LE SYSTEME DOMOTIQUE DOMOCAN

PAR BIGONOFF

INTERFACE CAN/RS232

Révision 5

1.Caractéristiques

1.1 Présentation générale

Avant de lire ce document, vous êtes invités à lire le document intitulé : « Présentation ».

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, domoboot.inc, et ccommunes.inc) utilisés pour les autres cartes.

Par contre, elle utilise le fichier de configuration Can « ParCan .inc » pour se paramétrer par défaut comme le reste de votre bus, et le fichier de configuration « configurations.inc » simplifiant le réglage des configurations du PIC®.

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. La description de ces trames étant commune à toutes les cartes d'interface Domocan, je vous renvoie au document « Communications d'interfaces » pour plus de détail. Ce point ne sera donc pas décrit dans ce document.

A partir de la révision logicielle 5.0, cette carte est entièrement paramétrable et permet de communiquer avec n'importe quel bus CAN sur ID 29 bits travaillant (mode extended) à une vitesse maximale de 1Mbits/s.

A partir de la révision logicielle 6.0, le PIC18F2680 a succédé à l'obsolète 18F258. Ceci a pour conséquence de permettre l'utilisation du mode 2 (FIFO), ainsi que d'augmenter significativement la taille de la mémoire tampon Can utilisée. Tout le logiciel a du être revu, et la compatibilité avec la nouvelle carte d'interface Can/Ethernet a nécessité la modification de toutes les communications USART.

Assurez-vous d'avoir la dernière révision du(des) logiciel(s) de contrôle et/ou de monitoring PC pour utiliser cette carte dans sa dernière révision, sans quoi des incompatibilités peuvent apparaître.

La vitesse de communication est limitée à 115200 bauds du fait que la plupart des ports Com des PC ne peuvent pas être initialisés au-delà de cette vitesse. Il vous est cependant loisible de modifier ce fait à la fois dans le logiciel Domogest et dans le logiciel PIC®, au niveau des sources des logiciels fournis.

La carte est compatible, moyennant une modification minime, avec les convertisseurs USB/RS232, les logiciels sont prévus en conséquence, l'explication est donnée dans ce document.

La carte RS232 travaille en standard 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, mais beaucoup moins en pratique, car les arrivées de trame CAN sont bufferisées dans la RAM du PIC®, et que chaque envoi d'une information du PC vers la carte se voit répondre par un accusé de réception, ce qui permet de synchroniser les émissions avec la vitesse de traitement du PIC®.

De plus cette carte utilise un filtre et un masque entièrement paramétrable, afin de ne laisser filtrer que les commandes qui nous intéressent, en cas de circonstances particulières. Ceci permet de réduire le débit des informations entrantes.

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

De toutes façons, souvenez-vous que cette vitesse ne pénalise en rien le débit de votre bus CAN, les cartes étant délocalisées et ne nécessitant pas de dialoguer avec le PC, excepté durant les phases de configuration.

1.2 Caractéristiques techniques

CAN :

Débit du bus (1,6)	**:** de 12.500 bits/s à 1Mbits/s (testé à 333Kbits/s)
Temps de traitement	**:** 2TQ (IPT = Information processing time)
Temps élémentaire TQ (1,2**)**	**:** de 0.05µs à 3.2µs
Synchronisation	**:** 1TQ fixé par le 18F2680 (Sync)
Temps de propagation (1,4)	**:** de 1 à 8 TQ (Prop)
Phase segment 1 (1,4)	**:** de 1 à 8 TQ (PS1)
Phase segment 2 (1,3,4)	**:** de 1 à 8 TQ. (PS2)
Resynchronisation (1,5)	**:** de 1 à 4 TQ (SJW = Synchronised Jump Width)
Echantillonnage (1)	**:** lecture unique ou triple lecture sur porte majoritaire
Buffer d'entrée	**:** 3068 octets, soit 236 trames CAN
Filtre et masque (1)	**:** 1 filtre et 1 masque paramétrables
Driver physique	**:** MCP2551 ou compatible
Trames acceptées	**:** étendues de type data, conformes à la norme CAN 2.0b

RS232 :

Débit du port	**:** 115200 bauds (modifiable dans fichier source)
Paramètres	**:** 1 start-bit, 8 bits de data, 1 stop-bit, pas de parité
Buffer d'entrée	**:** 80 octets, soit 5 trames
Buffer de sortie	**:** 80 octets, soit 5 trames
Contrôle du flux	**:** Aucun
Séparation des trames	**:** basée sur des trames de longueur fixe
Type de carte	**:** à microcontrôleur PIC18F2680 à 40Mhz
Driver physique RS232	**:** Max232 ou compatible

Divers :

Alimentation	**:** +12V, prise sur le bus DOMOCAN
Signalisations	**:** 1 Led trame RS232 entrante, 1 LED trame RS232 sortante

Notes ()

1) Paramétrable via la liaison RS232

2) Le temps élémentaire TQ (Time Quanta) est établi sous forme d'un prédiviseur. Ce prédiviseur peut varier sur les 18F2680 de 1 à 64. La formule est : TQ = TOsc * 2* prédiviseur, soit avec un quartz de 40Mhz :

- Temps minimal : (1/40.000.000) * 2 * 1 = 0.05µs
- Temps maximal : (1/40.000.000) * 2 * 64 = 3.2µs

3) PS2 doit à la fois être supérieur ou égal au temps de traitement interne (IPT = information processing time) fixé à 2TQ pour les 18F2680, et supérieur ou égal à PS1, bien que des essais montrent que l'interface fonctionne même si PS2 est inférieur à PS1.

4) La somme Sync + Prop +PS1 + PS2 doit être supérieure ou égale à 8 et le Nominal Bit Time, (Sync+Prop+PS1+PS2) * TQ, doit être supérieur ou égal à 1µs.

5) La resynchronisation (SJW) sera de préférence paramétrée sur la valeur «1 » si les cartes cibles du bus sont synchronisées par quartz. De toutes façons, je déconseille sous peine de déboires toute carte non cadencée par quartz sur votre bus Domocan.

6) Le débit est déterminé par la formule : Débit = 1 / ((Sync+Prop+PS1+PS2)*TQ), soit de 1/(25*3.2µs) = 12500 bits/s à 1/1µs = 1Mbits/s (voir remarque 4).

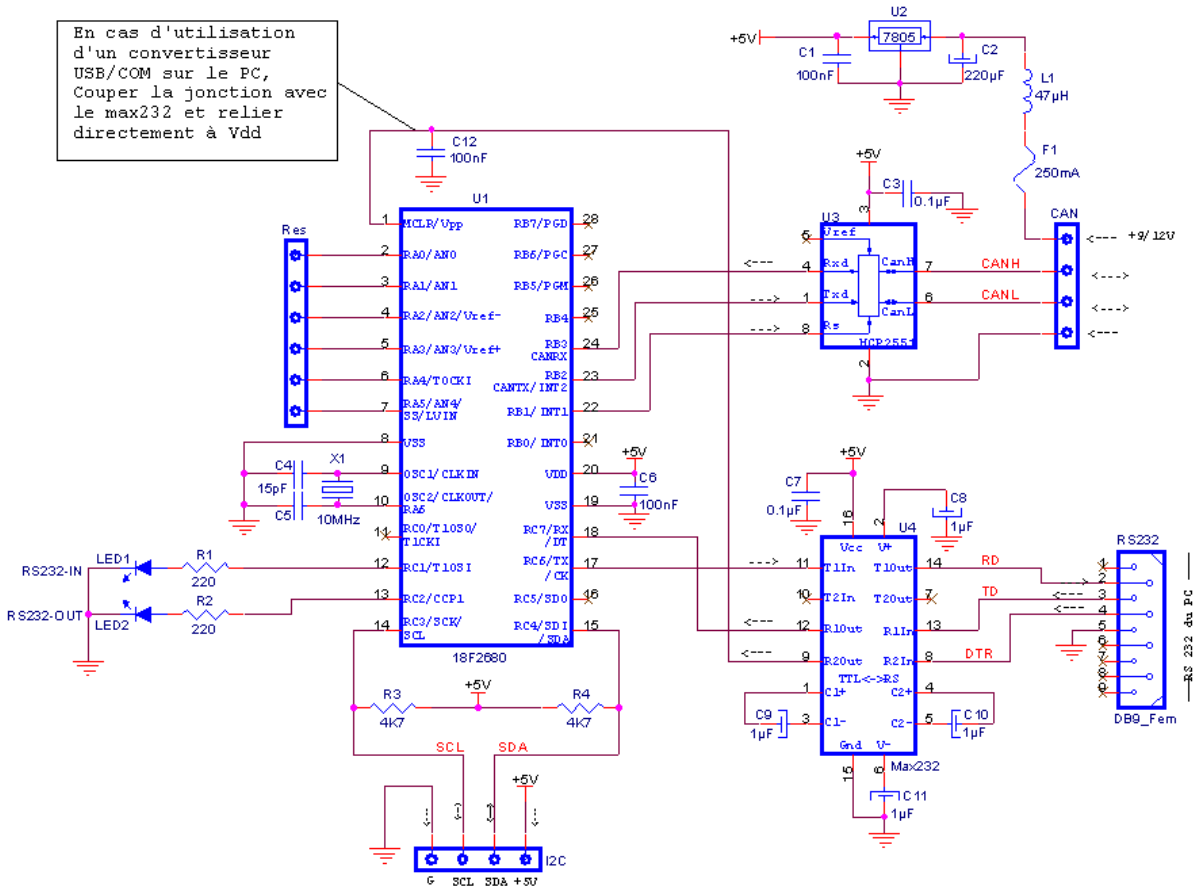
Les deux leds présentes sur la carte permettent de visualiser les trames au niveau du port RS232. Ceci signifie que seules seront signalées les trames CAN acceptées par le filtre et le masque de la carte. Le filtre et le masque sont mémorisés en eeprom de la carte.

Notes :

2. Réalisation pratique

2.1 Le schéma

Tout d'abord, voyons le schéma général :



Remarquez qu'à partir de la version 6.0 du logiciel, il est impératif d'utiliser un PIC® 18F2680. L'utilisation de ce PIC® nécessite une révision de Domogest au moins égale à 3.0.

Important : Si vous utilisez un convertisseur USB/Série au niveau de votre PC, coupez la liaison entre MCLR du PIC® et le max232, et reliez MCLR directement à Vdd (laissez le condensateur C12). De cette façon, le reset de l'interface ne sera plus effectif, mais l'interface fonctionnera cependant de façon correcte.

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. Du reste, c'est la seule méthode possible pour accéder aux 40Mhz autorisés.

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 femelle. Le contrôle de flux s'effectue par trames de longueur fixe et vérification temporelle. 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. Pour rappel, si vous utilisez un convertisseur USB/COM sur votre PC, il y a de fortes chances pour que ce signal ne soit pas géré correctement. Dans ce cas, il vous suffit de relier la pin MCLR du PIC® directement au +5V et votre interface fonctionnera sans problème.

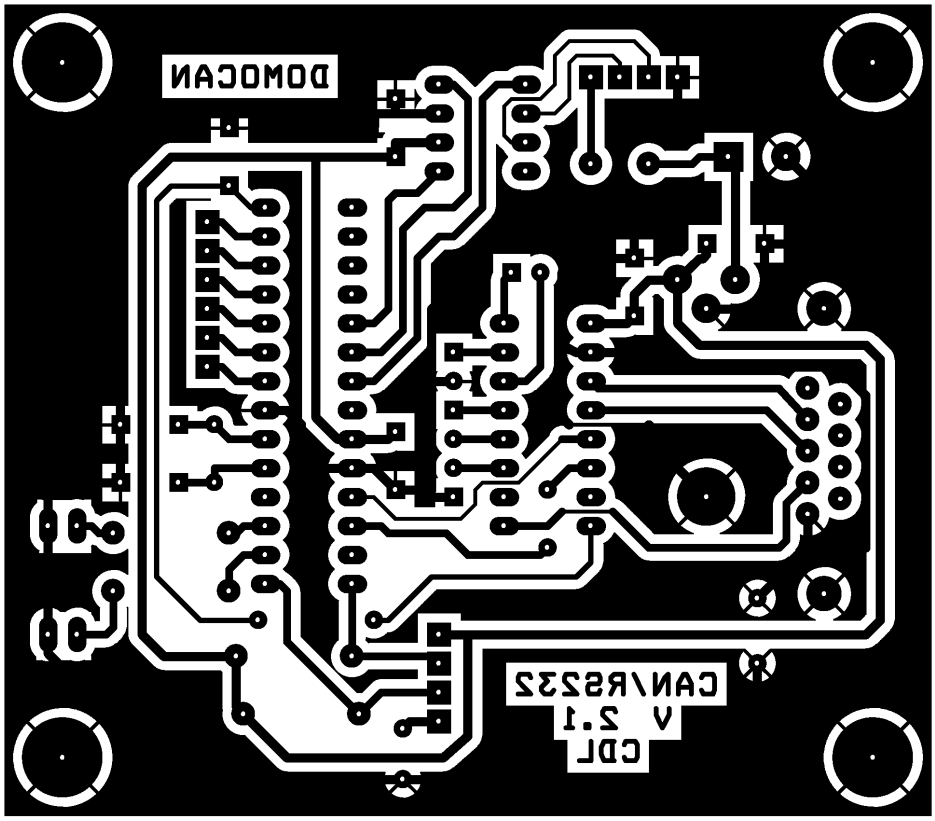
Le PIC® pilote également deux LEDs, chargées de visualiser le trafic RS232 entrant et sortant : du classique.

J'ai ajouté 1 connecteur et 2 résistances pour gérer un éventuel futur composant I^2C. Si vous voulez faire des économies vous pouvez vous en passer. Avec une modification du logiciel, l'interface pourrait également servir d'interface RS232 / I^2C ou Can / I^2C.

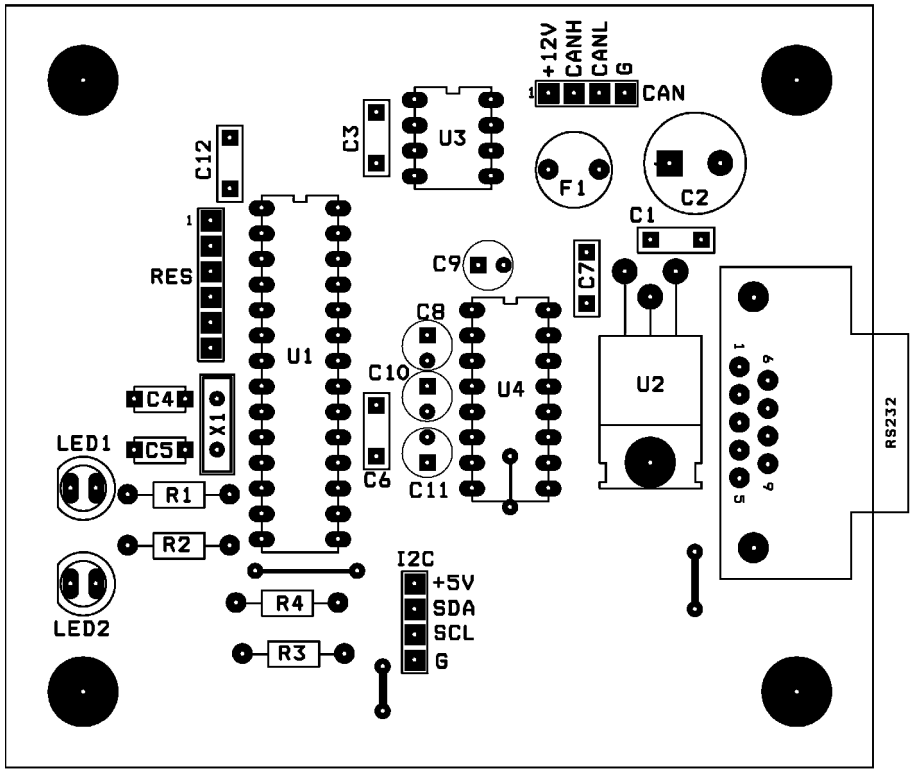
Un dernier connecteur donne accès à quelques pins de réserve, au cas où… De nouveau, vous n'êtes pas obligé de placer ce connecteur. Les dits connecteurs sont du reste de simples barres de contact sécables.

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 qui n'est pas à l'échelle pour que ce document soit complet.



Voici l'implantation des composants :

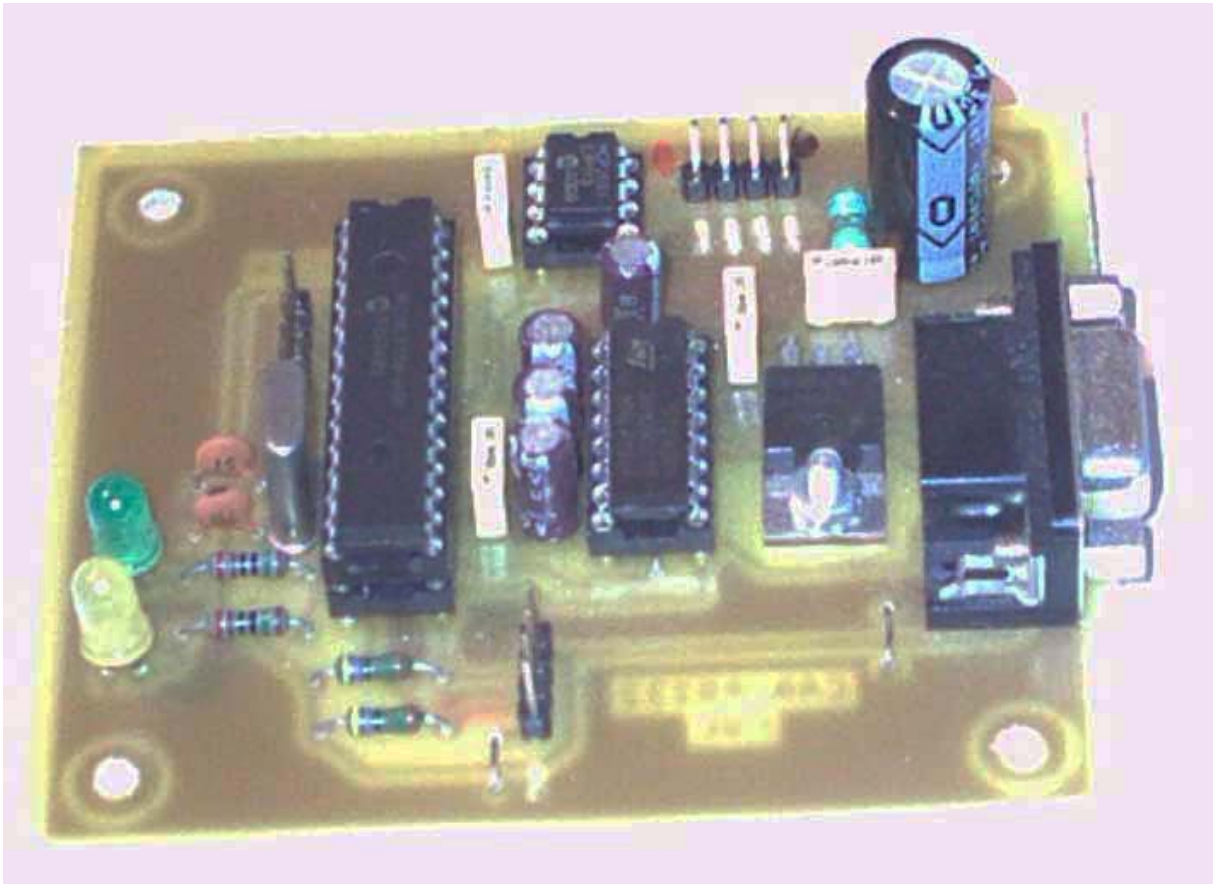


2.3 Réalisation pratique

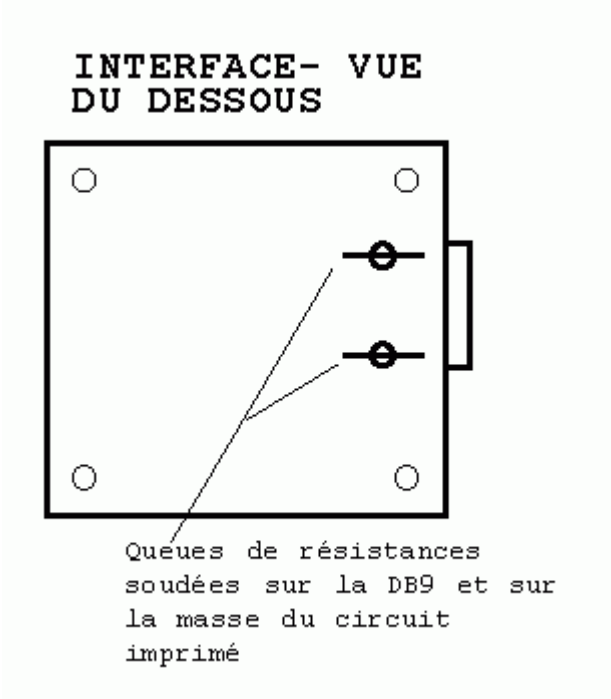
La réalisation pratique amène peu de commentaires. Soudez dans l'ordre habituel des composants. Le 7805 sera monté couché et vissé sur le circuit, il est inutile de lui prévoir un radiateur.

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 est toujours en service chez moi.



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.



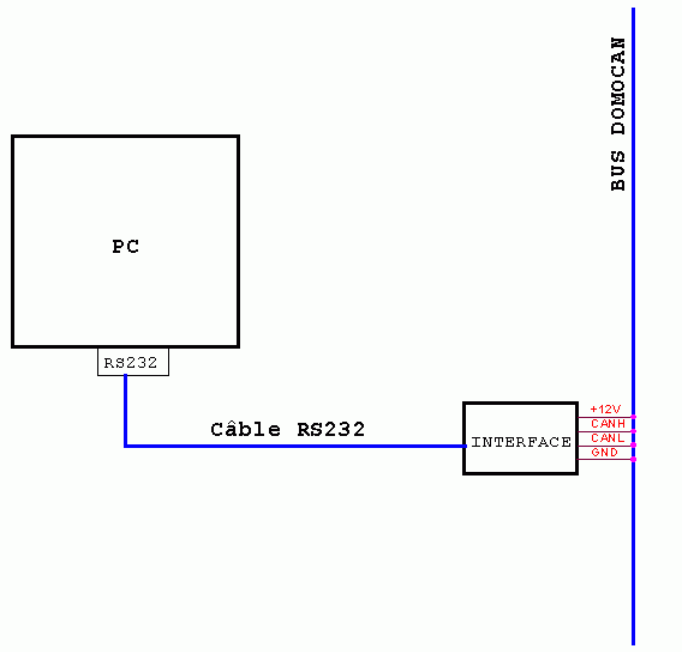
En procédant de la sorte, les pattes de maintien à clipser sont maintenant soudées, ce qui assure une meilleure mise à la masse et une meilleure rigidité mécanique dans le temps.

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 plus long, tout en respectant les limites de la norme RS232 (n'oubliez pas que vous travaillez quand même à 115.200 bauds, évitez les bricolages).

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 « PE » (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.

Notes :

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.

Les paramètres CAN sont paramétrables, de sorte que votre interface peut servir non seulement pour travailler sur Domocan, mais également sur la plupart des applications CAN. Paramétrez votre interface lors de la première utilisation AVANT de la connecter sur votre 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. 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, la trame d'interface. Chaque communication avec le PC s'effectue donc à l'aide de trames spécifiques qui indiquent la nature de leur contenu.

Comme signalé dans la partie de présentation, ce présent ouvrage n'aborde pas les différentes trames d'interface autorisées. Ceci s'explique par le fait qu'elles sont communes à d'autres cartes d'interface existantes ou à venir, et que je n'aime pas les redondances entre documents distincts.

Je vous renvoie donc au document « Communications d'interfaces » disponible sur mon site avec tous les autres documents Domocan.

3.3 L'Identificateur de PC

Si vous étudiez le document précité, vous y verrez la notion d'identificateur PC (PCID). Par convention, cette carte utilise un PCID fixé à 0xFA au niveau des logiciels d'accès (Domogest). Mais cette carte réagit en fait à n'importe quel identificateur, et répond correctement en conséquence. Ceci vous permet d'utiliser, par des astuces (modules RS232/Ethernet par exemple), plusieurs PC sur la même carte, et ce, strictement sans aucun problème.

Par défaut cette carte a cependant été prévue pour l'utilisation sur un seul PC, une carte d'interface de type Can/Ethernet sera plus à même de gérer les PC multiples de par sa nature.

3.4 Sécurités

Cette carte vérifie l'intégrité de la trame d'interface reçue selon les modalités suivantes :

1) N'est acceptée qu'une trame de 16 octets reçue sans trou dans la réception supérieur à une durée de 205µs, ce qui correspond à 2,4 octets.

2) Toute trame dont le checksum est incorrect est simplement ignorée (pas d'envoi d'accusé de réception)

3) Toute trame dont l'octet de commande ne correspond pas à un octet de commande standard tel que décrit dans le document « Communications d'interfaces » est rejetée

4) Toute trame dont le contenu ne correspond pas à ce qui est attendu en fonction de l'octet de commande se voit répondre par une erreur spécifique, et n'est pas traitée plus avant.

Ces sécurités sont largement plus que suffisantes pour une liaison série de ce type. Dans la pratique, les seules erreurs que j'ai rencontrées sont des erreurs de programmation du logiciel de communication en phase de développement, qui pouvait envoyer dès lors des commandes incorrectes.

Je doute fortement qu'excepté une coupure de la connexion en cours d'échange (ce qui est prévu), une erreur soit générée entre le PC et la carte d'interface. Mais, en tout état de cause, cette erreur serait gérée.

Au niveau du PC, pour toute trame d'interface envoyée vers la carte, le PC reçoit un accusé de réception permettant d'identifier la commande réellement reçue, son propre identificateur, ainsi qu'un statut d'exécution de la dite commande (OK, erreur, problème d'eeprom). Ceci assure une sécurité de communication théoriquement sans faille, le PC surveillant ce que reçoit la carte.

4. Analyse du logiciel PIC®

4.1 Le fichier CAN-RS232.asm

Je vais vous détailler ici le fonctionnement du logiciel, par l'étude de son fichier source. Ceci pourra vous servir pour vos réalisations personnelles.

Ce fichier utilise les fichiers d'inclusion « parcan.inc » et « configurations.inc

Remarquez que cette carte n'est pas bootloadable, c'est logique puisque la carte est reliée au PC par son interface RS232 et non par son interface CAN.

Par contre, si vous avez configuré corrrectement votre fichier « ParCan.inc » avec les paramètres de votre propre réseau Domocan, la carte sera déjà correctement configurée lorsque vous lancerez l'assemblage de votre fichier « CAN-RS232.HEX »

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                                                      *
;  --------------------                                                      *
;                                                                            *
; PIC requis : 18F2680                                                       *
;                                                                            *
;   Interface RS232 pour réseau domotique can "DOMOCAN" et autres bus CAN.   *
;   Permet l'interaction avec l'ensemble des cartes présentes                *
;   Permet le bootloading des cartes distantes                               *
;                                                                            *
;   Communication RS232 en 115200 bauds                                      *
;   Communication CAN entièrement paramétrable via la liaison RS232.         *
;                                                                            *
;*****************************************************************************
;                                                                            *
;    NOM               : Interface RS232 pour bus CAN                        *
;    Date création     : 31/05/2003                                          *
;    Date modification : 15/03/2008                                          *
;    Version           : 6.0                                                 *
;    Circuit           : Interface CAN/RS232                                 *
;    Auteur            : Bigonoff                                            *
;                                                                            *
;*****************************************************************************
;                                                                            *
;    Fichier requis: P18F2680.inc                                            *
;                    ParCAn.inc                                              *
;                    configurations.inc                                      *
;                                                                            *
;    Fréquence de travail : 40 MHz                                           *
;    Fréquence du quartz  : 10 MHz                                           *
;                                                                            *
;*****************************************************************************
;    Historique                                                              *
;    ----------                                                              *
;                                                                            *
;    V1.00 : le 10/06/2003 : Première version opérationnelle testée          *
```

```
;                                                                             *
;    V2.00 : 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                        *
;                                                                             *
;    V2.01 : le 14/12/2003 : correction d'un bug dans la gestion des buffers  *
;                                                                             *
;    V2.02 : le 22/01/2004 : limiter le déplacement à 127 (PLUSW)             *
;                                                                             *
;    V3.0 : le 06/05/2004 : remplacement du 18F248 par un 18F258 pour         *
;                           augmenter considérablement le buffer CAN          *
;                                                                             *
;    V3.1 : le 28/06/04   : Modification suite à la découverte d'un bug       *
;                           dans les pics. Lorsque le registre BSR pointe     *
;                           sur la banque 15. Modifications pour ne plus      *
;                           pointer sur la banque 15                          *
;                                                                             *
;    V3.2 : Le 27/12/04   : Prise en compte de 2 bugs découverts par          *
;                           Microchip : adresses 0x51 à 0x5D inutilisables    *
;                           et vérification de RXFUL sur interruption         *
;                           réception CAN                                     *
;                                                                             *
;    V3.2a: le 24/02/05   : Uniquement modifications de commentaires suite    *
;                           à la refonte de l'organisation des ID             *
;                                                                             *
;    V4.0 : le 19/05/05   : Augmentation du temps de propagation après        *
;                           sur câble pairé de 100m (câble ethernet).         *
;                           Diminution du phase segment 2 pour conserver la   *
;                           même vitesse.                                     *
;                                                                             *
;    V5.0 : le 21/03/06   : Paramétrage complet des paramètres CAN de         *
;                           l'interface via le port série. Modification       *
;                           complète des commandes 0x50 à 0x53                *
;                           Mémorisation des paramètres en eeprom             *
;                                                                             *
;    V6.0 : le 15/03/08   : Révision complète pour gestion par trames de      *
;                           longueur fixe pour compatibilité avec la carte    *
;                           d'interface EZL-50. Modification de toutes les    *
;                           commandes.                                        *
;                                                                             *
;*****************************************************************************
;  Pins utilisées :                                                           *
;  ----------------                                                           *
;                                                                             *
;    OSC1/CLKI        : Quartz 10 MHz                                         *
;    OSC2/CLKO/RA6    : Quartz 10 MHz                                         *
;                                                                             *
;    MCLR             : Entrée Reset PIC (active bas)                         *
;    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                                                *
```

```
; ---------------------------                                                 *
;                                                                             *
; Pour raison de compatibilité avec la carte CAN/Ethernet, on utilise des     *
; trames de longueur fixe : 16 octets.                                        *
;                                                                             *
; Octet 0     : Commande                                                      *
; Octet 1     : Toujours 0xFA pour la carte RS232, sauf pour la commande 0x70*
; Octet 2     : Nombre d'octets utiles de la trame (De octet3 à octet 14)     *
; Octets 3/14 : octets utiles complétés éventuellement par des dummy (0x00)   *
; Octet 15    : checksum = reste de la division par .256 des 15 octets        *
;                                                                             *
;  Pour la commande 0x70 (trame CAN), l'octet 1 vaut toujours 0xFF            *
;*****************************************************************************
; Commandes supportées Ethernet->PIC                                          *
; ----------------------------------                                          *
;                                                                             *
; 0x60 : Trame à envoyer sur le bus CAN                                       *
;        Octets de data : de 4 à 12 selon la trame CAN                        *
;         D0 : obligatoire : SIDH = destinataire                              *
;         D1 : obligatoire : SIDL = commande : si b7 = 1, trame remote        *
;         D2 : obligatoire : EIDH = cible                                     *
;         D3 : obligatoire : EIDL = paramètre                                 *
;         D4/D11 : facultatifs : DATA CAN D0 à D7                             *
;        Réponse : commande 0x50                                              *
;                                                                             *
; 0x41 : Abandon de la transmission CAN en cours                              *
;        Octet de data : 0                                                    *
;         D0/D11 : dummy bytes (0x00)                                         *
;        Réponse du pic : commande 0x51                                       *
;                                                                             *
; 0x42 : Modification du filtre et masque CAN (un seul filtre en service)     *
;        Octets de data : 8                                                   *
;         D0 : filtre 0 : bits 28 à 21 (SIDH = destinataire)                  *
;         D1 : filtre 0 : bits 20 à 16 (SIDL = commande)                      *
;         D2 : filtre 0 : bits 15 à  8 (EIDH = cible)                         *
;         D3 : filtre 0 : bits  7 à  0 (EIDL = paramètre)                     *
;         D4 : masque 0 : bits 28 à 21 (SIDH)                                 *
;         D5 : masque 0 : bits 20 à 16 (SIDL)                                 *
;         D6 : masque 0 : bits 15 à  8 (EIDH)                                 *
;         D7 : masque 0 : bits  7 à  0 (EIDL)                                 *
;         D8/D11 : dummy bytes (0x00)                                         *
;        Le pic passe en mode configuration, puis configure filtre et masque  *
;        Réponse du pic : commande 0x52                                       *
;                                                                             *
; 0x43 : Demande des valeurs du filtre et du masque                           *
;        Octet de data : 0                                                    *
;         D1/D11 : dummy byte (0x00)                                          *
;        Réponse du pic : 0x52                                                *
;                                                                             *
; 0x44 : Modification des paramètres CAN                                      *
;        Octets de data : 6                                                   *
;         D0 : TQ : Temps élémentaire de 1 à .64                              *
;         D1 : Temps de propagation de 1 à 8                                  *
;         D2 : Phase segment1 de 1 à 8                                        *
;         D3 : Phase segment2 de 1 à 8                                        *
;         D4 : Synchro Jump Width de 1 à 4                                    *
;         D5 : Sample (0 pour une lecture, 1 pour 3 lectures majoritaires)    *
;         D6/D11 : dummy bytes (0x00)                                         *
;        Les données sont mémorisées en EEPROM                                *
;        Réponse du pic : commande 0x54                                       *
;                                                                             *
```

```
; 0x45 : Demande des paramètres CAN                                          *
;        Octet de data : 0                                                   *
;        D0/D11 : dummy bytes (0x00)                                         *
;        Réponse du pic : commande 0x54                                      *
;                                                                            *
; Commandes supportées PIC->Ethernet                                         *
; ---------------------------------                                          *
;                                                                            *
; 0x70 : Envoi de la trame reçue du bus CAN                                  *
;        ID = 0xFF (broadcast software)                                      *
;        Octets de data : de 4 à 12 selon la trame CAN                       *
;         D0 : obligatoire : SIDH = destinataire                             *
;         D1 : obligatoire : SIDL = commande                                 *
;         D2 : obligatoire : EIDH = cible                                    *
;         D3 : obligatoire : EIDL = paramètre                                *
;         D4/D11 : facultatifs : DATA CAN D0 à D7                            *
;                                                                            *
; 0x50 : Accusé de réception d'une demande d'envoi vers CAN                  *
;        ID = ID de la machine qui a envoyé la commande 0x60                 *
;        Octet de data : 1                                                   *
;         D0 : Statut : 0 = OK, b0 = 1 -> erreur de commande                 *
;         D1/D11 : dummy bytes (0x00)                                        *
;                                                                            *
; 0x51 : Envoi des registres de statut de communication                      *
;        ID = ID de la machine qui a envoyé la commande 0x41                 *
;        Octets de data : 3                                                  *
;         D0 : TXB0CON                                                       *
;         D1 : COMSTAT                                                       *
;         D2 : Statut : 0 = OK, 1 = erreur de longueur de commande           *
;         D3/D11 : dummy bytes (0x00)                                        *
;                                                                            *
; 0x52 : Valeurs du filtre et du masque utilisé                              *
;        ID = ID de la machine qui a envoyé la commande 0x42 ou 0x43         *
;        Octets de data : 9                                                  *
;         D0 : filtre 0 : bits 28 à 21 (SIDH = destinataire)                 *
;         D1 : filtre 0 : bits 20 à 16 (SIDL = commande)                     *
;         D2 : filtre 0 : bits 15 à  8 (EIDH = cible)                        *
;         D3 : filtre 0 : bits  7 à  0 (EIDL = paramètre)                    *
;         D4 : masque 0 : bits 28 à 21 (SIDH)                                *
;         D5 : masque 0 : bits 20 à 16 (SIDL)                                *
;         D6 : masque 0 : bits 15 à  8 (EIDH)                                *
;         D7 : masque 0 : bits  7 à  0 (EIDL)                                *
;         D8 : Statut : b1=1 -> erreur eeprom b0 = 1 -> erreur de commande   *
;         D9/D11 : dummy bytes (0x00)                                        *
;                                                                            *
; 0x54 : Valeur des paramètres CAN actuels                                   *
;        ID = ID de la machine qui a envoyé la commande 0x44 ou 0x45         *
;        Octets de data : 7                                                  *
;         D0 : TQ : Temps élémentaire de 1 à .64                             *
;         D1 : Temps de propagation de 1 à 8                                 *
;         D2 : Phase segment1 de 1 à 8                                       *
;         D3 : Phase segment2 de 1 à 8                                       *
;         D4 : Synchro Jump Width de 1 à 4                                   *
;         D5 : Sample (0 pour une lecture, 1 pour 3 lectures majoritaires)   *
;         D6 : Statut : b1=1 -> erreur EEPROM, b0 = 1 -> erreur commande     *
;         D7/D11 : dummy bytes (0x00)                                        *
;                                                                            *
;****************************************************************************
```

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

```
;DEBUG EQU 0x01                ; enlever le ";" lors du debuggage
```

En phase de debuggage avec l'ICD2®, vous pouvez enlever le « ; » devant le « DEBUG ». Ainsi, l'assemblage se fera avec les options compatibles ICD2® (pas de watchdog par exemple), et les zones RAM réservées seront protégées.

N'oubliez pas de remettre le « ; » avant de votre assemblage définitif après debuggage.

```
TYPEPIC = 0x07                 ; 18F2680
```

Cette ligne indique les types de PIC® compatibles avec cette application, dans le standard Domocan. 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.

A partir de la révision 6.0, seuls les PIC18F2680 sont autorisés, quoi qu'il soit possible d'utiliser un 18F2580 (à vous dans ce cas de vérifier en conséquence et de faire les éventuelles modifications mineures, je n'ai pas testé).

```
 CONFIG WDTPS = 8              ; valeur postdiviseur watchdog (en ascii)
                               ; 1,2,4,8,16,32,64,128...32768
```

Le watchdog dépend étroitement du logiciel, aussi sa valeur n'est pas paramétrée dans le fichier « configurations.inc ». C'est pourquoi cette ligne est nécessaire. Une valeur de 8 fonctionne sans problème, vous pouvez tenter de diminuer pour plus de sécurité (je n'ai pas testé).

Suivent les inclusions des deux fichiers indispensables à l'assemblage correct du source :

```
   #include <ParCan.inc>           ; paramètres par défaut de Domocan
   #include <Configurations.inc>   ; configurations du pic
```

Si la carte d'interface est votre première carte Domocan, veillez à éditer le fichier « ParCan.inc » avec les paramètres de votre propre installation.

Suit ensuite une zone qui consiste en une série de vérifications sur la validité de votre fichier « ParCan.inc » :

```
;=============================================================================
;                      VERIFICATION DU FICHIER PARCAN.INC                     =
;=============================================================================

 IFNDEF TQ                     ; Si fichier paramètres Domocan pas trouvé
   ERROR "Fichier ParCan.inc non présent"
 ELSE
  IF (SJW ==0) || (SJW > 4)    ; vérification du SJW
   ERROR "Fichier ParCan.inc incorrect"
   MESSG "SJW non compris entre 1 et 4"
  ENDIF
  IF (TQ ==0) || (TQ > 64)     ; vérification du time quanta
   ERROR "Fichier ParCan.inc incorrect"
   MESSG "TQ non compris entre 1 et 64"
```

```
  ENDIF
  IF (SEG1 ==0) || (SEG1 > 8)   ; vérification du phase segment 1
   ERROR "Fichier ParCan.inc incorrect"
   MESSG "SEG1 non compris entre 1 et 8"
  ENDIF
  IF (PROP ==0) || (PROP > 8)   ; vérification du temps de propagation
   ERROR "Fichier ParCan.inc incorrect"
   MESSG "PROP non compris entre 1 et 8"
  ENDIF
  IF SAMPL > 1               ; vérification du sample
   ERROR "Fichier ParCan.inc incorrect"
   MESSG "SAMPL non compris entre 0 et 1"
  ENDIF
  IF (SEG2 ==0) || (SEG2 > 8)   ; vérification du phase segment 2
   ERROR "Fichier ParCan.inc incorrect"
   MESSG "SEG2 non compris entre 1 et 8"
  ENDIF
 ENDIF
```

Ceci constitue une vérification élémentaire pour savoir si vous n'avez pas placé dans ce fichier des valeurs incompatibles avec un fonctionnement normal du bus CAN. Cette partie ne produit aucun code dans le PIC®, il ne s'adresse qu'à l'assembleur MPASM®.

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)

VALPORTB  EQUB'00000110'        ; valeurs par défaut PORTB
DIRPORTB  EQUB'11111001'        ; direction PORTB
VALPORTC  EQUB'00011010'        ; valeurs par défaut PORTC
DIRPORTC  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.

```
#DEFINE   RXBxCON   RXB0CON      ; pour éviter les confusions quand on utilise
#DEFINE   RXBxSIDH  RXB0SIDH     ; l'access bank via WIN2/WIN0 de CANCON (mode 0)
#DEFINE   RXBxSIDL  RXB0SIDL     ; ou EWINx de ECANCON (mode 2)
#DEFINE   RXBxEIDH  RXB0EIDH
#DEFINE   RXBxEIDL  RXB0EIDL
#DEFINE   RXBxDLC   RXB0DLC
#DEFINE   RXBxD0    RXB0D0
#DEFINE   RXBxD1    RXB0D1
#DEFINE   RXBxD2    RXB0D2
#DEFINE   RXBxD3    RXB0D3
#DEFINE   RXBxD4    RXB0D4
#DEFINE   RXBxD5    RXB0D5
#DEFINE   RXBxD6    RXB0D6
#DEFINE   RXBxD7    RXB0D7
```

```
#DEFINE   TXBxCON   RXB0CON      ; pour éviter les confusions quand on utilise
#DEFINE   TXBxSIDH  RXB0SIDH     ; l'access bank via WIN2/WIN0 de CANCON (mode 0)
#DEFINE   TXBxSIDL  RXB0SIDL     ; ou EWINx de ECANCON (mode 2)
#DEFINE   TXBxEIDH  RXB0EIDH
#DEFINE   TXBxEIDL  RXB0EIDL
#DEFINE   TXBxDLC   RXB0DLC
#DEFINE   TXBxD0    RXB0D0
#DEFINE   TXBxD1    RXB0D1
#DEFINE   TXBxD2    RXB0D2
#DEFINE   TXBxD3    RXB0D3
#DEFINE   TXBxD4    RXB0D4
#DEFINE   TXBxD5    RXB0D5
#DEFINE   TXBxD6    RXB0D6
#DEFINE   TXBxD7    RXB0D7
```

Ce sont ici des alias pour l'accès aux différents registres CAN via l'access RAM. Pour rappel, ceci permet plus de clarté que d'écrire RXB0SIDH par exemple, alors qu'on accède en fait à TXB1SIDH. L'utilisation de TXBxSIDH permettra de fait de se rendre compte qu'on accède à un registre d'émission.

Analysons nos quelques macros :

```
;=============================================================================
;                              MACROS                                         =
;=============================================================================

                  ; passer CAN en configuration
                  ; --------------------------
SETCONFIG macro             ; passer le can en mode configuration
   movlw  B'10000000'       ; préparer demande configuration
   movwf  CANCON            ; et pointer sur buffer réception 0
   clrwdt                   ; effacer watchdog
   btfss  CANSTAT,OPMODE2   ; tester si passage effectué
   bra    $-(2*2)           ; non, attendre passage effectif
   endm
```

Cette macro permet de passer le module CAN en mode configuration, ce qui est indispensable pour accéder en écriture aux filtres, masques, et autres paramètres de fonctionnement.

Le datasheet précise qu'il est impératif d'attendre la confirmation du passage en mode configuration avant de procéder aux écritures, ce qui est réalisé par la boucle d'attente finale. Notez le « *2 » qui rappelle que chaque instruction 18F est codée sur un nombre d'octets multiple de 2.

```
                  ; passer CAN en mode 2 (FIFO)
                  ; --------------------------
SETMODE2 macro              ; passer le can en mode 2
   movlw  B'10000000'       ; demande de mode 2 FIFO
   movwf  ECANCON           ; dans registre de contrôle
   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
   bnz    $-(3*2)           ; pas commuté, attendre
   endm
```

Cette macro fait passer le module Can en mode 2 actif (FIFO). Les trames reçues sont gérées par une file d'attente automatique, ce mode n'existe que sur les PIC® munis d'un module ECAN, le module CAN ne pouvant fonctionner qu'en mode 0 (legacy). De nouveau une boucle d'attente est indispensable avant de pouvoir utiliser le module de façon opérationnelle sur le bus.

```
                    ; démarrer interruptions
                    ; ----------------------
STARTINT  macro           ; lancer les interruptions
   bsf    INTCON,GIEH     ; autoriser interruptions H.P.
   bsf    INTCON,GIEL     ; autoriser interruptions B.P.
   endm

                    ; couper interruptions
                    ; --------------------
STOPINT      macro        ; stopper les interruptions
   bcf    INTCON,GIEL     ; couper interruptions B.P.
   bcf    INTCON,GIEH     ; couper interruptions H.P.
   endm
```

Ces deux macros permettent respectivement de démarrer et de couper les interruptions dans le bon ordre, en cas d'interruptions hiérarchisées (avec priorités). Dans plusieurs errata Microchip®, on recommande cette façon de faire, donc, dans le doute, pourquoi s'en priver ? Je place donc ces deux macros systématiquement dans tous mes sources.

```
                    ; pointer sur buffers
                    ; -------------------
POINTRX0 macro            ; pointe sur buffer réception 0
   movlw  B'10010000'     ; configurer EWINx
   movwf  ECANCON         ; dans ECANCON
   endm
```

Cette macro est plus spéciale. En effet, il faut savoir que tous les registres CAN ne sont pas accessibles en access-bank. Pour éviter à l'utilisateur de devoir changer de banque pour les registres CAN les plus utilisés (buffers d'émission et de réception, particulièrement), Microchip® a utilisé l'astuce suivante :

Les adresses 0xF60 à 0xF6D (qui sont en access-bank) contiennent en principe les registres suivants (dans l'ordre croissant) : RXB0CON, RXB0SIDH, RXB0SIDL, RXB0EIDH, RXB0EIDL, RXB0DLC, RXB0D0, RXB0D1, RXB0D2, RXB0D3, RXB0D4, RXB0D5, RXB0D6, RXB0D7. Tous ces registres sont en fait ceux qui concernent le buffer de réception 0.

Ce buffer Can est le seul qui se situe en access-bank, à la mise sous tension, si vous accédez à ces 14 emplacements, vous accédez bel et bien au registre de réception 0.

Mais, et c'est là qu'est l'astuce (et la complication), vous disposez de la possibilité d'assigner ces emplacements au buffer physique réel de votre choix (dans une liste prédéfinie). Ceci s'effectue en modifiant certains bits d'un registre Can.

Or, ce registre et ces bits dépendent eux-mêmes du mode dans lequel vous avez configuré le module ECAN.

En mode 0 (legacy), qui est le mode par défaut à la mise sous tension, et également l'unique mode disponible dans les PIC® pourvus de modules CAN et non ECAN, comme c'est le cas pour un 18F258 par exemple, le registre qui contrôle l'accès est le registre CANCON.

Les bits WIN2 (b3) Win1 (b2) et Win0 (b1) indiquent à quel registre vous désirez accéder en réalité. Ces bits sont mis à 0 à la mise sous tension, et les choix sont les suivants :

```
111 = Receive Buffer 0
110 = Receive Buffer 0
101 = Receive Buffer 1
100 = Transmit Buffer 0
011 = Transmit Buffer 1
010 = Transmit Buffer 2
001 = Receive Buffer 0
000 = Receive Buffer 0
```

Remarquez qu'à la mise sous tension, les bits sont à 0, ce qui vous donne accès aux 13 registres du buffer RXB0.

Si vous ne touchez jamais à ces bits, chaque fois que vous accéderez à RXB0, vous accéderez bien au buffer de réception 0. Par contre, pour accéder aux autres buffers (émission et réception), étant donné qu'ils se trouvent hors de l'access-bank, vous devrez soit changer de banque (n'oubliez pas de gérer au niveau des interruptions), soit vous devrez utiliser l'adressage indirect.

Par contre, si vous utilisez ces bits, il faudra savoir que lorsque vous accéderez aux emplacements du buffer de réception 0, vous accéderez en fait au buffer pointé par les bits WIN2 à WIN0.

Petit exemple, soit ce code exécuté à la mise sous tension :

```
movf   RXB0CON,w       ; on charge le registre réel RXB0CON
bsf    CANCON,WIN1     ; WIN = 010
movf   RXB0CON,w       ; on charge le registre réel TXB2CON
```

Remarquez qu'il est très peu lisible de se rendre compte que RXB0CON ne concerne pas RXB0CON dans la réalité, le programme devient illisible. C'est pourquoi j'ai utilisé l'astuce des alias pour que le code soit plus clair. L'exemple précédent deviendra :

```
movf   RXB0CON,w       ; on charge le registre réel RXB0CON
bsf    CANCON,WIN1     ; WIN = 010
movf   TXBxCON,w       ; on charge le registre réel TXB2CON
```

De cette façon, on voit immédiatement en relisant le source qu'on accède à un registre d'émission dont on ignore le numéro, et donc on sait que ce numéro dépend d'autre chose (CANCON).

Dans notre programme, nous travaillons non pas en mode 0 (Legacy), mais en mode 2 (FIFO), mode qui nous donne plus de souplesse, mais qui n'est disponible que sur les PIC® munis de modules ECAN (18Fxx80 par exemple). Comme ce mode nécessite plus de registres, il y a plus de bits pour sélectionner le registre réel de destination.

La sélection n'est donc plus confiée aux 3 bits WIN du registre CANCON, mais aux 4 bits EWIN du registre ECANCON. Moyennant quoi le principe est exactement le même. EWIN4 est le bit 4, EWIN0 est le bit 0.

Voici les 32 possibilités de ces 4 bits :

```
00000 = Acceptance Filters 0, 1, 2 and BRGCON2, 3
00001 = Acceptance Filters 3, 4, 5 and BRGCON1, CIOCON
00010 = Acceptance Filter Masks, Error and Interrupt Control
00011 = Transmit Buffer 0
00100 = Transmit Buffer 1
00101 = Transmit Buffer 2
00110 = Acceptance Filters 6, 7, 8
00111 = Acceptance Filters 9, 10, 11
01000 = Acceptance Filters 12, 13, 14
01001 = Acceptance Filters 15
01010-01110 = Reserved
01111 = RXINT0, RXINT1
10000 = Receive Buffer 0
10001 = Receive Buffer 1
10010 = TX/RX Buffer 0
10011 = TX/RX Buffer 1
10100 = TX/RX Buffer 2
10101 = TX/RX Buffer 3
10110 = TX/RX Buffer 4
10111 = TX/RX Buffer 5
11000-11111 = Reserved
```

Dès lors, si vous reprenez notre macro

```
POINTRX0 macro           ; pointe sur buffer réception 0
   movlw B'10010000'     ; configurer EWINx
   movwf ECANCON         ; dans ECANCON
   endm
```

Vous constatez qu'en plaçant EWIN à 10000, vous accédez au registre de réception 0 (RXB0). Notez qu'il s'agit également de la valeur par défaut à la mise sous tension, donc, de nouveau, tant que vous ne manipulez pas ces bits explicitement, aucun risque de surprise.

La macro suivante, elle :

```
POINTTX0 macro              ; pointe sur buffer émission 0
   movlw B'10000011'        ; configurer EWINx
   movwf ECANCON            ; dans ECANCON
   endm
```

Place EWINx à 00011, et permet d'accéder au buffer d'émission 0 (TXB0). Ce registre d'émission est le seul que notre programme utilise, le débit CAN étant supérieur au débit RS232, et ceci assurant d'envoyer les trames dans le bon ordre.

Je pense que cette explication était nécessaire pour bien comprendre le fonctionnement de ces emplacements particuliers.

ATTENTION : En modificant les bits WINx (mode 0) ou EWINx (modes 1 et 2), vous accédez aux registres sélectionnés lorsque vous accédez aux emplacements de RXB0 indépendamment du mode d'accès : access-bank, mode banked, ou adressage indirect.

Dit de façon plus explicite, si vous modifiez les bits WIN ou EWIN (selon le mode) à une valeur pointant sur autre chose que RXB0, vous n'avez plus aucun moyen d'accéder aux registres RXB0, sauf en replaçant les bits WIN ou EWIN à une valeur pointant sur RXB0.

Si je reprends le premier exemple (mode 0), voici ce que ça donne :

```
movf   RXB0CON,w       ; on charge le registre réel RXB0CON
bsf    CANCON,WIN1     ; WIN = 010
movf   RXB0CON,w       ; on charge le registre réel TXB2CON
movff  RXB0CON,var     ; on copie bel et bien TXB2CON dans var
lfsr   FSR1,RXB0CON    ; on pointe sur l'adresse de RXB0CON
movff  INDF1,var       ; mais on copie bel et bien TXB2CON dans var
bcf    CANCON,WIN1     ; seule façon de reprendre accès à RXB0CON
movf   RXB0CON,w       ; on charge alors RXB0CON
movff  RXB0CON,var     ; on copie RXB0CON dans var
lfsr   FSR1,RXB0CON    ; on pointe sur l'adresse de RXB0CON
movff  INDF1,var       ; et on copie bel et bien RXB0CON dans var
```

Avec ces explications, vous ne devriez plus vous faire piéger.

Après ce petit aparté, voyons maintenant nos variables :

```
;=============================================================================
;                          VARIABLES ACCESS RAM                               =
;=============================================================================
; zone de 96 octets
; -----------------
 CBLOCK   0x00              ; zone access ram de la banque 0
   flags : 1                ; flags divers
                            ; b7/b1 : N.U.
                            ; b0 : 1 = erreur d'eeprom

   ptrcr : 2                ; pointeur prochaine trame CAN à recevoir
   ptrct : 2                ; pointeur prochaine trame CAN à traiter

   nbenv : 1                ; nombre d'octets déjà envoyés dans trame actuelle
   nbserout : 1             ; nombre de trames présentes dans buffer d'envoi
   numeserout : 1           ; numéro de la prochaine trame USART à écrire
   numsserout : 1           ; numéro de la prochaine trame USART à envoyer
   checkout : 1             ; checksum de la trame en cours d'envoi

   nbrec : 1                ; nombre d'octets reçus trame actuelle USART
   numrserin : 1            ; numéro de la prochaine trame à recevoir USART
   numtserin : 1            ; numéro de la prochaine trame à traiter USART
   nbserin : 1              ; nombre de trames valides dans le buffer USART
                            ; pourrait se recalculer à partir des 2 précédentes,
                            ; mais ne permettrait pas de remplir le buffer, et
                            ; ralentirait le traitement interrupt USART
   checkin : 1              ; checksum de la trame en cours de réception

   recid : 1                ; ID de la machine qui envoie la commande

   cmptlede : 1             ; compteur d'allumage pour led émission
```

```
   cmptledr : 1             ; compteur d'allumage pour led réception

   local01 : 1              ; variable locale
   local02 : 1              ; variable locale
   local03 : 1              ; variable locale
   local04 : 1              ; variable locale

   wreg_lp : 1              ; sauvegarde W pour interrupts L.P.
   status_lp : 1            ; sauvegarde STATUS pour interrupts L.P.
   fsr0_lp : 2              ; sauvegarde pour FSR0 interrupts L.P.
   ecancon_lp : 1           ; sauvegarde de ECANCON interrupts L.P.

   wreg_hp : 1              ; sauvegarde W pour interrupts H.P.
   status_hp : 1            ; sauvegarde STATUS pour interrupts H.P.
   fsr0_hp : 2              ; sauvegarde FSR0 interrupts H.P.

 ENDC
                  ; DEFINES POUR FLAGS
                  ; ------------------
#DEFINE ERREEP  flags,0     ; erreur d'eeprom constatée
```

Vous noterez que la RAM contient principalement différents pointeurs sur les buffers que nous allons examiner ci-dessous.

Notez également que les sauvegardes des registres pour les interruptions sont placées dans l'access-bank, ce qui permet de laisser plus de place dans le reste de la RAM pour y placer nos différents buffers.

A ce propos, notez que plusieurs registres sont sauvegardés à deux emplacements différents. En effet, ils sont modifiés 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. Remarquez également que FSR0 est constitué de deux registres de 8 bits (FSR0L/FSR0H), ce qui nécessite deux emplacements pour chacune des interruptions.

```
;=============================================================================
;                     VARIABLES BANQUE 0 NON ACCESS                           =
;=============================================================================
; zone de 160 octets
; ------------------
 CBLOCK   0x60
   serbufin : .80           ; 5 commandes série en attente de traitement
   serbufout : .80          ; 5 commandes série en attente d'envoi
 ENDC
```

Nous trouvons ici deux buffers pour la liaison série : un buffer de réception (serbufin) et un buffer d'émission. Le buffer de réception peut sembler inutile, puisque le PC attend une confirmation avant d'envoyer la commande suivante.

En fait, ça l'est effectivement, mais ce logiciel est identique à celui de la carte d'interface CAN/Ethernet. Or, sur cette carte, plusieurs PC peuvent être amenés à dialoguer simultanément avec la carte d'interface, ce qui justifie la présence de ce buffer.

Ca n'amenait de plus aucun bénéfice de le supprimer sur cette carte d'interface série, je l'ai donc laissé.

Sans compter que ça permet de relier la carte à un convertisseur ethernet/série, qui, lui, peut communiquer avec plusieurs PC simultanément.

Il en est de même pour le buffer de réception, vu qu'un buffer CAN est déjà en place. De nouveau, ça permettait également d'éviter des délais dans les analyses de trames reçues dans le cas où la carte ethernet dialoguait simultanément avec plusieurs PC distants.

```
;=============================================================================
;                     VARIABLES BANQUES 1 à 12                                =
;=============================================================================
;-----------------------------------------------------------------------------
; 18Fx680 : jusque 0xCFF en mode normal et 0xCF3 en mode debugger
;-----------------------------------------------------------------------------
 CBLOCK 0x100

   canbufin : .3071      ; 236 trames CAN en attente de traitement

 ENDC

LASTT EQU LASTRAM - .12   ; dernier emplacement pouvant accueillir une
                          ; trame de 13 octets
                          ; voir configurations.inc
```

Cette zone contient les trames CAN reçues du bus. La taille autorise en fait la mémorisation de 236 trames (en fait, 235, si vous étudiez attentivement le fonctionnement des pointeurs cycliques). Chaque trame contient en effet 13 octets : ID sur 4 octets/29 bits + nombre d'octets de data (DLC) sur un octet + un maxium de 8 octets de data CAN.

Les octets de data inutilisés sont réservés quand même, ça simplifie la gestion des buffers, et augmente la rapidité de réaction du logiciel. De plus, ça évite qu'une erreur occasionnelle ne désynchronise toutes les trames reçues.

La position LASTT est calculée dans le fichier « configurations.inc » et tient compte de l'activation ou non du debuggeur ICD2®. Il s'agit du dernier emplacement RAM utilisable.

Notez que, contrairement à un PIC® de la famille 16F, étant donné que les adresses RAM sont consécutives d'une banque à l'autre, il est très simple d'utiliser des zones à cheval sur plusieurs banques, je ne m'en suis évidemment pas privé.

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

```
;=============================================================================
;=============================================================================
;                                VECTEURS                                    =
;=============================================================================
;=============================================================================

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

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

   ORG 0x18             ; vecteur d'interruption basse priorité
                        ; traité juste dessous
```

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 le hardware du PIC®. Il s'agit donc plus de la destination des vecteurs, que des vecteurs eux-mêmes. Mais bon, vu que c'est la dénomination que Microchip® adopte, autant vous y habituer.

A titre de comparaison, 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 PIC®, 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                      =
;=============================================================================
;=============================================================================

               ; sauvegarde des registres
               ; -------------------------
   movff  STATUS,status_lp   ; sauver STATUS
   movff  WREG,wreg_lp       ; ainsi que WREG
   movff  FSR0H,fsr0_lp      ; sauver FSR0H
   movff  FSR0L,fsr0_lp+1    ; et FSR0L
```

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 ainsi que tout registre modifié par cette routine.

```
               ; scrutation des interruptions
               ; ----------------------------
   btfsc PIR3,RXBnIF        ; tester si réception d'une trame CAN
   rcall intcanrec          ; oui, traiter réception CAN

               ; 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 émission USART
               ; -------------------
intl2
   btfss PIE1,TXIE          ; tester si interrupt en service
   bra   intlrest           ; non, fin
   btfsc PIR1,TXIF          ; oui, tester si interrupt émission USART
   rcall inttx              ; traiter interrupt
```

Nous ne faisons rien d'autre ici que de déterminer la cause de l'interruption. Etant donné qu'il y a 3 sources d'interruption possibles en basse priorité, nous aurons 3 tests.

Les sources d'interruptions timer et USART sont coupées et remises en service en fonction des besoins du programme, il est donc impératif de tester également le bit de validation, sans quoi nous pourrions détecter une interruption qui n'est même pas en service. Souvenez-vous en effet (cours-part1) que les flags d'interruption sont susceptibles d'être positionnés même si l'interruption correspondante n'est pas en service.

Ces tests n'amènent aucun autre 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.

```
               ; restauration des registres
               ; --------------------------
intlrest
   movff fsr0_lp,FSR0H      ; restaurer FSR0H
   movff fsr0_lp+1,FSR0L    ; et FSR0L
   movff wreg_lp,WREG       ; ainsi que WREG
   movff status_lp,STATUS   ; et STATUS
   retfie                   ; puis, retour d'interruption
```

Rien de spécial, on restaure les registres qu'on avait sauvegardés, ce qui est effectué de nouveau aisément à l'aide d'instructions « movff », qui, je le rappelle, ont l'avantage de ne pas modifier les bits du registre « STATUS ».

Voyons maintenant notre routine de réception USART, qui est de plus notre seule routine haute priorité. Pour ceux que ça étonnerait, souvenez-vous que les trames RS232 sont reçues octet par octet (une interruption par octet), alors que les trames CAN sont reçues trame par trame (une interruption par trame complète). Ceci explique que nous devions réagir plus vite à la réception USART qu'à la réception CAN, afin de libérer le buffer le plus vite possible pour la réception de l'octet suivant. Ceci sans compter qu'en outre nous disposons d'une file FIFO de 8 trames CAN en réception, gérée par hardware.

```
;=============================================================================
;=============================================================================
;                          INTERRUPTIONS HAUTE PRIORITE                      =
;=============================================================================
;=============================================================================

;=============================================================================
;                   INTERRUPTION RECEPTION USART (H.P.)                      =
;=============================================================================
;-----------------------------------------------------------------------------
; on reçoit un octet en provenance du PC
; on sauve l'octet dans le buffer de réception USART : serbufin (banque 0)
; le buffer contient 5 trames de 16 octets
; le nombre d'octets déjà reçus pour la trame en cours est nbrec
; Le numéro de la trame à enregistrer est dans numrserin
; on n'utilise pas FAST à cause de bugs souvent constatés avec cette option,
; notamment en utilisant l'ICD2.
;-----------------------------------------------------------------------------
inth
                ; sauvegarde registres
                ; --------------------
   movff  WREG,wreg_hp       ; sauver WREG
   movff  STATUS,status_hp   ; et STATUS
   movff  FSR0H,fsr0_hp      ; sauver FSR0H
   movff  FSR0L,fsr0_hp+1    ; sauver FSR0L
```

Les plus assidus d'entre vous feront sans doute la remarque pertinente qu'il est inutile de sauvegarder STATUS et WREG, vu que vous sommes en interruption haute-priorité et que nous avons dès lors l'opportunité d'utiliser le mode « FAST » pour le retour d'interruption.

En fait, dans la pratique, ce mode m'a causé plusieurs désagréments. D'une part, il semble assez peu conciliable avec l'utilisation de l'ICD2® en phase de debuggage, il apparaît d'après Microchip® que les routines ICD2® utilisent déjà à leur profit le mode « FAST », qui ne serait dès lors plus disponible pour l'application.

Pire, à propos de certains bugs incompréhensibles, j'ai eu des allusions un peu évasives en provenance de Microchip® concernant ce fameux mode FAST. J'en ai déduit (peut-être à tort) qu'il ne semblait pas fiable, et que, dès lors, mieux valait ne pas y recourir.

Il se peut également que ces bugs n'aient été présents que dans les 18Fxx8, assez truffés de bugs en tout genre (ce qui explique la migration de Domocan sur les nouveaux xx80), et qu'avec les nouveaux modèles 18Fxx80 ceci soit résolu. Quoi qu'il en soit, dans le doute, je préfère m'en passer si l'utilisation de quelques instructions supplémentaires n'est pas un problème, ce qui est le cas ici.

Mais voyons la suite :

```
                ; sauver l'octet reçu
                ; -------------------
   setf  cmptledr           ; compteur d'allumage led réception
   bsf   LEDR               ; allumer led réception
   clrf  TMR0L              ; reset du timer 0
   lfsr  FSR0,serbufin      ; pointer sur buffer réception série
   swapf numrserin,w        ; numéro trame à écrire * 16
   addwf nbrec,w            ; ajouter n° de l'octet en cours
   addwf FSR0L,f            ; pointer sur emplacement correct
   movf  RCREG,w            ; charger octet reçu
   movwf INDF0              ; sauver dans l'emplacement prévu
```

Dans cette partie, on sauve l'octet reçu (en provenance du PC) dans le buffer de réception USART, en fonction du numéro de l'emplacement libre, et du numéro de l'octet en cours de réception. Nous resetons également le timer servant de détection d'absence de réception, afin d'éviter qu'il ne déborde tant que les octets arrivent normalement.

Chaque buffer réserve 16 octets, ce qui accélère le traitement, puisque la multiplication par 16 revient à swapper le numéro d'emplacement (Astuce : la multiplication par 16 d'un nombre de moins de 5 bits = décalage à gauche de 4 emplacements = swap du nombre)

Ensuite, nous allons traiter chaque octet en fonction de sa position dans la trame, étant donné qu'il y a 2 octets bien particuliers :

```
                ; traiter 1er octet de la trame
                ; -----------------------------
   tstfsz nbrec             ; tester si premier octet
   bra    inth1             ; non, voir autre cas
   bcf    INTCON,TMR0IF     ; oui, effacer flag timer0
   bsf    INTCON,TMR0IE     ; et mettre l'interruption timer0 en service
   incf   nbrec,f           ; comptabiliser l'octet reçu
   movwf  checkin           ; sauver octet reçu dans checksum
   bra    inthrest          ; et fin de traitement
```

L'octet 0 est le premier de la trame. Vu que c'est le début de la réception d'une nouvelle trame, nous lançons le timer chargé de vérifier l'absence de «trou » de réception. Nous en profitons également pour initialiser le checksum avec la valeur de ce premier octet.

```
                ; traiter 16ème octet de la trame
                ; --------------------------------
inth1
   incf   nbrec,f           ; incrémenter nombre d'octets déjà reçus
   btfss  nbrec,4           ; dernier?
   bra    inth2             ; non, traiter cas général
   clrf   nbrec             ; prochain octet = premier de la trame suivante
   bcf    INTCON,TMR0IE     ; stopper interrupts timer 0
   movf   INDF0,w           ; charger dernier octet (checksum)
   xorwf  checkin,w         ; comparer avec checksum calculé
   bnz    inthrest          ; pas identiques? on ignore la trame reçue
   movlw  -1                ; préparer rebouclage trame 4->0
   btfsc  numrserin,2       ; tester si trame actuelle = 4
   movwf  numrserin         ; oui, alors trame -1
   incf   numrserin,f       ; incrémenter numéro de trame
   incf   nbserin,f         ; incrémenter nombre de trames dans buffer
   bra    inthrest          ; et fin de traitement
```

Nous traitons ici le dernier octet de la trame, qui est le checksum. Celui-ci doit correspondre à la somme (sur un octet, ce qui correspond au reste de la division par 256 de cette somme) de tous les octets précédents. Si c'est bien le cas, la trame est prise en compte, sinon elle est simplement ignorée. Le timer de détection de trou de réception est stoppé, et la réception de la prochaine trame est préparée.

```
                ; traiter octets 1 à 14
                ; ---------------------
inth2
   addwf checkin,f       ; ajouter octet au checksum
```

Le traitement des autres octets de la trame correspond simplement à ajouter leur valeur au checksum courant. La sauvegarde de l'octet ayant déjà été effectuée.

```
                    ; restaurer registres
                    ; -------------------
inthrest
   movff fsr0_hp,FSR0H      ; restaurer FSR0H
   movff fsr0_hp+1,FSR0L    ; restaurer FSR0L
   movff status_hp,STATUS   ; restaurer STATUS
   movff wreg_hp,WREG       ; restaurer WREG
   btfss RCSTA,OERR         ; tester si overflow réception
   retfie                   ; non, OK
   bcf   RCSTA,CREN         ; oui, couper réception, effacer OERR
   bsf   RCSTA,CREN         ; relancer réception
   retfie                   ; et fin
```

Et enfin, nous terminons en restaurant de façon classique les registres sauvegardés, puisque nous n'utilisons pas le « retfie FAST ».

On en profite au passage pour vérifier que le module USART n'est pas bloqué suite à une erreur d'overflow, auquel cas on le libère.

Voyons maintenant notre routine d'interruption timer, qui vérifie l'absence d'un trou dans la réception :

```
;=============================================================================
;=============================================================================
;                            ROUTINES D'INTERRUPTION                         =
;=============================================================================
;=============================================================================

;=============================================================================
;                         INTERRUPTION TIMER 0 (L.P.)                        =
;=============================================================================
;-----------------------------------------------------------------------------
; Intervient si aucun octet n'est reçu en 205µs depuis la RS232
; Le timer ne doit jamais déborder, car sinon ça veut dire qu'on n'a pas
; reçu d'octets depuis 205µs alors qu'on ne les a pas reçu tous les 16.
; Il y a une marge de sécurité, car la réception d'un octet prend 87 µs
;-----------------------------------------------------------------------------
inttmr0
                    ; arrêt du timer
                    ; --------------
   bcfINTCON,TMR0IE,0       ; fin des interruptions timer
   clrf  nbrec              ; aucun octet valide reçu, trame annulée
```

```
            ; vérifier si réception pas bloquée
            ; ---------------------------------
   btfss  RCSTA,OERR         ; tester si overflow réception
   return                    ; non, OK
   bcf    RCSTA,CREN         ; oui, couper réception, effacer OERR
   bsf    RCSTA,CREN         ; relancer réception
   return                    ; fin d'interruption timer
```

Cette interruption intervient si une durée de 205µs a été comptabilisée sans qu'aucun octet normalement attendu n'arrive.

Cette routine se contente d'annuler les octets déjà reçus, et en profite pour vérifier qu'un blocage de l'USART ne se soit pas produit à cause d'une erreur de réception. Si c'est le cas, la réception est désactivée puis réactivée, ce qui acquitte l'erreur.

Voyons maintenant notre routine de réception de trame CAN en mode 2. Pour ceux qui avaient étudié les précédentes routines en mode 0 (avant la révision 6.0 du logiciel), tout est ici modifié, il s'agit d'une nouvelle façon de procéder.

```
;=============================================================================
;                 INTERRUPTION RECEPTION TRAME CAN FIFO (LP)                  =
;=============================================================================
;-----------------------------------------------------------------------------
; Réception d'une trame CAN dans la file FIFO (ECAN en mode 2)
; Les bits FPx de CANCON pointent sur l'actuel buffer à lire
; On accède au registre concerné avec les bits EWINx de ECANCON
; On copie la trame dans le buffer RAM : 13 octets par trame
; On utilise le buffer partant de l'adresse RAM tramesbuf jusqu'à la fin
; de la dernière banque du pic utilisé. La zone tient compte de la présence
; ou non d'un debugger ICD2
; LASTT indique le dernier emplacement valide pour sauver une trame
; la gestion est assurée par 2 pointeurs circulaires ptrr (ptr reçu) et
; ptrt (ptr traité). Si les 2 pointeurs pointent sur la même valeur, alors
; c'est que toutes les commandes sont traitées.
; Le débordement du buffer RAM n'est pas géré, le pic est suffisamment
; rapide pour ne pas s'en préoccuper, au vu des tâches à accomplir
; la structure en RAM est la suivante : SIDH,SIDL,EIDH,EIDL,DLC,DATA
; on doit sauver le destinataire car des cartes peuvent avoir besoin de
; commandes qui ne leur sont pas explicitement destinées (trames clock etc)
;-----------------------------------------------------------------------------
```

Comme indiqué, en mode 2 chaque trame arrive dans un buffer dont l'emplacement est indiqué par les 3 bits de poids faible du registre CANCON. Or, dans le mode 2, le registre réellement pointé lorsqu'on accède aux registres RXB0xxx dépend de la valeur présente dans les bits EWIN4 à EWIN0, nous en avons déjà parlé.

Pour accéder au buffer de réception concerné, il suffit simplement de recopier les 3 bits de poids faible de CANCON dans le registre ECANCON, puis forcer EWIN4 à 1. Ceci permet de pointer sur le buffer de réception dont le numéro est indiqué par CANCON, en accédant simplement aux registres RXB0xxx. (RXB0CON, RXB0SIDH, etc).

On obtient ainsi une file FIFO dont le numéro de la prochaine trame à lire est indiqué par CANCON. Tout est donc automatique, et les trames sont traitées dans leur ordre d'arrivée.

Les « alias » RXBx sont utilisés pour bien montrer qu'en fait on n'accède pas forcément au registre RXB0 en lui-même, mais en fait au registre pointé par ECANCON. Mais ça aussi, j'en ai déjà parlé.

Voyons comment ça se passe en pratique :

```
intcanrec
                  ; initialisations
                  ; ---------------
   movff ECANCON,ecancon_lp ; sauver ECANCON

            ; pointer sur le bon buffer de réception
            ; --------------------------------------
intcanrecloop
   movf   CANCON,w        ; charger registre CANCON
   xorwf  ECANCON,w       ; conserver tout bit différent de ECANCON
   andlw  B'00001111'     ; limiter la différences à FPx
   xorwf  ECANCON,f       ; inverser les bits EWIN différents de FP
   bsf    ECANCON,4       ; et forcer EWIN4 à 1
```

Le problème est d'amener les 4 bits FP de CANCON dans les bits EWIN de ECANCON, sans modifier les autres bits (principalement le bit FIFOWM). Je vous laisse réfléchir à l'astuce imaginée, essayez une autre méthode et comparez les longueurs obtenues.

On force ensuite EWIN4 à 1 pour indiquer qu'on désire accéder aux buffers de réception, et à partir de là, tout accès à un registre RXB0 (RXBx) permet d'accéder au registre contenant notre trame : enfantin et très souple d'utilisation.

```
               ; tester si message dans le buffer
               ; --------------------------------
   bcf    PIR3,RXBnIE           ; acquiter flag d'interruption
   btfsc  RXBxCON,RXFUL         ; tester si buffer plein
   bra    intcanrec2            ; oui, continuer traitement
   movff  ecancon_lp,ECANCON    ; non, restaurer ECANCON
   return                       ; et terminer traitement (FIFO vide)
```

Nous commençons par tester si le buffer est vide. Vous allez me dire qu'il est forcément plein, mais en fait pas forcément. D'une part, un bug dans certaines révisions de picxx80 fait que vous pouvez recevoir une interruption, dans des conditions particulières, même si le buffer est vide.

Mais, de toutes façons, procéder de la sorte est pratique. En effet, vous pouvez recevoir plusieurs trames lors de la même réception (vu le mode FIFO) : il vous suffit donc de traiter chaque trame jusqu'à ce que vous trouviez un buffer vide, ce qui indique que la file FIFO est vide et que donc vous avez tout traité.

La sortie de cette boucle est donc le « return » que vous avez ci-dessus, le test n'est donc pas un test inutile, le bug éventuel subsistant n'y change strictement rien.

```
               ; copier message dans buffer RAM
               ; ------------------------------
intcanrec2
   movff  ptrcr,FSR0H        ; pointer sur emplacement actuel
   movff  ptrcr+1,FSR0L      ; idem LSB
   movff  RXBxSIDH,POSTINC0  ; sauvergarder SIDH (destinataire)
```

```
movff  RXBxSIDL,POSTINC0  ; sauvegarder SIDL (commande)
movff  RXBxEIDH,POSTINC0  ; sauver EIDH (cible)
movff  RXBxEIDL,POSTINC0  ; sauver EIDL (paramètre)
movff  RXBxDLC,POSTINC0   ; sauvegarder RXB1DLC (nombre de data)
movff  RXBxD0,POSTINC0    ; sauver data 0
movff  RXBxD1,POSTINC0    ; sauver data 1
movff  RXBxD2,POSTINC0    ; sauver data 2
movff  RXBxD3,POSTINC0    ; sauver data 3
movff  RXBxD4,POSTINC0    ; sauver data 4
movff  RXBxD5,POSTINC0    ; sauver data 5
movff  RXBxD6,POSTINC0    ; sauver data 6
movff  RXBxD7,POSTINC0    ; sauver data 7
bcf    RXBxCON,RXFUL      ; libérer le buffer de réception
```

J'ai utilisé une succession d'instructions plutôt qu'une boucle. On ne manque pas de place en mémoire programme, et procéder de la sorte est plus rapide, c'est donc tout bénéfice. Une fois pointé sur le bon emplacement du buffer software (RAM), on recopie simplement la trame à cet emplacement.

Notez donc que vous avez en fait deux niveaux de buffer : un premier niveau « hardware » constitué de la file FIFO de réception CAN, et un second géré par software, nous permettant de stocker nos trames pour traitement ultérieur. Ceci devrait nous mettre à l'abri d'éventuelles saturations.

La routine se termine en libérant le buffer. A ce stade, si la file contient encore d'autres trames non traitées, le pointeur en CANCON est automatiquement modifié en conséquence.

```
            ; gérer le pointeur
            ; -----------------
movlw  0x0D            ; charger incrément pointeur
addwf  ptrcr+1,f       ; pointer sur suivant
clrf   WREG            ; pour report
addwfc ptrcr,f         ; ajouter report

movlw  HIGH(LASTT)     ; dernière banque valide
cpfseq ptrcr           ; tester si on est dans la dernière banque
bra    intcanrecloop   ; non, pas de problème, tester si autre trame présente
movlw  LOW(LASTT)      ; début de dernière trame valide
cpfsgt ptrcr+1         ; tester si dépassé
bra    intcanrecloop   ; non, c'est bon
movlw  HIGH(canbufin)  ; oui, banque du début du buffer
movwf  ptrcr           ; dans pointeur haut
movlw  LOW(canbufin)   ; poids faible début zone buffer
movwf  ptrcr+1         ; dans LSB pointeur
bra    intcanrecloop   ; tester si autre trame dans FIFO
```

On termine en calculant le prochain emplacement disponible dans notre buffer software, puis on retourne au début voir s'il y a encore des trames dans la file.

Passons ensuite à notre interruption d'émission USART. Celle qui envoie les trames préparées vers le PC.

```
;=============================================================================
;                    INTERRUPTION EMISSION USART (L.P.)                       =
;=============================================================================
;-----------------------------------------------------------------------------
; nbenv contient le nombre d'octets déjà envoyés dans la trame actuelle
; nbserout contient le nombre de trames présentes dans buffer d'envoi
; numsserout contient le numéro de la prochaine trame à envoyer
; checkout contient le checksum de la trame en cours d'envoi
; serbufout est le buffer d'envoi
;-----------------------------------------------------------------------------
```

Comme vous pouvez le constater, tout se contrôle à l'aide de 2 variables (compteur de trames restantes et compteur d'octets envoyés), ainsi qu'avec un pointeur sur la prochaine trame à envoyer

Le principe est assez simple : tant qu'on n'a pas traité les 16 octets de la trame, on continue l'envoi automatique par l'interruption. Si on a traité les 16 octets, on décrémente le nombre de trames restant à envoyer, et s'il n'y en a plus, on stoppe l'interruption d'envoi USART.

```
inttx
                ; pointer sur l'octet concerné
                ; ----------------------------
    lfsr   FSR0,serbufout     ; pointer sur buffer d'émission
    swapf  numsserout,w       ; numéro trame * 16
    addwf  nbenv,w            ; plus numéro de l'octet à envoyer
    addwf  FSR0L              ; pointer sur le bon octet
```

On commence par pointer sur l'octet à envoyer. Ceci se calcule aisément en ajoutant le numéro de l'octet à envoyer à l'emplacement du premier octet de la trame à envoyer. Ce premier emplacement s'obtient également aisément en multipliant par 16 (swapf) le numéro de la trame et en l'ajoutant à l'adresse de départ du buffer.

Dit sous forme mathématique :

Emplacement de l'octet = position du buffer + (numéro de trame *16) + numéro de l'octet

De nouveau, c'est plus long à expliquer qu'à programmer. Ensuite, on traite, comme pour la réception, en fonction du numéro de l'octet :

```
                ; traiter premier octet
                ; ----------------------
    setf   cmptlede        ; compteur d'allumage led émission
    bsf    LEDE            ; allumer led émission
    tstfsz nbenv           ; tester si premier octet
    bra    inttx2          ; non, voir autre cas
    movf   INDF0,w         ; charger en-tête trame
    movwf  TXREG           ; dans registre d'envoi
    movwf  checkout        ; et dans checksum
    incf   nbenv           ; pointer sur suivant
    return                 ; et fin
```

L'envoi du premier octet s'accompagne de sa sauvegarde dans le checksum, ce qui correspond à son initialisation.

```
                ; traiter dernier octet
                ; ----------------------
```

```
inttx2
   incf  nbenv,f           ; incrémenter numéro de l'octet
   btfss nbenv,4           ; dernier?
   bra   inttx3            ; non, traiter autres cas

   movff checkout,TXREG    ; oui, envoyer le checksum
   clrf  nbenv             ; prochain octet = en-tête

   movlw -1                ; préparer rebouclage trame 4->0
   btfsc numsserout,2      ; tester si trame actuelle = 4
   movwf numsserout        ; oui, alors trame -1
   incf  numsserout,f      ; incrémenter numéro de trame

   dcfsnznbserout,f        ; décrémenter nombre de trames restantes
   bcf   PIE1,TXIE         ; terminé, arrêt d'émission
   return                  ; fin de traitement
```

Le dernier octet est le checksum. Celui-ci a été calculé directement par cette routine d'interruption, il suffit donc de l'envoyer. Ensuite, on pointe sur la position suivante (buffer circulaire). On décrémente le nombre de trames restantes, s'il n'en reste plus, on stoppe l'émission en coupant l'interruption d'émission USART.

```
               ; traiter octets 1 à 14
               ; ---------------------
inttx3
   movf  INDF0,w           ; charger octet
   movwf TXREG             ; l'envoyer
   addwf checkout,f        ; et l'ajouter au checksum
   return                  ; fin de traitement
```

Enfin, tous les autres octets se traitent simplement en envoyant l'octet pointé et en l'additionnant au checksum en cours.

Nous en arrivons à notre routine d'initialisation :

```
;=============================================================================
;=============================================================================
;                           INITIALISATIONS                                  =
;=============================================================================
;=============================================================================
;-----------------------------------------------------------------------------
; contient les initialisations exécutées lors d'un reset
; ----------------------------------------------------------------------------
init
               ; initialisation PORTS
               ; --------------------
   movlw VALPORTB        ; valeur par défaut PORTB
   movwf LATB            ; dans PORTB
   movlw DIRPORTB        ; direction PORTB
   movwf TRISB           ; dans TRISB

   movlw VALPORTC        ; valeur par défaut PORTC
   movwf LATC            ; dans PORTC
   movlw DIRPORTC        ; direction PORTC
   movwf TRISC           ; dans TRISC
```

Rien que du classique, on configure nos ports utilisés (ici, PORTB et PORC) en direction, et en valeurs par défaut.

```
                ; 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
```

Ensuite on efface la RAM banque 0, ce qui permet en cas d'oubli d'initialisation de démarrer avec une RAM dont toutes les variables sont à 0x00. On peut s'en passer en initialisant chaque variable séparément, mais le temps n'est gagné qu'une seule fois à la mise sous tension de la carte, ce qui ne présente aucun intérêt dans ce genre d'application. Mieux vaut donc jouer la sécurité.

```
            ; vérifier paramètres CAN en eeprom
            ; ---------------------------------
   clrf  WREG            ; adresse 0 en eeprom
   rcall readeepw        ; lire octet 1 en-tête
   xorlw 'B'             ; comparer avec "B"
   bnz   init2           ; pas bon, paramètres par défaut
   rcall readeep         ; lire octet 2 en-tête
   xorlw 'G'             ; comparer avec "G"
   bz    init3           ; correspond, traitement normal
```

Ici, on teste si l'en-tête de la zone contenant les paramètres CAN est bien « BG », ce qui confirme que les données qu'on lit sont valides. Si ce n'est pas le cas, on réinitialise les paramètres avec les paramètres par défaut présents dans le fichier « parcan.inc » tel qu'il était le jour où vous avez assemblé votre programme (d'où l'importance de configurer ce fichier avant de programmer votre PIC®).

Si l'en-tête est incorrect, une erreur sera signalée au PC s'il demande une lecture des paramètres CAN. Dans ce cas, procédez à une écriture pour reformer l'en-tête. Une erreur de ce type peut avoir deux causes principales :

- Soit une erreur de vérification de l'eeprom après une écriture a empêché la validation de l'en-tête (parasite au moment de l'écriture, eeprom en fin de vie, erreur aléatoire).

- Soit la tension a été coupée alors que l'écriture en eeprom était en cours.

```
            ; eeprom incorrecte, paramètres CAN par défaut
            ; ----------------------------------------
init2
   bsf   ERREEP          ; signaler erreur d'eeprom
   movlw PARCAN_EEP         ; adresse des paramètres CAN en eeprom
   movwf EEADR           ; dans pointeur EEPROM
   movlw TQ              ; TQ du fichier de paramètres (défaut)
   rcall writeeep        ; écrire en eeprom
   movlw PROP            ; Temps de propagation par défaut
   rcall writeeep        ; écrire en eeprom
   movlw SEG1            ; phase segment1 par défaut
   rcall writeeep        ; écrire en eeprom
   movlw SEG2            ; phase segment2 par défaut
   rcall writeeep        ; écrire en eeprom
   movlw SJW             ; SJW par défaut
   rcall writeeep        ; écrire en eeprom
   movlw SAMPL           ; sample par défaut
   rcall writeeep        ; écrire en eeprom
```

Cette partie remet les valeurs par défaut en cas d'erreur EEPROM. Notez que les « TQ, PROP » etc sont remplacés par des valeurs figées au moment de l'assemblage. Une modification ultérieure de votre fichier « ParCan.inc » n'aura donc aucun effet si vous ne réassemblez pas votre programme et que vous ne reprogrammez pas votre PIC®. Par contre, vous pourrez toujours utiliser le paramétrage via votre PC et Domogest.

N'oubliez pas cependant qu'en cas de problème d'écriture en eeprom, votre carte reviendra à ses valeurs par défaut. Par sécurité, éditez de ce fait correctement votre fichier « ParCan.inc » avant d'assembler votre programme.

```
                ; initialiser CAN
                ; ---------------
init3
   SETCONFIG            ; passer CAN en mode configuration
   rcall setbrgconx     ; initialiser registres BRGCON 1 à 3
   movlw B'00100000'    ; niveau récessif forcé à Vdd, pas de mode capture
   movwf CIOCON         ; dans registre de contrôle
   rcall setmasfil      ; placer filtre et masque 0
   SETMODE2             ; valider CAN mode 2
```

Après avoir passé le module ECAN en mode configuration grâce à notre macro, nous commençons l'initialisation du module et nous appelons la routine d'initialisation des filtres et des masques.

```
   BANKSEL  BIE0        ; pointer sur banque  13
   setf  BIE0           ; valider interruptions sur buffers réception
   bcf   MSEL0,FIL0_1   ; correspondance filtre/masque
   bcf   MSEL0,FIL0_0   ; 0 0 = masque 0
   movlw B'00000001'    ; filtre 0 = ON, 1 à 7 = OFF
   movwf RXFCON0        ; dans registre de contrôle
   clrf  RXFCON1        ; couper filtres 8 à 15
   BANKSEL  0           ; repasser en banque 0
   POINTRX0             ; accéder au buffer de réception 0
```

Ensuite, nous mettons en service uniquement le filtre 0, que nous faisons correspondre avec le masque 0. C'est plus que suffisant pour une carte d'interface à en juger par la pratique. On valide également les interruptions de réception sur chaque buffer de réception utilisé.

Contrairement au mode 0, le mode 2 permet en effet d'établir une liaison entre n'importe quel filtre et n'importe quel masque.

```
   bcf   PIR3,RXBnIF        ; reset flag interrupt réception CAN
   bcf   IPR3,RXBnIP        ; interruption réception CAN basse priorité
   bsf   PIE3,RXBnIE        ; valider interruptions réception CAN (+ BIE0)
```

Dans cette portion, on valide les interruptions CAN en mode basse priorité.

```
            ; Initialisation USART
            ; --------------------
   movlw B'00001000'    ; BRG fixe et en mode 16 bits (haute précision)
   movwf BAUDCON        ; dans registre de contrôle
   movlw LOW(.86)       ; débit 23400 bauds théoriques...
   movwf SPBRG          ; ...et 114942 bauds réels...
   movlw HIGH(.86)      ; ...soit 0,2% d'erreur...
   movwf SPBRGH         ; ...2 registres = mode 16 bits haute précision
```

On établit ici le débit CAN à 115.200 bauds. On profite d'un nouveau mode de fixation de débit, le mode « haute précision » présent sur ces modèles de PIC®. Le registre SPBRG répond alors à la formule (en mode hi-speed) :

SPBRG = (Fosc/ (4 * débit)) – 1

Soit dans notre cas : (40.000.000/(4*115.200)) -1 = 85,8 que nous arrondirons à 86 (décimal)

Il faut alors placer cette valeur, codée sur 16 bits, dans les registres SPBRGH (poids fort) et SPBRG (poids faible)

Nous obtenons alors un débit réel de

Debit = Fosc/ (4 * (SPBRG+1))

Soit 40.000.000 / (4 *87) = 114942 bauds

Donc, une erreur de (114942 – 115200) / 115200, soit 0,22%

Ce qui est largement sous les 5% exigés (voir cours-part2).

```
movlw B'00100110'        ; transmission 8 bits asynchrone Hi-speed
movwf TXSTA              ; dans registre de contrôle
movlw B'10010000'        ; réception en service, port série en service
movwf RCSTA              ; dans registre de contrôle
```

On fixe alors ici le mode 8 bits asynchrone hi-speed

Ensuite, on prépare l'interruption émission USART en basse priorité sans la lancer, puis on lance l'interruption réception USART en mode haute-priorité (mode par défaut) :

```
bcf   IPR1,TXIP       ; interruption transmission basse priorité
bsf   PIE1,RCIE       ; interruption réception USART en service (H.P.)
```

Ceci étant fait, on s'intéresse alors à l'initialisation de nos pointeurs CAN :

```
               ; initialiser variables
               ; ---------------------
movlw HIGH(canbufin)     ; poids fort emplacement buffer CAN
movwf ptrcr              ; dans pointeur trame à recevoir
movwf ptrct              ; et dans pointeur trame à traiter
movlw LOW(canbufin)      ; poids faible emplacement buffer CAN
movwf ptrcr+1            ; dans pointeur trame à recevoir
movwf ptrct+1            ; et dans pointeur trame à traiter
```

Puis à l'initialisation de notre timer de détection de trou de communication :

```
               ; initialiser timer0
               ; ------------------
movlw B'11000010'        ; timer0 en service, mode 8 bits, prédiviseur 8
movwf T0CON              ; dans registre de contrôle (205µs)
bcf   INTCON2,TMR0IP     ; interruption timer0 basse priorité
```

Après en avoir terminé, on peut enfin mettre nos interruptions générales en service avec priorités. On utilise notre macro pour mettre les interruptions en service dans l'ordre correct, afin d'éviter tout problème (recommandé par Microchip®).

```
                ; initialiser interruptions
                ; -------------------------
   bsf    RCON,IPEN          ; mise en service des priorités interruptions
   STARTINT                  ; lancer interruptions
```

Nous en arrivons à notre programme principal, réduit à un rôle d'aiguillage, comme c'est souvent le cas en ce qui me concerne :

```
;=============================================================================
;=============================================================================
;                            PROGRAMME PRINCIPAL                              =
;=============================================================================
;=============================================================================
;-----------------------------------------------------------------------------
; Dirige vers la sous-routine concernée en fonction des commandes à exécuter
;-----------------------------------------------------------------------------
main
   clrwdt                ; reset watchdog

                  ; gérer leds
                  ; ----------
   dcfsnz cmptledr,f          ; décrémenter compteur de réception
   bcf    LEDR                ; 0, éteindre led
   dcfsnz cmptlede,f          ; décrémenter compteur d'émission
   bcf    LEDE                ; 0, éteindre led
```

On commence par gérer les leds d'émission et de réception. On utilise un compteur pour être certain que les leds s'allument un temps suffisant pour bien éclairer.

```
                  ; gérer trames USART reçues
                  ; -------------------------
   tstfsz nbserin             ; tester s'il y a des trames USART à traiter
   rcall  traitserin          ; oui, traiter trames USART (réception ethernet)
```

Si une trame USART est stockée dans le buffer de réception, et non encore traitée, on la traite.

```
                  ; gérer trames USART à envoyer
                  ; ----------------------------
   btfsc  PIE1,TXIE       ; tester si transmission USART en cours
   bra    main2           ; oui, on continue d'émettre
   tstfsz nbserout        ; non, tester si au moins une trame USART à envoyer
   bsf    PIE1,TXIE       ; oui, relancer la transmission
```

Cette partie demande une petite explication. Si on n'a pas d'émission en cours (interruptions émissions USART validée), alors on teste si on a quelque chose à envoyer. Si oui, on lance les interruptions USART.

Dit à la façon de l'algèbre de Boole, on peut dire que :

On lance l'émission si : On n'a pas lancé d'émission Et qu'on a quelque chose à envoyer

Evidemment, on remarque immédiatement (pour les plus férus en logique) que ça revient à dire :

On lance l'émission si on a quelque chose à envoyer

C'est logique, et le premier test est donc inutile…. Sauf que…..

Sauf que si une interruption survient entre les deux dernières lignes, et que l'interruption en question annule l'octet nbserout (le dernier octet de la dernière trame vient d'être envoyé), on va remettre les transmissions en service alors même qu'on n'a plus rien à envoyer.

Donc, on devrait couper les interruptions avant le test et les remettre après. Mais si vous tentez la manœuvre, vous verrez que c'est bien moins efficace que de tester simplement si l'interruption est déjà en service. En effet, si l'interruption USART est déjà en service, c'est qu'on a toujours quelque chose à envoyer, et, dès lors, autant ne rien faire.

Moralité, on ne peut pas faire l'économie du premier test : lorsque des interruptions sont en jeu, il faut toujours se méfier des raccourcis de raisonnement, et surtout penser qu'entre deux opérations portant sur une variable modifiée par une interruption, la dite interruption peut avoir modifié sa valeur.

```
                  ; gérer trames CAN reçues
                  ; -----------------------
main2
   movf  ptrct,w         ; pointeur MSB trame traitée
   cpfseq ptrcr          ; comparer avec MSB trame reçue
   bra   main3           ; pas identiques, traiter
   movf  ptrct+1,w       ; pointeur LSB trame traitée
   cpfseq ptrcr+1        ; comparer avec LSB trame reçue
main3
   rcall traitcanin      ; pas identique, une trame à traiter
   bra   main            ; boucler
```

Ici, nous aiguillons simplement le traitement des trames CAN reçues, pour peu qu'il y en aie, ce qui se vérifie par l'inégalité (16 bits) entre le pointeur de trame reçue et le pointeur de trame traitée.

```
;=============================================================================
;=============================================================================
;                       GESTION DU TRAFIC USART                               =
;=============================================================================
;=============================================================================

;=============================================================================
;                            TRAITER TRAMES USART                             =
;=============================================================================
;-----------------------------------------------------------------------------
; Trames en provenance du PC, et arrivant sur l'USART du pic
; nbserin contient le nombre de trames présentes dans le buffer software
; numtserin contient le numéro de la trame à traiter
; bufserin est le buffer de réception USART (5 * 16 octets)
; Le buffer contient les octets suivants :
; Octet 0     : Commande
; Octet 1     : Identification de l'émetteur (0xFA)
; Octet 2     : Nombre d'octets utiles de la trame (De octet3 à octet 14)
; Octets 3/14 : octets utiles complétés éventuellement par des dummy (0x00)
; Octet 15    : checksum : déjà vérifié dans interruption inth
```

```
;-------------------------------------------------------------------------
traitserin
               ; vérifier si place pour répondre
               ; -----------------------------
   movf  nbserout,w      ; charger nombre de trames dans buffer émission
   xorlw 0x05            ; tester si plein
   btfsc STATUS,Z        ; reste de la place pour répondre?
   return                ; non, inutile de traiter pour l'instant

               ; pointer sur la commande reçue
               ; -----------------------------
   lfsr  FSR0,serbufin   ; pointer sur buffer de réception
   swapf numtserin,w     ; charger numéro trame à traiter *16
   addwf FSR0L           ; pointer sur la bonne trame

                  ; préparer la réponse
                  ; -------------------
   lfsr  FSR1,serbufout  ; pointer sur buffer d'émission
   swapf numeserout,w    ; prochain emplacement libre buffer de sortie * 16
   addwf FSR1L,f         ; pointer sur début emplacement trame
   movf  POSTINC0,w      ; charger commande reçue
   movff POSTINC0,recid  ; sauver ID de l'émetteur de la commande
   rcall traitcom60      ; traiter les commandes
```

Cette routine contient deux parties. Cette première portion prépare le traitement de la trame USART reçue, et vérifie qu'il y a de la place dans le buffer d'émission pour répondre à la commande. Sans quoi, inutile de traiter de suite (surtout utile pour la carte ethernet, mais j'ai conservé pour la carte RS232).

S'il y a de la place, on prépare les pointeurs puis on part vérifier de quelle commande il s'agit en réalité.

La seconde partie, que voici, exécute le traitement après exécution de la commande :

```
               ; pointer sur trame suivante
               ; --------------------------
   decf  nbserin,f       ; une trame traitée
   bsf   PIE1,RCIE       ; relancer éventuellement interruptions réception
   movlw -1              ; préparer rebouclage trame 4->0
   btfsc numtserin,2     ; tester si trame actuelle = 4
   movwf numtserin       ; oui, alors trame -1
   incf  numtserin,f     ; incrémenter numéro de trame
   return                ; fin de traitement
```

De fait, on accuse traitement de la trame, et on calcule le pointeur sur la prochaine trame à traiter.

Ensuite, vient le traitement des différentes commandes, dont pour commencer la commande 0x60 (envoi d'une trame CAN sur le bus)

```
;===============================================================================
;                        TRAITER COMMANDE 0x60                                  =
;===============================================================================
;-------------------------------------------------------------------------------
; 0x60 : Trame à envoyer sur le bus CAN
;        Octets de data : de 4 à 12 selon la trame CAN
;         D0 : obligatoire : SIDH = destinataire
;         D1 : obligatoire : SIDL = commande : si b7 = 1, trame remote
;         D2 : obligatoire : EIDH = cible
;         D3 : obligatoire : EIDL = paramètre
;         D4/D11 : facultatifs : DATA CAN D0 à D7
;        Réponse : commande 0x50 avec statut dans D0
;
; identificateur CAN sur 4 octets alignés à droite. Le bit 7 de
; indique s'il s'agit d'une trame request
; on envoie tout via le buffer 0, comme ça tout arrive dans l'ordre
;-------------------------------------------------------------------------------
traitcom60
                    ; tester commande
                    ; ---------------
   xorlw 0x60               ; commande 0x60 ?
   bnz   traitcom41         ; non, voir si commande 0x41

                    ; préparer début réponse
                    ; ----------------------
   movlw 0x50               ; accusé de réception
   movwf POSTINC1           ; dans buffer commande
   movff recid,POSTINC1     ; IP de l'émetteur dans réponse
   movlw 0x01               ; 1 octet de data (statut)
   movwf POSTINC1           ; dans buffer d'envoi

                    ; vérifier longueur
                    ; -----------------
   movlw 0x03               ; nombre de data minimum-1
   cpfsgtINDF0              ; tester si au moins 4 data
   bra   traitcom60err      ; non, erreur

                ; Attendre buffer d'émission CAN libre
                ; ------------------------------------
traitcom60wait
   clrwdt                   ; reset watchdog
   POINTTX0                 ; pointer sur buffer émission 0
   btfsc TXBxCON,TXREQ      ; tester si buffer libre
   bra   traitcom60wait     ; non, attendre

                ; initialiser registres d'émission
                ; --------------------------------
   movf  POSTINC0,w         ; charge nombre d'octets utiles
   addlw -4                 ; moins l'ID
   movwf TXBxDLC            ; = nombre d'octet de data CAN
   movff POSTINC0,TXBxSIDH  ; placer SIDH
   btfsc INDF0,7            ; tester si trame remote demandée
   bsf   TXBxDLC,TXRTR      ; oui, indiquer remote
   rcall cmd2sidl           ; convertir commande en valeur SIDL
   movff POSTINC0,TXBxSIDL  ; placer SIDL
   movff POSTINC0,TXBxEIDH  ; placer EIDH
   movff POSTINC0,TXBxEIDL  ; placer EIDL

                    ; placer les DATA CAN
                    ; -------------------
   movff POSTINC0,TXBxD0    ; on place même les data qui ne servent....
```

```
   movff POSTINC0,TXBxD1    ; ....à rien, c'est plus rapide qu'une....
   movff POSTINC0,TXBxD2    ; ... boucle logicielle
   movff POSTINC0,TXBxD3    ; D3
   movff POSTINC0,TXBxD4    ; D4
   movff POSTINC0,TXBxD5    ; D5
   movff POSTINC0,TXBxD6    ; D6
   movff POSTINC0,TXBxD7    ; et D7 pour finir

                  ; lancer l'émission CAN
                  ; ---------------------
   bsf   TXBxCON,TXREQ      ; requête d'émission CAN

                  ; placer accusé de réception
                  ; --------------------------
   clrf  POSTINC1           ; D0 = 0 (statut OK)
traintcom60a
   clrf  POSTINC1           ; D1 = 0 (dummy)
   clrf  POSTINC1           ; D2 = 0 (dummy)
   clrf  POSTINC1           ; D3 = 0 (dummy)
   clrf  POSTINC1           ; D4 = 0 (dummy)
   clrf  POSTINC1           ; D5 = 0 (dummy)
   clrf  POSTINC1           ; D6 = 0 (dummy)
   clrf  POSTINC1           ; D7 = 0 (dummy)
   clrf  POSTINC1           ; D8 = 0 (dummy)
   clrf  POSTINC1           ; D9 = 0 (dummy)
   clrf  POSTINC1           ; D10 = 0 (dummy)
   clrf  POSTINC1           ; D11 = 0 (dummy)
   bra   nextserout         ; pointer sur trame suivante
   return                   ; et fin de traitement

               ; traiter erreur
               ; --------------
traitcom60err
   movlw 0x01               ; code d'erreur
   movwf POSTINC1           ; dans D0 réponse
   bra   traintcom60a       ; terminer avec des dummies
```

Rien de remarquable ici, si ce n’est qu’on envoie la trame dans le buffer d’émission CAN TXB0, pointé par RXB0 grâce à la macro POINTTX0.

L’envoi dans toujours le même buffer garantit que les trames sont envoyées dans le bon ordre. Ceci implique évidemment de devoir attendre le cas échéant que le buffer en question soit libre, ce qui est fait ici. Ca ne pose de plus strictement aucun problème, puisque le PC synchronise ses émissions sur les accusés de réception reçus par la carte.

Les autres commandes ne posent pas de problème particulier, je ne vous les recopie pas ici, référez-vous au fichier source et aux explications sur les commandes.

Je vais juste encore montrer quelques routines particulières, dont celle-ci :

```
;=============================================================================
;                      CONVERTIR COMMANDE EN SIDL                             =
;=============================================================================
;-----------------------------------------------------------------------------
; Convertit un octet au format de commande en valeur de SIDL
; format d'une 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
;-----------------------------------------------------------------------------
cmd2sidl
   movf   INDF0,w         ; charger octet à modifier
   andlw  0x03            ; garder b17 et b16
   movwf  local04         ; 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   local04,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
```

Cette commande est la conséquence de l'organisation particulière des bits dans les registres SIDL (masques, filtres, buffers d'émission, buffers de réception).

Dans ces registres, les bits sont répartis de la façon suivante :

b7 : bit 20 de l'ID CAN
b6 : bit 19 de l'ID CAN
b5 : bit 18 de l'ID CAN
b4 : toujours 0
b3 : 1 pour trames de type extended (cas de Domocan)
b2 : toujours 0
b1 : bit 17 de l'ID CAN
b0 : bit 16 de l'ID CAN

Ceci tient au fait que le module CAN est prévu à la fois pour travailler en CAN avec ID sur 11 bits (on s'arrête au bit 18) qu'en mode 29 bits (bits 17 à 0 en supplément), d'où la séparation constatée.

Or, au niveau de domogest, et de nos commandes d'interface, pour plus de visibilité, ces registres représentant en fait pour nous la commande CAN (concept Domocan), sont organisés comme suit :

b7 : 1 si commande remote, 0 pour commande data (en principe, 0 pour Domocan)
b6 : 0
b5 : 0
b4 : bit 20 de l'ID CAN
b3 : bit 19 de l'ID CAN
b2 : bit 18 de l'ID CAN
b1 : bit 17 de l'ID CAN
b0 : bit 16 de l'ID CAN

Ceci explique que les commandes CAN varient, dans Domocan, de 0x00 à 0x1F (commandes 0 à 31)

Et bien, la petite routine ci-dessus effectue la conversion entre une commande Domocan reçue et la valeur à placer dans le registre SIDL correspondant

Et, réciproquement, la routine suivante :

```
;*****************************************************************************
;                      CONVERTIR SIDL EN COMMANDE                             =
;*****************************************************************************
;-----------------------------------------------------------------------------
; convertit un registre SIDL en un octet de commande
; format de SIDL :
;     ID b20, ID b19, ID b18, 0, Extended bit, 0, ID b17, ID b16
; format d'une commande :
;     0, 0, 0, ID b20, ID b19, ID b18, ID b17, ID b16
;
; L'octet à modifier est pointé par FSR1, en sortie FSR1 est incrémenté
;-----------------------------------------------------------------------------
sidl2cmd
   movf  INDF1,w         ; charger octet à modifier
   andlw 0x03            ; garder b17 et b16
   movwf local04         ; sauver 2 derniers bits
   movlw 0xE0            ; préparer masque
   andwf INDF1,f         ; garder b20,b19,b18
   rrncf INDF1,f         ; 0,b20,b19,b18,0,0,0,0
   rrncf INDF1,f         ; 0,0,b20,b19,b18,0,0,0
   rrncf INDF1,f         ; 0,0,0,b20,b19,b18,0,0
   movf  local04,w       ; charger bits 17 et 16
   iorwf POSTINC1,f      ; 0,0,0,b20,b19,b18,b17,b16
   return                ; et retour
```

Transforme une valeur tirée d'un registre SIDL en une commande Domocan directement exploitable.

La routine suivante, quant à elle, va s'occuper de programmer nos filtres et masques CAN, sachant qu'en fait, dans cette carte, nous n'en utiliseront qu'un seul de chaque, contrairement aux autres cartes de notre bus Domocan :

```
;==============================================================================
;                        PLACER LES FILTRES ET LES MASQUES                     =
;==============================================================================
;------------------------------------------------------------------------------
; Place le filtre et le masque 0 pour la réception CAN (seuls utilisés)
; CAN fonctionne en mode 2 (FIFO)
; Le module doit être en mode configuration
; Les valeurs de filtre et masque sont en eeprom
;------------------------------------------------------------------------------
setmasfil
                ; vérification EEPROM
                ; -------------------
   movlw FILT_EEP        ; pointer sur filtres et masques EEPROM
   rcall readeepw        ; lire premier octet de validation
   xorlw 'B'             ; comparer avec "B"
   bnz   setmasfildef    ; pas bon, valeurs par défaut
   rcall readeep         ; lire second octet de validation
   xorlw 'G'             ; comparer avec "G"
   bnz   setmasfildef    ; pas bon, valeurs par défaut
```

Le paramétrage s'effectue en fonction des valeurs présentes en eeprom. Si l'en-tête de zone de l'eeprom est invalide, alors on placera les valeurs par défaut, qui consistent à laisser passer toutes les trames CAN de type extended.

```
                ; placer masque 0
                ; ---------------
   lfsr  FSR2,RXM0SIDH   ; pointer sur masque de réception 0
   rcall readeep         ; lire SIDH masque 0 eeprom
   movwf POSTINC2        ; dans SIDH (destinataire)
   rcall readeep         ; lire SIDL masque 0
   movwf POSTINC2        ; dans SIDL (commande)
   rcall readeep         ; lire EIDH masque 0
   movwf POSTINC2        ; dans EIDH (cible)
   rcall readeep         ; lire EIDL masque 0
   movwf POSTINC2        ; dans EIDL (paramètre)
```

Cette partie de code initialise les différents registres du masque 0, qui est celui utilisé. L'utilisation du registre d'indirection FSR2 évite d'avoir à changer de banque (je n'aime pas ça).

```
                ; placer filtre 0
                ; ---------------
   lfsr  FSR2,RXF0SIDH   ; pointer sur filtre de réception 0
   rcall readeep         ; lire SIDH filtre 0
   movwf POSTINC2        ; dans SIDH (destinataire)
   rcall readeep         ; lire SIDL filtre 0
   movwf POSTINC2        ; dans SIDL (commande)
   rcall readeep         ; lire EIDH filtre 0
   movwf POSTINC2        ; dans EIDH (cible)
   rcall readeep         ; lire EIDL filtre 0
   movwf POSTINC2        ; dans EIDL (paramètre)
```

Et ensuite la même chose pour le filtre 0.

```
   lfsr  FSR2,MSEL0      ; pointer sur bon registre
   bcf   INDF2,FIL0_1    ; correspondance filtre/masque
   bcf   INDF2,FIL0_0    ; 0 0 = masque 0
```

Ici, une particularité des modes 1 et 2 que j'ai déjà effleurée, c'est qu'on peut établir la correspondance entre un filtre et un masque. Autrement dit, définir à quel masque est lié un filtre donné. Nous établirons la correspondance entre le filtre 0 et le masque 0 pour cette application. Notez que ceci nous donne bien plus de souplesse que dans le cas du mode 0 (legacy) ou les correspondances sont figées.

```
                ; mettre le filtre 0 en service
                ; ------------------------------
   lfsr   FSR2,RXFCON0    ; pointer sur registres de contrôle
   movlw  B'00000001'     ; filtre 0 = ON, 1 à 7 = OFF
   movwf  POSTINC2        ; dans registre RXFCON0
   clrf   INDF2           ; couper filtres 8 à 15 dans RXFCON1
   return                 ; et fin
```

Nous disposons en mode 2 de 16 filtres, au lieu des 3 de l'unique mode 0 de nos anciens 18Fxx8. Nous devons décider lesquels sont en service, ce qui est de nouveau une possibilité qui n'était pas présente. Nous choisissons évidemment de ne mettre en service que le filtre 0

```
                   ; valeurs par défaut
                   ; ------------------
setmasfildef
   bsf    ERREEP           ; signaler erreur eeprom
   lfsr   FSR2,RXM0SIDH    ; pointer sur masque 0 SIDH
   clrf   POSTINC2         ; tous les destinataires acceptés
   movlw  B'00001000'      ; toutes les commandes (extended) acceptées
   movwf  POSTINC2         ; dans masque SIDL
   clrf   POSTINC2         ; toutes les cibles acceptées
   clrf   INDF2            ; tous les paramètres accepés
   return                  ; fin du traitement, tout est autorisé
```

Enfin, en cas d'erreur eeprom, nous effaçons tous les bits du masque (excepté le bit extended), ce qui nous autorise à recevoir toute trame CAN quelle qu'elle soit.

Je vais maintenant aborder une dernière routine, responsable du placement de nos paramètres CAN. Etant donné que nos paramètres sont modifiables par software, il est impossible de les configurer par des directives ne s'exécutant qu'à l'assemblage (j'ai constaté que plusieurs internautes pensaient que les directives était une sorte de langage C utilisé par le PIC®), contrairement à nos autres cartes Domocan. Il nous faut donc calculer tout ça de façon dynamique, c'est le rôle de cette routine :

```
;*****************************************************************************
;                            PLACER LES PARAMETRES CAN                        *
;*****************************************************************************
;-----------------------------------------------------------------------------
; Paramètre le bus CAN en fonction des valeurs en eeprom
; le module CAN doit être en mode configuration
;-----------------------------------------------------------------------------
setbrgconx
                   ; utiliser paramètres EEPROM
                   ; --------------------------
   movlw  0x02             ; adresse du premier paramètre CAN en eeprom
   movwf  EEADR            ; pointer sur TQ
   rcall  readeep          ; lire nbre de 0.05µs de TQ
   decf   WREG,w           ; convertir de 0 à 63
   andlw  0x3F             ; conserver les bits utiles
   movwf  BRGCON1          ; placer dans registre de configuration
```

Ici, on lit la valeur de QT présente en eeprom (préalablement vérifiée), et on convertit cette valeur en valeur pour le registre BRGCON1.

```
rcall readeep         ; lire temps de propagation
decf  WREG,w          ; convertir de 0 à 7
andlw 0x07            ; conserver les bits utiles
movwf BRGCON2         ; placer dans registre de configuration
```

Ensuite, on fait de même avec le temps de propagation, placé dans le registre BRGCON2.

```
rcall readeep         ; lire phase segment 1
decf  WREG,w          ; convertir de 0 à 7
rlcf  WREG,w          ; PS1 en b3/b1
rlcf  WREG,w          ; PS1 en b4/b2
rlcf  WREG,w          ; PS1 en b5/b3
andlw 0x38            ; conserver les bits utiles
iorwf BRGCON2,f       ; ajouter au registre de configuration

rcall readeep         ; lire phase segment 2
decf  WREG,w          ; convertir de 0 à 7
andlw 0x07            ; conserver les bits utiles
movwf BRGCON3         ; placer dans registre de configuration
```

On procède de même pour les 2 phases segments, placés en BRGCON2 et 3
Puis, pour le SJW, ajouté, lui, dans les bons bits de BRGON1 :

```
rcall readeep         ; lire SJW
decf  WREG,w          ; convertir de 0 à 3
rrncf WREG,w          ; SJW en b0/b7
rrncf WREG,w          ; SJW en b7/b6
andlw 0xC0            ; conserver les bits utiles
iorwf BRGCON1,f       ; ajouter au registre de configuration
```

Ne reste qu'à terminer avec le sample, placé dans le bit SAM de BRGCON2 :

```
rcall readeep         ; lire SAMPL
btfsc WREG,0          ; tester si SAMPL=1
bsf   BRGCON2,SAM     ; oui, valider SAM
```

La dernière opération consiste à indiquer qu'on travaille avec une durée de phase segment 2 programmable, ce qui vous autorise toutes les possibilités dans ParCan.inc :

```
bsf   BRGCON2,SEG2PHTS   ; indiquer que phase segment 2 programmable
return                   ; fin de configuration
```

Rien de spécial dans cette routine, mais il était bon de vous montrer quels registres étaient utilisés. Attention, il y a encore d'autres registres utilisés dans le mode 2, mais cette application utilisant leur valeur par défaut à la mise sous tension, il n'a pas été utile de les reconfigurer. Méfiez-vous cependant si vous créez votre propre application, n'oubliez pas de lire le datasheet.

Ceci clôture l'analyse de notre fichier source.

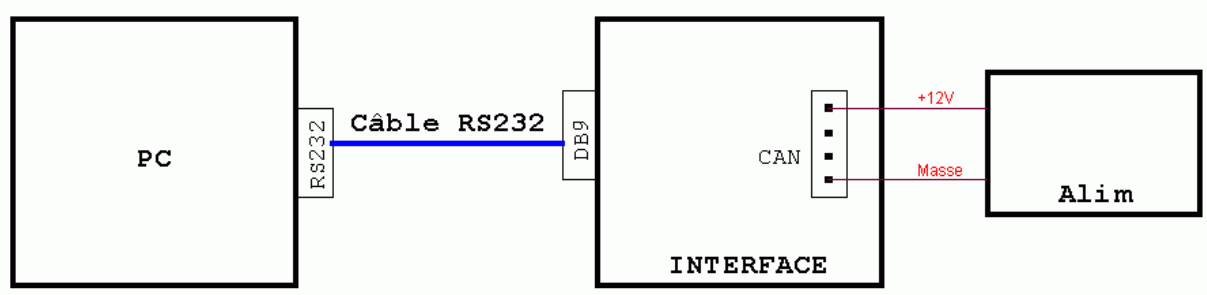
5. Mise en service

5.1 Connexion

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

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

Voici le schéma de connexion que vous devrez utiliser :



Je vous renvoie au mode d'emploi de Domocan pour savoir comment mettre en œuvre votre carte d'interface.

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 seront utilisés pour connecter votre bus Domocan à votre carte d'interface.

Evidemment, vous ne pourrez tester la partie communication de votre carte RS232 que lorsque vous aurez construit au moins une carte d'application Domocan.

Vous pouvez cependant vous assurer que votre carte répond déjà aux commandes de base. Rendez-vous pour ceci sur le mode d'emploi de Domogest.

Notes :

6. Utilisation du présent document

Cet ouvrage est destiné à être lu après les précédents documents du même projet disponibles sur mon site.

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 par émail.

Pour des raisons de facilité de maintenance, j'ai décidé que cet ouvrage serait disponible uniquement sur mon site : www.abcelectronique.com/bigonoff ou www.bigonoff.org

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/ou de l'application de l'ouvrage, des programmes, schémas, typons ou du matériel (liste non exhaustive).

Si vous travaillez sur la tension secteur, assurez-vous que vous avez les compétences nécessaires, et soyez prudents.

Comme pour toute réalisation personnelle, l'imposition de l'étiquette «CE » n'est pas obligatoire. Ceci ne vous dispense pas de devoir respecter la législation de votre pays au niveau des prescriptions électriques (RGIE par exemple), ainsi que des prescriptions concernant la compatibilité E.M. Il vous appartient de vérifier que votre système Domocan répond aux réglementations en vigueur.

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 des applications personnelles, n'hésitez pas à les faire partager par tous. Pour ce faire, vous pouvez me les envoyer.

Merci au webmaster de www.abcelectronique.com, pour son hébergement gratuit.

- Edition beta 1 terminée le 06/11/2003

- Révision 1 le 09/06/04 : Modification pour augmentation du buffer CAN : nouveau logiciel

- Révision 2 le 28/06/04 : Modification suite à la découverte d'un bug dans le PIC®. Nouveau logiciel.

- Révision 3 le 27/12/04 : Modifications suite à la découverte de 2 bugs supplémentaires dans les PIC®. Nouveau logiciel.

- Révision 4 le 26/02/06 : Modifications diverses suite à la refonte complète du système Domocan. Parmétrage possible complet de l'interface pour la rendre universelle.

- Révision 5 le 15/03/08 : Modifications profondes suite à la création de la carte d'interface CAN/Ethernet et de l'adoption d'un 18F2680 en mode 2 en place du 18F258 en mode 0, séparation des explications sur les trames dans un document séparé, analyse du logiciel en version 6.0

Réalisation : Bigonoff

Site : http://www.abcelectronique.com/bigonoff ou www.bigonoff.org
Email : bigocours@hotmail.com ou claude@bigonoff.org