Université Blaise Pascal

Polytech'Clermont-Ferrand

Note d'application

MISE EN ŒUVRE D'UNE COMMUNICATION PAR BUS CAN

REALISE PAR CHARLES LERY

Année universitaire : 2008/2009

Le CAN (Controller Area Network) est un bus de communication série développé à la fin des années 80 par l'entreprise allemande Robert Bosch. L'objectif était de fournir à l'industrie automobile un bus peu coûteux pour l'électronique embarquée des automobiles. Aujourd'hui, l'efficacité et la robustesse de ce protocole l'ont amené à être utilisé dans de nombreuses autres applications industrielles, en particulier celles nécessitant un débit important jusqu'à 1Mbits/s avec un très faible taux d'erreur.

De nombreux contrôleurs CAN sont aujourd'hui disponibles chez la plupart des fabricants, qui proposent aussi des versions de leurs microcontrôleurs avec des contrôleurs CAN intégrés.

Ce document présente un exemple d'implémentation d'un bus CAN reliant différentes plateformes : un microcontrôleur PIC18F458 disposant d'un contrôleur CAN, des modules MPPT (Maximum Power Point Tracking).

SOMMAIRE

INTRODUCTION

Le protocole de la communication par bus CAN est basé sur le principe de diffusion générale. En effet, lors de la transmission d'un message, aucune station (microcontrôleur et modules MPPT) n'est adressée en particulier, mais le contenu de chaque message est explicité par une identification reçue de façon univoque par tous les nœuds. Grâce à cet identificateur, ces derniers, qui sont en permanence à l'écoute du réseau, reconnaissent et traitent les messages qui les concernent. Cette présente note d'application décrira, de façon détaillée, dans une première partie la mise en œuvre d'une communication par bus CAN. En seconde partie, un exemple sera donné pour illustrer cette communication. Elle détaillera les échanges que peuvent avoir un microcontrôleur PIC18F458 avec des modules MPPT. Ces derniers permettent de récupérer l'image de la tension et du courant, délivrés par des panneaux solaires aux batteries, en servant d'intermédiaires. Par définition, ils permettent de faire fonctionner les panneaux solaires de façon à produire en permanence le maximum de leur puissance. Ainsi, quelles que soient les conditions météorologiques (températures et irradiation), et quelle que soit la tension de la batterie, la commande du convertisseur place le système au point de fonctionnement maximum. Ce point de fonctionnement obtenu correspond à une tension et un courant optimaux débités par les modules, appelés V_{opt} et I_{opt}.

1. PRINCIPE DE FONCTIONNEMENT

Du type multi-maître, orienté messages courts, le bus CAN est bien adapté à la scrutation de variables émises par des stations déportées. La norme Iso 11898 spécifie un débit maximum de 1Mbit/s. La longueur maximum du bus est déterminée par la charge capacitive et le débit. Les configurations recommandées sont les suivantes :

Débit	Longueur
1 Mbit/s	40 m
500 Kbit/s	100 m
100 Kbit/s	500 m
20 Kbit/s	1000 m

Figure : Débit de la communication par bus CAN en fonction de la distance

Le protocole est basé sur le principe de diffusion générale : lors de transmission, aucune station n'est adressée en particulier, mais le contenu de chaque message est explicité par une identification reçue de façon univoque par tous les abonnés. Grâce à cet identificateur, les stations, qui sont en permanence à l'écoute du réseau, reconnaissent et traitent les messages qui les concernent; elles ignorent simplement les autres. L'identificateur indique aussi la priorité du message, qui détermine l'assignation du bus lorsque plusieurs stations émettrices sont en concurrence. En version de base, c'est un nombre de 11 bits (CAN 2.0A), ce qui permet de définir jusqu'à 2048 messages plus ou moins prioritaires sur le réseau. Il existe une version étendue où le nombre de bits d'identifiants est de 29 (CAN 2.0B). Chaque message peut contenir jusqu'à 8 octets de données. Il est possible d'ajouter des stations réceptrices à un réseau CAN sans modifier la configuration des autres stations.

Format des trames

Les trames de données ou de requête sont composées de sept parties. Le format ci après est décrit pour des trames respectant le protocole 2.0A.

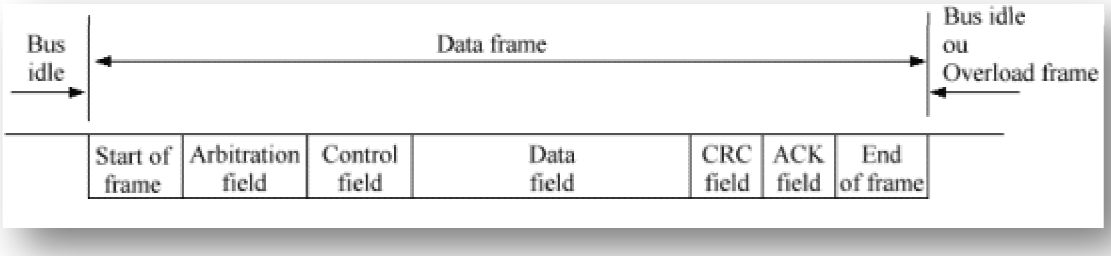


Figure : Composition d'une trame CAN

Start of frame marque le début d'une Data Frame (trame de données) ou d'une Remote frame (trame de requête). C'est un seul bit dominant. Il permet la synchronisation des autres nœuds avec celui qui émet une trame.

Arbitration field est composé des 11 bits de l'identificateur (CAN 2.0A) et d'un bit de RTR (Remote transmission Request) qui est dominant pour une trame de données, et récessif pour une trame de requête.

Control field est composé de 6 bits. Les 2 premiers sont des bits réservés et les 4 suivants constituent le Data Length Code (DLC). Le DLC indique le nombre d'octets du Data field. La valeur du DLC est forcément comprise entre 0 et 8, soit 9 valeurs. 4 bits dominants (0000) correspondent à la valeur 0 pour le DLC, tandis que 1 bit récessif et 3 bits dominants (1000) correspondent à la valeur 8.

Data field sont les données transmises par la Data frame. Il peut contenir de 0 à 8 octets, où chaque octet est transmis avec le bit de poids fort en premier.

CRC field est composé de la séquence de CRC sur 15 bits suivie du CRC delimiter (1 bit récessif). La séquence de CRC (Cyclic Redundancy Code) permet de vérifier l'intégrité des données transmises. Les bits utilisés dans le calcul du CRC sont ceux du SOF, de l'Arbitration field, du Control field et du Data field.

ACK field est composé de 2 bits, l'ACK Slot et l'ACK Delimiter (1 bit récessif). Le nœud en train de transmettre envoie un bit récessif pour l'ACK Slot. Un nœud ayant reçu correctement le message en informe le transmetteur en envoyant un bit dominant pendant le ACK Slot : il acquitte le message.

End of frame marque la fin des Data frame et Remote par une séquence de 7 bits récessifs.

Bit-stuffing : pour les Data Frames et les Remote Frames, les bits depuis le Start of frame jusqu'à la séquence de CRC sont codés selon la méthode du bit stuffing. Quand un transmetteur détecte 5 bits consécutifs de même valeur dans les bits à transmettre, il ajoute automatiquement un bit de valeur opposée.

Outre les trames de données et de requête, on a donc également des trames d'erreurs (émises par n'importe quel nœud dès la détection d'une erreur), et des trames de surcharge (ces trames correspondent à une demande d'un laps de temps entre les trames de données et de requêtes précédentes et successives.

Structure d'un réseau CAN

Pour envoyer ou recevoir un message à partir d'un microcontrôleur, il faut que les données transitent par différents modules. Ci-après, un schéma structurel d'un réseau CAN.

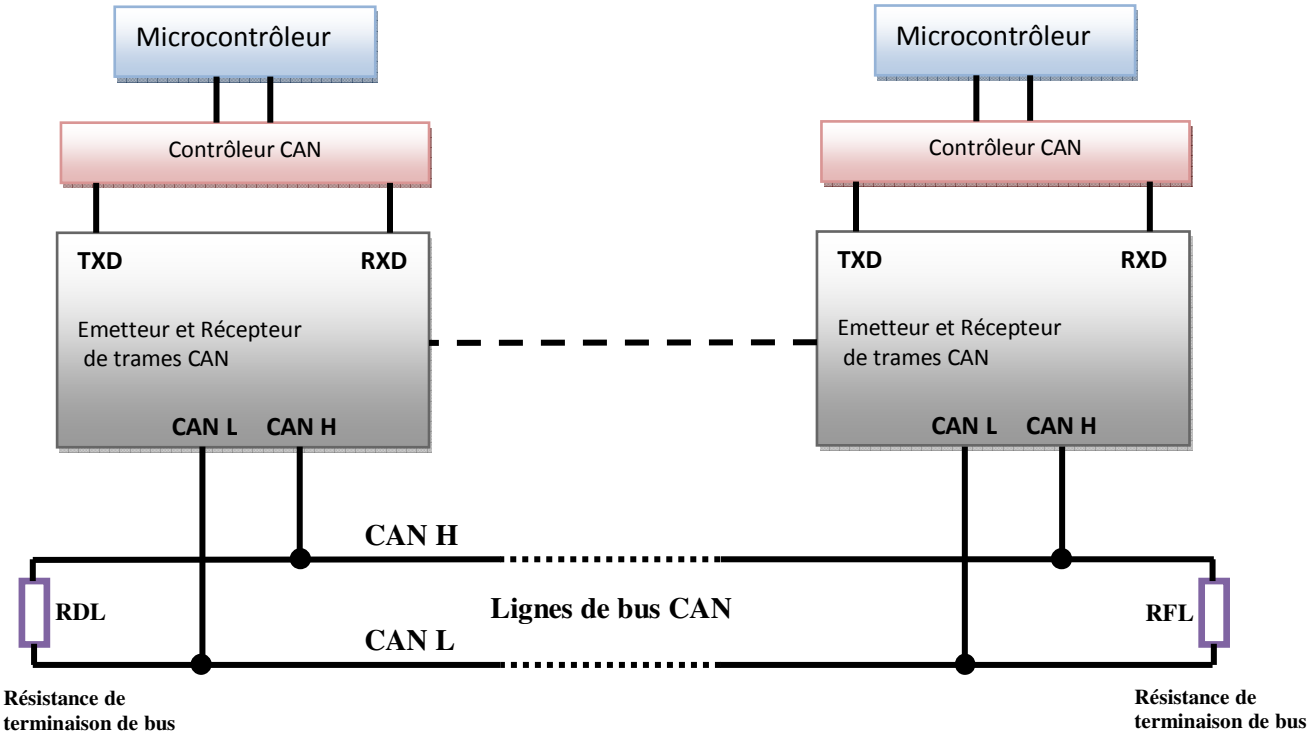


Figure : Structure d'un réseau CAN

Contrôleur CAN

Le bus CAN étant un bus complexe, il est difficile de le mettre en œuvre de manière totalement software. Les contrôleurs CAN permettent de gérer le bus de manière hardware.

On distingue deux types de contrôleurs :

- Les contrôleurs reliés aux microcontrôleurs par une liaison de type SPI , comme le MCP2515 de chez Microchip
- Les contrôleurs intégrés aux microcontrôleurs qui disposent donc, d'un module CAN avec les lignes TXCAN et RXCAN reportés sur les pins et destinés à êtres reliés au transceiver CAN.

Transceiver CAN

Le protocole CAN ne spécifie pas la couche physique, c'est pourquoi la plupart des contrôleurs CAN ne possèdent pas de circuits permettant de les connecter à un bus, qu'il soit filaire, à fibre optique ou tout autre mode de transmission possible. Un transceiver, tel que le 82C250 de Philips, permet de faire l'interface entre le contrôleur CAN et le bus physique.

Le transceiver 82C250 code les bits DOMINANTS et récessifs de la manière suivante :

- Etat DOMINANT (TX=0) CANH - CANL = 2V
- Etat RECESSIF (TX=1) CANH - CANL = 0V

Lors de la transmission d'un bit RECESSIF, le 82C250 pilote la paire différentielle avec une forte impédance, laissant ainsi la possibilité à tout autre nœud d'imposer une tension différentielle de 2V, correspondant à un état dominant.

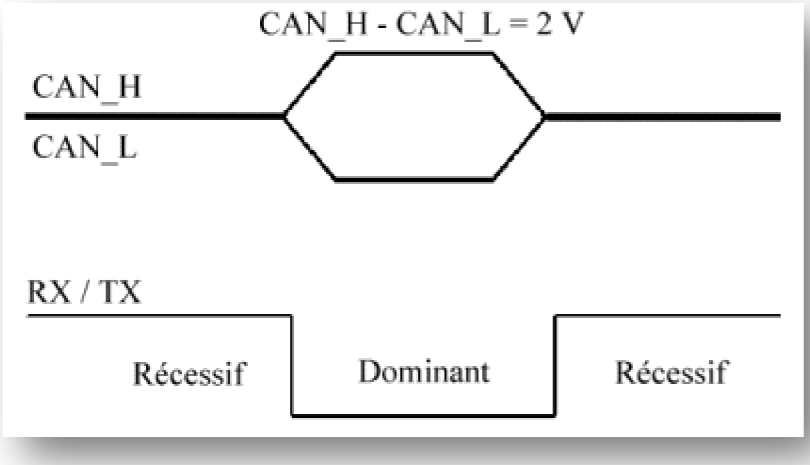


Figure : Etats de la paire différentielle d'un bus CAN

Remarque : l'état d'un bus est DOMINANT si un nœud transmet un bit DOMINANT. Il devient RECESSIF lorsque tous les nœuds transmettent un bit RECESSIF.

Résistance de terminaison de bus

Les résistances de terminaison de bus sont indispensables au bon fonctionnement de la communication. Elles sont composées d'une résistance de début de ligne (RDL) et d'une résistance de fin de ligne (RFL). Elles doivent être égales et on pourra les faire varier en fonction des spécifications de la communication que l'on souhaite obtenir.

Bus Length	Bus Cable		Termination Resistance	Maximum Baudrate
	Length-Related Resistance	Bus-Line Cross-Section		
0 … 40 m	70 mΩ/m	0.25 mm^2 … 0.34 mm^2 AWG23, AWG22	124 Ω (1%)	1 Mbit/s at 40 m
40 … 300 m	< 60 mΩ/m	0.34 mm^2 … 0.6 mm^2 AWG22, AWG20	127 Ω (1%)	500 kbit/s at 100 m
300 … 600 m	< 40 mΩ/m	0.5 mm^2 … 0.6 mm^2 AWG20	150…300 Ω	100 kbit/s at 500 m
600 m … 1 km	< 26 mΩ/m	0.75 mm^2 … 0.8 mm^2 AWG18	150…300 Ω	50 kbit/s at 1 km

Figure : Tableau des résistances de fin de ligne en fonction de la distance

2. APPLICATION

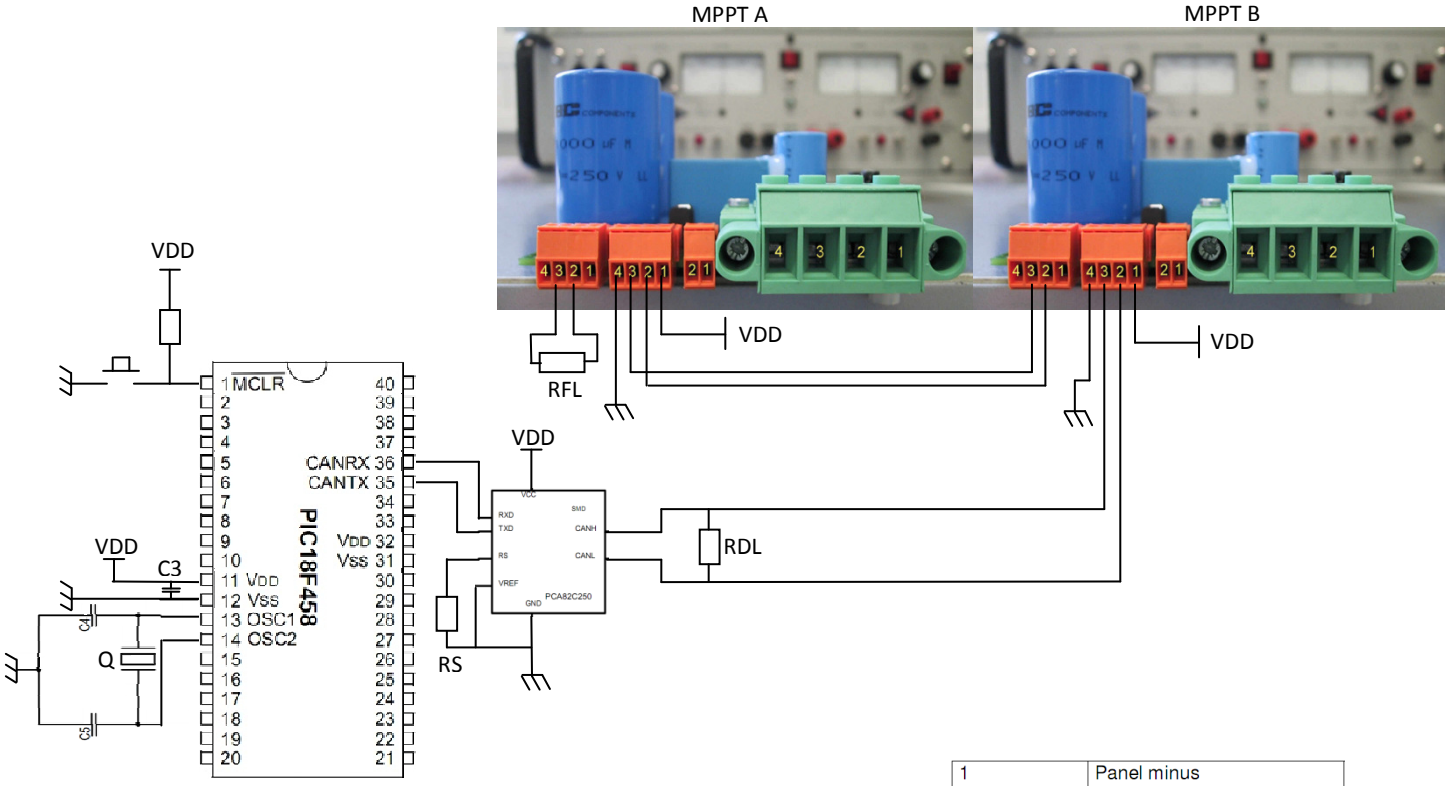
Description

Comme indiqué dans l'introduction, l'application est un système permettant à un microcontrôleur PIC18F458 de récupérer les images de la tension et du courant fournis par des panneaux solaires. Les panneaux étant au préalable raccordés à deux modules MPPT, le microcontrôleur récupère les informations sur ceux-ci via une communication par bus CAN.

Matériel nécessaire

Pour réaliser cette communication le choix s'est porté sur l'utilisation d'un PIC 18F458 pour ses spécificités techniques. Ce microcontrôleur dispose d'un contrôleur CAN intégré, par conséquent, moins de raccordements, moins de communications entre modules, donc moins de sources de problèmes lors de l'implémentation. Le transceiver utilisé est un PCA82C250 de chez Philips. Celui-ci permet de faire le lien entre le contrôleur de CAN et le bus physique.

Schéma de raccordement



1	Panel minus
2	Panel Plus
3	Battery +
4	Battery -
1	CAN In Ground****
2	CAN In Low****
3	CAN In High****
4	CAN In Supply Voltage****
1	CAN Out Ground****
2	CAN Out Low****
3	CAN Out High****
4	CAN Out Supply Voltage****
1	Shutdown +***
2	Shutdown -***

Tableau de raccordement Module MPPT

Avec :

C3 : condensateur de découplage 100 nF
C4 : condensateur du quartz 22pF
C5 : condensateur du quartz 22pF
RS : Résistance 470 Ω pour sélection grande vitesse
RDL : Résistance de début de ligne 124 Ω
RFL : Résistance de fin de ligne 124 Ω
VDD : tension d'alimentation 5 V
Q : Quartz de 4 MHz

Programmation logiciel

Plusieurs routines sont nécessaire afin de réaliser une communication par bus CAN. Ces routines se trouvent dans les fichiers can-18xxx8.c et can-18xxx8.h qui eux-mêmes sont dans les drivers du compilateur CCS C Compiler.

Seulement les routines principales sont décrites ci-après mais toutes les fonctions apparaissent annexes.

Routine *can_init()*

Cette routine initialise le périphérique CAN du PIC18F458. Elle configure la vitesse de transmission en appelant la routine can_set_baud().Elle configure également les filtres et les masques des buffers de réception. Ces derniers permettent de déterminer si un message reçu par le périphérique doit être transféré dans un buffer de réception. Lorsqu'un message est reçu par ce périphérique, le champ «Identifiant» du message est comparé avec les valeurs des filtres. En cas de correspondance, le message sera transféré dans un buffer de réception. Les masques associés aux filtres spécifient quels sont les bits de l'identifiant à examiner.

La table suivante montre la technique de comparaison utilisée ainsi que l'action correspondante :

Mask bit n	Filter bit n	Message Identifier bit n001	Accept or Reject bit n
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

Figure : Tableau expliquant le fonctionnement du filtre et du masque d'un périphérique CAN

Concernant la communication avec les modules MPPT, il faut déterminer les identifiants des messages qu'ils vont envoyer et de ce fait qui vont être reçus par le PIC18F458. L'identifiant est configurable selon la façon suivante :

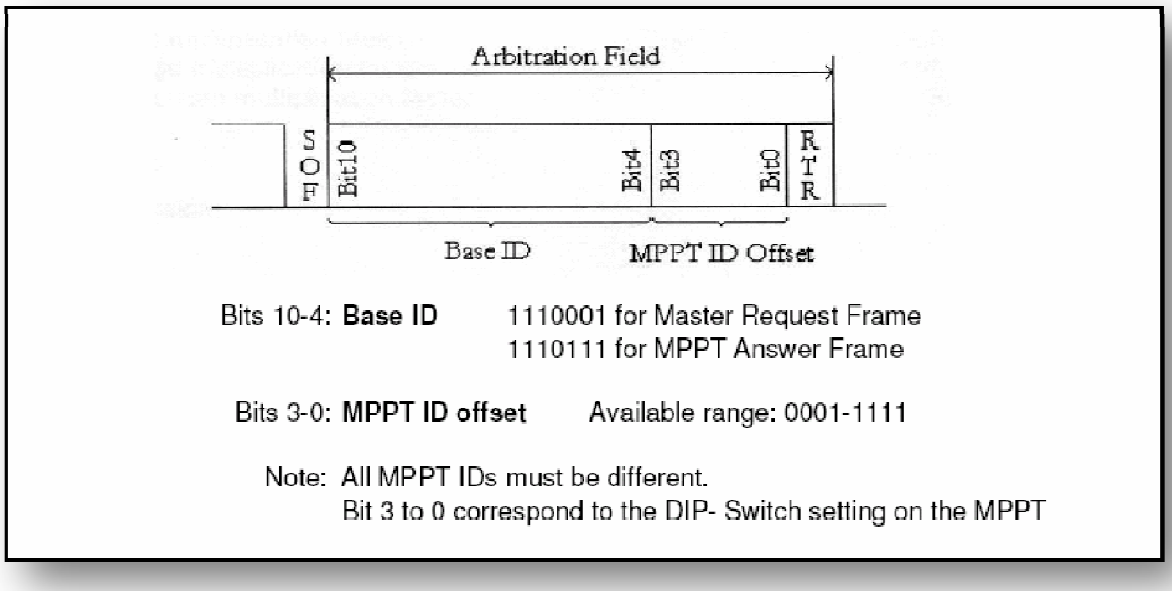


Figure : Fonctionnement des identifiants des modules MPPT

Ces modules fonctionnent en CAN 2.0A et les quatre derniers bits de l'identifiant sont configurables par un système de cavalier se trouvant sur les modules. De ce fait il serait possible de connecter 15 modules MPPT de ce type sur un même bus. Choisissons arbitrairement que :

- l'identifiant des trames de requêtes destinées au MPPTA est 11100010001 soit 0x711
- l'identifiant des trames de données provenant du MPPTA est 11101110001 soit 0x771
- l'identifiant des trames de requêtes destinées au MPPTB est 11100010010 soit 0x712
- l'identifiant des trames de données provenant du MPPTB est 11101110010 soit 0x772

Cette routine configure également le type de message que les buffers de réception doivent accepter, à savoir :

- tous les messages (CAN_RX_ALL)
- les messages valides uniquement (CAN_RX_VALID)
- seulement les messages valides avec des identifiants standards (CAN_RX_STD)
- seulement les messages valides avec des identifiants étendus (CAN_RX_EXT)

Pour l'application, les trames qu'envoient les modules MPPT sont standards, par conséquent on se place dans le troisième cas.

Les bits du port B qui sont concernés par le périphérique CAN, sont paramétrés également dans cette fonction :

- bit 3 du port B configuré en entrée (pin réception RX)
- bit 2 du port B configuré en sortie (pin émission TX)

Pour finir, cette routine place le périphérique CAN suivant 6 modes différents :

- Configuration Mode
- Disable mode
- Normal Operation mode
- Listen Only mode
- Loopback mode
- Error Recognition mode

Par défaut, le périphérique est placé en mode « *Normal* »

Remarque : Il ne faut pas oublier au préalable de se placer en mode « *Configuration Mode* » avant d'effectuer les paramétrages de la routine car sinon ils ne seront pas pris en compte.

Routine *can_set_baud()*

Tous les nœuds situés sur un bus CAN doivent être configurés pour le même débit. Le protocole CAN met en place un code NRZ sans codage d'horloge dans la trame. Il faut donc que les nœuds soient correctement configurés. Le débit nominal maximum est de 1 Mbps. Pour notre application nous nous plaçons à un débit nominal de 125kbits/s étant donné que les modules MPPT ont été configurés en usine à cette vitesse. A partir de ce débit, on définit le «Bit time» nominal :

$$T_{BIT} = \frac{1}{Débit\ nominal} = \frac{1}{125\ 000} = 8\ \mu s$$

Le *bit time* nominal doit être pensé comme étant divisé en « segments temporels » distincts et successifs.
On distingue 4 segments temporels :

- Le segment de synchronisation : SYNC_SEG
- Le segment « temps de propagation » : PROP_SEG
- Le segment « Phase buffer 1 » : PHASE_SEG1
- Le segment « Phase buffer 2 » : PHASE_SEG2

Ces segments temporels (et donc le T_{BIT} nominal) sont formés à partir d'unités de temps entières appelé «Time Quanta» (Quantum temporel) : T_Q

Par définition, le bit time nominal est programmable entre un minimum de 8.TQ et un maximum de 25.TQ. La relation suivante est utilisée pour calculer TQ :

$$T_{BIT} = TQ * (SYNC_{SEG} + PROP_{SEG} + PHASE_{SEG1} + PHASE_{SEG2})$$

SYN_SEG permet de synchroniser les différents nœuds sur le bus. La durée du segment de synchronisation est de 1 TQ.
PROP_SEG doit compenser les temps propagation physiques sur le support. Il s'agit de programmer des délais internes aux nœuds.

PHASE_SEG1 et **PHASE_SEG2** permettent de déterminer précisément la position du point d'acquisition du bit reçu. Ce point est situé entre les deux segments. Ceux-ci peuvent être réduits ou agrandis par le processus de resynchronisation.
La durée de ces segments est programmable de 1 à 8.TQ.

Le quantum temporel est programmable par un pré-diviseur (Baud Rate Prescaler) dont la valeur varie entre 1 et 64. La relation mathématique liant toutes ces valeurs est la suivante :

$$TQ(\mu s) = \frac{(2 * (BRP + 1))}{FOSC\ (MHz)}$$

De ce fait, en choisissant 8.TQ, nous obtenons un TQ de 1µs. Il en découle un BRP de 1.
Quelques contraintes sont à prendre en compte pour la programmation des segments temporels :

- PROP_SEG + PHASE_SEG1 ≥ PHASE_SEG2
- PHASE_SEG2 ≥ SJW

Pour l'application, la répartition des TQ que nous avons choisit est la suivante :

SYN_SEG = 1
PROP_SEG = 1
PHASE_SEG1 = 3
PHASE_SEG2 = 3

Remarque : Pour le changement de valeurs des segments, se reporter au fichier can-18xxx8.h

Routine *can_putd (id, data, len, priority, ext, rtr)*

Cette routine met dans un buffer de réception, les informations du message que l'on souhaite envoyer. Le message sera automatiquement envoyé lorsque le medium de communication sera disponible.

Les paramètres que l'on doit renseigner à l'appel de la fonction sont :

- id : identifiant de la trame de données
- data : pointeur sur le premier octet de données à envoyer
- len : le nombre d'octets de données que l'on souhaite envoyer
- priority : priorité du message à transmettre. Plus le chiffre est grand et plus prioritaire le message sera. Le chiffre doit être compris entre 0 et 3.
- ext : choisir TRUE lorsqu'on utilise un identifiant étendu, FALSE sinon.
- rtr : choisir TRUE pour envoyer une trame de requête, FALSE sinon

La fonction renvoie TRUE s'il n'y a pas eu de problème dans l'exécution de la fonction. Sinon elle renvoie FALSE et cela signifie que les informations du message n'ont pas été placées dans un buffer de transmission.

L'algorithme de la fonction peut s'écrire de la façon suivante :

Si un buffer d'émission est libre, autrement dit si l'on a la place pour envoyer un message :
- *Placer dans ce buffer :*
  - *L'identifiant de l'émetteur avec le type de message (requête ou données)*
  - *La priorité du message à envoyer*
  - *L'indication sur la quantité d'octet de données que l'on envoie (jusqu'à huit)*
  - *Le mode de communication que l'on utilise (standard ou étendu)*
  - *Les données que l'on veut envoyer*

Envoyer la trame lorsque le bus est libre

Routine *can_getd(id, data, len, stat)*

La fonction extrait les données d'un buffer de réception si des données sont présentes dans ce buffer, autrement dit, si un buffer de réception contient un nouveau message.

Elle renvoie :
- id : identifiant du message reçu
- data : pointeur sur le premier octet des données reçu si le message est du type données
- len : longueur des données reçues s'il y a données
- stat : caractéristiques des informations reçues (numéro du buffer de réception plein, indication si la trame est étendue ou standard, etc...)

La fonction retourne TRUE si des données se trouvaient dans un buffer de réception. Sinon la fonction renvoie FALSE et cela signifie qu'aucun message n'a été reçu.

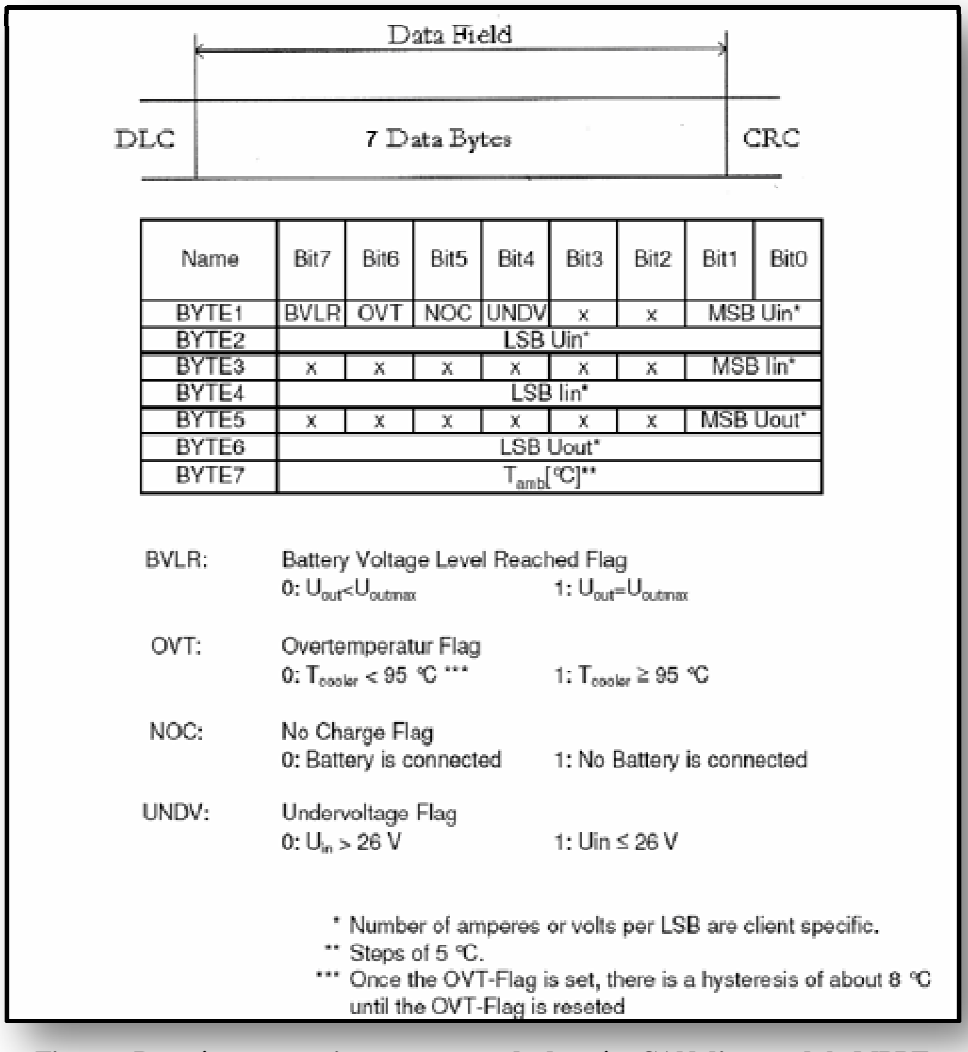
L'algorithme de la fonction peut s'écrire de la façon suivante :

Si un buffer de réception est plein, autrement si l'on a reçu un message :
- *Recherche du buffer de réception plein pour en extraire :*
  - *Le mode de communication que l'on utilise (standard ou étendu)*
  - *L'identifiant du message avec son type (requête ou données)*
  - *L'indication sur la quantité d'octets de données reçue (jusqu'à huit)*
  - *Les données qui se trouvent dans le message*
- *Remise à zéro des flags qui ont permis la détection d'un buffer plein*

Structure du programme principal

Les routines principales pour la communication CAN étant définies pour un PIC18F458, le programme principal peut désormais être créé.

Pour recevoir des informations provenant des modules MPPT, il faut d'abord en faire la demande et c'est pourquoi le microcontrôleur doit envoyer des trames de requête, destinées à chaque module. Ces modules vont traiter la demande et renverront par la suite une trame de données qui contiendra les informations sous la forme suivante :



Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
BYTE1	BVLR	OVT	NOC	UNDV	x	x	MSB Uin*	
BYTE2	LSB Uin*							
BYTE3	x	x	x	x	x	x	MSB Iin*	
BYTE4	LSB Iin*							
BYTE5	x	x	x	x	x	x	MSB Uout*	
BYTE6	LSB Uout*							
BYTE7	T_{amb}[°C]**							

BVLR: Battery Voltage Level Reached Flag
0: $U_{out}<U_{outmax}$ 1: $U_{out}=U_{outmax}$

OVT: Overtemperatur Flag
0: T_{cooler} < 95 °C *** 1: $T_{cooler} \geq 95$ °C

NOC: No Charge Flag
0: Battery is connected 1: No Battery is connected

UNDV: Undervoltage Flag
0: $U_{in} > 26$ V 1: Uin ≤ 26 V

* Number of amperes or volts per LSB are client specific.
** Steps of 5 °C.
*** Once the OVT-Flag is set, there is a hysteresis of about 8 °C until the OVT-Flag is reseted

Figure : Données que contient une trame de données CAN d'un module MPPT

Les informations que le microcontrôleur récupère sont le courant Iin et la tension Uout. Ces données sont codées sur 10 bits, par conséquent le microcontrôleur doit les extraire des 7 octets qu'il récupère de chaque module MPPT.

Pour la réception des trames de données, le PIC18F458 peut procéder suivant deux façons différentes. Il peur soit scruter les buffers de réception, ou soit être interrompu lorsqu'un

buffer de réception contient un message. Pour cette application, la première méthode est utilisée.

L'algorithme du programme principal peut s'écrire de la façon suivante :

Toutes les 100 ms :

- *Envoi d'une trame de requête au premier module MPPT*
- *Envoi d'une trame de requête au deuxième module MPPT*
- *Si un buffer de réception est plein :*
    - *Réception d'une trame en détectant si elle provient du module MPPT A ou B*
    - *Décodage des données pour en extraire les informations utiles, à savoir le courant et la tension*
    - *Calcul de la puissance de la partie des panneaux solaires dont les informations ont été reçues*

Pour utiliser les fonctions CAN décrites ci-dessus, procéder aux étapes suivantes :
1. Copiez les fichiers *can-18xxx8.c* et *can-18xxx8.h* dans le répertoire source de votre projet
2. Inclure le fichier *can18xx8.h* dans votre projet comme un fichier librairie 'H'
3. Ajouter le *#include can-18xx8.c* au début de chaque fichier *C* appelant les routines CAN

Programme principal pour la réception d'informations provenant de deux modules MPPT :

```
#include <18F458.h>
#include <can-18xxx8.c>

#fuses XT,NOPROTECT,NOLVP,NOWDT
#use delay(clock=4000000)


/******Declaration variables globales acquisition MPPT*****/
#define ID_ANSWER_MPPT_A    0x771
#define ID_ANSWER_MPPT_B    0x772
#define ID_REQUEST_MPPT_A   0x711
#define ID_REQUEST_MPPT_B   0x712

//declaration des variables qui servent a la reception d'une trame
struct rx_stat rxstat;
int32 rx_id;
int rx_len;
int buffer[8]={0,0,0,0,0,0,0,0};

float   tensionMPPT_A;
float   courantMPPT_A;
float   tensionMPPT_B;
float   courantMPPT_B;
float   calib_tens_MPPT=0.20879;
float   calib_cour_MPPT=0.00586;
float   powerMPPT_A=0.0;
float   powerMPPT_B=0.0;

int16   cour;
int16   tens;

//Declaration des variables pour l'emmission d'une trame
//de requete vers les modules MPPT
int buffem[8]={0,0,0,0,0,0,0,0};
/*********************************************************/

int16 msl;          //variable permettant la temporisation
                    //de l'envoi des trames de raquetes aux
                    //modules MPPT ainsi que la reception

/*********Declaration des routines d'interruption**********/
//interruption du timer 2 permettant de creer une temporisation
#int_timer2
void isr_timer2(void) {
    msl++; //incremantation de msl toutes milisecondes
}
/*********************************************************/
```

```
void main(void)
{

    //initialisation du timer 2 pour qu'il s'interrompe
    //toutes les millisecondes avec un quartz de 4Mhz
    setup_timer_2(T2_DIV_BY_1,250,2);

    //autorisation de l'interruption du timer 2
    enable_interrupts(INT_TIMER2);
    //autorisation de toutes les interruptions
    enable_interrupts (GLOBAL);

    can_init(); //initialisation du peripherique CAN

    while(1)
    {


        //envoie des trames de requetes vers les modules MPPT
        //toutes les 100ms
        if(msl > 100)
        {
        //recherche d'un buffer de transmission libre
        if ( can_tbe())
            {
                msl=0;//reinitialisation de la variable tempo

                //envoi trame de requete au MPPTA
                can_putd(ID_REQUEST_MPPT_A,buffem,8,0,FALSE,TRUE);

                //envoi trame de requete au MPPTB
                can_putd(ID_REQUEST_MPPT_B,buffem,8,0,FALSE,TRUE);
            }

        //on receptionne les trames provenants des modules MPPT
        //si l'un des deux buffers de reception est plein
        if ( can_kbhit() )
        {
            //on regarde si l'on receptionne les donnees du MPPT A
            if(can_getd(rx_id, buffer, rx_len, rxstat)){
                if (rx_id == ID_ANSWER_MPPT_A) {

                    //reception de la tension du module MPPT A
                    tens=buffer[4] & 0x03;
                    tens = tens << 8;
                    tens|=buffer[5];
                    tensionMPPT_A=tens*calib_tens_MPPT;

                    //reception du courant du module MPPT A
                    cour=buffer[2] & 0x03;
                    cour = cour << 8;
                    cour|=buffer[3];
                    courantMPPT_A=cour*calib_cour_MPPT;

                    //calcul de la puissance fournie par le module MPPT A
                    powerMPPT_A=tensionMPPT_A*courantMPPT_A;
                }
```

```
                //on regarde si l'on receptionne les donnees du MPPT B
                else if (rx_id == ID_ANSWER_MPPT_B) {
                    //reception de la tension du module MPPT B
                    tens=buffer[4] & 0x03;
                    tens = tens << 8;
                    tens|=buffer[5];
                    tensionMPPT_B=tens*calib_tens_MPPT;

                    //reception du courant du module MPPT B
                    cour=buffer[2] & 0x03;
                    cour = cour << 8;
                    cour|=buffer[3];
                    courantMPPT_B=cour*calib_cour_MPPT;

                    //calcul de la puissance fournie par le module MPPT B
                    powerMPPT_B=tensionMPPT_B*courantMPPT_B;
                }

            // on autorise a nouveau la reception de trame dans le buffer 0
            RXB0CON.rxful=0;

            // on autorise a nouveau la reception de trame dans le buffer 1
            RXB1CON.rxful=0;
            }
        }
        }
    }
}
```

3. Résultats

L'application fonctionne correctement avec deux modules MPPT. Mais son avantage est qu'elle est évolutive puisque l'on peut facilement rajouter d'autres nœuds sans grande difficulté.

4. Conclusion

L'implémentation d'une communication par bus CAN doit être réalisée progressivement sans sauter d'étapes car lors du débuggage, il peut y avoir énormément de raisons pour lesquels la communication de fonctionne pas.

5. Annexes

Fichier can18xxx8.c

Fichier can18xxx8.h