Gestion de bus CAN

Sylvain Choisel Matthieu Liger Vincent Oberlé

3 février 2000

Table des matières

Introduction

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, comme alternative aux encombrants et complexes câbles des modèles de l'époque.

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 1 Mbits/s avec un très faible taux d'erreur.

Le CAN est aussi devenu un standard international reconnu par l'ISO.

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. De nombreux packages de développement existent aussi sur le marché.

Ce document présente un exemple d'implémentation d'un bus CAN reliant différentes plateformes : une carte PowerPC 403 avec sa carte contrôleur CAN, un PC utilisant aussi une carte contrôleur CAN, et des noeuds autonomes.

Cet environnement est le résultat du développement :

- de plusieurs cartes mettant en œuvre des contrôleurs CAN,
- des librairies permettant d'utiliser ces cartes et de développer des applications utilisant le bus CAN,
- d'applications de développement et de démonstration.

Bonne lecture !

Chapitre 1

Le protocole CAN.

Le CAN est un protocole de communication série qui supporte efficacement le contrôle en temps réel de systèmes distribués tels qu'on peut en trouver dans les automobiles, et ceci avec un très haut niveau d'intégrité au niveau des données.

Le CAN a été standardisé par l'ISO dans les normes 11898 pour les applications à hauts débits et ISO 11519 pour les applications à bas débits.

1.1 Origines et utilisations du CAN.

1.1.1 Le CAN dans l'industrie automobile.

Pour satisfaire les exigences de plus en plus importantes du client en matière de sécurité et de confort, et pour se conformer aux les lois de réduction de la pollution et de la consommation de plus en plus drastiques, l'industrie automobile a développé de nombreux systèmes électroniques : systèmes anti-patinage, contrôle électronique du moteur, de l'air climatisé, fermeture centralisée des portes, etc.

La complexité de ces systèmes et la nécessité d'échanger des données entre eux signifient de plus en plus de câbles. A côté du coût très important de ce câblage, la place qui lui est nécessaire pouvait le rendre tout simplement impossible. Enfin, le nombre croissant de connections et de câbles posait de sérieux problèmes de fiabilité et de réparation.

Bosch, un important équipementier automobile, a fournit la solution dans le milieu des années 80 avec le bus CAN. L'entreprise allemande a définit le protocole et a autorisé de nombreux autres fabricants à développer des composants compatibles CAN.

Avec le protocole CAN, les contrôleurs, capteurs et actionneurs communiquent entres eux sur deux câbles à une vitesse pouvant aller jusqu'à 1 Mbits/s.

1.1.2 Autres applications industrielles.

Les contrôleurs CAN sont physiquement petits, peu coûteux et entièrement intégrés. Ils sont utilisables à des débits importants, en temps réel et dans des environnements difficiles. Enfin, les transmissions ont un haut niveau de fiabilité. C'est pourquoi ils ont été utilisés dans d'autres industries que l'automobile et des applications utilisant le CAN sont aujourd'hui disponibles dans l'agriculture, la marine, le matériel médical, les machines textiles, etc.

1.1.3 Perspectives.

On apprend dans une enquête de la publication Electronics Weekly du 15 novembre 1995 que, dans le monde :

- 5,5 millions de composants CAN ont été vendus en 1995,
- plus de 3 millions de bus CAN installés dans des véhicules et 6 millions dans des applications non automobiles,
- en 1999, 140 millions de composants CAN devraient avoir été vendus.

1.2 Le CAN dans le modèle ISO/OSI.

Le CAN étant un protocole réseau, il s'intègre dans la norme ISO/OSI[1] qui définit 7 couches permettant de couvrir la totalité d'un protocole.

Les différentes couches [5, pages 75–82] définissent les services du protocole.

Le tableau 1 page 7 (source : [2]) résume les couches utilisée par le protocole CAN.

<table>
<tr><td>Spécifications</td><td colspan="2">Couche OSI</td><td>Implémentation</td></tr>
<tr><td>à spécifier par l'utilisateur</td><td colspan="2">Couche application</td><td>Software embarqué ou non</td></tr>
<tr><td rowspan="5">Spécifications du protocole CAN</td><td rowspan="2">Couche de communication de données</td><td>Logical Link Control</td><td rowspan="4">On-chip hardware</td></tr>
<tr><td>Medium Access Control</td></tr>
<tr><td rowspan="2">Couche physique</td><td>Physical Signaling</td></tr>
<tr><td>Physical Medium Attachment</td></tr>
<tr><td>Medium Dependent Interface</td><td>Off-chip hardware</td></tr>
<tr><td></td><td colspan="2">Transmission Medium</td></tr>
</table>

TAB. 1.1 – *Couches OSI appliquées au CAN*

1.2.1 Le modèle ISO/OSI

1.2.1.1 La couche physique.

La première couche du modèle a pour but de conduire les éléments binaires jusqu'à leur destination sur le support physique. Elle fournit les moyens matériels nécessaires à l'activation, au maintien et à la désactivation de ces connections physiques.

Cette couche gère la représentation du bit (codage, timing, synchronisation), et définit les niveaux électriques, optiques,... des signaux ainsi que le support de transmission.

Le protocole CAN ne décrit que la représentation détaillée du bit (Physical Signalling), mais pas le moyen de transport et les niveaux des signaux de telles sorte qu'ils puissent être optimisés selon l'application.

1. ISO : *International Standards Organization*, OSI : *Open Systems Interconnection*

1.2.1.2 La couche liaison.

Elle fournit les moyens fonctionnels nécessaires à l'établissement, au maintien et à la libération des connexions entres les entités du réseau. Cette couche[2] devra notamment corriger les erreurs qui ont pu se produire au premier niveau (même s'il est impossible de corriger toutes les erreurs).

Le protocole CAN décrit entièrement cette couche.

La couche liaison est subdivisée en deux sous-couches. La sous-couche LLC (Logical Link Control) effectue :

- le filtrage des messages,
- la notification des surcharges (overload),
- la procédure de recouvrement des erreurs.

La sous-couche MAC (Medium Access Control), qui est le cœur du protocole CAN, effectue :

- la mise en trame du message,
- l'arbitrage,
- l'acquittement,
- la détection des erreurs,
- la signalisation des erreurs.

1.2.1.3 La couche réseau.

La couche réseau doit permettre d'acheminer correctement les paquets d'informations jusqu'à l'utilisateur final, en passant par des passerelles qui interconnectent plusieurs réseaux entre eux. Elle assure le contrôle des flux (pour tenir des temps de réponse acceptables), le routage des paquets, et l'adressage (pour l'ensemble des machines du monde !).

C'est aussi ici qu'interviennent les deux philosophies concurrentes des réseaux:

- Le mode connecté, à la base du protocole X.25, où l'émetteur et le récepteur se mettent d'accord sur un comportement commun.
- Le mode non connecté, à la base du protocole IP[3], sans contraintes pour l'émetteur vis-à-vis du récepteur.

Cette couche est vide dans le protocole CAN.

1.2.1.4 La couche transport.

La couche transport est le dernier niveau qui s'occupe de l'acheminement des informations. Elle doit optimiser la qualité de la transmission, notamment avec des outils de détection d'erreurs et des algorithmes de renvoi des messages perdus.

Cette couche est vide dans le protocole CAN.

2. aussi appelée couche de communication de donnée (Data Link Layer)
3. Internet Protocol

1.2.1.5 La couche session.

La couche de session permet aux différents éléments du réseau d'organiser et de synchroniser leur dialogue. Il faut en effet s'assurer si l'on veut émettre de l'information qu'un récepteur est là pour récupérer ce qui a été envoyé.

Cette couche est vide dans le protocole CAN.

1.2.1.6 La couche présentation.

Cette couche se charge de la syntaxe des informations que se communiquent les éléments du réseau, c'est-à-dire que ces éléments utilisent bien un langage commun pour transférer des données.

Cette couche est vide dans le protocole CAN.

1.2.1.7 La couche application.

C'est la dernière couche du modèle OSI. Elle donne aux applications le moyen d'accéder aux couches inférieures.

Cette couche a été normalisée en 1987 au sein d'une structure globale : la structure de la couche application, ou ALS (*Application Layer Structure*). Elle détermine comment différentes applications vont pouvoir coexister et utiliser des modules communs. De très nombreuses normes ont été définies sur cette base.

Cette couche n'est bien sûr pas vide pour le protocole CAN, mais sa spécification est laissée à l'utilisateur.

1.3 Fonctionnement du CAN.

1.3.1 Principes.

1.3.1.1 "Identifiers".

Les trames de données transmises par un nœud sur le bus ne contiennent ni une quelconque adresse du nœud expéditeur ou du nœud destinataire. C'est plutôt le contenu du message, sa signification[4] qui est précisé par un identificateur (ID). Chaque nœud recevant un message regarde si celui-ci est intéressant pour lui grâce à l'ID. Si c'est le cas, il le traite, sinon, il l'ignore.

Cet unique ID indique aussi la priorité des messages. Plus la valeur est faible, plus le message sera prioritaire. Si deux nœuds ou plus cherchent à avoir accès au bus en même temps, c'est celui de plus haute priorité qui gagne. Les messages de priorité inférieure seront automatiquement retransmis lorsque le bus sera libre.

1.3.1.2 Notions de bits dominants / récessifs.

La norme CAN ne spécifie pas de couche physique. Différentes implémentations sont donc possibles : filaire, HF, infrarouge, par fibre optique, etc.

Mais toute implémentation doit respecter le principe des bits dominants / récessifs. Chaque nœud doit pouvoir présenter sur le bus un bit appelé dominant (0 logique) et un bit appelé

4. son "meaning".

récessif (1 logique). Les implémentations doivent aussi respecter la règle suivante : si 2 nœuds présentent des niveaux logiques différents, le bit dominant s'impose.

1.3.2 Fonctionnement détaillé de l'arbitrage.

Dans un système typique, certains paramètres vont changer plus rapidement que d'autres. Ce sera par exemple la vitesse d'un moteur, tandis qu'un paramètre plus lent pourra être la température de l'habitacle.

Il est donc naturel que les paramètres qui varient le plus soient transmis le plus souvent et par conséquent doivent avoir une plus grande priorité.

Dans les applications en temps réel, ceci nécessite non seulement une vitesse de transmission importante, mais aussi un mécanisme d'allocation du bus efficace qui soit capable de traiter les cas où plus d'un nœud cherchent à transmettre en même temps.

Pour déterminer la priorité des messages, le CAN utilise la méthode CSMA/CD[5] avec la capacité supplémentaire de l'arbitrage non destructif[6] afin d'offrir une disponibilité maximale du bus.

Comme on l'a vu précédemment, la priorité d'un message est déterminée par la valeur de son ID. La valeur de chaque ID, et donc la priorité de chaque type de messages, est assignée durant la conception du système. Un certain nombre de standards ont été développés selon les domaines d'utilisation du bus CAN pour fixer la priorité des ID et permettre une interopératibilité des différents équipements.

Tout conflit de bus est résolu par le mécanisme du "ET cablé", c'est-à-dire qu'un état dominant écrase un état récessif. Concrètement, si plusieurs nœuds débutent leur trame en même temps, le premier qui présente un bit récessif alors qu'au moins un autre présente un bit dominant perd l'arbitrage (dans la trame, l'ID commence par le bit de poids fort).

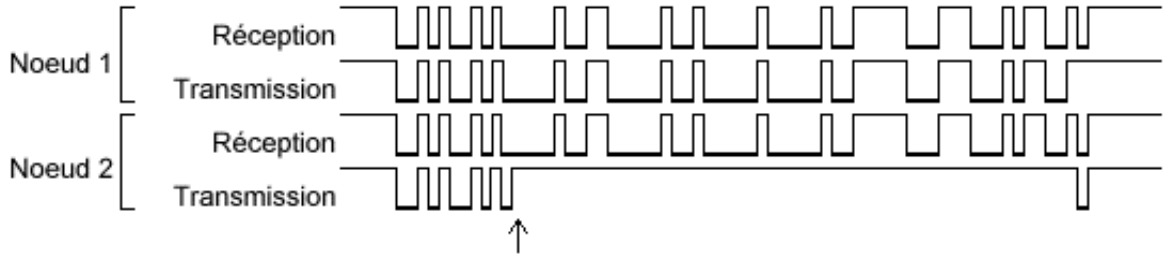


FIG. 1.1 – *Le nœud perd l'arbitrage au 11ème bit.*

Tout ce passe donccomme si le message de plus haute priorité était le seul à être transmis. Lorsqu'un nœud perd l'arbitrage, il devient automatiquement un récepteur du message en cours de transmission, et il n'essaiera de retransmettre son message que lorsque le bus sera à nouveau libre.

L'avantage d'un tel système est une utilisation meilleure du bus qu'avec d'autres mécanismes[7].

5. Carrier Sense, Multiple Access with Collision Detect.
6. Non-Destructive Bitwise Arbitration.
7. Tels que le mécanisme de "fixed time schedule allocation" du Token ring ou l'arbitrage destructif de l'Ethernet.

1.3.3 Transmission des messages.

L'information sur le bus est envoyée sous la forme de messages[8] au format fixé. Quand le bus est libre, n'importe quel nœud peut commencer à envoyer un message.

1.3.3.1 Protocoles 2.0A et 2.0B.

Le protocole CAN 2.0 comporte deux sous-spécifications qui diffèrent uniquement au niveau de la longueur de l'ID. La version 2.0A définit des ID de 11 bits et la version 2.0B des ID de 29 bits. On appelle ces trames respectivement des trames standards ("Standard Frames") et des trames étendues ("Extended Frames").

Le format standard est équivalent au format tel qu'il était décrit dans la version 1.2 du protocole. Le format étendu est une nouvelle fonctionnalité du protocole 2.0. Pour permettre le développement de contrôleurs assez simple, le support complet du format étendu n'est pas requis pour être conforme au protocole 2.0. Les contrôleurs sont considérés conforme au protocole 2.0 s'ils respectent les deux conditions suivantes :

- Le format standard doit être totalement supporté.
- Ils doivent être capables de recevoir des trames étendues, mais sans forcément être capable de les traiter. Elles ne doivent seulement pas être détruites.

On se référera à [1] en 10.3.1.2 pour le détail du fonctionnement de la compatibilité du CAN 2.0B par rapport au 2.0A.

1.3.3.2 Types de messages.

Quatre types de messages peuvent être transmis :

- Les Data Frames sont utilisées pour transporter des données sur le bus. Leur format est détaillé ci-après.
- Les Remote Frames sont utilisées par un noeud pour demander la transmission d'une Data Frame par d'autres noeuds avec le même ID. Deux éléments distinguent cette trame d'une trame de données normale : elles ne contient aucun bit de données et son bit RTR est récessif.
- Les Error Frames sont transmises par un noeud ayant détecté une erreur. Leur format et utilisation est détaillée par la suite.
- Les Overload Frames sont utilisées pour produire un délai entre deux Data ou Remote Frames successives. Voir [1] en 10.3.4.

1.3.4 Format des trames de données.

Les trames de données (Data Frames) sont composées de 7 parties détaillées ci-après.

Le format est indiqué pour des trames respectant le protocole 2.0A. On se référera à [1] pour le format des trames 2.0B.

8. Appelés aussi trames ou "frames".

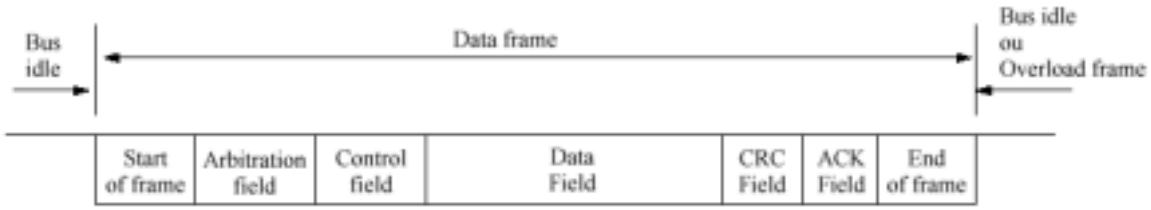


FIG. 1.2 – *Data Frame.*

1.3.4.1 Start of frame

Le bit Start of frame (SOF) marque le début d'une Data Frame ou d'une Remote frame. C'est un unique bit dominant.

Un nœud ne peut bien sûr débuter une transmission que si le bus est libre.

Ensuite, tous les autres nœuds se synchronisent sur SOF du nœud ayant commencé une transmission.

1.3.4.2 Arbitration field

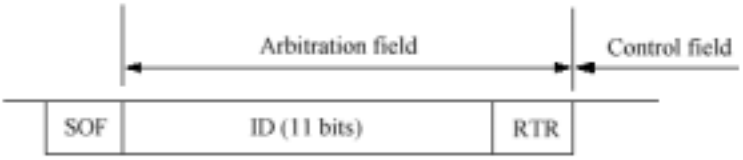


FIG. 1.3 – *Arbitration field : format standard.*

L'Arbitration field est constitué de l'identifieur et du bit RTR.

L'identificateur (ID) permet d'identifier le message. Il est transmis dans l'ordre ID10 à ID0, où ID0 est le bit le moins significatif.

Le bit RTR (Remote Transmission Request) caractérise les Remote Frames. Il est dominant dans les Data Frames et récessif dans les Remote Frames.

1.3.4.3 Control field

Le 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 dominant (1000) correspondent à la valeur 8.

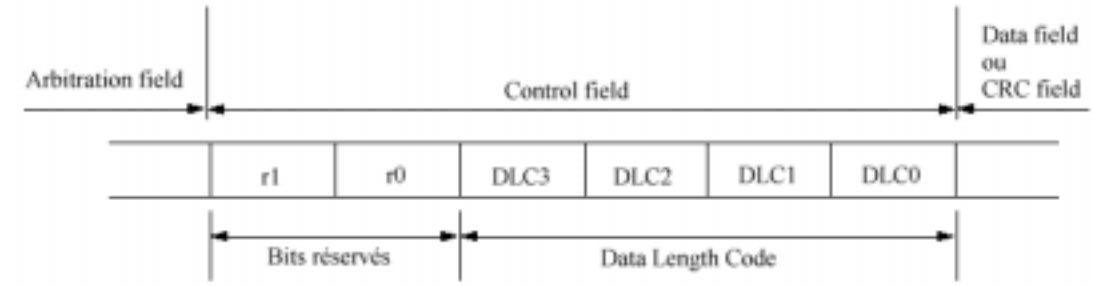


FIG. 1.4 – *Control field.*

1.3.4.4 Data field

Ce 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.

1.3.4.5 CRC field

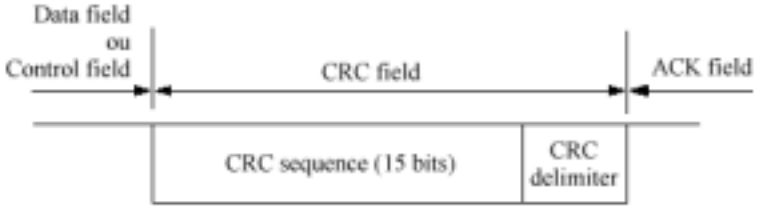


FIG. 1.5 – *CRC field.*

Le CRC field est composé de la séquence de CRC sur 15 bits suivi 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.

1.3.4.6 ACK field

Le ACK field est composé de 2 bits, l'ACK Slot et le ACK Delimiter (1 bit récessif). Le nœud en train de transmettre envoie un bit récessif pour le 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.

1.3.4.7 End of frame

Chaque Data frame et Remote frame est terminée par une séquence de 7 bits récessifs.
[PENDING : mettre une trame complète, page 96 de la doc]

1.3.5 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.

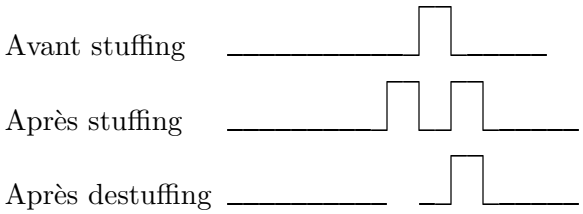


FIG. 1.6 – *Exemple de bit stuffing.*

1.3.6 Détection et gestion des erreurs.

Le CAN a été créé pour opérer dans des environnements difficiles et c'est pourquoi il comprend de nombreux mécanismes de détection d'erreur.

1.3.6.1 Types d'erreurs

Le CAN implémente cinq mécanismes de détection des erreurs, 2 au niveau bits (le "bit monitoring" et le "bit stuffing") et 3 au niveau messages (vérification du CRC, de la forme des trames et de l'acquittement).

Ces cinq types d'erreurs différents qui peuvent être détectée par un nœud sont :

Bit error Un nœud envoyant un bit sur le bus regarde aussi en même temps les bits qu'il reçoit (Bit monitoring). Il considère comme une erreur de bit lorsque le bit envoyé est différent du bit reçu, à l'exception de l'envoi d'un bit récessif durant l'arbitrage (cas de la perte d'arbitrage) ou pendant le ACK Slot (trame acquittée).

Stuff error Le nœud détecte une erreur de stuffing lorsqu'il reçoit 6 bits consécutifs de même valeur dans une partie d'un message qui devrait être codé avec la méthode du bit stuffing.

CRC error Une erreur de CRC est détectée lorsque le CRC calculé par un récepteur est différent de la valeur du CRC contenu dans la trame.

Form error Une "Form error" est détectée lorsqu'un bit qui devrait être à une certaine valeur est à une valeur différente (un délimiteur par exemple).

ACK error Le transmetteur détecte une erreur d'acquittement lorsqu'il ne reçoit pas de bit dominant pendant le ACK Slot.

1.3.6.2 Trames d'erreurs

Une trame d'erreur est constituée de deux parties. La première est formée par la superposition des différents "Error flags" mis par les nœuds du bus. La seconde partie est un délimiteur.

Un nœud qui détecte une erreur la signale en envoyant un Error flag. Celui-ci viole la règle du bit stuffing (6 bits dominants consécutifs) et par conséquent, tous les autres nœuds détectent aussi une erreur et commencent à envoyer un Error flag. La séquence de bits dominants qui existe alors sur le bus est le résultat de la superposition de plusieurs Error flags, et sa longueur varie entre 6 et 12 bits.

Il existe deux types d'Error flags :

Active error flag : 6 bits dominants consécutifs.

Passive error flag : 6 bits récessifs consécutifs, jusqu'à ce qu'ils soient écrasés par des bits dominants.

L'Error delimiter est composé de 8 bits récessifs. En fait, après avoir transmis son Error flag, chaque nœuds envoie des bits récessifs et observe le bus jusqu'à ce qu'il détecte un bit récessif, après quoi il envoie encore 7 bits récessifs supplémentaires.

1.3.6.3 Gestion et confinement des erreurs

Le confinement des erreurs est un mécanisme permettant de faire la différence entre des erreurs temporaires ou permanentes. Les erreurs temporaires peuvent être causées par des glitchs par exemple, tandis que des erreurs permanentes sont en général due à de mauvaises connections ou à des composants défaillants.

Ce système va permettre d'enlever un nœud défaillant du bus qui sinon aurait pu perturber les autres nœuds.

Un nœud peut être dans trois états : error-active, error-passive ou bus-off.

1. Un nœud en mode d'erreur actif (error-active) peut prendre part normalement dans la communication sur le bus. Il transmettra un Active error flag s'il détecte une condition d'erreur.
2. Un nœud en mode d'erreur passif (error-passive) peut prendre part dans la communication, mais s'il détecte une condition d'erreur sur le bus, il transmettra un Passive error flag. Ce mode indique un nœud à problèmes.
3. Un nœud en mode bus-off n'est pas autorisé à avoir une quelconque influence sur le bus.

Deux compteurs d'erreurs sont implémentés dans chaque nœud : celui des erreurs en transmission (Transmit error count) et celui des erreurs en réception (Receive error count).

Les grandes règles de modifications des compteurs d'erreurs sont les suivantes. Elles sont détaillées avec leurs exceptions dans [1] en 5.2.

- Lorsqu'un récepteur détecte une erreur, son Receive error count est augmenté de 1.
- Lorsqu'un transmetteur envoie un Error flag, son Transmit error count est augmenté de 8.
- Après une transmission réussie, le Transmit error count est diminué de 1.
- Après une réception réussie, le Receive error count est diminué de 1.

Un nœud est en mode error-active si ses deux compteurs d'erreurs sont inférieurs à 127. Il est en error-passive si l'un des deux est compris entre 128 et 256. Si le Transmit error count est supérieur à 256, le nœud est en bus-off.

1.3.6.4 Probabilité de détection des erreurs

Le protocole CAN est extrêmement fiable au de niveau de la détection des erreurs.

Les erreurs survenants sur tous les nœuds sont détectables à 100 % .

Au niveau des erreurs survenants sur un seul nœud, rien que la vérifcation du CRC peut permettre de détecter jusqu'à 5 erreurs de bits avec une probabilité de 100 % . La probabilité qu'une erreur ne soit pas détectée par le mécanisme du CRC est de 3.10^{-5}.

Avec tous les autres mécanismes de détection, la vraie valeur est de 10^{-11}.

De plusieurs appareils identiques, celui que vous choisirez sera le seul à ne pas fonctionner.
Généralisation de l'Axiome du Choix de Picavet.

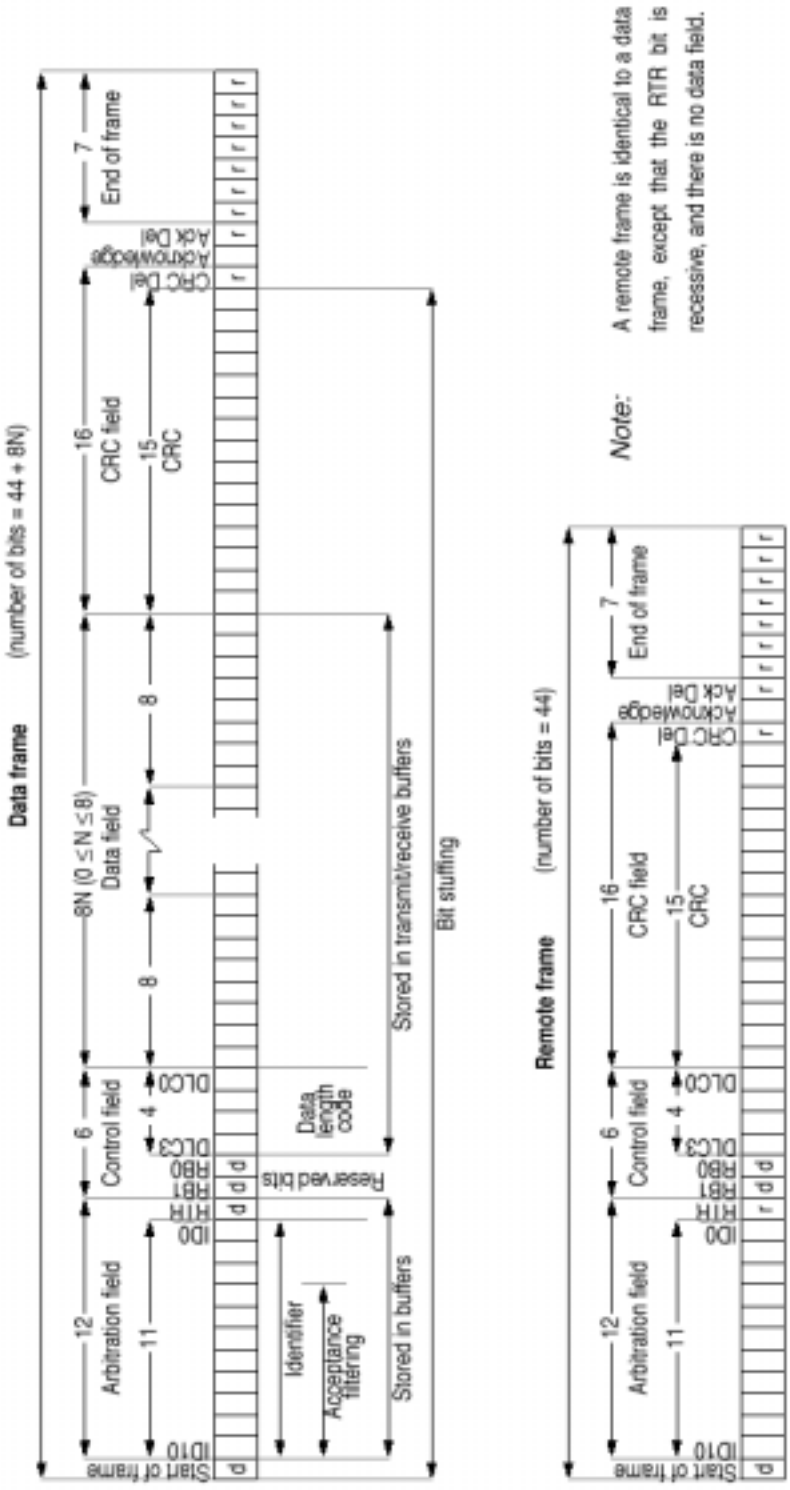


FIG. 1.7 – *Formats des trames (1 sur 2).*

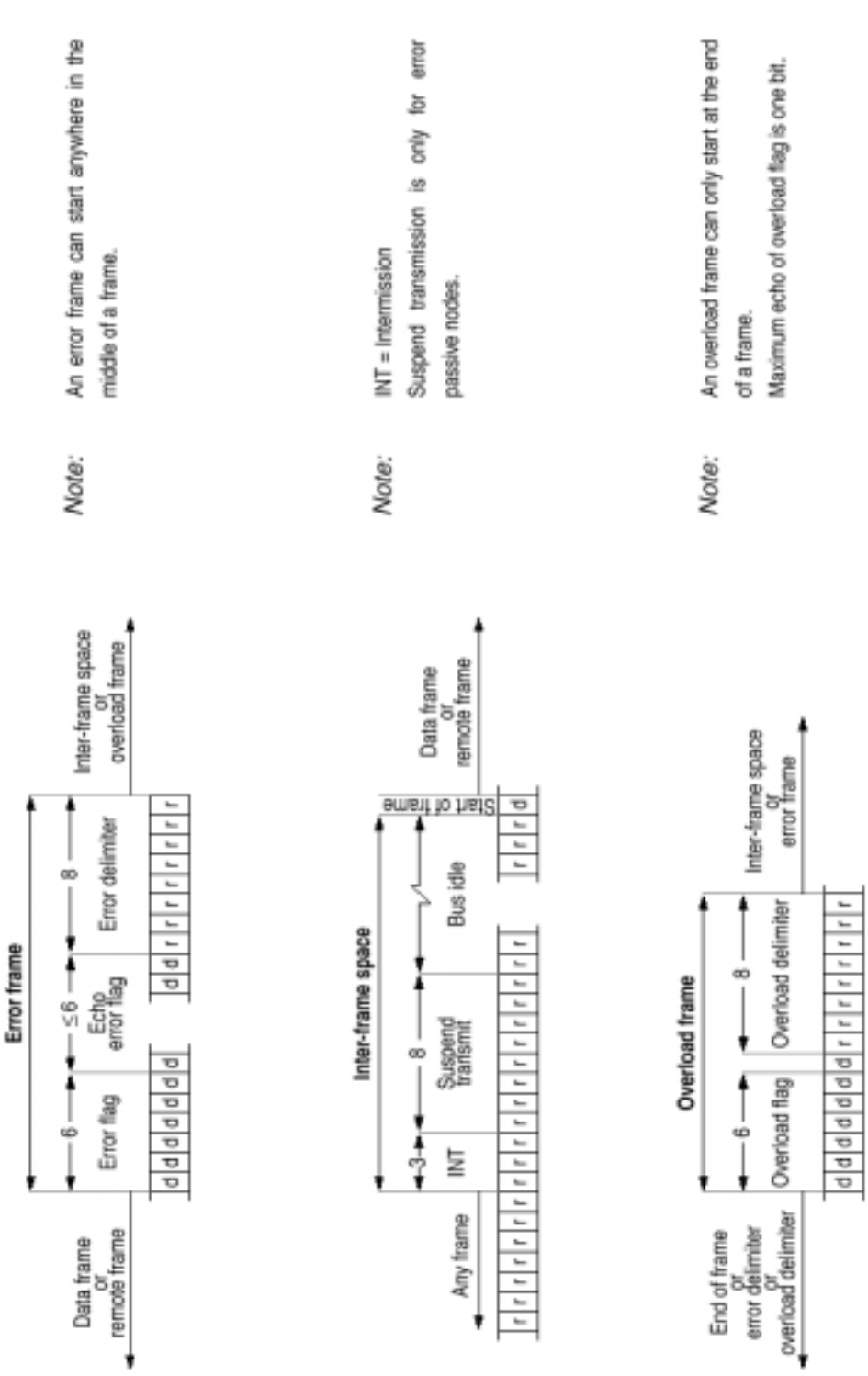


FIG. 1.8 – *Formats des trames (2 sur 2).*

Chapitre 2

Les composants CAN

2.1 Le P82C150

Le P82C150 est un CAN Serial Linked I/O (SLIO), un n œud CAN sans intelligence.

Ce périphérique inclut un contrôleur CAN et 16 pins d'entrées/sorties. Son contrôleur CAN respecte les spécifications 2.0A et 2.0B (en mode passif) du protocole CAN. Il inclut un oscillateur interne l'affranchissant d'une horloge externe.

Ses 16 pins d'E/S sont individuellement configurables en mode analogique ou digital. Il est ainsi possible d'avoir jusqu'à 16 entrées digitales, avec la possibilité d'une transmission automatique d'un message au changement d'une des entrées. Le 82C150 intègre un ADC sur 10 bits, qu'il est possible de multiplexer sur 6 entrées. On peut aussi configurer jusqu'à 16 sorties 3 états ou 2 sorties quasi-analogiques (DPM[1] avec une précision de 10 bits.

L'oscillateur interne limite le débit entre 20kbits/s et 125kbits/s. Cet oscillateur interne, un RC, implique aussi une procédure complexe de calibration au RESET du SLIO et ensuite régulièrement à l'utilisation. Ces deux limitations peuvent être dépassées en utilisant un oscillateur externe.

L'absence d'intelligence du SLIO oblige à le mettre sur un bus où un nœud plus "intelligent", typiquement un microcontrôleur sera capable de le configurer et de le commander.

2.1.1 Configuration du SLIO

2.1.1.1 Choix de l'ID

L'ID du 82C150 se fixe de façon hard. En fait, seuls 4 bits (sur les 11) sont paramètrables. Ceci limite en pratique le nombre de SLIO sur un réseau à 16.

2.1.1.2 Calibration

Afin de minimiser l'encombrement des systèmes à base de 82C150, celui-ci ne nécessite pas d'horloge externe grâce à l'utilisation d'un circuit RC interne. Ce circuit se calibre grâce à certaines trames qu'il recoit.

Plus concrètement, l'utilisation du 82C150 implique de le calibrer lors de son reset grâce à des messages de calibration, puis de le maintenir calibré en lui envoyant régulièrement ce message de calibration (environ tous les dixièmes de secondes pour un débit de 50kbits/s).

1. Discrete Pulse Modulation

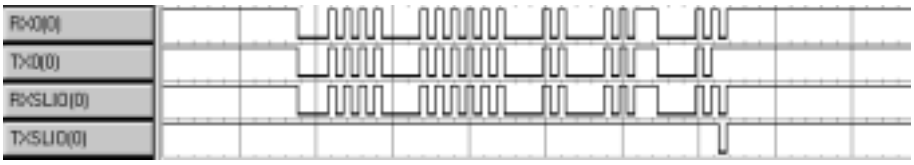


FIG. 2.1 – *Message de calibration*

Un message de calibration est un message avec un ID particulier (imposé) dont la trame (non imposée) a pour caractéristique de présenter un nombre important de transitions 1-0. Au reset, il faut envoyer ce message plusieurs fois au SLIO, qui, lorsqu'il sera correctement calibré, renverra un message d'identification, le *sign on message*. Celui-ci indique l'ID du 82C150 ainsi que l'état des 16 entrées digitales.

2.1.2 Utilisation du 82C150

A l'exception de l'ID, le pins du 82C150 se configurent et se commandent entièrement avec des trames de data. Ces trames permettent d'écrire dans les registres du 82C150. Celui-ci envoie aussi des trames indiquant l'état de ses registres.

2.1.2.1 Format des trames

Les trames envoyées ou reçues par le SLIO ont toujours 3 octets (sauf les remote frame bien sûr). Le premier de ces bytes indiquent le numéro du registre du SLIO. Les registres étant de 16 bits, les 2 bytes suivant indiquent le contenu du registre spécifié par l'adresse.

2.1.2.2 Les registres

Le 82C150 possède 9 registres. En voici le détail :

- Le *Data Input Register* contient l'état des 16 pins en entrées. L'état de ce registre sera transmis soit après réception d'un remote frame ou d'un data frame de premier octet le numéro de ce registre, soit au changement d'état de l'une des entrées si le 82C150 est configuré ainsi.
- Les registres *Positive Edge Register* et *Negative Edge Register* permettent de configurer, pour chaque bits, à quel moment doit être envoyé la data frame contenant l'état des pins. Autrement dit, si par exemple le bit 10 du *Positive Edge Register* est à 1, une data frame sera automatiquement envoyé lorsque l'entrée 10 passe à 1, pour peu qu'elle soit configurée en entrée digitale.
- Le *Data Output Register* permet d'indiquer l'état des pins configurés en sortie.
- Le *Output Enable Register* permet de configurer les pins en sorties digitales.
- L'*Analog Configuration Register* configure l'ADC.
- Les registres *DPM1* et *DPM2* configurent les sorties quasi-analogiques. Celles-ci sont sur les pins 10 (DPM1) et 4 (DPM2).
- L'*ADC Register* contient le résultat d'une conversion analogique–digitale. Une lecture de ce registre lance une conversion.

Il est important de noter que toute écriture dans l'un de ces registres provoque l'envoi d'une data frame avec la valeur du registre modifié.

2.2 Le PCA82C250

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.

2.2.1 Codage physique des bits

La seule contrainte pour une couche physique, est l'implémentation d'un ET cablé: si un seul noeud transmet un bit DOMINANT, l'état du bus est DOMINANT, et si **tous** les noeuds transmettent un état RECESSIF, l'état du bus est RECESSIF.

Le transceiver 82C250 code les bits DOMINANT et récessifs de la manière suivante:
- Etat DOMINANT (TX=0) CANH - CANL = 2V
- Etat RECESSIF (TX=1) CANH - CANL = 0V

NB: 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 noeud d'imposer une tension différentielle de 2V, correspondant à un état dominant.

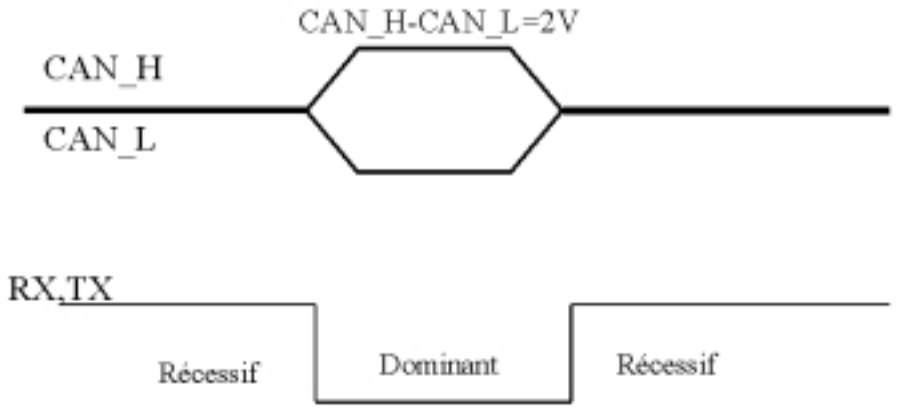


FIG. 2.2 – *Couche Physique du CAN*

2.2.2 Débit

Le débit maximal d'un noeud utilisant un du 82C250 est de 1Mbit/s, soit la limite théorique de débit d'un bus CAN. En cas d'utilisation du 82C2500 à débit réduit pour environnements critiques, il est possible de régler le slew-rate des transitions DOMINANT-RECESSIF, afin de limiter les RFI.

2.3 Le SJA1000

2.3.1 Présentation

Le SJA1000 est un contrôleur CAN ne nécessitant qu'un microcontrôleur externe. Il a été développé par Philips en 1997 comme un successeur (compatible) du 82C200.

2.3.1.1 Caractéristiques

- Buffer de réception de 64 octets
- Supporte le CAN 2.0A et 2.0B
- Débit jusqu'à 1Mbit/s[2]
- Compteurs d'erreur avec accès lecture/écriture
- Interruptions pour chaque type d'erreur sur le bus
- Capture de la dernière erreur disponible
- Détails sur le bit d'ID ayant causé une perte d'arbitration
- Mode "Listen Only" (aucune action sur le bus)

2.3.2 Le mode BasicCAN

Ce mode est un mode simplifié dont la principale utilité est la compatibilité totale avec le 82C200 (prédecesseur du SJA1000)

2.3.3 Le mode PeliCAN

Le mode Pelican est le mode dans lequel il faut utiliser le SJA1000 pour avoir accès à toutes ses fonctionalités (CAN 2.0B, registres d'erreur, informations sur arbitration perdue...).

2.3.4 Les principaux registres

2.3.4.1 Le registre MODE

Ce registre contrôle le mode de fonctionnement du SJA1000. Il précise en particulier si le SJA1000 est en mode normal ou en mode de réinitialisation ("Reset Mode"). Outre le mode normal, le SJA1000 peut fonctionnner dans les modes :

Listen Only : Aucune **action** sur le bus.

Self test MODE : Acquittement par un autre nœud non-nécessaire pour une bonne transmission.

Sleep MODE : Déconnexion du bus tant qu'il n'existe aucune activité sur celui-ci.

2. En fait environ 600Kbit/s de données **utiles**

Le registre **MODE** permet également de régler différents modes de fonctionnement du filtre d'acceptation (voir plus bas).

2.3.4.2 Le registre COMMAND

Comme son nom l'indique, le registre **COMMAND** est utilisé pour commmander le contrôleur CAN. Ces commandes peuvent-être:

Transmit Request (bit 0) : Demande la transmission des données présentes dans le buffer de transmission.

Abort Transmission (bit 1) : Annule une demande de transmission faite **et qui n'a pas encore commencé**. Si elle a déja commencé, elle se termine.

Release Receive Buffer (bit 2) : Libère la place du dernier message reçu dans le buffer de reception.

Clear Data Overrun (bit 3) : Ré-initialise l'indicateur de surcharge.

Go To sleep (bit 4) : Ordonne au SJA1000 de passer en Sleep mode.

2.3.4.3 Le registre STATUS

Receive Buffer STATUS (bit 0) : vaut 1 si un message est présent dans le buffer de réception.

Data Overrun STATUS (bit 1) : vaut 1 si un message a été perdu à cause d'une saturation du buffer de reception.

Transmit Buffer STATUS (bit 2) : vaut 1 si le buffer de transmission est prêt à recevoir des donnée.

Transmission Complete (bit 3) : vaut 1 si la dernière transmission demandée a été correctement transmise et acquittée.

Receiving STATUS (bit 4) : vaut 1 si un message est en cours de réception.

Transmit STATUS (bit 5) : vaut 1 si un message est en cours de transmission.

Error STATUS (bit 6) : vaut 1 si le compteur d'erreur de transmission ou celui de réception a atteint la limite d'alerte (96 par défaut).

Bus STATUS (bit 7) : vaut 1 si le noeud est connecté au bus (ni en Sleep **MODE** ni Bus-Off pour cause d'erreurs à répétitions.

2.3.4.4 Le registre INTERRUPT

Ce registre permet d'autoriser la génération d'interruption sur la broche $\overline{INT}$ pour les évènements suivants:

- Réception d'un message (bit 0)
- Transmission d'un message compléte
- Changement des **STATUS** d'erreurs (Error Status et Bus Status)
- Changement du **STATUS** de saturation (**Data Overrun STATUS**)
- Wake-up interrupt (reveil du noeud pour cause d'activité sur le bus)
- Passage du **MODE** "error-active" au **MODE** "error-passive" ou vice-versa (seuil de 127 des compteurs d'erreurs)
- Perte d'une arbitration
- Détection d'une erreur sur le bus

2.3.4.5 Les registres BUS_TIMING0 et BUS_TIMING1

Ces registres permettent de régler la durée d'un bit. Bien évidemment, le bit-timing doit être le même pour chaque nœud présent sur le bus.

2.3.5 Le buffer de réception

Le buffer de réception est implémente sous la forme d'une FIFO de 64 octets. Le SJA1000 donne accès a une "fenêtre" qui permet à l'utilisateur de lire le dernier message reçu. Une fois cette lecture faite, l'utilisateur peut ordonner au SJA1000 de libérer la place correspondante dans le buffer. Ceci aura pour conséquence de déplacer la fenêtre vers le message suivant. La figure 2.3.5 permet de visualiser le fonctionnement du buffer.

En plus de la fenêtre de réception, l'utilisateur peut également accéder directement aux 64 octets du Buffer. On dispose pour cela d'un registre précisant la position du dernier message reçu (RX_BUFFER_START_ADDRESS) et d'un registre précisant le nombre de messages présents dans le buffer (RX_MESSAGE_COUNTER).

Lorsqu'un message arrive alors que le buffer de réception est plein, le message arrivant est perdu (par **ce** noeud), et l'indicateur de saturation ("Data Overrun Satus") est positionné.

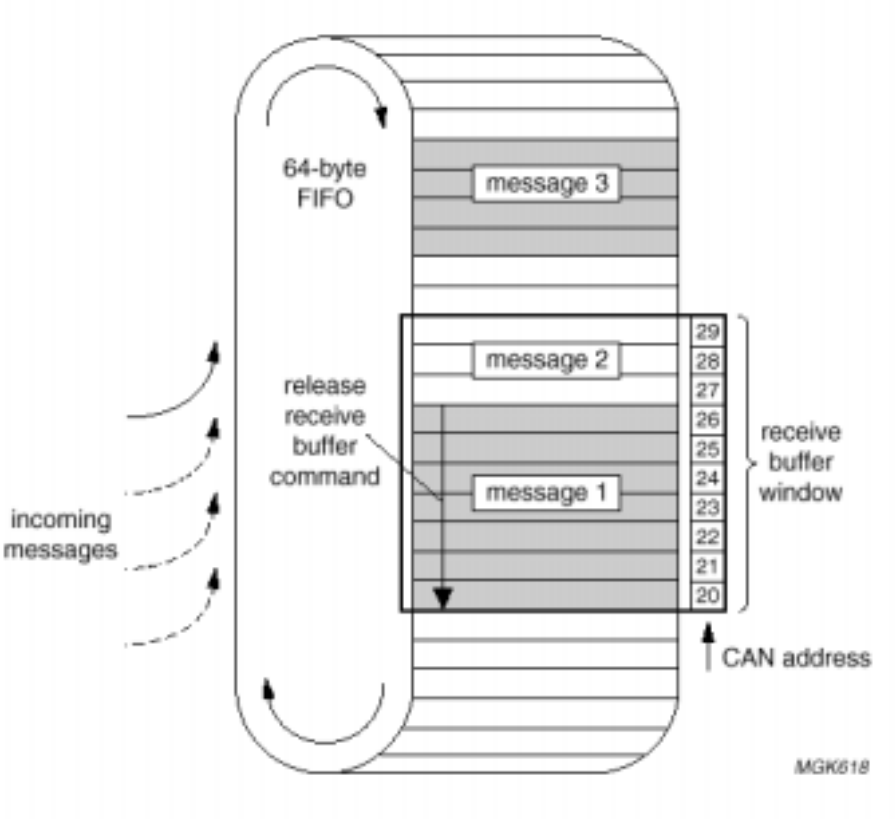


FIG. 2.3 – *Buffer de réception*

2.3.6 Le buffer d'émission

Le buffer de transmission est un simple buffer de 12 octets (jusqu'à 4 octets d'ID pour le CAN 2.0B et 8 octets de données).

Une fois que l'utilisateur a rempli ce buffer, il peut faire une demande de transmission.

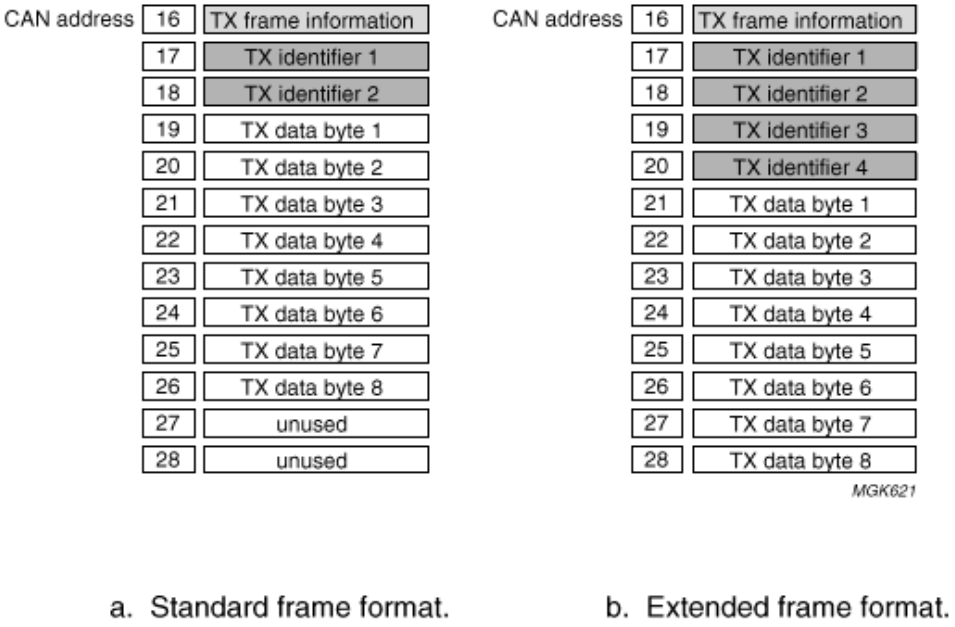


FIG. 2.4 – *Buffer de transmission*

2.3.7 Le filtre d'acceptation

Le filtre d'acceptation (*Acceptance Filter*) a pour rôle de contrôler les identificateurs (*identifiers*) des messages présents sur le bus avant de les laisser entrer dans le buffer de réception. Une bonne utilisation de ce filtre permet d'éviter une saturation du buffer de réception.

Le filtre d'acceptation est constitué des *Acceptance Code Registers* et "Acceptance Mask Register").

Les *Acceptance Code Registers* contiennent l'ID "optimal" attendu par le SJA1000 (c'est à dire celui que devra avoir un message pour pouvoir entrer dans le buffer de réception, si les *Acceptance masks* sont tous à 0).

Les *Acceptance Mask Registers* précisent les bits de l'ID qu'il faut tester (0 si un bit doit être contrôlé et 1 s'il n'a pas d'importance).

Si ce qui refusait de marcher depuis des semaines marche soudainement, c'est que quelque chose d'autre ne va plus fonctionner.
Loi de de la Mécanique en Robotique de Corwin.

Chapitre 3

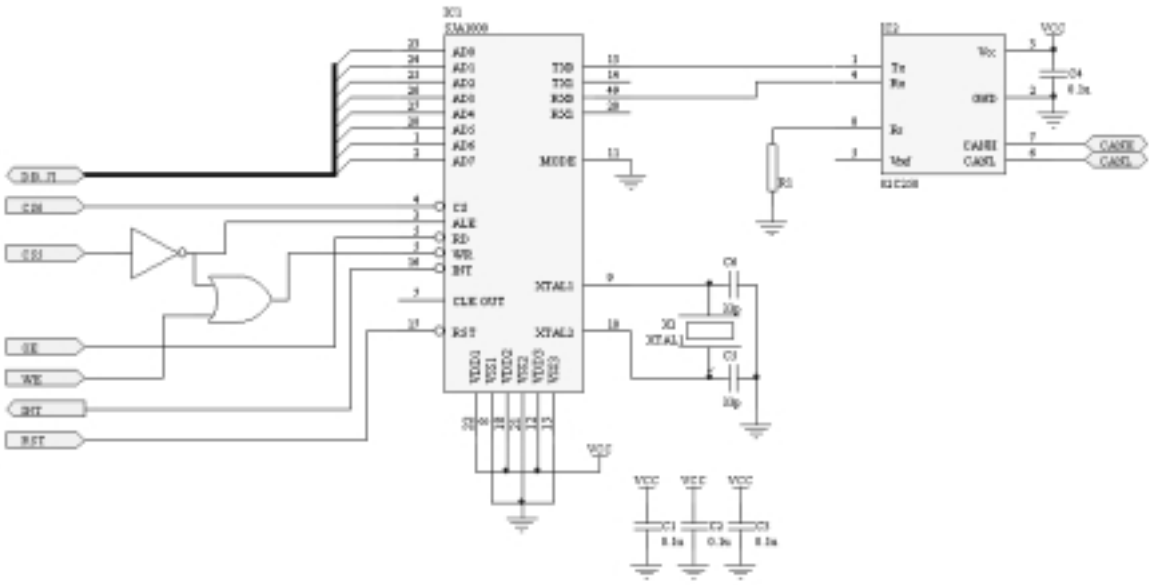
Cartes développées

3.1 Carte SJA1000-PPC

Comme nous l'avons vu précédemment, le SJA1000 est un composant complet et assez simple d'emploi pour ajouter des fonctionnalités CAN à un système à base de microcontrôleur. Le 68HC912 de Motorola, intégrant un module CAN, n'étant pas encore au point, nous nous sommes tournés vers le PowerPC 403, dont la carte de développement existait déjà.

La carte d'interface (wrappée) implémente la partie protocole CAN ainsi que la couche physique de type bifilaire différentielle grâce au 82C250.

3.1.1 Description



- IC1: SJA1000
- IC2: 82C250
- IC3: EP600
- X1: Quartz 10MHz

– R1: strap (définit la pente des transitions)
– C1 à C4: 100 nF (découplage)
– C5, C6: 33 pF

Les signaux de contrôle venant du PowerPC lors de cycles de lecture/écriture n'ont pas tout à fait le même format que ceux attendus par le SJA1000. La "glue logic" a été inclue dans un EPLD, ce qui permet une adaptation très simple à d'autres microcontrôleurs.

Le choix des broches rend possible l'utilisation d'un PAL22V10 à la place de l'EP600.

Programme de l'EPLD

```
SC/ML/VO
ESIEE
March 1999

OPTIONS: TURBO=ON
PART: 5C060

INPUTS: CS5@2
        CS6@3
        OE@4
        WBE0@5

OUTPUTS: ALE@22
        CS@21
        RD@20
        WR@19

NETWORK:

CS5=INP(CS5)
CS6=INP(CS6)
OE =INP(OE)
WBE=INP(WBE0)

ALE1=NOT(CS5)
WRITE=OR(WBE,ALE1)

ALE=CONF(ALE1,VCC)
CS=CONF(CS6,VCC)
RD=CONF(OE,VCC)
WR=CONF(WRITE,VCC)

END$
```

3.1.2 Utilisation

Les adresses et les données doivent être multiplexées sur le bus du SJA1000, c'est-à-dire que l'adresse est présentée en premier, puis ensuite viennent les données.

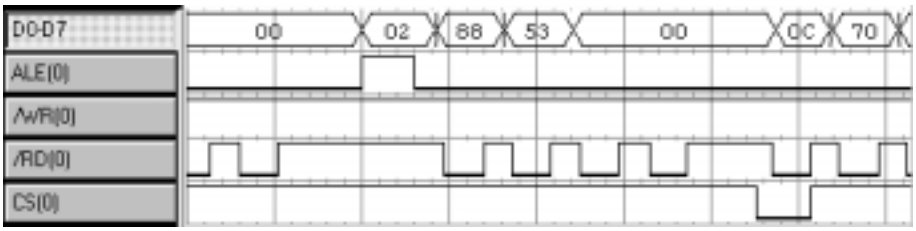


FIG. 3.1 – *Lecture d'un registre*

La lecture d'un registre se fait alors en deux temps : écriture de l'adresse (avec le CS5 du PPC positionné), puis lecture des données (avec CS6). De même l'écriture se fera en deux fois.

3.1.2.1 Macros de lecture-écriture

La lecture et l'écriture dans les registres du SJA1000 ont été implémentées sous forme de macros (fichier `rwsja.h`)

Macros de lecture-écriture

```
/*** Macros de lecture-ecriture SJA1000-PPC***/

/* Macro wrsjai : ecrit la donnee d dans le registre r */
   .macro  wrsjai d, r

    li   r2,\r
    stb    r2,0(r18)
    li   r2,\d
    stb    r2,0(r19)

   .endm


/* Macro wrsja : registre p1 du PPC -> p2 du SJA1000 */
   .macro  wrsja p1, p2

    li     r2,\p2
    stb    r2,0(r18)
    stb    \p1,0(r19)

   .endm


/* Macro pour lire une donnee du SJA (registre p1 -> reg p2) */
   .macro  rdsja p1, p2

    li     r2,\p1
    stb    r2,0(r18)
    lbz    \p2,0(r19)
```

```
.endm
```

3.1.2.2 Exemples d'utilisation

`wrsjai 0xDB,OUTPUT_CONTROL`
Pour écrire 0xDB dans le registre OUTPUT_CONTROL

`wrsja r3,TX_DATA`
Pour écrire l'octet de poids faible du registre r3 dans le registre TX_DATA

`rdsja STATUS,r3`
Pour lire le registre STATUS et placer le résultat sur l'octet de poids faible de r3

3.2 Carte SLIO

3.2.1 Description

Cette carte de démonstration est construite autour du SLIO[1] 82C150 (Philips). La configuration des entrées/sorties a été choisie de manière à donner une bonne idée des possibilités de ce SLIO. Elle a été développée et routée sous Protel.

- P0-P3 : ID / Entrées digitales (switches)
- P5 : Entrée analogique ADC1
- P6 : Entrée analogique ADC2 (potentiomètre)
- P7-P9,P11,P12 : Sorties digitales (LEDs)
- P13 : Sélection des entrées
- P15-P17 : Réservé multiplexage ADC

Les 4 entrées digitales sont multiplexées avec les 4 bits d'ID, eux aussi déterminés avec des switches. C'est un EPLD qui assure le multiplexage, commandé par la pin 14 du SLIO (qui n'est donc plus disponible comme entrée/sortie).

Les LEDs sont montées sur des connecteurs, ce qui permet d'utiliser ces sorties pour une autre utilisation.

Une des entrées analogiques est utilisée par un potentiomètre, l'autre peut servir pour un retour tachymétrique (pour un asservissement de moteur), ou pour toute autre source de tension entre 0 et 5V. Le multiplexage des entrées analogiques (en interne) occupe malheureusement 3 broches supplémentaires (P14 à P16) comme l'indique le schéma.

Enfin les sorties analogiques de type DPM (Discrete Pulse Modulation) sont amplifiées par des buffers (7407) avant d'attaquer des transistors de puissance. L'emploi de transistors PNP plutôt que NPN n'a pas de raison particulière, il est seulement dû à l'approvisionnement.

3.2.2 Utilisation

3.2.2.1 Calibration

Comme décrit dans la description du 82C150, il est nécessaire, pour que le SLIO reste calibré, de lui envoyer régulièrement un *calibration message*. Cette calibration doit se faire tous les 3000 à 8000 bit-times selon la documentation de Philips ([3]).

1. Serial Linked Input/Output

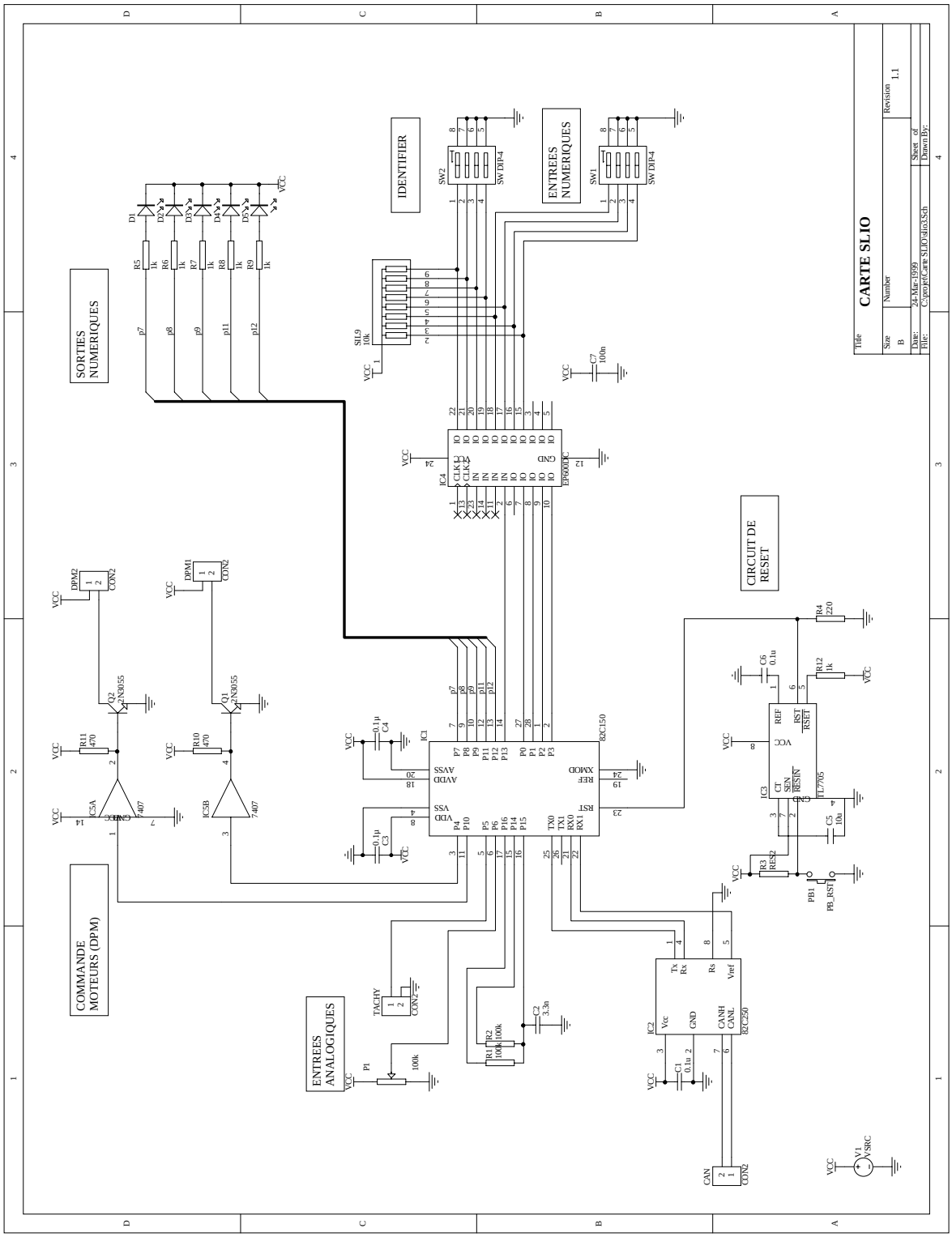


FIG. 3.2 – *Schéma de la carte SLIO*

Par exemple, pour un débit de 50 kbit/s, soit un bit-time de 20 μs, 5000 bit-times durent un dixième de seconde. Notre gestion des calibrations est décrite dans la partie Programmation PowerPC (page 39).

3.2.2.2 Pilotage des sorties analogiques et digitales

Il est tout d'abord nécessaire de configurer le registre OUTPUT_ENABLE des SLIOs. Sur la carte SLIO, étant donné les sorties utilisées, ce registre doit être mis à la valeur 0x3F90.[2]

Une fois la configuration faite,

- Pour une sortie digitale écrire dans le registre DATA_OUTPUT.
- Pour une sortie DPM écrire dans les registre DPM1 ou DPM2.

3.2.2.3 Lecture des entrés analogiques ou digitales

Pour lire l'état des entrés digitales, il suffit de faire une lecture du registre DATA_INPUT Pour lire la valeur des ADCs, il faut configurer le registre ANALOG_CONFIGURATION[3]:

- Ecrire 0xE0 pour une conversion sur l'ADC1
- Ecrire 0xC0 pour une conversion sur l'ADC2.

Ces valeurs spécifient à la fois la configuration interne des multiplexeurs internes du SLIO (pour "router" le signal jusqu'au module ADC), et la demande de conversion (bit 8)

Une fois ces donnés reçues, le SLIO concerné effectue la conversion, et envoie une trame contenant le contenu de son registre ADC, c'est-à-dire le résultat de la conversion.

Moins un appareil remplit de fonctions, plus il les remplira parfaitement.
Remarque inquiète : Les ordinateurs sont les appareils les plus multifonctionnels qui soient, et notre société repose dessus.
Principe des Appareils Multifonctions.

2. Pour effectuer cette affectation, se réfèrer à la page 39
3. Pour effectuer cette configuration voir page 39

Chapitre 4

Programmation PPC

4.1 Bibliothèque `can.s`

La bibliothèque `can.s` comporte les fonctions de base nécessaires à l'utilisation de la carte SJA1000-PPC:

`sja_init_pelican :` Initialise le SJA1000
`receive_trame :` Réception d'une trame sur le bus
`send_trame :` Envoi d'une trame sur le bus
`send_trame_sans_verif :` Envoi d'une trame prioritaire sur le bus
`print_trame :` Affiche une trame sur un terminal

Ces fonctions sont toutes programmées en assembleur.

4.1.1 Représentation machine des trames CAN

Pour rendre plus aisée la manipulation des trames CAN avec le PPC403, nous avons défini un format de représentation, que nous avons utilisée dans chacune des procédures de la bibliothèque `can.s`

Afin d'optimiser le traitement des trames, le format retenu est très proche de celui des registres du SJA1000:

r3	XX	FFI	ID1	ID2
r4	Data1	Data2	Data3	Data4
r5	Data4	Data5	Data6	Data7

4.1.2 Le fichier `pelican.h`

Le fichier `pelican.h` comporte les définitions d'équivalence de tous les registres du SJA1000 lorsque celui-ci est en mode PeliCAN

4.1.2.1 Procédure `sja_init_pelican`

Le procédure `sja_init_pelican` permet d'initialiser le SJA1000 en mode PeliCAN. La structure de cette procédure est celle donnée par la notice d'application de Philips ([4]).

1. Demande de passage en mode RESET
2. Attente que le passage soit fait
3. Demande de passage en mode PeliCAN
4. Attente que le passage soit fait
5. Configuration des Acceptance Masks
6. Configuration du bit timing
7. Configuration les autorisations d'interruptions (INTERRUPT_ENABLE)
8. Configuration les drivers de sortie (OUTPUT_CONTROL)
9. Demande de passage en mode normal
10. Attente que le passage soit fait

D'autre part, la procédure donnée ici est une configuration possible, mais il est évident qu'elle peut être adaptée au cas-par-cas.

En particulier, la version donnée laisse passer tous les messages vers le buffer de réception... ce qui n'est pas toujours souhaitable !

Il est à noter que l'initialisation du SJA1000 provoque la perte du contenu des buffers de réception et de transmission.

Procédure `sja_init_pelican`.

```
/*************************************
 * sja_init_pelican                  *
 * Initialisation du SJA1000         *
 * params:aucun                      *
 * resultat:aucun                    *
 * GPR modifies:aucun                *
 *************************************/
sja_init_pelican:

    push r3

attente_mode_reset:

    /* passage en reset */
    wrsjai  0x01,MODE

    rdsja   MODE,r3
    andi.   r3,r3,0x01
    beq     attente_mode_reset

attente_pelican:

    /* Passage en Mode PeliCAN (bit 7) */
    wrsjai  0x80,CLOCK_DIVIDER;
```

```
    rdsja   CLOCK_DIVIDER,r2
    andi.   r2,r2,0x80
    beq     attente_pelican

    /* on accepte tout */
    wrsjai  0xFF,ACCEPTANCE_MASK_0
    wrsjai  0xFF,ACCEPTANCE_MASK_1
    wrsjai  0xFF,ACCEPTANCE_MASK_2
    wrsjai  0xFF,ACCEPTANCE_MASK_3

    /* doit etre regle en fct des autres noeuds */
    /* Bit time=20us pour un Xtal de 16MHz*/

    wrsjai  0x07,BUS_TIMING_0
    wrsjai  0x4D,BUS_TIMING_1

    /*Desactivation des interruptions */
    wrsjai  0x00,INTERRUPT_ENABLE

    wrsjai 0xDB,OUTPUT_CONTROL

    /* Bit 0 = 0: Sortie du mode RESET
       Bit 1 = 0: Pas le mode Listen Only (donne des ACK)
       Bit 2 = 0: Pas le mode Self Test
       Bit 3 = 1: Acceptance filter mode = single */

attente_fin_mode_reset:

     wrsjai 0x08,MODE

     rdsja       MODE,r3
     andi.       r3,r3,0x01
     bne         attente_fin_mode_reset

mode_normal:

       pop r3
       blr
```

4.1.2.2 Procédure `receive_trame`

La procédure `receive_trame` permet de regarder dans le buffer de réception du SJA1000 afin de voir si une trame y a été reçue. Si tel est le cas, la trame présente sera mise dans les registres r3, r4, r5 du PPC selon le format défini précédemment.

1. Test du bit 0 du registre STATUS du SJA1000. Si 1: on a reçu une trame. Si 0: le buffer est vide on ressort de la procédure.
2. Positionnement r6 à 1 pour signaler la présence d'une trame valide dans r3, r4, r5
3. Lecture des registres du buffer de réception: RX_DATA, RX_DATA+1...

4. Libération la place du dernier message dans le buffer (positionnement du bit 2 du registre COMMAND)
5. Si une surchage a eu lieu, on réinitialise l'indicateur (positionnement du bit 3 du registre COMMAND)

Procédure `receive_trame`.

```
/*****************************************************************
 * RECEIVE_TRAME                                                 *
 * Reception d'une trame.                                        *
 * Registres de resultat: r3 = ID, r4 = Data 1-4, r5 = Data 5-8  *
 *                        r6=0 si pas de trame, 1 sinon          *
 *****************************************************************/
receive_trame:

        pushlr
        li      r3,0x0

attente_trame:

        rdsja   STATUS,r6
        andi    r6,r6,0x01     /* Receive buffer plein ? */
        beq     fin_receive_trame   /* si =0 on sort */

        rdsja   RX_FRAME_INFO,r3
        slwi    r3,r3,0x8

        rdsja   RX_ID_1_S,r6     /* ID */
        or      r3,r3,r6
        slwi    r3,r3,0x8

        rdsja   RX_ID_2_S,r6
        or      r3,r3,r6

        rdsja   RX_DATA_S,r4    /* Data  1*/
        slwi    r4,r4,8

        rdsja   RX_DATA_S+1,r6    /* Data 2 */
        or      r4,r4,r6
        slwi    r4,r4,8

        rdsja   RX_DATA_S+2,r6    /* Data 3 */
        or      r4,r4,r6
        slwi    r4,r4,8

        rdsja   RX_DATA_S+3,r6      /* Data 4 */
        or      r4,r4,r6

        rdsja   RX_DATA_S+4,r5    /* Data 5 */
        slwi    r5,r5,8

        rdsja   RX_DATA_S+5,r6    /* Data 6 */
        or      r5,r5,r6
        slwi    r5,r5,8

        rdsja   RX_DATA_S+6,r6    /* Data 7 */
```

```
        or      r5,r5,r6
        slwi    r5,r5,8

        rdsja   RX_DATA_S+7,r6    /* Data 8 */
        or      r5,r5,r6

        wrsjai  0x04,COMMAND      /* Release receive buffer */

        li      r6,0xFF

        /* clear le data overrun status si il y a lieu*/
        rdsja   STATUS,r2
        andi.   r2,r2,0x02
        beq     fin_receive_trame
        wrsjai  0x08,COMMAND

fin_receive_trame:

        poplr
        blr
```

4.1.2.3 Procédure `send_trame`

La procédure `send_trame` permet d'envoyer sur le bus CAN une trame présente dans les registres r3, r4, r5 dans le format défini auparavant.

La structure de la procédure est celle donnée par la notice d'application de Philips ([4]) :

1. Attente de la disponibilié du buffer de transmission (i.e attente que son contenu ait été transmis)
2. Ecriture des octets de contrôle dans le registre TX_FRAME_INFO du SJA1000
3. Ecriture des données dans le buffer : TX_DATA,TX_DATA+1...
4. Demande de transmission : positionnement du bit 0 du registre COMMAND).

4.1.2.4 Procédure `send_trame_sans_verif`

La procédure `send_trame_sans_verif` est quasiment identique à la procédure `send_trame`. La différence réside dans le fait que la procédure `send_trame_sans_verif` ne vérifie pas la disponibilité du buffer de transmission avant d'écrire dedans. Elle est donc susceptible d'en écraser le contenu.

La procédure `send_trame_sans_verif` est donc à utiliser avec précautions. Elle est utile lorsque l'utilisateur à besoin de transmettre un message en priorité absolue par rapport aux autres.

Procédure `send_trame_sans_verif`.

```
/*************************************************************
 * SEND_TRAME_SANS_VERIF                                     *
 * Envoi d'une trame sans verification du "tx buffer ready". *
```

```
 * Registres: r3 = FFI, r4 = Data 1-4, r5 = Data 5-8         *
 * NB: Ces registres ne sont pas modifies.                   *
 ************************************************************/
send_trame_sans_verif:

        pushlr
        push r3

        wrsja   r3,TX_ID_2_S

        srwi    r3,r3,8
        wrsja   r3,TX_ID_1_S

        srwi    r3,r3,8
        wrsja   r3,TX_FRAME_INFO

        mr      r3,r4
        wrsja   r3,TX_DATA_S+3
        srwi    r3,r3,8
        wrsja   r3,TX_DATA_S+2
        srwi    r3,r3,8
        wrsja   r3,TX_DATA_S+1
        srwi    r3,r3,8
        wrsja   r3,TX_DATA_S

        mr      r3,r5
        srwi    r3,r3,8
        wrsja   r3,TX_DATA_S+7
        srwi    r3,r3,8
        wrsja   r3,TX_DATA_S+6
        srwi    r3,r3,8
        wrsja   r3,TX_DATA_S+5
        srwi    r3,r3,8
        wrsja   r3,TX_DATA_S+4

        wrsjai  0x01,COMMAND /* envoi */


        pop r3
        poplr
        blr
```

4.2 Bibliothèque `slio.s`

La bibliothèque `slio.s` comporte les fonctions de base pour la gestion de nœuds de type SLIO-82C150.

`calibration_slio` : Calibre tous les SLIOs présents sur le bus
`send_calibration` : Envoie un message de calibration sur le bus
`make_slio_ffi` : Initialise r3 (FFI+ID1+ID2) en vue de l'envoi d'un message à un SLIO

`wr_reg_slio :` Ecrit une donnée dans un des registres d'un SLIO
`test_slio_id :` Teste si le message contenu dans r3, r4, r5 provient d'un SLIO

4.2.1 Le fichier `slio.h`

Le fichier `slio.h` contient les définitions d'équivalence des registres des SLIOs

4.2.1.1 Procédure `calibration_slio`

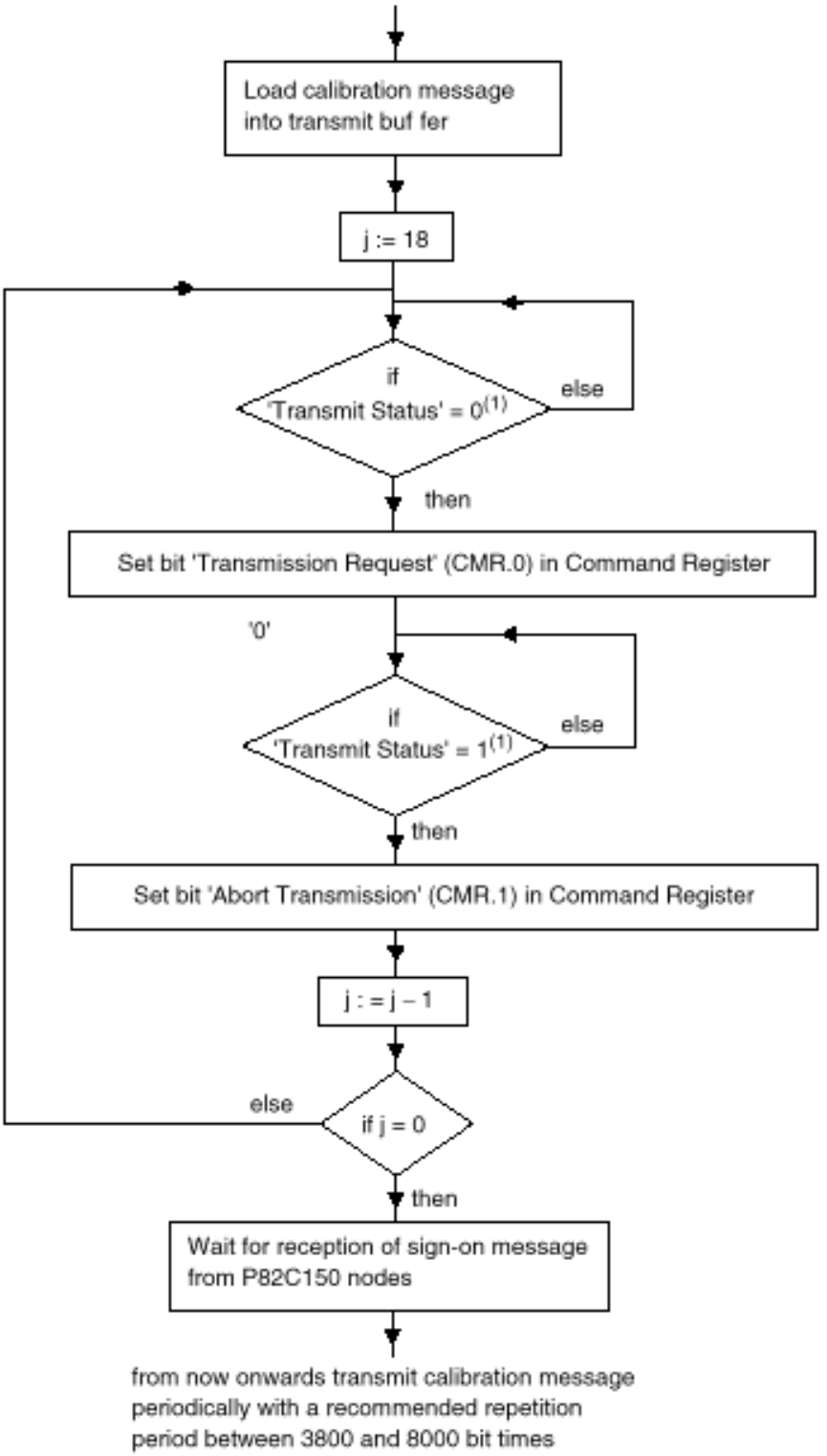


FIG. 4.1 – *Calibration des SLIOs.*

Cette procédure permet à l'utilisateur d'effectuer la première calibration des SLIOs du bus. Comme expliqué auparavant, la calibration des nœuds de type SLIO est indispensable à toute communication avec des SLIOs.

Procédure `calibration_slio`.

```
/**********************
 * CALIBRATION_SLIO   *
 * parametres: aucun  *
 * GPR modifies:aucun *
 **********************/
calibration_slio:

    pushlr
    push r3
    push r4

    li      r4,0x30


attente_buffer:

    rdsja   STATUS,r3
    andi.   r3,r3,0x04     /* TXbuffer pret ?  */
    beq     attente_buffer /* si = 0 on attend */

    /* on place le message de calibrationd dans le buffer*/
    wrsjai  0x40,TX_ID_2_S
    wrsjai  0x15,TX_ID_1_S
    wrsjai  0x02,TX_FRAME_INFO

    wrsjai   0xAA,TX_DATA_S
    wrsjai   0x04,TX_DATA_S+1

    /* on attend que la transmission precedente soit terminee*/
attente_fin_transmission_prec:
    rdsja  STATUS,r3
    andi.  r3,r3,0x20
    bne    attente_fin_transmission_prec

    /* on fait une demande d'envoi*/
    wrsjai  0x01,COMMAND

    /* On attend que la transmission ait commence*/
attente_debut_transmission:
    rdsja  STATUS,r3
    andi.  r3,r3,0x20
    beq    attente_debut_transmission

    /* on avorte la transmission*/
    wrsjai 0x02,COMMAND
```

```
/* on boucle 48 fois*/
boucle_de_calibration:

    subic.   r4,r4,1
    bne      attente_fin_transmission_prec

fin_calibration:

   pop r4
   pop r3
   poplr
   blr
```

NB: Au début de l'exécution de la procédure de calibration, les SLIOs n'acquittent pas les messages présents sur le bus. Dans le cas où seuls le PPC et les SLIOs sont prsents sur le bus, le SJA1000 du PPC comprendra que son message a été mal reçu et il renverra son message jusqu'à acquittement. C'est pourquoi on annule la transmission.

4.2.1.2 Procédure `send_calibration`

La procédure `send_calibration` est utilisée pour envoyer sur le bus un message de calibration destiné aux SLIOs présents sur le bus. Typiquement, cette procédure est utilisée pour effectuer la nécessaire calibration périodique des SLIOs.

Procédure `send_calibration`.

```
/*****************************
* SEND_CALIBRATION           *
* envoie un message de       *
* calibration sur le bus     *
* parametres:aucun           *
* registres modifies:r3,r4   *
*****************************/

send_calibration:

    pushlr


    /* 0x00,0000 0010 000 1010 1010 0 0000*/
    ld32b    r3,0x00021540

    /*          1010 1010 0000 0100*/
    ld32b    r4,0xAA040000

    bl       send_trame_sans_verif

    poplr
    blr
```

4.2.1.3 Procédure `make_slio_ffi`

La procédure `make_slio_ffi` est utilisée pour écrire dans r3 les informations nécessaires à l'envoi d'un message à un certain SLIO, spécifié dans r6. On utilisera pour cela la procédure `send_trame`.

Procédure `make_slio_ffi`.

```
/*******************************
 * MAKE_SLIO_FFI               *
 * initialisation de r3 pour   *
 * le slio precise sur r6      *
 * params:r6                   *
 * GPRs modifies:aucun         *
 *******************************/
make_slio_ffi:

    pushlr

    push r6
    push r7

    li   r3,0x0
    mr   r8,r6

FFI:

    /* FF=0,RTR=0,DLC=3*/
    li       r7,0b00000011
    slwi     r7,r7,16
    or       r3,r3,r7

id:

    /* r7=debut de l'id du SLIO (fixe)*/
    /* 0101 0000 1000 0000*/
    li       r7,0x5080
    or       r3,r3,r7

    /* r6=id*/
    andi.    r6,r6,0xF

    /* on recupere P0 ds r7*/
    andi.    r7,r6,0x1
    /* que l'on decale de 8+0 pr le mettre ds r3*/
    slwi     r7,r7,0x8
    or       r3,r3,r7

    /* on recupere P1 ds r7*/
    srwi     r6,r6,1
    andi.    r7,r6,0x1
    /* que l'on decale de 8+1 pr le mettre ds r3*/
```

```
    slwi     r7,r7,0x9
    or       r3,r3,r7

     /* on recupere P2 ds r7*/
    srwi     r6,r6,1
    andi.    r7,r6,0x1
    /* que l'on decale de 8+2 pr le mettre ds r3*/
    slwi     r7,r7,0xA
    or       r3,r3,r7

    /* on recupere P3 ds r7*/
    srwi     r6,r6,1
    andi.    r7,r6,0x1
    /* que l'on decale de 8+5 pr le mettre ds r3*/
    slwi     r7,r7,13
    or       r3,r3,r7

    /* r3=0000 0011 01P31 0P2P1P0  1000 000 */

     pop r7
     pop r6

     poplr
     blr
```

Exemple d'utilisation `li r6,0x0A`
`bl make_slio_ffi`

r3 est alors initialisé pour un envoi au SLIO d'ID 0x0A.

4.2.1.4 Procédure `wr_reg_slio`

La procédure `wr_reg_slio` est utilisée pour initialiser les registres r4 et r5 en vue de l'écriture d'une donnée dans un registre d'un SLIO.

La donnée à écrire est placée dans r7, et le numéro de registre dans r6. Pour l'envoi à un SLIO, il est évidemment nécessaire d'initialiser correctement le registre r3. On peut utiliser pour cela la procédure `make_slio_ffi`.

Procédure `wr_reg_slio`

```
/*****************************
* WR_REG_SLIO                *
* adapte une trame CAN       *
* pour ecrire r7 (sur 16b)   *
* dans r6 (sur 4b)           *
* ATTENTION: suppose r3 (FFI*
* et ID) deja mis a jour     *
```

```
* parametres:r4,r5,r6       *
* resultat:aucun            *
* GPR modifies:r4,r6        *
****************************/
wr_reg_slio:

    push     r7

    li       r4,0x0
    andi.    r7,r7,0xFFFF

    /* on met r6 sur les pds faibles du byte1*/
    slwi     r6,r6,24
    or       r4,r4,r6

    /* on met r7 sur les bytes2 et 3 */
    slwi     r7,r7,8
    or       r4,r4,r7

    pop      r7
    blr
```

Exemple d'utilisation La séquence de code ci-dessous écrit 0x32E5 dans le registre 0x7 (DATA_OUTPUT) du SLIO 0x0A.

```
li r6,0x0A
bl make_slio_ffi

li r7,0x32E5
li r6,0x3
bl wr_reg_slio

bl send_trame
```

4.2.1.5 Procédure `test_slio_id`

La procédure permet de tester si le message contenu dans r3,r4,r5 a un ID correspondant à un SLIO. Si oui, r6 est mis à 1,et le numéro de SLIO est mis dans r7.

Procédure `test_slio_id`

```
/*********************************
* TEST_SLIO_ID                   *
* renvoie r6=1 si le message     *
* contenu dans r3     vient      *
* d'un slio, et si oui, renvoie  *
* son N d'id dans r7             *
* GPR modifies:r6,r7             *
*********************************/
```

```
test_slio_id:
   pushlr


   /* on ne s'interesse qu'a l'id fixe d'un slio*/
   /*1101 1000 1110 0000*/
   andi. r6,r3,0xD8E0

   /* 0101 0000 1010 0000*/
   cmpi  0,0,r6,0x50A0
   /* s'il ne correspond pas on sort tt de suite*/
   bne   prov_non_slio

prov_slio:

   /* on recupere (P2 P1 P0) sur r6*/
   andi. r6,r3,0x700
   srwi  r6,r6,8

   /* on recupere P3 dans r7*/
   andi. r7,r3,0x2000
   srwi  r7,r7,10

   /* on y ajoute r6*/
   or    r7,r7,r6
   /*r7=0x(P3 P2 P1 P0)*/
   li    r6,0x1
   b     fin_test_slio_id

prov_non_slio:

   andi.  r6,r6,0x0

fin_test_slio_id:


   poplr
   blr
```

4.3 MoniCAN

MoniCAN est un outil de développement et de démonstration pour bus CAN, fait pour être associé à la carte SLIO décrite précédemment. De plus, le programme MoniCAN est un exemple d'application CAN utilisant les bibliothèques `can.s`, `slio.s` ainsi que le programme loader `loadCAN.s`

MoniCAN permet à un utilisateur de contrôler jusqu'à 16 cartes SLIO simultanément à l'aide d'un terminal de type VT100.

4.3.1 Caractéristiques

- Calibration et recensement de tous les SLIOs sur le BUS (jusqu'à 16)
- Commande des 5*16 sorties digitales
- Lecture des 4*16 entrées digitales
- Commande des 2*16 sorties DPM
- Lecture des 2*16 entrées ADC
- Programme uploadable par un superviseur
- Messages uploadables par un superviseur (mail, messages divers...)

4.3.2 Fonctionnement

```
                *** MoniCAN   (C) ESIEE 1999 ***
SLIO | DIGITAL OUTPUTS | DIGITAL INPUTS  | ADC1 | ADC2 | DPM1 | DPM2
  0  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  1  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  2  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
* 3  |  2000          |  6001           |  XX  |  XX  |  00  |  00
  4  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  5  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  6  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  7  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  8  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  9  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  A  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  B  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  C  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  D  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  E  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
  F  |  XXXX          |  XXXX           |  XX  |  XX  |  XX  |  XX
Last frame received : ID1:53 ID2:A0 DATA1:23 DATA2:20 DATA3:00

-----------------------------------------------------------------
 ID_slio:3 Data input:6001
 ID_slio:3 Data_output:2000
Next command >   SLIO:_     I/O:_    PIN:_     VALUE:__
```

FIG. 4.2 – *Ecran de contrôle de MoniCAN*

4.3.2.1 Initialisation

1. Initialisation de l'écran du terminal
2. Initialisation du SJA1000
3. Lancement de la procédure de calibration `calibration_slio`
4. Attente de la réception de trames

5. Pour chaque message de signature venant d'un SLIO :
   - Initialisation des sorties digitales et DPM des SLIOs reconnus
   - Signalisation de la présence des SLIOs sur l'écran du terminal

4.3.2.2 Fonctionnement normal

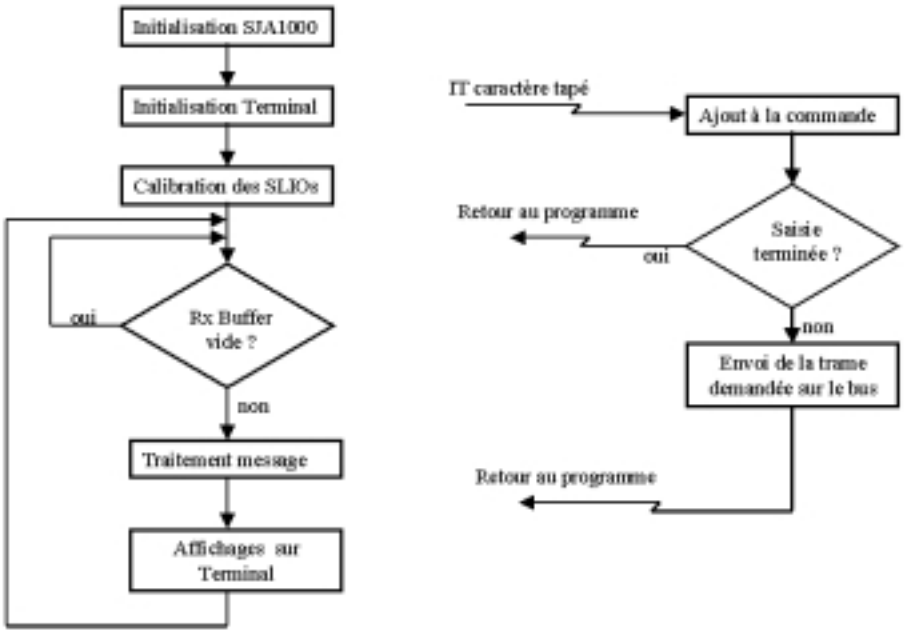


FIG. 4.3 – *Fonctionnement MoniCAN*

Les commandes entrées par l'utilisateur sont gérées par interruptions, alors que la réception et le traitement des trames est fait par scrutation. Ce choix ce justifie car le traitement d'une trame reçue est toujours le même quels que soient les évènements antérieurs, alors que la saisie des commandes se fait séquentiellement.

Lors de la saisie complète d'une commande par l'utilisateur, une trame est envoyée au SLIO demandé. Cette trame spécifie au SLIO le registre à modifier et la valeur à lui affecter. Une fois cette affectation faite, le SLIO renvoie une trame comportant la nouvelle valeur du registre. Cette trame est alors capturée, et la valeur du registre affichée sur le terminal.

Ainsi, les 'états des entrés/sorties des SLIOs ne sont affichés que lorsque les SLIOs les ont confirmés, et non pas lorsque l'utilisateur en a fait la demande.

4.3.2.3 Traitement d'une trame reçue

1. Test de l'ID:
   - Si l'ID est l'ID d'un SLIO, on met à jour l'affichage du registre concerné sur l'écran du terminal.

- Si l'ID correspond à une demande d'affichage d'un message (mail,...). On prévient l'utilisateur par un message sur l'écran du terminal
- Si l'ID correspond à une demande de chargement de fichier par un superviseur, on exécute le programme loadCAN (qui se trouve en FLASH-EEPROM)

2. Retour à la boucle de scrutation

4.3.3 Utilisation

4.3.3.1 Envoi d'une commande

1. Taper le numéro de SLIO demandé (de 0 à F)
2. Taper 'I' pour une demande d'entrée et 'O' pour une demande de sortie
3. Taper le numéro de Pin demandée
4. Taper s'il y a lieu la valeur de la sortie (00/01 pour une sortie digitale, 00-FF pour une sortie DPM)

NB:
- Pour un bon fonctionnement, le clavier du terminal doit être en mode CAPS-LOCK, et le baudrate à 19200 bauds.
- L'affichage en bas de l'écran permet de suivre la commande tapée

4.3.3.2 Lecture d'un message envoyé par un superviseur

Lorsqu'un superviseur désire afficher un message/mail sur l'écran de l'utilisateur, il envoie un message avec un ID particulier. Le message `you've got mail !(press Ctrl R to read it)` s'affiche en bas de l'écran du terminal. Lorsque l'utilisateur tape effectivement Ctrl R, l'écran de gestion des SLIOs s'efface, et on signale au superviseur que l'on est prêt à afficher son message.

4.3.3.3 Chargement d'un nouveau programme

Pour charger un nouveau programme dans le PPC (par exemple une nouvelle version de MoniCAN), le superviseur doit envoyer une Remote Frame d'ID 0. Une fois cette trame reçue, Monican effectue un branchement vers le programme loader en FLASH-EEPROM, lequel signale au superviseur qu'il est prêt à recevoir le programme. Pour plus de détails, voir le chapitre sur le programme `loadcan.s`.

Remarque importante : A chaque compilation du loader, il faut modifier la valeur de "loader" dans le fichier `config.h` à l'adresse du label "loader" du fichier `.L` généré lors de la compilation du programme `loadcan.s`

4.4 Le Loader CAN

4.4.1 Fonctionnement

Le loader CAN (`loadcan.s`) permet à un superviseur de transmettre à un noeud PPC, un fichier s29, qui sera éxecuté.

La seule exigence pour le superviseur est de pouvoir transmettre un fichier ASCII par bus CAN.

4.4.1.1 Transmission du programme.

A la mise sous tension, le loader CAN se met en attente d'une trame de commande ("Remote frame") d'identificateur 0 et de DLC 0. Le superviseur doit envoyer une telle trame lorsqu'il est prêt à envoyer le fichier .s29. Le loader répond à cette trame de commande par une trame de données ayant le même ID, et ne comportant aucune données. Cette trame signale au superviseur qu'il peut commencer la transmission.

Le superviseur doit ensuite transmettre le fichier s29 en ASCII avec 8 octets de données par trame, sauf pour la dernière trame (le test du DCL est le moyen utilisé par le loader pour détecter la fin de la transmission). Une fois la transmission ASCII terminée, le loader convertit le fichier ASCII en fichier binaire.

4.4.1.2 Implémentation en mémoire.

Le loader charge le fichier s29, en ASCII à l'adresse `debut_fichier`, specifiée dans le fichier de configuration `config_loader.h`.

Puis, lors de la conversion, l'exécutable est implanté a l'adresse specifiée dans le s29.

4.4.2 Utilisation

4.4.2.1 Compilation du loader

Bien évidemment, le loader doit être implementé en Flash-EEPROM afin d'autoriser la chargement d'un programme dès la mise sous tension. Par consèquent, le linkage du loader doit inclure le bootstrap (objet boot.o).

4.4.2.2 Compilation du programme à loader

Le programme à charger ne doit en revanche pas etre linké avec boot.o, et il ne doit pas non-plus inclure de symbole global de branchement au boot. Pour contrôler l'adresse d'implantation en mémoire du programme chargé, il suffit de préciser l'adresse dans le fichier de link du programme à loader.

4.4.2.3 Exemple

Pour un développement facile, il est utile d'avoir deux scripts de compilation et deux fichiers de link (un pour le loader, et un pour le programme à charger)

Le programme ci-après est un exemple fichier de link pour le loader. Le fichier de loader y est implémenté en debut de Flash-EEPROM (en laissant 2000h de place pour le procédures d'interruption)

Exemple fichier de link pour le loader.

```
SECTIONS
{
   output_boot 0xfffffffc :
   {
   boot.o(.text)
   }
   output 0xfffe2000 :
```

```
   {
   *(.text)
   *(.data)
   }
}
```

Le programme ci-après est un exemple de fichier de link pour le programme à loader. On peut y remarquer l'absence de link avec l'objet boot.o (le branchement étant effectué par le loader).

Exemple fichier de link pour le programme à loader.

```
SECTIONS
{

   output_it 0x30000 :
   {
   ith.o(.text)
   }
   output 0x32000 :
   {
   *(.text)
   *(.data)
   }
}
```

Avec un tel fichier de link, le programme à charger sera implanté en 30000h (les procédures d'interruption se situant entre 30000h et 32000h).

A noter que l'adresse d'implémentation du programme loadé doit bien évidemment se situer en RAM.

L'arrivée des microprocesseurs dans les équipements de sécurité automobile permettra de faire des accidents high-tech. Prédiction de Krashtest Dummy.

Chapitre 5

Environnement JerryCAN

CANAPI et JerryCAN constituent un ensemble de bibliothèques et de programmes permettant d'utiliser une carte SJA 1000 pour PC. Ils sont entièrement développés en Java 2[1].

CANAPI est une API[2] permettant de gérer des trames CAN et d'utiliser la carte SJA 1000. JerryCAN est un outil permettant de recevoir et d'envoyer des trames CAN, ainsi que de contrôler la SJA 1000. Cette application se base sur CANAPI.

5.1 Carte SJA 1000 pour PC.

5.1.1 Description.

Le SJA 1000 possède un unique bus Adresse - Données. La carte pour PC simplifie l'utilisation du composant en démultiplexant le bus et en permettant un vrai mapping en mémoire de la carte. Il devient alors possible de l'utiliser en effectuant de simple opérations de lecture - écriture en mémoire.

5.1.2 Librairies en C.

Les fichiers **.h** décrits ci-dessous définissent des constantes pour tous les registres un mode BasicCAN et en mode PeliCAN. Ils contiennent aussi un certain nombre de procédures utiles telles que des procédures d'initialisation, d'envoi et de réception de trames.

Le terme de librairie au sens C est un peu abusif ici car tout le code source est contenu dans les fichiers **.h**. Cela en simplifie l'utilisation.

Les procédures ont été développées et testées avec le compilateur **bcc** de Borland dans sa version 4.52 sous Windows 95. Elles ne devraient pas être utilisables sous Windows NT car elles adressent directement la mémoire.

5.1.2.1 Mode BasicCAN.

Le fichier `basiccan.h` contient :

- les `define` pour tous les registres du SJA 1000 en mode BasicCAN.
- les `define` pour faciliter la configuration des registres.

1. qui est en fait le JDK 1.2.
2. *Application Programming Interface*, interface de programmation.

– une structure `Message` représentant un message à envoyer. Cette structure est composée des champs :

id1 : 8 bits de poids forts de l'ID.
id2 : 3 bits de poids faibles de l'ID, RTR, DLC.
data : un tableau de 8 octets.

– des fonctions de base d'utilisation du SJA 1000 :

init_sja1000_it : initialisation du SJA 1000.
init_sja1000 : initialisation du SJA 1000.
send_trame : envoi d'une trame.
receive_trame : réception d'un message.

Le fichier `util.h` contient des fonctions utiles :

print_reg : affichage de l'état de tous les registres en BasicCAN.
show_status : affichage du détail du registre STATUS.
show_control : affichage du détail du registre CONTROL.

Le fichier `slio.h` contient de fonctions pour le SJA 1000 en mode BasicCAN permettant de calibrer des SLIOs sur le bus et de leur envoyer des ordres.

calibrate_slio : calibration et initialisation du SLIO. Le "sign-on" message est renvoyé.
send_slio : envoie une trame au SLIO, puis attend, récupère et renvoie le message renvoyé par le SLIO.

5.1.2.2 Mode PeliCAN

Le fichier `pelican.h` contient :

– les `define` pour tous les registres du SJA 1000 en mode PeliCAN.
– une structure `Message` représentant un message à envoyer. Cette structure est composée des champs :

id1 : 8 bits de poids forts de l'ID.
id2 : 3 bits de poids faible de l'ID.
dlc : nombre d'octets à envoyer, -1 si c'est une Remote Frame.
data : un tableau de 8 octets.

– des fonctions de base d'utilisation du SJA 1000 :

init_sja1000_peli_acc : initialisation du SJA 1000.
init_sja1000_peli : initialisation du SJA 1000.
send_trame_peli : envoi d'une trame.
receive_trame_peli : réception d'un message.

Le fichier `utilp.h` contient des fonctions utiles en PeliCAN :

print_reg_peli : affichage des registres.
show_status : affiche l'état du registre STATUS.
print_little : affiche le message passé en paramètres sur une ligne.

5.1.3 Programmes

Les programmes décrits dans le tableau ci-dessous utilisent les librairies. Le code source en C se trouve dans les fichiers `programme.c`. Les programmes ont été développées et testées avec le compilateur **bcc** de Borland dans sa version 4.52 sous Windows 95 (en mode DOS).

Nom	Description	Mode
control	Affiche le détail du registre CONTROL.	BasicCAN
decode	Décode une trame CAN. La trame est entrée bit par bit.	Indifférent
nbrx	Affiche le nombre de message dans le buffer de réception.	PeliCAN
recbper	Affiche en permanence les trames reçues.	BasicCAN
receiveb	Attend une trame et l'affiche.	BasicCAN
receivep	Attend une trame et l'affiche.	PeliCAN
recpper	Affiche en permanence les trames reçues.	PeliCAN
regb	Affiche les registres.	BasicCAN
regp	Affiche les registres.	PeliCAN
rx	Affiche le contenu du buffer de réception.	BasicCAN
sendb	Envoie une trame de 8 octets sur le bus et attend un acquittement.	BasicCAN
sendfile	Envoie un fichier sur le bus CAN. Utilisation : `sendfile ID FILE_NAME`	BasicCAN
sendp	Envoie une trame de 8 octets sur le bus et attend un acquittement.	PeliCAN
sendrem	Envoie une Remote Frame.	PeliCAN
slioadc	Effectue un polling sur un ADC du SLIO.	BasicCAN
slioinit	Initialise le SLIO.	BasicCAN
sliomphi	Initialise le SLIO. Amélioration par rapport aux indications de Philips.	BasicCAN
sliotest	Initialise et teste le SLIO.	BasicCAN
statper	Affiche le registre STATUS dès qu'il change.	Indifférent
status	Affiche le détail du registre STATUS.	Indifférent

5.2 Avantages de Java pour le protocole CAN.

A priori, l'utilisation du langage de haut niveau qu'est Java pour un développement très bas niveau peut sembler curieuse. En fait, de nombreux arguments justifient ce choix.

5.2.1 Rapidité de développement

Un langage de haut niveau et possédant des bibliothèques extrêmement complètes permet de ne pas réinventer la roue à chaque fois et d'avoir une productivité bien supérieure à celle que l'on pourrait avoir en C par exemple.

Ceci s'inscrit dans une tendance générale du développement logiciel qui veut qu'on n'utilise les langages de bas niveau tels que le C que lorsque c'est absolument nécessaire (besoin de contrôler le hardware précisément, pas d'autres langages disponibles), et que l'on s'oriente de plus en plus vers des outils de type AGL[3].

3. Atelier de Génie Logiciel.

5.2.2 Réutilisabilité

Un des grands avantages qu'apporte langage un orienté objet est qu'il permet de créer un code qui sera plus facilement réutilisable qu'avec un langage utilisant une autre philosophie. Dans cette optique, Java va plus loin que le C++, tout en restant plus simple et plus pratique à utiliser que des langages orientés objets plus "extrêmes" tel que scriptsizeTalk.

Ces qualités ont permis la création d'une API réutilisable dans d'autres projets nécessitants l'utilisation d'un bus CAN.

5.2.3 Portabilité

Une des qualités originelles de Java est sa portabilité, le "write once, run anywhere". CAN API ne respecte pas tout à fait cette règle puisqu'il a fallu développer une petite librairie qui permette d'accèder à la carte. En effet, cette opération étant dépendante du système d'exploitation, il n'est pas possible de l'effectuer directement à partir de Java (sinon le code ne serait plus portable).

Néanmoins, il suffira de réécrire cette petite librairie sous un autre environnement pour pouvoir y utiliser immédiatement toutes les classes.

Actuellement, seule une librairie pour Windows 95 a été développée.

5.2.4 Développement de Java dans les applications embarquées

Le choix de Java se justifie aussi par l'importance de plus en grande qu'il prend dans les applications embarquées. En effet, utiliser Java dans des téléphones portables, des autoradios, voire des cartes à puce présente l'avantage de grandement réduire les coûts de développement qui, historiquement propriétaires, nécessitaient d'être pratiquement recommencés de zéro à chaque nouvelle génération d'appareils.

Le bus CAN étant lui aussi lié à l'informatique embarquée, l'utilisation de Java pour contrôler un bus CAN se justifiait tout à fait.

On notera aussi qu'avec Jini [4], la nouvelle technologie de Sun, créateur de Java, la tendance actuelle est de plus en plus de rendre le réseau transparent pour l'utilisateur et pour les applications. Nul doute que le CAN sera aussi de plus en plus influencé dans l'avenir par cette tendance.

5.3 CANAPI.

CANAPI est la bibliothèque de base permettant de gérer le protocole CAN ainsi que le SJA 1000. Cette bibliothèque est composée de trois packages : `canapi.can20` contient des classes d'utilisation du protocole CAN, `canapi.sja1000` des classes pour utiliser le SJA 1000, et `canapi.app` diverses classes applicatives.

Les grandes caractéristiques de cette librairie sont :

- le support des modes BasicCAN et PeliCAN, et ceci de façon transparente,
- la simplicité de configuration du SJA 1000,
- le support des applications multithreadées.

4. Voir http://www.sun.com/jini/

D'autre part, toute la librairie a été développée dans l'optique de pouvoir la réutiliser pour d'autres applications.

5.3.1 Package `canapi.can20`.

Le package `canapi.can20` contient des classes modélisant des trames respectant le protocole CAN 2.0. Dans l'implémentation actuelle, seul le protocole CAN 2.0 A est supporté, mais la conception des classes les rend facilement extensible pour qu'elles supportent aussi le protocole CAN 2.0 B.

La classe `ID` modélise un identificateur de trame CAN. La classe représente par défaut un ID pour le protocole CAN 2.0 A. Il est possible de créer un ID à partir d'un entier (type `int`) ou d'une chaîne de caractères (objet de la classe `String`). Des méthodes utilitaires sont proposées pour ne récupérer que certains bits de l'ID. Enfin, la classe implémente aussi l'interface `Comparable`, ce qui permet d'utiliser `ID` dans des structures de données gérant le tri. Ainsi, le tri par ordre de priorités de messages à envoyer est immédiat.

La classe `Frame` est une classe abstraite modélisant les comportements communs des deux types de trames de données, les *Data Frames* et les *Remote Frames*. Au niveau de l'implémentation, cette classe ne contient qu'un seul champ de données, un objet de la classe `ID`.

Les deux classes `DataFrame` et `RemoteFrame` héritent de la classe `Frame` et rajoutent le comportement spécifique des deux types de trames.

Toutes les méthodes des classes de ce package ne sont pas synchronisées, il s'agit donc de les utiliser avec précautions dans une application multithreadée.

5.3.2 Package `canapi.sja1000`.

Le package canapi.sja1000 contient des classes permettant d'utiliser le SJA 1000.

5.3.2.1 Accès à la carte.

Le SJA 1000 est mappé en mémoire, son utilisation passe donc par des lectures et des écritures en mémoire. Ceci pose un problème dans le cas d'une utilisation avec Java. En effet, Java est conçu pour être indépendant de la plate-forme sur laquelle s'exécute l'application. Le langage ne possède donc aucune méthode pour écrire ou lire directement à une adresse en mémoire.

Pour résoudre ce problème, il est possible de passer par l'intermédiaire de JNI, *Java Native Interface*, qui permet d'appeler des procédures dans un autre langage que Java. La solution retenue a donc été d'écrire deux procédures en C effectuant l'une un `inportb` et l'autre un `outportb`, et de les appeler ensuite depuis les classes Java.

Ces méthodes sont appelées depuis la classe abstraite `MemoryMappedDevice` qui modélise une carte mappée en mémoire. Pour ensuite effectivement avoir une classe qui représente une carte, il faut créer une classe qui hérite de cette classe et qui pourra utiliser les méthodes `read` et `write`.

On notera que ces deux méthodes sont `protected`, il est en effet raisonnable de n'autoriser qu'une classe de type *carte* à les appeler. D'autre part, ces deux méthodes implémentent un adressage relatif afin d'en faciliter l'utilisation.

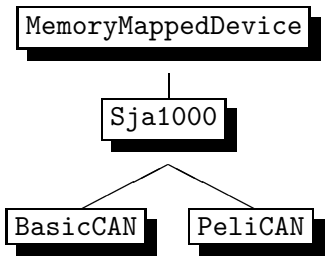


FIG. 5.1 – *Hiérarchie des classes représentant la carte.*

Cette solution possède malheureusement ses limites. En effet, la création de méthodes natives est contraire au principe de portabilité de Java. Pour utiliser les API sur une autre plate-forme, il faudra réimplémenter les méthodes C sur chaque plate-forme.

D'autre part, l'implémentation actuelle de ces méthodes sous Windows 95 utilise le mode de comptabilité MS-DOS, autrement dit la librairie DLL contenant les deux méthodes n'est pas un vrai driver 32 bits. En conséquence, les API ne sont pas utilisable sous Windows NT actuellement, et pour qu'elles le deviennent, il faudra écrire un vrai driver.

Autre limitation, il n'est pas possible d'utiliser les interruptions du SJA 1000 car la gestion des interruptions sous Windows passe obligatoirement par l'écriture d'un driver.

5.3.2.2 Comment porter CAN API sur un autre environnement.

Pour faire tourner CAN API sous un autre environnement que Windows, il faut réecrire le code du fichier `MemoryMappedDevice.c`, c'est-à-dire le code des procédures déclarées dans `canapi_sja1000_MemoryMappedDevice.h`. Ce dernier est généré par la commande :

```
javah canapi.sja1000.MemoryMappedDevice
```

Ensuite, `MemoryMappedDevice.c` doit être compilé sous forme d'une librairie. Sous Windows, avec Visual C++ 4 :

```
c:\Program Files\DevStudio\Vc\bin\vcvars32.bat
```

pour initialiser les variables du compilateur.

Puis :

```
cl -Ic:\jdk1.1.6\include -Ic:\jdk1.1.6\include\win32
-LD MemoryMappedDevice.c -Fedevice.dll
```

pour compiler la librairie.

5.3.2.3 Classe `Sja1000`.

Revenons sur le package `canapi.sja1000`. La classe `Sja1000` hérite de `MemoryMappedDevice` et représente une carte SJA 1000. Elle est abstraite et implémente toutes les méthodes communes aux deux modes de fonctionnement du SJA 1000 que sont les modes BasicCAN et PeliCAN. On a donc logiquement deux classes, nommées `BasicCAN` et `PeliCAN`, qui héritent de `Sja1000` et qui représentent une carte fonctionnant respectivement en mode BasicCAN et PeliCAN.

Ces classes contiennent toutes les méthodes pour utiliser facilement le SJA 1000. Outre des méthodes pour lire un bit du registre STATUS ou lire les registres d'erreurs par exemple, les principales méthodes sont `init` (qui prend en paramètre un objet de configuration sur lequel

nous reviendrons) pour initialiser le SJA 1000, `send` pour envoyer une trame et `receive` pour recevoir une trame. Détail important, ces méthodes en particulier, bien que fonctionnant différemment dans les modes BasicCAN ou PeliCAN, sont utilisables à partir d'une référence de type `Sja1000`[5].

A l'utilisation, cela peut se traduire par un code du style :

```
Sja1000 sja;
if (isBasic) sja = new BasicCAN();
else sja = new PeliCAN();
...
sja.send(aFrame);
```

On se référera à la documentation Javadoc pour la liste et le détail de toutes les méthodes.

5.3.2.4 Configuration du SJA 1000.

Pour ne pas rendre la taille et la complexité de ces classes trop importante, toute la gestion de la configuration du SJA 1000 a été séparée dans d'autres classes spécialisées. Ce sont les classes `SjaConfig`, `BasicCANConfig` et `PeliCANConfig`, où dans le même esprit que précédemment, `BasicCANConfig` et `PeliCANConfig` héritent toutes les deux de `SjaConfig`. Elles contiennent tous les paramètres de configuration du SJA 1000 (Acceptance Code, Bus Timings,...) avec pour chacun une valeur par défaut.

Les accesseurs, c'est-à-dire les méthodes `get` et `set` qui permettent d'accéder aux données, effectuent le formatage nécessaire et pour les méthodes `set`, les tests de validité qui permettent d'éviter de donner des paramètres de configuration non supportés par le SJA 1000. Les objets de ces classes sont en particulier passés en paramètre à la méthode `init` de la classe `Sja1000`.

5.3.2.5 Accès concurrents à la carte.

Le problème Un des problèmes qui se posent rapidement lors du développement d'une application évoluée utilisant une ressource telle que la carte SJA 1000 est l'accès concurrent à cette carte par plusieurs parties de l'application.

Autrement dit, il est nécessaire de prévoir dès la conception l'utilisation de la librairie CAN dans un environnement multithreadé.

Ce problème est encore plus flagrant en Java qui implémente la gestion des threads dans le langage lui-même, ce qui a pour conséquence que la plupart des applications Java en font un usage important.

La classe `SjaManager`. Dans CAN API, les problèmes liés à la gestion des threads ont été résolus en centralisant toutes les opérations sur la carte SJA 1000 dans un unique thread, qui est le seul autorisé à accéder à la carte. Ce thread est implémenté par la classe `SjaManager`. Les deux principes de ce thread sont les suivant :

- il communique par messages avec les autres threads qui doivent s'être enregistrés au préalable auprès de lui.
- les autres threads communiquent avec lui en positionnant des marqueurs.

5. grâce aux méthodes abstraites et à l'héritage virtuel... voir le code source pour les détails.

La fonction `run()` [6] du thread scrute régulièrement :

- les registres du SJA 1000 et il indique leur modification aux threads enregistrés.
- les marqueurs positionnés par les autres threads et il modifie les registres du SJA 1000 en conséquence.

A l'exception de ce thread "principal", AUCUN autre thread ne doit accéder aux registres du SJA 1000, c'est à dire utiliser les classes `Sja1000` (cette limitation ne s'applique bien sûr qu'aux applications multithreadées, les autres pouvant utiliser directement les classes `Sja1000`).

Communication thread principal (`SjaManager`) vers les autres threads. `SjaManager` gère un certain nombre de propriétés avec des méthodes du type `addUneProprieteListener` et
`removeUneProprieteListener`.

Ces propriétés sont :

Receive : un message est arrivé.
Status : le registre STATUS a été modifié.
Error : les registres d'erreurs ont été modifiés (Mode PeliCAN uniquement).

Par exemple, supposons qu'une classe veuille être informée des messages arrivés et recevoir ces messages arrivés [7]. Il lui faudra d'abord implémenter l'interface `PropertyChangeListener` et avoir une méthode `propertyChange` qui sera appelée à chaque réception d'un message. Enfin, il lui faudra s'enregistrer auprès de l'objet `SjaManager` pour lui indiquer qu'elle veut recevoir les messages arrivés avec la méthode `addReceiveListener` de `SjaManager`.

Exemple de réception d'une trame avec `SjaManager`.

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;
import canapi.can20.Frame;
import canapi.sja1000.*;

/**
 * Affiche les trames recues.
 * Ne pas oublier de mettre la librairie "device.dll"
 * dans le meme repertoire.
 */
public class ReceivedFramePrinter implements PropertyChangeListener {
    private SjaManager sjaManager;

    public ReceivedFramePrinter (SjaManager sm) {
        sjaManager = sm;
        // Enregistrement aupres de SjaManager comme
        // voulant recevoir les trames recues
```

6. En Java, on met dans la méthode run() d'une classe implémetant un thread ce qui devra être exécuté par le thread.
7. Voir la classe `jerrycan.ReceiverPanel` pour un autre exemple complet.

```
        sjaManager.addReceiveListener(this);
        System.out.println("J'ecoute...");
    }

    /** Appele lorsqu'un message est recu. */
    public void propertyChange (PropertyChangeEvent event) {
        // On recupere le message recu
        Frame frame = (Frame)(event.getNewValue());
        // et on l'affiche
        System.out.println("Trame recues : " + frame);
    }

    /**
     * Desenregistre la classe aupres du SjaManager
     * pour arreter de recevoir des messages.
     */
    public void stopListening () {
        sjaManager.removeReceiveListener(this);
    }

    public static void main (String[] args) {
        Sja1000 sja = new BasicCAN();
        SjaConfig config = SjaConfig.getDefaultConfig();
        SjaManager sjaManager = new SjaManager(sja);
        sjaManager.start();
        sjaManager.init(config);
        ReceivedFramePrinter app = new ReceivedFramePrinter(sjaManager);
    }
}
```

Communication des autres threads vers le thread principal (`SjaManager`). `SjaManager` gère des marqueurs positionnés par les autres threads, et lui indiquant les commandes à appliquer au SJA 1000 (des "command request").

Ces marqueurs sont :

Send : un message se trouve dans la file d'attente et doit être envoyé. Une gestion de la confirmation des envois se basant sur l'acquittement des trames est aussi implémenté.

Init : le SJA doit être réinitialisé (Reset du SJA 1000 et nouvelle configuration).

Abort : la transmission actuelle doit être annulée.

AbortAll : la transmission actuelle et toutes les transmissions en attente doivent être annulées.

Par exemple, pour envoyer une trame, il suffit d'appeler send de l'objet `SjaManager` avec en paramètres la trame à envoyer et, éventuellement, si l'on souhaite la confirmation de l'envoi de la trame, un objet d'une classe implémentant l'interface `NeedSendConfirmation` dont la méthode `confirm` sera appelée pour confirmer l'envoi.

Exemple d'envoi d'une trame avec `SjaManager`.

```
import canapi.can20.*;
import canapi.sja1000.*;

/** Envoi d'une trame */
public class SendFrame implements NeedSendConfirmation {

    public SendFrame (SjaManager sjaManager, Frame frame) {
        System.out.println("Envoi de : " + frame);
        sjaManager.send(frame, this);   // on souhaite une confirmation
    }

    /** Confirmation de l'envoi et de l'acquittement de la trame. */
    public void confirm (Frame frame) {
        //System.out.println("Envoi confirme");      // DEBUG
        System.out.println("Envoi de : " + frame.shortString() + " confirme");
        System.exit(0);
    }

    public static void main (String[] args) {
        Sja1000 sja = new BasicCAN();
        SjaConfig config = SjaConfig.getDefaultConfig();
        SjaManager sjaManager = new SjaManager(sja);
        sjaManager.start();
        sjaManager.init(config);

        // Envoi une remote frame d'ID 14
        Frame frame = new RemoteFrame(14);
        SendFrame app = new SendFrame(sjaManager, frame);
    }
}
```

5.3.3 Package `canapi.app`.

Ce package contient des classes permettant, à partir des fonctionnalités de base du SJA 1000 telles que l'envoi et la réception d'une trame, d'effectuer des opérations plus complexes.

En l'occurrence, le package contient dans l'implémentation actuelle une classe `FileSender` permettant d'envoyer des données plus longues que les 8 octets d'une trame de données. Cette classe prend les données soit à partir d'un fichier, soit à partir d'un objet `String` et les envoie sur le bus avec un ID spécifié selon la procédure suivante : toutes les trames ont une longueur (DLC) de 8 octets, sauf la dernière. Ce DLC différent de 8 permet de connaître la fin du fichier[8].

Le package contient aussi un classe `Slio` non totalement implémentée permettant de gérer des SLIO se trouvant sur le bus.

8. il est possible que la dernière trame soit vide.

5.4 Application JerryCAN.

JerryCAN est une application utilisant CANAPI. Son objectif est double : démontrer une partie des possibilités du bus CAN, du contrôleur SJA 1000 et de la librairie CAN API, et être utilisé comme application facilitant le développement d'applications utilisant le bus CAN.

5.4.1 Fichier de configuration.

Un certain nombre d'options de JerryCAN peuvent être réglées dans le fichier de configuration `jerrycan.properties`. Le format de ses entrées est :

`nom_du_paramètre = paramètre`

5.4.2 Assistant de configuration

JerryCAN permet d'effectuer la configuration du SJA 1000 de façon simple et graphique grâce à un assistant de configuration fonctionnant sur la même philosophie que ceux que l'on trouve dans les environnements graphiques tels que Windows. Il permet de choisir le mode de fonctionnement du SJA 1000 (BasicCAN ou PeliCAN), de régler l'Acceptance Filter, les bus timings, le registre OUTPUT CONTROL qui définit les fonctions et le mode de fonctionnement des pattes TX0 et TX1, d'activer ou de désactiver la sortie de l'horloge ou le transceiver interne.

Lors du lancement de JerryCAN, l'assistant de configuration est automatiquement affiché. L'utilisateur peut accepter la configuration par défaut et choisir "Terminer" ou modifier cette configuration.

FIG. 5.2 – *Ecran de l'assistant de configuration.*

L'assistant utilise un modèle d'assistant général du package `oberle.awt.assistant` implémentant les fonctions de base telles que le passage d'un écran à l'autre ou l'affichage. Ce modèle se

spécialise en créant des classes pour chaque écran de configuration ainsi qu'une classe globale, l'*AssistantManager*, qui va contrôler l'assistant, en vérifiant par exemple que l'utilisateur a bien entré les bonnes données avant de lui donner le droit de passer à l'écran suivant.

Des modèles de création d'un assistant peuvent être trouvés dans les fichiers du le répertoire `src/oberle/awt/assistant`:

- `Exemple_gestionnaire_assistant.txt`
- `Exemple_test_gestionnaire_assistant.txt`
- `Exemple_panel_assistant.txt`

5.4.3 Gestion du Manager.

Après la fin de l'assistant de configuration, une instance de la classe `SjaManager`, le manager, est créé. Le manager est démarré et le SJA 1000 initialisé, puis l'utilisateur peut commencer à utiliser JerryCAN. Il est possible d'arrêter ou de redémarrer le manager dans le menu *SJA 1000*. Si le manager est arrêté, la LED en bas à droite de la fenêtre principale est rouge, sinon, lorsque le manager est prêt à être utilisé, la LED est verte.

Il est aussi possible de régler la vitesse du manager grâce à l'option *Vitesse du Manager* dans le menu SJA 1000 qui affiche une fenêtre de réglage. Plus le manager fonctionnera vite, plus les trames seront rapidement envoyées ou moins il y aura de risques qu'un Data Overrun survienne. Néanmoins, il peut être intéressant de ralentir le manager pour libérer des ressources machines. Par défaut, le manager est réglé à sa vitesse maximale dans JerryCAN. La valeur est modifiable avec le paramètre `managersleepingtime` du fichier de configuration. Attention, 0 correspond à la vitesse maximale car la valeur correspond en fait à la pause en milisecondes effectuée par la boucle principale de la méthode `run()` de `SjaManager`.

Il est aussi possible dans le menu SJA 1000 d'effectuer une nouvelle configuration du SJA 1000, de le réinitialiser ou encore d'annuler tous les envois en cours en cas de problème sur le bus par exemple.

5.4.4 Outils de JerryCAN.

Le menu *Outils* propose plusieurs choix affichant les fenêtres suivantes.

Les trois premiers outils fonctionnent sur un principe de polling et proposent un bouton *Démarrer le polling / Arrêter le polling* qui permet de choisir de recevoir ou de ne pas recevoir les informations. En effet, le polling consommant des ressources, il peut être utile de le désactiver.

5.4.4.1 Visualisateur de trames reçues

Cette fenêtre permet d'afficher les trames reçues par le SJA 1000. La fenêtre du haut affiche en permanence la dernière trame reçue, tandis que la fenêtre du bas affiche toutes les trames reçues lors d'un clic sur le bouton *Rafraîchir*. Le bouton *Effacer* permet de vider la liste des trames reçues. Enfin une option *Ecrire dans le fichier de log* permet d'écrire toutes les trames reçues dans un fichier dont le nom est précisé dans le fichier de propriétés de JerryCAN par le paramètre `logfile`.

5.4.4.2 Visualisateur du registre STATUS

Cette fenêtre affiche le détail de chaque bit du registre STATUS.

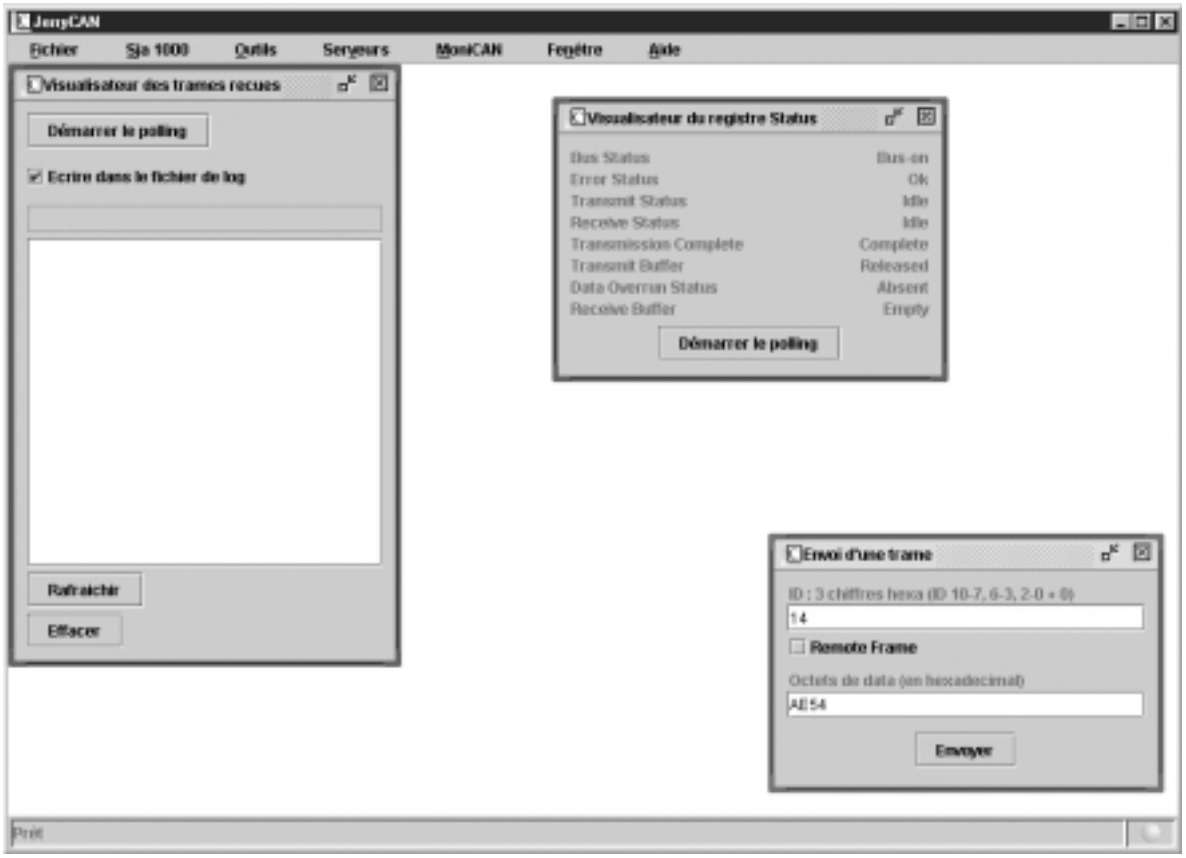


FIG. 5.3 – *Ecran de JerryCAN.*

5.4.4.3 Visualisateur des registres d'erreurs

Cette fenêtre affiche le détail des registres **ArbitrationLostCapture** (indique l'emplacement de la dernière perte d'arbitration), **ErrorCodeCapture** (donne des informations sur la dernière erreur survenue telles que sont type ou son emplacement), **ReceiveErrorCounter** (nombre d'erreurs en réception) et **TransmitErrorCounter** (nombre d'erreurs en transmission).

5.4.4.4 Envoi d'une trame

Cette fenêtre permet d'envoyer une trame en précisant l'ID, si c'est une Remote Frame, et si ce n'est pas le cas, les données de la trame.

Le format de l'ID est un nombre en hexadécimal sur trois chiffres de la forme :

Chiffre de gauche :	ID10 ID9 ID8 ID7
Chiffre du milieu :	ID6 ID5 ID4 ID3
Chiffre de droite :	ID2 ID1 ID0 X

5.4.4.5 Envoi d'un fichier

Cette fenêtre permet d'envoyer un fichier sur le bus avec l'ID précisé dont le format est identique à celui de la fenêtre d'envoi de trame. Le fichier est envoyé par série de trames

de données de 8 octets sur le bus, sauf la dernière trame dont la longueur est inférieure à 8 (éventuellement 0 si le nombre d'octets du fichier est un multiple de 8). Cette dernière trame permet de repérer la fin du fichier.

5.4.4.6 Envoi d'un programme (loader)

Cette fenêtre permet d'envoyer un fichier sur le bus de telle sorte que, si ce fichier est un fichier S29 par exemple, il puisse être chargé par un autre nœud du bus.

Le protocole d'envoi est le suivant : une Remote Frame est envoyée avec l'ID précisé par le paramètre `idloader` de fichier de propriétés. Ensuite le loader attend une réponse, c'est à dire une nouvelle Remote Frame avec le même ID. Lorsqu'il a reçu cette réponse, il envoie le fichier de la même manière que la fenêtre *Envoi d'un fichier*.

En pratique, le nœud distant qui doit charger le programme ne doit envoyer la réponse que lorsqu'il est prêt à recevoir le programme.

5.4.5 Serveur de mails.

Il est possible d'envoyer les mails de l'utilisateur sur le bus grâce à l'option *Démarrer le serveur Mail* du menu *Serveurs*. L'utilisateur doit alors rentrer son login et son mot de passe. Le serveur de mail à consulter est précisé par le paramètre `host` du fichier de configuration, tandis que l'intervalle de vérification de la présence de nouveau mail est précisé par le paramètre `mailsleepingtime`.

Lorsqu'un nouveau mail arrive, une Remote Frame est envoyée sur le bus avec l'ID précisé par le paramètre `idmail`. Si le serveur reçoit une réponse, c'est à dire une Remote Frame avec le même ID, il envoie un fichier contenant le mail sur le bus selon la procédure d'envoi de fichier.

Il est aussi possible d'utiliser le serveur de mail comme une application autonome en exécutant la commande :

```
java jerrycan.mail.MailWatcher user password
```

5.4.6 CAN WebServer.

La classe `jerrycan.web.WebServer` est un serveur Web permettant d'accéder à l'état du bus depuis Internet.

Ce serveur attend les connections des browsers sur le port 80 par défaut. Lorsqu'une page lui est demandée, il regarde l'état du bus et génère une page HTML en fonction des informations récupérées.

CAN WebServer peut fonctionner soit en application autonome, soit intégré dans JerryCAN. Pour le lancer comme application autonome, exécuter la commande :

```
java jerrycan.web.WebServer
```

Dans JerryCAN, le serveur peut être démarré ou arrêté dans le menu *Serveurs*.

5.4.7 Test intéressant.

Dans une utilisation classique, d'autant plus en laboratoire, on voit rarement les compteurs d'erreurs bouger. On peut faire une expérience intéressante en envoyant un fichier sur le bus et pendant l'envoi, couper le fil du bus. Si on a activé le polling des registres d'erreurs, on voit celui des erreurs en transmission augmenter très rapidement jusqu'à 256. D'autre part,

le bit Error Status du registre STATUS est activé. Si on rebranche alors le bus, le compteur rediminue, mais plus lentement qu'il n'avait augmenté. Lorsqu'il atteint 96, le bit Error Status repasse à OK.

Ce n'est qu'après avoir résolu un problème particulièrement ardu avec l'outil complexe qu'on se rend compte que i était en fait une intensité.
Loi Secrète d'Ampère.

FIG. 5.4 – *Exemple de pages générées par CAN WebServer.*

Liste des annexes.

En annexe à ce rapport, on trouvera :

- Les sources des programmes en C permettant d'utiliser la carte PC.
- Les sources PPC permettant d'utiliser la carte PPC-SJA 1000 ainsi que les sources de MoniCAN.
- La documentation des classes Java composants l'API CAN.

D'autre part, ce rapport est accompagné d'un CD-ROM contenant

- L'intégralité des sources du projet.
- Les documentations de toutes les classes Java.
- De nombreuses documentations concernant le CAN.
- Et plein d'autres choses encores...

Les sources Java étant trop importantes en taille, elles sont uniquement sur le CD-ROM. De plus, seules les documentations des classes Java utiles pour un autre développement, c'est-à-dire celles de l'API CAN sont fournies en annexes. Celles des autres classes sont fournies sur le CD-ROM.

Table des figures

Bibliographie

[1] Motorola. Can protocol. Technical report, Motorola, 1997.
[2] Dominique Paret. *Le bus CAN*. DUNOD, 1996.
[3] Philips. P82c150 data sheet. Technical report, Philips, 1996.
[4] Philips. Sja1000 application note an97076. Technical report, Philips, 1997.
[5] Guy Pujolle. *Les réseaux*. Eyrolles, 1997.