pas le premier, sauter
bsf LEDR                  ; si premier, allumer led de réception
bcf INTCON,TMR0IF         ; effacer flag timer0
bsf INTCON,TMR0IE         ; et mettre l'interruption timer0 en service

```

Nous chargeons le nombre d'octets déjà reçus, puisque nous allons en avoir besoin pour déterminer l'emplacement de sauvegarde de l'octet reçu. Nous en profitons pour tester s'il s'agit du premier reçu. Si oui, on allume la LED de réception, on resette le flag d'interruption du timer0, puis on met ce timer en service.

Notez qu'on aurait pu se passer du test, et effectuer ces opérations systématiquement, ça ne nuit en rien, mais bon, c'est plus élégant comme ceci.

Vous remarquez que c'est à partir de la réception du premier octet qu'on déclenche l'interruption du timer0. A partir de ce moment, tous les octets devront arriver dans un intervalle de 205 µs, sous peine de considérer la trame terminée.

Il nous faut maintenant placer l'octet reçu dans la zone de réception USART :

```

; sauver l'octet, pointer sur suivant
; -----
inth1
    lfsr    FSR1, tramerec      ; pointer sur le premier octet de la trame
    movff   RCREG, PLUSW1      ; sauver octet reçu au bon emplacement

```

Constatez la facilité d'utilisation des nouveaux modes d'adressage indirect. Le nombre d'octet précédemment chargé dans W permet de pointer sur le bon emplacement, après que le pointeur ait été initialisé sur le début de la zone à l'aide d'une instruction « lfsr ». Ecrivez le code correspondant pour un 16Fxxx pour vous en convaincre.

Passons à la gestion du nombre d'octets reçus :

```

incf    nbrec, f              ; incrémenter compteur d'octets
movlw   .34                  ; 34 octets maximum
cpfslnbrec                    ; tester si déjà reçu 34 octets
bcf     RCSTA, CREN           ; oui, fin de réception

```

Nous commençons simplement par incrémenter le nombre d'octets reçus. Ensuite, on vérifie si on a déjà reçu 34 octets, qui est la taille maximale autorisée pour une trame RS232. Si c'est le cas, par sécurité, on coupe la réception, l'interruption timer0 sera alors déclenchée 205µs plus tard, pour valider la fin de la trame.

Cette mesure est impérative pour éviter un écrasement de notre zone de data pour le cas où le PC enverrait des trames RS232 trop longues (erreur dans le logiciel de contrôle).

Il ne nous reste plus qu'à restaurer notre registre FSR1 :

```

; restaurer registres
; -----
movff   fsr1l_th, FSR1L      ; restaurer FSR1L
movff   fsr1h_th, FSR1H      ; restaurer FSR1H
retfie FAST                  ; et fin

```

La restauration se passe de commentaire, excepté le commentaire qui dit qu'on se passe de commentaire.

Vous constatez qu'on utilise le paramètre « FAST », qui nous permet d'éviter de devoir restaurer « STATUS » et « WREG ». « BSR » n'avait pas besoin, de toutes façons, d'être restauré.

Ceci termine nos routines d'interruption. Nous passons maintenant à notre programme principal, en commençant par les initialisations :

```

=====
;                                     INITIALISATIONS                               =
;
=====

```

```

;-----
; contient les initialisations exécutées lors d'un reset
; -----
init
    ; initialisation PORTS
    ; -----
    movlb 0x0F          ; pointer banque 15 (pour registres CAN)
    bsf   OUTRS          ; préparer niveau haut sur ligne RS
    movlw TRISCVAL       ; valeur pour TRISC
    movwf TRISC          ; dans registre direction

    movlw TRISBVAL       ; valeur pour TRISB
    movwf TRISB          ; dans registre de direction
    bsf   PORTB,2        ; ligne au repos si pas de CAN

```

Rien que du classique. Nous forcerons la ligne « OUTRS » à 1 dès la mise en service. Ceci empêche le MCP2551 de fonctionner avant que l'intégralité des initialisations soient terminées.

Dans le même ordre d'idées, on force la ligne RB2 (CANTX) à 1, ce qui correspond à un niveau CAN récessif. Ce sont des précautions. En réalité, ces lignes ne sont pas indispensables si vous analysez soigneusement la procédure d'initialisation.

Mais bon, si vous trouvez qu'il est important de gagner 0,2µs entre la mise sous tension de l'interface et le moment où l'interface est prête à travailler, n'hésitez pas à supprimer ces deux lignes.

Passons à l'USART :

```

    ; Initialisation USART
    ; -----
    movlw B'00100100'    ; transmission en service, asynchrone H.S.
    movwf TXSTA           ; dans registre de contrôle
    movlw B'10010000'    ; réception en service, port série en service
    movwf RCSTA           ; dans registre de contrôle
    movlw .21             ; pour 115200 bauds, à 1,3%
    movwf SPBRG           ; dans baud rate generator
    bcf   IPRI,TXIP       ; interruption transmission basse priorité
    bsf   PIE1,RCIE       ; interruption réception USART en service (H.P.)

```

Nous configurons le module USART pour une liaison asynchrone à 115200 bauds. L'erreur sera de 1,3%, ce qui ne posera aucun problème. Ce module générera une interruption basse priorité sur l'émission, et une interruption haute priorité sur la réception.

On se contente de préparer l'interruption pour la réception, et de donner un niveau basse priorité sur l'interruption de transmission (par défaut, les interruptions sont toutes en haute priorité). La mise en service des interruptions d'émission se fera dans le programme principal au moment opportun.

Passons au CAN

```

    ; initialiser CAN
    ; -----
    SETCONFIG             ; passer le CAN en mode configuration
    bsf   RXB0CON,RXM1    ; messages étendus uniquement dans buffer 0

```

```

    bsf    RXB1CON,RXM1      ; messages étendus uniquement, pour buffer 1
    bsf    RXB0CON,RXB0DBEN ; si buffer 0 plein, autorise écriture dans 1
    lfsr   FSR2,RXF0SIDH    ; pointer sur premier registre filtre CAN
initlbootl
    setf   POSTINC2          ; FF dans le filtre ou le masque
    btfss  FSR2L,5           ; tester si zone 0xF00 / 0xF1F terminée
    bra    initlbootl        ; non, registre suivant

    movlw  B'00000001'       ; Syncho = 1TQ : TQ = 2*2/Fosc = 0,1µs
    movwf  BRGCON1           ; dans registre de contrôle
    movlw  B'11111010'       ; phase segment 2 programmable
                                ; 3 samplings sur porte majoritaire
                                ; phase segment 1 = 8 TQ
                                ; propagation = 3 TQ
    movwf  BRGCON2           ; dans registre de contrôle
    movlw  B'00000111'       ; phase segment 2 = 8
                                ; durée d'un bit = syncho+prop+seg1+seg2
                                ; = 1+3+8+8 = 20. 1bit = 20*0,1µs = 2µs
                                ; débit can = 1 / Tbit = 1/2µs = 500Kbits/s
    movwf  BRGCON3           ; dans registre de contrôle

    bsf    CIOCON,ENDRHI     ; niveau récessif = 1 sur pin CANTX

    bcf    PIR3,RXB1IF       ; reset flag interrupt réception buffer1
    bcf    PIR3,RXB0IF       ; idem buffer 0
    bsf    PIE3,RXB1IE       ; autoriser interruptions réception buffer 1
    bsf    PIE3,RXB0IE       ; idem buffer 0
    bcf    IPR3,RXB0IP       ; interrupt buffer 0 L.P.
    bcf    IPR3,RXB1IP       ; interrupt buffer 1 L.P.

```

Rien de plus ici que ce que nous avons déjà vu dans le document « présentation ».

N'oublions pas d'initialiser notre timer 0 :

```

                                ; initialiser timer0
                                ; -----
    movlw  B'11000010'       ; timer0 en service, mode 8 bits, prédiviseur 8
    movwf  TOCON              ; dans registre de contrôle
    bcf    INTCON2,TMR0IP     ; interruption timer0 basse priorité

```

L'initialisation est un peu différente que pour un 16F. On utilisera comme prévu les interruptions basse priorité, et on utilisera le timer en mode 8 bits, comme sur les 16F. Le prédiviseur de 8 nous donne un temps de débordement de :  $0,1\mu s * 256 * 8 = 204,8\mu s$ , ce qui est bien le temps prévu.

On procède à l'initialisation de nos variables, qui consiste, ici, à effacer entièrement la banque 0.

```

                                ; effacer la RAM banque 0
                                ; -----
    lfsr   FSR0,0x00          ; pointer sur l'adresse 0
initl
    clrf   POSTINC0           ; effacer emplacement pointé, et incrément pointeur
    btfss  FSR0H,0           ; banque 0 terminée?
    bra    initl              ; non, emplacement suivant

```

Une grande partie de ces effacements est inutile, mais on a tout notre temps à la mise sous tension, et la procédure est plus simple que d'initialiser variable par variable. Ainsi, en cas d'ajout, on est certain de ne rien avoir oublié.

Il ne nous reste plus ensuite qu'à lancer les interruptions et le module CAN :

```

                ; initialiser interruptions
                ; -----
bsf    RCON,IPEN      ; mise en service des priorités interruptions
movlw  B'11000000'    ; interruptions basse et haute priorité en service
movwf  INTCON         ; dans registre d'interruption 1

SETNORMAL            ; lancer le module CAN

```

C'est terminé pour les initialisations. Voyons le programme principal :

```

;=====
;=====
;                               PROGRAMME PRINCIPAL                               =
;=====
;=====
;-----
; Dirige vers la sous-routine concernée en fonction des commandes à exécuter
;-----
main
    clrwdt                ; reset watchdog
    btfsc TRAMEREC        ; tester si trame RS232 reçue à analyser
    rcall traittrec       ; oui, traiter trame reçue

    btfsc TXSTA,TRMT      ; tester si buffer d'émission vide
    bcf  LEDE             ; oui, éteindre led émission

    movf ptrct,w          ; charger pointeur buffer CAN à traiter
    cpfseqptrcr           ; comparer avec buffer réception
    rcall traitcanin      ; pas identiques, une trame CAN à traiter

    bra  main             ; boucler

```

Comme vous pouvez le constater, il se limite à une boucle. A l'intérieur de cette boucle, on guette l'arrivée d'un des 2 événements asynchrones possibles :

- Une trame complète RS232 a-t-elle été reçue ?
- Une trame CAN a-t-elle été reçue ?

Si oui, on traite le cas correspondant. On profite également de cette boucle pour éteindre la led d'émission une fois que le buffer d'émission USART est vide.

Le programme principal étant terminé, examinons les événements reçus possibles, en commençant par la réception d'une trame CAN sur le bus CAN :

```

;=====
;                               TRAITER TRAME CAN RECUE                               =
;=====
;-----
; La trame CAN reçue se trouve dans bufincan.
; La trame est pointée par ptrct.
; la trame sera envoyée sur le port RS232, précédée de 2 octets :

```

```
; 0x70 (réception trame CAN) et longueur de la trame CAN
```

L'en-tête rappelle que la trame reçue se trouve dans le buffer, à l'emplacement pointé par ptrct. La trame ne devra pas être analysée, mais simplement renvoyée sur le port série vers le PC, précédée de l'octet 0x70 et d'un octet qui indique le nombre d'octets que comporte la trame CAN.

```
traitcanin
; initialisation pointeur
; -----
lfsr  FSR0,bufincan      ; pointer sur buffer réception CAN
movff ptrct,FSR0L        ; pointer sur la trame concernée
```

On commence donc par pointer sur l'octet pointé par ptrct. La procédure peut paraître curieuse, mais est logique. La première ligne initialise FSR0L et FSR0H pour pointer sur le premier octet du buffer. Or, ce buffer commence au début d'une banque, ce qui fait que FSR0L vaut 0.

Pour pointer sur le bon octet, il faut ajouter la valeur de ptrct à FSR0L. Or, ajouter ptrct à « 0 » équivaut tout simplement à placer ptrct dans FSR0L. Tout est question de logique et de facilité.

```
; attendre fin d'une émission RS232 en cours
; -----
traitcanin1
clrwdt      ; effacer watchdog
btfsc PIE1,TXIE ; tester si émission RS232 en cours
bra  traitcanin1 ; oui, attendre avant de modifier trame d'envoi
```

Puisque nous allons devoir préparer une trame à destination du PC, il faut nous assurer que cette zone n'est pas utilisée par l'envoi non terminé d'une trame précédente. Il nous suffit pour ceci de vérifier que les interruptions d'émission USART ne sont pas en service. Si c'est le cas, l'interruption USART n'aura plus besoin de lire dans la zone que nous allons utiliser.

Notez qu'il se peut fort bien que l'émission ne soit pas terminée, puisque le buffer d'émission USART peut ne pas être vide, mais ce n'est pas notre problème, l'important est que notre zone ne soit plus utilisée. De plus, en procédant de la sorte, on accélère le débit vers le PC, puisqu'on peut préparer une trame alors que la trame précédente n'a pas encore fini d'être émise.

```
; initialiser la trame
; -----
movlw 0x70      ; charger commande
movwf tramesend ; = premier octet de la trame
movlw 0x04      ; au moins 4 octets de donnée (ID)
movwf tramesend+1 ; dans longueur data RS232
```

On commence par placer les deux premiers octets de la trame à émettre, à savoir 0x70 suivi par le nombre d'octets qui suivent l'en-tête. Comme nous aurons au minimum 4 octets à envoyer (l'ID de la trame), on placera par défaut la valeur 0x04 comme second octet.

```
; traiter identificateur
; -----
```

```

movff POSTINC0, tramesend+2 ; copier registre SIDH
movff POSTINC0, tramesend+3 ; copier registre SIDL
movff POSTINC0, tramesend+4 ; copier registre EIDH
movff POSTINC0, tramesend+5 ; copier registre EIDL

```

Le transfert des 4 octets d'ID ne pose aucun problème. Il suffit de copier les 4 octets pointés par FSR0 dans leur emplacement respectif dans l'emplacement de destination. 4 octets ne justifient pas une boucle avec double adressage indirect, si vous n'êtes pas convaincus, essayez de le faire en moins de 4 lignes.

Se pose cependant un problème. En effet, si on analyse les registres CAN du PIC, SIDL (qui contient notre extension de commande) contient donc les 5 octets utilisés pour notre réseau DOMOCAN. Le problème, c'est que ces bits ne sont pas alignés à droite, mais répartis sur l'intégralité du registre de la façon suivante :

```

b7 : bit 20 de l'ID
b6 : bit 19 de l'ID
b5 : bit 18 de l'ID
b4 : toujours 0
b3 : 1 si ID étendu, 0 si ID sur 11 bits
b2 : toujours 0
b1 : bit 17 de l'ID
b0 : bit 16 de l'ID

```

Ceci peut sembler curieux, mais, si vous y regardez de plus près, vous constatez qu'en fait, les 3 bits de poids fort complètent les 3 bits de poids faible de l'ID de base (8 bits + 3 bits), alors que les 2 bits de poids faible complètent le poids fort d'un ID étendu (2 bits + 16 bits supplémentaires).

Mais bon, bref, tout ceci ne nous arrange pas. Nous, nous voulons un octet de 5 bits alignés à droite, qui représente notre extension de commande. Bref, de la forme :

```

b7 : 0
b6 : 0
b5 : 0
b4 : bit 20 de l'ID
b3 : bit 19 de l'ID
b2 : bit 18 de l'ID
b1 : bit 17 de l'ID
b0 : bit 16 de l'ID

```

En réalité, le bit 7 sera utilisé comme drapeau pour indiquer si on a affaire à une trame de type remote ou à une trame de type data. Comme notre DOMOCAN ne s'occupe que des trames DATA, ce bit sera toujours à 0. Notez cependant que votre interface permet de dialoguer avec un bus CAN utilisant les trames remotés, il est prévu pour.

Bref, tout ceci pour dire qu'il va nous falloir effectuer une petite transformation :

```

lfsr FSR2, tramesend+3 ; pointer sur registre SIDL
rcall sidl2ext          ; transformer valeurs SIDL en commandes extended

```

Une petite sous-routine va se charger de transformer l'octet pointé par FSR2 en partant du format du registre « SIDL » vers le format utilisé sur notre DOMOCAN. Notez qu'on aurait pu reporter cette transformation sur le logiciel PC, mais en ayant expérimenté les deux solutions, celle-ci est au final plus pratique.

Notez la disparition du bit pour les trames étendues, inutile puisque toutes les trames traitées seront étendues.

```

; traiter bit remote
; -----
btfss INDF0,RXRTR      ; tester si trame remote reçue
bra   traitcanin4      ; non, traiter les data
bsf   tramesend+3,7    ; oui, positionner bit 7 de l'extension de commande
bra   traitcanin3      ; et pas de data

```

Ici, nous traitons justement du cas des trames remotes : Si on reçoit une trame remote (indiqué dans le registre suivant du module CAN), alors on positionne le bit 7 de l'extension de commande. C'est une pure convention. Dans le cas d'une trame remote, il n'y a pas d'octet de data, par définition, ce qui explique le saut.

```

; traiter longueur des data
; -----
traitcanin4
movf  POSTINC0,w        ; charger nombre de data
bz    traitcanin3      ; aucune data, sauter
addwf tramesend+1,f    ; ajouter au nombre de data RS232
movwf local01         ; sauver dans compteur de boucles

```

Cet octet contient également le nombre d'octets de data présentes, pour le cas d'une trame de type « data », ce qui sera notre cas. Il suffira d'ajouter ce nombre à l'emplacement contenant le nombre d'octets présents (pour l'instant, 4). Cette même valeur servira de compteur de boucles pour la copie des octets de data.

Un utilisateur attentif aurait pu se dire qu'il est inutile de copier les trames dans la zone, et qu'on pourrait indiquer directement d'envoyer celles présente dans le buffer d'émission CAN.

Cependant, si vous tentez d'écrire le programme correspondant, il sera plus long et plus compliqué.

```

; traiter les data CAN
; -----
lfsr  FSR2,tramesend+6 ; pointer sur emplacement data dans trame
traitcanin2
movff POSTINC0,POSTINC2 ; transférer un octet de data
decfsz local01,f       ; décrémenter compteur de data
bra   traitcanin2      ; pas dernière, suivante

```

De nouveau une boucle simple, contenant une instruction très puissante, puisqu'elle transfère d'emplacement mémoire à emplacement mémoire deux valeurs pointées par deux registres d'index, et ensuite effectue l'incrémentation automatique des deux registres. Essayez d'écrire ça pour un 16F, vous verrez la différence.

Notre trame est maintenant complète, il ne nous reste plus qu'à l'envoyer :

```

; lancer l'émission RS232
; -----
traitcanin3
    movf   tramesend+1,w      ; charger nombre de data RS232
    addlw  0x02              ; ajouter 2 (commande + len)
    movwf  nbsend            ; dans nombre d'octets à envoyer
    SENDRS                  ; lancer émission RS232

```

Le nombre d'octets à envoyer est simplement le nombre d'octets de la trame CAN incrémenté des deux octets initiaux. La macro lance les interruptions USART. Dès que le buffer d'émission sera libre (immédiatement, ou après fin de l'émission de la trame précédente), une interruption sera générée et l'émission pourra commencer.

Il ne nous reste plus qu'à signaler que nous avons correctement terminé de traiter cette trame, ce qui revient simplement à déplacer notre pointeur vers le début de la trame suivante :

```

; gérer le pointeur
; -----
    movlw  0x0D              ; charger incrément pointeur trame traitée
    addwf  ptrct,f           ; pointer sur suivante
    btfsc  STATUS,C          ; tester si débordement
    clrf   ptrct             ; oui, retour au début
    return                          ; fin du traitement

```

S'il n'y a pas de nouvelle trame CAN présente dans le buffer d'entrée, alors le pointeur ptrct aura une valeur égale au pointeur ptrcr, et on ne traitera pas de nouvelle trame pour l'instant. Dans le cas contraire, le programme principal renverra de nouveau sur cette sous-routine à sa boucle suivante.

Le traitement d'une trame CAN étant terminé, il nous faut examiner la réception d'une trame RS232 en provenance du PC :

```

;=====
;                               TRAITER TRAME RS232 RECUE                               =
;=====
;-----
; La trame reçue se trouve dans tramerec
; le nombre d'octets reçus dans nbrec
; les 2 premiers octets contiennent la commande et le nombre de data
;-----
traittrec

```

Petit rappel habituel indiquant les emplacements principaux concernant notre traitement.

```

; vérifier longueur de trame
; -----
    movf   tramerec+1,w      ; charger nombre de data de la commande
    addlw  0x02              ; ajouter 2 pour commande et longueur
    cpfseqnbrec              ; comparer avec nombre d'octets reçus
    bra    traitrecend       ; pas identiques, trame incorrecte

```

Une trame RS232 correcte aura donc un nombre d'octets reçus égal à 2 (2 octets initiaux) incrémenté du nombre d'octets qui accompagnent cette commande. C'est ce que se charge de vérifier cette portion de code. Si les longueurs ne correspondent pas, on ignore la trame.

```

; attendre fin de précédente émission

```

```

; -----
clrwdt          ; effacer watchdog
btfss TXSTA,TRMT ; tester si émission en cours
bra    $-4      ; oui, attendre (évite les écrasements)

```

Toute commande valide RS232 reçue est accompagnée d'une réponse vers le PC. Avant de pouvoir répondre, il faut que l'émission de la trame précédente soit terminée, c'est cette vérification qui est faite ici.

Etant donné que pour des raisons de facilité et d'efficacité, nous allons dans ce qui suit écrire directement dans le buffer d'émission USART, on attend que l'émission soit complètement terminée avant de poursuivre.

Les plus tatillons pourront se dire qu'on pouvait faire ça plus tard, au dernier moment, pour gagner du temps. C'est effectivement vrai, mais ça imposait plusieurs répétitions, pour ne gagner que quelques  $\mu$ s, le temps d'émission d'un octet nécessitant pas moins de 860 cycles d'instruction pour notre PIC. On ne va donc pas chicaner pour essayer de gagner le temps d'un quart d'octet en émission, d'autant qu'il ne s'agit pas de trames CAN entrantes, mais de réponse à une commande envoyée.

Notez le « \$-4 » qui remonte à la ligne « clrwdt », chacune des 2 instructions étant composée de 1 mot, soit 2 octets.

Il nous reste maintenant à traiter chacune des commandes possibles reçues :

```

;=====
;                               traiter commande 0x50                               =
;=====
;-----
; Demande de passage du bus CAN en débit 500Kbits/s
; octets de données reçus : aucun
; réponse du pic : 0x50 0x00
;-----
traitrec50

```

Après un petit rappel de la signification et de la composition de la commande, examinons son traitement :

```

; tester commande 0x50
; -----
movlw 0x50          ; valeur commande 0x50
cpfseqtramerec      ; comparer avec commande reçue
bra    traitrec51    ; pas identique, sauter
tstfsztramerec+1    ; tester si longueur data = 0
bra    traitrecend    ; non, trame incorrecte

```

On vérifie s'il s'agit de la commande 0x50, sinon on passe à la commande suivante. La commande est une commande sans octet de data, on vérifie si c'est bien le cas, sinon la commande est ignorée.

Notez qu'on n'utilise pas ici de trames d'erreur, c'est inutile, chaque commande valide est accompagnée d'une réponse (écho).

On passe au traitement proprement dit :

```

; modifier débit CAN
; -----
SETCONFIG          ; passer en mode configuration
movlw B'00000001'  ; Syncho = 1TQ : TQ = 4/Fosc = 0,1µs
movwf BRGCON1      ; dans registre de contrôle
SETNORMAL          ; remettre CAN en service

```

C'est très simple, on modifie simplement le temps unitaire CAN, à savoir ici 0,1µs. Souvenez-vous qu'un bit CAN durera dans notre cas 20 temps unitaires, soit 2µs, ce qui correspond bien à un débit de 500 Kbits/s.

Ce registre n'étant accessible qu'en mode configuration, on passe le module CAN dans ce mode avant de procéder à la modification. Bien évidemment on le remet dans le mode de travail directement après.

Par défaut, l'interface est paramétrée à cette valeur, vous avez donc peu de chance d'avoir à utiliser cette commande.

```

; envoyer l'écho
; -----
movlw 0x50          ; charger écho
bra sendecho        ; et lancer

```

On charge ensuite la valeur 0x50, qui sera renvoyée, accompagnée d'une longueur de 0, vers le PC. La routine « sendecho » se chargera de cette émission.

C'est tout pour la commande 0x50. Passons à la commande 0x51 :

```

;=====
;                               traiter commande 0x51                               =
;=====
;-----
; Demande de passage du bus CAN en débit 100Kbits/s
; octets de données reçus : aucun
; réponse du pic : 0x51 0x00
;-----
traitrec51
; tester commande 0x51
; -----
movlw 0x51          ; valeur commande 0x51
cpfseqtramerec      ; comparer avec commande reçue
bra traitrec52      ; pas identique, sauter
tstfsztramerec+1    ; tester si longueur data = 0
bra traitrecend     ; non, trame incorrecte

; modifier débit CAN
; -----
SETCONFIG          ; passer en mode configuration
movlw B'00001001'  ; Syncho = 1TQ : TQ = 2*10/Fosc = 0,5µs
movwf BRGCON1      ; dans registre de contrôle
SETNORMAL          ; remettre CAN en service

; envoyer l'écho
; -----
movlw 0x51          ; charger écho

```

```
bra sendecho ; et lancer
```

Elle est strictement identique dans son traitement à la commande précédente. Le temps unitaire sera placé à 0,5µs, ce qui donne un débit de 100Kbits/s.

Ceci sera utile si vous avez décidé de faire travailler votre réseau DOMOCAN à 100 Kbits/s. N'oubliez cependant pas dans ce cas d'effectuer la commutation AVANT de connecter l'interface sur votre réseau.

Souvenez-vous, et c'est important, qu'il est INTERDIT d'utiliser sur le même réseau des périphériques travaillant à des vitesses différentes. Ceci est vrai même si les périphériques ne communiquent pas explicitement, car une partie du traitement est automatique et hardware (Acknowledge, trames ERROR...).

Je vous conseille, si vous avez un réseau à 100Kbits/s de changer simplement la valeur de BRGCON au niveau de la routine d'initialisation, ceci vous évitera bien des déboires.

Alors, vous allez me dire : « Mais pourquoi cette possibilité de commutation ? » Tout simplement parce que je travaille sur d'autres projets qui, eux, nécessitent une liaison à 100Kbits/s, et que placer cette possibilité dans le PIC m'évite de reprogrammer sans arrêt mon interface ou d'en réaliser deux. Comme j'ai prévu cette possibilité pour moi, je ne vais pas passer mon temps à la supprimer pour vous, on ne sait jamais.

La commande suivante est déjà plus utile :

```
;=====
;                               traiter commande 0x52                               =
;=====
;-----
; Demande du contenu des filtres et masques CAN courants.
; octets de données reçus : aucun
; réponse du pic : 0x52 0x20 (32 octets de données), suivis par les
; identificateurs b28 à b0 pour les filtres 0 à 6, puis les masques 0 et 1
;-----
traitrec52
```

Avec cette commande, on va demander au PIC de renvoyer le contenu de ses 6 filtres et de ses 2 masques. Ceci indique quelles trames vont être acceptées par le module CAN du PIC, comme nous le verrons plus loin.

Nous allons donc renvoyer ces 8 registres, codés chacun sur 4 octets (taille de l'ID), ce qui donne un total de 32 octets à renvoyer. Ajoutez à ça les 2 octets initiaux, ceci donne une taille totale de 34 octets, qui est la plus longue trame RS232 autorisée.

```
                ; tester commande 0x52
                ; -----
movlw 0x52      ; valeur commande 0x52
cpfseqtramerec  ; comparer avec commande reçue
bra traitrec53  ; pas identique, sauter
tstfsztramerec+1 ; tester si longueur data = 0
bra traitrecend ; non, trame incorrecte
```

De nouveau, on commence par vérifier numéro de commande et longueur.

```

; passer le CAN en mode config
; -----
SETCONFIG          ; passer en mode configuration

; initialiser début de trame
; -----
movlw 0x52          ; écho
movwf tramesend      ; premier octet à envoyer
movlw 0x20          ; 32 octets de données
movwf tramesend+1    ; second octet à envoyer
addlw 0x02          ; plus deux
movwf nbsend        ; égal nombre d'octets à envoyer

```

L'accès aux registres nécessite de passer en mode configuration, ce qui est fait. Ensuite, on prépare les deux octets d'en-tête et le nombre total d'octets à émettre.

On copie ensuite l'intégralité des 32 registres concernés, tels quels :

```

; copier registres bruts dans trame à envoyer
; -----
lfsr FSR0, tramesend+2 ; pointer sur premier emplacement de données
lfsr FSR2, RXF0SIDH    ; pointer sur premier registre filtre CAN
traitrec52a
movff POSTINC2, POSTINC0 ; copier un registre dans la trame
btfss FSR2L, 5          ; tester si zone 0xF00 / 0xF1F terminée
bra   traitrec52a       ; non, registre suivant

```

De nouveau, on fait appel à des instructions typiques du 18F. A ce stade, nous avons dans la trame les 32 registres. Seulement, chaque registre « SIDL » possède exactement la même structure que pour l'ID. Autrement dit, il nous faut également transformer ces registres en valeurs compatibles avec nos extensions de commande :

```

; convertir SIDL en commande extended
; -----
lfsr FSR2, tramesend+3 ; pointer sur SIDL filtre 0 à envoyer
movlw 0x08             ; pour 6 filtres et 2 masques
movwf local01          ; dans compteur de boucles
traitrec52b
rcall sidl2ext         ; convertir SIDL en commande étendue
movlw 0x04             ; distance entre 2 SIDL
addwf FSR2L, f         ; pointer sur suivant
decfsz local01, f      ; décrémente compteur de boucles
bra   traitrec52b      ; pas fini, suivant

```

6 filtres et 2 masques, ça fait 8 registres « SIDL » à transformer. Chacun comportant 4 octets, une distance de 4 octets sépare les différents « SIDL ». La boucle ci-dessus permet de mettre en conformité chacun des 8 octets concernés. De nouveau, le pointeur FSR2 indique l'octet à reformater.

La trame est prête et complète, ne reste qu'à l'envoyer :

```

; envoi du message
; -----
SENDRS          ; lancer les interruptions émission RS232

```

Rien de plus simple, il s'agit de notre macro habituelle. Ne reste alors qu'à remettre le module en mode normal et à terminer le traitement

```

; remettre CAN en service
; -----
SETNORMAL          ; CAN en service
bra   traitrecend   ; fin du traitement

```

La commande suivante effectue l'opération inverse, c'est-à-dire l'envoi des 6 filtres et 2 masques destinés à être écrits dans le module CAN du PIC.

Cette commande est indispensable. N'oubliez pas qu'à l'initialisation, les filtres et masques sont configurés pour n'accepter aucune trame CAN. Sans l'envoi d'une commande de ce type, aucune trame CAN n'arrivera à votre PC :

```

;=====
;                               traiter commande 0x53                               =
;=====
;-----
; Réception du contenu des filtres et masques CAN courants.
; octets de données reçus : 32 suivis par les
; identificateurs b28 à b0 pour les filtres 0 à 6, puis les masques 0 et 1
; réponse du pic : 0x52 0x00
;-----
traitrec53
; tester commande 0x53
; -----
movlw 0x53          ; valeur commande 0x53
cpfseqtramerec      ; comparer avec commande reçue
bra   traitrec54     ; pas identique, sauter
movlw 0x20          ; nombre de data prévues
cpfseqtramerec+1    ; comparer avec data reçues
bra   traitrecend    ; pas identique, trame incorrecte

```

Le début du traitement est identique, vérification du numéro de commande, puis vérification de la longueur de la trame. Vient ensuite le traitement proprement dit :

```

; passer le CAN en mode config
; -----
SETCONFIG          ; passer en mode configuration

; convertir les extended en valeurs SIDL
; -----
lfsr FSR0,tramerec+3 ; pointer sur SIDL filtre 0
movlw 0x08           ; pour 6 filtres et 2 masques
movwf local01        ; dans compteur de boucles
traitrec53a
rcall ext2sidl        ; convertir extended en SIDL
movlw 0x04           ; distance entre 2 SIDL
addwf FSR0L,f         ; pointer sur suivant
decfsz local01,f      ; décrémente compteur de boucles
bra   traitrec53a     ; pas fini, suivant

```

Ici, nous avons reçu du PC des registres « SIDL » en format « extension de commande ». Or, nous devons placer des valeurs dans nos registres CAN. Il nous faut donc effectuer l'opération inverse aux précédentes, à savoir remettre ces informations dans un format compatible avec le format des registres « SIDL ».

La procédure est identique, c'est une autre sous-routine qui se charge de la transformation, l'octet à modifier étant cette fois pointé par FSR0.

```

; configurer les registres
; -----
lfsr   FSR0,RXF0SIDH      ; premier registre à configurer
lfsr   FSR2,tramerec+2     ; première data reçue
traitrec53b
movff  POSTINC2,POSTINC0   ; transférer la data dans le registre
btfss  FSR0L,5             ; tester si zone 0xF00 / 0xF1F terminée
bra    traitrec53b         ; non, suivant

```

Il suffit ensuite simplement de copier les 32 registres reçus dans leur emplacement respectif, la procédure est simple.

Ne reste plus qu'à replacer le PIC en mode normal, puis à envoyer l'écho de confirmation de la commande :

```

; remettre le CAN en mode normal
; -----
SETNORMAL                ; CAN en service

; envoyer l'écho
; -----
movlw  0x53              ; charger écho
rcall  sendecho          ; envoyer écho

```

Ceci termine le traitement de cette instruction. Voyons maintenant la commande 0x54 :

```

;=====
;                               traiter commande 0x54                               =
;=====
;-----
; Annulation de l'émission en cours
; réponse : 0x54,0x02,TXB0CON,COMSTAT
;-----
traitrec54
; tester commande 0x54
; -----
movlw  0x54              ; valeur commande 0x54
cpfseq tramerec          ; comparer avec commande reçue
bra    traitrec60         ; pas identique, sauter
tstfsz tramerec+1        ; tester si longueur data = 0
bra    traitrecend        ; non, trame incorrecte

```

Cette commande stoppe l'émission CAN en cours. Elle surtout utile si vous rencontrez des problèmes, car elle revoie le contenu de deux registres qui sont une image de la qualité de la liaison CAN. En cas de problèmes répétitifs, l'examen de ces registres vous renseignera utilement sur l'état de votre liaison. L'état des bits sera explicitement décrit par le logiciel de contrôle.

Le traitement en lui-même se résume à ceci :

```

; stopper l'émission
; -----
bcf    TXB0CON,TXREQ    ; stopper émission

```

Ne nous reste qu'à envoyer la réponse :

```

; envoyer réponse
; -----
movlw  0x54                ; charger commande (écho)
movwf  tramesend            ; dans trame à envoyer
movlw  0x02                ; 2 data à envoyer
movwf  tramesend+1          ; dans longueur de data
movlw  0x04                ; en tout, 4 octets à envoyer
movwf  nbsend               ; dans compteur d'octets
movff  TXB0CON,tramesend+2; registre TXB0CON dans premier octet donnée
movff  COMSTAT,tramesend+3; registre COMSTAT dans second octet donnée
SENDRS                      ; lancer les interruptions émission RS232
bra    traitrecend          ; fin du traitement

```

Vous constatez qu'on répond par une trame 0x54 accompagnée de la valeur des deux registres. Tout ceci sera traité automatiquement par le logiciel PC.

Il nous reste à voir la principale commande, la seule qui vous ouvre l'accès vers le bus CAN, la commande 0x60 :

```

;=====
;                                     traiter commande 0x60                                     =
;=====
;-----
; Réception d'une trame CAN en provenance du PC pour envoi sur le bus CAN
; longueur de 4 à 12
; identificateur sur 4 octets alignés à droite. Le bit 7 du second octet
; indique s'il s'agit d'une trame request
; de 0 à 8 octets de données
; réponse écho 0x60 0x00 : trame envoyée
;                0x6F 0x00 : longueur incorrecte
; on envoie tout via le buffer 0, comme ça tout arrive dans l'ordre
;-----
traitrec60

; tester commande 0x60
; -----
movlw  0x60                ; valeur commande 0x60
cpfseqtramerec             ; comparer avec commande reçue
bra    traitrecend          ; pas identique, sauter

; vérification de la longueur
; -----
movlw  0x04                ; nombre de data minimum
cpfsgttramerec+1           ; tester si au moins 4 data
bra    traitrec60er         ; non, erreur
movlw  .13                 ; nombre de data maximum +1
cpfslttramerec+1           ; tester si moins de 13 octets
bra    traitrec60er         ; non, erreur

```

Le principe de traitement initial est toujours le même. Cette trame RS232 a une longueur variable, puisqu'une trame CAN aura une longueur comprise (pour nous) entre 4 et 12 octets.

```

; Attendre buffer d'émission libre

```

```

; -----
clrwdt          ; reset watchdog
btfsc TXB0CON,TXREQ ; tester si buffer 0 libre
bra    $-4      ; non, attendre

```

On attend que le buffer d'émission 0 CAN soit libre. Inutile d'en utiliser plusieurs, puisque le débit RS232 est inférieur au débit CAN, et, de plus, pratiquer de cette façon permet d'envoyer les trames dans l'ordre, ce qui facilite la gestion au niveau du PC.

```

; initialiser registres d'émission
; -----
traitrec60b
    clrf  TXB0DLC          ; effacer registre de contrôle
    btfsc tramrec+3,7      ; tester si remote demandé
    bsf   TXB0DLC,TXRTR    ; oui, positionne request
    lfsr  FSR0,tramrec+3    ; pointer sur extension de commande
    rcall ext2sidl         ; convertir ext en valeur SIDL

```

On initialise le registre « DLC » qui précise le nombre d'octets de data et s'il s'agit d'une trame remote ou data. Ici, on s'occupe tout d'abord de l'octet « remote ». Ensuite, on convertit notre extension de commande en valeur de registre « SIDL ». Ceci étant fait, on initialise nos 4 registres d'ID :

```

; remplir les identificateurs
; -----
movff tramrec+2,TXB0SIDH ; charger SIDH
movff tramrec+3,TXB0SIDL ; charger SIDL
movff tramrec+4,TXB0EIDH ; charger IEDH
movff tramrec+5,TXB0EIDL ; charger EIDL

```

Reste ensuite à copier dans les registres concernés nos éventuelles data :

```

; remplir les data
; -----
movlw .4          ; car 4 identificateurs
subwf tramrec+1,w ; garder nombre d'octets de data
bz    traitrec60c ; pas de data, sauter
movwf local01     ; sauver nombre de data
iorwf TXB0DLC,f   ; indiquer nombre de data CAN
lfsr  FSR0,TXB0D0 ; pointer sur registre data0
lfsr  FSR2,tramrec+6 ; pointer sur data0 reçue
traitrec60bl
    movff POSTINC2,POSTINC0 ; transférer une data
    decfsz local01,f        ; décrémente compteur d'octets
    bra   traitrec60bl      ; pas dernier, suivant

```

Remarquez qu'on en a profité au passage pour terminer l'initialisation de « DLC » en y plaçant le nombre effectif d'octets de data CAN.

Ne reste plus qu'à lancer l'émission CAN puis à envoyer l'écho de réponse :

```

; lancer l'émission
; -----
traitrec60c
    bsf   TXB0CON,TXREQ ; requête d'émission CAN
    movlw 0x60          ; pour écho OK
    bra   sendecho      ; envoyer écho

```

```

; envoyer écho erreur
; -----
traitrec60er
    movlw 0x6F          ; pour code d'erreur
    bra   sendecho      ; envoyer écho

```

C'est terminé pour le traitement des différentes commandes. Reste à examiner les routines dont nous avons parlé :

```

;=====
;                               fin du traitement de la commande reçue      =
;=====
;-----
; 2 points d'entrée
; sendecho envoie la réponse RS232 à la commande reçue, puis termine le
; traitement
; traitrecend termine simplement le traitement
;-----
sendecho
    movwf TXREG          ; directement dans buffer d'envoi
    clrf tramesend       ; nombre d'octets de données = 0
    movlw 0x01           ; un octet encore à envoyer (la longueur)
    movwf nbsend         ; dans compteur d'octets restants
    SENDRS              ; lancer les interruptions émission RS232

```

La routine « sendecho » place l'octet reçu dans « W » directement dans le registre d'émission USART. Ensuite, on indique que le second octet qu'il faudra envoyer sera 0x00, puis on indique qu'il reste un seul octet à envoyer et on lance l'émission USART.

```

traitrecend
    clrf nbrec           ; préparer réception nouvelle trame
    bsf   RCSTA,CREN     ; remettre réception en service
    bcf   TRAMEREC       ; acquitter action
    return              ; et retour

```

On termine simplement en effaçant le nombre d'octets reçus, en relançant la réception USART, puis en acquittant l'action.

Il ne nous reste plus que nos deux sous-routine de conversion de format :

```

;*****
;                               CONVERTIR SIDL EN EXTENSION DE COMMANDE      *
;*****
;-----
; convertit un registre SIDL en un octet d'extension de commande
; format de SIDL :
;   ID b20, ID b19, ID b18, 0, Extended bit, 0, ID b17, ID b16
; format d'une extension de commande :
;   0, 0, 0, ID b20, ID b19, ID b18, ID b17, ID b16
;
; L'octet à modifier est pointé par FSR2
;-----
sidl2ext
    movf  INDF2,w        ; charger octet à modifier
    andlw 0x03           ; garder b17 et b16
    movwf local02        ; sauver 2 derniers bits
    movlw 0xE0           ; préparer masque

```

```

andwf INDF2,f          ; garder b20,b19,b18
rrncf INDF2,f          ; 0,b20,b19,b18,0,0,0,0
rrncf INDF2,f          ; 0,0,b20,b19,b18,0,0,0
rrncf INDF2,f          ; 0,0,0,b20,b19,b18,0,0
movf local02,w         ; charger bits 17 et 16
iorwf INDF2,f          ; 0,0,0,b20,b19,b18,b17,b16
return                 ; et retour

```

Cette sous-routine n'appelle pas de commentaire particulier, il vous suffit de suivre le déroulement des opérations. La sous-routine inverse est la suivante :

```

;*****
;                                CONVERTIR EXTENSION DE COMMANDE EN SIDL      *
;*****
;-----
; Convertit un octet au format extension de commande en valeur de SIDL
; format d'une extension de commande :
;      0, 0, 0, ID b20, ID b19, ID b18, ID b17, ID b16
; format de SIDL :
;      ID b20, ID b19, ID b18, 0, Extended bit, 0, ID b17, ID b16
;
; Le bit extended sera mis à 1, puisqu'on travaille toujours avec des
; commandes étendues. Pour les masques, pour lesquels l'extended bit n'existe
; pas, ça n'a pas d'importance, ce bit est à lecture seule dans ces registres
;
; L'octet à modifier est pointé par FSR0
;-----
ext2sidl
    movf INDF0,w          ; charger octet à modifier
    andlw 0x03            ; garder b17 et b16
    movwf local02         ; sauver 2 derniers bits
    rlncf INDF0,f         ; décaler vers la gauche
    rlncf INDF0,f         ; décaler vers la gauche
    rlncf INDF0,f         ; décaler vers la gauche
    movlw 0xE0            ; préparer masque
    andwf INDF0,f         ; b20,b19,b18,0,0,0,0,0
    movf local02,w        ; charger bits 17 et 16
    iorwf INDF0,f         ; b20,b19,b18,0,0,0,b17,b16
    bsf INDF0,EXID        ; b20,b19,b18,0,1,0,b17,b16
    return                ; et retour
END

```

Cette routine effectue l'opération inverse. Notez que le bit EXID est positionné systématiquement, ce qui signifie simplement qu'on traite des trames de type « extended ».

Ceci clôture l'analyse de notre fichier source.

**Notes :**

## **5. Mise en service**

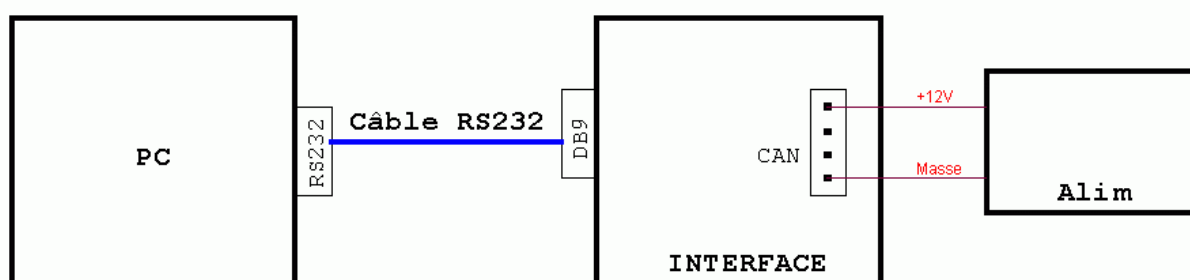
### **5.1 Connexion**

Il va s'agir ici de connecter votre interface à votre PC. Puisque vous réalisez ce projet dans l'ordre, vous n'avez à ce stade aucune carte DOMOCAN à votre disposition. Nous ne pourrions donc tester la partie CAN que lorsque nous aurons réalisé au moins une carte DOMOCAN. Les explications seront donc données dans la documentation de la carte « gradateur 16 ».

Néanmoins, nous pouvons déjà tester la bonne liaison avec le PC, ce qui garantit déjà qu'une bonne partie de votre interface fonctionne.

Pour ça, il vous faudra charger le logiciel de contrôle « Domogest » afin de procéder aux premiers tests. Les captures d'écrans ont été effectuées sur la révision 0.0.1 alpha. Etant donné qu'il s'agit d'un logiciel non terminé, il est fort probable que vous constatiez des différences avec la révision que vous chargerez vous-même. Ces différences ne devraient cependant pas perturber la compréhension des manipulations. Vous comprendrez qu'il m'est difficilement possible de modifier ce document à chaque modification du logiciel Domogest.

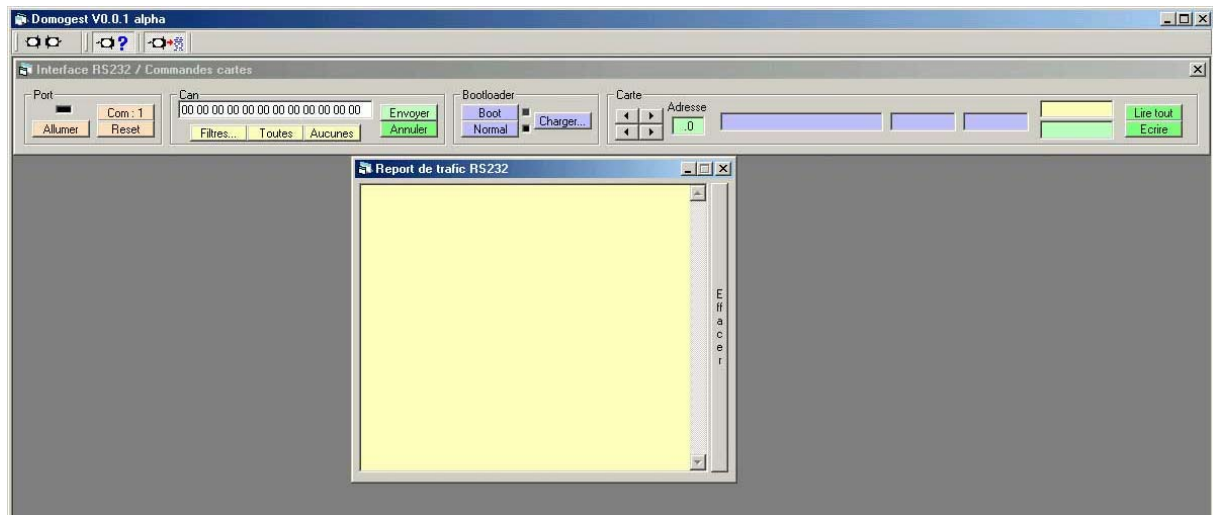
Une fois le logiciel téléchargé et installé, vous connectez votre carte d'interface à votre PC. Vous alimentez la carte en 12V non régulés à partir de la pin 1 et 4 du connecteur CAN de votre interface. Les pins CANH et CANL ne seront pas utilisées pour ces essais.



### **5.2 Premier lancement de Domogest**

Tout est prêt pour le premier lancement de Domogest. Tous les réglages que vous effectuerez dans le logiciel (position et taille des fenêtres, port etc.) sont mémorisés pour être restitués au prochain redémarrage. Si vous voulez revenir aux réglages par défaut, effacez le fichier « domogest.cfg » présent dans votre répertoire d'installation avant de relancer le programme.

Avec la révision 0.0.1 alpha, limitée aux fonctions du port série, nous obtenons l'écran suivant :



La barre d'outil est une «barre magique », elle est constituée de plusieurs barres d'outils, déplaçables horizontalement à l'aide de petites poignées, comme dans MPLAB6. Dans cette vue, vous en avez deux :

- Celle de gauche est la barre « commande » qui contient pour l'instant un seul bouton
- La seconde est la barre « vues » qui permet l'ouverture et la fermeture des différentes fenêtres

Si vous n'avez pas les deux fenêtres précédentes ouvertes, utilisez les boutons de la barre d'outils « vues » pour obtenir cet affichage. Les deux premiers boutons de cette barre permettent l'ouverture de :

- La fenêtre de manipulation des cartes DOMOCAN et de l'interface, que nous appellerons fenêtre « série »
- La fenêtre de report des trames reçues par l'interface, que nous appellerons fenêtre « report ».

Notez que le premier bouton de la barre d'outil « commandes » permet d'ouvrir et de fermer le port série sans devoir ouvrir la fenêtre « série ». Ce sera pratique une fois le numéro de port correctement configuré.

La fenêtre « report » permet d'obtenir une trace de toutes les trames entrantes, avec en plus un commentaire en clair et en français sur chaque trame reçue. Ceci pourra vous être utile en phase d'expérimentation. Lorsque vous utiliserez le logiciel sur une installation opérationnelle, cette fenêtre pourra rester fermée sans inconvénient.

Notez que vous disposez de la possibilité de minimiser la fenêtre, ce qui permet de mémoriser les derniers événements reçus sans encombrer l'écran.

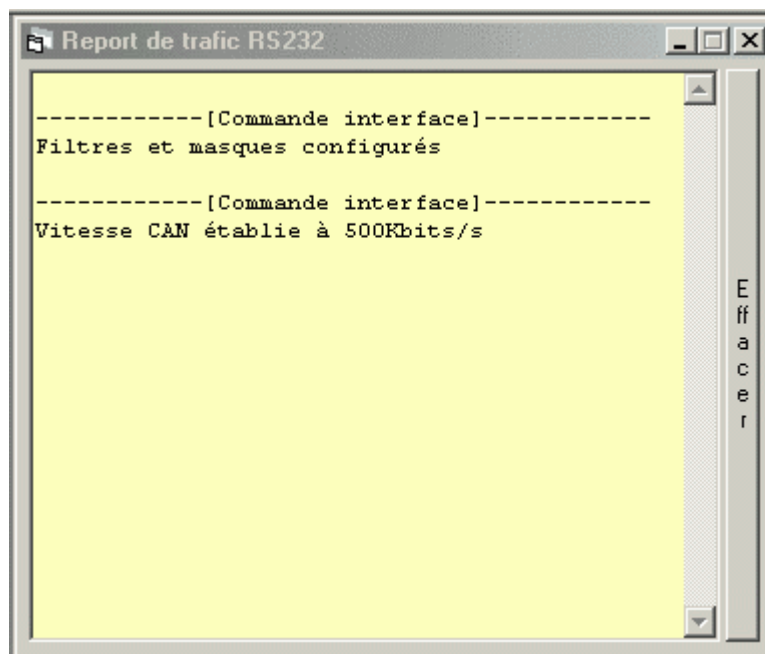
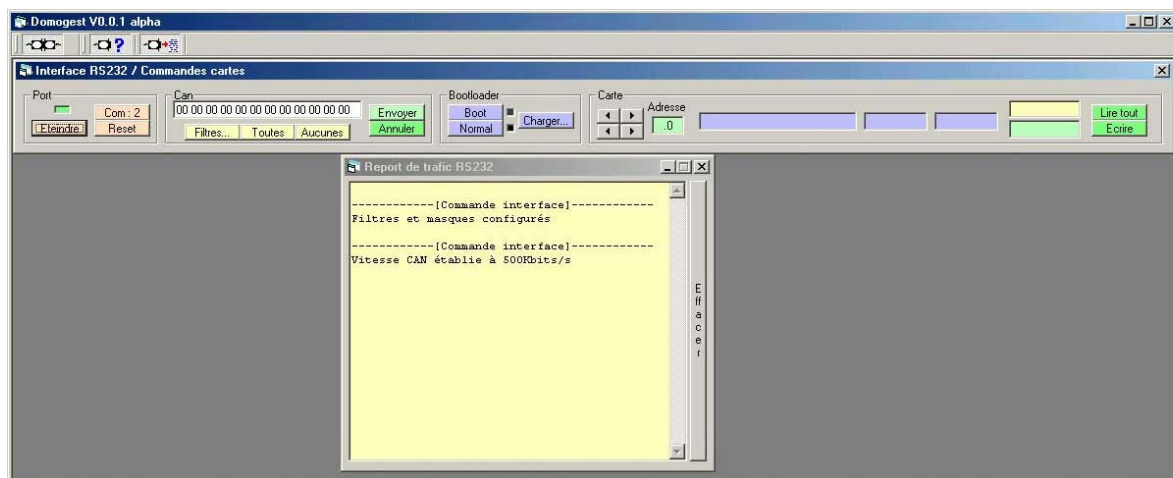
Notez que toute fenêtre ouverte traite les trames entrantes, et donc monopolise du temps CPU de votre PC. Pensez-y si vous avez une petite machine. J'utilise pour ma part un Athlon 1500XP+ sous Windows Millenium.

### 5.3 Vérification de l'interface

Il nous reste maintenant à vérifier que notre interface fonctionne. Commencez par choisir le numéro de votre port com, par clics successifs sur le bouton « com » de la frame « Port ». Une frame est un petit cadre interne à la fenêtre, son nom est toujours en haut et à gauche du cadre.

Chaque pression incrémente le numéro de port, de 1 à 8, ensuite on revient au numéro 1. Le numéro du port sélectionné s'inscrit directement dans le bouton. Ceci permet d'économiser de la place, car notre écran sera chargé lorsque nous mettrons les trames en service.

Une fois le port choisi, pressez sur la touche « Allumer », ou sur le premier bouton de la barre d'outils « commandes ».



Vous constatez que l'interface répond aux commandes PC, puisque les filtres et masques ont été configurés, et que la vitesse a été confirmée à 500 Kbits/s. Si ce n'est pas le cas, vérifiez votre connexion, votre port comm et votre interface.

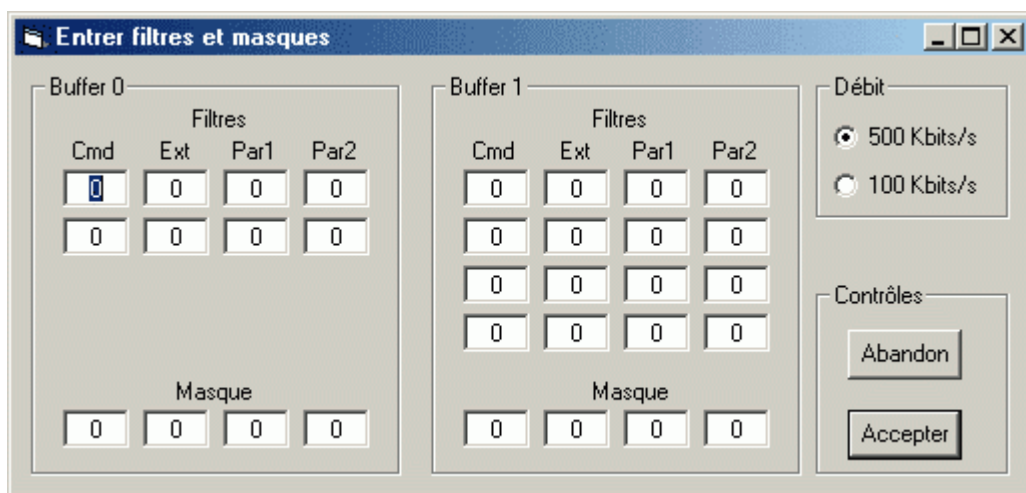
Le bouton <Reset> vous permet d'effectuer un reset hardware de votre interface, par action sur la pin MCLR.

Par défaut, les masques ont été placés avec 29 bits à « 0 », ce qui permet la réception de toutes les trames CAN entrantes.

Pour effectuer des tris de trames, vous disposez des 3 boutons situés au dessous dans la frame « Can ». Vous pouvez à l'aide de ces boutons :

- Configurer le fonctionnement de la partie CAN de l'interface : <filtres...>
- Configurer automatiquement les filtres et masques pour accepter toutes les trames : <toutes>
- Configurer automatiquement les filtres et masques pour refuser toutes les trames : <aucunes>

Tentons une configuration manuelle, en cliquant sur le bouton <filtres...> :



Dans cette fenêtre, vous pouvez modifier les filtres et les masques pour accepter ou refuser n'importe quel type de trame.

Notez que c'est ici que vous pouvez changer la vitesse de votre bus CAN si vous avez programmé vos cartes CAN avec un débit de 100Kbits/s. Attention, le choix d'une vitesse non conforme avec celle de vos cartes DOMOCAN risque de bloquer l'intégralité du bus en cas de connexion de l'interface : soyez prudents.

Le choix d'une valeur 0x00 dans les 4 octets d'un des masques suffit à autoriser toutes les trames, ce qui est le cas par défaut.

Soyez prudents si vous faites des essais, car vos réglages sont mémorisés et seront rappelés lors du prochain redémarrage. Je vous conseille après vos essais « à vide », d'effacer le fichier « domogest.cfg » après la sortie de votre programme.

L'ordre des octets est identique à celui des identificateurs pour chaque filtre et masque, c'est-à-dire la commande, puis l'extension de commande, suivis par le paramètre1, puis le paramètre2. En conséquence, le second octet est codé sur 5 bits, et ne peut atteindre qu'une valeur de 0x1F maximum.

Les octets de data n'interviennent pas dans les filtres et les masques, seul l'ID compte.

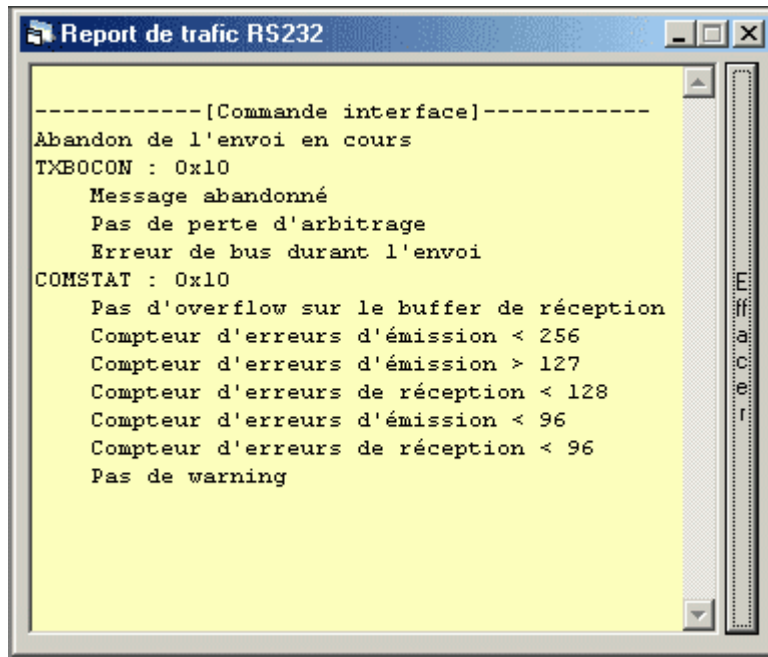
L'action sur le bouton <Accepter> valide vos choix et envoie ceux-ci directement vers l'interface. En cas d'abandon, les actions ne seront pas prises en compte.

Nous allons maintenant voir ce qui se passe si on tente d'envoyer une trame CAN alors que le bus n'est pas connecté :

Commencez par presser le bouton <Reset> pour être certain que vos précédentes manipulations n'ont pas bloqué le programme (je vais expliquer pourquoi).

Ensuite, vous pressez le bouton <Envoyer> de la frame « Can » une seule fois. Ceci provoque l'émission de la trame CAN située dans la zone texte à gauche du bouton.

Ensuite, vous pressez le bouton <Annuler>. Ce bouton provoque l'envoi d'une trame RS232 0x54. Souvenez-vous qu'en réponse, le PIC renvoie le contenu de 2 registres. J'ai inclus l'interprétation des bits de ces registres dans Domogest, ce qui vous donne :



Vous remarquez que le compteur d'erreurs d'émission est maintenant supérieur à 127, ce qui signifie que le module CAN a effectué au moins 128 tentatives d'émission de la trame, et qu'il s'est aperçu que personne ne répondait jamais.

Remarquez que le compteur d'erreurs d'émission est inférieur à 256, car le module s'est mis en suspend d'envoi. En effet, s'apercevant d'une anomalie, il se retire de l'émission du bus dans l'attente d'un retour à la normale, afin de ne pas perturber le bus en cas de problème hardware.

Tout ça automatiquement et sans aucune ligne de code dans notre programme assembleur. Notez l'énorme avantage par rapport à un bus du style PC.

Si vous avez la curiosité de regarder vos Leds durant la prochaine action sur <Envoyer>, vous remarquerez que chacune flash un petit coup. En effet, une trame RS232 d'envoi a été reçue par la carte, et une réponse a été envoyée (écho).

Ceci se passe de même aussi longtemps que vous faites suivre chaque envoi par une annulation. Si, par contre, vous effectuez un second envoi sans annuler, vous aurez la led de réception qui va s'allumer, mais pas celle d'émission. Pourquoi ?

Simplement parce que la commande précédente a placé la trame dans le buffer d'émission 0, la carte a répondu par l'écho 0x60, 0x00. Autrement dit, réception et émission RS232.

La seconde trame est reçue par la carte, donc allumage de la led de réception. Seulement, il est impossible à la routine de traitement de placer la trame dans le buffer 0, puisque celui-ci contient toujours la précédente trame qui n'a pas été envoyée du fait de l'absence du bus. La led émission ne s'allumera donc pas.

Si vous insistez encore, plus rien ne s'allume :

En effet, la troisième trame ne sera même pas reçue, puisque la routine est toujours bloquée dans l'attente de la libération du buffer d'émission. Votre interface est définitivement bloquée, jusqu'à ce que vous actionniez le bouton <Reset> qui effectuera un reset hardware.

Si tout s'est bien passé, vous avez de bonnes raisons de penser que votre interface est opérationnelle. Ne restera qu'à tester la partie « CAN », ce que nous ferons dès l'étude de notre carte gradateur 16 sorties. De ce fait, même si vous ne comptez pas réaliser cette carte, lisez le document s'y rapportant pour savoir comment manipuler la partie CAN de votre logiciel.

## **5.4 Rappels sur les filtres et les masques**

Pour éviter de relire le cours-part5, et surtout parce que ce cours n'est pas terminé à l'heure où j'écris ces lignes, je vais rappeler le fonctionnement des filtres et des masques CAN, et plus particulièrement sur notre système DOMOCAN.

La première chose à se souvenir, c'est que les ID utilisés dans notre système sont constitués de 29 bits répartis sur 4 octets. La répartition est la suivante :

Octet 1	: bits 28 à 21 de l'ID
Octet 2	: bits 20 à 16 de l'ID (donc seulement 5 bits)
Octet 3	: bits 15 à 8 de l'ID
Octet 4	: bits 7 à 0 de l'ID.

Chaque masque et chaque filtre est composé de 4 octets, dont chaque bit correspond à un bit de l'ID, et ce, dans le même ordre.

Les masques et les filtres permettent d'autoriser la réception d'une trame CAN donnée. Lors de la réception, chaque ID de la trame arrivant sur le module est comparée aux masques et filtres. Le résultat de la comparaison permettra de décider quelle trame est admise, et sur quel buffer elle le sera.

Il faut maintenant comprendre que notre PIC dispose de deux buffers de réception, le buffer0 et le buffer1. Chaque buffer dispose d'un seul masque et de plusieurs filtres, 2 pour le buffer0 et 4 pour le buffer1.

Buffer0 : masque0, filtre0 et filtre 1

Buffer1 : masque1, filtre2, filtre3, filtre4 et filtre5

La comparaison sera effectuée bit par bit, le bit0 de l'ID de la trame avec les bits0 des masques et filtres et ainsi de suite pour chaque bit.

La table de vérité est la suivante :

- Si le bitx du masque = 0, le bitx de l'ID de la trame sera toujours accepté
- Si le bitx du masque = 1, le bitx de l'ID sera accepté si et seulement si il est égal au bit du filtre en cours de comparaison

Bit du masque	Bit du filtre	Bit de l'ID	Résultat
0	Quelconque	Quelconque	Bit accepté
1	0	0	Bit accepté
1	1	0	Bit refusé
1	0	1	Bit refusé
1	1	1	Bit accepté

Ceci concerne un bit. Vous voyez que si le bit du masque vaut 0, on accepte d'office le bit de l'ID.

On acceptera une trame si chacun des bits de son ID a été accepté sur base de la table de vérité précédente.

La comparaison s'effectuera sur base du masque du buffer en cours de comparaison, et avec chacun des filtres du buffer à tout de rôle. Une fois la comparaison achevée, on recommence la comparaison avec les filtres et masques du buffer suivant (pour simplifier).

Pour être acceptée, une trame doit avoir tous ses bits acceptés avec le filtre en cours, on n'accepte pas une trame dont une partie des bits sont acceptés par un filtre, et le reste par un autre.

Prenons un exemple concret. Imaginons que nos filtres et masques soient configurés comme ceci :

**Entrer filtres et masques**

Buffer 0				Buffer 1			
Filtres				Filtres			
Cmd	Ext	Par1	Par2	Cmd	Ext	Par1	Par2
F0	0A	02	0	50	06	F8	0
38	0A	02	0	40	06	0	0
				38	06	04	0
				38	06	04	0

Masque				Masque			
F0	1F	FF	FF	FE	1F	FF	0

**Débit**  
☒ 500 Kbits/s  
☐ 100 Kbits/s

**Contrôles**  
 Abandon  
 Accepter

Voyons tout d'abord pour le buffer 0 :

Il faut commencer par regarder le masque. On voit qu'on prend tout en compte (bits à « 1 »), excepté les 4 octets de poids faible de notre ID, soit, d'après nos conventions, les 4 bits de poids faible de notre commande.

Ceci revient à dire qu'on accepte les commandes par groupe de 16. Tous les autres bits sont pris en considération. Si on regarde le filtre 0, on voit qu'on accepte :

- Les commandes 0xF0 à 0xFF (puisque les 4 bits de poids faible du masque sont à 0)
- Les extensions de commande de type « carte » (extension = 0x0A)
- Les commandes en provenance de la carte d'adresse 0x02
- Qui ont un numéro de réseau de 0x00

Pour plus de détails, voir le document « présentation ».

Le filtre 2 nous indique qu'on accepte :

- Les commandes 0x30 à 0x3F (masque 0xF0)
- Les commandes de type « carte »
- En provenance de l'adresse 0x02
- Avec un paramètre complémentaire = 0x00

En résumé, sera admise sur le buffer0 toute trame de type carte provenant de la carte d'adresse 0x02 et de numéro de réseau 0x00 qui contient une commande de 0xF0 à 0xFF ou de 0x30 à 0x3F.

Si on regarde le buffer1, on constate qu'on ignore les 3 derniers bits de la commande. On accepte donc les trames par groupes de 8 en ce qui concerne les numéros de commande. L'extension doit être conforme à celle indiquée, mais le paramètre2 est indifférent.

Le premier filtre (filtre2) nous indique qu'on accepte :

- Les commandes 0x50 à 0x57 (puisque seuls les 3 derniers bits sont ignorés)
- Les commandes de type « signal » (0x06)
- Dont le paramètre 1 vaut 0xF8

- Quel que soit le paramètre 2

Le second filtre (filtre3) nous indique qu'on accepte :

- Les commandes 0x40 à 0x47
- Les commandes de type « signal »
- Dont le paramètre 1 vaut 0x00
- Quel que soit le paramètre 2

Le filtre4 accepte :

- Les commandes de 0x38 à 0x3F
- Les commandes de type « signal »
- Dont le paramètre 1 vaut 0x04
- Quel que soit le paramètre 2

Le filtre5 est identique au filtre4. En effet, il n'existe pas de possibilité de ne pas utiliser un filtre. Aussi, si vous n'en avez pas besoin, le plus simple est de le mettre à la même valeur qu'un autre des filtres utilisés.

Si vous avez compris ce qui précède, vous avez tout compris sur le fonctionnement des filtres et des masques dans notre système DOMOCAN.

Sinon, ne vous inquiétez pas trop, vous pouvez utiliser le système sans modifier ces valeurs, la compréhension viendra à force de relecture.

**Notes :**

## **6. Utilisation du présent document**

Cet ouvrage est destiné à être lu après le document « présentation » disponible sur mon site.

Ne vous inquiétez pas si vous n'avez pas tout compris à la première lecture, vous verrez qu'au fur et à mesure de l'avancée du projet, en étant de plus en plus pratique, certaines notions vont devenir évidentes.

Communiquez à l'auteur (avec politesse) toute erreur constatée afin que la mise à jour puisse être effectuée dans l'intérêt de tous, si possible en utilisant le livre de report d'information présent sur la page de téléchargement de cet ouvrage ou par email.

Pour des raisons de facilité de maintenant, j'ai décidé que cet ouvrage serait disponible uniquement sur mon site : [www.abcelectronique.com/bigonoff](http://www.abcelectronique.com/bigonoff)

Aussi, si vous trouvez celui-ci ailleurs, merci de m'en avertir.

Bien entendu, j'autorise (et j'encourage) les webmasters à placer un lien vers le site, inutile d'en faire la demande. Je ferai de même en retour si la requête m'en est faite. Ainsi, j'espère toucher le maximum d'utilisateurs.

Le présent ouvrage peut être utilisé par tous, la modification et la distribution sont interdites sans le consentement écrit de l'auteur.

Tous les droits sur le contenu de cet ouvrage, et sur les programmes qui l'accompagnent demeurent propriété de l'auteur.

L'auteur ne pourra être tenu pour responsable d'aucune conséquence directe ou indirecte résultant de la lecture et de l'application de l'ouvrage ou des programmes.

Toute utilisation commerciale est interdite sans le consentement écrit de l'auteur. Tout extrait ou citation dans un but d'exemple doit être accompagné de la référence de l'ouvrage.

Si vous avez aimé cet ouvrage, si vous l'utilisez, ou si vous avez des critiques, merci de m'envoyer un petit mail, ou mieux, de poster un message sur mon site. Ceci me permettra de savoir si je dois ou non continuer cette aventure avec les parties suivantes.

Certains continuent à envoyer des messages sur mon ancienne adresse. Prenez connaissance de la bonne adresse, pour ne pas encombrer des adresses non concernées. Vous risquez de plus d'attendre longtemps votre réponse.

Sachez que je réponds toujours au courrier reçu, mais notez que :

- Je ne réalise pas les programmes de fin d'étude pour les étudiants (même en payant), c'est une demande qui revient toutes les semaines dans mon courrier. Tout d'abord je n'ai pas le temps, et ensuite je ne pense pas que ce soit un bon service. Enfin, pour faire un peu d'humour, si je donnais mes tarifs, ces étudiants risqueraient un infarctus.

- Je n'ai malheureusement pas le temps de debugger des programmes complets. Inutile donc de m'envoyer vos programmes avec un message du style « Ca ne fonctionne pas, vous pouvez me dire pourquoi ? ». En effet, je passe plus de 2 heures par jour pour répondre au courrier, si, en plus, je devais debugger, j'y passerais la journée. Vous comprenez bien que c'est impossible, pensez que vous n'êtes pas seul à poser des questions. Posez plutôt une question précise sur la partie qui vous semble inexacte.
- Si vous avez des applications personnelles, n'hésitez pas à les faire partager par tous. Pour ce faire, vous pouvez me les envoyer. Attention cependant, faites précéder votre envoi d'une demande, je vous redirigerai alors sur une autre boîte, celle-ci étant limitée à 512K.

Je remercie tous ceux qui m'ont soutenu tout au long de cette aventure, et qui se reconnaîtront. Nul doute que sans les nombreux encouragements reçus, ce livre n'aurait jamais vu le jour.

Merci au webmaster de [www.abcelectronique.com](http://www.abcelectronique.com), pour son hébergement gratuit.

Edition beta 1 terminée le 06/11/2003

### **Réalisation : Bigonoff**

Site : <http://www.abcelectronique.com/bigonoff>

Email : [bigocours@hotmail.com](mailto:bigocours@hotmail.com) (Attention BIGOCOURS PAR BIGONOFF)