Institut d'Électronique du Sud
(Unité mixte de Recherche 5214 CNRS-UMII)
Université Montpellier II, CC 083
F-34095 Montpellier Cédex 5, France

Rapport carte contrôleur

Projet ROBUSTA

-

Polytech'Montpellier

Version 1

Institut d'Électronique du Sud
(Unité mixte de Recherche 5214 CNRS-UMII)
Université Montpellier II, CC 083
F-34095 Montpellier Cédex 5, France

PAGE D'ANALYSE DOCUMENTAIRE

TITRE : Polytech'Montpellier

MOTS CLES: Carte contrôleur.

RESUME : Rapport sur le travail effectué.

COMPOSANTE : UM2-Polytech

AUTEUR(S) : BOCH Jérôme à partir du rapport PIFE de Xavier CADOL et Stéphanie PEREZ.

COORDONNEES: boch@ies.univ-montp2.fr

DATE DE CREATION : 14/04/2008.

DATE DE LA DERNIERE MISE A JOUR : 07/05/2008

STATUT : NON DEFINITIF.

Institut d'Électronique du Sud
(Unité mixte de Recherche 5214 CNRS-UMII)
Université Montpellier II, CC 083
F-34095 Montpellier Cédex 5, France

Institut d'Électronique du Sud
(Unité mixte de Recherche 5214 CNRS-UMII)
Université Montpellier II, CC 083
F-34095 Montpellier Cédex 5, France

1. Premier prototype

Une platine a été réalisée pour simuler la communication entre les cartes avec un bus CAN. Nous allons donc expliquer le fonctionnement de cette platine ainsi que justifier le choix des composants.

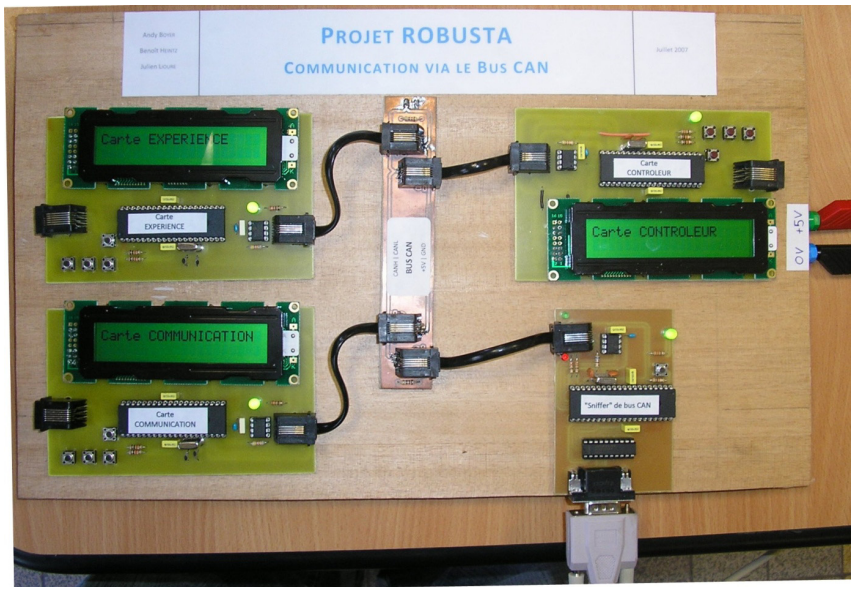


Figure 1 : Platine de travail

1.1. Description de la platine de simulation

La platine est composée principalement de trois cartes qui représentent chacune la carte contrôleur, communication et expérience. Ces cartes communiquent à l'aide d'un bus CAN. La quatrième carte s'appelle « sniffer » et son rôle est de lire les trames sur le bus CAN et de les afficher en temps réel sur une interface PC. Cette carte n'est bien sur qu'une façon de pouvoir contrôler notre travail et sert de test et donc ne sera pas utilisée lors de la création du satellite final. Toutefois son rôle est fondamental à la mise en place du satellite. Nous reviendrons sur son fonctionnement dans un chapitre suivant.
Les trois cartes principales sont équipées d'un PIC 18F458, d'un transceiver MCP 2551 et d'un écran LCD. L'écran comme la carte sniffer ne servent que de contrôle durant la phase de développement et ne seront pas intégré au satellite final.

Le PIC 18F458 a été retenu car certaines de ses sorties sont adaptées au bus CAN. Il dispose aussi d'assez de convertisseurs analogique/numérique nécessaires à la réalisation des expériences de la carte expérience.
De plus il intègre des fonctions intéressantes pour notre travail tels que le Sleepmode (mise en veille des cartes pour diminuer la consommation) et le Watch DOG Timer (évite au programme de tourner en boucle infinie).

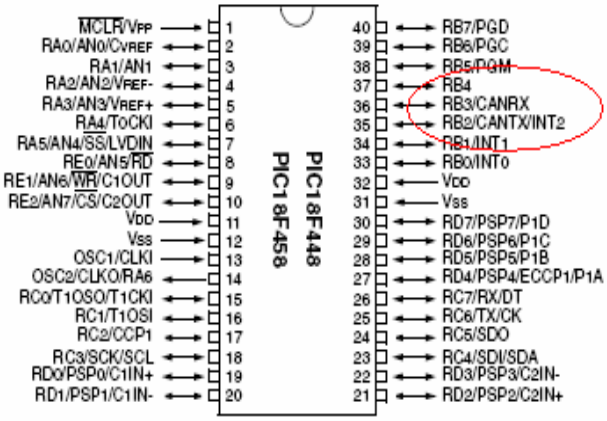


Figure 3 : PIC 18F458

Pour les essais, nous travaillons avec un 18F458 mais on utilisera au final pour le satellite ROBUSTA un PIC de type 18C… car ils sont non reprogrammables ,cela entraîne une robustesse supplémentaire de ces PIC aux conditions spatiales. Les solutions possibles sont un 18C658 (68 broches) ou un 18C858 (84 broches).

Bien que le PIC utilise des sorties adaptées au bus CAN, nous sommes obligés d'ajouter un transceiver MCP 2551 entre le PIC et le bus. Ce composant permet une adaptation électrique et a été retenu car son fabriquant est le même que celui du microcontrôleur.

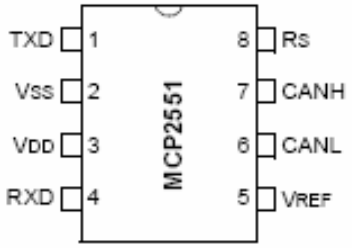


Figure 4 : Transceiver

Institut d'Électronique du Sud
(Unité mixte de Recherche 5214 CNRS-UMII)
Université Montpellier II, CC 083
F-34095 Montpellier Cédex 5, France

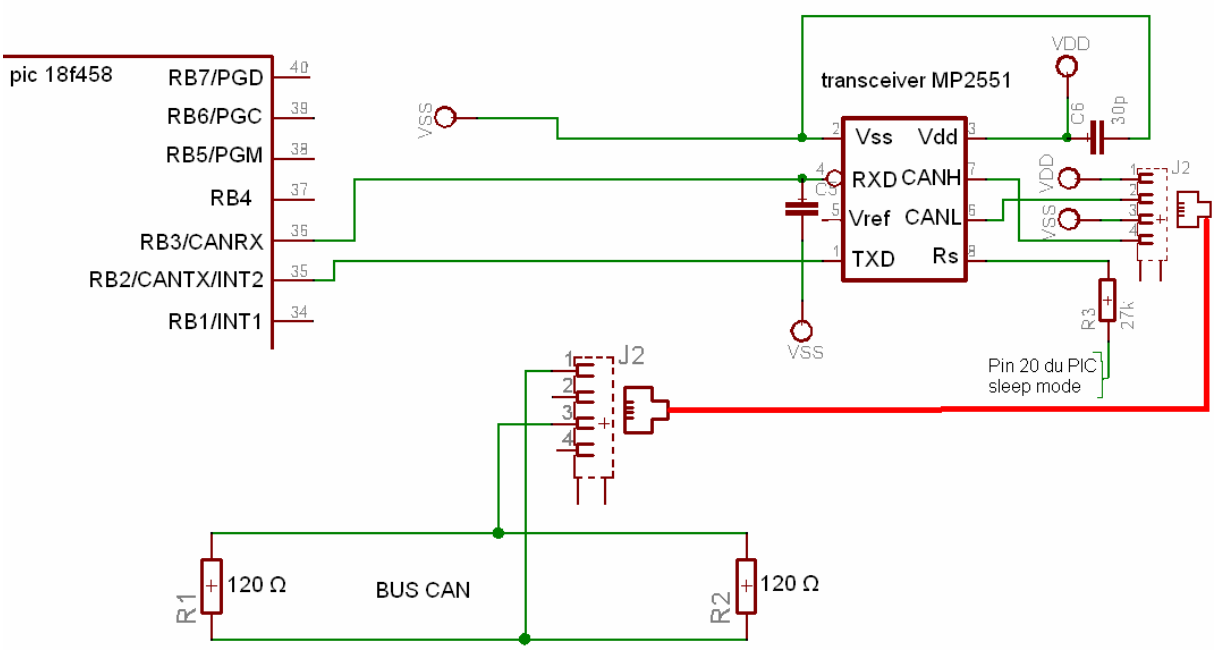


Figure 5 : Le bus CAN.

Au niveau de la connectique, pour les prototypes des connecteurs RJ12 sont utilisés. Le câblage est le suivant :

1 CAN+
2 CAN-
3 GND
4 +5VDC
5 -5VDC
6 +12VDC

1.2. La carte Sniffer

Comme nous l'avons expliqué précédemment, la carte Sniffer est un outil de test. Elle sera indispensable lors des phases finales de conception du satellite car elle permettra de vérifier le fonctionnement de celui-ci jusqu'au dernier moment avant le lancement (notamment le bon fonctionnement de la partie puissance qui sera vitale au tout début de la mission).
Son rôle est de lire les trames sur le bus CAN et de les afficher en temps réel sur une interface PC. La réception d'un message est signalé par une interruption,à la suite de celle si nous stockons le message dans une variable. Il est possible, inversement, d'envoyer par le biais du PC une trame sur le bus CAN,elle ne sera envoyée que lorsque le bus sera libre. Cette carte nous sera très utile par la suite lorsqu'il n'y aura plus les écrans LCD, nous pourrons voir tout ce qui se passe à l'intérieur du satellite.
Cette communication entre le PC et la carte est réalisée par un port série RS 232.
On utilise comme interface le logiciel de Monsieur Bigonoff que l'on peut facilement trouver sur son site. (http://www.abcelectronique.com/bigonoff/)

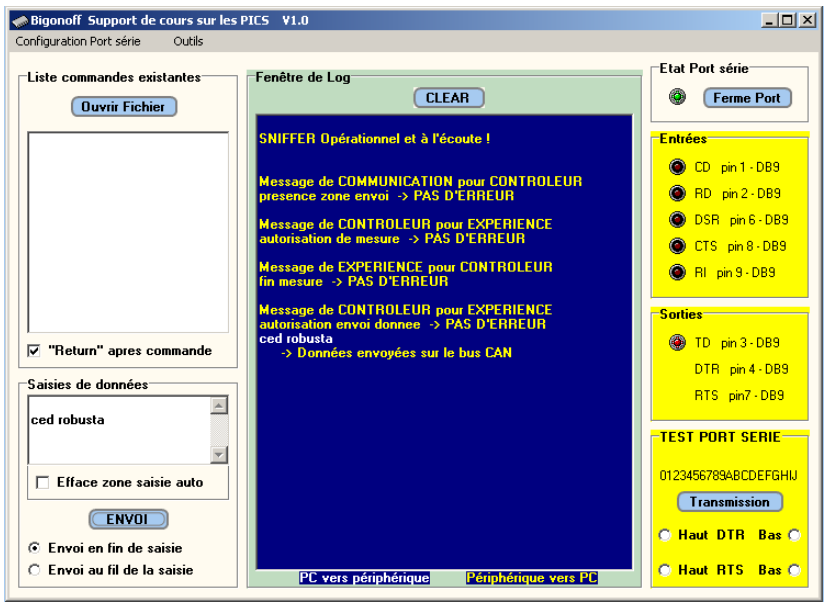


Figure 56 : Interface graphique du « sniffer »

Pour pouvoir utiliser l'interface, il suffit de choisir dans les configurations le port COM et de régler la vitesse à 9600 Bauds. Ensuite il faut cliquer sur Ouvrir Port en haut à droite.

Nous avons modifié le programme initial de cette carte pour améliorer la lecture sur l'interface et l'adapter au changement du projet.
Pour envoyer une donnée du PC au bus CAN un protocole a été mis au point.
Dans l'ordre, il faut fixer l'expéditeur, le destinataire, le type de communication (c'est-à-dire s'il s'agit d'un message ou d'une donnée) et le message ou la donnée.

Type de carte :
- a→ Carte Alimentation
- c→ Carte Contrôleur
- e→ Carte Expérience
- k→ Carte Communication

Type de communication :
- m→ Message
- d→ Données

Type de message :
- a→ Défaut de consommation
- b→ Reset
- c→ Présence dans zone d'envoi
- d→ Transmission données
- e→ Données reçues
- f→ Autorisation de mesure
- g→ Fin de mesure
- h→ Autorisation envoi de données
- i→ Envoi de données

j→ Fin d'envoi

Par exemple, on souhaite simuler l'envoi du message Reset de Contrôleur à Expérience.
Il suffit de taper dans la fenêtre « Saisie de données » : cemb
De même, si l'on souhaite simuler l'envoi de la donnée ROBUSTA à la carte Communication par la carte Contrôleur, il suffit d'écrire : ced robusta

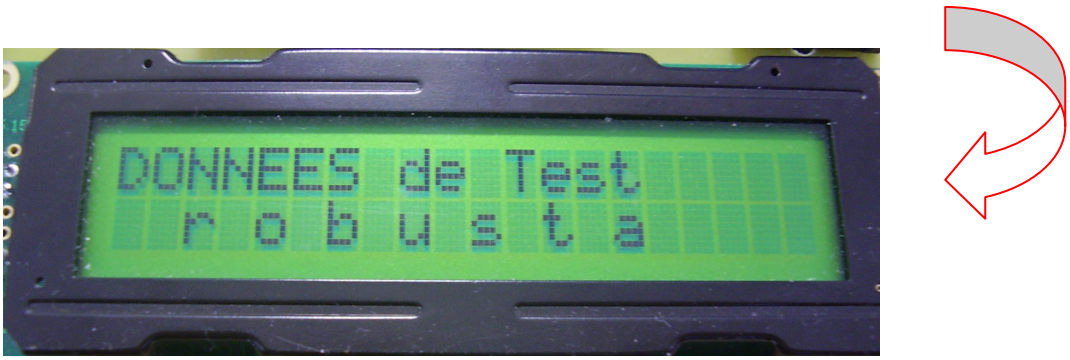


Figure 7 : Photo de l'écran LCD de la carte de test expérience

1.3. Le bus CAN

Bien que le bus I2C soit plus simple à mettre en œuvre, le bus CAN présente beaucoup d'avantage pour notre application. Le bus CAN est un bus série composé d'une paire différentielle (CAN H et CAN L) qui lui permet d'être plus robuste aux perturbations électromagnétique.
En effet si une perturbation survient, elle modifiera le potentiel des deux lignes sans modifier leur différence de potentiel.

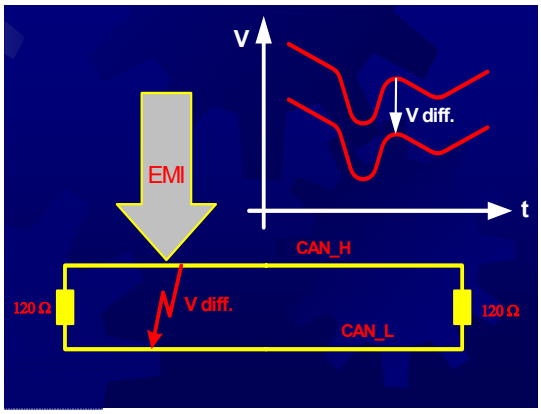


Figure 8 : Bus CAN soumit à une perturbation

De plus les paires différentielles sont torsadées ce qui réduit les perturbations radioélectriques.

Le protocole du bus CAN est basé sur le principe de diffusion. Le rôle de l'identifiant est de permettre aux différentes cartes du satellite de reconnaître les trames qui leurs sont adressées et d'ignorer les autres. Cet identifiant, de longueur 11 bits pour un message standard

ou 29 bits pour un message étendu, permet de programmer des messages avec différente priorité. Il est alors possible de rajouter des stations à un réseau sans modifier la configuration des autres stations. La vitesse maximale du bus CAN est de 1 Mbits/s. Le champ longueur indique la longueur des données à suivre.

Figure 9 : Trame du bus CAN

1.4. Protocole utilisé pour la communication entre les cartes

La figure ci-dessous résume le protocole que nous utilisons pour communiquer PIC à PIC. Nous avons utilisé le protocole du bus CAN et nous avons modifié son identifiant afin de pouvoir clairement connaître l'expéditeur, le destinataire, le type de message et faire un contrôle de parité.

Nous utilisons les quatre premiers bits pour définir l'expéditeur et les quatre suivants pour définir le destinataire. Il est possible bien sur de définir ces termes avec seulement deux bits mais pour une raison de robustesse aux erreurs il est préférable de les coder sur au moins trois bits. Quatre bits ont été choisis pour faciliter la programmation.

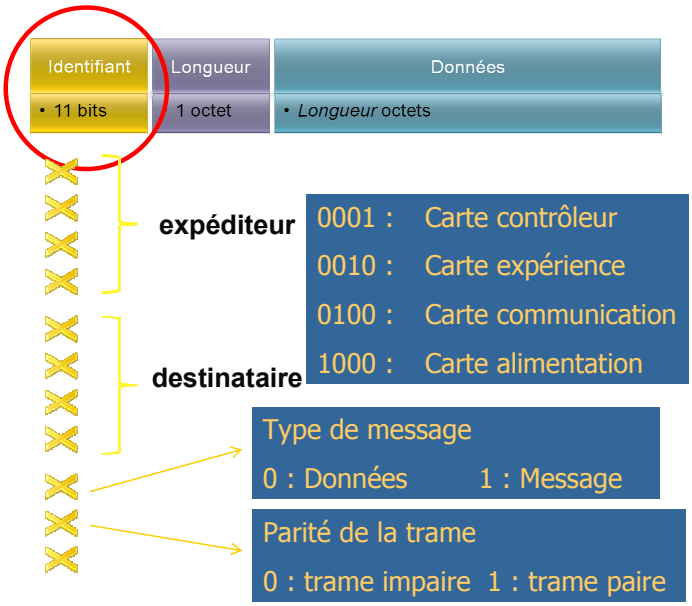


Figure 10 : Protocole de communication mis en place pour le satellite

2. Ordonnancement et système d'exploitation[j1]

La partie qui va suivre traite de l'ordonnancement effectué par la carte contrôleur. Cette partie est fondamentale, en effet la puissance du satellite étant limité une expérience ne doit pas s'effectuer en même temps que l'envoie des données à la station sol. De cet ordonnancement est né un système d'exploitation dont nous expliquerons le fonctionnement.

2.1. Ordonnancement

La carte contrôleur a pour rôle d'ordonnancer les taches au sein du satellite.
Nous avons donc réfléchi au fonctionnement des différentes cartes et nous avons réalisé un schéma qui défini les différentes communications entre les cartes.
Cet ordonnancement a été implémenté dans les PIC et maintenant la carte contrôleur réagi d'elle-même aux messages des autres cartes.

1- Envoi des données niveau de tension / courant délivré
2- Défaut de consommation
3- Reset
4- Signalisation présence dans zone d'envoi
5- Transmission des données provenant d'expérience
6- Donnée bien reçu
7- Autorisation de mesurer
8- Fin des mesures
9- Autorisation d'envoi des données
10- Envoi des données

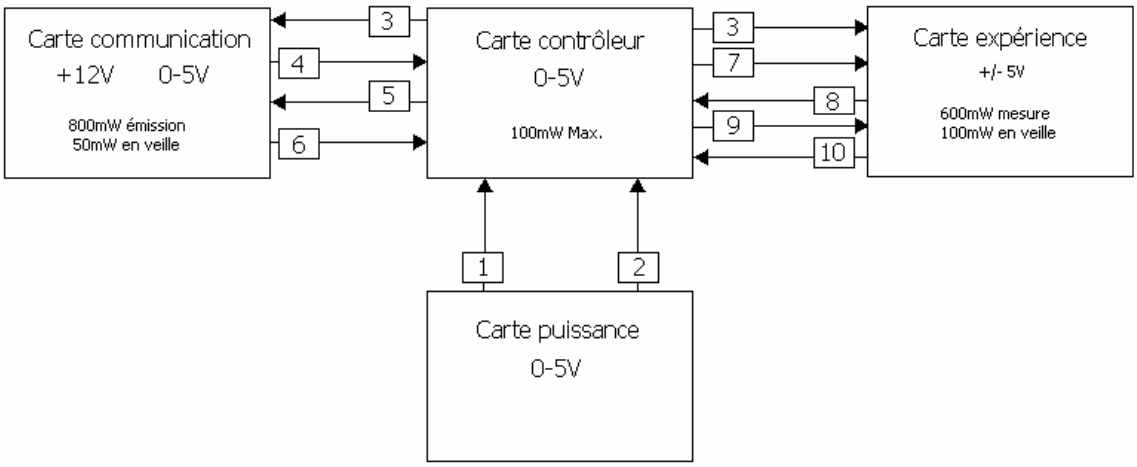


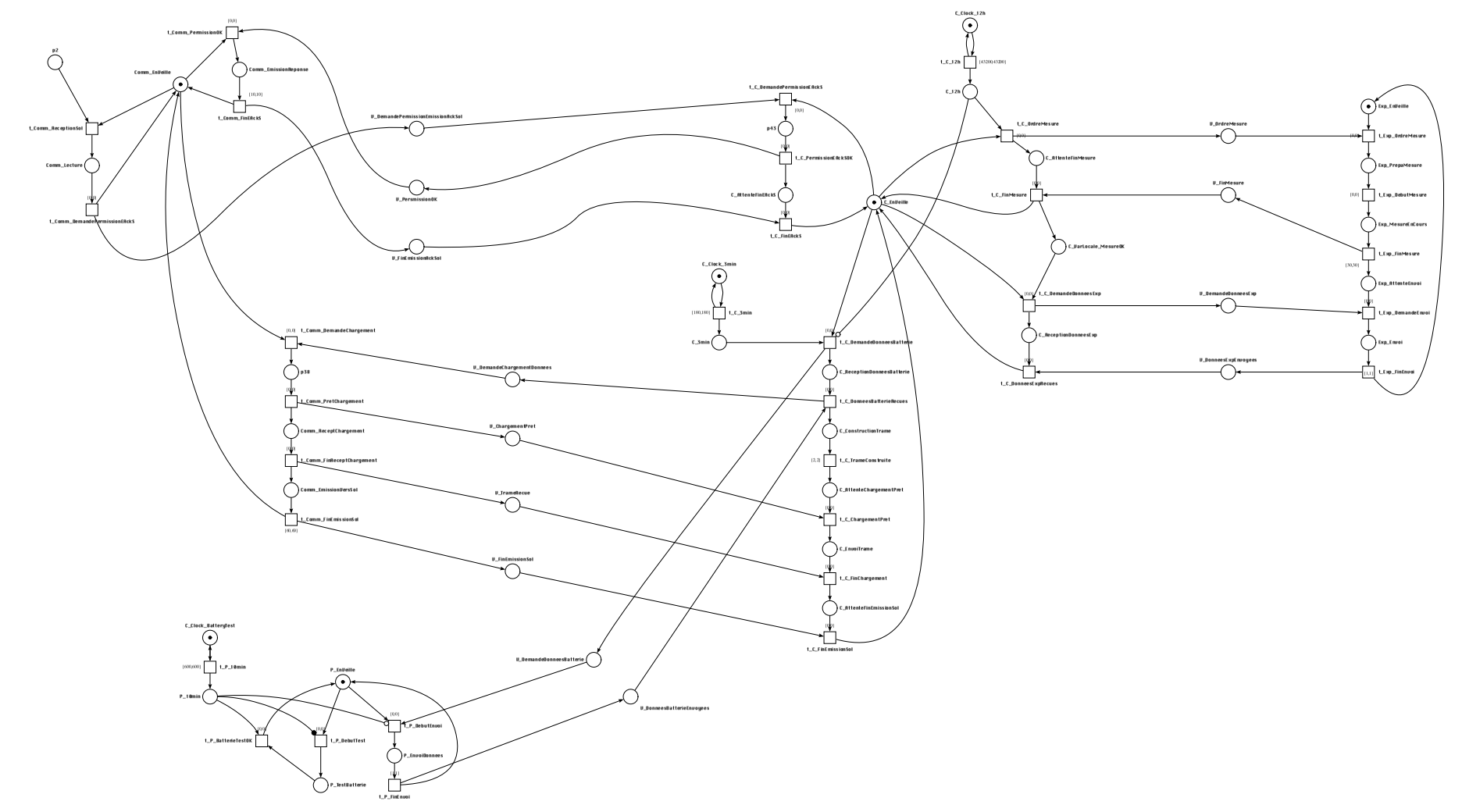
Figure 11 : Ordonnancement

Ce schéma n'est pas du tout définitif. Un nouvel ordonnancement est en cours de définition en utilisant les réseaux de Pétri. Le logiciel utilisé est TINA et est disponible sur le site du LAAS (www.laas.fr/tina).

A tout moment un reset peu être exécuté par une des cartes, nous avons donc pensé à sauvegarder l'état du système avant un reset.

2.2. Compilateur C.

A partir du logiciel TINA, un code C peut être généré. Ce code sera implémenté dans le microcontrôleur PIC grâce au compilateur MICRICHIP C18.


p2
Comm_EnVeille
t_Comm_PermissionOK
Comm_EmissionReponse
t_Comm_FinRck5
t_Comm_ReceptionSol
Comm_Lecture
t_Comm_DemandePermissionRck5
U_DemandePermissionEmissionRckSol
U_PermissionOK
U_FinEmissionRckSol
t_C_DemandePermissionRck5
p43
t_C_PermissionRck5OK
C_AttenteFinRck5
t_C_FinRck5
C_EnVeille
C_Clock_12h
t_C_12h
C_12h
t_C_OrdreMesure
C_AttenteFinMesure
t_C_FinMesure
C_BarLocale_MesureOK
U_OrdreMesure
U_FinMesure
t_C_DemandeDonneesExp
C_ReceptionDonneesExp
t_C_DonneesExpRecues
U_DemandeDonneesExp
U_DonneesExpEnvoyees
Exp_EnVeille
t_Exp_OrdreMesure
Exp_PrepaMesure
t_Exp_DebutMesure
Exp_MesureEnCours
t_Exp_FinMesure
Exp_AttenteEnvoi
t_Exp_DemandeEnvoi
Exp_Envoi
t_Exp_FinEnvoi
C_Clock_5min
t_C_5min
C_5min
t_Comm_DemandeChargement
p10
t_Comm_PretChargement
Comm_RecepChargement
t_Comm_FinRecepChargement
Comm_EmissionVersSol
t_Comm_FinEmissionSol
U_DemandeChargementDonnees
U_ChargementPret
U_TrameRecue
U_FinEmissionSol
t_C_DemandeDonneesBatterie
C_ReceptionDonneesBatterie
t_C_DonneesBatterieRecues
C_ConstructionTrame
t_C_TrameConstruite
C_AttenteChargementPret
t_C_ChargementPret
C_EnvoiTrame
t_C_FinChargement
C_AttenteFinEmissionSol
t_C_FinEmissionSol
C_Clock_BatteryTest
t_P_10min
P_10min
P_EnVeille
t_P_AtteinteTest0K
t_P_DebutTest
P_TestBatterie
t_P_DebutEnvoi
P_EnvoiDonnees
t_P_FinEnvoi
U_DemandeDonneesBatterie
U_DonneesBatterieEnvoyees

3. Le De-latcheur

Un composant électronique soumis à une contrainte radiative peut perdre sa fonctionnalité, notamment à cause de l'effet latchup. Le latchup correspond à la création d'un chemin de conduction direct entre la masse et l'alimentation ce qui amène une augmentation de la consommation et un risque de destruction du composant.

Lors d'une réunion il nous a été demandé de mettre en place un De-latcheur. Le but du De-latcheur est de limiter le courant d'alimentation lorsqu'un événement radiatif se produit. Pour limiter ce courant nous avons utilisé le composant MAXIM MAX891L.
Le montage utilisé est le suivant :

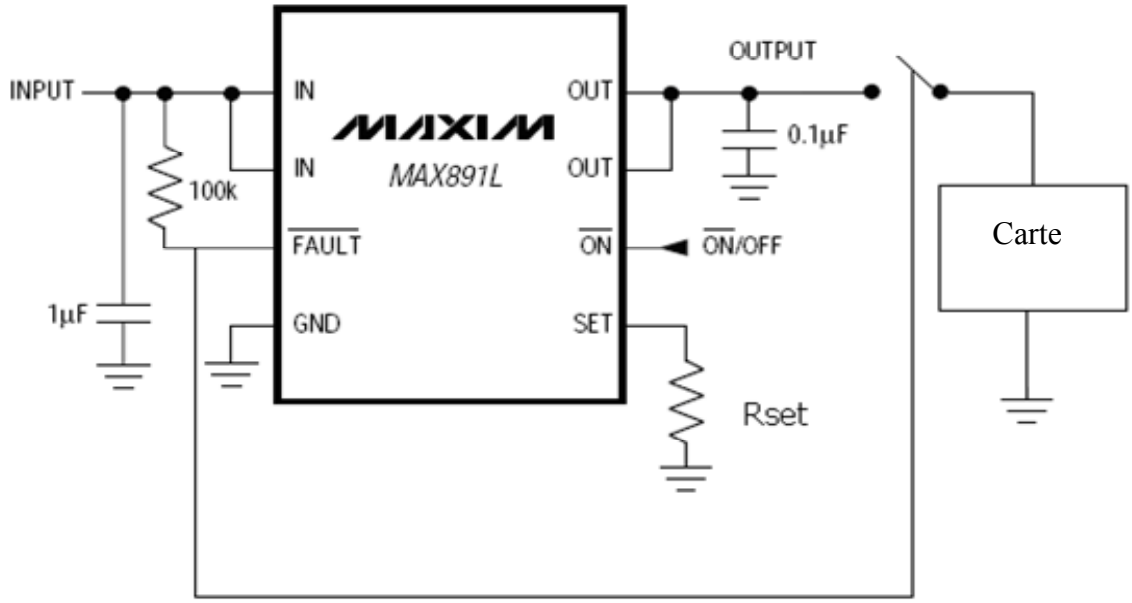


Figure 14 : De-Latcheur

La résistance Rset permet de programmer le courant limite de sortie. Le calcul de cette résistance s'obtient par :

$$Iset = Ilimit / Iratio$$ Avec Ilimit le courant désiré
Iratio est pris à 965 mais peut varier

$$Rset = 1.24 / Iset$$

Le courant programmé doit se situer entre : $100\ mA \leq Ilimit \leq 500\ mA$

Nous avons fait des premiers tests avec différentes résistances Rset en mesurant le courant à la sortie du De-latcheur. Nous avons ensuite rajouté une résistance variable en sortie

pour observer la limitation de courant. Nous avons remarqué que la tension de sortie diminuait si le courant s'amenait à être trop grand.

Nous obtenons les résultats suivants :

Rset	Ilimit	Iout à vide	FAULT niveau bas
15 KΩ	80 mA (hors limite)	110 mA	63 mA
12 kΩ	100 mA	114 mA	80 mA
10 KΩ	120 mA	136 mA	83 mA
5.6 KΩ	255 mA	250 mA	170 mA
250 Ω	4 A (hors limite)	Pas stable car dépasse la limite programmable de 500 mA	
100 kΩ	12 mA (hors limite)	43.66 mA la limite programmable est 100 mA	

Nous remarquons que le courant de sortie diffère du courant théorique. La différence peut venir du calcul, en effet Iratio est pris à 965 mais en réalité il est compris entre 805 et 1210. Si Iratio est compris dans cet intervalle nos mesures cooréspondent. Lorsque le courant programmé atteint les limites de la programmation nous avons un courant de sortie qui n'est pas stable.
Par la suite il faudra utiliser l'entrée FAULT, cette entrée passe au niveau bas lorsque le courant dépasse la limite. Dans le cas d'un latchup il faut déconnecter la carte de l'alimentation, après rétablissement de celui-ci le circuit fait un reset. Nous n'avons pas eu le temps de mettre en place en sortie du De-latcheur un interrupteur à l'aide de transistor Mos qui déconnecterait le circuit en cas de sur courant.
Il y aura un De-latcheur sur chaque carte, lors d'un reset d'une carte il faudrait que celle-ci envoie à la carte contrôleur qu'elle vient de démarrer. La carte contrôleur pourra alors contrôler se qui se passe au sein du satellite.

4. Autre fonctions

Deux autres fonctions sont actuellement en voie d'intégration sur la carte contrôleur :

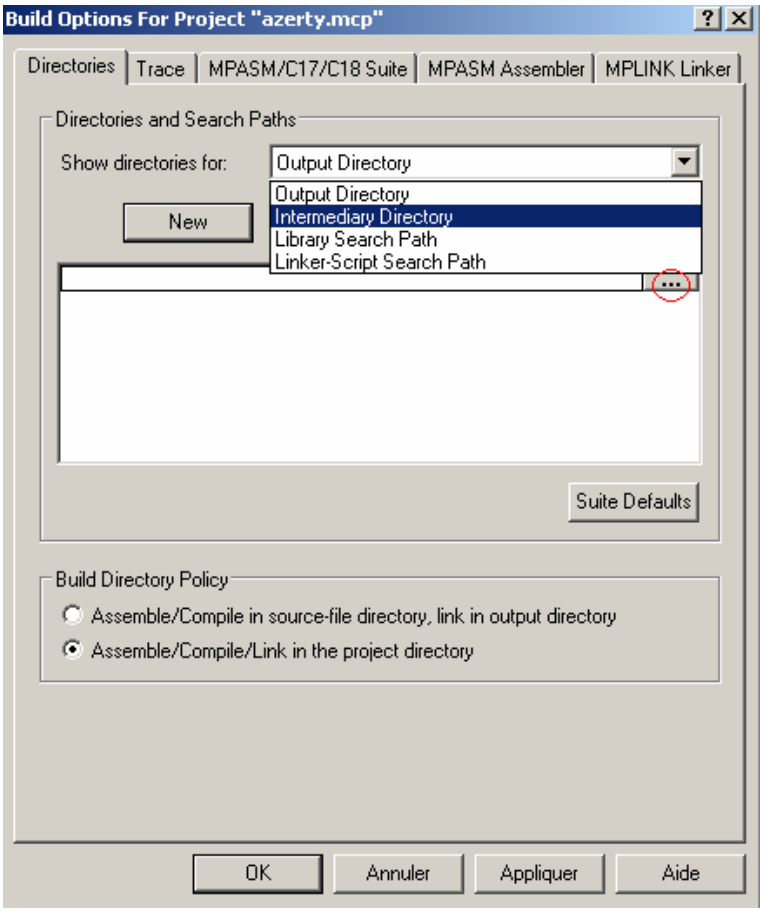
- Une horloge temps réel.
- Une mémoire série 1024 kbits.

5. Annexes

5.1. Utilisation du logiciel MPLAB

Cette partie est faite pour les personnes n'ayant jamais utilisés ce logiciel.
Tout d'abord, il faut penser à télécharger le compilateur MCC18 que vous trouverez gratuitement en version étudiante sur le site de Microchip.
Création d'un nouveau projet :

-Project → New choisir le nom et l'emplacement du projet
-Configure → Select Device choisir le pic utilisé
-Project → Select language toolsuite
Active device : Microchip C18 toolsuite
Ensuite rechercher dans le répertoire MCC18 les liens des fichiers demandés
-Dans la petite fenêtre en haut à gauche : clic droit sur Linker scripts pour aller chercher dans le dossier MCC18 du compilateur le fichier .lkr du pic utilisé.



-Project → Buildoption → project
-Dans l'onglet directories :
Pour chaque Show directories for → Intermediary directory
Library search path
Linker script search path
Cliquer sur New puis sur les trois petits points et chercher dans MCC18 respectivement le dossier h, lib et lkr

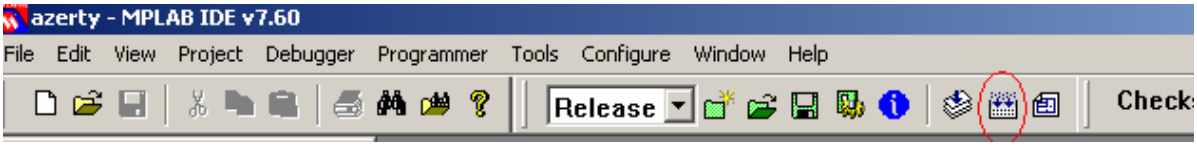
Institut d'Électronique du Sud
(Unité mixte de Recherche 5214 CNRS-UMII)
Université Montpellier II, CC 083
F-34095 Montpellier Cédex 5, France

Création d'un fichier :

Files → New
Sauvegarder le fichier dans le dossier du projet en .c ou .h selon l'utilisation
Dans la petite fenêtre en haut à gauche, clic droit sur Source files (pour les .c) ou Header files (pour les .h) et Add file…

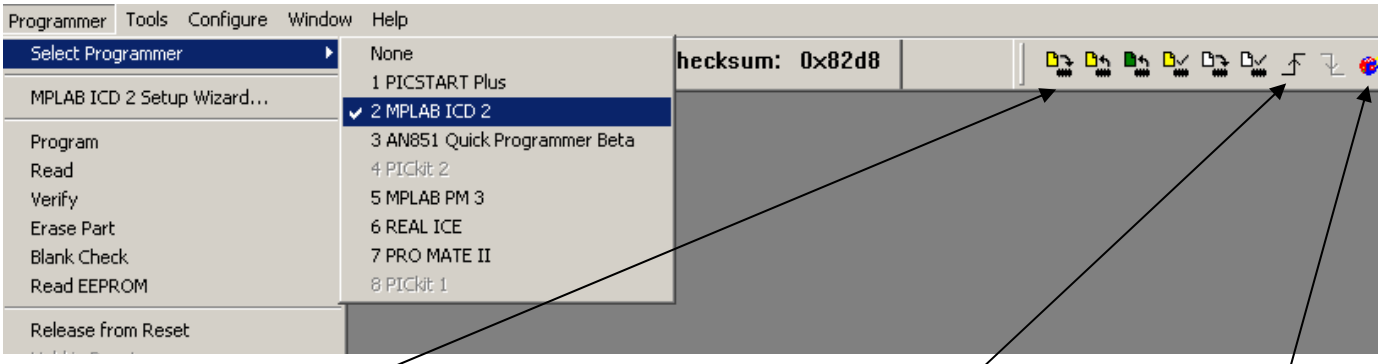
Compiler le programme :

Il suffit de cliquer sur l'icône entouré Build.



Pour transférer le programme sur le PIC :

Debbuger → Select tool → ICD2 puis utiliser les boutons qui sont apparus en haut à droite.

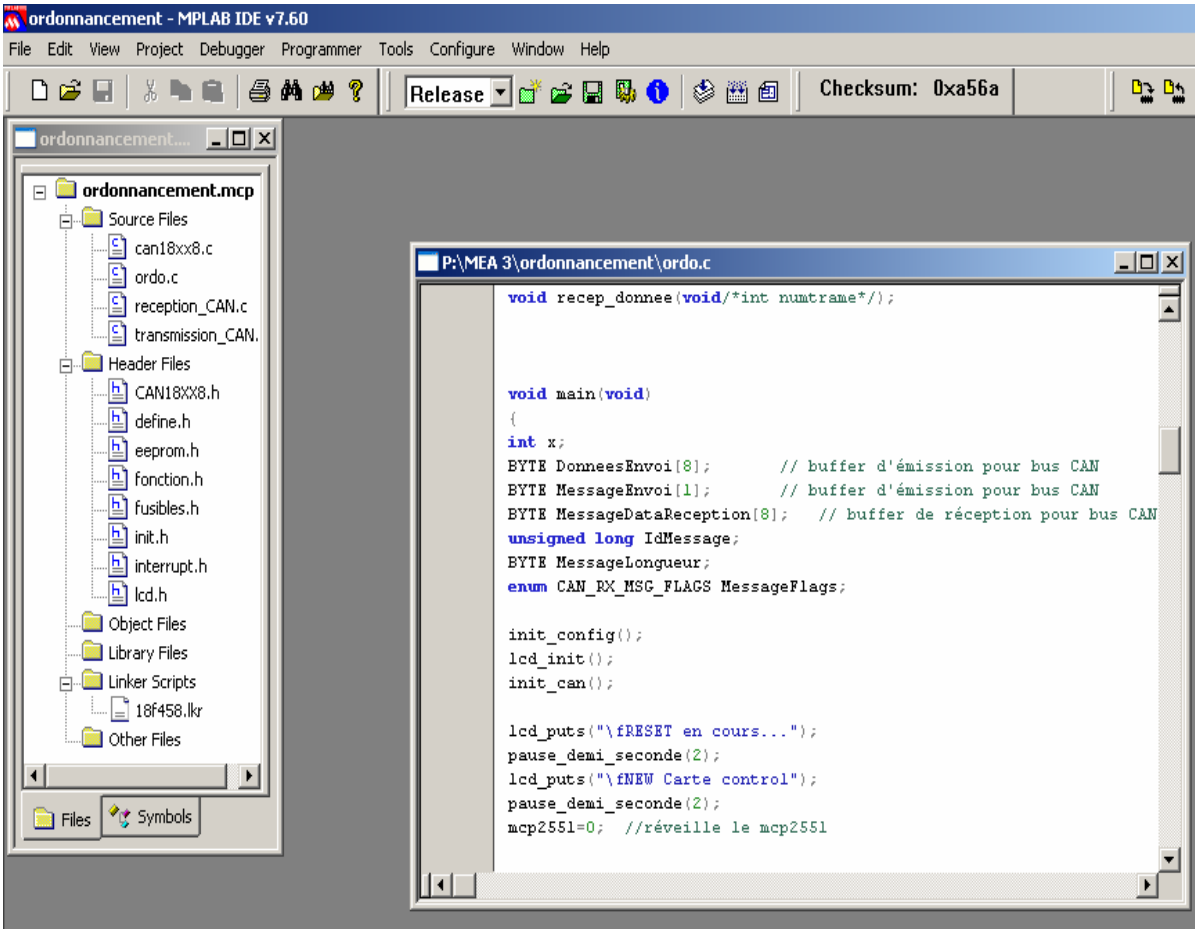


Permet de transférer le programme vers le PIC

Permet de lancer le programme sur le PIC

Connecte l'ICD2 au PIC

Exemple de projet :



Lors de la conception d'un projet, pour plus de lisibilité, il est conseillé de faire plusieurs fichiers. Ici nous avons la fonction principale main dans un fichier puis nous avons séparé les fonctions de réception, transmission, les interruptions et les initialisations.

5.2. Les fusibles

Pour commencer, nous réglons les fusibles afin de bien configurer le PIC dans le fonctionnement souhaité. Ces fusibles sont des registres de configuration. Cette configuration est faite dans un fichier fusibles.h.

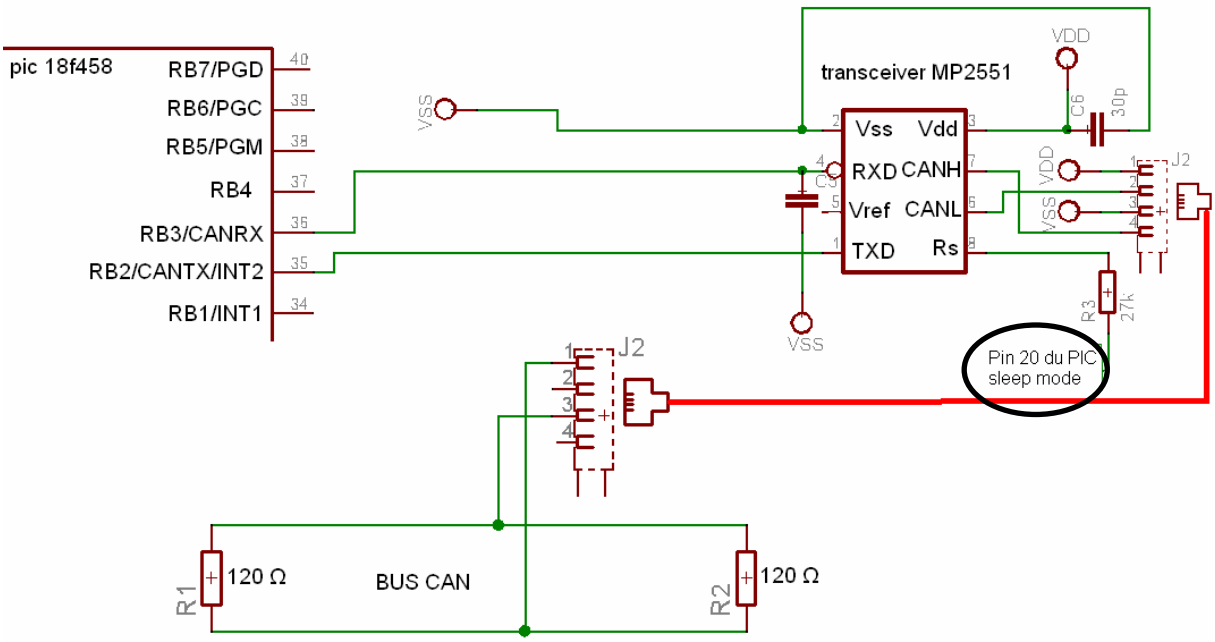
Dans fusibles.h, on a donc :

```
#pragma config OSC=HS          oscillateur en mode High Speed
#pragma config PWRT = OFF
#pragma config BOR = ON
#pragma config WDT = OFF
#pragma config WDTPS = 1
#pragma config LVP = OFF
#pragma config DEBUG = OFF
#pragma config CP0 = OFF
#pragma config CPB = OFF
#pragma config WRT0 = OFF
#pragma config WRTB = OFF
#pragma config EBTR0 = OFF
#pragma config EBTRB = OFF
```

Nous laissons volontairement le Watch Dog Timer à OFF, nous l'activerons manuellement dans le programme. Il faudra peut être changé le Watch Dog Timer Postscaler (WDTPS) pour régler la fréquence du Watch Dog.

Institut d'Électronique du Sud
(Unité mixte de Recherche 5214 CNRS-UMII)
Université Montpellier II, CC 083
F-34095 Montpellier Cédex 5, France

5.3. Les pins du PIC utilisées par le bus CAN



Le bus CAN utilise trois pins du PIC.
Sur le PIC 18F458, les pins 36 (CANRX) et 37 (CANTX) sont utilisées pour le module CAN.
Nous utilisons aussi un port de sortie, ici la pin 20 du port D, pour ordonner la mise en veille du transceiver.

Programmation de ces pins :
TRISB = 0b00000000;
PORTB= 0;
TRISB = 0b00001000; CANRX configuré en tant qu'entrée et CANTX en tant que sortie.

5.4. Programmation du bus CAN

Dans cette partie nous faisons la synthèse des fonctions associées au bus CAN qui permet de l'initialiser et de le configurer. Ces fonctions sont dans le fichier can 18xx8.c On retrouve les explications complètes dans la datasheet intitulée **PIC18C CAN Routines in 'C'**.

5.4.1. Fonction CANInitialize

```
void CANInitialize (    BYTE SJW,
                        BYTE BRP,
                        BYTE PHSEG1,
                        BYTE PHSEG2,
                        BYTE PROPSEG,
                        enum CAN_CONFIG_FLAGS config);
```

Les différents objets de la fonction (de SJW à PROPSEG) permettent de programmer la vitesse du bus.
La vitesse maximale du bus CAN est de 1Mbit/s.
Avec un oscillateur de 20MHz (Fosc) pour obtenir la vitesse maximale, on configure :

SJW= 1
BRP= 1
PHSEG1= 4
PHSEG2= 3
PROSEG= 2

Le calcul permettant d'expliquer ces résultats se situe page 233 de la datasheet intitulée **PIC 18F XX8**

Le calcul de la vitesse du bus s'obtient de la manière suivante :

$Tq = (2*BRP)/Fosc$

$8*Tq \leq Tbit \leq 25*Tq$

$Tbit = Tq*(SJW + PROPSEG + PHSEG1 + PHSEG2)$

Notre configuration donne:

$Tq = 0.1\mu s$
$Tbit = 10*Tq = 1\mu s$
$D = 1Mbit/s$

Une fois la configuration de la vitesse faite, nous disposons d'une énumération de flag (voir page 4 de la datasheet intitulée **PIC18C CAN Routines in 'C'**). Ces flags nous permettent de choisir un certain nombre d'options relatives au CAN, notamment le type de trame. Dans

notre configuration nous avons choisi une trame standard et la non utilisation de la ligne de filtre pour le réveil du CAN.

Au final la configuration est :

```
CANInitialize (1, 1, 4, 3, 2, CAN_CONFIG_VALID_STD_MSG &
                              CAN_CONFIG_LINE_FILTER_OFF);
```

5.4.2. Fonction CANSetOperationMode

```
CANSetOperationMode (CAN_OP_MODE_CONFIG)
```

Le PIC 18F458 possède six modes d'utilisation (voir datasheet **PIC18FXX8** page 226). La fonction CANSetOperationMode sert à configurer le mode du PIC. Pendant la phase d'initialisation il faut se mettre en mode configuration(CAN_OP_MODE_CONFIG) puis passer en mode normal(CAN_OP_MODE_NORMAL) pour son utilisation.

Dans la phase d'initialisation il nous faut initialiser les itérations du CAN.
La configuration est la suivante :

```
PIE3bits. RXB0IE=1;
PIE3bits. RXB1IE=1;
IPR3bits.RXB0IP=1;
IPR3bits.RXB1IP=1;
PIE3bits.WAKIE=1;
```

5.4.3. Fonctions CANSetMask et CANSetFilter

Le Pic possède deux buffers de réception, Buffer 0 et Buffer 1. Lors de l'initialisation du PIC nous pouvons utiliser des filtres qui auront pour but de filtrer les messages pour les mettre dans l'un des buffers ou les rejeter. Cette initialisation se fait à l'aide des fonctions CANSetMask et CANSetFilter. La première sert de masque, elle permet de définir les bits de l'identifiant qui seront considérés lors du filtrage. La seconde sert à indiquer les trames qui pourront passer dans les buffers. (Pour plus de détail voir page 230 de la datasheet **PIC18FXX8**).

Exemple :

Masque pour le buffer 0, seul les bits à 1 seront considérer.

```
CANSetMask (CAN_MASK_B1, 0b0000010011110000,
            CAN_CONFIG_STD_MSG);
```

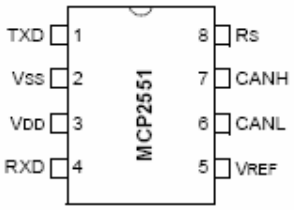
Le buffer 0 possède deux filtres (numéroté de 1 à 2), ils sont considérés de plus haute priorité. Le masque indique que seuls les bits 4, 5, 6,7 et 10 sont à considérer.

Le filtre 1 nous indique que les bits 6 et 10 doivent être à 1 et les autres à zéro.

CANSetFilter (CAN_FILTER_B1_F1, 0b0000010001000000,
CAN_CONFIG_STD_MSG);

Le buffer 1 possède quatre filtres numéroté de 1 à 4.
Ces filtres sont utiles, ils permettent de ne pas considérer les messages qui ne sont pas pour la carte.

5.5. Le transceiver MCP 2551



Sur notre circuit, il est nécessaire d'ajouter un transceiver afin de faire la liaison entre le PIC et le protocole du bus CAN.

Dans le programme principal on configure :

#define mcp2551PORTDbits.RD1

Le port RD1 est relié au Rs du transceiver, ce port permet d'ordonner la mise en veille du transceiver. Ainsi nous avons :

{ Rs = 1 mise en veille
Rs = 0 actif

5.6. Les fonctions du PIC

5.6.1. Fonction CANSendMessage

La fonction CANSendMessage est une fonction qui nous est fourni, elle sert à envoyer un message sur le bus. La syntaxe est la suivante :

CANSendMessage (unsigned long **Id**, BYTE ***Data**, BYTE **DataLen**, enum
CAN_TX_MSG_FLAGS **MsgFlags**)

{ **Id**: correspond à l'identifiant de la trame
Data : ce sont les données de la trame
DataLen : correspond à la longueur des données
MsgFlags: cette énumération de flag permet de configurer la priorité d'envoie (CAN_TX_PRIORITY_3 étant la plus prioritaire) et le type de la trame (standard ou étendue).

5.6.2. Fonction CANReceiveMessage

La fonction CANReceiveMessage est une fonction qui nous permet de récupérer les données d'une trame. La syntaxe est la suivante :

CANReceiverMessage (unsigned long ***Id***, BYTE ***Data***, BYTE ***DataLen***,
enum CAN_RX_MSG_FLAGS ***MsgFlags***)

Id: permet de récupérer l'identifiant
Data : permet de récupérer les données
DataLen : permet de récupérer la longueur des données
MsgFlags: les flags nous permettent de voir dans quel filtre le message a été reçu. Ils nous permettent aussi de voir les erreurs qui ont été détectées.

Pour avoir plus d'information sur ces fonctions voir la datasheet **PIC18C CAN Routines in 'C'.**

5.6.3. Sleepmode

Le PIC et le transceiver peuvent se mettre en veille, le réveil s'effectue par le passage d'une trame sur le bus.
Etape à effectuer pour la mise en veille :

- En premier, mettre en veille le MCP2551 car lors du passage en mode disable le CAN envoie deux trames qui réveilleraient le PIC si le transceiver n'était pas en veille.
- Passer le CAN en mode disable.
- Activer les interruptions du CAN.

Exemple de fonction de mise en veille :

```
void mode_veille (void)
{
mcp2551=1;    //mcp en veille important l'endormir avant mode sleep

CANSetOperationMode (CAN_OP_MODE_SLEEP);       //sleep du CAN

PIE3bits.WAKIE=1;           //active les IT du BUS CAN
}
```

Le passage d'une trame sur le bus est signalé par une interruption. L'interruption se voit par le bit PIE3bits.WAKIF.
Les étapes du réveil sont :

- Désactiver les interruptions du réveil
- Réveiller le MCP2551 pour pouvoir récupérer la trame
- Passer le PIC en mode Normal

Exemple de réveil :

```
if (PIR3bits.WAKIF == 1)
        {

        PIE3bits.WAKIE=0;   // désactive les its

        mcp2551=0;              // réveil du transceiver

        PIR3bits.WAKIF=0;    // remet à zéro l'interruption

        CANSetOperationMode(CAN_OP_MODE_NORMAL);  // passage en
                                                                Mode normal
        }
```

Nous avons remarqué que certaines fois lors de sont réveil le Pic n'avait pas le temps de considérer la trame qui l'a réveillée.

Watch Dog Timer

Le Watch Dog Timer est une fonctionnalité du Pic qui nous permet d'éviter de rentrer dans une boucle infini.
Pour sélectionner cette fonction, nous laissons WDT à OFF et pour l'instant WDTPS à 128 dans les fusibles.
Dans la fonction main() pour activer la fonction Watch Dog Timer nous faisons :

WDTCONbits.SWDTEN =1;

Il faut tout le long du programme remettre à zéro le Timer du Watch Dog Timer cela ce fait par :

_asm clrwdt _endasm

Nous avons pensé à désactiver cette fonctionnalité lors du Sleepmode pour cela il faut faire :

WDTCONbits.SWDTEN =0 ;

Avec un Postscaler de 128 nous avons une période de Watch Dog Timer comprise entre :

126 ms ≤ WDT période ≤ 4.2 s

D'où 4.2 secondes est la période maximale que nous pourrions avoir avec le Pic 18F458.

5.6.4. Lecture/Ecriture EEPROM

Fonction de lecture des eeproms situées dans le PIC. Etapes à suivre :

- Mettre adresse de lecture dans EEADR
- Dans EECON1 mettre EEPGD, CFGS et RD à zéro
- Les données lues se récupèrent dans EEDATA

Exemple de fonction :

```
int Read_Eeprom (int adresse)
{
EEADR = adresse;
EECON1bits.EEPGD = 0;
EECON1bits.CFGS = 0;
EECON1bits.RD = 1;

return EEDATA;
}
```

Fonction d'écriture des eeproms. Etapes à suivre :

- Mettre l'adresse d'écriture dans EEADR
- Mettre les données dans EEDATA
- Dans EECON1 mettre EEPGD et CFGS à zéro
- Dans EECON1 mettre WREN à un. Pensez à le remettre à zéro à la fin de l'écriture
- Désactiver des interruptions
- Partie en langage assembleur à ajouter :

```
_asm
        MOVLW 0x55
        MOVWF EECON2, 0
        MOVLW 0xAA
        MOVLW EECON2, 0
_endasm
```

- Dans EECON1 mettre WR à 1 tant que cette valeur est à un, le Pic est en court d'écriture, son passage à zéro signifie la fin de l'écriture

- Remettre dans EECON1 WREN à zéro
- Réactiver les interruptions

Exemple de fonction d'écriture :

```
static void Write_Eeprom (int adresse, int data)
{

EEADR = adresse;
EEDATA = data;

EECON1bits.EEPGD = 0;
EECON1bits.CFGS =0;
EECON1bits.WREN =1;

INTCONbits.GIE =0;          //désactive les interruptions GLOBALES
INTCONbits.PEIE =0;         //désactive les interruptions
                            PERIPHERIQUES/EXTERNES

//partie en assembleur obligatoire pour l'écrire
_asm
MOVLW 0x55
MOVWF EECON2, 0
MOVLW 0xAA
MOVWF EECON2, 0
_endasm

EECON1bits.WR =1;

// Attente de l'interruption de fin d'écriture
while (EECON1bits.WR == 1);

EECON1bits.WREN = 0;

INTCONbits.GIE =1;          //active les interruptions GLOBALES
INTCONbits.PEIE =1;         //active les interruptions
                            PERIPHERIQUES/EXTERNES
return 1;
}
```

Vecteur d'interruption de fin d'écriture

```
if(PIR2bits.EEIF)
{
```

```
        PIR2bits.EEIF =0;

        EECON1bits.WREN = 0;

}
```

Nous ne pouvons lire ou écrire qu'un octet à la fois. En tout il y a 256 octets dans les eeproms de l'adresse 00h à FFh. (Voir page 59 de la datasheet **PIC18FXX8**).

5.6.5. Lecture et écriture des états

Pour le système d'exploitation nous avons besoin de stocker l'état du système dans la mémoire eeprom. Pour s'assurer d'avoir le bon état nous triplons le stockage de l'état.

Exemple de l'écriture de l'état :

```
static void write_etat(int etatini)
{
        write_eeprom(0, etatini);
        write_eeprom(1, etatini);
        write_eeprom(2, etatini);
}
```

Nous stockons l'état courant dans trois espaces mémoire.

Lors de la lecture de l'état, si deux états sont identiques on considère que c'est l'état courant.

Exemple de la lecture :

```
static int read_etat()
{
        int etat1,etat2,etat3;
        etat1=read_eeprom(0);
        etat2=read_eeprom(1);
        etat3=read_eeprom(2);

        if(etat1= =etat2)
        { return etat1;}

        if(etat1= =etat3)
        { return etat1;}

        if(etat2= =etat3)
        { return etat2;}
         else {return attente;}
}
```