REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
ECOLE NATIONALE POLYTECHNIQUE.
DEPARTEMENT D'ELECTRONIQUE

Mémoire de fin d'études

En vue de l'obtention du

Diplôme d'ingénieur d'état en Electronique

Thème
Controleur CAN sur une carte DE2 d'ALTERA

Réalisé par :
SEFSAFI Assia

Proposé et dirigé par :
Mr.SADOUN Rabah

Promotion 2010/2011

ملخص

تستخدم شبكة CAN على نطاق واسع في مجال السيارات وغيره من المجالات الصناعية الأخرى. يوجد العديد من الحلول التي تسمح بالتواصل عبر هذه الشبكة مثل استخدام معالج بروتوكول او ميكروكنترولر يتضمن معالج بروتوكول. مع ظهور الدارة القابلة للبرمجة FPGA يمكننا الحصول على معالج بروتوكول و ميكروكنترولر على نفس الرقاقة عن طريق استعمال تقنية نظام على رقاقة SOPC . هذا يمثل الهدف من مشروعنا الذي سينفذ على شريحة ALTERA DE2

الكلمات المفتاحية :.CAN, VHDL, FPGA, SOPC

Résumé

Le réseau CAN (Controller Area Network) est largement utilisé dans le domaine de l'automobile en particulier et dans le domaine industriel en général. Afin de pouvoir communiquer sur un bus CAN, plusieurs solutions s'offrent aux utilisateurs : des gestionnaires de protocole de type *stand alone* ou des microcontrôleurs à contrôleur CAN intégré. Avec l'apparition des FPGA, on peut aujourd'hui envisager un contrôleur CAN piloté par un microcontrôleur sur une seule puce en utilisant l'approche SOPC. Ceci va faire l'objet de notre projet où on va mettre en œuvre un contrôleur CAN décrit en langage VHDL sur une carte DE2 d'ALTERA.

Mots clés : CAN, VHDL, FPGA, SOPC.

Abstract

The Controller Area Network (CAN) is widely used in designing car applications and also for industrial applications. To be able communicating in a bus CAN, different solutions are given for users: a standalone protocol handler or a microprocessor with a built-in CAN controller. With the apparition of FPGA, we can consider a CAN controller driven by a microcontroller in one chip using the development of SOPC. This will be the object of our project in which we are going to implement a CAN controller modeled using VHDL language in a DE2 board card of ALTERA.

Key words : CAN, VHDL, FPGA, SOPC.

Remerciements

Louange à Allah tout puissant qui m'a guidée pour l'accomplissement de ce modeste travail.

Un grand merci à M.SADOUN Rabah qui m'a guidée au cours de ces mois durant lesquels son aide précieuse, ses conseils et orientation m'ont été d'un précieux apport afin de mener à beau et à bien ce travail que j'espère être digne de la confiance qu'il a placée en moi.

J'aimerai exprimer ma gratitude aux personnes qui m'ont fait l'honneur de participer au jury de ce mémoire: Mr MEHENNI Mohamed, Professeur à l'Ecole Nationale Polytechnique d'Alger, d'avoir accepté de présider le jury et Mr L.Abdeloual, d'avoir bien voulu examiner mon travail.

Je remercie aussi tous les enseignants de l'Ecole Nationale Polytechnique, spécialement ceux des départements des Sciences Fondamentales et d'Electronique, pour leur apport en savoir.

Mes remerciements, vont au personnel de l'Ecole et à toute personne dévouée au service de l'Ecole Nationale Polytechnique.

Enfin j'adresse mes sincères remerciements à toute personne qui m'a soutenue et encouragée au cours de la réalisation de mon projet.

Dédicaces

En signe d'amour et de gratitude et de respect je dédie ce modeste travail :

A mes très chers qui m'ont couvert d'amour, de soutient qu'ils trouvent dans ce mémoire le fruit de leur travail :

Mon père que je ne remercierai jamais assez pour tout ce qu'il a fait pour moi, que dieu le garde.

Ma maman qui m'abreuve d'amour et d'affection intarissable, source de mon bonheur et ma raison d'être,

A mes chers grands-parents que dieu les protège, ainsi que mes oncles et tantes, cousins et cousines.

A mon unique frère Mustapha,

A ma sœur et amie Soumia et notre p'tite et adorable sœur Noufida,

A tous mes amis qui m'ont toujours soutenu et encouragé et avec qui j'ai passé de bon moments. En particulier Kaouther, Nadjeh, Amel, Hadjer, Nadjib, Amine, Atmane…

Et bien sur sans oublier Jacob et Mahmoud qui m'ont supportée et qui ont su avec leur enthousiasme et leur bonne humeur m'aider à bien mener ce travail.

ASSIA

Sommaire

INTRODUCTION GENERALE……………………………………………………1

CHAPITRE 1 : Etude théorique du bus CAN
1.1- Introduction……………………………………………………………………...2
1.2- Rétrospective…………………………………………………………………….2
1.3- Généralités sur le Bus CAN……………………………………………………..2
1.3.1- Définition……………………………………………………………………2
1.3.2- Spécifications………………………………………………………………...2
1.4- Le bus CAN et le modèle OSI……………………………………………………..4
1.5- Couche physique……………………………………………………………………5
1.5.1- Les supports de transmission………………………………………………...5
1.5.2- Les adaptateurs ligne…………………………………………………………...5
1.5.3- Bit CAN………………………………………………………………………...6
1.5.3.1- Valeurs du bus……………………………………………………...6
1.5.3.2- Codage du bit………………………………………………………7
1.5.3.3- Construction du bit time……………………………………………8
1.5.3.4- Le *nominal bit time* …………………………………………………..9
1.5.3.5- Description des segments du *nominal bit time*……………………………..9
1.5.4- Synchronisation du bit………………………………………………………..10
1.5.4.1- Problèmes de synchronisation………………………………………10
1.5.4.2- Règles de synchronisation…………………………………………...11
1.6- Couche liaison de données……………………………………………………………12
1.6.1- Les types de trames……………………………………………………………..12
1.6.1.1- Trame de données……………………………………………………13
1.6.1.2- Trame de requête…………………………………………………….16
1.6.1.3- Période inter trame…………………………………………………..17
1.6.1.4- Conditions de surcharge………………………………………………18
1.6.2- Traitement des erreurs…………………………………………………………..19
1.6.2.1- Les différents types d'erreur……………………………………………19
1.6.2.2- Signalisation des erreurs…………………………………………………20
1.6.2.3- Recouvrement des erreurs……………………………………………..21
1.7- Les composants CAN……………………………………………………………………..23
1.7.1- Architecture générale des composants CAN……………………………………23
1.7.2- Les familles des composants……………………………………………………24
1.8- Conclusion………………………………………………………………………………25

CHAPITRE 2 : *Etude du contrôleur SJA1000*
2.1- Introduction……………………………………………………………………………26
2.2- présentation du SJA1000……………………………………………………………….26
2.3- Caractéristiques…………………………………………………………………………26
2.4- Architecture du SJA1000………………………………………………………………26
2.4.1- Description des différents blocs du SJA1000…………………………………27
2.4.1.1- Bit Timing Logic…………………………………………………27
2.4.1.2- Bit Stream Processor……………………………………………..27
2.4.1.3- Buffer de réception……………………………………………………28
2.4.1.4- Buffer d'émission…………………………………………………..28

2.4.1.5- Le filtre d'acceptation………………………………………………………29
2.5- Modèle de programmation du SJA1000……………………………………………………..29
2.5.1- Les différents registres du SJA1000………………………………………………..29
2.5.1.1- Le registre MODE………………………………………………………..29
2.5.1.2- Le registre COMMAND……………………………………………………30
2.5.1.3- Le registre STATUS…………………………………………………………30
2.5.1.4- Le registre INTERRUPT……………………………………………………31
2.5.1.5- Le registre de contrôle………………………………………………………31
2.5.1.6- Le registre clock divider…………………………………………………….32
2.5.1.7- Le registre BUS TIMING0………………………………………………….32
2.5.1.8- Le registre BUS TIMING1………………………………………………….32
2.5.1.9- Les différents registres du filtre d'acceptation……………………………32
2.5.1.10- Configuration du filtre d'acceptation……………………………………33
2.5.1.11- Plan d'adressage du SJA1000……………………………………………...36
2.5.2- Procédure de communications CAN……………………………………………….36
2.5.2.1- Mode *polling*…………………………………………………………………..36
2.5.2.2- Mode interruptible………………………………………………………….37
2.5.2.3- Initialisation du SJA1000…………………………………………………..38
2.5.2 .4- Transmission d'une trame en mode *polling*……………………………...39
2.5.2.5- Réception d'une trame en mode *polling*……………………………………40
2.6- Conclusion……………………………………………………………………………………40

CHAPITRE 3 : *Implémentation d'un contrôleur CAN sur la carte ALTERA DE2*

3.1- Introduction…………………………………………………………………………………41
3.2- Réalisation d'un système sur puce……………………………………………………………41
3.2.1- Principe……………………………………………………………………………..41
3.2.2- Définition d'un IP core……………………………………………………………..41
3.2.3- Flot de conception d'un SOPC……………………………………………………..41
3.3- Carte DE2 et environnement de développement de SOPC ALTERA………………………42
3.3.1- Présentation de la carte ……………………………………………………………42
3.3.2- Architecture Système………………………………………………………………43
3.3.2.1- L'IP NIOS II………………………………………………………………43
3.3.2.2- Les caractéristiques du NIOS II …………………………………………43
3.3.3- Le bus Avalon………………………………………………………………………44
3.3.3.1- Présentation…………………………………………………………………44
3.3.3.2- Caractéristiques…………………………………………………………….45
3.3.4- Environnement de développement…………………………………………………46
3.3.6.1- Configuration matérielle……………………………………………………47
3.3.6.2- Configuration logicielle…………………………………………………….47
3.3.5 - L'IP core CAN………………………………………………………………………48
3.3.5.1- Fonctionnement de l'IP core du contrôleur CAN………………………48
3.4.1.1- Description des signaux…………………………………………………50
3.4- Mise en œuvre………………………………………………………………………………51
3.4.1- Simulation de l'IP core…………………………………………………………………51
3.4.1.1- simulation de l'étape d'écriture dans un registre du contrôleur…………51
3.4.1.2- Simulation de l'étape de lecture à partir d'un registre du contrôleur……52
3.4.1.3- Simulation de l'émission d'une trame à partir du contrôleur……………53
3.4.2- Implémentation de l'IP core………………………………………………………55
3.4.2.1- Utilisation du bus avalon…………………………………………………55
3.4.2.2- Utilisation des PIO…………………………………………………………57

3.5- Conclusion……………………………………………………………………………61

CONCLUSION GENERALE…………………………………………………...62
BIBLIOGRAPHIE…………………………………………………………63
ANNEXES…………………………………………………………………...65

Liste des figures

1.1 Réseau CAN……3
1.2 Le bus CAN et le modèle OSI……5
1.3 Paire filaire simple……6
1.4 Paire filaire torsadée……6
1.5 Exemple de Drivers de lignes le 82C250……6
1.6 Niveaux de tension du bus CAN low speed……7
1.7 Niveaux de tension du CAN high speed……7
1.8 Exemple du codage NRZ du bus CAN……8
1.9 Construction du Time Quantum……8
1.10 Durée des segments du nominal bit time……10
1.11 Problème de synchronisation……10
1.12 Erreur de phase……12
1.13 Trame de données……13
1.14 Champ d'arbitrage……14
1.15 Champ de commande……14
1.16 Champ CRC……15
1.17 Champ d'acquittement……16
1.18 Trame de requête……17
1.19 Période d'intertrame……18
1.20 Trame de surcharge……18
1.21 Les sources d'erreur dans une trame CAN……20
1.22 Construction de la trame d'erreur……20
1.23 Trame d'erreur active……21
1.24 Trame d'erreur passive……21
1.25 Compteur d'erreur et état d'un nœud……22
1.26 Constitution d'un nœud CAN……24

2.1 Schéma de fonctionnement du SJA1000……27
2.2 Fonctionnement du BTL et du BSP……27
2.3 Buffer de réception……28
2.4 Buffer de transmission……28
2.5 Filtre d'acceptation en mode single pour une trame standard……33
2.6 Filtre d'acceptation en mode single pour une trame étendue……34
2.7 Filtre d'acceptation en mode dual pour une trame standard……35
2.8 Filtre d'acceptation en mode dual pour une trame étendue……35
2.9 Flot de communication en mode *polling*……36
2.10 Flot de communication en mode interruptible……37
2.11 Initialisation du contrôleur……38
2.12 Transmission d'une trame……39
2.13 Réception d'une trame……40

3.1 Flot de conception d'un SOPC……42
3.2 Schéma fonctionnel de la carte ALTERA DE2……43
3.3 Architecture d'un processeur NIOS II……44
3.4 Architecture du bus Avalon……45
3.5 Flot de conception d'un SOPC sur ALTERA……46

3.6 Schéma fonctionnel de l'IP core……………………………………………………48
3.7 Schéma bloc de l'IP core……………………………………………………………...50
3.8 Simulation de l'étape d'écriture dans un registre du contrôleur…………………………52
3.9 Simulation de l'étape lecture à partir d'un registre du contrôleur……………………..53
3.10 Simulation de la partie initialisation du contrôleur………………………………………53
3.11 Remplissage du buffer de transmission………………………………………………...54
3.12 Trame envoyée sur la ligne de transmission……………………………………………55
3.13 Cycle de lecture à partir d'un registre du contrôleur……………………………………...56
3.14 Cycle d'écriture dans un registre du contrôleur…………………………………………...56
3.15 Connexion de l'IP core au NIOS II à travers le bus avalon………………………………57
3.16 Le composant CAN………………………………………………………………………57
3.17 Le composant généré par le SOPC Builder………………………………………………58
3.18 Organigramme de la fonction write_reg…………………………………………………..59
3.19 Organigramme de la fonction read_reg……………………………………………………60
3.20 Remplissage du buffer de transmission……………………………………………………61

Liste des tableaux

1.1 Distance maximale entre deux nœuds en fonction du débit……………………………………3
1.2 Caractéristiques du CAN low et high speed……………………………………………………4

2.1 Fonctions des différents bits du registre MODE……………………………………………29
2.2 Fonctions des différents bits du registre COMMAND……………………………………..30
2.3 Fonctions des différents bits du registre STATUS…………………………………………30
2.4 Fonctions des différents bits du registre INTERRUPT ……………………………………31
2.5 Fonctions des différents bits du registre CONTROL………………………………………..31
2.6 Fréquence de l'horloge CLOCKOUT…………………………………………………………32
2.7 Signification des différents bits du registre BUS_TIMING0…………………………………32
2.8 Signification des différents bits du registre BUS_TIMING1…………………………………32

3.1 Signaux entre le bus Avalon et un périphérique esclave……………………………………45
3.2 Signaux entre le bus avalon et l'IP core………………………………………………………56

Acronymes

CAN	Controller Area Network
ECU	Electronic Central Units
OSI	Open System Interconnection
CSMA/BA	Carrier Sense Multiple Access-Bitwise Arbitration
PLS	Physical Signaling
PMA	Physical Medium Access
MDI	Medium Dependent Interface
MAC	Medium Access Control
LLC	Logical Link Control
NRZ	Non Return to Zero
SOF	Start Of Frame
CRC	Cyclic Redundancy Code
SJW	Synchronization Jump Width
RTR	Remote Transmission Request
DLC	Data Length Code
REC	ReceiveError Counter
TEC	Transmit Error Counter
SLIO	Serial Linked Input Output
GPIO	General Parallel Input Output
FPGA	Filed Programmable Gate Array
IP	Intellectual Property
BTL	Bit Timing Logic
BSP	Bit Stream Processor
BRP	Baud Rate Prescalar
BDF	Bloc Diagram File
PTF	Peripheral Template File

Introduction Générale

Introduction générale

L'origine du bus CAN est intimement liée au domaine de l'électronique automobile. Ce dernier est caractérisé par un environnement hostile (bruit très important) et un câblage dense, de plus en plus complexe dû à une intégration et à une automatisation de plusieurs fonctions. Ces fonctions (ordinateur de bord, système électronique de stabilité ou ESP, d'antiblocage des roues ou ABS, d'anti patinage ou ASR et autres) font toutes appel à l'électronique embarquée représentant un système complexe construit autour d'unités de commande et de contrôle (souvent des microcontrôleurs) nécessitant une communication fiable.

Pour satisfaire ces exigences de plus en plus importantes et dès 1985, un bus système série fut présenté. Il était le fruit d'une collaboration entre l'université de Karlsruhe et la société Bosch. Il est connu sous le nom de bus CAN (Controller Area Network). Il répondait à trois contraintes majeures à savoir la fiabilité de fonctionnement, le temps réel et le coût.
Ce bus est bâti autour de l'interfaçage des signaux de transmission, d'algorithmes de réjection de bruits et d'un protocole de transmission. Cet ensemble de fonctions est concrétisé sous forme d'un circuit dénommé contrôleur CAN couplé à des adaptateurs de ligne. Il est physiquement petit, peu coûteux et entièrement intégré. Il est utilisable à des débits conséquents dans des environnements difficiles et offre des transmissions à haut niveau de fiabilité.
Ces avantages ont fait que le bus CAN voit son utilisation s'étendre très vite à d'autres secteurs autres que l'automobile. On peut citer l'avionique, l'industrie textile, l'agriculture, la marine, le biomédical, les ascenseurs [2] et autres.

La réduction du câblage n'exclut pas aussi l'idée de la réduction de la taille des systèmes électroniques ainsi que le temps de leur mise sur le marché. Une solution novatrice a vu le jour sous l'appellation SOC (*System on Chip)* ou SOPC *(System On Programmable Chip).* Elle permet de concevoir des systèmes entiers sur une même puce. Une implémentation SOPC est moins coûteuse, consomme moins, est moins sensible au bruit et plus fiable qu'un système identique réalisé à l'aide de circuits discrets. Les éléments sur lesquels se base le développement des SOPC sont les circuits programmables adossés à l'utilisation d'IP core (ou noyaux IP); modules formant une librairie de circuits destiné chacun à assurer une fonction précise.

Le développement que nous devons entreprendre s'inscrit dans ces deux thématiques à savoir l'implémentation d'un IP Core CAN interfacé au soft processeur NIOS II.

Pour présenter notre développement, nous avons structuré notre mémoire en trois parties. La première donnera une présentation générale du bus de communication CAN (Controller Area Network) et un aperçu sur le protocole qui gère l'échange d'information sur ce bus.
Sachant que notre objectif est la mise en œuvre d'un contrôleur IP Core CAN sur une carte DE2 et tenant compte du fait que le SJA1000 en est le modèle, nous détaillerons dans la deuxième partie l'architecture, le modèle de programmation et le fonctionnement de ce contrôleur.
La dernière partie sera consacrée à la description de l'implémentation réalisée et les résultats de mise en œuvre obtenus.

Chapitre I

Etude théorique du bus CAN

I.1. Introduction

Pour bien appréhender notre développement, il nous ait paru nécessaire d'aborder le bus CAN sous ses divers aspects. Outre son origine et les étapes qui ont conduit à son développement, nous aborderons sa modélisation, ses spécifications et les circuits conduisant à son implémentation.

I.2. Rétrospective

Depuis le début des années quatre vingt, de nombreux systèmes électroniques ont fait leur apparition dans le domaine de l'automobile. Dès début 1981, quelques grandes sociétés automobiles s'intéressèrent à des systèmes de communication en temps réel entre différents microcontrôleurs, concernant notamment le contrôle moteur, des organes vitaux tels que les freins, les correcteurs de trajectoire et autres. Il fallait donc un bus capable d'assurer des communications multi-maîtres rapides, sur une distance correcte et peu sensibles au bruit. Plusieurs bus ont été développés et avaient de grandes qualités mais ne permettaient de satisfaire qu'une partie des fonctions envisagées en raison de l'insuffisance des débits ou (et) de la fiabilité des informations transportées. En 1983, le leader allemand d'équipement automobile Robert Bosch prit la décision de développer un bus de communication satisfaisant à ses propres exigences. Ce n'est qu'au printemps 1986 que ce bus portant le nom de *Controler Area Network* CAN vit le jour. Il était le résultat d'une collaboration entre la société BOSCH, l'université de Karlsruhe et INTEL. En 1987, la réalité prit la forme des premiers siliciums fonctionnels. En 1991, une première voiture (allemande), haut de gamme, équipée de cinq Electronic Central Units (ECU) et d'un bus CAN fonctionnant à 500kb/s sortit des chaines de production. [4]

I.3. Généralités sur le Bus CAN

I.3.1. Définition

Le bus CAN est un véritable réseau respectant le modèle d'interconnexion des systèmes ouverts OSI de l'ISO. Les normes ISO le décrivent comme un protocole de communication série qui supporte de façon efficiente la distribution de commandes en temps réel avec un haut niveau de sécurité. Ses domaines de prédilection couvrent généralement les applications réseaux à haut débit avec une haute fiabilité de transmission et à concept de câblage multiplexé bas coût. Il doit fonctionner dans un environnement limité et sévère comme une usine, un atelier, une voiture…etc. [4]

I.3.2. Spécifications générales

Le réseau CAN fonctionne en topologie bus série auquel il faut ajouter des terminaisons de ligne dont la valeur dépend de la longueur du média [4].

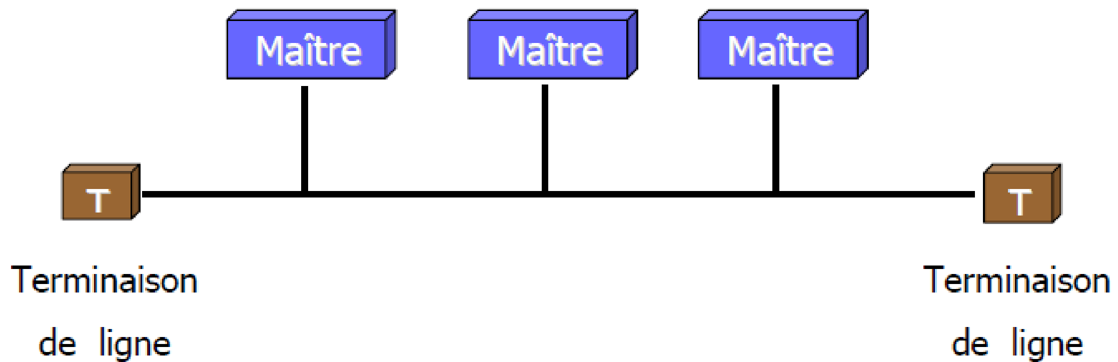


Figure 1.1 : Réseau CAN

L'architecture du réseau CAN est de type multi-maître uniquement. Il n'utilise pas la notion d'esclave. Les nœuds peuvent parler spontanément avec un protocole d'accès CSMA/BA (Carrier Sense Multiple Access – Bitwise Arbitration). C'est un arbitrage par identificateur de message. La transmission est donnée au message qui possède l'identificateur de priorité la plus élevée. Cette technique permet de donner la priorité aux nœuds qui doivent réagir rapidement.

La longueur maximum du bus est déterminée par la charge capacitive et le débit [4]. Les configurations recommandées sont les suivantes sachant que la norme spécifie un débit maximum de 1Mbit/s :

Débit	Distance (m)
1 Mbit/s	40
500 kbit/s	130
250 kbit/s	270
125 kbit/s	530
100 kbit/s	620
50 kbit/s	1300
10 kbit/s	6700

Tableau 1.1 : Distance maximale entre deux nœuds en fonction du débit.

Les normes ISO ont spécifié deux documents pour définir les spécifications du bus CAN. On parle des normes ISO 11898 pour la communication en série de données en basse vitesse et l'ISO 11519 pour la communication en série de données à vitesse élevée.

Ces deux normes diffèrent dans le débit supporté, le nombre de nœuds qui peuvent être connectés au bus, et les spécifications électriques du bus.

Le tableau ci-dessous résume les principales différences entre les deux types de bus notamment sur les débits supportés.

Paramètres	CAN low speed	CAN high speed
Débit	125 kb/s	125 kb/s à 1 Mb/s
Nombre de nœuds sur le bus	2 à 20	2 à 30
Niveau dominant	CAN H=4V CAN L=1V	CAN H=3,5V CAN L=1,5V
Niveau récessif	CAN H=1,75V CAN L=3,25V	CAN H=2,5V CAN L=2,5V
Caractéristique du câble	30 pF entre les câbles de ligne	2*120àΩ
Tension d'alimentation	5V	5V

Tableau 1.2: Caractéristiques du CAN low et high speed [4]

I.4. Le bus CAN et le modèle OSI

Selon la norme Bosch, le protocole CAN couvre seulement deux des sept couches du modèle d'interconnexion des systèmes ouverts OSI (Open System Interconnection) de l'ISO. On retrouve ainsi la couche liaison de données (couche 2 du modèle) et la couche physique (couche 1 du modèle). Une couche application est bien entendu nécessaire pour le fonctionnement du bus, cependant cette dernière n'a pas été spécifiée par la norme Bosch, elle est plutôt laissée au bon usage de l'utilisateur.

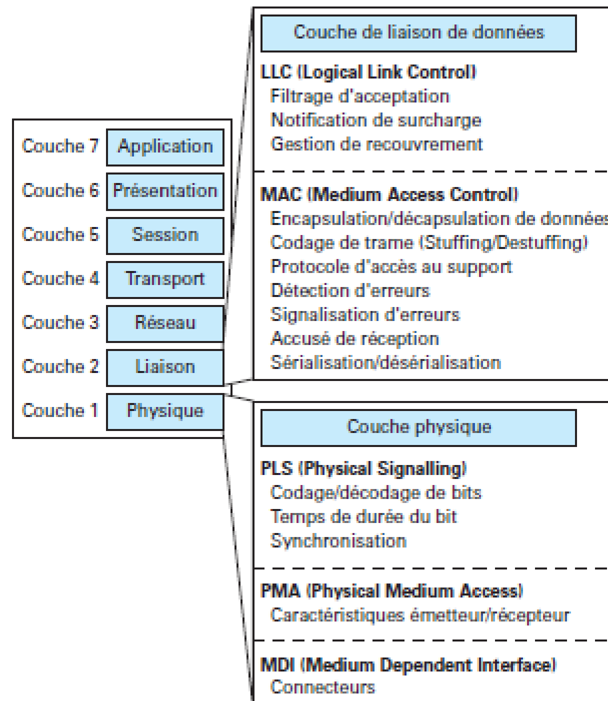


Figure 1.2 : le bus CAN et le modèle OSI [9]

I.5. Couche physique

La couche physique est divisée en trois sous-couches PLS (Physical Signalling), PMA (Physical Medium Access), et MDI (Medium Dependent Interface). Elle définit comment le signal est transmis et a par conséquent pour rôle d'assurer le transfert physique des bits entre les différents nœuds. Cette couche s'occupe donc :

- de gérer la représentation du bit (codage, timing…).
- de gérer la synchronisation bit.
- de définir les niveaux électriques ou optiques des signaux.
- de définir le support de transmission.

I.5.1. Les supports de transmission

Le support de transmission n'a pas été prédéfini par le document de référence du protocole laissant la liberté du choix aux utilisateurs. On peut retrouver différents médias tels que :

- Supports optiques sur fibres optiques,
- Supports électromagnétiques : ondes radiofréquences et ondes infrarouges,
- Supports filaires sous toutes leurs formes : monofilaires et bifilaires. [4]

Ce sont les paires filaires simples et torsadées qui sont les plus employées pour assurer le transport des trames CAN, vue leur grande capacité de se libérer des parasites provenant du monde extérieur. Ceci est dû au fait que lors de l'utilisation de ce type de média, la lecture du signal se fait en mode différentiel, et comme le signal parasite se présente avec les mêmes phase et amplitude la différence sera nulle et le signal utile ne risque pas d'être altéré. Ces paires sont constituées de deux fils : CAN L (CAN Low), CAN H (CAN High).

Figure 1.3 : Paire filaire simple [web1]

Figure 1.4 : Paire filaire torsadée [web2]

I.5.2. Les adaptateurs lignes

L'intérêt du CAN est de pouvoir transporter de l'information donc d'utiliser un support de transmission pour véhiculer les messages. Il existe donc des circuits drivers de lignes capables de piloter le support de transmission. Ils servent d'interface entre le contrôleur CAN et le support de transmission. [9]

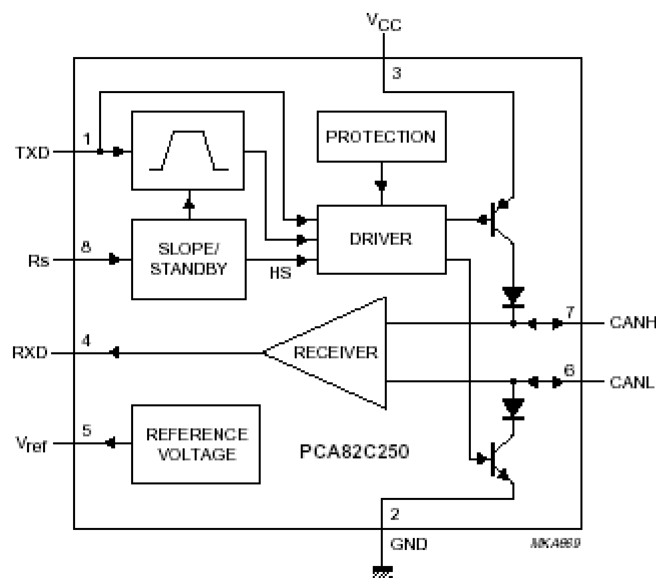


Figure 1.5 : Exemple de Drivers de lignes le 82C250 [9]

I.5.3. Bit CAN

I.5.3.1. Valeurs du bus

Les valeurs qui circulent sur le bus sont des valeurs dites dominantes ou récessives. Dans le cas d'une transmission simultanée de bits récessifs et dominants, la valeur résultante du bus sera dominante (équivalence avec un ET câblé). On peut dire que l'état dominant est l'état logique 0 et le récessif est l'état logique 1.

Ci-dessous deux figures montrant les valeurs du bus dans les deux cas CAN *low speed* et CAN *high speed.*

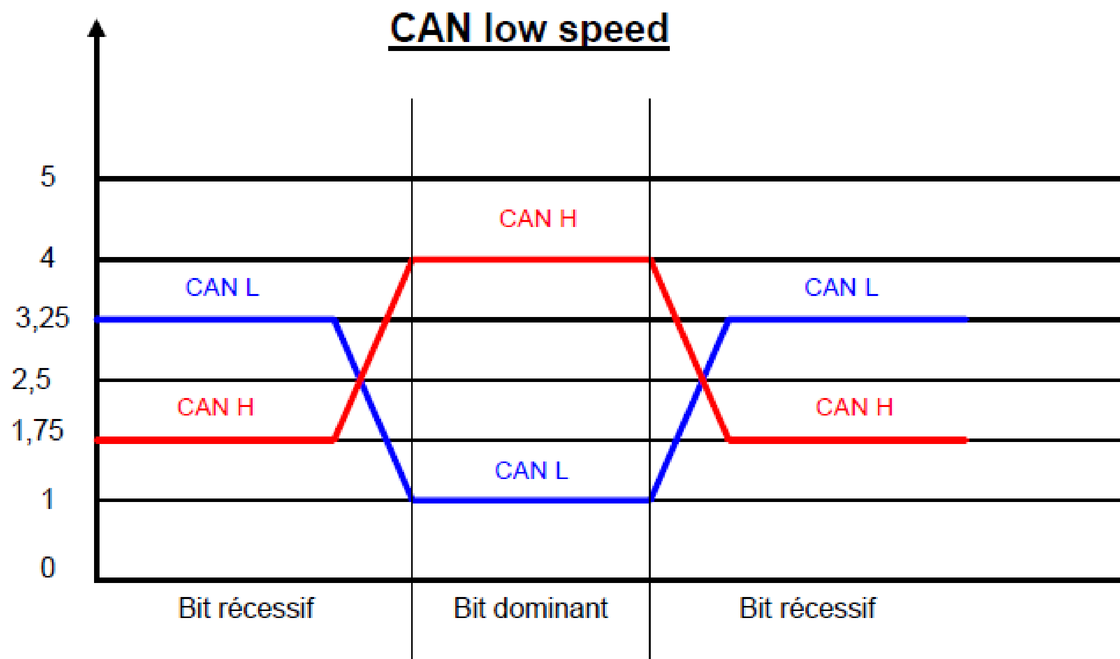


Figure 1.6: Niveaux de tension du bus CAN low speed [8]

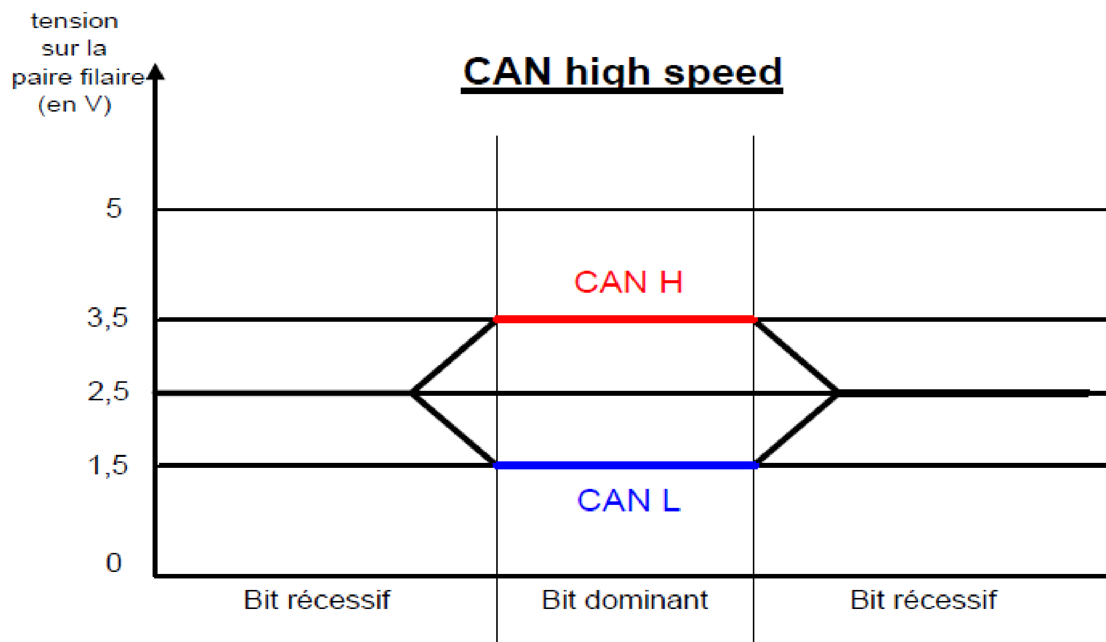


Figure 1.7 : Niveaux de tension du bus CAN high speed [8]

I.5.3.2. Codage du bit

Dans le protocole CAN, la méthode de codage de ligne choisie pour la transmission des données sur le bus est la méthode NRZ (Non Return to Zero).

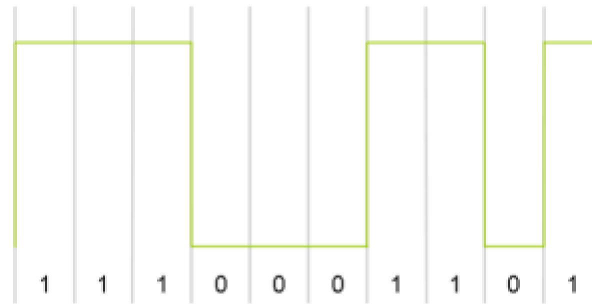


Figure 1.8: Exemple du codage NRZ du bus CAN [web2].

L'utilisation de cette méthode de codage peut poser des problèmes de fiabilité dans le cas d'une importante chaîne de bits identiques. Afin de sécuriser la transmission des messages, on utilise alors la méthode dite de Bit-Stuffing. Le principe de cette méthode est d'insérer un bit de polarité contraire dès que l'on a émis 5 bits de même polarité sur le bus. C'est ce que l'on appelle des bits de remplissage. Par principe, ceci allonge le temps de transmission d'un message mais participe activement à sécuriser son contenu lors de son transport. On obtient ainsi dans le message un plus grand nombre de transitions ce qui permet de faciliter la synchronisation en réception par les nœuds. Un récepteur CAN procédera donc à la fonction inverse « le déstuffage » à la réception en retirant ces bits de remplissage n'ayant servi qu'au transport.

I.5.3.3. Construction du bit time

Dans le protocole CAN, une période d'horloge correspond à ce que l'on appelle le Nominal Bit Time.

Le Time Quantum

Le Time Quantum est une unité de temps qui est construite à partir de la période de l'horloge interne du nœud. La valeur du pré-scalaire m dont la valeur peut varier de 1 à 32 détermine le rapport entre le Time Quantum et le Minimum Time Quantum :

TIME_QUANTUM=m*période de l'horloge de l'oscillateur [4]

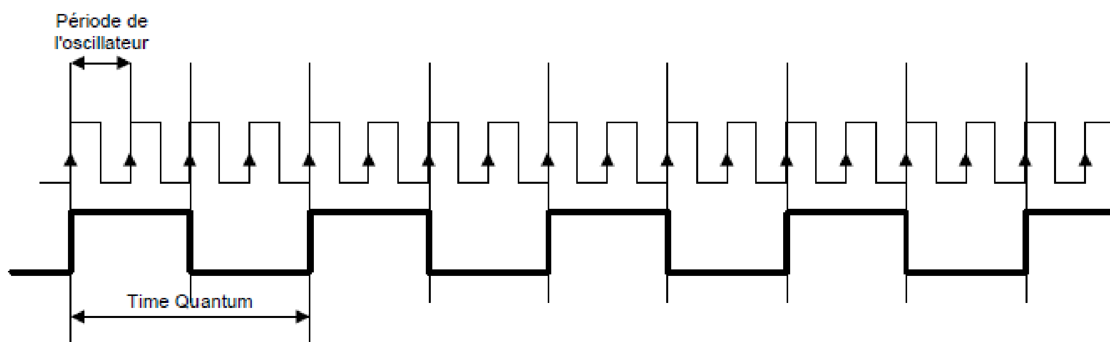


Figure 1.9 : Construction du Time Quantum [8]

Dans l'exemple ci-dessus, le facteur m est égal à 4.

I.5.3.4. Le nominal bit time

Le Nominal Bit Time représente en fait la durée du bit sur le bus. Cette durée est étroitement liée à la période de l'horloge. Ainsi, la durée du bit time de chaque circuit est construite à partir d'un nombre déterminé de périodes d'horloge issue de l'horloge interne de chaque circuit CAN.

La norme BOSCH décrit avec précision la composition de ce Nominal Bit Time qui est divisé en plusieurs segments :

• le segment de synchronisation,

• le segment de propagation,

• le segment de buffer phase1,

• le segment de buffer phase2.

I.5.3.5. Description des segments d'un nominal bit time

Le segment de synchronisation

Le segment de synchronisation est utilisé pour synchroniser les différents nœuds du bus. Une transition (de 0 à 1 ou de 1 à 0) doit s'effectuer dans ce segment pour permettre une resynchronisation des horloges des différents nœuds récepteurs. Ce segment dure 1 *time Quantum*.

Le segment de propagation

Le segment de propagation est utilisé pour compenser les phénomènes de temps de propagation sur le bus. Il peut durer de 1 à 8 *time Quantum*.

Les segments buffer phase1 et buffer phase2

Les segments ″buffer phase1″ et ″buffer phase2″ sont utilisés pour compenser les erreurs de phase détectées lors des transitions. Ces segments peuvent aussi être allongés ou raccourcis lors des phénomènes de resynchronisation. Ils durent chacun 1 à 8 *time Quantum*.

Le point d'échantillonnage

Le point d'échantillonnage ou *sample point* est l'instant précis où l'on échantillonne le bit incident présent sur le bus et par conséquent où l'on peut le lire, l'interpréter et définir sa valeur. Il est situé à la fin du segment de ″buffer phase1″.

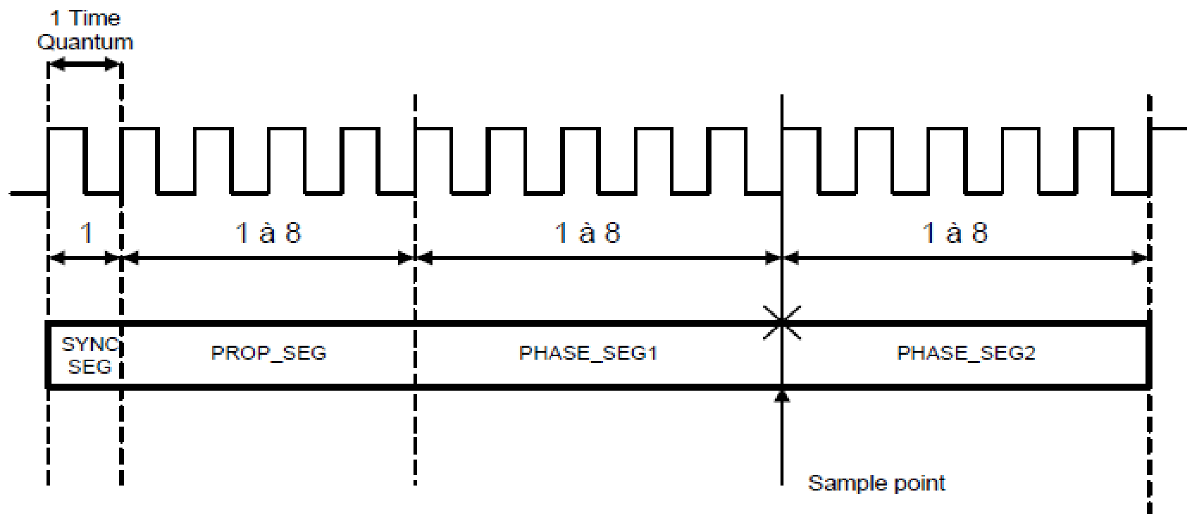


Figure 1.10: Durée des segments du nominal bit time [8]

I.5.4. Synchronisation du bit

I.5.4.1. Problèmes de synchronisation

On dit qu'il y a problème de synchronisation lorsque la valeur lue sur le bus au moment de l'échantillonnage n'est pas la valeur émise. Ceci peut être dû au fait que les Nominal Bit Time des différents nœuds présents sur le bus ne soient pas synchronisés.

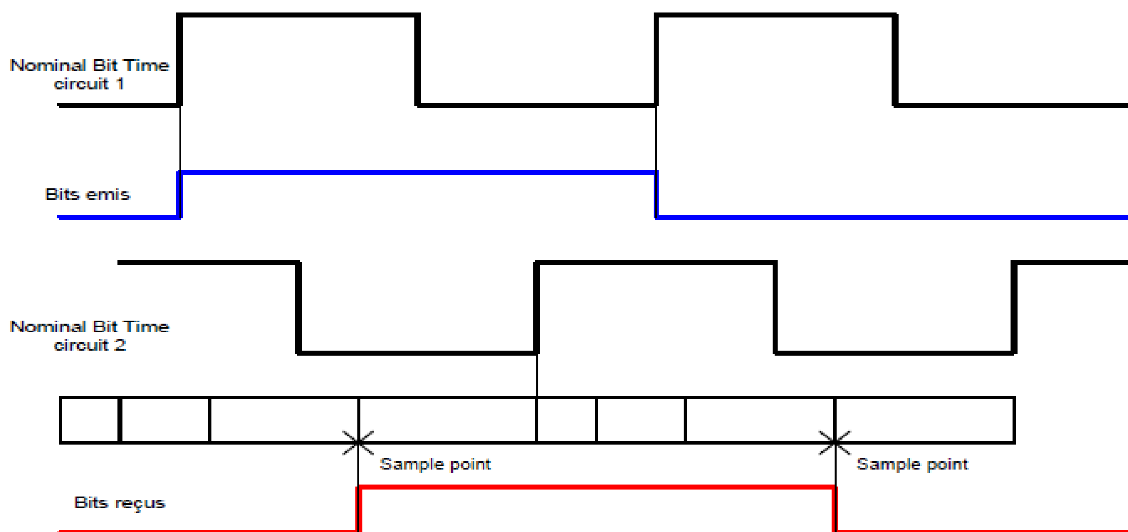


Figure 1.11: problème de synchronisation [8]

La norme BOSCH prévoit donc des règles de synchronisation du Nominal Bit Time de chaque circuit pour éviter les inconvénients exposés ci-dessus.

I.5.4.2. Règles de synchronisation

a) Notion de SJW

Afin de remédier aux problèmes de synchronisation et de s'assurer que la valeur lue est toujours la bonne, il a été défini une butée à ne pas dépasser. Elle s'appelle la SJW (*Synchronization Jump Width)*. Sa valeur est programmée lors de l'initialisation de chaque nœud et elle ne change pas durant le fonctionnement. Elle est comprise entre 1 et le minimum de (4, la durée du buffer phase1).

b) Notion d'erreur de phase

Cette erreur notée e est signalée lorsqu'une transition d'un bit est détectée en dehors du segment de synchronisation. Le calcul de **e** est fait de la manière suivante :

− **e = 0**, si la transition s'effectue dans le segment de synchronisation.

− **e > 0**, si la transition s'effectue avant le point d'échantillonnage.

− **e < 0**, si la transition s'effectue après le point d'échantillonnage.

c) Hard-synchronization

Une *hard-synchronization* a lieu lors de l'émission du SOF et lorsque durant la transmission de la trame une transition a lieu dans le segment de synchronisation. Il résultera de ce type de synchronisation l'abandon du *Nominal bit time en cours*, et dés le *Time Quantum* suivant un autre *Nominal Bit Time* redémarre dés le début.

d) Resynchronisation

Les deux notions introduites précédemment (la SJW et l'erreur de phase) gèrent Le calcul et l'ordre de resynchronisation selon les règles suivantes :

- Si l'erreur de phase est nulle (e = 0), l'effet de la resynchronisation est le même que celui de la *hard-synchronization*.

- Si l'erreur de phase est positive et inférieure en valeur absolue à SJW (0<e<SJW), le segment phase buffer1 sera rallongé de e.

- Si l'erreur de phase est négative, mais inférieure à SJW en valeur absolue (e<0 et |e|<SJW) le segment phase buffer2 est raccourci de e.

- Si l'erreur de phase est positive et supérieure ou égale SJW (e>0 et e>SJW), le segment de phase buffer1 est rallongé de SJW.

- Enfin, si l'erreur de phase est négative et supérieure à SJW (en valeur absolue -e<0 et |e|>SJW) le segment de phase buffer2 est raccourci de SJW. [8]

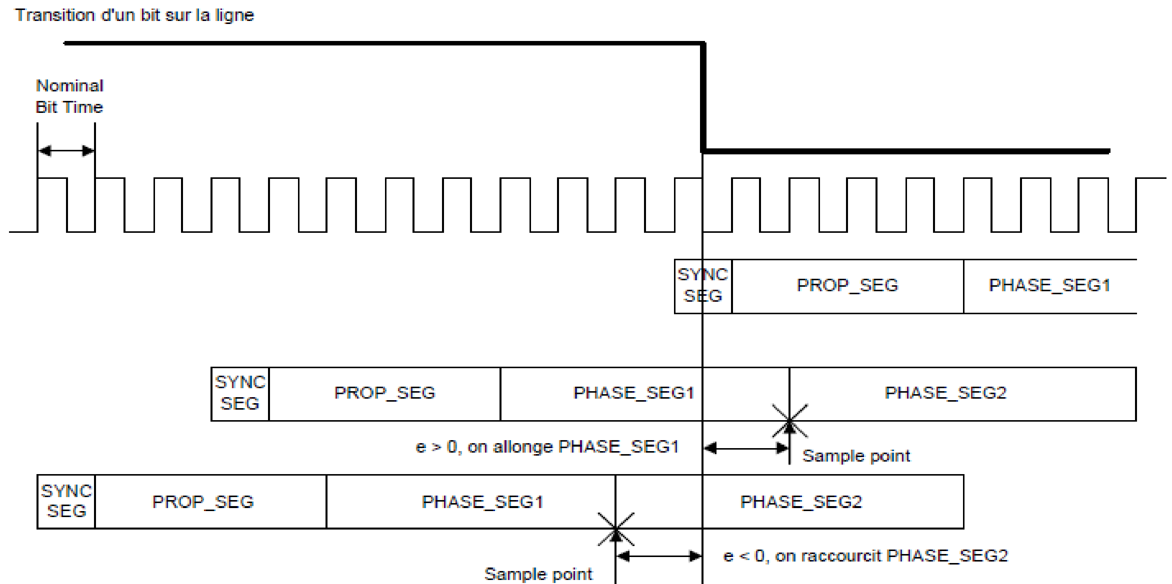


Figure 1.12: erreur de phase [8]

La figure ci-dessus donne un exemple des conséquences des emplacements des transitions sur la longueur des segments du Nominal Bit Time.

I.6. Couche liaison de données

La couche liaison de données est subdivisée en deux sous-couches (LLC Logic Link Control), et MAC (Medium Access Control). La sous-couche MAC représente le noyau du protocole CAN. Elle a pour fonction de présenter les messages reçus en provenance de la sous-couche LLC et d'accepter les messages devant être transmis vers la sous-couche LLC. Elle est responsable de :

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

La sous-couche LLC s'occupe quant à elle :

- du filtrage des messages.
- de la notification de surcharge (Overload).
- de la procédure de recouvrement des erreurs.

I.6.1. Les types de trames

Le transfert des messages se manifeste et est commandé à l'aide de deux types de trames spécifiques et d'un intervalle de temps les séparant. Il s'agit de la trame de données, et la trame de requête.

Par ailleurs il existe deux types de format (format standard, format étendu) pour les trames, et ils diffèrent seulement l'un de l'autre par l'identificateur (identificateur de 11 bits pour les trames standards, de 29 bits pour les trames étendues).

I.6.1.1. Trame de données

Une trame de données transporte les données des émetteurs vers les récepteurs. Elle se décompose en 7 champs différents :

- le début de trame SOF (*Start Of Frame*), 1 bit dominant.
- le champ d'arbitrage, (12 bits pour une trame standard et 30 bits pour une trame étendue).
- le champ de commande, 6 bits.
- le champ de données, 0 à 64 bits.
- le champ de CRC (*Cyclic Redundancy Code*), 16 bits.
- le champ d'acquittement *(Acknowledge*), 2 bits.
- le champ de fin de trame EOF (*End Of Frame*), 7 bits récessifs

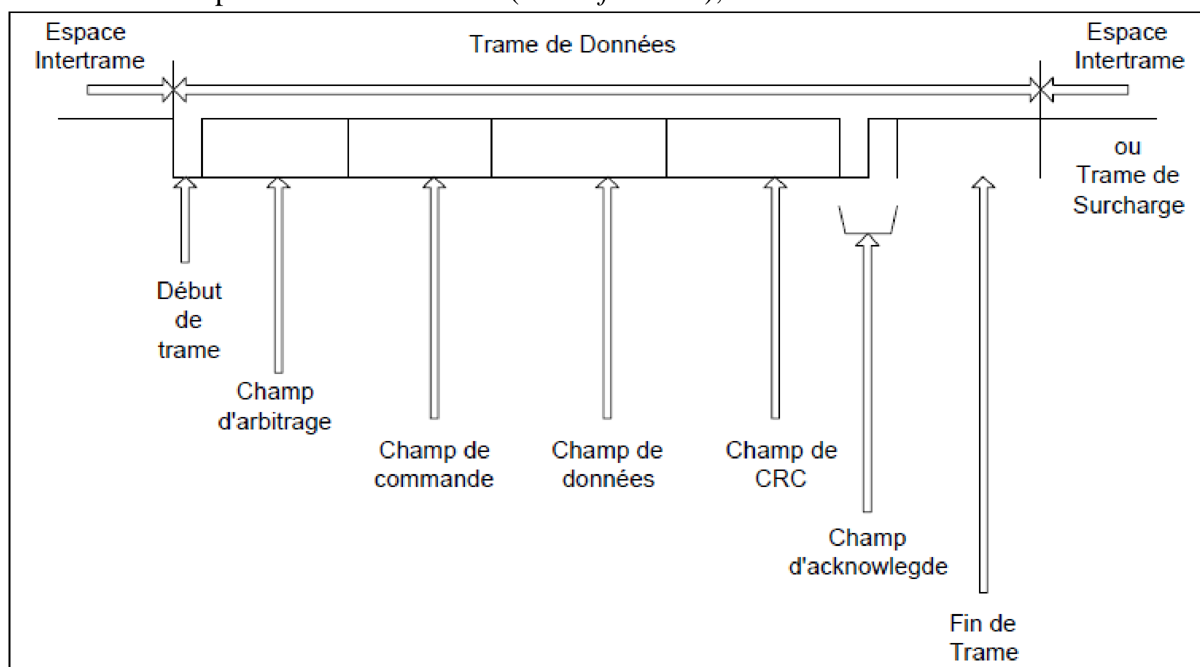


Figure 1.13: Trame de données [8]

a) Début de trame

Le début de la trame (SOF) est constitué d'un seul bit dominant signalant à toutes les stations le début d'un échange. Le début de trame n'est effectif que si le bus était précédemment au repos.

b) Champ d'arbitrage

Dés que le bus est libre, chaque station désirant envoyer un message peut le faire. Afin de déterminer lequel des nœuds est le plus prioritaire, on utilise un procédé d'arbitrage. Le champ pendant lequel s'effectue ce dernier est constitué des bits de l'identificateur et d'un bit de RTR (*Remote Transmission Request*) qui est dominant une trame de données.

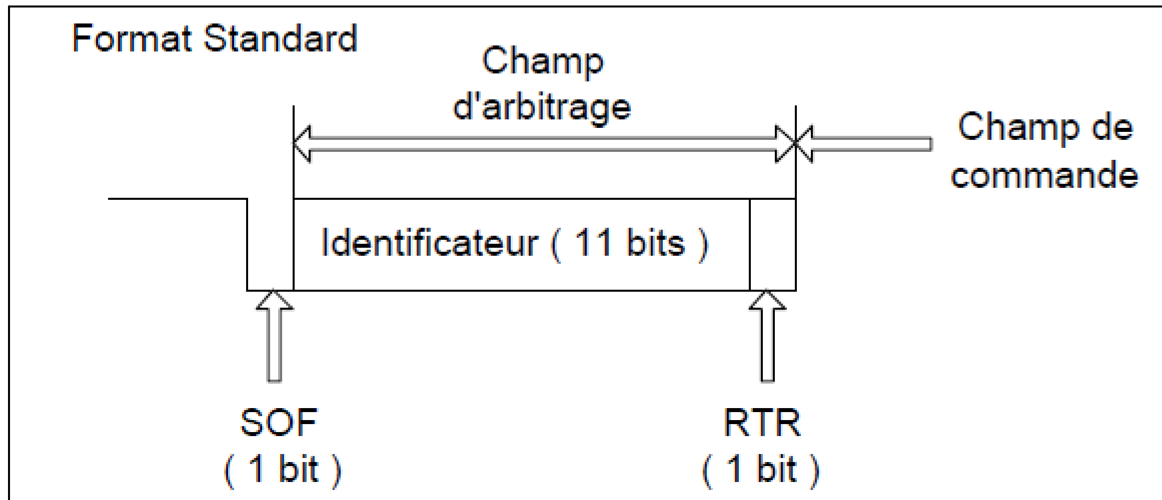


Figure 1.14: Champ d'arbitrage [8]

L'identificateur

Pour assurer le routage du message, le contenu d'un message est repéré par un identificateur. Cet identificateur n'indique pas la destination du message mais la signification des données du message. Ainsi tous les nœuds reçoivent le message, et chacun est capable de savoir grâce au système de filtrage de message si ce dernier lui est destiné ou non.

c) Champ de commande

Il est constitué de 6 bits. Les deux premiers (r1 et r0 dans une trame standard) sont des bits réservés et leur rôle est d'assurer des compatibilités futures (par exemple avec les trames étendues). Les quatre derniers bits permettent de déterminer le nombre d'octets de données contenus dans le champ de données pour une trame de données.

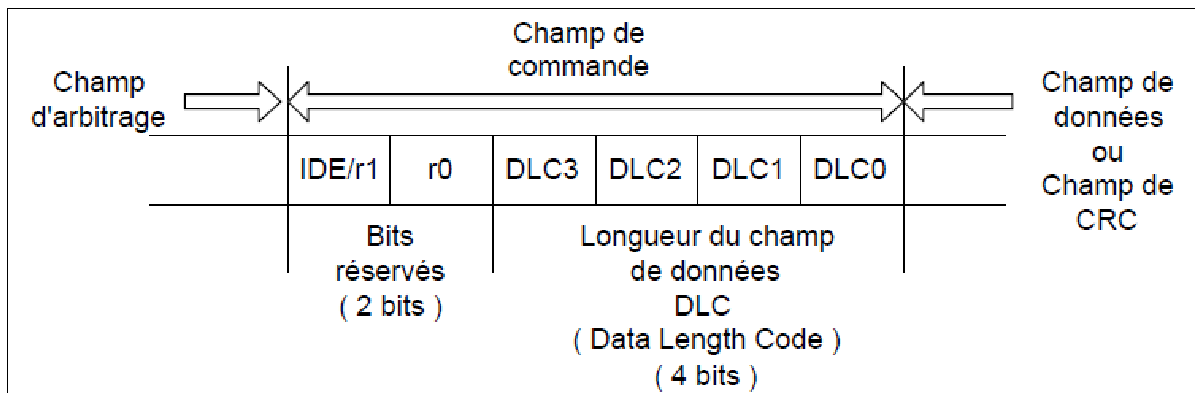


Figure 1.15: Champ de commande [8]

d) Champ de données

Le champ de données a une longueur qui peut varier de 0 à 64 bits (0 à 8 octets). Cette longueur a été déterminée lors de l'analyse du champ de contrôle.

e) Champ CRC

Le champ de CRC est composé de 16 bits. La séquence CRC calculée est contenue dans les 15 premiers bits, tandis que le dernier bit est un délimiteur de fin de champ de CRC qui est toujours récessif.

Ce champ de CRC permet de s'assurer de la validité du message transmis, et tous les récepteurs doivent procéder à cette vérification. Seuls les champs de SOF, d'arbitrage, de contrôle et de données sont utilisés pour le calcul de la séquence CRC. Elle est calculée par la procédure suivante :

- le flot de bits (hors les bits-Stuffing), constitué des bits depuis le début de la trame jusqu'à la fin du champ de données est interprété comme un polynôme f(x) avec des coefficients 0 et 1 affectés à la présence ou l'absence de chaque bit. Le polynôme obtenu est alors multiplié par x^{15}.

- le polynôme ainsi formé est divisé par le polynôme générateur $g(x)=x^{15}+x^{14}+x^{10}+x^{8}+x^{7}+x^{4}+x^{3}+1$. La chaîne de bits correspondante à ce polynôme est : 1100010110011001.

- Le reste de la division du polynôme f(x) par le polynôme générateur g(x) constitue la séquence CRC de 15 bits.

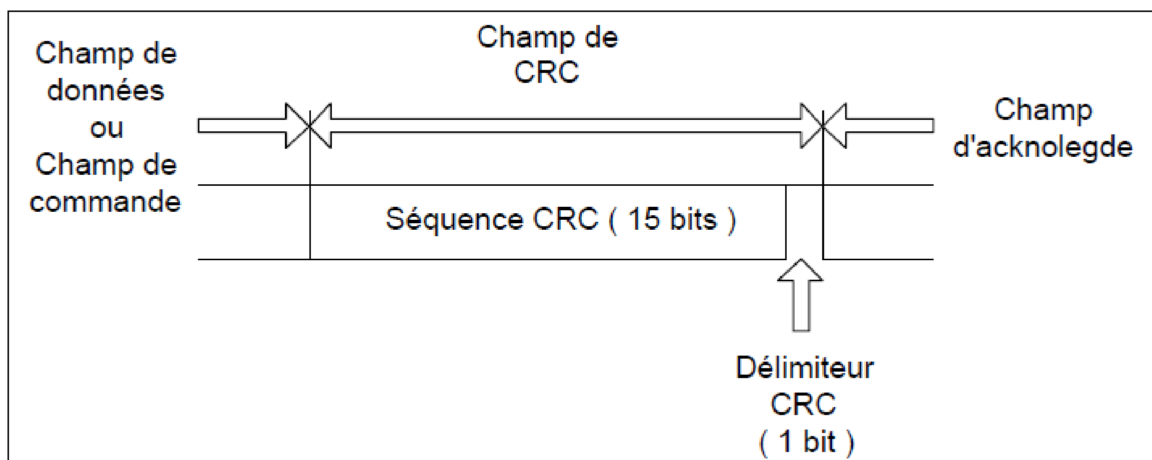


Figure 1.16: Champ CRC [8]

f) Champ d'acquittement

Le champ d'acquittement possède 2 bits. La station émettrice de la trame laisse le bus libre (ce qui correspond à deux bits récessifs) et elle passe en mode réception.

Le premier bit correspond à l'acquittement par l'ensemble des nœuds ayant reçu le message. Si aucune erreur n'a été détectée par un nœud (après calcul du CRC), ce dernier émet un bit dominant sinon il émet une trame d'erreur.

Le second bit est un bit délimiteur d'acquittement qui doit toujours être récessif

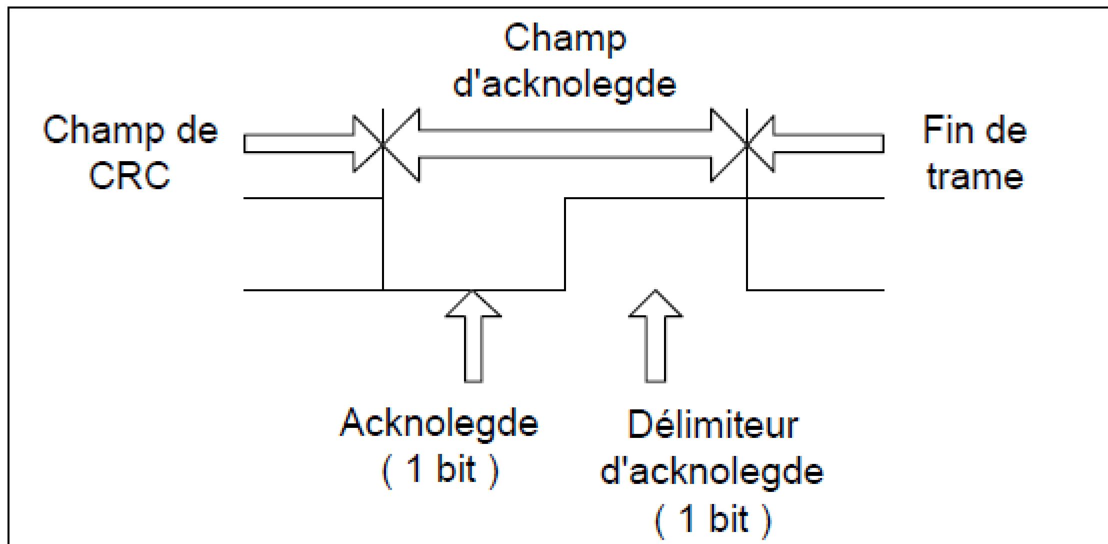


Figure 1.17: Champ d'acquittement [8]

g) Champ de fin de trame

Ce champ de fin de trame est constitué de 7 bits récessifs. Ce champ étant fixe, il est nécessaire de désactiver le codage (à l'émission) et le décodage (à la réception) suivant la règle du Bit-Stuffing.

I.6.1.2. Trame de requête

Un nœud peut demander à un autre nœud d'envoyer une trame de données, et pour cela il envoie lui-même une trame de requête. La trame de données correspondante à la trame de requête initiale possède le même identificateur.

Une trame de requête est constituée de la même manière qu'une trame de données avec un bit RTR à l'état récessif et un champ de données vide. Les bits DLC du champ de commande déterminent le nombre d'octets de données dont a besoin le nœud émetteur.

Les règles de construction des autres divers champs d'une trame de requête sont les mêmes que dans le cas d'une trame de données.

Si deux nœuds émettent chacun une trame possédant le même identificateur (c'est à dire qu'un nœud émet une trame de données et l'autre une trame de requête), l'arbitrage sur le bit de RTR va donner la priorité à la trame de données.

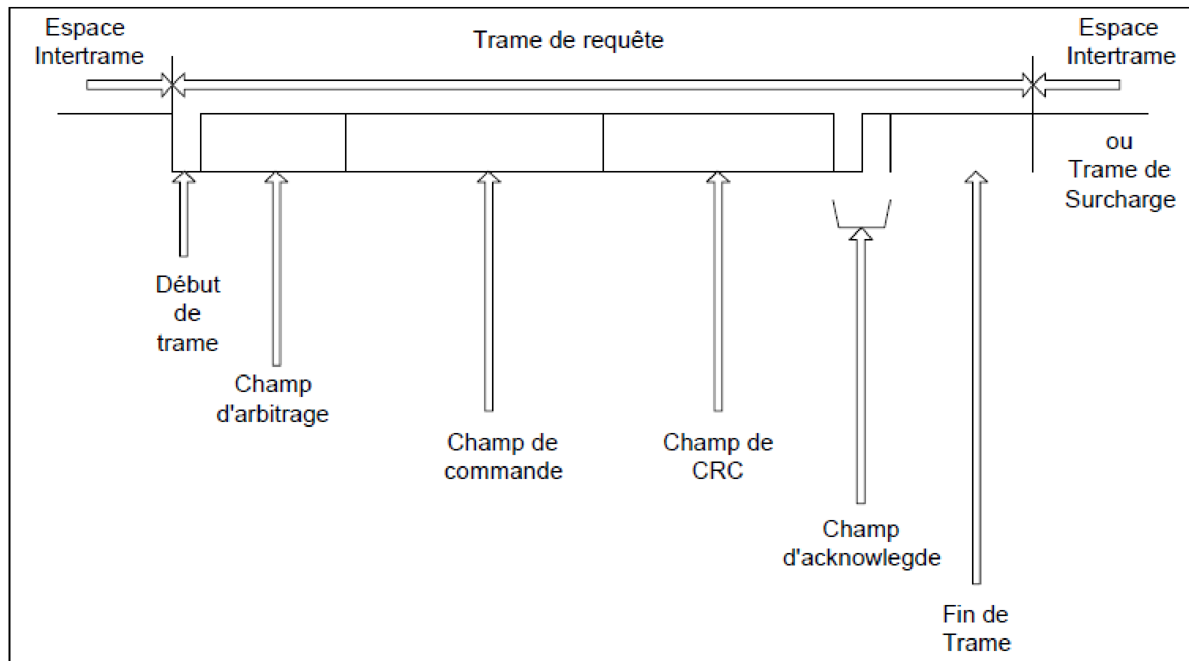


Figure 1.18: Trame de requête [8]

I.6.1.3. Période d'inter-trame

Elle sépare les trames de données ou de requête des autres trames (de données, de requête, d'erreur, et de surcharge). Il s'agit d'une suite de plusieurs bits récessifs. Cette période se compose de deux ou trois champs selon les circonstances.

Le champ d'inter-mission

Ce champ est constitué de 3 bits récessifs consécutifs. L'émission d'une nouvelle trame durant ce champ n'est pas autorisée.

Le champ de Bus Idle

Ce champ désigne que le bus est au repos, il se caractérise par l'absence de trame circulant sur le bus. Il est constitué d'une succession de bits récessifs dont le nombre peut être arbitrairement choisi.

Le champ de suspension de transmission

Ce champ est constitué de 8 bits récessifs. Il n'est envoyé que par les stations qui sont en mode *error passive* après avoir émis une trame d'erreur. Il se situe entre le champ d'intermission et le champ *bus idle*.

Une nouvelle trame pourra démarrer durant ce champ, ce qui aura pour conséquence que le nœud en *error passive* se tait et devient récepteur du nouveau message.

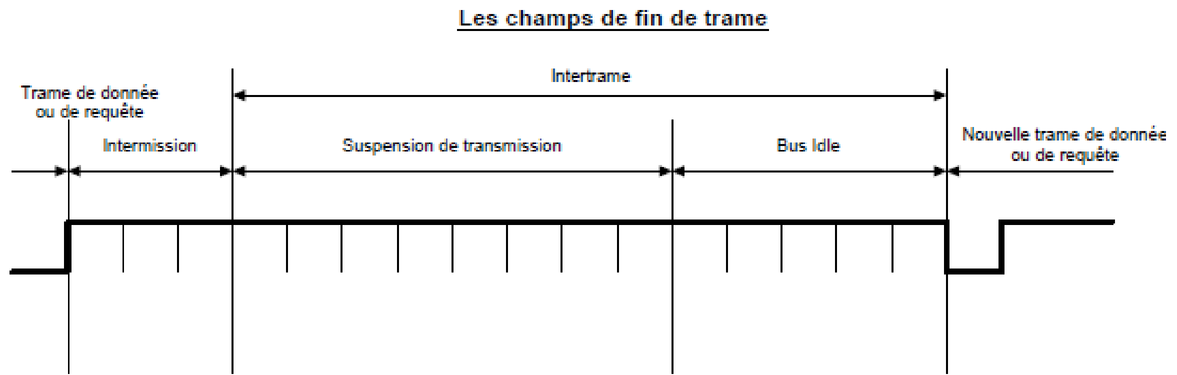


Figure 1.19: Période d'intertrame [8]

I.6.1.4. Conditions de surcharge

Il y a deux sortes de conditions de surcharge qui mènent toutes les deux à la transmission d'une trame de surcharge:

- ✓ les conditions internes d'un récepteur qui nécessitent un certain temps pour accepter la prochaine trame de données ou de requête.
- ✓ la détection d'un bit dominant durant la phase intermission. Dans ce cas le démarrage de la trame de surcharge a lieu juste après la détection du bit dominant.

Une trame de surcharge est formée de deux champs :

- le drapeau de surcharge (*Overload Flag*) constitué de six bits dominants,

- le délimiteur de surcharge (*Overload Delimiter*) constitué de huit bits récessifs.

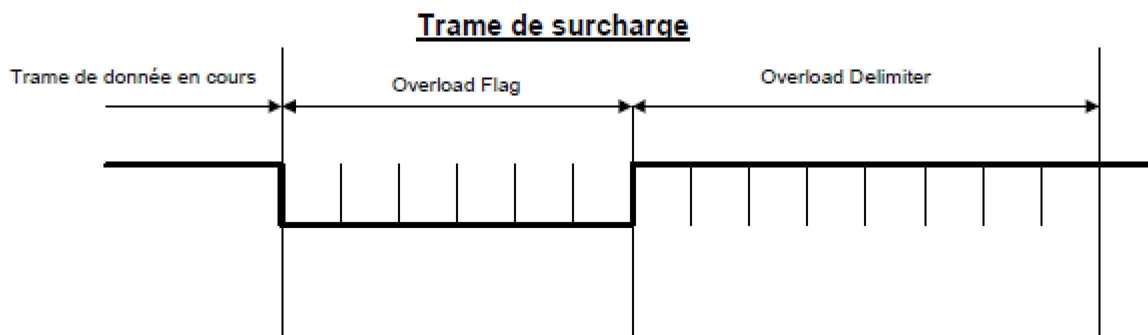


Figure 1.20: Trame de surcharge [8]

Dès qu'une trame de surcharge est émise, les autres nœuds voient sur le bus une suite de six bits dominants qui ne respectent pas la règle du Bit-Stuffing. Ils émettent à leur tour une trame de surcharge. Seulement deux trames de surcharges consécutives sont autorisées sur le bus (pas plus de 12 bits dominants consécutifs émis sur le bus).

I.6.2. Traitement des erreurs

Dans le but d'obtenir la plus grande sécurité lors des transferts sur le bus, des dispositifs de signalisation, de détection d'erreurs, et d'autotests ont été implémentés sur chaque nœud d'un réseau CAN. On dispose ainsi d'un monitoring bus (vérification du bit émis sur le bus), d'une procédure de contrôle de l'architecture du message, d'une méthode de Bit-Stuffing. On détecte alors toutes les erreurs au niveau des émetteurs. La probabilité totale de messages entachés d'erreurs est inférieure à $4.7*10^{-11}$.

I.6.2.1. Les différents types d'erreurs

a) Le bit Error

Lors de l'émission d'une trame, le nœud émetteur procède à une vérification du bus afin de signaler une erreur de type *bit error* si elle a lieu. Cette erreur est signalée dans le cas où le niveau émis par le nœud n'est pas le même qui circule sur le bus sauf dans les cas suivants :

- Durant le champ d'arbitrage, si le nœud envoie un bit récessif et détecte un bit dominant cela signifie une perte d'arbitrage.

- Lors de *l'acknowledge slot*.
- Durant le flag d'erreur passive.

b) L'erreur de stuffing

Une erreur de Stuffing est détectée à chaque fois qu'il y a 6 bits identiques ou plus consécutifs sur le bus. Cette erreur n'est détectée que pour les champs *start of frame*, d'arbitrage, de commande, de donnée, et jusqu'à la fin du champ CRC.

c) L'erreur de Cyclic Redundancy Code (CRC Error)

Si la valeur du CRC calculée par le récepteur est différente de celle envoyée par l'émetteur, il y a erreur de CRC (*CRC Error*).

d) L'erreur d'Acknowledge Delimiter

Une erreur d'*Acknowledge Delimiter* est signalée lorsque le récepteur n'observe pas un bit récessif lors du champ d'*Acknowledge Delimiter*. Il en est de même pour le CRC Delimiter.

e) L'erreur de Slot Acknowledge (Acknowledgment Error)

Une erreur de *Slot Acknowledge* est signalée par l'émetteur s'il ne lit pas un bit dominant lors du champ de *slot acknowledge*.

La figure ci dessous résume les différents types d'erreurs et leur validité suivant l'endroit où l'on se trouve dans la trame.

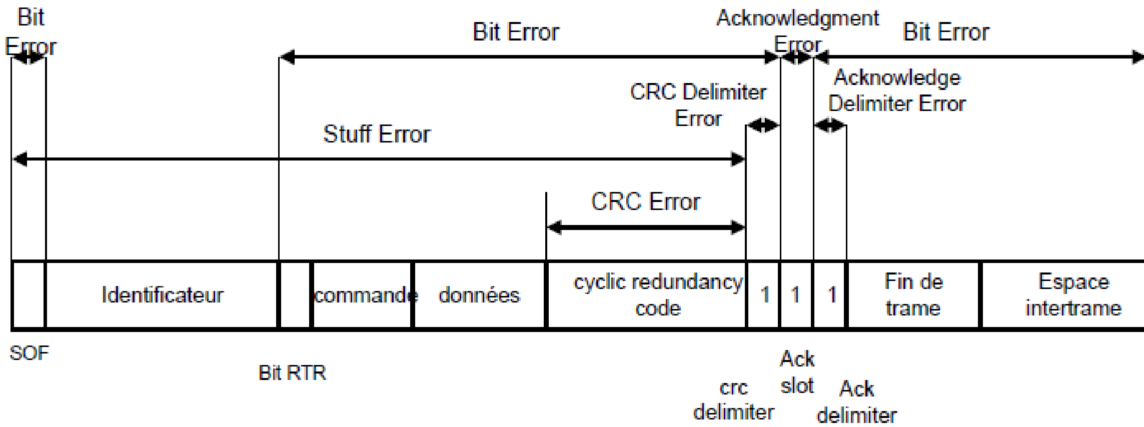


Figure 1.21: Les sources d'erreurs dans une trame CAN [8]

I.6.2.2. signalisation des erreurs

Tous les messages entachés d'erreur sont signalés au niveau de chaque nœud par un drapeau d'erreur. Les messages erronés ne sont pas pris en compte.

Les trames d'erreurs sont constituées de deux champs :

- le drapeau d'erreur,
- le délimiteur de champ.

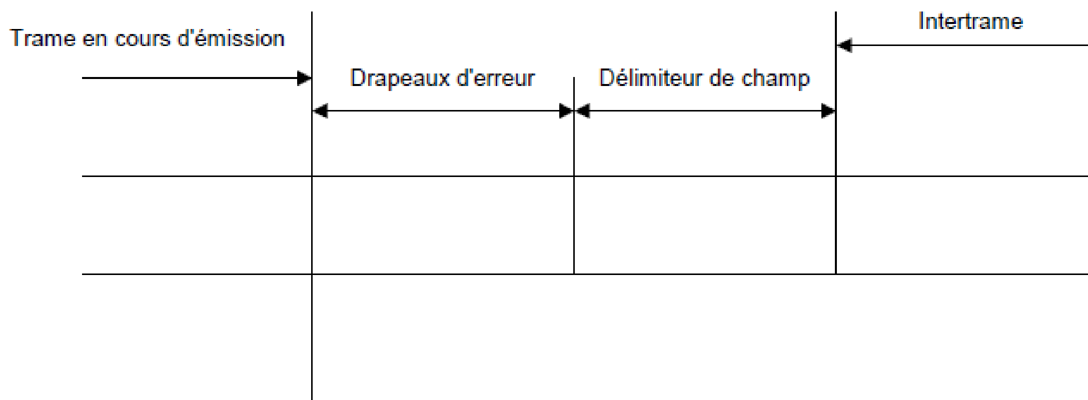


Figure 1.22: Construction de la trame d'erreur [8]

Dans une trame d'erreur Le champ des drapeaux peut être constitué de deux sortes de drapeaux :

- les drapeaux d'erreur active (Active Error Flag),
- les drapeaux d'erreur passive (Passive Error Flag).

Les trames diffèrent suivant le type de drapeaux qu'elles contiennent.

a) La trame d'erreur active

Elle est formée de six bits dominants consécutifs pour le champ de drapeau suivi de huit bits récessifs pour le délimiteur de champ. Par construction, la trame d'erreur brise la règle du Bit-Stuffing. Les autres récepteurs vont donc se mettre à émettre des trames d'erreurs à la fin du premier drapeau.

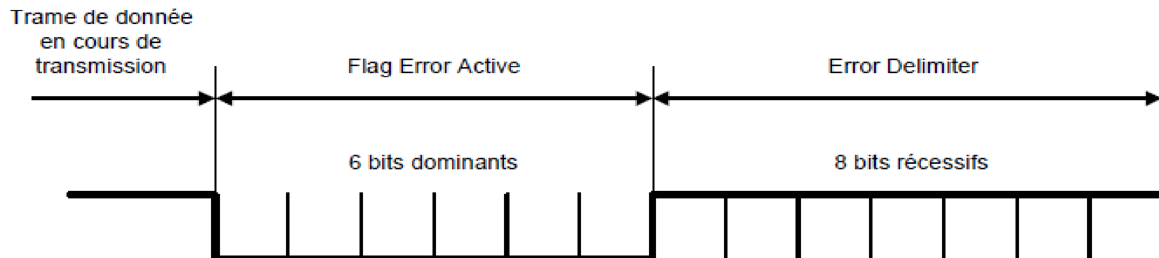


Figure 1.23: Trame d'erreur active [8]

b) La trame d'erreur passive

Cette trame est formée de six bits récessifs pour le drapeau et de huit bits récessifs pour le délimiteur de champ. Le champ du drapeau brise de nouveau la règle du *Bit-Stuffing* et les émetteurs envoient à tour de rôle des trames d'erreur. Une trame d'erreur active est prioritaire sur une trame d'erreur passive si elles sont envoyées en même temps.

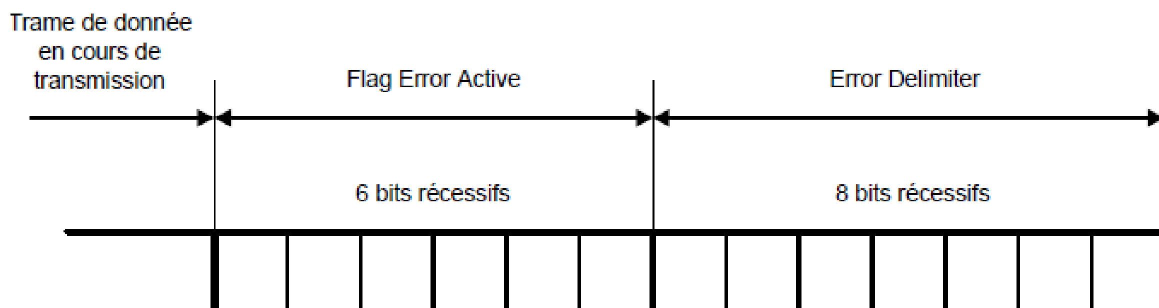


Figure 1.24: Trame d'erreur passive [8]

I.6.2.3. Recouvrement des erreurs

Le recouvrement des erreurs est assuré par la retransmission automatique de la trame erronée jusqu'à ce que l'émission de cette trame s'effectue sans erreur. Le temps de recouvrement (entre le moment où l'on détecte l'erreur et le moment où redémarre le nouveau message) est au maximum de 29 bits pour le CAN 2.0A s'il n'y a pas d'autres erreurs détectées et de 31 bits pour le CAN 2.0B.

La validité du message est acquise s'il n'y a aucune erreur depuis le SOF (*Start Of Frame*) jusqu'à la fin de trame.

a) Les modes d'erreur

Mode d'erreur active

Un nœud en mode erreur active continuera de recevoir et d'émettre normalement. Lors de la détection d'une erreur, il transmettra une trame d'erreur avtive.

Mode d'erreur passive

Un nœud en mode erreur passive continuera de recevoir et d'émettre normalement. Lors de la détection d'une erreur, il transmettra une trame d'erreur passive.

Mode Bus Off

On dit qu'un nœud est en mode *bus off* lorsqu'il se déconnecte totalement du bus (les drivers de lignes ne sont plus actifs). Le protocole autorise un nœud *bus off* à redevenir *error active* (en ayant remis tous ses compteurs d'erreurs à zéro) après que celui-ci ait observé, sans erreur sur le bus 128 séquences de onze bits récessifs (montrant ainsi que de nombreux messages dont les bits de *ACKnowledge delimiter*, *end of frame*, et ceux du champ d'intermission sont bien passés).

b) La gestion des modes d'erreur

Chaque nœud contient deux compteurs d'erreur, un pour les erreurs en réception REC *Receive Error Counter* et l'autre pour les erreurs en émission TEC *Transmit Error Counter.*

Selon les valeurs qu'indiquent ces deux compteurs, diffère le mode d'erreur du nœud en question. La figure ci-dessous donne les règles de passage dans les différents modes :

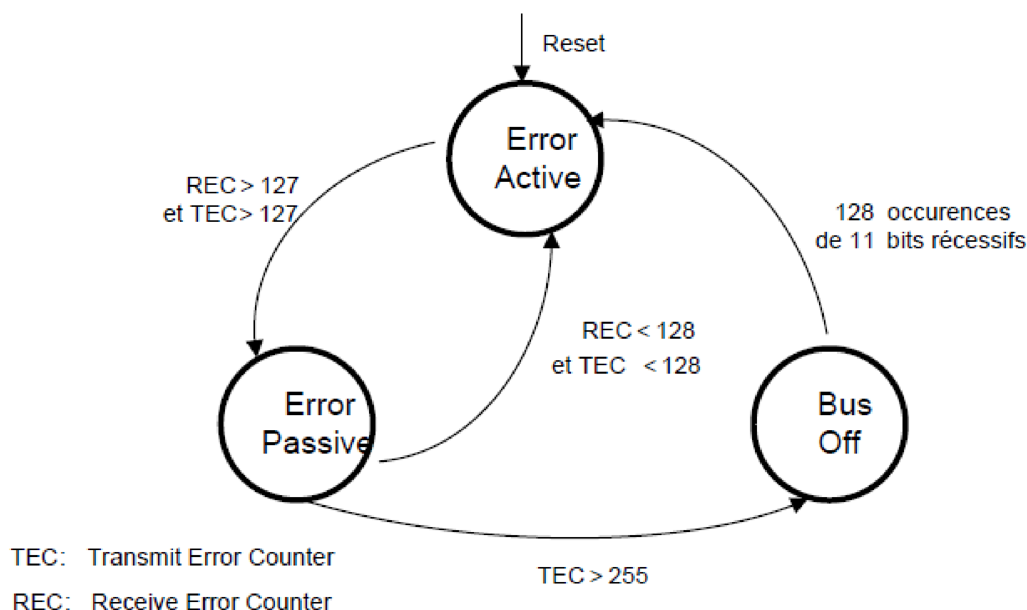


Figure 1.25: Compteur d'erreur et état d'un nœud [8]

La station est en mode d'erreur active à l'initialisation et garde ce mode tant que le compteur de réception ainsi que le compteur d'émission ont une valeur inférieure à 127.

La station est en mode d'erreur passive si la valeur de l'un des compteurs est supérieure ou égale à 128 et inférieure à 255.

La station est en mode *bus off* si la valeur du compteur TEC est supérieure à 255.

c) Les règles d'incrémentation et de décrémentation des compteurs d'erreur

Il existe plusieurs règles qui gèrent l'incrémentation et la décrémentation des compteurs d'erreur. Il faut bien signaler que lorsqu'une erreur est détectée les compteurs s'incrémentent plus vite qu'ils ne se décrémentent lorsque la trame reçue est correcte.

Les règles d'incrémentation et de décrémentation des compteurs sont les suivantes.

	Compteur d'erreur de réception	Compteur d'erreur d'émission
Quand un émetteur détecte un *bit error* pendant l'envoi d'un *active error flag* ou d'un *overload flag*		s'incrémente de 8
L'émetteur envoie un *error flag*		s'incrémente de 8
L'émetteur envoie un *error flag* quand il détecte un *acknowledgement error* pendant l'envoi du *passive error flag*.		Pas de changement
L'émetteur envoie un *error flag* pour un *bit error* pendant le champ d'arbitrage		Pas de changement
Un récepteur détecte une erreur	s'incrémente de 1	
Un récepteur détecte une erreur de *bit error* durant l'*overload flag* ou l'*error active flag*	s'incrémente de 8	
Un récepteur détecte un bit dominant comme étant le premier bit après avoir envoyer un *flag error*	s'incrémente de 8	
Un nœud détecte le 14ème bit consécutif dominant dans le cas d'une *overload flag* ou d'un *active error flag*.	s'incrémente de 8	s'incrémente de 8
Un nœud détecte le 8ème bit consécutif dominant dans le cas d'un *passive error flag*.	s'incrémente de 8	s'incrémente de 8
Un nœud détecte une séquence de 8 bits dominants	s'incrémente de 8	s'incrémente de 8

Tableau 1.3 : Incrémentation et décrémentation des compteurs d'erreur.

I.7. Les composants CAN

I.7.1. Architectures générales des composants CAN

Selon l'utilisation du réseau CAN on peut détecter plusieurs types de composant pouvant véhiculer des trames sous le protocole CAN et sur plusieurs supports de transmission.

La première solution consiste à l'utilisation d'un gestionnaire de protocole (ou *stand alone protocol handler*) ; de même l'adaptateur de ligne est un module séparé, de façon à laisser une grande liberté d'utilisation suivant les supports de transmission choisis.

Lors d'intégration plus poussée, l'architecture se réduit car les utilisateurs, pour des raisons de prix et/ou de performances, recherchent des composants comprenant le gestionnaire de

protocole et le processeur de traitement numérique sur un même bloc. On arrive dans ce cas à des solutions dites microcontrôleurs à gestionnaire CAN intégré.

Enfin, pour des applications simples dans lesquelles la nécessité de disposer concrètement d'une puissance de calcul CPU à bord du nœud CAN n'est pas vraiment indispensable, il est bon d'avoir à sa disposition des composants capables de communiquer selon le protocole CAN, mais de complexité beaucoup moins grande. Ces dispositifs assurent généralement de simples fonctions d'entrées/sorties pilotées sur le réseau.

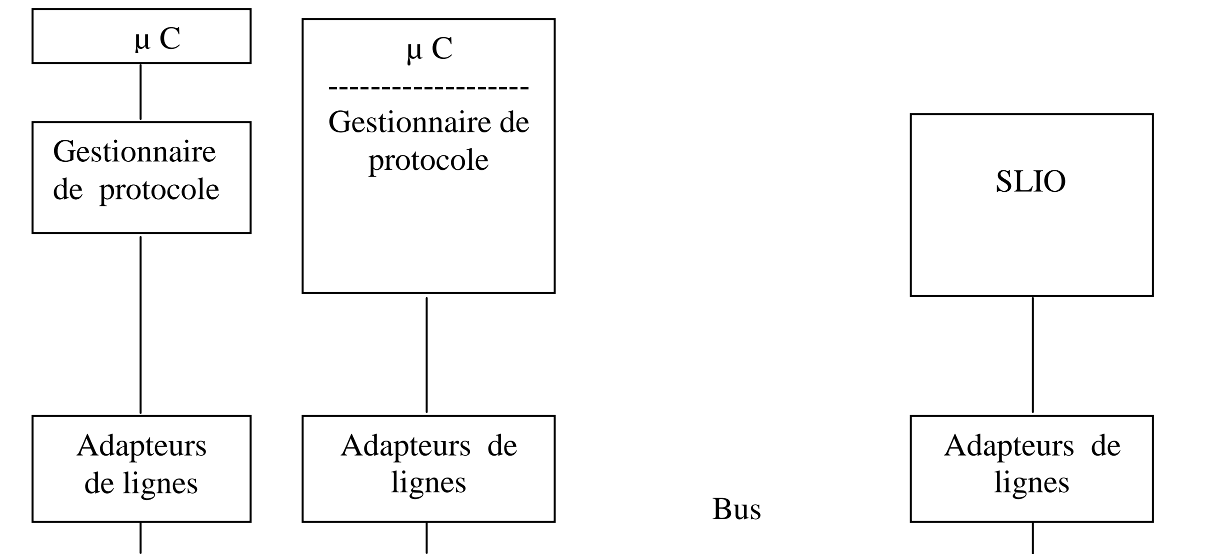


Figure 1.26 : Constitution d'un nœud CAN

I.7.2. Les familles des composants

I.7.2.1. Gestionnaire de protocole

Il assure la gestion du protocole. Ces circuits sont aussi appelés *protocol handler* de type *stand alone*. Ils sont généralement incapables de fonctionner seuls et doivent être pilotés par un micro-contrôleur (8/16/32bits).

I.7.2.2. Microcontrôleur à CAN intégré

Afin de réduire le coût que peut représenter l'ensemble des deux composants (gestionnaire de protocole et microcontrôleur) et la surface de circuit imprimé, il a été proposé des microcontrôleurs comportant un gestionnaire CAN à son bord.

I.7.2.3. SLIO

Les fabricants de composants ont conçu quelques circuits afin de réaliser des nœuds simples à fonctionnalités d'entrées/sorties. C'est ce que l'on appelle les *Serial Linked Input Output devices*.

I.8. Conclusion

L'objectif qui nous a été assigné, à savoir l'étude et la mise en œuvre d'un contrôleur CAN, nous a conduit à la présente étude. Si les notions que nous avons présentées vont au-delà de la mise en œuvre d'un contrôleur CAN, il nous a semblé utile de les présenter.

Le contrôleur objet de notre étude est déduit du très connu SJA1000. Nous présenterons les détails de son architecture et son fonctionnement dans la partie qui va suivre. On ne perdra pas de vue ses modes d'interfaçage nécessaires à son intégration dans une architecture système donnée. C'est le cas de notre projet.

Chapitre II

Etude du contrôleur SJA1000

II.1. Introduction

Nous avons introduit dans notre chapitre précédant le bus CAN comme norme à travers ses diverses spécifications. L'exploitation effective de cette norme passe par son implémentation réelle sous forme d'un contrôleur. Divers circuits existent, qu'il soit en mode *standalone* ou intégré à un microcontrôleur. On peut citer le 82526, le 82527 et le 8xC196CA d'Intel ou le 81C90, le 81C91 et le 80C515 de Siemens. Philips a produit le 82C200, le XA-CAN et le plus connu SJA1000. Nous allons nous intéresser, dans le contexte de notre projet, à un *IP core* dont la description est déduite de ce dernier. Il est donc à notre avis important d'en faire une présentation sous ses deux aspects, architectural et modèle de programmation.

II.2. Présentation du SJA1000

Le SJA1000 fait partie de la famille des gestionnaires de protocole (*prtocol handler*) de type *standalone.* C'est un contrôleur CAN ne nécessitant qu'un microcontrôleur externe. Il a été développé par Philips en 1997 comme un successeur (compatible) du 82C200. [7]

II.3. Caractéristiques

Le SJA1000 a été conçu pour être compatible avec son prédécesseur le 82C200. De plus, des nouvelles fonctions ont été implémentées telles que :

- Buffer de réception de 64 octets ;
- Supporte le CAN 2.0A et 2.0B (trame standard et trame étendue) ;
- Compteurs d'erreur avec accès lecture/écriture ;
- Interruptions pour chaque type d'erreur sur le bus ;
- Capture de la dernière erreur qui a eu lieu ;
- Détails sur le bit de l'identificateur ayant causé une perte d'arbitration ;
- Mode *Listen Only* (aucune action sur le bus).
- Mode *self test.*

Pour cela deux modes de fonctionnement ont été définis le mode BasicCAN qui est un mode simplifié dont la principale utilité est la compatibilité totale avec le 82C200, et le mode PeliCAN dans lequel on peut exploiter toutes les caractéristiques citées ci-dessus. [10]

II.4. Architecture du SJA1000

Dans la figure suivante on voit tous les blocs qui font partie du SJA1000, ainsi que les différents signaux d'entrée et de sortie.

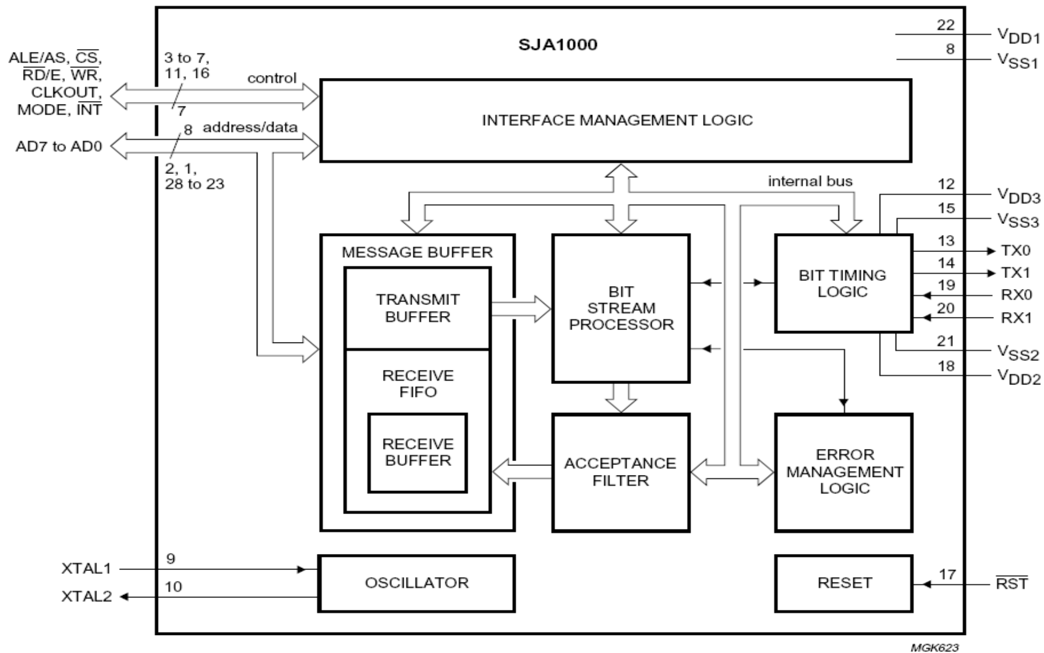


***Figure 2.1 : Schéma de fonctionnement du SJA1000* [10].**

II.4.1. Descriptions des différents blocs du SJA1000

II.4.1.1. Bit Timing Logic

Le *bit timing logic* contrôle la ligne du bus et règle la durée du bit. Il est synchronisé au flux de bits sur le bus CAN. Le BTL procure aussi des segments de temps programmables afin de compenser les retards dus aux propagations et aux erreurs de phase, et définit le point d'échantillonnage ainsi que le nombre d'échantillons qu'il faut prendre. Les deux registres utilisés par ce bloc sont le registre *BUS TIMING0* et le registre *BUS TIMING1.*

II.4.1.2. Bit Stream Processor

Le *bit stream processor* contrôle le flux de données entre le buffer de transmission, la FIFO de réception et le bus CAN. Il s'occupe aussi de la détection d'erreur, l'arbitration, la technique du *stuffing*, et la gestion des erreurs sur le bus CAN.

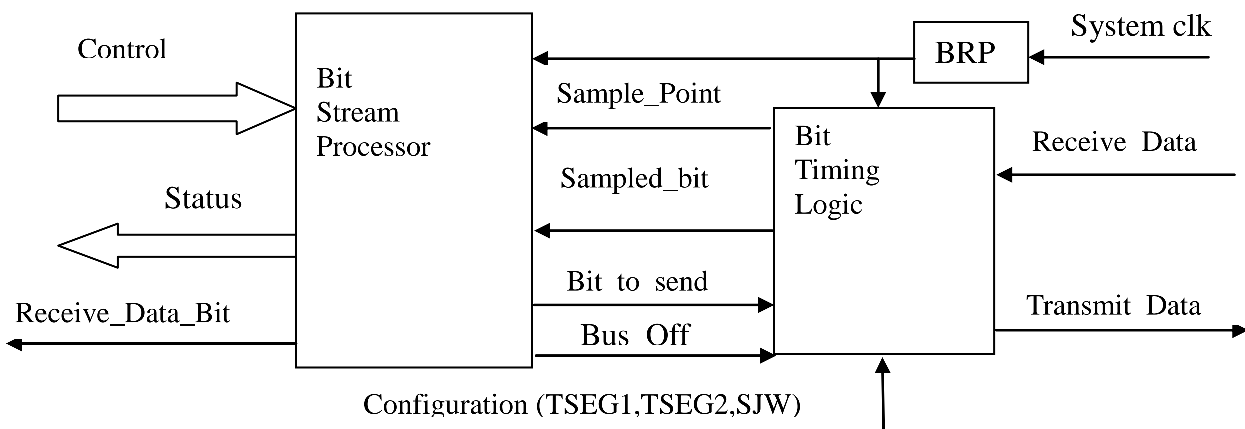


Figure 2.2: Fonctionnement du BTL et du BSP

II.4.1.3. Buffer de réception

Le buffer de réception est une interface entre le filtre d'acceptation et le microcontrôleur. Il enregistre les messages reçus du bus CAN. Il est implémenté sous la forme d'une FIFO de 64 octets. Le SJA1000 donne accès a une "fenêtre" de 13 octets 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. En plus de la fenêtre de réception, l'utilisateur peut également accéder directement aux 64 octets du Buffer. Lorsqu'un message arrive alors que le buffer de réception est plein, le message arrivant est perdu, et l'indicateur de saturation (*Data Overrun STATUS*) est activé.

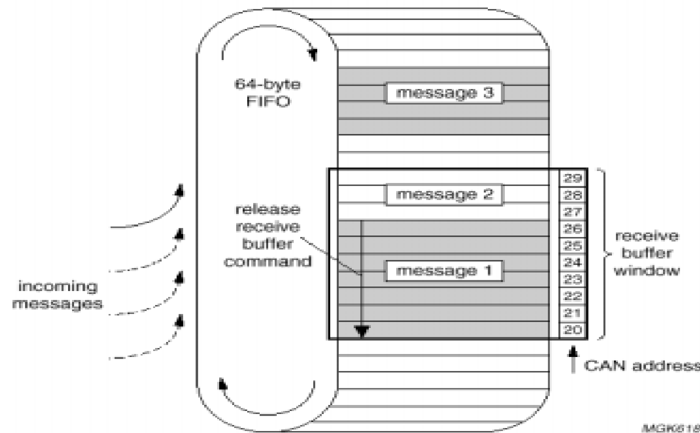
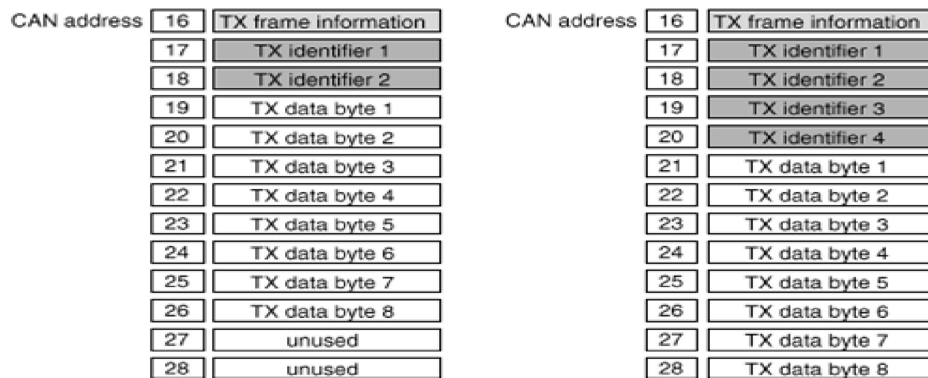


Figure 2.3 : Buffer de réception [14]

II.4.1.4. Buffer d'émission

Le buffer de transmission est une interface entre Le microcontrôleur et le *Bit Stream Processor* (BSP), capable d'enregistrer un message complet dédié à être transmis dans le réseau CAN. C'est un simple buffer de 12 octets (jusqu'`a 4 octets d'identificateur 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.



a)trame standard b) trame étendue

Figure 2.4: Buffer de transmission [14].

II.4.1.5. Le filtre d'acceptation

Le filtre d'acceptation (*Acceptance Filter*) a pour rôle de contrôler les identificateurs des messages présents sur le bus, avant de leur permettre d'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 fonctionne en deux différents modes, le mode single et le mode dual.

II.5. Modèle de programmation du SJA1000

II.5.1. Les différents registres du SJA1000

II.5.1.1. Le registre MODE

Ce registre contrôle le mode de fonctionnement du SJA1000. Il est spécifique au mode PeliCAN.

Outre le mode normal, le SJA1000 peut fonctionner dans les modes :

- *Listen only* où aucune action sur le bus.
- *Self test MODE* où un acquittement par un autre nœud n'est pas nécessaire pour une bonne transmission.
- *Sleep MODE* où le nœud se déconnecte du bus tant qu'il n'existe aucune activité sur celui-ci.

Le registre MODE permet également de régler différents modes de fonctionnement du filtre d'acceptation.

Bit	**Symbole**	**nom**	**Fonctionnement**
MOD.7, 6 et 5	-	-	Réservés (toujours à0)
MOD.4	SM	*Sleep Mode*	SM=1 sleep ; le contrôleur CAN entre en mode sleep s'il n'y a pas d'interruption en attente et pas d'activité sur le bus. SM=0 wake-up ; le contrôleur se réveille s'il dormait
MOD.3	AFM	*Acceptance Filter Mode*	AFM=1mode single. AFM=0 mode dual ;
MOD.2	STM	*Self Test Mode*	STM=1 self test ; une transmission réussie n'a pas besoin d'acknowledge à la réception. STM=0 normal ; un acknowledge est demandé pour une transmission réussie.
MOD.1	LOM	*Listen Only Mode*	LOM=1 listen only ; dans ce mode le contrôleur CAN ne va envoyer aucun acknowledge vers le bus CAN même si un message a été reçu avec succès, les compteurs d'erreur s'arrêtent à la valeur 0. LOM=0 normal
MOD.0	RM	*Reset Mode*	RM=1 reset ; la détection d'un bit RM actif entraîne l'abandon d'une transmission/réception en cours et le nœud se met en mode reset. RM=0 normal ; après la transition 1 à 0 de ce bit le contrôleur CAN retournera en mode opérationnel.

***Tableau 2.1 : Fonctions des différents bits du registre MODE* [10].**

II.5.1.2. Le registre COMMAND

Comme son nom l'indique, ce registre est utilisé pour commander le contrôleur CAN. Il apparait au microcontrôleur comme une *write only memory* (une mémoire à accès d'écriture seulement).

Bit	**Symbole**	**Nom**	**Fonctionnement**
CMR.7, 6et 5	-	-	Réservés
CMR.4	GTS	*Go To Sleep*	GTS=1 sleep ; ordonne au SJA1000 de passer en sleep mode. GTS=0 wake up ;mode normal
CMR.3	CDO	*Clear Data Overrun*	CDO=1 clear ; réinitialise l'indicateur de surcharge. CDO=0 aucune action n'est observée.
CMR.2	RRB	*Release Re-ceive Buffer*	RRB=1 released ; libère la place du dernier message reçu dans le buffer de réception. RRB=0 aucune action
CMR.1	AT	*Abort Transmission*	AT=1 présent ; Annule une demande de transmission faite et qui n'a pas encore commencé. Si elle a déjà commencé, elle se termine.
CMR.0	TR	*Transmission Request*	TR=1 présent ; Demande la transmission des données présentes dans le buffer de transmission.

Tableau 2.2 : Fonctions des différents bits du registre COMMAND [10].

II.5.1.3. Le registre STATUS

Le contenu de ce registre reflète l'état du contrôleur CAN. Le registre d'état apparait au microcontrôleur comme une *Read Only Memory* (une mémoire à lecture seule).

Bit	Symbole	Nom	Fonctionnement
SR.7	BS	*Bus Status*	BS=1 bus-off ; le SJA1000 ne participe pas aux activités du bus. BS=0 bus-on ; le SJA1000 participe aux activités du bus.
SR.6	ES	*Error Status*	ES=1 error ; l'un des compteurs a atteint la limite d'alerte. ES=0 ok ; les deux compteurs sont en dessous de la limite d'alerte.
SR.5	TS	*Transmit Status*	TS=1 transmission ; un message est en cours de transmission. TS=0 au repos.
SR.4	RS	*Receive Status*	RS=1 en réception ; un message est en cours de réception. RS=0 au repos.
SR.3	TCS	*Transmission Com-plete Status*	TCS=1 complétée ; la dernière transmission demandée a été correctement transmise et acquittée. TCS=0 incomplète ; la dernière transmission n'est pas encore complétée.
SR.2	TBS	*Transmit Buffer Status*	TBS=1 libéré ; le buffer de transmission est prêt à recevoir des données.TBS=0 occupé ; un message est en cours de transmission ou il attend d'être transmis.
SR.1	DOS	*Data Overrun Status*	DOS=1 surcharge ; un message a été perdu à cause d'une saturation du buffer de réception. DOS=0 ; aucune saturation de donnée n'est détectée
SR.0	RBS	*Receive Buffer Sta-tus*	RBS=1 plein ; un message est présent dans le buffer de récep-tion. RBS=0 vide ; aucun message n'est présent.

Tableau 2.3 : Fonctions des différents bits du registre STATUS [10].

II.5.1.4. Le registre INTERRUPT

Ce registre permet d'autoriser la génération d'interruption sur la broche *INT*.

Bit	Symbole	Nom	fonctionnement
IR.7, 6 et 5	-	-	Réservés
IR.4	WUI	*Wake-Up Interrupt*	WUI=1 activé ; réveil du nœud pour cause d'activité sur le bus. WUI=0 désactivé ;
IR.3	DOI	*Data Overrun Interrupt*	DOI=1 activé ; lors de la transition 0 à 1 du DOS. DOI=0 désactivé ;
IR.2	EI	*Error Interrupt*	EI=1 activé ; Changement des STATUS d'erreurs (ES et BS).EI=0 désactivé ;
IR.1	TI	*Transmit Interrupt*	TI=1 activé ; quand le TBS change de 0 à 1 (libéré). TI=0 désactivé ;
IR.0	RI	*Receive Interrupt*	RI=1 activé ; quand la FIFO de réception n'est pas vide. RI=0 désactivé ;

Tableau 2.4 : Fonctions des différents bits du registre INTERRUPT [10].

II.5.1.5. Le registre de contrôle

Ce registre est spécifique au mode BasicCan. Son contenu est utilisé pour changer le comportement du contrôleur. Ses bits sont activés ou désactivés par le microcontrôleur qui utilise ce registre comme une mémoire avec accès d'écriture et de lecture.

Bit	Symbole	Nom	Fonctionnement
CR.7, 6 et 5	-	-	réservés
CR.4	OIE	*Overrun Interrupt Enable*	OIE=1 autorisé ; si le bit *data overrun* est activé le microcontrôleur reçoit un signal d'interruption de surcharge. OIE=0 interdit ; le microcontrôleur ne reçoit aucun signal d'interruption de surcharge.
CR.3	EIE	*Error Interrupt Enable*	EIE=1 autorisé ; si le BS ou l'ES change, le microcontrôleur reçoit un signal d'interruption d'erreur. EIE=0 interdit ; le microcontrôleur ne reçoit aucun signal d'interruption d'erreur.
CR.2	TIE	*Transmit Interrupt Enable*	TIE=1 autorisé ; quand un message est transmis avec succès ou le buffer de transmission est accessible, un signal d'interruption de transmission est envoyé vers le microcontrôleur. TIE=0 interdit ; le microcontrôleur ne reçoit aucun signal d'interruption de transmission.
CR.1	RIE	*Receive Inerrupt Enable*	RIE=1 autorisé ; quand le message est reçu sans erreur, le SJA1000 envoi un signal d'interruption de réception vers le microcontrôleur. RIE=0 interdit ; le microcontrôleur ne reçoit aucun signal d'interruption de réception de la part du SJA1000
CR.0	RR	*Reset Request*	RR=1 présente ; la détection d'une RR entraîne l'abandon d'une transmission/réception en cours et le contrôleur entre en mode reset. RR=0 absente ; lors de la transition 1 à 0 du RR le SJA1000 retourne au mode opérationnel.

Tableau 2.5 : Fonctions des différents bits du registre CONTROL [10].

II.5.1.6. Le registre clock divider

Ce registre contrôle la fréquence de l'horloge CLOCKOUT du contrôleur vers le microcontrôleur, et donne la possibilité de la désactiver en mettant le 4ème bit du registre à 1. De même le choix du mode de fonctionnement peliCAN ou basicCAN se fait à l'aide du bit 7 de ce registre on le met à 1 pour le mode PeliCAN et à 0 pour le mode BasicCAN. Ci-dessous un tableau donnant la valeur de la fréquence de l'horloge CLOCKOUT en fonction des bits 0,1, et 2 du registre *clock divider* :

CD2.	CD1.	CD0.	Fréquence de CLOCKOUT
0	0	0	$f_{sja}/2$
0	0	1	$f_{sja}/4$
0	1	0	$f_{sja}/6$
0	1	1	$f_{sja}/8$
1	0	0	$f_{sja}/10$
1	0	1	$f_{sja}/12$
1	1	0	$f_{sja}/14$
1	1	1	f_{sja}

Tableau 2.6: Fréquence de l'horloge CLOCKOUT [10].

II.5.1.7. Le registre BUS_TIMING0

Le contenu de ce registre définit les valeurs du diviseur de vitesse de transmission (BRP : *Baud Rate Prescalar*) et du SJW : *Synchronization Jump Width*.

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
SJW.1	SJW.0	BRP.5	BRP.4	BRP.3	BRP.2	BRP.1	BRP.0

Tableau 2.7: Fonctions des différents bits du registre BUS_TIMING0 [10].

II.5.1.8. Le registre BUS_TIMING1

Son contenu définit la longueur de la période du bit, le positionnement du point d'échantillonnage, et le nombre d'échantillons qui seront pris à chaque point d'échantillonnage.

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

Tableau 2.8: Fonctions des différents bits du registre BUS_TIMING1 [10].

Lorsque le bit7 vaut 1 le bus est échantillonné trois fois, il n'est échantillonné qu'une seule fois lorsqu'il vaut 0. [10]

II.5.1.9. Les différents registres du filtre d'acceptation

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

a) Les Acceptance Mask Registers

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

b) Les Acceptance Code Registers

Les *Acceptance Code Registers* contiennent l'identificateur "optimal" attendu par le SJA1000; les bits de l'*Acceptance Code Register* doivent être égaux aux bits de l'identificateur qui sont marqués pertinents par *l'Acceptance Mask Register*. Si les conditions comme décrites dans l'équation suivante sont remplies, le message est accepté :

[(ID.10 à ID.3) ≡ (AC.7 à AC.0)] ∨ (AM.7 à AM.0) ≡ 11111111

Pour une réception réussie du message, chaque comparaison individuelle doit donner un avis favorable.

II.5.1.10. Configuration du filtre d'acceptation

a) Configuration du filtre en mode single

Dans cette configuration un seul filtre de longueur de 32 bits est défini. La correspondance entre les bits du message et ceux du filtre dépend du format de la trame reçue.

Trame standard

Si une trame standard est reçue, l'identificateur au complet y compris le bit RTR et les deux premiers octets du champ de donnée sont utilisés pour le filtrage d'acceptation. Les messages peuvent aussi être acceptés s'il n'y a aucune donnée ou bien un seul octet est disponible.

Les quatre bits les moins significatifs du registre ACR1 ne sont pas utilisés alors on doit les programmer comme des bits sans importance en mettant AMR1.3, AMR1.2, AMR1.1, et AMR1.0 à 1.

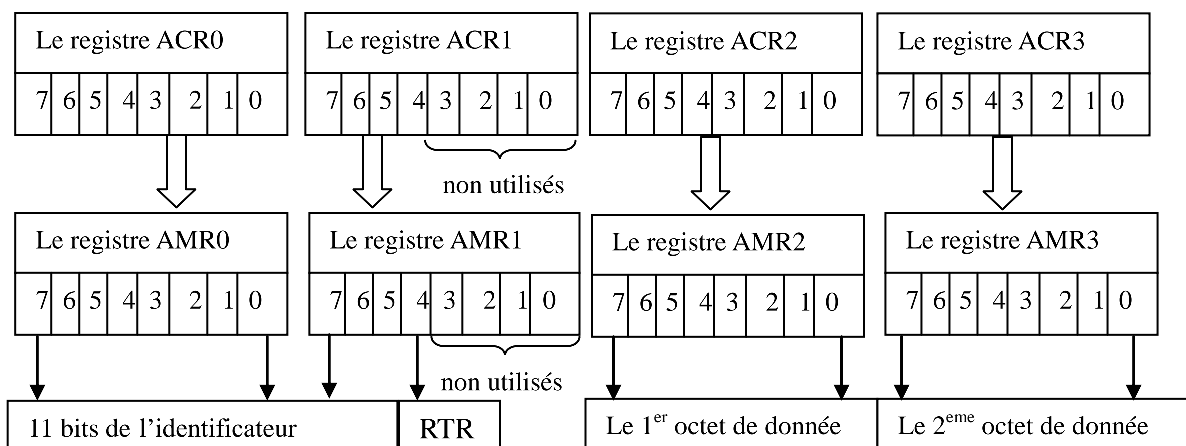


Figure 2.5 : Filtre d'acceptation en mode single pour une trame standard.

Trame étendue

Si une trame étendue est reçue, l'identificateur au complet y compris le bit RTR est utilisé pour le filtrage d'acceptation.

Les deux bits les moins significatifs du registre ACR3 ne sont pas utilisés alors on doit les programmer comme des bits sans importance en mettant AMR3.1, et AMR3.0 à 1.

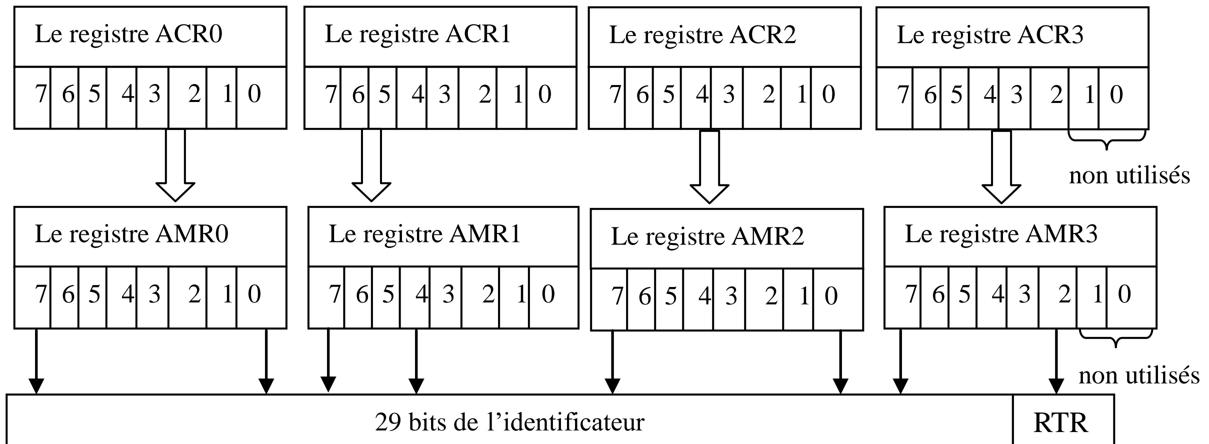


Figure 2.6 : Filtre d'acceptation en mode single pour une trame étendue.

b) Configuration du filtre en mode dual

Dans cette configuration deux filtres sont définis, un message reçu est comparé avec les deux filtres afin de décider si le message doit être copié dans le buffer de réception ou non. Si seulement l'un des filtres affirme la validité du message, le message sera accepté.

Trame standard

Si une trame standard est reçue, le premier filtre compare l'identificateur au complet y compris le bit RTR et le premier octet du champ de donnée. Le second filtre compare l'identificateur au complet y compris le RTR. S'il n'y a aucune donnée, les quatre bits les moins significatifs des registres AMR1 et AMR3 doivent être mis à 1 (sans importance). Alors les deux filtres fonctionneront de la même manière utilisant l'identificateur et le bit RTR

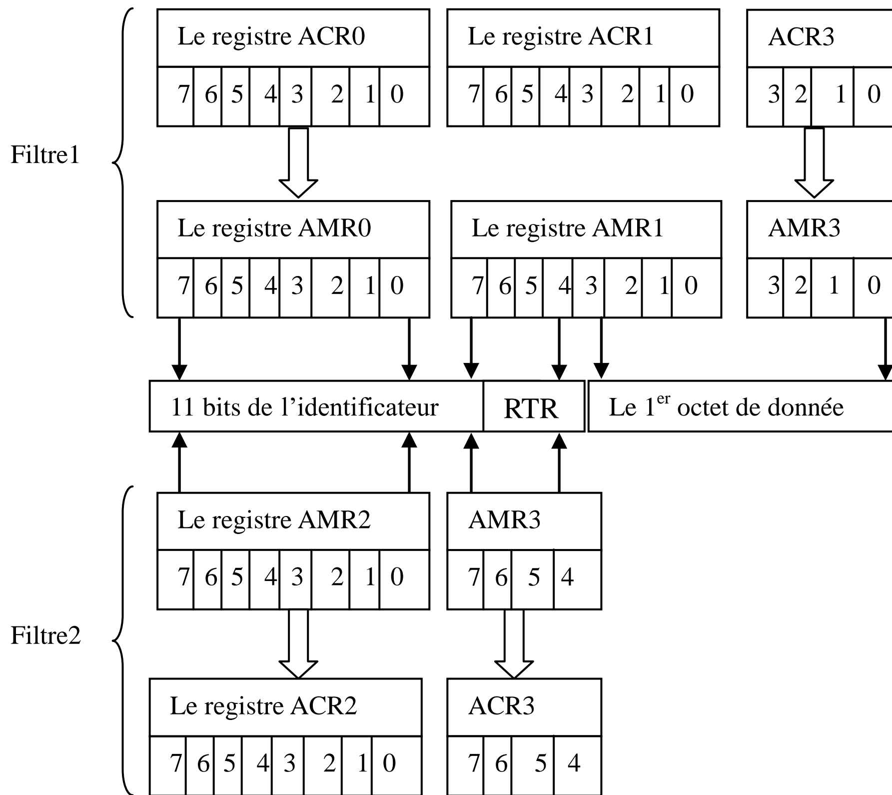


Figure 2.7 : Filtre d'acceptation en mode dual pour une trame standard.

Trame étendue

Si une trame étendue est reçue, les deux filtres fonctionnent de la même manière, les deux filtres comparent les deux octets de l'identificateur.

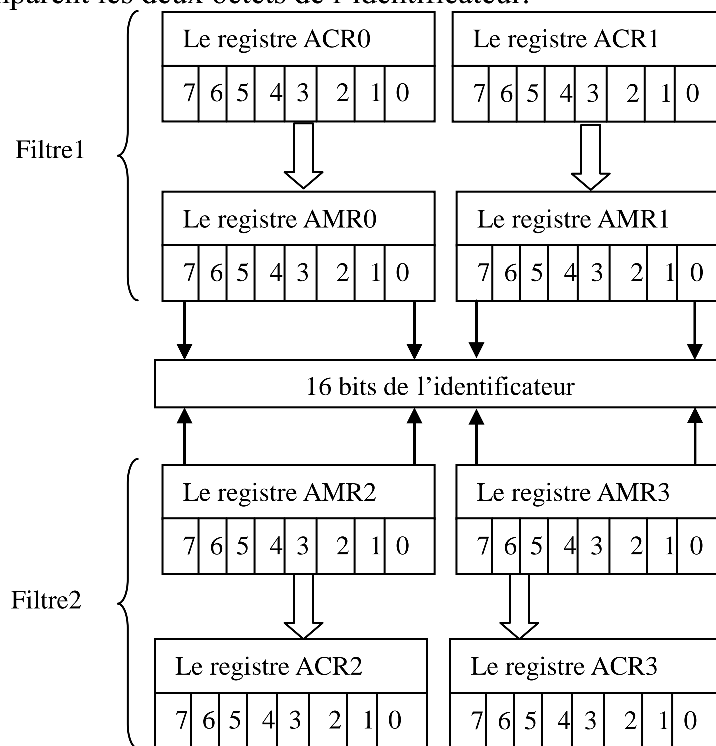


Figure 2.8: Filtre d'acceptation en mode dual pour une trame étendue

II.5.1.11. Plan d'adressage du SJA1000

Il faut noter que quelques registres cités ci-dessus ne sont disponibles que pour le mode PeliCAN. On donne un tableau en annexe n°1 résumant les propriétés des différents registres : leurs adresses, leurs accessibilités en lecture ou en écriture pendant les deux modes, le mode reset et le mode opérationnel ainsi que les valeurs de leurs bits renvoyées pendant le premier mode. [6]

II.5.2. Procédure de communications CAN

Afin de pouvoir entamer une communication au sein du SJA1000, on peut se mettre en mode interruptible ou en mode *polling*. En général les étapes à suivre sont les suivantes :

- initialiser le SJA1000 pour choisir le mode d'utilisation et pour configurer ses différents registres tels que le bus timing0, le bus timing1, le filtre d'acceptation…etc.
- préparer le message à transmettre et faire une demande de transmission.
- Réagir en cas de réception d'un message.
- Réagir en cas d'erreur de transmission.

II.5.2.1. Mode polling

Dans ce mode, on entame une transmission ou une réception selon le contenu du registre STATUS. L'acheminement d'une communication en mode *polling* est donné par l'organigramme suivant :

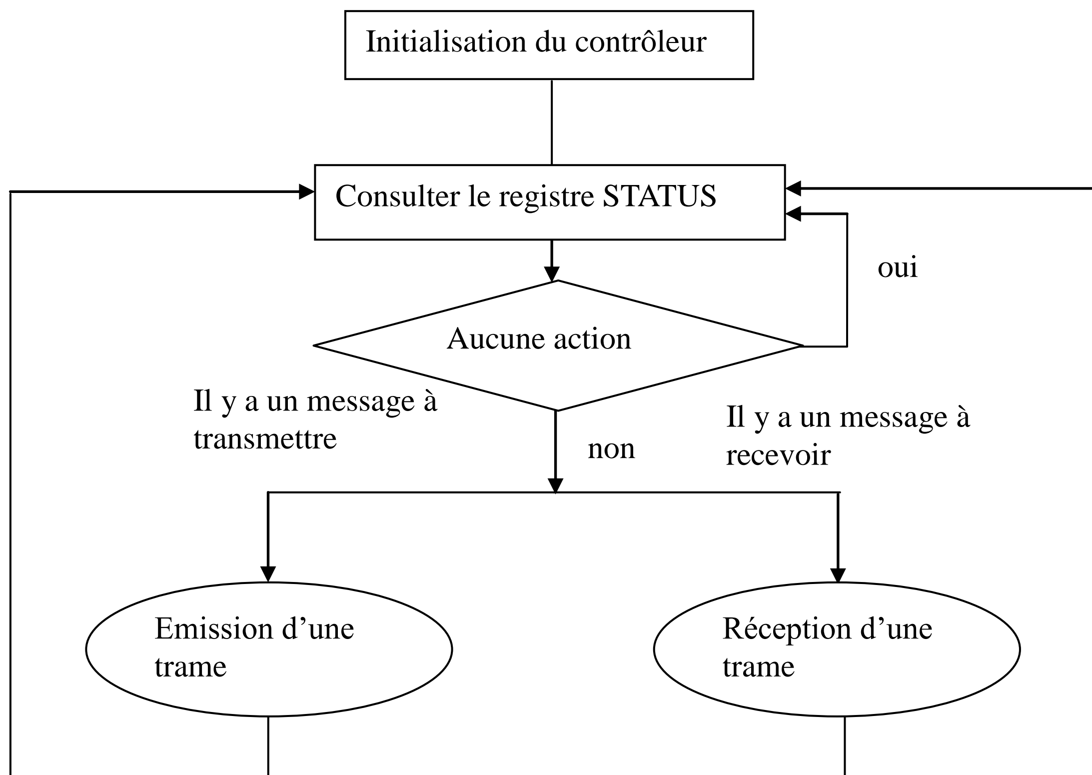


Figure 2.9 : Flot de communication en mode polling

II.5.2.2. Mode interruptible

Pour pouvoir travailler en ce mode [8], les différentes interruptions doivent être activées. Le microcontrôleur se met en mode récepteur ou émetteur selon le type d'interruption reçu. L'acheminement d'une communication en mode interruptible est donné par l'organigramme suivant :

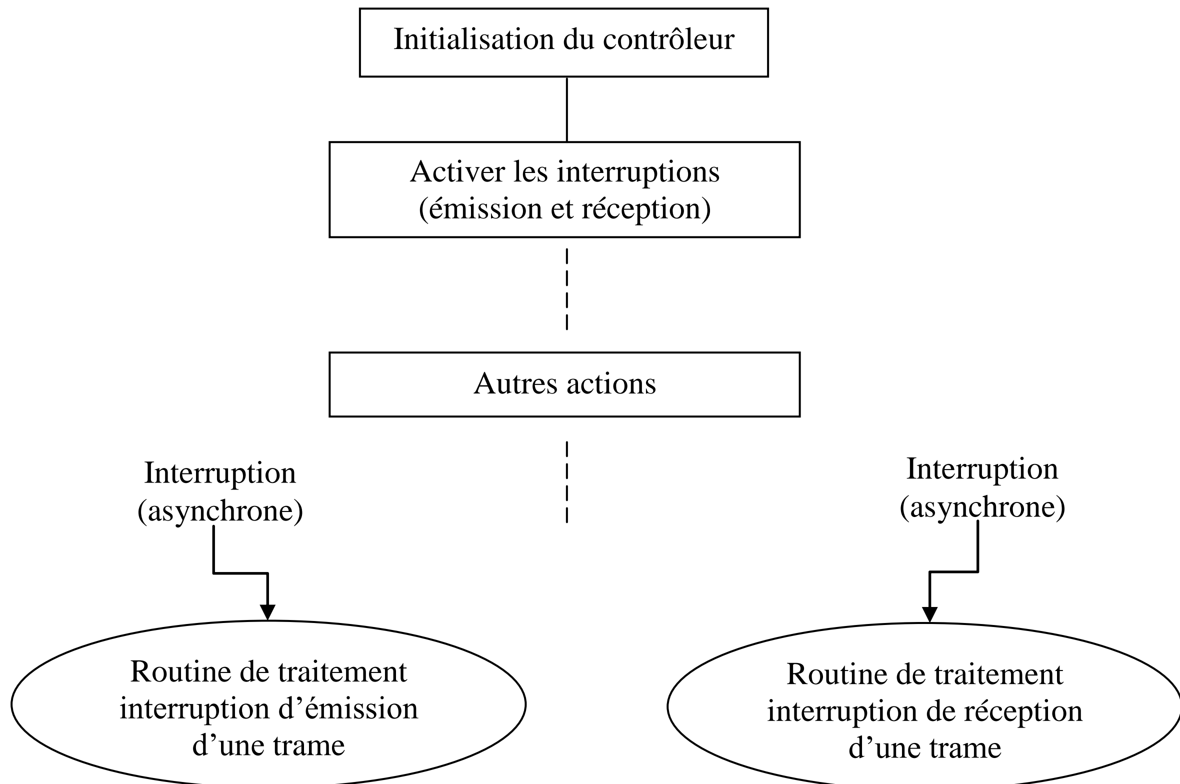


Figure 2.10 : Flot de communication en mode interruptible.

II.5.2.3. Initialisation du SJA1000

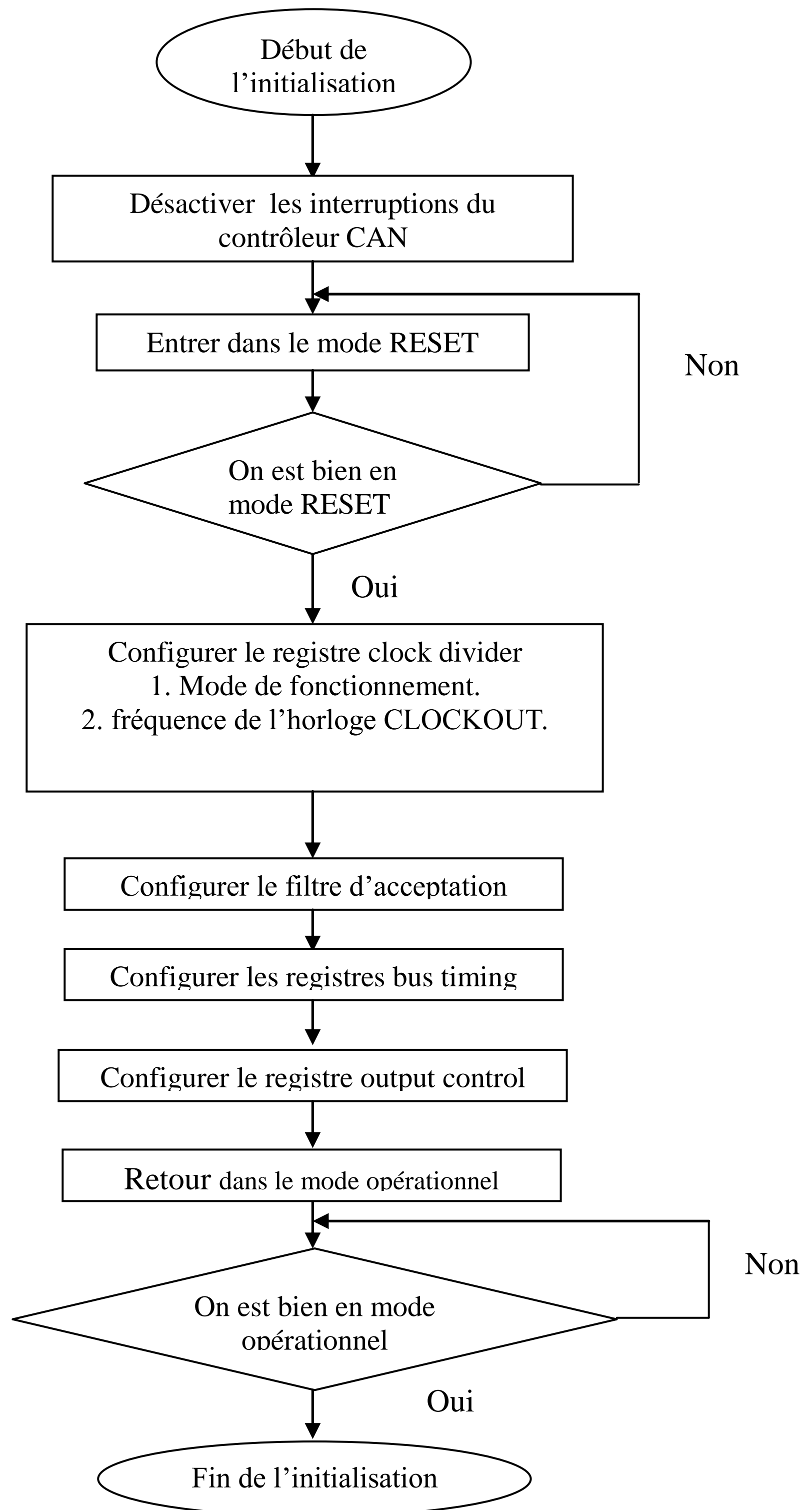


Figure 2.11: Initialisation du contrôleur

Lors de l'initialisation du SJA1000, on fait une demande pour passer en mode RESET afin de pouvoir configurer les registres suivants :

- Le registre mode afin de choisir :
  - Le mode de fonctionnement du filtre d'acceptation,
  - Le mode *listen only* ou mode normal,
  - Le mode self test ou mode normal.
- Le registre *clock divider* afin de choisir :
  - Le mode PeliCAN ou BasicCAN,
  - Désactiver ou activer et choisir la fréquence de l'horloge clockout.
- Les registres *acceptance code* et *acceptance mask* afin de définir :
  - Le code d'acceptation pour les messages qui vont être reçus,
  - Les bits qui doivent être contrôlés et ceux qui n'ont pas d'importance.
- Les registres BUS TIMING pour définir :
  - Le *Baud Rate Prescalar* (BRP),
  - La durée des différents segments,
  - Le nombre d'échantillons,
  - La *Synchronization Jump Width* (SJW).

II.5.2.4. Transmission d'une trame en mode polling

Pour transmettre une trame on commence par remplir le buffer de transmission, ensuite faire une demande de transmission « *transmission Request* » dans le registre de commande.

- Le buffer de transmission est occupé : on surveille le registre STATUS périodiquement jusqu'à ce que le buffer de transmission soit libre.
- Le buffer de transmission est libre : on écrit le message à transmettre dans ce dernier et on active le bit 1 du registre de commande ce qui aura pour conséquence de transmettre le message en question.

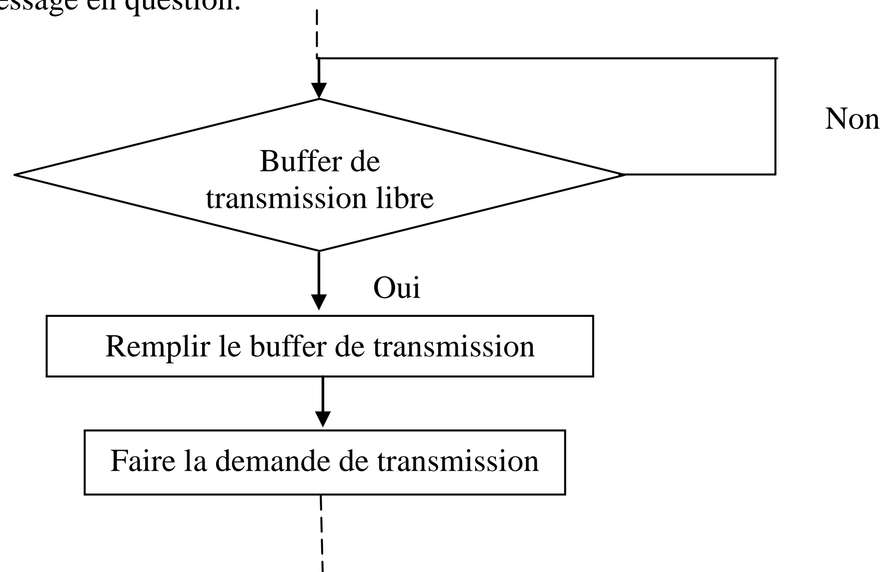


Figure 2.12: Transmission d'une trame

II.5.2.5. Réception d’une trame en mode polling

Les messages reçus sont placés dans le buffer de réception, une fois le message prêt le bit *Receive Buffer Status* du registre STATUS. Une réception d’interruption a lieu si on est en mode interruptible. Après avoir reçu le message on libère le buffer de transmission en activant le bit *Release Receive Buffer* du registre de commande.

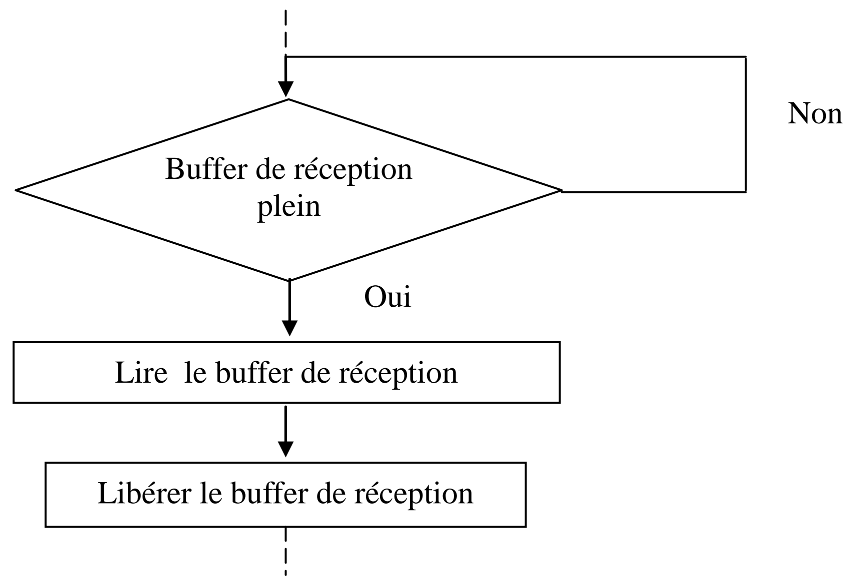


Figure 2.13:Réception d’une trame

II.6. Conclusion

Cette brève étude nous a permis de voir que le SJA1000 est un composant assez simple d’emploi. Elle va être utile dans la suite de notre projet qui a pour but la mise en œuvre d’un contrôleur CAN sur la carte DE2 d’ALTERA.

Sachant l’idée d’utilisation d’un SoC, deux voies sont possibles pour l’intégration d’un contrôleur CAN ; utilisation d’un circuit discret ou exploitation d’un IP Core dédié. Dans notre cas, on a choisi naturellement l’utilisation du *VHDL CAN Protocol Controller* [web3]. Afin de pouvoir utiliser ce dernier, il était prudent de procéder à la vérification de son bon fonctionnement en se basant sur les notions requises dans ce chapitre. On était donc amené à faire une étude globale de la description VHDL. Cette étude ainsi que l’implémentation de l’*IP core* vont faire l’objet de la dernière partie de notre projet.

Chapitre III

Implémentation d’un contrôleur CAN sur une carte DE2 d’Altera

III.1. Introduction

Le développement d'architectures à l'aide de FPGA nécessite l'emploi d'outils de développement. Un outil de développement comporte un logiciel de conception et une carte pour la vérification du fonctionnement de l'architecture conçue.

Dans une première partie de ce chapitre on va introduire les *System On Chip* en général tout en définissant les éléments utilisés dans leur développement. Ensuite on va détailler le cas de notre projet, où on va donner une description de la carte DE2 d'ALTERA, ainsi que le flot de conception. Après quoi, on décrira le fonctionnement de l'*IP core « VHDL CAN Protocol Controller* » et les résultats de sa mise en oeuvre.

III.2. Réalisation d'un système sur puce

III.2.1. Principe

Les *System On Chip* représentent des blocs fonctionnels regroupés sur une même puce pour donner un système complet. Au début, les SOC ont été créés pour le développement d'ASIC mais ont été étendus pour le développement de FPGA. C'est à ce stade que la notion de SOPC pour *System on Programmable Chip* a été introduite. Le SOC est figé et n'est donc pas réutilisable pour une autre application. Par contre, le SOPC est un composant reconfigurable pour diverses applications.

Deux types de fonctions qui constituent un SOPC peuvent être détectés : des fonctions standard ou des fonctions spécialisées.

III.2.2. Définition d'un IP core

Les éléments utilisés par le SOPC sont des éléments réutilisables de nature variée. Chaque bloc constituant un SOPC, appelé bloc IP, réalise une fonction spécifique, dont la complexité est variable. On retrouve des mémoires, des entrées/sorties, des processeurs sous forme d'IP core. [1]

III.2.3. Flot de conception d'un SOPC

Le développement d'un SOPC se fait en trois étapes principales: le développement de la partie hardware, le développement de la partie software, et puis la dernière étape consiste à implémenter la partie hardware sur le FPGA et exécuter la partie software au sein de l'architecture implémentée.

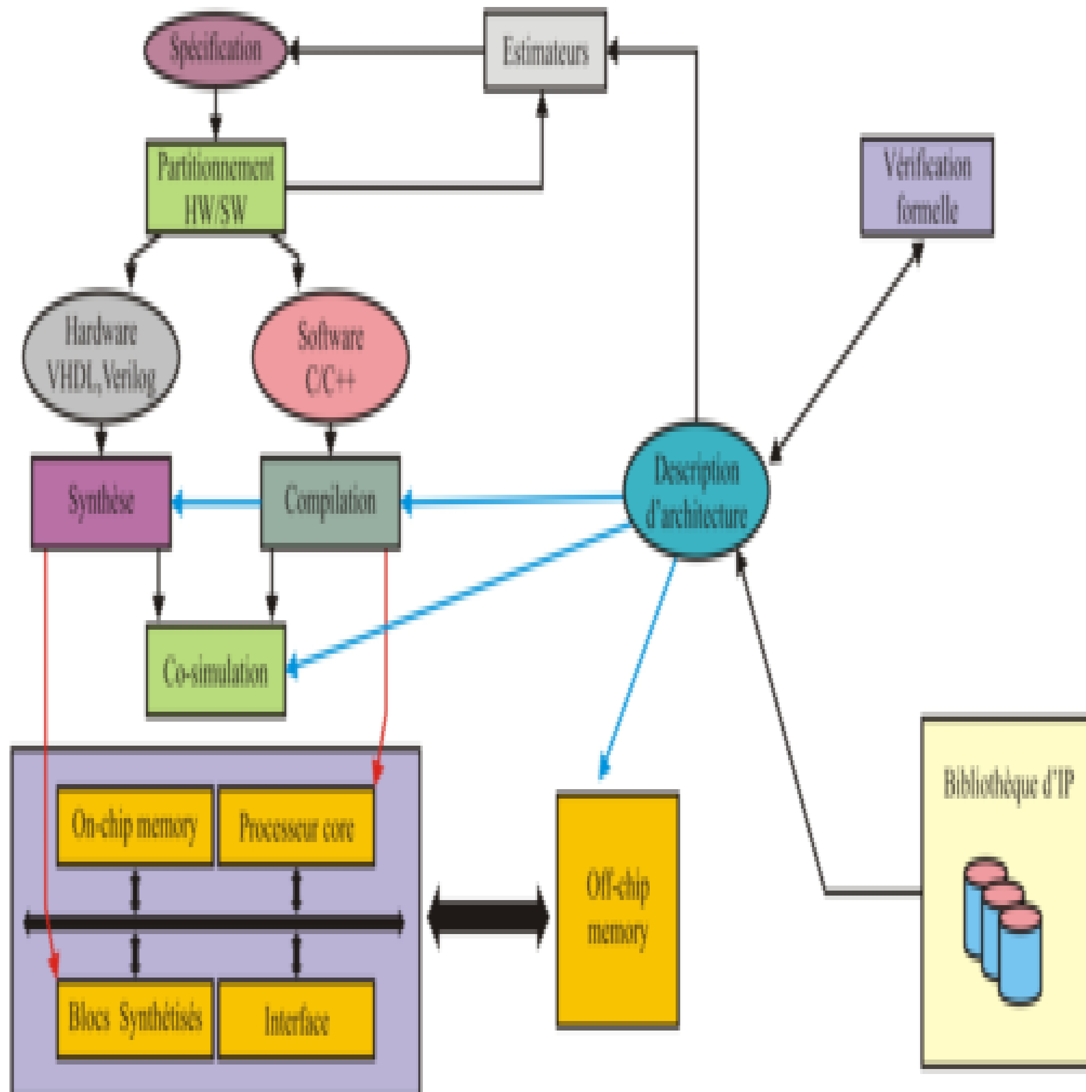


Figure 3.1 : Flot de conception d'un SOPC [16].

La configuration matérielle consiste en l'utilisation d'un ou plusieurs microprocesseurs, le choix des périphériques, et de la mémoire. On peut soit utiliser la mémoire du circuit FPGA ou bien une mémoire externe. L'utilisation des différents composants choisis sera décrite dans la partie logicielle en langage C/C++.

III.3. Carte DE2 et environnement de développement de SOPC ALTERA

III.3.1. Présentation de la carte

La carte sur laquelle se déroule le projet est une ALTERA *"DE2 Development and Education Board"*. Cette carte a été conçue afin de permettre l'apprentissage d'environnements embarqués, plus particulièrement au niveau des FPGAs. La carte utilise un gros FPGA le Cyclone II 2C35 (35.000 LE), comme cœur et dispose de mémoire (SRAM, SDRAM et Flash), de convertisseurs audio, vidéo et TV, ainsi que des interfaces Ethernet et USB ; elle est munie d'affi-

cheurs LED et LCD et d'un grand nombre de *switch* et de boutons poussoirs [3]. Cette carte est fournie avec un ensemble de logiciels de développement tel que Quartus et NIOS IDE.

La figure ci-dessous montre le schéma fonctionnel de la carte en question.

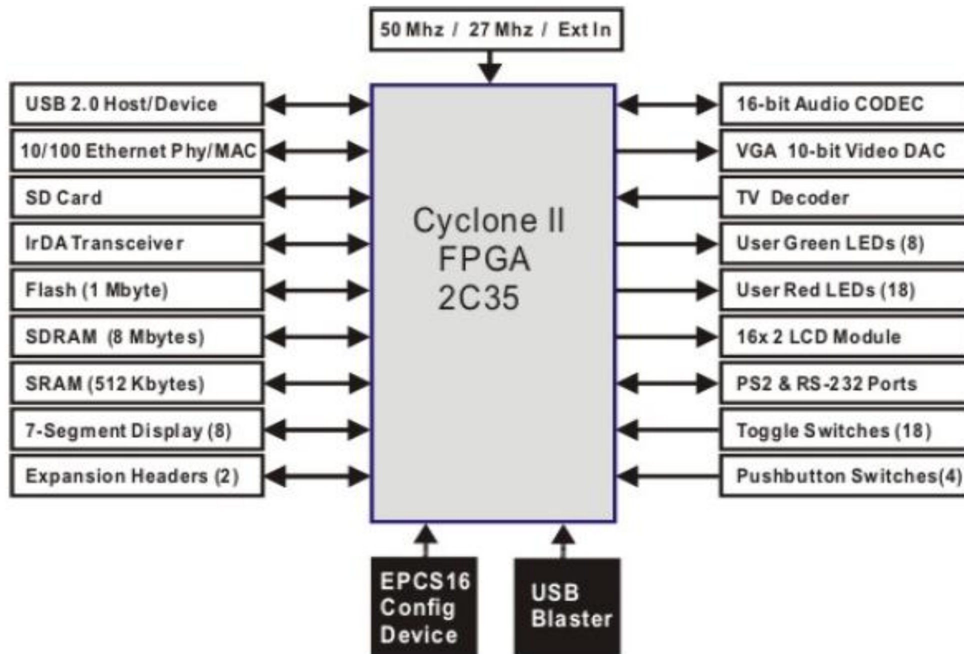


Figure 3.2: Schéma fonctionnel de la carte DE2 d'ALTERA [3].

III.3.2. l'architecture système

III.3.2.1. L'IP NIOS II

Le processeur *softcore* NIOS II est un cœur de processeur, pouvant être exploité sur tout FPGA de la marque ALTERA. Il peut s'associer à un grand nombre de périphériques pour former un système complet tels que :

- Mémoire.
- Timer.
- Liaison série UART.
- Interface écran LCD.
- E/S parallèles.
- Interfaces Ethernet.
- JTAG.
- Etc.

III.3.2.2. Les caractéristiques du NIOS II

C'est un processeur RISC (microprocesseur à jeu d'instruction réduit) avec une architecture interne de type HARVARD, et un bus de largeur de 32 bits. Ses performances sont de 30 à 80 MIPS.

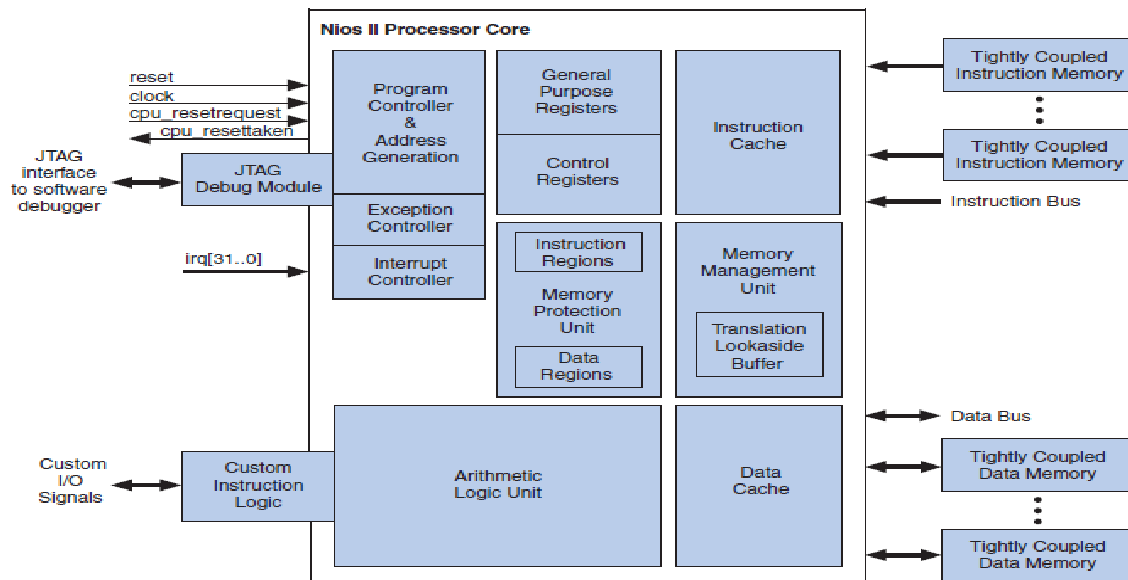


Figure 3.3: Architecture d'un processeur NIOS II [11].

Lors de la configuration du processeur NIOS II, il est possible de choisir entre trois versions : une première version *Economy* qui utilise moins de surface de silicium du composant FPGA, elle occupe 540 LEs pour le cyclone II, une deuxième version *Standard* qui permet un compromis entre surface et rapidité, cette version occupe 1030 LEs et enfin une dernière version *Fast* qui est la plus rapide des deux autres versions et occupe 1600 LEs. Ces trois versions permettent à l'utilisateur de choisir entre performance et espace, c'est-à-dire en termes de *Logic Elements* occupés par NIOS.

III.3.3. Le bus Avalon

III.3.3.1. Présentation

Le bus Avalon a été conçu par ALTERA afin de connecter le processeur NIOS aux différents périphériques lors du développement des SOPC.

Avalon est une interface qui indique les connexions de ports entre les composants maîtres et esclaves, et indique la synchronisation par laquelle ces composants communiquent.

Les objectifs principaux de la conception du bus Avalon sont :

- Un protocole simple de communication.
- Utilisation optimisée de ressource : Conserver les éléments logiques (LEs) à l'intérieur du circuit logique programmable.

Le bus Avalon transfère un octet, un demi-mot, ou un mot (8, 16, ou 32 bits) entre un périphérique maître et un périphérique esclave. Après qu'un transfert se soit accompli, le bus est sur le prochain coup d'horloge disponible pour un nouveau transfert.

III.3.3.2. Caractéristiques

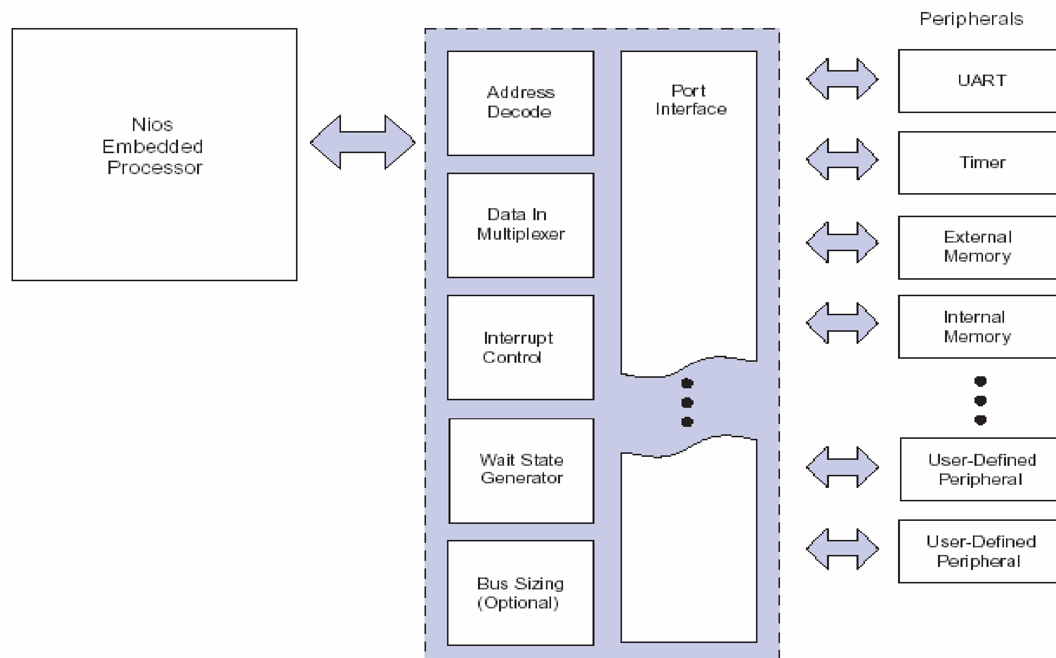


Figure 3.4: Architecture du bus Avalaon [13]

Le bus Avalon est décomposé dans la figure ci dessus. Il comprend un contrôleur pour la gestion des interruptions, un décodeur d'adresses, un multiplexeur de donnée et générateur de cycle d'attente (*Wait States*). Les différents signaux gérés par le bus Avalon sont résumés dans le tableau suivant :

Type de signal	largeur	direction	commentaires
clk	1	Entrée	Les transferts sur le bus sont synchrones à cette horloge
Reset	1	Entrée	Reset du système
chipselect	1	Entrée	Sélectionne le périphérique pour le transfert
adress	1-32	Entrée	Signal d'adresse du bus
byteenable	0,2, 4	Entrée	Valide certaines lignes de données durant le transfert
read	1	Entrée	Signal de lecture
readdata	1-32	Sortie	Signal des données à lire depuis le périphérique
write	1	Entrée	Signal d'écriture
writedata	1-32	Entrée	Signal des données à écrire sur le périphérique
waitrequest	1	Sortie	Fait patienter le bus Avalon
readyfordata	1	Sortie	Le périphérique est prêt à recevoir des données
dataavailable	1	Sortie	Le périphérique est prêt à envoyer des données
irq	1	Sortie	Demande une interruption
resetrequest	1	Sortie	Le périphérique demande de réinitialiser tout le système

Tableau 3.1 : Signaux entre le bus Avalon et un périphérique esclave [6].

III.3.4. Environnement de développement

La conception d'un système basé sur un microprocesseur NIOS II s'articule sur l'utilisation de plusieurs logiciels : le Quartus II contenant l'outil *SOPC Builder* et le NIOS IDE. Le flot de conception de la figure suivante regroupe les différents outils utilisés pour développer un SOPC dans un environnement ALTERA ainsi que l'acheminement de toutes les étapes.

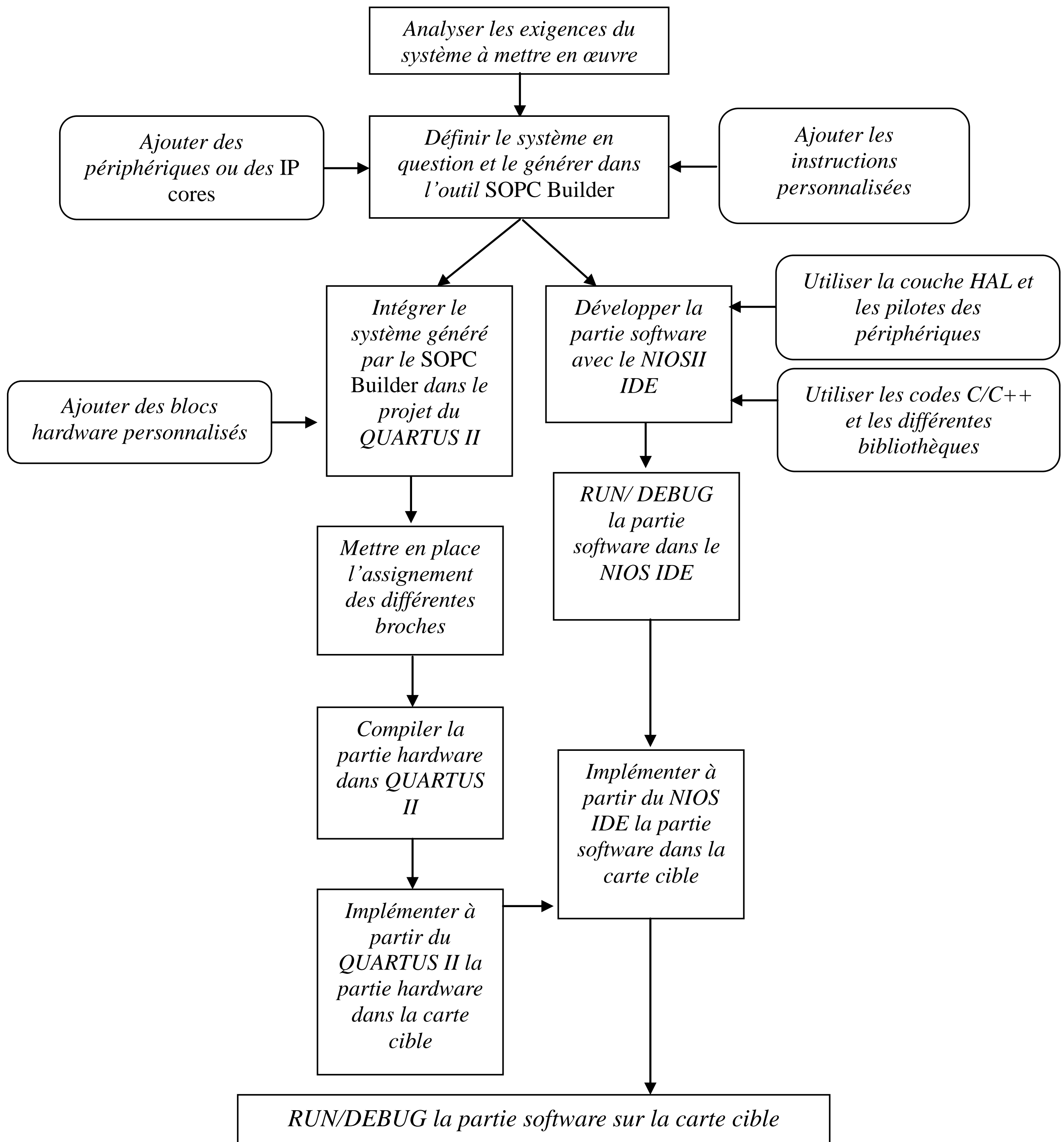


Figure 3.5: Flot de conception d'un SOPC sur ALTERA.

III.3.4.1. Configuration matérielle

Environnement de développement Quartus

Quartus II est un logiciel utilisé pour l'analyse et la synthèse de descriptions HDL (*Hard Description Language*) de différents modules. Il permet grâce au *SOPC Builder*, de choisir tout les composants qui feront partie du système désiré. Il permet aussi de mettre en place l'assignement des différents composants selon la carte cible. Ce même logiciel contient l'outil *programmer* qui est utilisé pour l'implémentation de la partie *hardware* sur le FPGA.
Il existe plusieurs modes de saisie utilisés au sein du Quartus, ceux auxquels on s'est intéressé sont les suivants:

- Mode de saisie graphique par association de symboles, les fichiers associés à ce mode sont les fichiers : .BDF (*Block/Schematic File*).
- Mode textuel, on parle de description AHDL, VHDL et Verilog, les fichiers associés à ce mode sont les fichiers : .VHD et .V

Le SOPC Builder

Le SOPC Builder est un outil du logiciel Quartus, il permet à l'utilisateur de définir les différents éléments qui composent le système que l'on souhaite réaliser. Il permet aussi à l'utilisateur de créer ses propres composants et de les ajouter à la bibliothèque des IP core.
Une fois le système défini sur le *SOPC Builder*, ce dernier va générer un fichier avec l'extension PTF (*Peripheral Template File*). Lorsqu'un périphérique est rajouté au système, le contenu du fichier PTF correspondant est quant à lui rajouté au fichier PTF décrivant le système complet. Le fichier PTF généré correspond à une image du système et de ses paramètres au format texte.

La fenêtre du *SOPC Builder* est divisée en deux parties. La partie gauche représente la bibliothèque du *SOPC Builder* contenant les composants qui sont utilisables. La partie droite liste tous les éléments qui constituent le système en cours de développement ainsi que le *mapping* mémoire et les interruptions associées à ces périphériques.
En plus du fichier PTF le SOPC Builder génère d'autres fichiers nécessaires à la suite de développement, on peut trouver :

- Le fichier de conception (.sopc) qui englobe le contenu hardware du système *SOPC Builder* ;
- Le fichier d'information (.sopcinfo) : il contient une description compréhensible par l'utilisateur du fichier (.sopc).
- Les fichiers (HDL) qui décrivent le système *SOPC Builder*.

Le Quartus utilise ces fichiers pour compiler la conception globale et la mettre sous forme d'un fichier (.sof).
C'est ce fichier (.sof) qui sera implémenté dans la carte cible. Après la configuration, le FPGA se comportera exactement comme décrit dans la conception matérielle.

III.3.4.2. Configuration logicielle

Après avoir générer le système à l'aide du *SOPC Builder* on peut commencer à développer l'application C/C++ avec le logiciel NIOS IDE qui comprend un compilateur C/C++, un débogueur et un éditeur. Ce même logiciel nous permet de télécharger notre application et de l'exécuter sur le FPGA configuré selon l'architecture matérielle.

III.3.5. L'IP core CAN

III.3.5.1. Fonctionnement de l'IP core du contrôleur CAN

La figure ci-dessous représente le schéma fonctionnel de *l'IP core « VHDL CAN protocole controller ».*

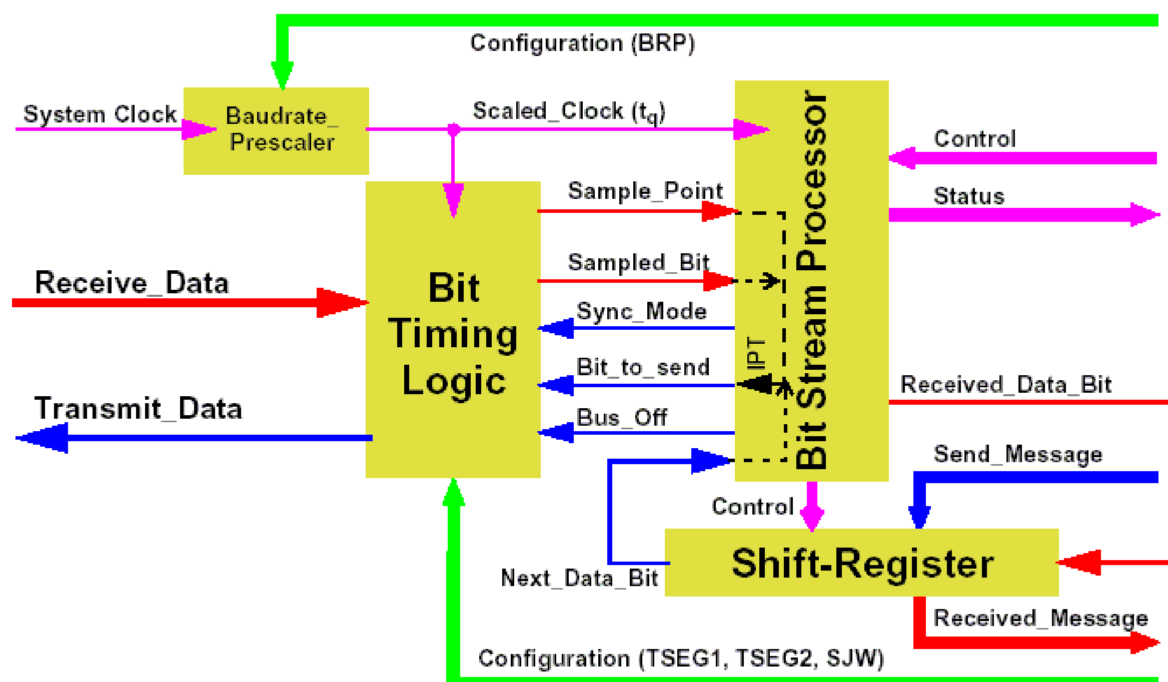


Figure 3.6: Schéma fonctionnel de l'IP core [web3].

La description VHDL de l'*IP core* se compose de plusieurs entités, chacune effectuant une fonction bien précise, avec l'existence d'une entité principale. Cette dernière va regrouper toutes les autres entités et les associer, elle représente le fonctionnement global du contrôleur. On va détailler les fonctions réalisées par chaque bloc de la description VHDL:

CAN_ACF

Cette première entité décrit la gestion du filtre d'acceptation, le fonctionnement de cette dernière se résume dans la comparaison entre ses différents signaux d'entrée. D'un coté les registres contenant l'identificateur ainsi que les deux premiers octets du champ de donnée et de l'autre coté les registres ACR et AMR. Cette comparaison se fait selon le mode de fonctionnement du filtre ainsi que le type de trame, ces derniers sont déterminés par le reste des signaux d'entrée. En revanche cette entité ne retourne qu'un seul signal de sortie qui vaut 1 pour indiquer que le message a été accepté ou 0 pour le refus de ce dernier.

CAN_FIFO

Comme son nom l'indique cette entité gère le fonctionnement de la FIFO qui représente le canal de transmission. On aura à la sortie de ce bloc la donnée lue pendant un cycle de lecture dans le cas ou le signal *fifo_selected* est actif, ainsi que d'autre information grâce aux autres signaux de sortie : *overload*, *info_empty* …etc.

CAN_CRC

Comme on l'avait vu précédemment afin de vérifier l'intégrité des données envoyées le contrôleur procède au calcul du *Cyclic Redundancy Code* sur un vecteur de 15 bits. Ce calcul est effectué par cette partie de la description.

CAN_IBO

Elle représente un registre à décalage qui reçoit les données du contrôleur et les envoie sur le bus et vice versa.

CAN_REGISTER

Cette entité ainsi que d'autres *CAN_REGISTER_ASYN, CAN_REGISTER_ASYN_SYN* sont mises en point pour la gestion des différents registres du contrôleur, le CAN_REGISTER détermine la longueur des différents registre comme étant de 8 bits. Il recopie les données en entrée dans le bus de donnée de sortie si le signal *Write Enable* est actif.

CAN_BTL

Le bloc *Bit Timing Logic* gère la durée du bit pour les données reçues ainsi que celles à envoyer. Alors comme sorties de cette entité on a le point d'échantillonnage, le bit échantillonné ainsi que les signaux de synchronisation.

CAN_BSP

Le *Bit Stream Processor* gère le flux de données et s'occupe de la mise en trame des données, pour cela cette entité fait appel à d'autres entités : la CAN_ACF, la CAN_FIFO, la CAN_CRC, et la CAN_IBO.

CAN_REGISTERS

Cette entité regroupe les différentes entités traitant le fonctionnement des registres internes du contrôleur telle que : CAN_REGISTER, CAN_REGISTER_ASYN, et CAN_REGISTER_ASYN_SYN.

CAN_TOP

Cette entité est l'entité englobant toutes les autres, CAN_BSP, CAN_BTL, et CAN_REGISTERS. Ses signaux d'entrée et de sortie sont ceux qui vont être utilisés pour la configuration de l'IP core.

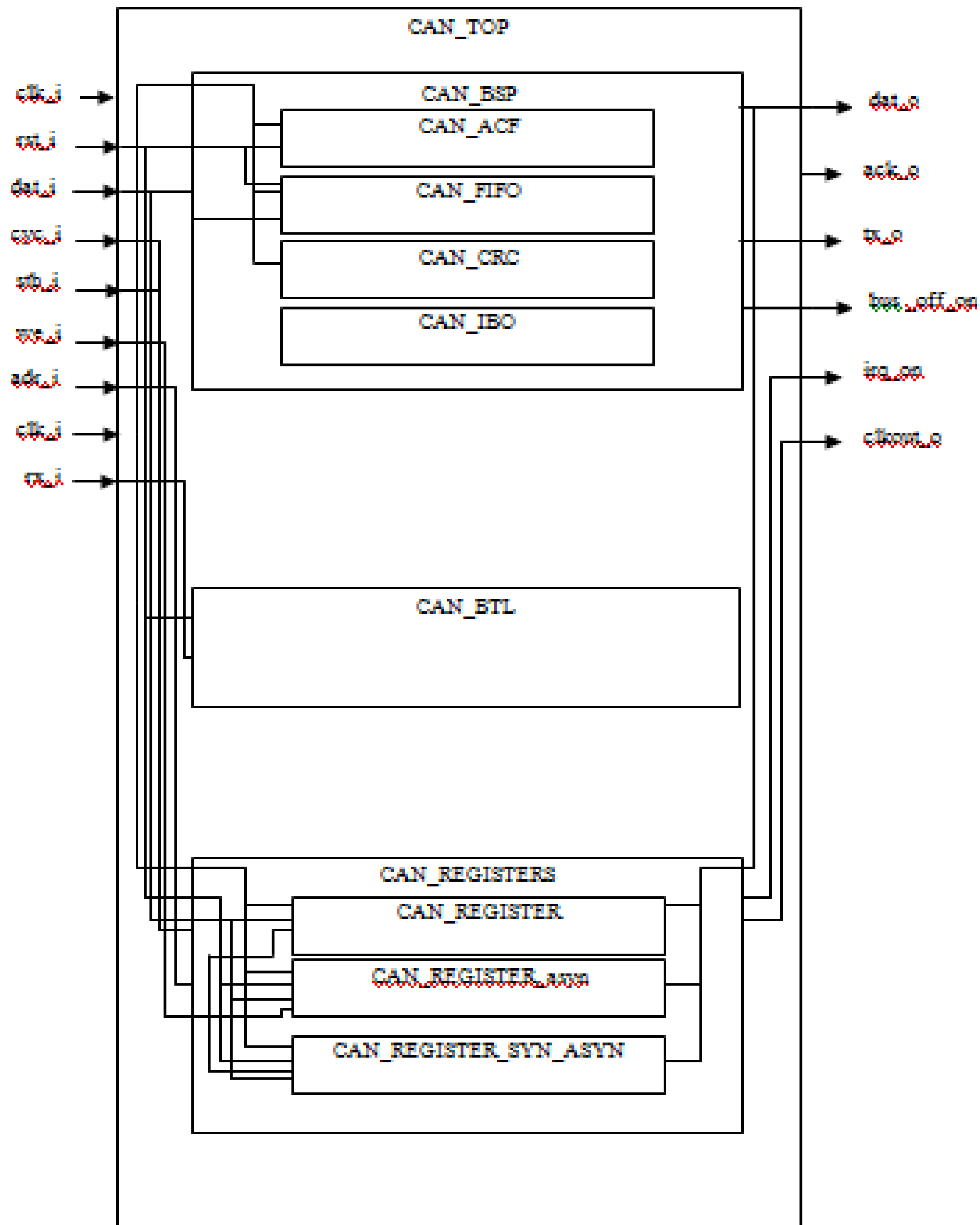


Figure 3.7 : Schéma bloc de l'IP core

III. 3.5.2. Description des signaux

Les signaux d'entrée

wb_clk_i : il représente le signal d'horloge provenant du microcontrôleur.

wb_rst_i : il représente le signal Reset qui permet au contrôleur de remettre à zéro les différents signaux, il est actif à l'état 1.
wb_we_i: le signal *Write Enable* détermine si le circuit est en mode écriture ou lecture. Pour effectuer une écriture il suffit de mettre ce signal à l'état 1, et à 0 pour le mode lecture.
wb_cyc_i : Ce signal détermine le début d'un cycle d'écriture ou de lecture (suivant la valeur de we). Il est actif à l'état 1, alors pour entamer une procédure de lecture ou d'écriture il doit etre à l'état 1.
wb_stb_i : C'est le signal Strobe, il valide le lancement de l'écriture ou de la lecture (suivant la valeur de we). Ce signal est actif à l'état 1. Le passage à l'état haut de ce signal doit se faire quant wb_cyc_i est aussi actif.
wb_data_i : c'est un vecteur de 8bits qui représente le vecteur de Données entrantes.
Il s'agit du vecteur de Données sortantes. Il est composé de 8 bits.
wb_adr_i : c'est le bus d'Adresse. Il détermine le registre auquel on s'intéresse. Il est codé sur 8 bits.
Clk_i : l'horloge de fonctionnement du composant

Les signaux de sortie

bus_off_on : C'est un signal actif à l'état bas, il indique que le contrôleur est déconnecté du bus à cause du nombre des erreurs détectées par ce dernier.
wb_ack_o : Il s'agit du signal Acknowledge, il a pour rôle de signaler quand la lecture / écriture sur le bus wb_data_i (wb_data_o) est terminée.
wb_data_o : c'est un vecteur de 8 bits qui représente le vecteur de données sortantes.
Irq_on : il représente le signal d'interruption, il est actif à l'état bas.
Clkout_o : le signal d'horloge de sortie, la fréquence de cette horloge dépend de la fréquence du clk_i et peut être contrôlé par l'un des registres du composant.

III.4. Mise en œuvre

III.4.1. Simulation de l'IP core

Avant d'implémenter notre *IP core* sur la carte il était prudent de simuler le comportement du code VHDL, pour vérifier son bon fonctionnement. Le Quartus II dispose d'un outil de simulation où on pourra effectuer la simulation du contrôleur. Le simulateur propose une fenêtre « *objects* » qui contient les signaux d'entrée et de sortie nécessaires pour la simulation. Il suffit donc d'ajouter ses signaux à la fenêtre « *wave* » dans laquelle on pourra imposer des vecteurs de test aux signaux d'entrée, et de voir la réponse à ces vecteurs sur les signaux de sortie sous forme de chronogramme.

III.4.1.1. simulation de l'étape d'écriture dans un registre du contrôleur

Afin de pouvoir vérifier la procédure d'écriture dans les registres on a choisi le registre *clock divider* qui se trouve à l'adresse 0x1F et dont les trois premiers bits déterminent la fréquence de l'horloge clkout, dans notre cas on s'est intéressé au quatrième bit qui une fois mis à 1 désactive l'horloge clkout, on peut voir ce résultat dans la figure ci-dessous.

Figure 3.8: Simulation de l'étape d'écriture dans un registre du contrôleur

Pour réaliser cette simulation il suffisait de mettre le signal wb_we_i à 1 on indique ainsi que c'est un cycle d'écriture on a mis ensuite le signal wb_cyc_i à1 de même pour le signal wb_stb_i, l'adresse du registre en question doit être introduite dans le vecteur wb_adr_i, le vecteur des données entrantes wb_dat_i va porter la valeur 0x08. C'est cette dernière une fois écrite dans le registre « clock driver » qui va désactiver l'horloge clockout. Comme on le voit sur la figure, à la fin du cycle d'écriture, ceci est indiqué par le signal wb_ack_o, l'horloge clkout_o est désactivée.
Il faut bien signaler que le registre choisi n'est modifiable qu'en mode reset, le passage à ce mode apparait dans la figure, il se fait en mettant la donnée 0x01 dans le registre qui se trouve à l'adresse 0x00 ce qui correspond à la mise à 1 du bit0 « *reset request* » du registre de contrôle.

III.4.1.2. Simulation de l'étape de lecture à partir d'un registre du contrôleur

On va lire à partir du registre STATUS qui se trouve à l'adresse 0x02. Dans cette phase il suffit de mettre le signal wb_we_i à 0 et les signaux wb_cyc_i et wb_stb_i à 1, l'adresse du registre STATUS doit être introduite dans le bus wb_adr_i. Comme on le voit sur la figure ci-dessous une fois que le signal wb_ack_o passe à 1 on pourra lire la donnée sur le bus wb_data_o.

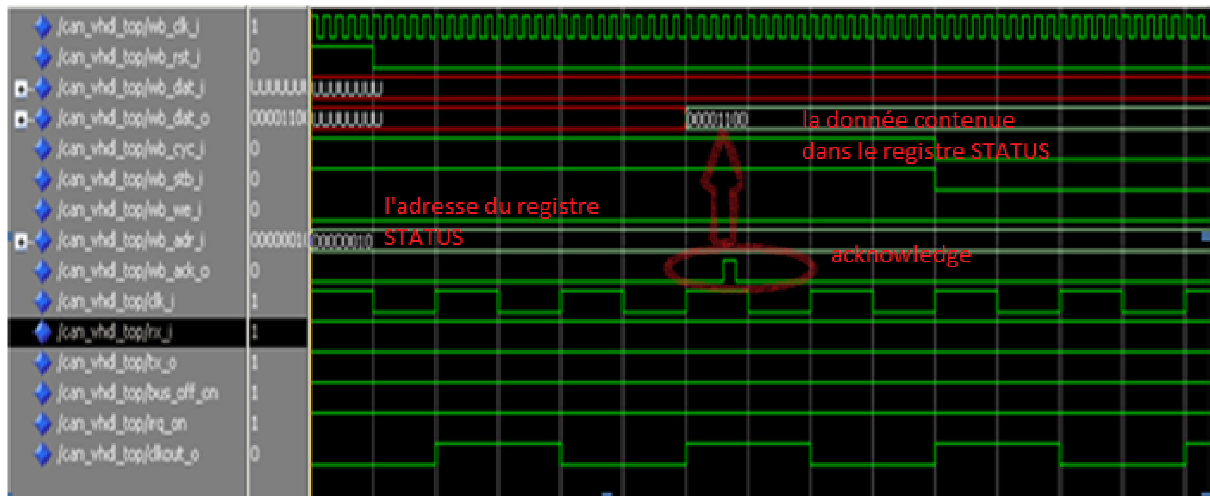


Figure 3.9: Simulation de l'étape lecture à partir d'un registre du contrôleur.

III.4.1.3. Simulation de l'émission d'une trame à partir du contrôleur

Initialisation du contrôleur

Cette partie a pour but de configurer les différents registres du contrôleur afin qu'il réalise le fonctionnement souhaité, c'est dans cette étape que se fait le choix du mode de fonctionnement PeliCAN ou BasicCAN, la mise en point de filtre d'acceptation avec ses différents registres, le choix de la longueur du bit ainsi que des paramètres de synchronisation. La plupart de ces registres ne sont configurables qu'en mode RESET alors pour commencer on passe dans ce mode et une fois l'initialisation terminée on revient dans le mode opérationnel.
La figure ci-dessous montre la simulation de la partie initialisation du contrôleur.

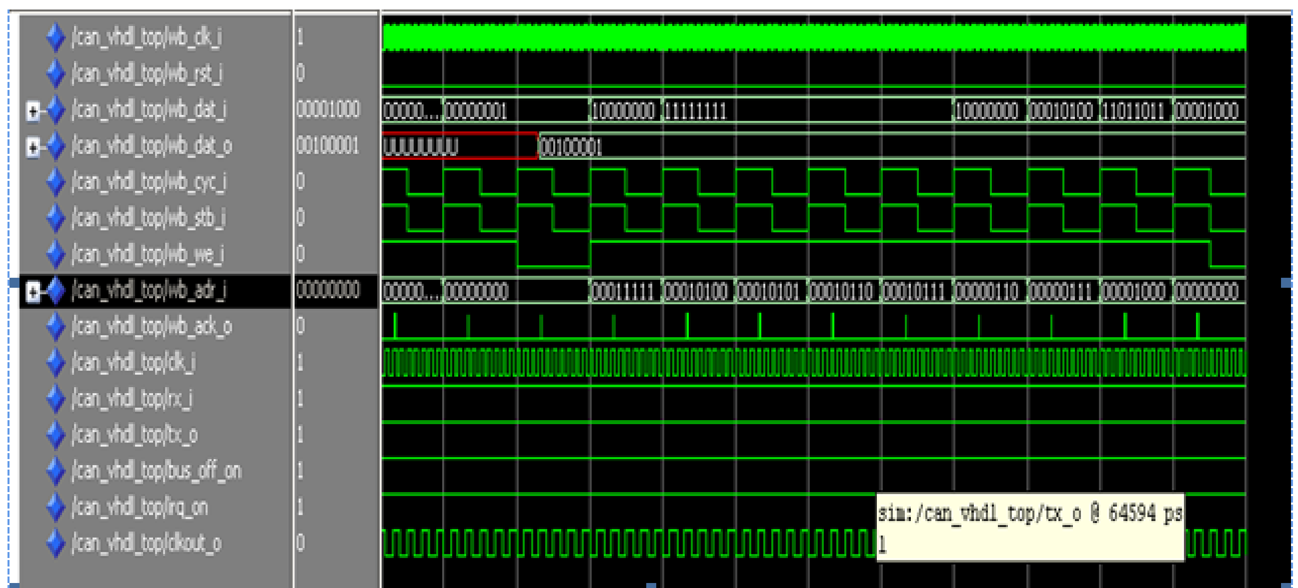


Figure 3.10: Simulation de la partie initialisation du contrôleur.

Pour effectuer cette simulation il a suffit d'envoyer la bonne donnée à la bonne adresse, en effectuant des cycles d'écriture pour la configuration des registres et des cycles de lecture pour la vérification de leur état.

Transmission de la trame

Après avoir initialisé le contrôleur on pourrait transmettre une trame si on le souhaite, il suffit de vérifier l'état du contrôleur à partir du registre d'état pour voir si le buffer de transmission est libre. Si c'est bien le cas on remplie ce dernier par la donnée à envoyer ainsi que son identificateur, et pour terminer on fait une demande de transmission en mettant le bit0 « *Transmission Request* » du registre COMMAND à 1.

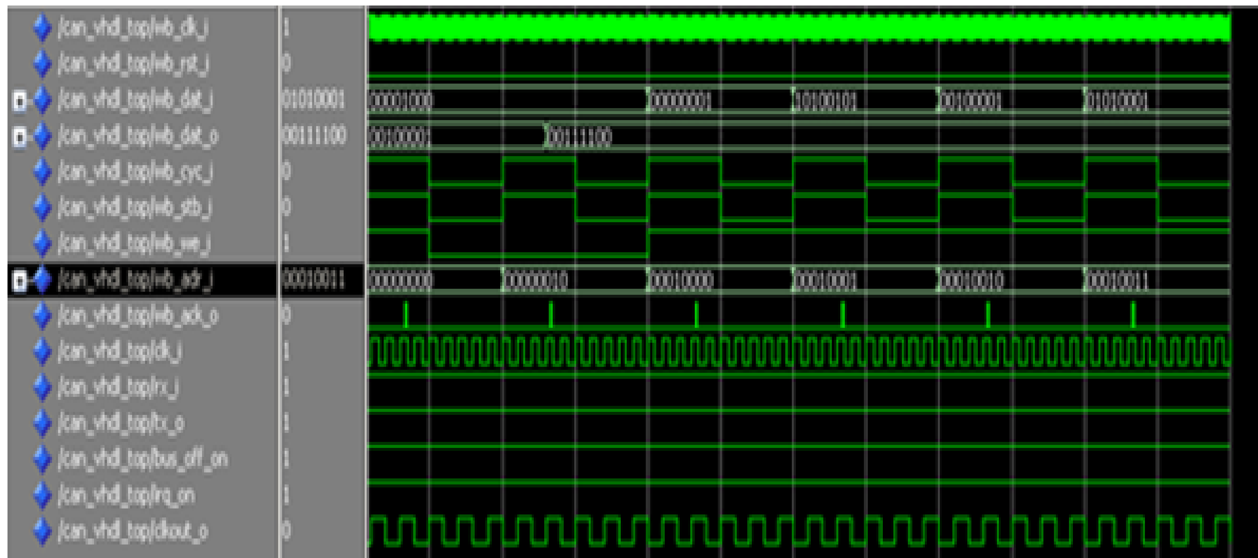


Figure 3.11: Remplissage du buffer de transmission.

Lors de la simulation de cette partie on a rencontré quelques problèmes car comme on l'a vu précédemment le nœud émetteur procède à une vérification du bus pour l'arbitrage et pour vérifier s'il n'y a pas lieu d'une erreur de type « *bit error* » c'était la cause pour laquelle on recevait une trame d'erreur au lieu de la trame de donnée. Afin de remédier à ce problème on a fait des modifications dans la description de l'*IP core* qui avaient pour but de désactiver le *bit error*. Après ces modification la simulation donnait bien la trame de donnée mais on ne recevait que le début de la trame : le SOF, l'identificateur, le bit RTR et le champ de commande. La trame qui devait être reçue est la suivante :
SOF Identificateur RTR r0 r1 DLC3 DLC2 *stuff* DLC1 DLC0 data1

0 10100101001 0 0 0 0 0 1 0 1 01010001

Sur la figure montrant la simulation on voit bien que la trame envoyée n'est pas complète et que le contrôleur la renvoie, vu le mécanisme de recouvrement des erreurs.

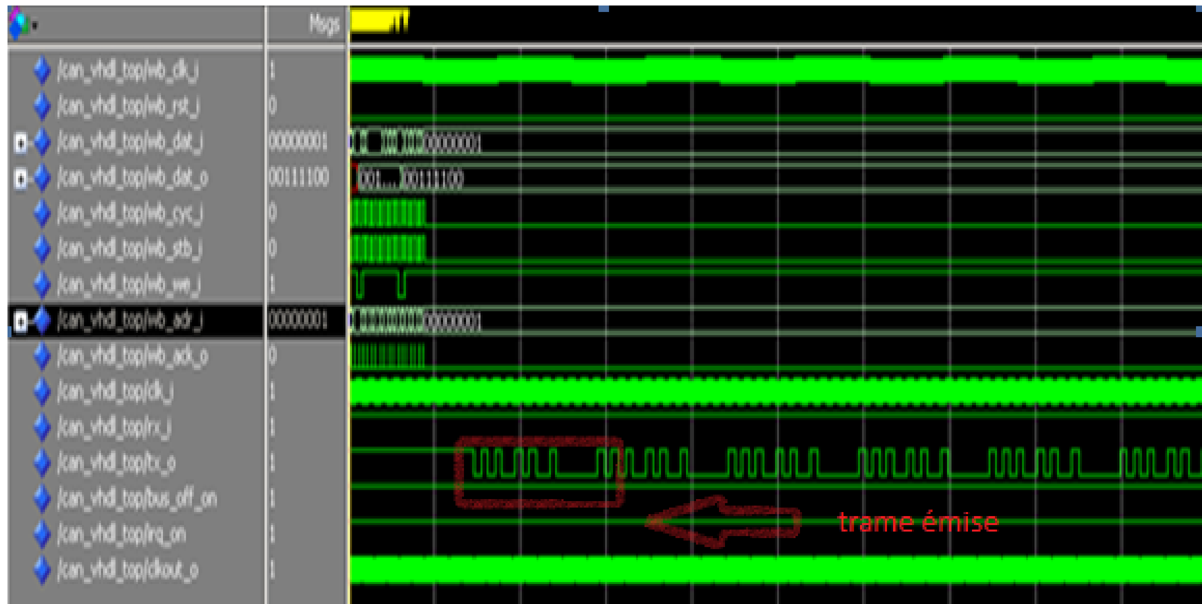


Figure 3.12: Trame envoyée sur la ligne de transmission.

Tenant compte de ces résultats obtenus on réalise que la partie concernant l'émission de trame de l'*IP core* ne fonctionne pas bien. Afin de remédier à ce problème il fallait faire une étude détaillée de la description VHDL ce qui permettra d'apporter les modifications nécessaires.

Cependant, et pour le critère de temps, on a préféré prendre l'IP core tel qu'il est, et l'implémenter sur la carte.

III.4.2. Implémentation de l'IP core

Pour l'implémentation de notre *IP core* sur la carte DE2 d'ALTERA nous avions deux manières de faire, la première est de l'ajouter à la bibliothèque des *IP core* reconnus par le *SOPC Builder* en définissant une interface Avalon slave et le connecter au NIOS II à travers le bus Avalon. La deuxième manière est de connecter le composant créé grâce au logiciel Quartus II au NIOS II via les PIO (*Parallel Input Output*). Dans cette partie on va exposer les deux manières de faire mais pour notre cas et selon le critère du temps consacré pour le projet on a préféré procéder par la deuxième méthode.

III.4.2.1. Utilisation du bus avalon

Pour l'interfaçage de l'*IP core* on a besoin d'un nombre de signaux pour se connecter au bus avalon. Le *SOPC Builder* nous donne la possibilité de définir l'interface Avalon slave selon les différents signaux d'entrée et de sortie du contrôleur. Le tableau suivant résume les interfaces utilisées pour chaque signal.

Nom du signal	interface	Type du signal	Longueur	Direction
wb_clk_i	clock	clk	1	In
wb_rst_i	reset_sink	reset	1	In
wb_data_i	avalon_slave_0	writedata	8	In
wb_data_o	avalon_slave_0	readdata	8	Out
wb_cyc_i	avalon_slave_0	begintransfer	1	In
wb_stb_i	avalon_slave_0	chipselect	1	In
wb_we_i	avalon_slave_0	write	1	In
wb_adr_i	avalon_slave_0	address	8	In
wb_ack_o	conduit_end	export	1	Out
clk_i	conduit_end	export	1	In
rx_i	conduit_end	export	1	In
tx_o	conduit_end	export	1	Out
bus_off_on	conduit_end	export	1	Out
irq_on	interrupt_sender	irq	1	Out
clkout_o	conduit_end	export	1	Out

Tableau 3.2: signaux entre le bus avalon et l'IP core.

Les signaux qui sont définis de type « export » sont les signaux qu'on veut recevoir ou ceux qu'on veut introduire nous même. Par exemple la ligne tx_o sur laquelle on souhaite voir la trame émise par le contrôleur. Les autres signaux ont pour interface l'Avalon slave et permettent d'écrire et de lire à partir des registres du contrôleur. Les deux figures suivantes montrent un cycle d'écriture et un cycle de lecture.

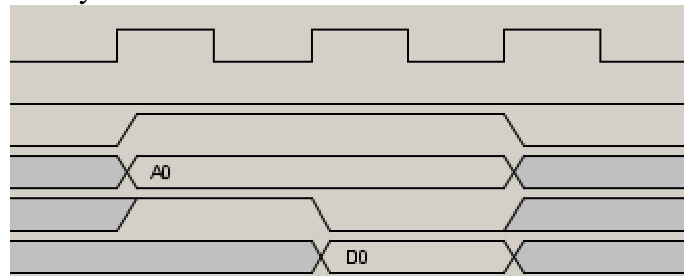


Figure 3.13: Cycle de lecture à partir d'un registre du contrôleur.

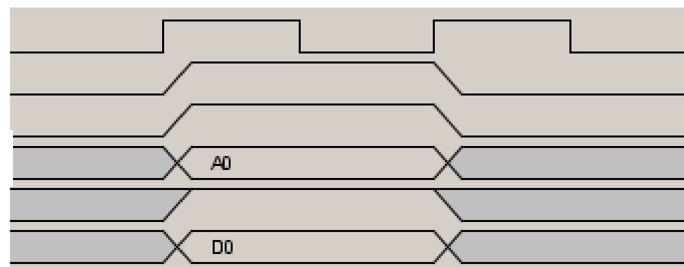


Figure 3.14: Cycle d'écriture dans un registre du contrôleur.

Une fois les signaux définis on peut ajouter l'*IP core* à la bibliothèque du *SOPC Builder* comme on le voit sur la figure ci-dessous :

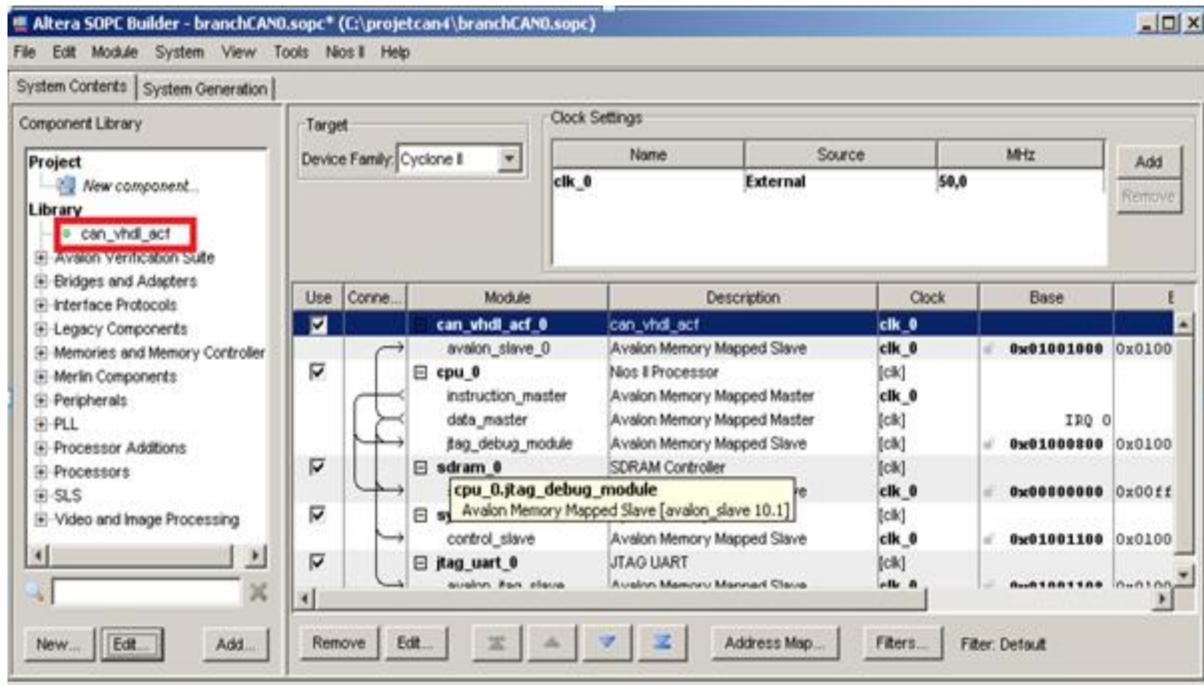


Figure 3.15: Connexion de l'IP core au NIOS II à travers le bus avalon.

L'architecture qui sera implémentée sur la carte est donnée en annexe n°2, cette architecture consomme 124 éléments logiques d'un ensemble de 35000 ce qui est équivalent à moins de 1% des ressources du FPGA.

Après avoir implémenté cette dernière il suffira pour la suite de passer à la partie soft qui permet de piloter le contrôleur à partir du NIOS II. 124 (moins de 1%)

III.4.2.2. Utilisation des PIO

Développement matériel

Le contrôleur CAN a été créé à partir du logiciel Quartus II qui contient la description VHDL du contrôleur et nous permet de créer le symbole correspondant. Il utilise 124 éléments logiques. La figure du composant en question est donnée ci-dessous :

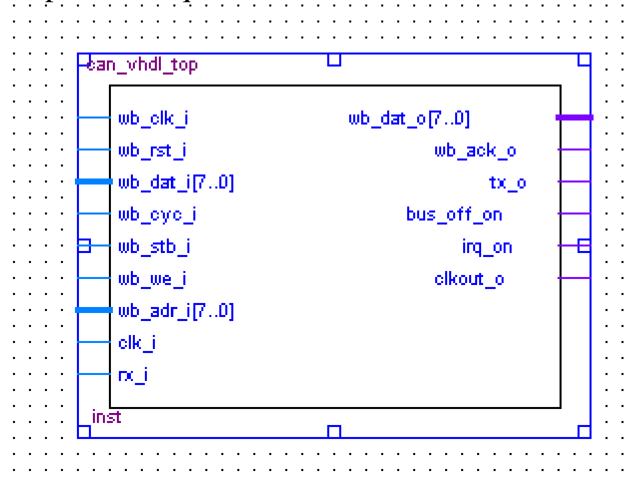


Figure 3.16: Le composant CAN

Pour ce qui est du microcontrôleur le *SOPC Builder* nous permet d'ajouter les composants suivants :

Module PLL : Ce module a été ajouté grâce à la bibliothèque *mega function* d'ALTERA qui contient un nombre important de modules réutilisables. La PLL sert d'un coté à créer l'horloge utilisée par la SDRAM et d'un autre à créer l'horloge du composant CAN qui doit être de 24MHz.

Le processeur NIOS : on a choisit la version *fast* du processeur NIOS II qui permet d'interpréter les instructions et de traiter les données du programme qui sert à piloter le contrôleur CAN.

La mémoire SDRAM : permettant de stocker le code et les données pour le processeur.

Un protocole d'interface série (JTAG_UART) permettant de télécharger, de débuguer le code sur la cible et de communiquer avec le PC.

Pour le reste se sont des entrées/sorties parallèles. Le composant (rassemblant tous les périphériques) généré par *SOPC Builder*, est donné par la figure ci-dessous.

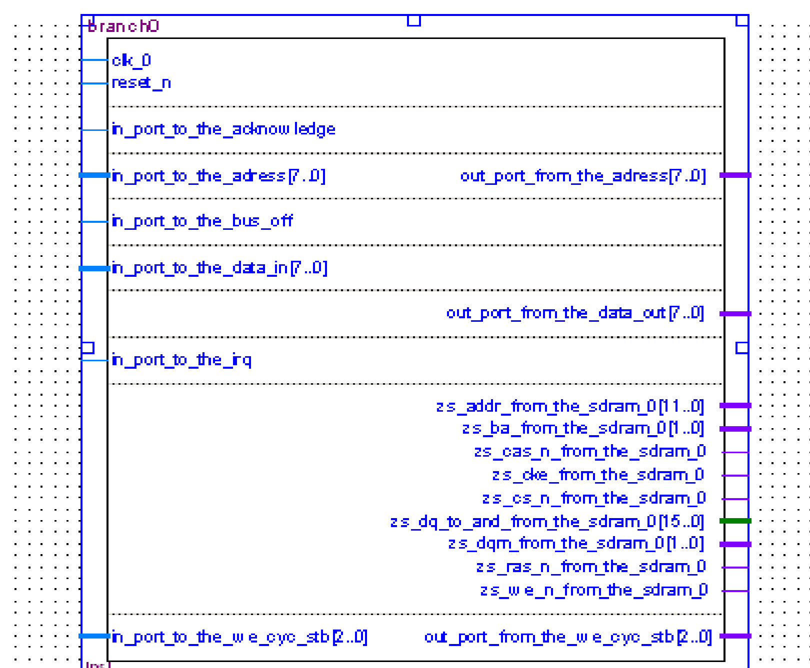


Figure 3.17: Le composant généré par le SOPC Builder.

Il suffit maintenant de faire le branchement entre le contrôleur CAN et le composant généré ainsi que l'assignement des différentes broches. Cette étape est réalisée dans l'environnement Quartus. Le câblage est donné en annexe n°3.
L'architecture obtenue qui contient le composant CAN, le microcontrôleur et les différentes connexions entre ces deux composants, consomme 3000 éléments logiques de l'ensemble de 35000, ce qui correspond à 12% des ressources du FPGA. Cette architecture sera compilée et comme résultat à cette compilation on va obtenir un fichier .sof. On va charger ce dernier sur la carte à partir de l'outil *programmer* du Quartus II pour définir l'architecture matérielle du circuit programmable.

Développement logiciel

Pour développer la partie logicielle de l'implémentation nous allons nous baser sur le modèle de programmation du SJA1000 décrit par la figure 2.9 en utilisant le langage C.
Le fait d'avoir utilisé des PIO nous a permis d'utiliser les fonctions d'écriture IOWR et de lecture IORD dont les prototypes sont :

- mettre une donnée data dans l'adresse « addr+off »

void IOWR (char addr,char off,char data)

-lire la donnée se trouvant à l'adresse « addr+off»

char IORD (char addr,char off)

Comme on l'a vu dans la partie simulation, l'initialisation du contrôleur consiste à configurer un ensemble de registres pour un fonctionnement précis de ce dernier. De là on voit l'intérêt de définir une fonction write_reg, qui a pour mission d'écrire dans les registres et dont l'organigramme est donné par la figure suivante :

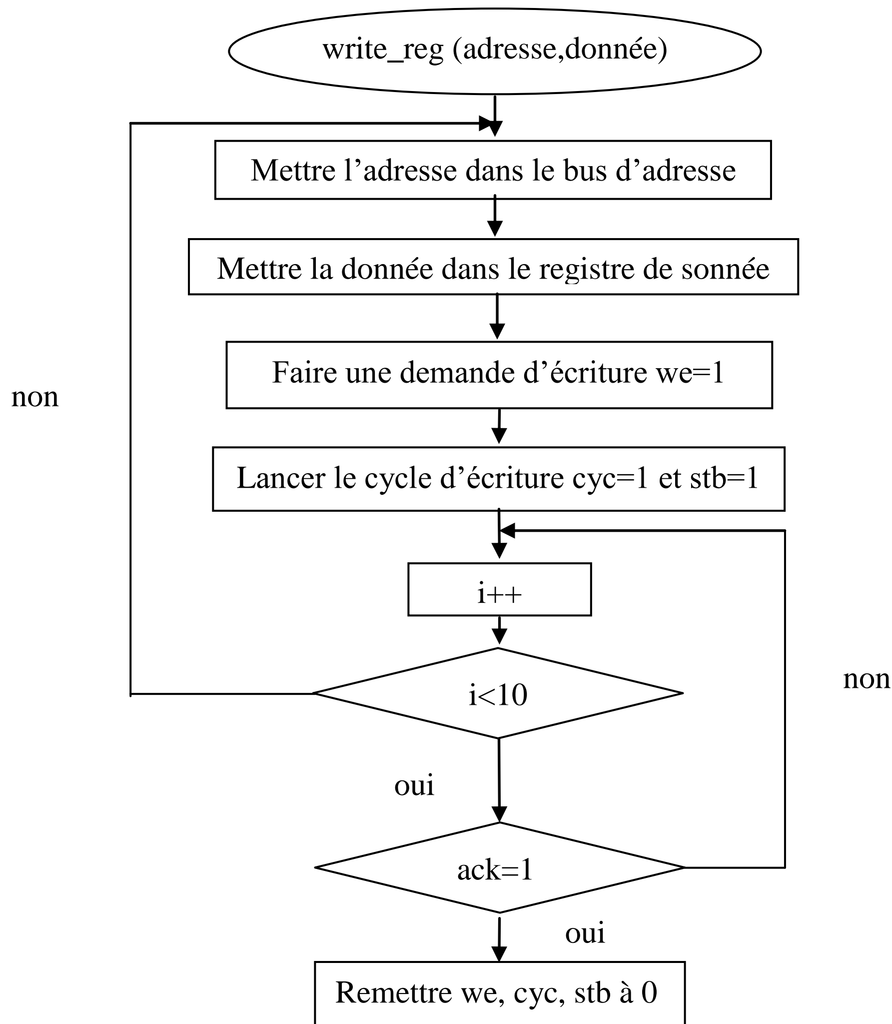


Figure 3.18 : Organigramme de la fonction write_reg.

Une autre fonction s'impose afin de permettre la vérification de l'état des différents registres, c'est la fonction read_reg qui permet de lire à partir des registres du contrôleur.
L'organigramme de cette fonction est donné par la figure suivante :

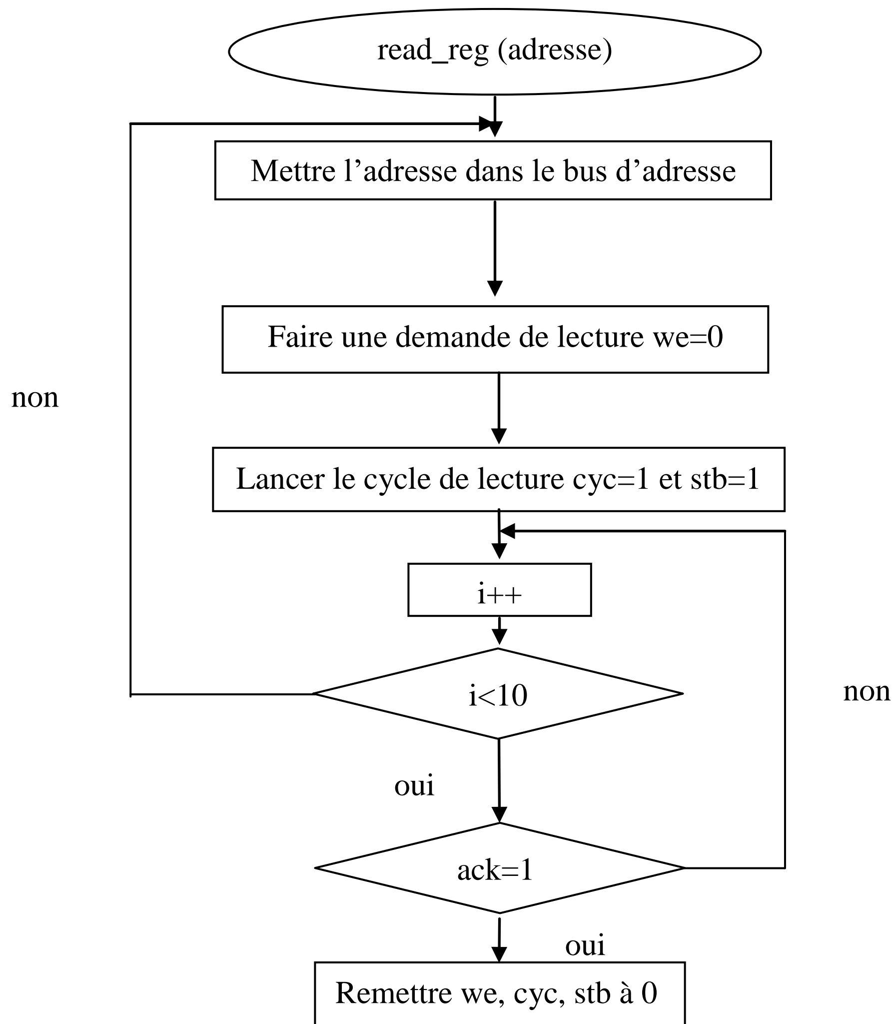


Figure 3.19 : Organigramme de la fonction read_reg.

Le code de ces fonctions ainsi que la fonction d'initialisation du contrôleur, de transmission et de réception de trame sont donnés en annexe n°4.

On est arrivé à la dernière partie de notre implémentation qui consiste à charger l'architecture matérielle conçue (celle de l'annexe n°4) sur la carte et d'exécuter la partie soft (le code donné en annexe n°4).

Nous avons pu vérifier le fonctionnement de notre implémentation en utilisant pour un début la console du NIOS IDE, nous avons donc réussi à exploiter les différents registres du contrôleur dans ses deux modes le BasicCAN et le PeliCAN.

On est donc parvenu à l'initialiser. La deuxième étape consistait à faire une transmission de trame à partir du contrôleur en gardant en vue les résultats de la simulation. Nous avons donc commencé par remplir le buffer de transmission, ensuite nous avons procédé à une vérifica-

tion pour voir si les valeurs à transmettre sont bien dans le buffer. Cette étape est montrée par la figure suivante :

```
hello_world_small.c    hello_world_small.c

        /*introduire les 8bits de l'identificateur*/
        write_reg(basiccan_ID1,0x01);
       /*suite de l'identificateur+le RTR*/
        write_reg(basiccan_ID2,0x21);
       /*introduire le premier octet de donnée*/
        write_reg(basiccan_data1,0x51);
        Data1=read_reg(basiccan_data1);
        printf("la donnée est =%x\n",Data1);
        ID1=read_reg(basiccan_ID1);
        printf("la première partie de l'ID est=%x\n",ID1);
        ID2=read_reg(basiccan_ID2);
        printf("la deuxième partie de l'ID est =%x\n",ID2);
    }
    /*faire la demande de transmission*/
    write_reg (command,0x01);
}
void reception (void) /*recevoir une trame*/
```

```
Problems  Console  Properties
hello_world_small_0 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (26/06/11 23:04)
il y a une trame à transmettre
la donnée est =51
la première partie de l'ID est=1
la deuxième partie de l'ID est =21
la trame a été transmise
```

Figure 3.20 : Remplissage du buffer de transmission

Il restait donc à vérifier que la trame circulait bien sur la ligne de transmission. Pour réaliser cette vérification nous avions besoin d'un analyseur logique, qui va être connecté à la broche GPIO correspondante à la sortie tx_o du contrôleur. L'utilisation de l'analyseur n'a pas conduit à un bon résultat, la ligne tx n'affichait que des 1 ce que signifiait l'absence de transmission. On a donc procédé à une vérification de l'état du contrôleur à travers le registre STATUS. L'information extraite de ce dernier affirmait que le buffer de transmission était plein et qu'il y avait une transmission en cours. Le problème résidait donc au niveau de la ligne tx, ou bien de l'analyseur logique.

III.5. Conclusion

On vient d'étudier l'implémentation de l'*IP core VHDL CAN Protocol Cotroller*. Une présentation générale de l'approche SOPC, de la carte ainsi que les logiciels utilisés était abordée. L'exploitation de la description VHDL du contrôleur nous a poussé à étudier globalement son fonctionnement et grâce à sa simulation, on a pu mettre en lumière le dysfonctionnement de certaines parties de l'*IP core*.

L'implémentation de l'*IP core* a été presque menée à terme, on a réussi à développer une architecture matérielle et logicielle permettant de piloter les différents registres du *VHDL CAN Protocol Controller*, cependant on n'a pas réussi à envoyer une trame sur la ligne tx.

Conclusion Générale

Conclusion générale

Grâce à notre travail on a pu aborder un concept de communication qui mérite une attention particulière ; le bus CAN. Notre cible était l'exploitation de ce dernier en utilisant l'approche SOPC et notre but était l'implémentation d'un contrôleur CAN sur la carte DE2 d'ALTERA.

Dans notre travail nous avons tout d'abord donné un aperçu sur les principes de base du bus CAN. Nous avons ensuite détaillé le fonctionnement du composant sur lequel était basée la description VHDL du contrôleur à implémenter. Un dernier chapitre a été consacré pour la présentation de l'environnement de travail ainsi que la mise en œuvre du projet.

Pour la réalisation de notre projet, on a travaillé avec l'environnement ALTERA en utilisant la carte DE2 pour l'implémentation, le logiciel Quartus II muni de l'outil *SOPC Builder* pour le développement de la partie *hard* et le logiciel NIOS IDE pour le développement de la partie *soft*.

Avant d'attaquer l'implémentation du contrôleur nous avons procédé à une simulation de la description VHDL, cette dernière nous a permis de mettre en lumière le dysfonctionnement de certaines parties de l'*IP core*.

Afin de réaliser l'étape suivante qui a consisté en l'implémentation du contrôleur CAN, il fallait le connecter au processeur NIOS II d'ALTERA. Pour cela nous avions deux manières de faire, à travers l'utilisation du bus AVALON ou des ports d'entrée/sortie parallèles PIO. Nous avons choisi la deuxième méthode afin de pouvoir écrire et lire à partir des registres du contrôleur en utilisant les fonctions d'écriture et de lecture des PIO. Cette méthode est plus simple certes, mais consommatrice (12% des ressources du FPGA) en termes d'éléments logiques par rapport à l'autre méthode.

A travers les résultats de l'implémentation nous avons pu voir que l'architecture conçue permettait de commander et de configurer le contrôleur CAN, cependant nous avons rencontré des problèmes dans l'émission et la réception d'une trame.

En conclusion, les perspectives que l'on peut envisager pour faire suite à ce travail, sont l'étude détaillée de la description VHDL du contrôleur CAN, et d'apporter les modifications nécessaires à cette description afin d'obtenir un bon fonctionnement. Pour ce qui est de l'implémentation de l'*IP core,* le branchement du contrôleur au processeur NIOS II pourrait se faire par l'intermédiaire du bus AVALON, cette méthode serait plus économique en terme de ressources des FPGA ou en terme du nombre d'éléments logiques consommés (moins de 1%) par l'architecture conçue.

Bibliographie

[1] Ahmed Ben ATITALLAH, Etude et Implantation d'Algorithmes de Compression d'Images dans un Environnement Mixte Matériel et Logiciel, thèse de Doctorat, Electronique, École doctorale des sciences physiques et de l'ingénieur, université Bordeaux I 2007.

[2] Ahmed Rachid, Frédéric COLLET, Bus CAN, université de Picardie Jules-Verne, Technique de l'ingénieur S8140.

[3] Altera corporation, ALTERA DE2 board Development and Education Board, Guetting Started Guide, 02 octobre 2005.

[4] Dominique PARET, Le bus CAN description de la théorie à la pratique, Dunod, Paris 1996.

[5] Fabien L'EXCELLENT, Mise en œuvre d'une application de traitement d'image sur la plate forme ML310**.**

[6] Fahmi GHOZZI, Optimisation d'une bibliothèque de modules matériels de Traitement d'images. Conception et test VHDL, implémentation sous forme FPGA, thèse de Doctorat, Electronique, École doctorale des sciences physiques et de l'ingénieur, université Bordeaux I 2003.

[7] Guerrin Guillaume, Guers Jérôme, Guinchard Sébastien, Les réseaux VAN-CAN, Informatique et réseaux, février 2005.

[8] Patrice KADIONIK, Le bus CAN, Ecole Nationale Supérieure Electronique Informatique et Radiocommunications, Bordeaux 2001.

[9] Philippe Hoppenot, le réseau CAN, département GEII, université d'Evry Val d'ESSONE, avril 2003.

[10] PHILIPS Semicondictors, Datasheet du SJA1000 stand-alone CAN controller, Product Specification, Integrated Circuits, 04 janvier 2000.

[11] PHILIPS Semiconductors, SJA1000 stand-alone CAN controller, Application note AN97076, Philips Electronics N.V. 1997.

[12] P. Kadionik , Guide d'utilisation de XENOMAI sur processur softcore NIOS II, Projet RTEl4I, Laboratoire IMS, IPB ENSEIRB, université de Bordeaux, 15 février 2011.

[13] Robert Bosch GmbH, CAN spécification, version 2.0, septembre 1991.

[14] Sylvain CHOISEL, Matthieu LIGER, Vincent OBERLE, Gestion du bus CAN, 3 février 2000.

Webographie :

[web1] www.irisbachelard.free.fr

[web2] www.wikipédia.com

[web3] www.opencores.org

[web4] www.whatis.techtarget.com

Annexe 1

PLAN D’ADRESSAGE DU SJA1000

Registre	bits	Adresse du registre		Mode opérationnel		Mode REST		Valeurs des bits en mode reset
		basicCAN	peliCAN	Lecture	écriture	lecture	écriture	
MODE	MOD7	_	0	valable	valable	valable	valable	0
	MOD6							0
	MOD5							0
	MOD4							0
	MOD3							x
	MOD2							x
	MOD1							x
	MOD0							1
CONTROL	CR7	0	_	valable	valable	valable	valable	0
	CR6							x
	CR5							1
	CR4							x
	CR3							x
	CR2							x
	CR1							x
	CR0							1
COMMAND	CMR7	1	_	FFH	valable	FFH	valable	1
	CMR6							1
	CMR5							1
	CMR4							1
	CMR3							1
	CMR2							1
	CMR1							1
	CMR0							

								1
STATUS	SR7 SR6 SR5 SR4 SR3 SR2 SR1 SR0	2	2	valable	_	valable	_	x x 0 0 x 1 0 0
INTERRUPT	IR7 IR6 IR5 IR4 IR3 IR2 IR1 IR0	3	3	valable	_	valable	_	1 1 1 0 0 x 0 0
INTERRUPT ENABLE	IER7 IER6 IER5 IER4 IER3 IER2 IER1 IER0	_	4	valable	valable	valable	valable	x x x x x x x x

Filtre d'acceptation	Les ACR et les AMR	4 5	16-19 20-23	_	_	valable	valable	x x
BUS TIMING0	BTR0	6	6	valable	_	valable	valable	x
BUS TIMING1	BTR1	7	7	valable	_	valable	valable	x
Output con-trol	OCR	8	8	valable	_	valable	valable	x
Buffer de transmission	TXinfo	_	16	_	Valable	_	_	x
	ID1	10	17	_	valable	_	_	x
	ID2+RTR+ DLC	11	18	_	valable	_	_	x
	ID3	_	_ ou 19	_	valable	_	_	x
	ID4	_	_ ou 20	_	valable	_	_	x
	Data1	12	19ou 21	_	valable	_	_	x
	Data2	13	20ou 22	_	valable	_	_	x
	Data3	14	21ou23	_	valable	_	_	x
	Data4	15	22ou24	_	valable	_	_	x
	Data5	16	23ou25	_	valable	_	_	x
	Data6	17	24ou26	_	valable	_	_	x
	Data7	18	25ou 27	_	valable	_	_	x
	Data8	19	26ou 28	_	valable	_	_	x
Buffer de réception	RXinfo	_	16	valable	_	_	_	x
	ID1	20	17	valable	_	_	_	x
	ID2+RTR+ DLC	21	18	valable	_	_	_	x
	ID3	_	_ ou 19	valable	_	_	_	x
	ID4		_ ou 20	valable	_	_	_	x
	Dtat1	_	19ou 21	valable	_	_	_	x
		22	20ou 22	valable	_	_	_	x

	Data2	23	21ou23	valable		_	_	x
	Data3	24	22ou24	valable	_	_	_	x
	Data4	25	23ou25	valable	_	_	_	x
	Data5	26	24ou26	valable	_	_	_	x
	Data6	27	25ou 27	valable	_	_	_	x
	Data7	28	26ou 28	valable	_	_	_	x
	Data8	29			_			
RX error counter	REC	_	14	valable	_	valable	valable	x
TX error counter	TEC	_	15	valable	_	valable	valable	x
Arbitration lost capture	ALC	_	11	valable	_	valable	_	x
Error code capture	ECC	_	12	valable	_	valable	_	x
Error warning limit	EWL	_	13	valable	_	valable	valable	96
RX message counter	RMC	_	29	valable	_	valable	_	0
RX buffer start adress	RBSA	_	30	valable	_	valable	valable	x
Clock divider	CDR	31	31	valable	valable	valable	valable	x

Annexe 2

SCHEMATIQUE DU BRANCHEMENT DE L'IP CORE A TRAVERS LE BUS AVALON

CAN0

clk_0
reset_n
rx_i_to_the_can_vhdl_acf_0
bus_off_on_from_the_can_vhdl_acf_0
clkout_o_from_the_can_vhdl_acf_0
tx_o_from_the_can_vhdl_acf_0
wb_ack_o_from_the_can_vhdl_acf_0
zs_addr_from_the_sdram_0[11..0]
zs_ba_from_the_sdram_0[1..0]
zs_cas_n_from_the_sdram_0
zs_cke_from_the_sdram_0
zs_cs_n_from_the_sdram_0
zs_dq_to_and_from_the_sdram_0[15..0]
zs_dqm_from_the_sdram_0[1..0]
zs_ras_n_from_the_sdram_0
zs_we_n_from_the_sdram_0

inst

PIN_D13
PIN_N25
PIN_F23
PIN_E16
PIN_D25
PIN_J22
PIN_AA6
PIN_AC3
PIN_AB4
PIN_AD3
PIN_AE2
PIN_AE3
PIN_AD2
PIN_Y5

Annexe 3

SCHEMATIQUE DU BRANCHEMENT DE L'IP CORE A TRAVERS LES PIO

essai0

clk_0
reset_n
in_port_to_the_acknowledge
out_port_from_the_adress[7..0]
in_port_to_the_bus_off_on
out_port_from_the_cyc_stb_we_rst[3..0]
in_port_to_the_data_in[7..0]
out_port_from_the_data_out[7..0]
in_port_to_the_irq_on
out_port_from_the_rx
zs_addr_from_the_sdram_0[11..0]
zs_ba_from_the_sdram_0[1..0]
zs_cas_n_from_the_sdram_0
zs_cke_from_the_sdram_0
zs_cs_n_from_the_sdram_0
zs_dq_to_and_from_the_sdram_0[15..0]
zs_dqm_from_the_sdram_0[1..0]
zs_ras_n_from_the_sdram_0
zs_we_n_from_the_sdram_0

can_vhdl_top

wb_clk_i
wb_rst_i
wb_dat_i[7..0]
wb_cyc_i
wb_stb_i
wb_we_i
wb_adr_i[7..0]
clk_i
rx_i
wb_dat_o[7..0]
wb_ack_o
tx_o
bus_off_on
irq_on
clkout_o
inst2

pllsdram

inclk0
c0
c1
inclk0 frequency: 50.000 MHz
Operation Mode: Normal

Clk	Ratio	Ph (dg)	DC (%)
c0	1/1	-27.00	50.00
c1	1/2/25	0.00	50.00

inst9
Cyclone II

DRAM_CLK
PIN_AA7

Annexe 4

Partie Soft

```
#include "sys/alt_stdio.h"
#include <stdio.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#define  BASEadress 0x01003030                  /*l'adresse de base du bus d'adresse*/
#define  BASEcyc_stb_we_rst 0x01003040          /*l'adresse de base du   cyc,stb,,we et rst*/
#define  BASEdata_out 0x01003010                /*l'adresse de base du bus de  donnée de sortie*/
#define  BASEacknowledge 0x01003020             /*l'adresse de base de  l'aknowledge*/
#define  BASEdata_in 0x01003000                 /*l'adresse de base du bus de donnée d'entrée*/
#define  BASErx 0x01003070                      /*l'adresse de base de rx*/
#define  mode 0x00                              /*l'adresse du registre mode*/
#define  TX_INFO 0x10                           /*l'adresse du registre TX_INFO*/
#define  pelican_ID1 0x11                       /*l'adresse du 1er octet du buffer de réception en mode
                                                pelican*/
#define  basiccan_ID1 0x0A                       /*l'adresse du 1er octet du buffer de réception en
                                                mode basiccan*/
#define  pelican_ID2 0x12                       /*l'adresse du 2ème octet du buffer de réception en
                                                mode pelican*/
#define  basiccan_ID2 0x0B                      /*l'adresse du 2ème octet du buffer de réception en
                                                mode basiccan*/
#define  pelican_data1 0x13                     /*l'adresse du 3ème octet du buffer de réception en
                                                mode pelican*/
#define  basiccan_data1 0x0C                    /*l'adresse du 3ème octet du buffer de réception en
                                                mode basiccan*/
#define  command 0x01                           /*l'adresse du registre COMMAND*/
#define  status 0x02                            /*l'adresse du registre d'état*/
#define  interrupt 0x04                         /*l'adresse du registre INTERRUPT pour le mode
                                                pelican*/
#define  clock_divider 0x1F                     /*l'adresse du registre clock divider*/
#define  pelican_ACF1 0x14                      /*l'adresse des registres du filtre d'acceptation mode
                                                pelican*/
#define  pelican_ACF2 0x15                       /*l'adresse des registres du filtre d'acceptation mode
                                                pelican*/
#define  pelican_ACF3 0x16                       /*l'adresse des registres du filtre d'acceptation mode
                                                pelican*/
#define  pelican_ACF4 0x17                      /*l'adresse des registres du filtre d'acceptation mode
                                                pelican*/
#define  basiccan_ACF1 0x04                      /*l'adresse des registres du filtre d'acceptation mode
                                                basiccan*/
#define  basiccan_ACF2 0x05                      /*l'adresse des registres du filtre d'acceptation mode
                                                basiccan*/
#define  pelican_timing0 0x05                    /*l'adresse du registre bit_timing0 mode pelican*/
#define  basiccan_timing0 0x06                   /*l'adresse du registre bit_timing0 mode basiccan*/
#define  pelican_timing1 0x06                   /*l'adresse du registre bit_timing1 mode pelican*/
#define  basiccan_timing1 0x07                  /*l'adresse du registre bit_timing1 mode basiccan*/
#define  output_control 0x08                     /*l'adresse du registre output control*/
int fonct;
char info;
char ID[4];
char donnee[8];
/************************la fonction d’écriture*************************/
```

```
void write_reg (char adresse,char donnee)/*écrire dans un  registre*/
{
    IOWR (BASEadress,0,adresse);                  /*mettre l'adresse du registre dans le bus d'adresse*/
    IOWR (BASEdata_out,0,donnee);                 /*mettre la donnée à écrire dans le bus de donnée*/
    IOWR (BASEcyc_stb_we_rst,0,0x01);             /*mettre we à 1*/
     IOWR (BASEcyc_stb_we_rst,0,0x07);            /*mettre cyc et stb à 1*/
     while (IORD(BASEacknowledge,0)==0)           /*attendre l'acknowledge*/
     {}
     IOWR (BASEcyc_stb_we_rst,0,0x00);            /*remettre les signaux we,cyc et stb à 0*/
  }
/*****************************************************************/
/************************la fonction de lecture************************/
  char read_reg (char adresse)                    /*lire à partir d'un registre*/
   {
   char donnee;
   IOWR (BASEadress,0,adresse);                   /*mettre l'adresse du registre dans le bus d'adresse*/
   IOWR (BASEcyc_stb_we_rst,0,0x00);              /*mettre we à 0*/
   IOWR (BASEcyc_stb_we_rst,0,0x06);              /*mettre cyc et stb à 1*/
   while (IORD(BASEacknowledge,0)==0)             /*attendre l'acknowledge*/
   {}
   donnee=IORD (BASEdata_in,0);                   /*lire la donnée*/
   IOWR (BASEcyc_stb_we_rst,0,0x00);              /*remettre le signaux we,cyc et stb à 0*/
   return donnee;
}
/*********************************************************************/
/************************la fonction de transmission************************/
void transmit (void)                              /*transmettre une trame*/
{
  char ID1;
  char ID2;
  char Data1;
  if (fonct==1)                                   /*mode pelican*/
  {
          write_reg (TX_INFO,0x01);               /*TX_FRAME_INFO*/
          write_reg (pelican_ID1,0xA5);           /*introduire les 8bits de l'identificateur*/
          write_reg (pelican_ID2,0x21);           /*suite de l'identificateur+le RTR*/
          write_reg(pelican_data1,0x51);          /*introduire le premier octet de donnée*/
   }
   Else                                           /*mode basiccan*/
   {
         write_reg(basiccan_ID1,0xA5);                 /*introduire les 8bits de l'identificateur*/
          write_reg(basiccan_ID2,0x21);                /*suite de l'identificateur+le RTR*/
         write_reg(basiccan_data1,0x51);               /*introduire le premier octet de donnée*/
    }
   write_reg (command,0x01);                      /*faire la demande de transmission*/
  }
/*******************************************************************/
/************************la fonction de réception************************/
void reception (void)                             /*recevoir une trame*/
{
   int i=0;
```

```
   if (fonct==1)                                    /*mode pelican*/
   {
          info=read_reg (TX_INFO);                         /*RX_FRAME_INFO*/

           if ((info& 0x80)==1)
          {
                   printf ("trame etendue\n");

                   for (i=0;i<3;i++)                                /*lire l'ID*/
                   {
                            ID[i]=read_reg(17+i);
                    }
                   for (i=4;i<11;i++)                               /*lire la trame reçue*/
                   {
                            donnee[i]=read_reg(17+i);
                   }
          }
          else
          {
                   printf ("trame standard\n");
                   for (i=0;i<1;i++)                      /*lire l'ID*/
                   {
                             ID[i]=read_reg(17+i);
                   }
                   for (i=2;i<9;i++)                      /*lire la trame reçue*/
                   {
                            donnee[i]=read_reg(17+i);
                   }
            }
   }
   else                                              /*mode basiccan*/
   {
             for (i=0;i<1;i++)                            /*lire l'ID*/
             {
                   ID[i]=read_reg(20+i);
              }
             for (i=2;i<9;i++)                         /*lire la trame reçue*/
             {
                    donnee[i]=read_reg(20+i);
              }
   }
write_reg (command,0x04);                        /*libérer le buffer de réception*/
}
 /************************************************************************/
/************************la fonction d'initialisation************************/
 void init_can (void)
{
   char reg;
   int waitreset=0;
   IOWR (BASEcyc_stb_we_rst,0,0x08);          /*mettre le rst à 1*/
   for (waitreset=0;waitreset<10000000;waitreset++)
```

```
{}
IOWR (BASEcyc_stb_we_rst,0,0x00);          /*on a besoin de juste une impulsion du reset*/
IOWR (BASErx,0,0x01);                      /*mettre l'entrée rx à 1*/
printf ("on va initialiser le controleur CAN\n");
write_reg (mode,0x01);                     /*passage au mode reset*/
printf ("le controleur est en mode reset\n");
reg=read_reg(mode);
printf ("le controleur est en mode=%x\n",reg);
write_reg(clock_divider,0x00);             /*configurer clock divider register*/
reg=read_reg(clock_divider);
printf ("le mode de fonctionnement=%x\n",reg);
if ((reg&0x80)==0x80)
{
      fonct=1;
}
else
{
      fonct=0;
}
if (fonct==1)
{
       printf ("on est en mode pelican\n");
       write_reg (interrupt,0x00);         /*désactiver les interruptions*/
      write_reg(pelican_ACF1,0xFF);              /*configurer les ACR et AMR*/
      write_reg(pelican_ACF2,0xFF);
      write_reg(pelican_ACF3,0xFF);
      write_reg(pelican_ACF4,0xFF);
      write_reg(pelican_timing0,0x41);     /*configurer le bus timing0 BRP=000001 et
                                           SJW=01*/
      write_reg(pelican_timing1,0x25);     /*configurer le bus timing1 TSEG1=0101
                                           TSEG2=010*/
      write_reg (output_control,0xDB);           /*configurer le registre output control*/
      write_reg (mode,0x08);                     /*passage en mode opérationnel*/
}
else
{
       printf("on est en mode basiccan\n");
       write_reg(basiccan_ACF1,0xFF);            /*configurer les ACR et AMR*/
       write_reg(basiccan_ACF2,0xFF);
       write_reg(basiccan_timing0,0x41);   /*configurer le bus timing0 BRP=000001 et
                                           SJW=01*/
       write_reg(basiccan_timing1,0x25);   /*configurer le bus timing1 TSEG1=0101
                                           TSEG2=010*/
       write_reg (output_control,0xDB);       /*configurer le registre output control*/
       write_reg (mode,0x00);                    /*passage en mode opérationnel*/
}
printf ("fin de l'initialisation\n");
}

/***********************************************************************/
/*********************** fonction principale ************************/
```

```
int main()
{
  char etat;
  init_can();
  printf ("le controleur CAN a été initialisé\n");
  etat=read_reg(status);
  if ((etat&0x04)==0x04)
        {
              printf ("il y a une trame à transmettre \n");
              transmit();
              printf ("la trame a été transmise\n");
        }
  if ((etat&0x11)==0x11)
        {
              printf ("on va recevoir une trame\n");
              reception ();
              if (fonct==1)
                    {
                     printf ("format de la trame=%x\n",info&0x80);
                    if ((info&0x80)==1)
                    {
                          printf ("la trame reçue est une trame étendue\n");
                          printf ("l'identificateur=%x\n",ID[0]);
                          printf ("l'identificateur=%x\n",ID[1]);
                          printf ("l'identificateur=%x\n",ID[2]);
                          printf ("suite de l'ID=%x\n",ID[3]&0xF8);
                          printf ("la nombre d'octet de donnee=%x\n",info&0x0F);
                          printf ("le premier octet de donnée=%x\n",donnee[0]);
                          printf ("le deuxième octet de donnée=%x\n",donnee[1]);
                          printf ("le troisième octet de donnée=%x\n",donnee[2]);
                          printf ("le quatrième octet de donnée=%x\n",donnee[3]);
                          printf ("le cinquième octet de donnée=%x\n",donnee[4]);
                          printf ("le sixième octet de donnée=%x\n",donnee[5]);
                          printf ("le septième octet de donnée=%x\n",donnee[6]);
                          printf ("le huitième octet de donnée=%x\n",donnee[7]);
                    }
                     else
                    {
                          printf ("la trame reçue est une trame standard\n");
                          printf ("l'identificateur=%x\n",ID[0]);
                          printf ("suite de l'ID=%x\n",ID[1]&0xE0);
                          printf ("la nombre d'octet de donnee=%x\n",info&0x0F);
                          printf ("le premier octet de donnée=%x\n",donnee[0]);
                          printf ("le deuxième octet de donnée=%x\n",donnee[1]);
                          printf ("le troisième octet de donnée=%x\n",donnee[2]);
                          printf ("le quatrième octet de donnée=%x\n",donnee[3]);
                          printf ("le cinquième octet de donnée=%x\n",donnee[4]);
                          printf ("le sixième octet de donnée=%x\n",donnee[5]);
                          printf ("le septième octet de donnée=%x\n",donnee[6]);
                          printf ("le huitième octet de donnée=%x\n",donnee[7]);
                     }
```

```
        }
        else
        {
                        printf ("l'identificateur=%x\n",ID[0]);
                        printf ("suite de l'ID=%x\n",ID[1]&0xE0);
                        printf ("le nombre d'octet de donnee=%x\n",ID[1]&0x0F);
                        printf ("le premier octet de donnée=%x\n",donnee[0]);
                        printf ("le deuxième octet de donnée=%x\n",donnee[1]);
                        printf ("le troisième octet de donnée=%x\n",donnee[2]);
                        printf ("le quatrième octet de donnée=%x\n",donnee[3]);
                        printf ("le cinquième octet de donnée=%x\n",donnee[4]);
                        printf ("le sixième octet de donnée=%x\n",donnee[5]);
                        printf ("le septième octet de donnée=%x\n",donnee[6]);
                        printf ("le huitième octet de donnée=%x\n",donnee[7]);
        }
    }
    return 0;
}
```