VersaCAN
Versatile CAN Board

Davide Rivola

EPFL-LSM  2004
# 1. Introduction

1.1. Introduction ................................................................................................... 4

# 2. Objectifs

2.1. Objectifs ......................................................................................................... 5

# 3. CAN

3.1. Introduction ..................................................................................................... 5

3.2. Couche OSI 1: Couche physique.................................................................... 7

3.2.1. Introduction ........................................................................................... 7

3.2.2. Mode de transmission............................................................................. 7

3.2.3. Interface (connecteurs) ..................................................................... 10

3.2.4. Transceiver ..................................................................................... 10

3.3. Couche OSI 2: Couche de liaison des données ........................................ 10

3.3.1. Introduction .................................................................................... 10

3.3.2. Arbitrage du bus .............................................................................. 11

3.3.3. Format des trames CAN .................................................................... 12

3.3.4. Bit Stuffing ..................................................................................... 14

3.3.5. Détection des erreurs ....................................................................... 14

3.3.6. Gestion des erreurs .......................................................................... 15

3.3.7. Message extended CAN .................................................................... 16

3.3.8. Contrôleurs CAN .............................................................................. 16

3.4. Couche OSI 7: Couche application ........................................................... 17

3.4.1. Introduction .................................................................................... 17

3.4.2. Exemple de couche OSI 7: CANopen .................................................. 17

3.4.3. Autres exemples .............................................................................. 18

# 4. USB

4.1. Introduction ................................................................................................. 19

4.2. Connexion physique.............................................................................. 19

4.3. Communications ................................................................................... 20

4.4. Les drivers côté "host" ......................................................................... 21

4.5. Contrôleurs USB .................................................................................... 22

# 5. Fonctionnement de VersaCAN

5.1. Vue d'ensemble du système................................................................... 23

5.2. Mise en place du système ..................................................................... 24

5.3. Utilisation VersaCAN ........................................................................... 24

5.3.1. Configuration initiale de la carte ................................................... 24

5.3.2. Modalité maître ............................................................................... 25

5.3.3. Modalité esclave ............................................................................. 29

5.4. Utilisation VersaCAN Monitor .............................................................. 31

5.4.1. Interface du logiciel VersaCAN Monitor ........................................... 31

5.4.2. Connexion avec la carte maître VersaCAN ....................................... 31

5.4.3. Changer les paramètres de la connexion ........................................ 32

5.4.4. Visualiser les informations relatives à un certain capteur ................... 33

5.4.5. Tenir trace de l'activité du bus CAN .................................................. 33

5.4.6. Régler les paramètre de transmission automatique d'un capteur........ 34

5.4.7. Régler les valeurs limites d'un capteur .............................................. 35

5.4.8. Visualiser les informations relatives à VersaCAN Monitor ................. 36

# 6. Protocoles de communication

6.1. Couche application du CAN bus en VersaCAN ..................................... 37

6.1.1. Introduction .................................................................................... 37
6.1.2. Structure des commandes ................................................................. 37
6.1.3. Set de commandes du maître .......................................................... 38
6.1.4. Set de commandes de l'esclave ....................................................... 40
6.1.5. Monitorage des esclaves ................................................................. 43
6.1.6. Monitorage des données des capteurs ............................................. 45
6.2. Protocole du bus USB pour VersaCAN ............................................. 46
   6.2.1. Introduction .................................................................................. 46
6.2.2. Structure des commandes ............................................................... 46
6.2.3. Set de commandes du coté "host" ................................................... 46
6.2.4. Set de commandes du coté carte maître VersaCAN ....................... 48
6.2.5. Déroulement de la communication ................................................. 49
7. Détails techniques .............................................................................. 50
   7.1. Microcontrôleur Atmel ATmega 162 ................................................ 50
   7.2. Contrôleur CAN Philips SJA1000 .................................................... 51
   7.3. Transceiver CAN Philips PCA82C250 .............................................. 55
   7.4. USB FIFO FTDI FT245BM .............................................................. 55
8. Réalisation de VersaCAN .................................................................... 57
   8.1. Introduction .................................................................................... 57
   8.2. Implémentation hardware de VersaCAN ......................................... 57
   8.3. Implémentation software de VersaCAN ........................................... 61
   8.4. Implémentation de VersaCAN monitor .......................................... 63
9. Logiciels de développements et outils hardware ................................ 64
   9.1. WinAVR ....................................................................................... 64
   9.2. Atmel AVR Studio 4 ..................................................................... 64
   9.3. PonyProg 2000 ............................................................................ 65
   9.4. Interface de programmation ISP .................................................... 66
   9.5. Peak-System CAN-USB Interface .................................................. 66
   9.6. Borland C++ Builder 5 ................................................................ 68
   9.7. MProg 1.0 ................................................................................... 68
   9.8. Protel 99 SE ............................................................................... 69
10. Conclusions ....................................................................................... 70
11. Développements futurs ..................................................................... 71
12. Remerciements .................................................................................. 71
13. Bibliographie ..................................................................................... 72
Annexes ................................................................................................. 73
1. Introduction

Le sujet de ce projet de diplôme est la réalisation et l’implémentation hardware et software d’une carte microcontrôleur multifonction (VersaCAN). La carte est dotée d’une interface CAN, USB et un écran alphanumérique.

Le CAN (Controller Area Network) est un protocole de communication série qui supporte efficacement le contrôle en temps réel de systèmes distribués avec un très haut niveau d’intégrité à niveau des données. Ce protocole a été développé par l’industrie de l’automobile dans les années 80. Aujourd’hui, son efficacité et sa robustesse ont élargi son domaine d’application au domaine des télécommunications et de l’automatisme.

USB (Universal Serial Bus) est un bus de communication très répandu dans les systèmes informatiques actuels. Né de l’initiative de Compaq, Microsoft, NEC et Intel, il permet d’assurer une connexion rapide, fiable, flexible et économique entre PC et ses périphériques.

En figure 1 on voit le système final qui sera conçu. Chaque carte (esclave) sera connectée directement à un capteur, les données brutes provenant du capteur seront traitées localement par la carte et seront transmises à des intervalles réguliers sur le bus CAN. Toutes les données des cartes slaves seront récupérées par une carte centrale (maître). Le maître n’est rien d’autre que la même carte multifonction fonctionnant dans une modalité différente. Le maître aura la possibilité de visualiser localement les données avec son propre écran alphanumérique ou de les envoyer travers du bus USB à un PC standard.
2. Objectifs

Le but de ce projet est de mettre en place un réseau de capteurs, où les mesures peuvent être lues à niveau local ou à distance depuis un PC standard.

L'idée était de travailler en parallèle avec un projet du LSM en cours d'exécution. Dans ce projet, débuté par M. Zennaro et continué par mon assistant S. Lopez, un réseau de capteurs développés par LEM Group1 est réalisé par des systèmes FPGA intégrant un core CAN, plus précisément un core CAN HurriCANe développé par ESA.

La carte versatile (VersaCAN) devait être constituée par des éléments électroniques certifiés à niveau industriel, comme le contrôleur CAN SJA1000 de Philips et le contrôleur USB FT245BM. La couche application devait être implémentée en software dans le microcontrôleur donnant une certaine flexibilité au projet.

L'application finale, utilisée comme démonstration de la réussite du projet, était d'implémenter sur VersaCAN un réseau de capteurs LEM. Et, si c'était possible, chercher de s'interfacer avec la carte FPGA développée par mon assistant.

Cette application finale avait le but de démontrer que le système développé pouvait assurer l'implémentation facile de différentes applications.

3. CAN

3.1. Introduction

CAN a été développé à l’origine par Bosch à la fin des années 80’ pour remplir les besoins de l’industrie de l’automobile. En effet un nombre toujours plus important de composants électroniques et informatiques était présent dans une voiture. Un protocole d’interconnexion simple, fiable, rapide et économique était demandé par l’industrie. CAN est devenu très rapidement un standard "de facto", et il a été implémenté par un nombre important d’industries. Avec les spécifications CAN 2.0B Bosch a décidé de fixer le standard et actuellement le protocole est défini par les normes ISO N°11519 (CAN Low Speed) et N°11898 (CAN High Speed).

Grâce à tout cela, le bus de communication série est devenu utilisé de façon très répandue par l’industrie aérospatiale, de la production industrielle et de l’automobile.

Raisons clefs de l’utilisation de la technologie CAN:

- Coût réduit des connexions
- Coût réduit des composants
- Nombre en continue hausse des puces CAN
- Une base de connaissance toujours plus grande
- Une variété importante de produits basés sur CAN
- Des outils de développement toujours plus faciles à utiliser

1 www.lem.ch
En figure 2 on peut voir pourquoi un système basé sur CAN est si intéressant:

![Figure 2: Avantages d'un système CAN](image)

Dans une architecture centralisée traditionnelle toute la puissance de calcul doit être concentrée dans un module central monolithique. Il doit être capable de traiter toutes les données qui arrivent au module central travers un nombre considérable d'interfaces. Tout cela engendre une complexité importante dans la conception de ce module. En même temps cette approche monolithique rend une mise au jour future du système très complexe. L'ajout d'une caractéristique supplémentaire pourrait être empêché par un manque d'interfaces ou de puissance de calcul.

Par contre dans le modèle du système basé sur CAN la puissance de calcul est partagé entre des modules plus simples. Il y a un prétraitement des données provenant des périphériques directement à niveau du module. L'information est ensuite envoyée sur le bus CAN pour un traitement successif ou pour l'exécution de certaines actions. On obtient ainsi un système économique (modules plus simples à fabriquer) et modulaire (un module supplémentaire peut être facilement connecté au bus CAN).

Comme on verra plus en détail dans les chapitres suivants, CAN est un protocole de communication série qui peut être utilisé pour transmettre jusqu'à 8 bytes de données à l'intérieur d'un seul message. Pour des quantités plus importantes de données des messages multiples sont communément utilisés. Les messages peuvent être transmis à une vitesse maximale de 1 Mbit/s.

CAN supporte la transmission de données entre plusieurs noeuds, aucun maître n'est requis pour superviser l'activité du réseau. Les messages CAN sont organisés par bits. Ils commencent toujours par un indicateur de "start of message", incluent un adresse (appelé "identifier") et contiennent des données et un CRC ("Cyclic Redundancy Check"). Chaque message doit recevoir un "acknowledgement" par les membres du réseau.

Dans les prochains chapitres, les différents aspects techniques du protocole CAN seront traités. Pour mieux structurer les informations, les chapitres seront subdivisés par les différentes couches du modèle OSI qui forment le standard CAN.

Bref explication d'OSI : au début des années 80, les participants de différents comités de standardisation avaient décidé de définir un modèle logique décrivant les différents éléments qui permettaient à deux systèmes informatiques de communiquer. Près de dix ans après le début des travaux, le modèle OSI ("Open Systems Interconnection") fut accepté. Ce modèle a pour but de séparer tous les processus et protocoles prenant part à une communication réseau et de les regrouper en sept couches en fonction de leur tâche.
Ces sept couches sont :

<table>
<thead>
<tr>
<th>Couche 7</th>
<th>Application</th>
<th>FTP, SMTP, Novell NDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Couche 6</td>
<td>Présentation</td>
<td></td>
</tr>
<tr>
<td>Couche 5</td>
<td>Session</td>
<td>DNS, SNA, MV, Novell Netbios</td>
</tr>
<tr>
<td>Couche 4</td>
<td>Transport</td>
<td>TCP, UDP, Novell SPX</td>
</tr>
<tr>
<td>Couche 3</td>
<td>Réseau</td>
<td>IP, IPX, Appletalk DDP</td>
</tr>
<tr>
<td>Couche 2</td>
<td>Liaison de données</td>
<td>Ethernet, Token-Ring, ATM LANE</td>
</tr>
<tr>
<td>Couche 1</td>
<td>Physique</td>
<td>Câble Coax, Twisted Pair, Fibre Optique</td>
</tr>
</tbody>
</table>

Table 1: Couches OSI

Le but de cette standardisation était de définir les interfaces entre chaque couche. Chaque couche est responsable de recevoir et transmettre l'information à ses deux couches voisines, indépendamment des autres couches.

Chaque ordinateur doit implémenter ces sept couches s'il entend communiquer sur le réseau. En revanche, si on applique ce modèle à un système CAN, seulement trois couches sont implémentées:

<table>
<thead>
<tr>
<th>Couche 7</th>
<th>Application</th>
<th>CANopen, DeviceNet, PCAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Couche 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Couche 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Couche 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Couche 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Couche 2</td>
<td>Liaison de données</td>
<td>Contrôleur CAN (Philips SJA1000)</td>
</tr>
<tr>
<td>Couche 1</td>
<td>Physique</td>
<td>Transceiver CAN (Philips PCA82C250)</td>
</tr>
</tbody>
</table>

Table 2: Couches OSI avec CAN

3.2. Couche OSI 1: Couche physique

3.2.1. Introduction

La couche physique du CAN, celle à niveau plus bas, décrit les propriétés électriques, mécaniques et fonctionnelles de la transmission physique des données entre les différents nœuds. La couche physique est concernée par la tache d’encoder les données qui arrivent depuis les couches supérieures dans une séquence de bit qui peuvent être transmises sous forme de signal physique grâce au mode de transmission.

Dans le nœud de réception la tache inverse est exécutée en décodant le signal physique dans des données ensuite transmises aux couches supérieures. La synchronisation des bits et le timing de ces derniers font aussi partie de la couche physique, comme aussi la description de l’interface au mode (câbles, connecteurs, assignement des pins).

3.2.2. Mode de transmission

Le mode de transmission physique n’est pas décrit à l’intérieur des spécifications CAN. La couche I peut être réalisé ainsi avec des différents types de mode de transmission:

- Twisted Pair
- Single Wire
- Fibre Optique
- RF
Le plus répandu est le "Twisted Pair", qui consiste typiquement dans un bus différentiel composé par deux fils avec chaque signal défini comme CAN_L (CAN LOW) et CAN_H (CAN HIGH) (En figure 3).

![Figure 3: Signal Twisted Pair (High Speed)](image)

Le nœud effectue une mesure différentielle sur les deux lignes. On définit dominant l'état du bus si la différence de tension entre CAN_H et CAN_L est positive (typiquement 2V). Dans le cas inverse, on définit le bus en état récessif. Cette mesure différentielle est très importante parce qu'elle permet de minimiser les interférences de l'environnement par rapport à un mode "Single Wire".

En effet le principe est très simple: une interférence externe cause un changement des niveaux de tension dans les deux lignes CAN_H et CAN_L, mais vu que l'interférence est identique dans les deux câbles, la valeur différentielle ne change pas. De cette façon la transmission de l'information est garantie.

![Figure 4: Réseau CAN "Twisted Pair"](image)

Un réseau CAN "Twisted Pair" (en figure 4) est terminé avec des résistances de 120 Ohm. Cette résistance correspond à l'impédance de la ligne du bus. En effet dans un bus qui n'est pas terminé, les signaux électriques réfléchissent à la fin de la ligne électrique et ils vont se superposer en l'endroit où le nœud est placé (en figure 5) en perturbant la lecture du nœud.

---

2 Figure 3.5.1-2, page 113 de la référence [1]
3 Figure 3.3-2, page 101 de la référence [1]
Il faut aussi remarquer qu’il y a une liaison étroite entre la vitesse de transmission et la longueur du bus. Une formule permet de calculer la longueur maximale pour un certain Bit Rate:

\[ L_{\text{max}} = \frac{x}{2 \cdot \text{BitRate}} - t_{\text{el}} \]

[1]

Où \( x \) indique le pourcentage de temps total utilisé pour la propagation (on assume \( x=0.85 \) pour nos estimations) et \( t_{\text{el}} \) le délai de temps de l’unité transceiver (on assume 300ns). On retrouve en figure 6 les résultats de cette formule:
3.2.3. Interface (connecteurs)

Différents standards peuvent être choisis comme interface au mode de transmission. Pour ce projet le standard CiA DS-102 a été choisi. CiA (CAN-in-Automation) a établi des spécifications qui indiquent les vitesses (de 10KBit/s jusqu'à 1 Mbit/s), les recommandations pour les câbles et les connecteurs. Ces derniers sont des connecteurs SUB-D9 avec la suivante configuration des pins (en figure 7):

![Connecteurs SUB-D9](image)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>CAN_L (dominant low)</td>
<td>CAN bus signal</td>
</tr>
<tr>
<td>3</td>
<td>CAN_GND</td>
<td>CAN ground</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>CAN_SHLD</td>
<td>Optional shield</td>
</tr>
<tr>
<td>6</td>
<td>GND</td>
<td>Optional CAN ground</td>
</tr>
<tr>
<td>7</td>
<td>CAN_H (dominant high)</td>
<td>CAN bus signal</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>CAN_V+</td>
<td>Optional external voltage supply Vcc</td>
</tr>
</tbody>
</table>

*Figure 7: Connecteurs standard CiA DS-102*

3.2.4. Transceiver

Le protocole CAN ne spécifie pas la couche physique, c'est pour ce motif que la plupart des contrôleurs CAN ne possèdent pas de circuits permettant de les connecter à un bus, qu'il soit filaire, à fibre optique ou tout autre mode de transmission possible. Un transceiver, tel que le 82C250 de Philips, permet d'assurer l'interface entre le contrôleur CAN et le bus physique. Le transceiver reçoit la séquence de bit à envoyer sur le bus et la transforme dans un signal physique adapté à ce dernier. Le chapitre 7.3. traite en détail le fonctionnement du transceiver 82C250 de Philips.

3.3. Couche OSI 2: Couche de liaison des données

3.3.1. Introduction

La transmission des données est gérée par la définition de la couche de liaison des données. La première fonction de cette couche est l'organisation des trames de données, lesquelles ne contiennent pas seulement que des données à transmettre mais aussi des informations de contrôle et des codes de détection des erreurs. Les informations de contrôle transmises permettent au nœud récepteur de reconnaître s'il va être concerné par la trame reçue et de détecter si la trame a été corrompue par une erreur de transmission. Les trames corrompues sont normalement rejetées par le nœud récepteur et automatiquement retransmises par le nœud transmetteur.

---

6 Référence [3]

7 Figure 3.5.2-1, page 115 de la référence [1]
La deuxième fonction concerne le contrôle de l’accès d’un nœud au mode de transmission ("Medium Access and Control", MAC). Ce dernier est nécessaire pour faire face aux problèmes causés par plusieurs nœuds qui cherchent d’accéder au bus, soit en prévenant le problème dès le début, soit en reconnaissant le conflit d’accès trouvant une solution qui permet de ne pas perdre des données.

3.3.2. Arbitrage du bus

Les trames de données transmises par un noœud sur le bus ne contiennent ni quelconque adresse du noœud expéditeur ni du noœud destinataire. C’est plutôt le contenu du message, sa signification qui est précisé par un champ d’identification ("Identifier Field", IF). Chaque noœud recevant un message regarde si celui-ci est intéressant pour lui grâce à l’IF. Si c’est le cas, il le traite, sinon, il l’ignore. Cet unique IF indique aussi la priorité des messages. Plus la valeur est faible, plus le message sera prioritaire. Si deux noœuds ou plus cherchent à avoir accès au bus en même temps, c’est celui de plus haute priorité qui gagne. Les messages de priorité inférieure seront automatiquement retransmis lorsque le bus sera libre.

Ce mécanisme, nommé arbitrage, est défini par la couche MAC (Mode Access Control). Pour assurer ce mécanisme toutes sortes de couches physiques CAN doivent supporter la représentation d’un état récessif et d’un état dominant dans le mode de transmission.

La transmission peut être en état récessif seulement si aucun noœud du bus est en train de transmettre un bit dominant. Si un noœud commence à transmettre un bit dominant (l’interface physique est implémentée de façon que plusieurs noœuds puissent intervenir sur le bus en même moment sans aucun dégât), l’état du bus devient dominant et les autres noœuds arrêtent de transmettre en attendant que le bus soit libre.

Tous conflits du bus sont résolus par le mécanisme du "ET câblé" (en figure 8), c’est-à-dire qu’un état dominant écrase un état récessif. Concrètement, si plusieurs noœuds débutent leur trame en même temps, le premier qui présente un bit récessif alors qu’au moins un autre présente un bit dominant perd l’arbitrage (en figure 9) (dans la trame, l’IF commence par le bit de poids fort).

Figure 8: Bus du type "ET câblé"\(^8\)

---

\(^8\) Figure 2.1-1, page 44 de la référence [1]
Figure 9: Mécanisme d’arbitrage

Tout ce passe donc comme si le message de plus haute priorité était le seul à être transmis. Lorsqu’un noeud perd l’arbitrage, il devient automatiquement un récepteur du message en cours de transmission, et il n’essaiera pas de retransmettre son message jusqu’au bus sera à nouveau libre. L’avantage d’un tel système est une utilisation meilleure du bus par rapport aux autres mécanismes (Ethernet par exemple).

3.3.3. Format des trames CAN

Le protocole CAN supporte deux types de types de message : Standard (CAN 2.0 A) et Extended (CAN 2.0 B). La seule différence est dans le numéro de bits utilisés pour le champ d’arbitration (IF), 11 pour le premier cas et 29 pour le deuxième. Dans ce paragraphe le format standard CAN 2.0A est traité. Le format extended est expliqué en chapitre 3.3.7.

En figure 10 le format d’une trame d’un message standard CAN:

Figure 10: Format d’une trame standard\(^9\)

\(^9\) Figure 2.2.1-1, page 48 de la référence [1]
Le format d’un message standard CAN est composé par :

- **Start Of Frame (SOF)**: un bit qui indique le début du message CAN (Hard Synchronisation).

- **Arbitration Field**: ce champ est composé par "Identifier Field" (11 bit en ce cas) qui identifie le message et le bit Remote Transmission Request (RTR) bit, utilisé pour distinguer entre message de type Data Frame et Remote Frame (expliqués plus avant).

- **Control Field**: ce champ est composé par le Identifier Extension (IDE) bit, celui-ci permet de distinguer si le message est du type standard ou extended, en plus on y trouve le Data Length Code (DLC), utilisé pour indiquer le nombre de byte de données contenu dans le data field. Si on est en train d’envoyer un Remote Frame, le DLC contient le numéro de bytes demandés.

- **Data Field**: avec une dimension qui peut varier entre 0 et 8 bytes, dans ce champ on trouve les données requises par le type de message qu’on est en train d’envoyer.

- **Cyclic Redundant Check (CRC)**: le CRC permet de vérifier l’intégrité des données envoyées.

- **Acknowledge Field (ACK)**: Les 2 bits de ACK sont envoyés comme récessives (1) et si les nœuds on reçu correctement les données ils vont le substituer avec un bit dominant (0).

- **End Of Frame (EOF)**: champ qui indique la fin du message

- **Intermission Frame Space (IFS)**: c’est un nombre minimal de bits qui sépare le message successif.

Le protocole CAN définit quatre types de messages différents:

- **Data Frame**: est un message régulier qui transmet des données du transmetteur aux récepteurs. Dans ce cas RTR doit être à 0 et le DLC doit indiquer le nombre de byte qu’on est en train d’envoyer. Celui-ci est le type de message le plus commun.

- **Remote Frame**: est transmit par un noeud qui demande des données. Le Remote Frame ne contient pas de données, mais il contient le ID du type de données demandées, le RTR bit mis à un, et il indique dans le champ DLC le nombre de bytes requis comme réponse. Les autres nœuds vont répondre avec un Data Frame avec même ID, DLC et les données demandées.

- **Error Frame**: est un message envoyé par n’importe quel nœud qui détecte une erreur sur le bus. Ce type de message est composé principalement par des bits dominant, il force le nœud, qui est en train de transmettre, à s’interrompre et il assure que tous les nœuds peuvent être conscients de l’erreur.

- **Overload Frame**: est utilisé quand un noeud a besoin d’un délai entre la réception de Data Frames successifs.
3.3.4. Bit Stuffing

Pour les Data Frames et les Remote Frames, les bits depuis le Start of frame jusqu'à la séquence de CRC sont codés selon la méthode du bit stuffing. Quand un transmetteur detecte 5 bits consécutifs de même valeur dans les bits à transmettre, il ajoute automatiquement un bit de valeur opposée (en figure 11).

Avant stuffing

|   |   |   |

Après stuffing

|   |   |

Après destuffing

|   |

Figure 11: Exemple de bit stuffing

3.3.5. Détection des erreurs

Le protocole CAN, étant originairement conçu pour l'industrie de l'automobile, doit assurer une haute fiabilité des données. Pour ce motif plusieurs mécanismes de détection des erreurs ont été spécifiés:

- Bit Check
- Frame Check
- Cyclic Redundancy Check
- Acknowledgment Check
- Stuff Rule Check

Bit Check

Chaque nœud expéditeur, surveille si le niveau du bus diffère du niveau transmis. Si la valeur ne correspond, pas un "bit error" est détecté. Le Bit Check permet ainsi de détecter très efficacement des erreurs globales. L'imposition d'un niveau dominant sur un bus à l'état recessif n'est pas considérée comme un "bit error" lors de la phase d'arbitrage et du acknowledgement.

Frame Check

Un contrôle est aussi effectué à niveau des trames elles-mêmes. Toutes les trames contemplées par le protocole CAN contiennent des champs de bits spécifiques fixes (bits de délimitation par exemple). Chaque nœud contrôle toutes les trames et détecte un "form error" en cas du non-respect de ces spécifications.

Cyclic Redundancy Check

Pour permettre à un nœud destinataire de vérifier l'intégrité des données reçues, le principe du contrôle cyclique de redondance (CRC) est appliqué au protocole CAN. Ce méthode généralement appliqué aux systèmes de transmission de données garanti une très haute probabilité de détection des erreurs.

Avec CRC, la trame à transmettre est représentée comme une trame polynomiale laquelle est divisée par un polynôme générateur. Le reste de cette division avec module 2 est la séquence CRC transmise sur le bus. Le nœud destinataire récupère les données et cette séquence CRC. Il calcule ensuite une séquence CRC avec les données de la trame reçue et la compare ensuite avec le CRC reçu. Si les deux valeurs correspondent, il y a une très grande probabilité que le message a été transmis sans erreurs.
Les coefficients de la trame polynomiale sont formés depuis les valeurs binaires sans Bit Stuffing provenant du SOF, le champ d'arbitrage, le champ de contrôle et le champ de données. Le tout étendu avec 15 bits bit de poids faible avec valeur 0. Le polynôme générateur de 15ième degré spécifié par le CAN protocole est donnée par:

\[ X^{15} + X^{14} + X^{10} + X^8 + X^7 + X^4 + X^3 + 1 \]

**Acknowledgment Check**

Un noeud expéditeur s'attend qu'au moins un noeud destinataire donne un acknowledgement dans le slot ACK de la trame de message transmise. En cas d'un manque d'acknowledgement (état recessif le long de tout le ACK slot) le noeud expéditeur détecte un "acknowledgement error".

**Stuff Rule Check**

Chaque noeud détecte une violation de la règle de Bit Stuffing ("stuff error") dès que six bits consécutifs au même niveau sont détectés dans une trame de message. De conséquence chaque noeud qui détecte un fanion d'erreur interprète ça comme une violation de la règle de Bit Stuffing et génère lui-même une trame d'erreur.

3.3.6. Gestion des erreurs

La détection des erreurs peut être efficace seulement si les noeuds CAN peuvent distinguer entre des petites interférences et des pannes permanentes. Pour cela chaque noeud dispose de deux compteurs d'erreurs à 8 bits :

- Transmit Error Counter (TEC)
- Receive Error Counter (REC)

Le noeud incrément ces compteurs chaque fois qu'il détecte un des ces cinq types d'erreurs (bit error, stuff error, CRC error, form error or acknowledgment error) et il décément le compteur chaque fois qu'il reçoit correctement un message. Ce mécanisme permet de séparer facilement les petites interférences des pannes permanentes.

Un noeud peut entrer dans un des ces trois états définis, à dépendance des valeurs des compteurs d’erreurs (en figure 12). Plus en détail :

- Si les deux compteurs ont des valeurs inférieures à 128 on se trouve dans l’état **Error Active**. Dans cet état le noeud peut encore participer dans les communications normales.

- Si tous les deux compteurs ont des valeurs supérieures à 127 on se trouve dans l’état **Error Passive**. Le message d'erreur envoyé par le noeud contient 6 bits récessifs, de cette façon le noeud ne perturbe plus le bus.

- Quand le Transmit Error Counter (TEC) arrive à 255 on atteint l’état **Bus Off**. Dans cet état le noeud arête d’influerenc le bus. Il pourra éventuellement redevenir **Error Passive** seulement après un certain période d’inactivité sur le bus ou re-initialisation.
3.3.7. Message extended CAN

Un message extended est assez similaire au message standard. La différence principale est le numéro de bits utilisés pour le ID. Pour ce motif le ID de 11 bits (appelé base identifier) est étendu par un champ de 18 bit (appelé identifier extension) qui est positionné juste après le bit IDE (Identifier Extension Bit). En effet le bit IDE permet de distinguer entre les deux types de trames de message. Le bit IDE est recessif de façon que les messages standard aient la priorité et puissent co-exister sur le même bus.

Les contrôleurs CAN qui supportent les messages extended peuvent aussi recevoir et envoyer messages standard. Quand un contrôleur qui supporte seulement les messages standard est utilisé dans un réseau, les autres contrôleurs doivent nécessairement envoyer que des messages standard.

Des messages extended ne peuvent pas être traités par des contrôleurs CAN 2.0A. Ils existent quand même des contrôleurs qui supportent seulement les messages standard mais reconnaissent les messages extended et les ignorent (version 2.0B Passive).

3.3.8. Contrôleurs CAN

Toutes les fonctions nécessaires pour le traitement du protocole des couches OSI 1 et 2 et aussi des fonctions supplémentaires sont implémentées dans des contrôleurs de protocole. Ces fonctionnalités sont disponibles à une CPU travers une simple interface. Cette dernière doit simplement fournir les messages à transmettre au contrôleur CAN et lire les messages reçus.

Fonctionnalités d’un contrôleur CAN:

- Arbitrage du bus
- Sérialisation et desérialisation de trame à envoyer/recevoir
- Génération et contrôle du CRC
- Détection et notification des erreurs
- Création d’une trame CAN
- Insertion et effacement des Stuffing Bits
- Génération et contrôle du bit d’acknowledgement
- Synchronisation de la séquence de bit reçue

Figure 2.1-3, page 45 de la référence [1]
Le fait de performer toutes les fonctions reliées au protocole dans un hardware séparé permet d'alléger le contrôleur principal du système (CPU) de ces taches temporelles critiques. En plus, le contrôleur CAN assure des fonctionnalités supplémentaires spécifiques (décrites en chapitre 7.2).

Filtrage des messages

La transmission des données sur un réseau CAN est basée sur le principe du "broadcasting" des messages. Cela signifie que tous les messages transmis sur le bus sont accessibles depuis tous les nœuds du bus. Le problème est que dans beaucoup de cas seulement certains nœuds auront besoin de recevoir certains messages. Pour cette raison presque tous les contrôleurs CAN intègrent un mécanisme de filtrage des messages: seulement les messages avec un IF pertinent seront notifiés au processeur central. Ce mécanisme, appelé "acceptance filtering" est traité plus en détail dans le chapitre 7.2.

3.4. Couche OSI 7: Couche application

3.4.1. Introduction

Beaucoup d'implantations CAN requièrent des services qui sont au-delà des fonctionnalités de base spécifiées par la couche de liaison de données mais qui peuvent être réalisé à niveau de la couche application selon le modèle OSI. Par exemple, la transmission et la réception de données dépassant la limite de huit bytes par message. Pour remplir ce besoin plusieurs organisations ont développé des couches application.

En effet le protocole de base CAN ne spécifie que les premiers deux niveaux du modèle de référence ISO OSI, les niveaux supérieurs sont libres et l'adoption de protocoles standard ou propriétaires est possible. Les développeurs software du système doivent effectuer ce choix.

Un protocole de haut niveau est quand même requis pour définir comment les 11/29 bits du IF (Identifier Field) et les 8 bytes de données du CAN bus sont utilisés entre les différents nœuds. Actuellement différents standards qui utilisent le bus CAN sont à disposition du développeur software.

Dans le cadre du projet le choix de développer une couche application propriétaire très simple a été effectué. Cette couche sera décrite en détail dans le chapitre 6.1. Dans les chapitres suivants des protocoles d'haut niveau standards seront expliqués à titre d'exemple.

3.4.2. Exemple de couche OSI 7: CANopen

Un des protocoles les plus utilisés est CANopen (CiA, 1996). CANopen est une implémentation de CAL (prochain chapitre) et il est défini par le profil de communication CANopen Cia DS-301. CANopen assume que le hardware du nœud est doté d'un transceiver et un contrôleur CAN comme spécifié par la norme ISO 11898. Cette famille de profile spécifie des mécanismes de communication standardisés et des fonctionnalités du nœud.

A dépendance du cahier de charge du système développé, est important de savoir que CANopen offre un Bit Rate minimal de 10 KBit/s (permettant d'atteindre une longueur du bus de 5 Km) et la possibilité de connecter jusqu'à 127 nœuds en chaque réseau CAN.

CANopen a été développé originalement pour des systèmes de contrôle du mouvement industriel, mais des réseaux CANopen sont utilisés aussi dans autres champs
d'application: transports publiques, véhicules tout-terrain, équipements médicales et applications domotiques.
Le protocole CANopen définit quelques types de messages pour gérer différents types de données:

- Messages PDO (Process Data Object) pour données temps-réel
- Messages SDO (Service Data Object) pour données de configuration des noeuds
- Messages NMT pour données de gestion du réseau

En outre, c'est intéressant de commenter la fonction du message SYNC, qui permet le synchronisme entre les messages temps-réel. Cette fonctionnalité coordonne la transmission des messages PDO.

3.4.3. Autres exemples

Quelques détails à propos d'autres couches application:

- **CAL (CAN Application Layer)**

Nommé et basé sur un protocole développé originellement par Philips Medical Systems, CAL est une couche spécifié et maintenu actuellement par le groupe d'utilisateur CAN in Automation (CiA). CAL est "royalty-free" et peu être implémenté librement. Les spécifications CAL (documents de référence DS-201...207) peuvent être obtenues chez CiA.

- **PCAL (Portable CAL)**

Un autre exemple d'une implémentation CAL, connue comme P-CAL (Portable CAN Application Layer), a été développé par l'Université des Forcés Armées Fédérales de Munich, Allemagne. Différentes licences (code source ou binaire) sont disponibles pour une gamme de processeurs.

- **DeviceNet**

DeviceNet est une couche application approuvée par CiA basé sur CAN 2.0A et il est largement utilisé dans des applications d'automation industrielle. DeviceNet (développé originellement par Rockwell/Allen-Bradley) est maintenant un bus ouvert géré par une organisation indépendante connue comme l'Open DeviceNet Vendor Association (ODVA), des copies des spécifications peuvent être obtenues par cette association. Les acheteurs des spécifications reçoivent une licence illimitée, "royalty-free" pour développer des produits DeviceNet compatibles.

- **SDS (Smart Distributed System)**

SDS est aussi une couche approuvée par CiA. Développée par Honeywell, une des utilisations principales de SDS sont les applications de contrôle des machines.
4. USB

4.1. Introduction

Le bus USB ("Universal Serial Bus") est né à cause des difficultés toujours plus grandes que les utilisateurs PC expérimentaient en essayant de connecter une gamme toujours plus grande de périphériques aux propres ordinateurs. Les fabricants de PC portables étaient intéressés à des connecteurs avec des profils plus petits par rapport à des connecteurs parallèles. Le but était aussi de réduire le nombre toujours plus important de câbles et transformateurs.

Premièrement USB offre une connectivité plus simple aux utilisateurs. USB élimine le besoin de différents connecteurs pour imprimantes, claviers, souris et autres périphériques. Tous types de données sont supportées, dès les simples mouvements de la souris jusqu'à l'audio et vidéo numérique.

USB est un protocole série et une connexion physique, qui transmet les données de façon différentielle sur un "Twisted Pair". Une autre paire de fils permet d'alimenter les périphériques depuis le "host" (normalement le PC).

Dans un système USB il y a toujours un appareil "host" (typiquement un PC). Le "host" est le maître du système USB et il contrôle et il planifie les activités de communication. Les périphériques, les nœuds contrôlés par USB, sont des esclaves répondants aux commandes provenant depuis le "host".

Quand un périphérique est connecté à un réseau USB, le "host" communique avec le périphérique pour apprendre son identité et trouver ainsi le driver requis pour le périphérique en question (un proces appelé énumération). Pour éviter le réglage d'interrupteurs DIP ou des problèmes avec des IRQ comme en passé, le "host" donne une adresse à la périphérique pendant l'énumération.

4.2. Connexion physique

Le standard USB spécifie deux types de câbles et deux variants de connecteurs. Câbles "High-Speed", avec un blindage assez robuste, qui peuvent atteindre une vitesse maximale de 12 Mbit/s et câbles "Low-Speed" avec une vitesse maximale de 1,5 Mbit/s.

Chaque câble USB dispose d'un connecteur "A" d'un côté et un connecteur "B" de l'autre (en figure 13). Les connecteurs "A" vont vers une connexion vers l'haut ("host"), les connecteurs "B" vers une connexion vers le bas (périphérique USB). Vu que les deux types de connecteurs sont différents, une installation incorrecte des câbles est impossible.
4.3. Communications

Les communications USB prennent place entre le "host" et les "endpoints" situés dans les périphériques. Un "endpoint" est une partie du périphérique adressable de façon unique qui peut envoyer ou recevoir des données. Quatre bits définissent les adresses des "endpoints" du périphérique; des codes indiquent aussi la direction et le type de la transmission.

L'idée des "endpoints", rappresent un concept important pour les transactions USB, celui des "pipes" (tuyaux). Toutes les transmissions se passent à l'intérieur de "pipes" virtuels qui connectent les "endpoints" avec le "host". Quand une connexion est établie avec le périphérique, chacun "endpoint" renvoie une description, une structure de données qui indiquent à l'"host" la configuration de "l'endpoints". Avec ces données l'"host" peut établir la communication.

USB supporte quatre types de transfert de données:

- Control Transfer
- Isochronous Transfer
- Bulk Transfer
- Interrupt Transfer

**Control Transfer:** échange la configuration et les informations des commandes entre l'esclave et le maître. Des CRCs contrôlent les données et demandent des retransmissions pour garantir l'intégrité des paquets envoyés.

**Bulk Transfer:** permet de déplacer des grosses quantités de données quand le temps de livraison n'est pas critique. Imprimantes et scanners sont des exemples d'applications typiques. Les Bulk Transfers utilisent les parties de bande passante qui ne sont pas utilisées en les remplissant complètement quand il n'y a pas d'événements plus importants. L'intégrité des Bulk Transfers est garantie par de contrôles CRC.

**Interrupt Transfer:** permet de transférer des petites quantités de données où un traitement immédiat de l'information est demandé (souris et claviers par exemple). Les données sont aussi contrôlées par CRC.
**Isochronous Transfer:** géré les données envoyées en continue, par des périphériques audio et vidéo par exemple. Les données sont critiques à niveau du temps, pour ce motif l'accès au bus doit être garanti. Il n'y a pas de CRC, le système doit être capable de tolérer occasionnellement des données corrompues.

### 4.4. Les drivers côté "host"

USB est un standard complexe qui demande un énorme quantité de support logiciel, soit du côté "firmware", soit du côté "host". Le développement d'un driver USB n'est pas évident. Pour ce motif Windows 98 et versions suivantes (on va traiter seulement les drivers Windows) incluent un support complet à niveau des drivers pour des applications communes USB.

Microsoft a spécifié un modèle de couches driver nommé WDM (Windows 32 Driver Model) (en figure 14). L'application Windows (grâce à des appelles aux API Windows) communique avec des classes ou des drivers personnalisés présents dans WDM.

A l'intérieur de WDM ces appelles engendrent des transfères de données entre les différentes couches utilisant des IRP (I/O request packets) jusqu'à arriver à la transmission physique sur le bus USB.

![Figure 14: Modèle WDM](image)

Le driver bas niveau "USB Bus Driver" gère la gestion de la puissance, l'énumération et les différentes transactions USB. Plus bas de ça, le "Host Controller Device", communique directement avec le hardware USB présent dans le PC. Tous ces deux drivers sont fournis directement avec l'installation de Windows, il n'y a pas besoin de les écrire et les modifier.

Windows, respectant les spécifications USB, segmente les drivers dans des classes, où le hardware supporté par une classe spécifique partage des interfaces similaires. Une classe définit une spécification de base pour un set de fonctionnalités. Tous les périphériques compris dans une certaine classe demandent un support software comparable.

Un exemple typique est donné par la classe "Human Interface Device" (HID), qui supporte des périphériques comme souris, joysticks et claviers. Un autre est la classe des écrans, qui contrôle la position, dimensions et alignement de l'image sur les écrans. Ces classes sont déjà présentes en Windows, si on veut par exemple ajouter un support HID dans une application, on peut s'interfacer avec cette classe sans écrire un driver nous-mêmes.
Une alternative aux classes est donnée par les drivers customisés. Un driver customisé exploite les fonctionnalités d'un particulier périphérique USB. Si par exemple un système d'acquisition de données a été construit, il y a forte probabilité que des classes de drivers ne soient pas disponibles. Pour ce motif un driver customisé doit être implémenté. Le discours est similaire pour des périphériques les quelles fonctionnalités dépassent celles d'une classe standard, dans ce cas aussi un driver customisé doit être écrit.

4.5. Contrôleurs USB

USB est lié au marché de grosse production des ordinateurs. Dizaines de fabricants proposent des centaines des puces USB.

Les composants USB sont difficiles à catégoriser, mais ils tombent généralement dans trois catégories:

- Contrôleurs USB pour le côté "host": ces contrôleurs sont intégrés directement sur la carte mère des ordinateurs. Ils ne sont pas très intéressant pour des projets de périphériques embarqués.

- Contrôleurs USB "stand-alone" pour le côté périphérique: ces contrôleurs (des FIFO/UART intelligents) gèrent la communication entre le bus USB est un microprocesseur central.

- Processeur avec interface USB: les contrôleurs sont ici directement intégrés dans le microcontrôleur lui-même.

Dans le chapitre 9.4. le contrôleur USB FTDI FT245BM, opté pour ce projet, sera expliqué en détail.
5. Fonctionnement de VersaCAN

5.1. Vue d’ensemble du système

La carte VersaCAN se présente de cette façon (en figure 15):

**Figure 15: Carte VersaCAN**

**Prise d'alimentation**

Cette prise d'alimentation, un jack standard, permet d'alimenter la carte VersaCAN. La carte dispose d'un régulateur de tension de 5V. Il faut pourtant alimenter cette carte avec 9V minimales. La carte nécessite normalement 80 mA.

**Connecteur USB**

Un connecteur USB standard de type B relie VersaCAN à l'host.

**Connecteur CAN bus**

Connecteur standard CiA DS-102. Plusieurs détails en chapitre 3.2.3. Ce connecteur relie la carte VersaCAN avec les autres.

**Porte d'expansion**

Cette porte d'expansion générique permet de connecter différents types de périphériques à VersaCAN. Elle fournit alimentation, un signal d'horloge à 16 MHz et 9 bits de portes I/O.

**Porte de programmation**

Ce connecteur, identique, au celui des cartes de développement Atmel STK 300, permet de programmer le microcontrôleur Atmel. Un LED rouge à coté de la porte s'allume pendant que la programmation est en cours.
Bouton de RESET
Permet de mettre la carte VersaCAN en RESET.

Contraste LCD
Trimmer qui permet de régler le contraste de l'écran LCD.

Boutons de navigation
Ces quatre boutons permettent de naviguer dans le système de menu implémenté dans VersaCAN. En figure 16 les fonctions des boutons sont décrites:

- UP, direction vers le haut, navigation et incrémentation des paramètres
- DOWN, direction vers le bas, navigation et décrémentation des paramètres
- ENTER, confirmation du choix
- BACK, annulation du choix, retour au menu précédent

5.2. Mise en place du système
La mise en place est très évidente:

- Prendre les cartes VersaCAN
- Les brancher au réseau CAN
- Brancher les capteurs aux cartes
- Connecter le câble USB à la carte maître
- Brancher l'alimentation

L'utilisateur peut passer maintenant à la configuration et l'utilisation des cartes VersaCAN.

5.3. Utilisation VersaCAN
5.3.1. Configuration initiale de la carte
Chaque carte VersaCAN contient le même "firmware" (logiciel), l'utilisateur peut choisir au premier démarrage de la carte si elle va être un maître ou un esclave. Ce choix initial n'est pas définitif, la modalité de fonctionnement de la carte peut être changée en tout moment. Voici comment l'affichage va se présenter pendant le premier démarrage (en figure 17).
En appuyant sur les touches UP et DOWN on peu déplacer l'indicateur et choisir la modalité maître ou esclave. Des que le choix a été fait, il faut le confirmer en appuyant la touche ENTER. Si maître a été choisi la modalité va être lancée directement dans le cas du choix de la modalité esclave il faut par contre choisir le MAC ID associé à la carte (en figure 18):

Les touches UP et DOWN de varier la valeur du MAC ID. Il faut aussi remarquer que l'utilisation du bouton BACK permet de retourner au menu initial. ENTER permet de confirmer le choix. Si le MAC ID utilisé n'est pas déjà utilisé sur le bus CAN la carte active la modalité esclave. En cas contraire le maître décide un nouveau MAC ID qui est encore libre et le notifie à l'esclave (en figure 19):

Dans ce cas l'utilisateur confirme qui l'as vu le message avec ENTER et la carte peut enfin montrer l'écran principal de l'esclave.

**5.3.2. Modalité maître**

L'écran principal de la modalité maître indique à l'utilisateur les informations suivantes:

- Version du firmware maître
- Nombre d'esclaves (capteurs) connectés sur le bus CAN
- Etat de la connexion USB

Dans l'écran en figure 20, l'utilisateur peut remarquer que la carte utilise la version 1.0 du firmware, 15 capteurs sont connectés sur le réseau et il y a une connexion en cours avec le logiciel de contrôle VersaCAN Monitor.
Si l'utilisateur veut exécuter des commandes il doit entrer dans le menu en appuyant ENTER. Il peut ensuite naviguer dans ce menu avec les quatre boutons. En figure 21 la structure du menu est indiquée:

**Menu 1: Sensors Info**

Avec cette fonctionnalité l'utilisateur peut afficher les valeurs liées à un certain noeud. Dans le cas où aucun nœuds serait connectée au réseau CAN une notification avertie l'utilisateur de l'impossibilité d'effectuer cette opération (figure 22):
Dans le cas positif l'utilisateur peut choisir le nœud intéressé (figure 23):

![Figure 23: Informations sur les esclaves](image)

Dans l'exemple figure 23 le nœud 30 possède actuellement un valeur instantanée de 262 et un RMS calculé de 1021. Les bouton UP et DOWN permettent de choisir des autres nœuds (s'ils sont disponibles).

**Menu 2: Set Auto Mode**

Cette partie de menu permet d'indiquer à un certain nœud d'envoyer des valeurs à des intervalles périodiques. L'écran demande à l'utilisateur de déterminer les paramètres suivants (figure 24):

![Figure 24: Set Auto Mode](image)

1. Capteur concerné par le réglage
2. Type de données que le capteur doit envoyer, soit un valeur instantanée, soit un valeur RMS calculé
3. Période de l'intervalle de transmission du capteur concerné

Le maître envoie ensuite la commande sur le réseau CAN et retourne à l'écran principal.
Menu 3: Set Threshold

Le maître peut aussi indiquer à un nœud les valeurs limites à surveiller. Si un capteur dépasse ces valeurs limites un message d'alarme est envoyé au maître qui va le notifier à l'utilisateur en affichant un écran d'alarme (en figure 25).

![Figure 25: Ecran d'alarme](image)

Pour choisir les différents paramètres l'utilisateur se retrouve avec un menu assez similaire au menu 2 (en figure 26).

![Figure 26: Set Threshold](image)

1. Capteur concerné par le réglage
2. Type de données que le capteur doit envoyer, soit une valeur instantanée, soit une valeur RMS calculé
3. Valeur limite minimale du capteur à surveiller
4. Valeur limite maximale du capteur à surveiller
5. Nombres d’échantillons du capteur utilisés pour calculer la valeur RMS

Le maître envoie ensuite la commande sur le réseau CAN et retourne à l’écran principal.
**Menu 4: Set VersaCAN**

Ce menu permet de régler la modalité de la carte VersaCAN, en ce cas on peut changer le maître dans un esclave. Le menu et la procédure sont exactement identiques à celle traitée en chapitre 5.3.1.

**Menu 5: About**

Le menu about donne des informations supplémentaires à propos de la carte VersaCAN et la version du firmware. En figure 27 l'écran qu'on retrouve avec VersaCAN Master 1.0:

![Figure 27: Ecran About](image)

5.3.3. Modalité esclave

L'écran principal de la modalité esclave indique à l'utilisateur les informations suivantes:

- Version du firmware esclave
- Nombre d'identification du nœud (MAC ID)
- Valeur instantanée du capteur
- Valeur RMS calculés sur un certain nombre d'échantillons du capteur

Dans l'écran en figure 28, l'utilisateur peut remarquer que la carte utilise la version 1.0 du firmware, la carte possède le ID 30, la valeur instantanée mesurée est 262 et la valeur RMS calculée est 1021.

![Figure 28: Ecran principal de l'esclave](image)

Si l'utilisateur veut exécuter des commandes il doit entrer dans le menu en appuyant ENTER. Il peut ensuite naviguer dans ce menu avec les quatre boutons. En figure 29 la structure du menu est indiquée:
Le menu de l’esclave est très limité par rapport au celui du maître (chapitre 5.2.2.).

**Menu 1: Set VersaCAN**

Ce menu permet de régler la modalité de la carte VersaCAN, en ce cas on peut changer l’esclave dans un maître. Le menu et la procédure sont exactement identiques à celle traitée en chapitre 5.3.1.

**Menu 5: About**

Le menu about donne des informations supplémentaires à propos de la carte VersaCAN et la version du firmware. En figure 30 l’écran qu’on retrouve avec VersaCAN Slave 1.0 :

![Figure 30: Ecran About](image)
5.4. Utilisation VersaCAN Monitor

5.4.1. Interface du logiciel VersaCAN Monitor

L'interface de VersaCAN Monitor 1.0 est limitée à l’essentiel pour démontrer les fonctionnalités de connexion avec la carte VersaCAN. En figure 31 les différents éléments de l'interface sont indiqués:

- Liste dynamique des capteurs connectés sur le bus CAN
- Valeurs relatives au capteur sélectionné dans la liste des capteurs
- Fenêtre dynamique indiquant l'activité sur le bus CAN

![Figure 31: Éléments de l'interface](image)

5.4.2. Connexion avec la carte maître VersaCAN

Pour initialiser une connexion avec la carte maître l'utilisateur doit seulement appuyer le bouton de connexion ou choisir le commande correspondant dans le menu (figure 32):

![Figure 32: Connexion avec la carte](image)
Si la carte VersaCAN est connectée correctement au bus USB, la connexion est établie et VersaCAN monitor peut visualiser les informations relatives à la situation actuelle du réseau CAN. En figure 33 les capteurs présents sont affichés à l'écran:

![Figure 33: Connexion établie](image)

### 5.4.3. Changer les paramètres de la connexion

Pour initialiser une connexion avec la carte maître l'utilisateur doit seulement appuyer le bouton des "settings" ou choisir le commande correspondant dans le menu (figure 34):

![Figure 34: Changer les paramètres](image)

L'utilisateur peut ensuite régler ces différents paramètres de la communication USB (en figure 35):

![Figure 35: Fenêtre de réglage des paramètres](image)
Les paramètres disponibles sont:

- BaudRate, vitesse de transmission
- Data Bits, nombre bits utilisés pour les données
- Stop Bits, nombre de bits qui indiquent la fin du message
- Parity, indique si on utilise le bit de contrôle de la parité

Le bouton OK confirme la sélection et change effectivement les paramètres de connexion.

5.4.4. Visualiser les informations relatives à un certain capteur

Pour visualiser les informations relatives à un capteur spécifique l’utilisateur doit simplement cliquer avec le bouton gauche sur le capteur désiré. VersaCAN Monitor visualise à côté les informations relatives (en figure 36):

![Figure 36: Informations relatives au capteur](image)

5.4.5. Tenir trace de l’activité du bus CAN

L’utilisateur peut tenir trace de l’activité du bus CAN. Pour activer cette fonctionnalité il doit appuyer le bouton "Run". VersaCAN affiche les derniers activité dans la fenêtre (en figure 37) et enregistre aussi toutes les informations dans un fichier nommé "log_0001.txt".

![Figure 37: Fenêtre de surveillance](image)
L'utilisateur peut ensuite arrêter le traçage avec "Stop", effacer les enregistrements avec "Reset" ou créer un nouveau fichier de log avec "New". Le nom du nouveau fichier est "log_x.txt" avec un nombre x qui incrément à chacun appui du bouton "New".

5.4.6. Régler les paramètre de transmission automatique d'un capteur

VersaCAN Monitor peut aussi régler les paramètres de transmission automatique d'un capteur. L'utilisateur sélectionne le capteur désiré et appuie sur le bouton droit de la souris. Il clique "Period..." dans le menu contextuel (en figure 38):

![Figure 38: Set Auto Mode](image.png)

La fenêtre de réglage apparaît (en figure 39):

![Figure 39: Fenêtre de réglage](image.png)

L'utilisateur choisit:

- Période
- Type de données

Il confirme ensuite la configuration choisie avec "OK". Le réglage est exécuté.
5.4.7. Régler les valeurs limites d'un capteur

VersaCAN Monitor peut aussi régler les valeurs limites d'un capteur. L'utilisateur sélectionne le capteur désiré et appuie sur le bouton droit de la souris. Il clique "Threshold..." dans le menu contextuel (en figure 40):

![Figure 40: Set Threshold](image)

La fenêtre de réglage apparaît (en figure 41):

![Figure 41: Fenêtre de réglage](image)

L'utilisateur choisit:

- Valeur Limite Minimale (Min)
- Valeur Limite Maximale (Max)
- Nombre des échantillons utilisés pour le calcul du RMS (Samples)
- Type de données

Il confirme ensuite la configuration choisie avec "OK". Le réglage est exécuté.
5.4.8. Visualiser les informations relatives à VersaCAN Monitor

Pour visualiser les informations relatives à VersaCAN Monitor l'utilisateur doit appuyer le bouton "About" ou choisir le commande correspondant dans le menu (figure 42):

![Figure 42: About](image)

La fenêtre "About" apparaît (en figure 43):

![Figure 43: Fenêtre About](image)

5.4.9. Déconnexion de la carte maître VersaCAN

Pour déconnecter la carte maître l'utilisateur doit appuyer le bouton de déconnexion ou choisir le commande correspondant dans le menu (figure 44):

![Figure 44: Déconnexion](image)
6. Protocoles de communication

6.1. Couche application du CAN bus en VersaCAN

6.1.1. Introduction

Les différents types d’implémentation de couche application standard (couche OSI 7) ont été introduits dans le chapitre 4.4. Ce type de couches possède des caractéristiques et des fonctionnalités très intéressantes pour le projet. Néanmoins le choix de développer une couche propriétaire a été fait.

Ce choix était motivé par les besoins du projet, l'implémentation d'une couche propriétaire permet de réduire à l'essentiel les fonctionnalités de la couche application en rendant l'implémentation software plus simple et facile à tester. La couche implémentée aux fonctionnalités suivantes:

- Requête des valeurs des capteurs (val. instantanée et RMS)
- Monitorage de l'état des esclaves
- Réglage des modalités de fonctionnement des esclaves
- Détectie d'un nombre d'identification des esclaves dupliqués
- Assignation automatique d'un nombre d'identification en cas de duplication
- Détectie de déconnexion d'un esclave

Il faut remarquer que l'implémentation de la couche OSI 7 est complètement à niveau du software. Pour ce motif le choix fait n'est pas du tout limitant.

6.1.2. Structure des commandes

Les commandes utilisées pour ce projet sont implémentées à niveau de la couche OSI 7. Cela signifie qu'ils doivent rentrer dans la structure des trames définis par les spécification de la couche OSI 2. Les différents éléments de la trame ont été définies de cette façon:

**Champ d'arbitration**

Les premiers 5 bits représentent le MAC ID. Cette variable identifie univoquement l'esclave. La couche application crée supporte pour ce motif un nombre maximal d'esclaves sur le réseau.

Dans le cas des commandes du maître le MAC ID est utilisé pour indiquer à quel nœud la commande réfère. Quand l'esclave envoie un message au maître, le MAC ID indique la provenance du message au maître.

Les autres 6 bits identifient le commande en question. Un grande attention a été donnée pour donner priorité aux commandes critiques (données périodiques, alarmes) par rapport à des commandes moins critiques (réponse à des polling).

**Champ RTR**

Le champ RTR indique si le message est une requête de Remote Frame. Cette fonctionnalité n'est pas implantée très souvent. C'est le cas pour ce projet aussi. L'implémentation choisie n'utilise que des trames de données.

**Champ des données**

Le champ des données du protocole CAN n'a pas une très grosse capacité. Seulement huit bytes peuvent être envoyés dans une trame CAN. Pour ce motif une utilisation la plus optimale a été recherchée. Par exemple:
Trois arguments de tailles suivantes doivent être estoqués dans la trame:

- Arg 0: 10 bits
- Arg 1: 8 bits
- Arg 2: 2 bits

Les arguments seront estoqués de cette façon dans la trame:

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Arg 0</td>
<td>Arg 1</td>
<td>Arg 2</td>
</tr>
</tbody>
</table>

La méthode de estoquage est évidente, les bytes sont remplis depuis le poids plus fort en direction du poids faible (bit 7 -> bit 0). Si la dimension d'un argument dépasse les 8 bits, les bits restants sont estoqués dans le byte suivant.

6.1.3. Set de commandes du maître

Le carte VersaCAN maître peut envoyer 9 types de messages aux esclaves (en table 3)

<table>
<thead>
<tr>
<th>Arbitration Field</th>
<th>CMD</th>
<th>LEN</th>
<th>ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 9 8 7 6 5 4 3 2 1 0</td>
<td>ID du nœud (dest) 0</td>
<td>Automatic_mode 2</td>
<td>2</td>
</tr>
<tr>
<td>0 0 0 0 0 1 0 0</td>
<td>ID du nœud (dest) 0</td>
<td>Automatic_mode 2</td>
<td>2</td>
</tr>
<tr>
<td>0 0 0 0 1 1 0 0</td>
<td>ID du nœud (dest) 0</td>
<td>Automatic_mode 2</td>
<td>2</td>
</tr>
<tr>
<td>0 0 1 1 0 0</td>
<td>Poll_rms 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 0 1</td>
<td>Poll_rms 0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1 0 0</td>
<td>ID du nœud (dest) 0</td>
<td>Req_inst 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0 0 1 0</td>
<td>ID du nœud (dest) 0</td>
<td>Req_rms 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 0 1 0</td>
<td>ID du nœud (dest) 0</td>
<td>Check_node 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1 1 0 0</td>
<td>ID du nœud (dest) 0</td>
<td>Slave_ping 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1 0 0 0</td>
<td>ID du nœud (dest) 0</td>
<td>Change_MAC_ID 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1 0 0 1</td>
<td>ID du nœud (dest) 0</td>
<td>Reset_DUP_Alarm 0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Set de commandes du maître

Voici une explication succincte des différents messages, dans les chapitres 6.1.5 et 6.1.6. l'utilisation des ces commandes est expliquée en détail.

**Automatic_mode**

Met le nœud en mode automatique, en cette modalité le nœud envoie une valeur du capteur à des intervalles périodiques réguliers. Le master indique au nœud la période des intervalles et le type de valeur à envoyer.

Structure du champ des données:

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Période</td>
<td>Type</td>
</tr>
</tbody>
</table>

La période est calculée de cette façon:

\[
\text{per} = 2^x \cdot t
\]
Où \( x \) est la valeur donnée comme argument 0 dans le message et \( t \) un période de base indiqué dans le nœud (125 ns dans l'implémentation).

Le type est indiqué de cette façon en l'argument 1:

- \( 0xF0 = \text{Valeur instantanée} \)
- \( 0x0F = \text{Valeur RMS} \)

**Threshold**

Indique les valeurs limites pour le nœud, si le nœud dépasse ces valeurs limites, il va envoyer un message d'alarme au maître.

**Structure du champ des données:**

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>Valeur Limite Minimale</td>
<td>Valeur Limite Maximale</td>
<td>Type</td>
<td>Ech. RMS</td>
<td></td>
</tr>
</tbody>
</table>

Les arguments sont assez évidents à comprendre, les premiers deux indiquent les valeurs limites (10 bits), ensuite le type de valeur à contrôler est indiqué. Dans ce cas aussi on a le même format de type utilisé auparavant. Le dernier argument indique à l'esclave combien de valeurs instantanées doit utiliser pour calculer la valeur RMS.

**Poll_inst**

Demande à tous les nœuds d'envoyer une valeur instantanée, cette commande n'a pas d'arguments.

**Poll_rms**

Demande à tous les nœuds d'envoyer une valeur RMS, cette commande n'a pas d'arguments.

La valeur RMS est calculée par l'esclave par la méthode du "Root Mean Square". La formule utilisée est:

\[
x_{rms} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_N^2}{N}}
\]  

**Req_inst**

Demande à un nœud spécifique d'envoyer une valeur instantanée, cette commande n'a pas d'arguments.

**Req_rms**

Demande à un nœud spécifique d'envoyer une valeur RMS, cette commande n'a pas d'arguments.

**Check_node**

Demande au nœud d'envoyer son état actuel, cette commande n'a pas d'arguments.
**Slave_ping**

Demande au nœud de confirmer sa présence sur le bus. Le nœud a 100 ms de temps pour répondre à ce message.

Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Ping ID</td>
</tr>
</tbody>
</table>
```

Ce message possède un seul argument: Ping ID. L’esclave va indiquer cette valeur dans le message de réponse au maître pour rendre valide ce dernier.

**Change_MAC_ID**

Demande au noeud de changer son MAC ID. Ce message est utilisé dans le cas de l'utilisation sur le réseau d'un MAC ID dupliqué pour résoudre ce conflit.

Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>MAC ID</td>
</tr>
</tbody>
</table>
```

Comme argument on a le nouveau MAC ID à donner à l’esclave.

**Reset_DUP_alarm**

Demande au noeud d'éteindre son alarme, cette commande est envoyée seulement après que le MAC ID dupliqué de l'esclave a été changé. En ce cas, l'alarme de duplication peut être éteinte. Cette commande n'a pas d'arguments.

### 6.1.4. Set de commandes de l’esclave

Le carte VersaCAN esclave peut envoyer 8 types de messages aux esclaves (en table 4):

<table>
<thead>
<tr>
<th>Arbitration Field</th>
<th>CMD</th>
<th>LEN</th>
<th>ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0 0 0 0 0</td>
<td>ID du nœud (source) 0</td>
<td>DUP_MAC_ID_req</td>
<td>0 0</td>
</tr>
<tr>
<td>1 0 0 0 0 1 1</td>
<td>ID du nœud (source) 0</td>
<td>Automatic_mode_ans</td>
<td>3 2</td>
</tr>
<tr>
<td>1 0 0 0 1 1</td>
<td>ID du nœud (source) 0</td>
<td>Threshold_alarm</td>
<td>4 3</td>
</tr>
<tr>
<td>1 0 1 1 0 0</td>
<td>-</td>
<td>Poll_inst_ans</td>
<td>2 1</td>
</tr>
<tr>
<td>1 0 1 1 0 1</td>
<td>Poll_rms_ans</td>
<td>2 1</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1 0 0</td>
<td>ID du nœud (source) 0</td>
<td>Req_inst_ans</td>
<td>2 1</td>
</tr>
<tr>
<td>1 1 0 1 0 1</td>
<td>ID du nœud (source) 0</td>
<td>Req_rms_ans</td>
<td>2 1</td>
</tr>
<tr>
<td>1 1 1 0 1 0</td>
<td>ID du nœud (source) 0</td>
<td>Check_node_ans</td>
<td>7 7</td>
</tr>
<tr>
<td>1 1 1 1 0 0</td>
<td>ID du nœud (source) 0</td>
<td>Slave_ping_ans</td>
<td>1 1</td>
</tr>
</tbody>
</table>

*Table 4: Set de commandes de l’esclave*

Voici une explication succincte des différents messages, dans les chapitres 6.1.5. et 6.1.6. L'utilisation de ces commandes est expliquée en détail.
**DUP_MAC_ID_req**

Message envoyé par l'esclave juste après la connexion au réseau. Cette commande notifie la présence d'un nouvel esclave et son propre MAC ID. Cette commande n'a pas d'arguments.

**Automatic_mode_ans**

Valeur périodique envoyée par l'esclave.

**Structure du champ des données:**

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

Les deux arguments sont la valeur mesurée par l'esclave et l'indication de type de cette valeur.

**Threshold_alarm**

Notification d'alarme au maître. Ce message est envoyé quand un dépassement des valeurs limite se vérifie dans l'esclave. L'esclave notifie la valeur, la valeur limite et le type de valeur.

**Structure du champ des données:**

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

Les deux arguments sont la valeur mesurée par l'esclave, la valeur limite qui a enclenché l'alarme et l'indication de type de cette valeur.

**Poll_inst_ans**

Valeur instantanée envoyée au maître.

**Structure du champ des données:**

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

**Poll_rms_ans**

Valeur RMS envoyée au maître.

**Structure du champ des données:**

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
**Req_inst_ans**

Valeur instantanée envoyée au maître.

Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
```

**Req_rms_ans**

Valeur RMS envoyée au maître.

Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
```

**Check_node_ans**

Notifie l'état de l'esclave au maître.

Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
```

Les différents arguments sont les mêmes vues précédemment. Dans le dernier argument, l'état des alarmes, le bit 1 correspond au DUP_ALARM, le bit 0 au DUP_WARNING.

**Slave_ping_ans**

Confirme au maître la présence du nœud sur le bus CAN.

Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
```

Ping ID, identifie à quelle commande slave_ping le nœud est en train de répondre. Ping ID rend la tâche de surveillance du maître plus aisée.
6.1.5. Monitorage des esclaves

Le maître est responsable du monitorage des esclaves sur le réseau. Il mémorise les MAC ID et l'état des nœuds connectés, vérifie la validité des MAC ID en évitant les conflits sur le réseau et détecte la déconnexion éventuelle des nœuds.

Connexion d'un nouvel esclave au réseau (MAC ID libre)

Le mécanisme de reconnaissance d'un nouvel esclave sur le bus est décrit (en figure 45):

- Après connexion et démarrage le nœud notifie sa présence avec le message "Dup_MAC_ID_Req".
- Les autres nœuds contrôlent si le MAC ID correspond aux propre MAC ID. S'il ne correspond pas ils n'entreprennent pas d'actions.
- Le maître contrôle si le MAC ID est libre et enregistre l'utilisation du MAC ID.

Connexion d'un nouvel esclave au réseau (MAC ID dupliqué)

Le mécanisme de reconnaissance d'un nouvel esclave dans le cas où le nouveau MAC ID existe déjà sur le bus est décrit en figure 46 et 47 :

11 Diagrammes créés par Serge Lopez - LSM/EPFL
Après connexion et démarrage le nœud notifie sa présence avec le message `Dup_MAC_ID_Req`.

Les autres nœuds contrôlent si le MAC ID correspond aux propres MAC ID. Si un nœud trouve une correspondance, il active son "Duplicate Alarm"

Quand un nœud est en "Duplicate Alarm" il ne transmet rien et il attend de recevoir la commande `Reset_DUP_Alarm`. Ce commande indique au nœud que le conflit a été résolu.

Le maître contrôle que le MAC ID soit libre et il découvre le conflit.

Le maître envoie la commande `Change_MAC_ID` au nœud qui est en train de causer le conflit.

---

**Figure 46: Nouvel esclave (avec problèmes) - Part 1**

**Figure 47: Nouvel esclave (avec problèmes) – Part 2**
• Le nœud qui cause le conflit reçoit la commande \textit{Change\_MAC\_ID} et notifie le changement au maître avec un nouveau message \textit{Dup\_MAC\_ID\_Req}.
• Avec la réception de ce dernier message le maître peut vérifier que le conflit a été résolu. Il peut ainsi notifier la résolution du conflit au nœud en état d’alarme avec la commande \textit{Reset\_DUP\_Alarm}.
• Le nœud reçoit le message, annule l’alarme et le système retourne finalement dans un état de fonctionnement normal où toutes les esclaves ont un MAC ID univoque.

Déconnexion d’un esclave

Si un esclave est déconnecté du réseau le maître doit être capable de détecter cet événement, mettre à jour ses tables d’état et libérer le MAC ID qui était occupé par l’esclave déconnecté.

Le maître va pourtant envoyer à des intervalles de 100 ms des commandes \textit{slave\_ping}. Les esclaves ont une fenêtre de temps de 100 ms pour notifier au maître la propre présence avec le message \textit{slave\_ping\_ans}. Si un esclave ne parvient pas à répondre dans le temps préfixé il est considéré déconnecté par le maître et son MAC ID est éliminé dans la table d’état.

L’esclave qui subit un RESET attend 200 ms avant de notifier ça présence au maître pour éviter tous conflits avec le mécanisme de détection de connexion et celui de la détection de déconnexion. Cette attente donne le temps au maître de détecter la déconnexion temporaire de l’esclave et éviter de donner un erreur de MAC ID dupliqué lorsque l’esclave se reconnecte.

6.1.6. Monitorage des données des capteurs

Le monitorage des données des capteurs est assuré par différents mécanismes. Soit le maître demande directement au nœud concerné d’envoyer une valeur. Soit le nœud envoie la valeur quand certaines conditions sont remplies:

\textbf{Valeurs automatiques}

Le nœud peut être configuré pour envoyer des données à des intervalles réguliers. Cette configuration est donnée par le message \textit{automatic\_mode}. Le message indique l’interval périodique de temps et le type de données (Inst. ou RMS) à envoyer. Si la période indiqué par ce message est égale à 0 le nœud arrête d’envoyer des messages périodiques.

Les valeurs périodiques sont transmises par le message \textit{automatic\_mode\_ans}. Ce message contient la valeur et le type de valeur.

\textbf{Alarmes valeurs critiques}

Le maître peut aussi indiquer des valeurs limites à l’esclave. Si la valeur lits sur le capteur est majeure de la valeur limite maximale ou mineure de la valeur limite minimale, le nœud notifie au maître une alarme.

Le maître utilise le message \textit{threshold} pour indiquer au nœud valeur min, max, type de valeurs et nombre d’échantillons pour calculer la valeur RMS.

L’alarme est notifiée au maître par le biais du message \textit{threshold\_alarm}. Valeur, valeur limite et type de données y sont indiquées.
Polling des valeurs

Le maître a aussi la possibilité de demander en n'importe quel moment des valeurs aux esclaves. Les messages utilisables sont: poll_inst, poll_rms, req_inst, req_rms.

Les nœuds répondent à ces messages avec: poll_inst_ans, poll_rms_ans, reqInst_ans, req_rms_ans.

6.2. Protocole du bus USB pour VersaCAN

6.2.1. Introduction

Comme expliqué en détail en chapitre 5, le carte VersaCAN, dans sa configuration complète, dispose d'une interface USB. Un logiciel (VersaCAN Monitor) peut dans ce cas s'interfacer avec le maître du réseau CAN pour surveiller et commander ce réseau depuis un PC standard.

Un set de commandes a été défini pour permettre la communication entre le logiciel et la carte maître VersaCAN. Ce set, très basique, peut être facilement étendu pour implémenter des fonctionnalités supplémentaires à l'architecture VersaCAN

6.2.2. Structure des commandes

La structure est très simple, le driver USB fourni gratuitement par FTDIchip géré tous les problèmes à niveau des trames des messages. En utilisant les API FTDI, le logiciel peut ouvrir facilement une connexion directe avec le contrôleur USB. A ce point là les commandes sont simplement des blocs de données avec la structure suivante:

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmd Length</td>
<td>Cmd ID</td>
<td>Slave ID</td>
<td></td>
</tr>
</tbody>
</table>

Premièrement Cmd Length indique la longueur totale de la commande, ensuite le Cmd ID (16 bits) spécifie le type de commande et après le Slave ID indique quel esclave doit être concerné par les messages.

Les bytes suivants contiennent les arguments relatifs au message spécifique. Ces arguments sont estoqués avec la même méthode pour la couche application du protocole CAN spécifiée dans le chapitre 6.1.2.

6.2.3. Set de commandes du coté "host"

Le logiciel Windows VersaCAN Monitor peut envoyer 6 types de messages (en table 5):

<table>
<thead>
<tr>
<th>Command Name</th>
<th>LEN</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>usb_enter_link</td>
<td>4</td>
<td>0x0101</td>
</tr>
<tr>
<td>usb_exit_link</td>
<td>4</td>
<td>0x0102</td>
</tr>
<tr>
<td>can_auto_mode</td>
<td>6</td>
<td>0x0201</td>
</tr>
<tr>
<td>can_threshold</td>
<td>9</td>
<td>0x0202</td>
</tr>
<tr>
<td>can_get_status</td>
<td>4</td>
<td>0x0203</td>
</tr>
<tr>
<td>refresh_data</td>
<td>4</td>
<td>0x0204</td>
</tr>
</tbody>
</table>

Table 5: Set de commandes USB du coté "host"
Voici une explication succincte des différents messages:

**usb_enter_link**

Indique au maître VersaCAN d'entrer en modalité de connexion. Dans cette modalité il acceptera les autres commandes depuis le PC et il notifiera l'activité du bus CAN au logiciel VersaCAN Monitor.

Cette commande n'a pas d'arguments.

**usb_exit_link**

Arrête la communication avec le maître. Après ce commandé le maître n'envoie plus de message au logiciel et attend que le logiciel envoie un commande *usb_enter_link* pour reprendre la connexion.

Cette commande n'a pas d'arguments.

**can_auto_mode**

Le message demande au maître d'envoyer un message CAN du type *automatic_mode* au nœud concerné (indiqué dans le byte 3 du message).

Structure du champ des données:

<table>
<thead>
<tr>
<th>Byte 4</th>
<th>Byte 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Période</td>
<td>Type</td>
</tr>
</tbody>
</table>

**can_threshold**

Le message demande dans ce cas au maître d'envoyer un message CAN du type *threshold* au nœud concerné.

Structure du champ des données:

<table>
<thead>
<tr>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
<th>Byte 7</th>
<th>Byte 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>Valeur Limite Minimale</td>
<td>Valeur Limite Maximale</td>
<td>Type</td>
<td>Ech. RMS</td>
<td></td>
</tr>
</tbody>
</table>

**can_get_status**

Le message demande dans ce cas au maître d'envoyer un message CAN du type *check_node* au nœud concerné.

Cette commande n'a pas d'arguments.

**refresh_data**

Ce message indique au maître d'envoyer toutes les données à propos les esclaves connectés sur le bus CAN. Les données incluent valeurs instantanées, RMS et les états des esclaves.

Cette commande n'a pas d'arguments.
6.2.4. Set de commandes du coté carte maître VersaCAN

La carte maître VersaCAN peut envoyer 4 types de messages à l'ordinateur (en table 6):

<table>
<thead>
<tr>
<th>Command Name</th>
<th>LEN</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>update_inst</td>
<td>6</td>
<td>0x0301</td>
</tr>
<tr>
<td>update_rms</td>
<td>6</td>
<td>0x0302</td>
</tr>
<tr>
<td>update_status</td>
<td>10</td>
<td>0x0303</td>
</tr>
<tr>
<td>alarm_notify</td>
<td>8</td>
<td>0x0304</td>
</tr>
<tr>
<td>update_sensor</td>
<td>5</td>
<td>0x0305</td>
</tr>
</tbody>
</table>

*Table 6: Set de commandes USB du maître VersaCAN*

Voici une explication succincte des différents messages:

**update_inst**

Notifie au logiciel PC le rafraîchissement d'une valeur instantanée en indiquant le nœud en question.

Structure du champ des données:

<table>
<thead>
<tr>
<th>Byte 4</th>
<th>Byte 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Valeur Inst</td>
<td></td>
</tr>
</tbody>
</table>

**update_rms**

Notifie au logiciel PC le rafraîchissement d'une valeur RMS en indiquant le nœud en question.

Structure du champ des données:

<table>
<thead>
<tr>
<th>Byte 4</th>
<th>Byte 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Valeur RMS</td>
<td></td>
</tr>
</tbody>
</table>

**update_status**

Notifie au logiciel PC le rafraîchissement de la table d'état en indiquant le nœud en question.

Structure du champ des données:

<table>
<thead>
<tr>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
<th>Byte 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6 5 4 3 2 1 0</td>
<td>7</td>
<td>6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Valeur Limite Minimale</td>
<td>Valeur Limite Maximale</td>
<td>Période</td>
<td>Per Type</td>
</tr>
</tbody>
</table>

**alarm_notify**

Notifie au logiciel PC la réception d'une alarme par le maître depuis un nœud en indiquant le nœud en question.
Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 4</th>
<th>Byte 5</th>
<th>Byte 6</th>
<th>Byte 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
```

- **Valeur**
- **Valeur Limite**
- **Ech. RMS**

**update_sensor**

Notifie l’état du capteur (connecté/déconnecté).

Structure du champ des données:

```
<table>
<thead>
<tr>
<th>Byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
```

- **Etat**

### 6.2.5. Déroulement de la communication

Pour établir une connexion entre le maître et le logiciel VersaCAN Master, ce dernier envoie un message *usb_enter_link*. L’esclave, après réception de cette commande, entre en connexion.

Le logiciel envoie ensuite un message *refresh_data*. Ce message indique au maître d’envoyer toutes les données à propos les esclaves connectés sur le bus CAN. Les données incluent valeurs instantanées, RMS et les états des esclaves.

Toute activité sur le bus CAN est notifiée au logiciel PC travers la connexion USB. Le maître assure cette notification par les commandes suivantes:

- `update_inst`
- `update_rms`
- `update_status`
- `alarm_notify`

Le logiciel ne se limite seulement à surveiller l’activité il peut aussi donner des commandes aux esclaves avec:

- `can_auto_mode`
- `can_threshold`
- `can_get_status`

Pour arrêter la connexion le logiciel envoie un message *usb_exit_link*. Après réception de cette commande le maître CAN arrête d’envoyer des messages.
7. Détails techniques

7.1. Microcontrôleur Atmel ATmega 162

Le microcontrôleur ATmega162 (en figure 48) est un microcontrôleur à 8-bit en technologie CMOS basé sur l'architecture RISC AVR. En exécutant une instruction par chaque cycle d’horloge, ATmega162 arrive à des performances atteignant les 1 MIPS par MHz en donnant la possibilité aux développeurs du système d'optimiser la consommation d'énergie par rapport à la vitesse d'exécution.

![Diagramme à bloc du ATmega 162](image.png)

Le core de AVR combine un set d'instruction assez considérable avec 32 registres de travail. Tous les 32 registres sont connectés directement avec l'Unité Arithmétique Logique (ALU), permettant à deux registres indépendants d’être accédés par une seule instruction exécutée en un seul cycle d’horloge. L'architecture résultant est plus efficiente du coté code et du coté performances.

ATmega162 possède les caractéristiques suivantes:

- 16 KB de mémoire FLASH programmable en modalité ISP (In-System-Program)
- 512 bytes de mémoire EEPROM
- 1 KB de mémoire SRMA
- Interface de mémoire externe
- 35 lignes I/O
- 32 registres de travail
- Interface JTAG permettant programmation, débug et Boundary-Scan
• 4 Timers/Compteurs configurables
• 2 USART série programmables
• 1 Watchdog Timer programmable avec oscillateur interne
• 1 porte série SPI
• 5 modalités d’économie de puissance

La mémoire FLASH ISP intégrée peut être reprogrammée par:

• Interface série SPI
• Programmateur conventionnel parallèle
• Interface JTAG
• Programme de “Boot” directement depuis le microcontrôleur lui-même

Dans le cas de la dernière option, un programme de démarrage est écrit dans un zone spécifique de la mémoire FLASH. Le programme de démarrage continue à marcher pendant que la mise à jour de la mémoire FLASH est en cours. De cette façon la mise à jour par des interfaces personnalisées peut être garantie.

Atmel ATmega 162 est un microcontrôleur puissant qui permet des solutions économiques et très flexibles dans plusieurs type d’applications embarquées. ATmega 162 AVR est supporté par plusieurs outils de développement comme:

• Compilateurs C
• Assembleurs de macros
• Logiciel de débug/simulation
• Emulateurs "In-Circuit"
• Kit d’évaluations

Les outils optés pour ce projet vont être traités en détail dans le chapitre 10.

7.2. Contrôleur CAN Philips SJA1000

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

Caractéristiques:

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

Le mode BasicCAN

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

Le mode PeliCAN

Le mode PeliCAN est le mode dans lequel il faut utiliser le SJA1000 pour avoir accès à toutes ses fonctionnalités (CAN 2.0B, registres d’erreur, informations sur arbitration perdue...).
Les principaux registres:

Le registre MODE

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

- Listen Only : Aucune action sur le bus.
- Self test MODE: Acquittement par un autre noeud non nécessaire pour une bonne transmission.
- Sleep MODE: Déconnexion 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 (voir plus bas).

Le registre COMMAND

Comme son nom l’indique, le registre COMMAND est utilisé pour commander le contrôleur CAN. Ces commandes peuvent être:

- Transmit Request (bit 0) : Demande la transmission des données présentes dans le buffer de transmission.
- Abort Transmission (bit 1) : Annule une demande de transmission faite et qui n’a pas encore commencé. Si elle a déjà commencé, elle se termine.
- Release Receive Buffer (bit 2) : Libère la place du dernier message reçu dans le buffer de réception.
- Clear Data Overrun (bit 3) : Réinitialise l’indicateur de surcharge.
- Go To Sleep (bit 4) : Ordonne au SJA1000 de passer en Sleep mode.

Le registre STATUS

- Receive Buffer STATUS (bit 0) : vaut 1 si un message est présent dans le buffer de réception.
- Data Overrun STATUS (bit 1) : vaut 1 si un message a été perdu à cause d’une saturation du buffer de réception.
- Transmit Buffer STATUS (bit 2) : vaut 1 si le buffer de transmission est prêt à recevoir des données.
- Transmission Complète (bit 3) : vaut 1 si la dernière transmission demandée a été correctement transmise et acquittée.
- Receiving STATUS (bit 4) : vaut 1 si un message est en cours de réception.
- Transmit STATUS (bit 5) : vaut 1 si un message est en cours de transmission.
- Error STATUS (bit 6) : vaut 1 si le compteur d’erreur de transmission ou celui de réception a atteint la limite d’alerte (96 par défaut).
- Bus STATUS (bit 7) : vaut 1 si le noeud est connecté au bus (ni en Sleep MODE ni Bus-Off pour cause d’erreurs à répétitions.
Le registre INTERRUPT

Ce registre permet d’autoriser la génération d’interruption sur la broche INT pour les événements suivants:

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

Les registres BUS TIMING0 et BUS TIMING1

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

Le buffer de réception

Le buffer de réception est implémenté sous la forme d’une FIFO de 64 octets. Le SJA1000 donne accès à une "fenêtre" qui permet à l’utilisateur de lire le dernier message reçu. Une fois cette lecture faite, l’utilisateur peut ordonner au SJA1000 de libérer la place correspondante dans le buffer. Ceci aura pour conséquence de déplacer la fenêtre vers le message suivant. La figure 49 permet de visualiser le fonctionnement du buffer.

En plus de la fenêtre de réception, l’utilisateur peut également accéder directement aux 64 octets du Buffer. On dispose pour cela d’un registre précisant la position du dernier message reçu (RX BUFFER START ADDRESS) et d’un registre précisant le nombre de messages présents dans le buffer (RX MESSAGE COUNTER). Lorsqu’un message arrive alors que le buffer de réception est plein, le message arrivant est perdu (par ce nœud), et l’indicateur de saturation ("Data Overrun Status") est positionné.

Figure 49: Buffer de réception
Le buffer d’émission

Le buffer de transmission est un simple buffer de 12 octets (jusqu’à 4 octets d’ID pour le CAN 2.0B et 8 octets de données) (en figure 50). Une fois que l’utilisateur a rempli ce buffer, il peut faire une demande de transmission.

Le filtre d’acceptation

Le filtre d’acceptation (en figure 51) (Acceptance Filter) a pour rôle de contrôler les identificateurs (identifiers) des messages présents sur le bus avant de les laisser entrer dans le buffer de réception. Une bonne utilisation de ce filtre permet d’éviter une saturation du buffer de réception. Le filtre d’acceptation est constitué des Acceptance Code Registers et "Acceptance Mask Register".

Figure 50: Buffer d’émission

Figure 51: Filtre d’acceptation

\[\text{Figure 4.2.1-7, page 145 de la référence [1]}\]
Les "Acceptance Code Registers" contiennent l'ID "optimal" attendu par le SJA1000 (c'est à dire celui que devra avoir un message pour pouvoir entrer dans le buffer de réception, si les "Acceptance Masks" sont tous à 0). Les Acceptance Mask Registers précisent les bits de l'ID qu'il faut tester (0 si un bit doit être contrôlé et 1 s'il n'a pas d'importance).

7.3. Transceiver CAN Philips PCA82C250

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

Le transceiver 82C250 code les bits DOMINANT et récessifs de la manière suivante (en figure 52):

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

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

Figure 52: Filtre d'acceptation

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

7.4. USB FIFO FTDI FT245BM

Le contrôleur FT245BM rentre dans la catégorie "stand-alone" décrite en chapitre 5.5. Ce contrôleur permet de transmettre de façon économique des données jusqu’à une vitesse 8 Mbit/s. Son design simple doté d'une FIFO (First In, First Out) le rend très facile à interfacier avec des ports I/O d'un microcontrôleur.

L'utilisation est très simple:

Transmission:

Pour envoyer des données de la périphérique vers l'host ça suffit d'écrire de données de dimension d'un byte pendant que le pin TXE# (Transmit Enable) est à niveau 0. Si le buffer de transmission du contrôleur (384 bytes) est plein ou il est occupé à traiter les dernier bytes écrit, la puce met TXE# à niveau 1 pour empêcher l’écriture de données supplémentaires. Dès que le contrôleur sera libre TXE# retournera à son état normal (0).
Réception:

Quand le host envoie des données vers la périphérique, FT245BM met RXF# (Receive Fetch) à 0 pour indiquer au microcontrôleur que au moins un byte de données est disponible pour la lecture dans le buffer. La périphérique alors lit les données jusqu’à RXF# retourne à 1 en indiquant qu’il n’y a plus de données à lire.

L’implémentation du coté host est très simple aussi. Le fabricant, FTDIchip) donne deux types de drivers:

- FTDI’s virtual COM port driver
- FTDI’s D2XX driver

Avec le premier driver le contrôleur USB est vu dans Windows comme un porté série virtuelle (COM4 par exemple). N’importe quelle application qui supporte la porte série peut envoyer et recevoir des données depuis le contrôleur.

Le deuxième driver, appelé D2XX, une application "ad hoc" peut accéder "directement" au contrôleur grâce à des API fournies par le driver D2XX. Des applications d’exemple avec code source (VB, Visual C++, Delphi, C++ Builder, .Net) sont disponibles sur le site du fabricant.

Il faut aussi remarquer que le driver D2XX est plus performant (vitesse maximale de 8Mbit/s) mais il est disponible seulement pour la plateforme Windows. Par contre la porte série est disponible pour Mac OS X et elle est supportée nativement dans le kernel Linux depuis la version 2.4.
8. Réalisation de VersaCAN

8.1. Introduction

Comme indiqué dans le cahier de charge, 10 cartes VersaCAN ont été produites. Deux entre eux disposent aussi de l'interface USB. Toutes les cartes sont connectées ensemble sur le bus CAN (figure 53). Le firmware des cartes est absolument similaire, l'utilisateur peut choisir localement sur la carte si elle va travailler en modalité esclave ou maître. Si les cartes ne possèdent pas d'écran la modalité peut être aussi établie directement pendant la programmation de la carte. Dans les chapitres suivants les différents aspects de la réalisation du projet VersaCAN seront traités.

![Figure 53: Schéma du système](image)

8.2. Implémentation hardware de VersaCAN

Le sujet de ce chapitre est de montrer le design hardware de la carte VersaCAN (en figure 54) et son implémentation sur un PCB (Printed Circuit Board). La carte est, soit l'interface entre le capteur et le réseau CAN (esclave), soit l'interface entre le réseau CAN et le bus USB (maître).

Protel SE 99 (chapitre 9.8.) a été utilisé pour créer le schématique et le PCB. Ils sont inclus en annexe.
Le système a été premièrement testé avec une carte de développement STK 500 de Atmel et une plaque Hirschmann. La communication entre le microcontrôleur et les différentes puces présentes dans le design a été testée. Les différents outils de développement, expliqués en chapitre 9, ont été testés et leur conformité au besoin du projet a été vérifiée.

Après ce pas essentiel un dessin schématique assez complet a été mis en place. Le suivant PCB a été crée (figure 55 et 56):
Certains composants ont été mis en place sur l'autre face du PCB:

![Image of PCB with components](image)

*Figure 56: Composants sur l'autre face de la carte*

Plusieurs composants ont été déjà introduits et expliqués dans le chapitre 7. Les autres seront expliqués le long de ce chapitre.

**Alimentation**

L'alimentation du circuit est réglée par un régulateur de tension 7805. Le régulateur accepte jusqu'à 32 V à l'entrée et donne à la sortie 5 V de tension continue. Un jumper à côté du régulateur permet de choisir si l'alimentation provient de la prise externe ou du bus CAN. Dans le cas de l'alimentation depuis le bus CAN la tension est de 24 V. La résistance des câbles est de l'ordre de 50 mΩ/m. La tension diminue le long du câble CAN à cause de l'effet Joule, avec 24 V on peut assurer les 5 V requises par la carte même en cas de câble très longs (>100m).

**Oscillateur 16 MHz**

À côté du microcontrôleur est placé l'oscillateur à 16 MHz. Il s'agit d'un quartz à 16 MHz couplé avec des capacités de 22 pF. Le microcontrôleur prend le signal de l'oscillateur comme horloge interne. Le microcontrôleur est aussi configuré pour sortir un signal horloge d'amplitude 5V sur le pin 0 de la porte B. Ce signal est en suite utilisé comme signal d'horloge pour le contrôleur CAN SJA1000.

**Oscillateur 6 MHz**

Il y a un autre composant qui demande un signal d'horloge: le contrôleur USB FT245BM. Malheureusement ce type de contrôleur n'accepte pas un signal de 16 MHz. Pour ce motif à ses entrées un oscillateur composé d'un quartz à 6 MHz et des capacités couplées de 22 pF a été placé.

**EEPROM**

Une EEPROM (modèle M93C46) est connectée au contrôleur USB. Elle permet de configurer les paramètres du contrôleur, son Vendor ID et son Product ID.

**Contrôleur CAN**

Le contrôleur CAN SJA1000 communique avec le microcontrôleur avec l'interface de mémoire externe. Son emplacement d'adresse de base dans le microcontrôleur est 0x1000.
**Contrôleur USB**

Le contrôleur USB FT245BM est communiqué avec le microcontrôleur avec l'interface de mémoire externe. Son emplacement d'adresse de base dans le microcontrôleur est 0x0800.

**Sélection du contrôleur actif sur le bus du microcontrôleur**

Les contrôleurs CAN et USB sont tous les deux connecté sur le bus du microcontrôleur. Il faut s'assurer que seulement un contrôleur puisse accéder à ce bus en même temps.

Le SJA1000 dispose d'un pin CS ("Chip Select"). Quand sur CS un signal bas (0) est présent, le SJA1000 est actif.

Le FT245BM ne dispose pas par contre d'un pin "Chip Select", on a alors connecté des portes logiques NAND entre le microcontrôleur et le contrôleur USB. De cette façon, seulement si à l'entrée de la porte logique est présent un niveau logique haut, le contrôleur USB peut perturber le bus.

Avec cette architecture le SJA1000 est active avec un signal logique bas et le FT245BM avec un signal logique haut. Les deux pins CS ont été pourtant connectés ensemble et reliés au pin 4 de la porte C. Ce pin correspond au bit 12 de l'adresse.

En récapitulant si on donne l'adresse 0x0800 (correspondant à la puce USB) le bit 12 de l'adresse sera à niveau haut. La puce USB sera active.

Si on donne par contre l'adresse 0x1000 (contrôleur CAN) on a un niveau bas qui correspond à l'activation du contrôleur en question.

**Driver LCD**

Beaucoup de pins I/O sont occupés par le bus adresse/donnée et la porte d'expansion vers le capteur. Pour limiter l'utilisation des pins on a placé un circuit driver (en figure 57) entre le microcontrôleur et l'écran LCD. Ce circuit permet de piloter le LCD avec seulement deux pins au lieu des 7 minimales normalement requises.

![Figure 57: Driver du LCD](image-url)
8.3. Implémentation software de VersaCAN

Le software de VersaCAN (firmware) a été développé presque complètement en langage C, seulement des petits bouts de code ont été écrits en langage Assembler. Dans ce rapport la structure de base du firmware est décrite mais pour une compréhension du firmware dans sa totalité une lecture approfondie du code et des commentaires est vivement recommandée.

**Architecture du code**

Le code source est composé de ces fichiers (table 7):

<table>
<thead>
<tr>
<th>Fichier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>types.h</td>
<td>Ce fichier comporte les définitions des différents types de variables et des structures.</td>
</tr>
<tr>
<td>versacan.h</td>
<td>Définitions relatives au programme principal.</td>
</tr>
<tr>
<td>versacan.c</td>
<td>Programme principale.</td>
</tr>
<tr>
<td>sja1000.c</td>
<td>Driver SJA1000, gère les interruptions, transfère les messages à transmettre, lit les messages reçus.</td>
</tr>
<tr>
<td>lem.c</td>
<td>Couche application CAN pour ce projet. Se trouve au milieu entre le programme principal et le driver CAN.</td>
</tr>
<tr>
<td>usb.c</td>
<td>Couche application USB, gère la communication avec le contrôleur USB FT245BM.</td>
</tr>
<tr>
<td>lcd_io.c</td>
<td>Driver LCD, gère la communication avec le LCD.</td>
</tr>
</tbody>
</table>

Table 7: Architecture du code

La carte VersaCAN doit être capable de gérer les événements de l'utilisateur, comme la navigation du menu sur l'écran et en même temps de gérer des tâches en "background" comme recevoir des messages CAN et les traiter. Cette gestion simultanée de différentes tâches demande une conception soignée de la structure de l'application.

**Interruptions**

Deux interruptions hardware ont été définies.

- Un timer, chaque 100 ms il enclenche une interruption et il incrémenté des variables. Ce timer est utilisé pour contrôler les états des boutons et détecter si des esclaves ont été déconnectés du bus CAN
- Une interruption sur le contrôleur CAN, quand un message CAN arrive le contrôleur CAN active l'interruption et les messages sont copiés depuis le contrôleur CAN dans un buffer qui réside dans la SRAM du microcontrôleur.

**Événements**

Pour gérer les différentes taches un système de boucle d'événements à été mis en place. L'application principale, initialise les variables, les périphériques et entre dans une boucle infinie (while(1)) à l'intérieur de cette boucle la fonction get_event() est continûment appelée.

On peut faire un exemple: get_event() retourne l'événements "EV_ENTER", cela signifie que le bouton ENTER a été appuyé, le programme alors vérifie en quel état se trouve l'application, par exemple on est dans l'écran principal de l'application, dans ce cas alors la fonction display_menu() est appelée et le menu apparaît sur l'écran.

Mais vu que les événements donnés par l'utilisateur se passent de façon sporadique par rapport au nombre de cycles par secondes du microcontrôleur. La fonction get_event()  

---

13 Plusieurs idées de la structure ont été inspirées par Yampp, un excellent lecteur MP3 avec uC Atmel [www.yampp.com]
donne dans la majorité des cas "EV_IDLE" (pas d'événements). Alors la boucle d'événements continue à appeler de façon répétée `get_event()`.

Mais qu'est qu'elle fait exactement `get_event()`? Cette fonction lit dans le buffer d'événements s'il y a quelque valeur, appelle la fonction `idle()` et ensuite retourne l'événement.

**Fonction idle()**

Jusqu'à maintenant on a vu que généralement `get_event()` est appelé de façon répétée. Un petite partie du temps total est occupée par le traitement des événements, le reste du temps le système continue à appeler `idle()`.

Cette fonction gère les boutons, les messages CAN et les messages USB.

On peut affirmer alors que la structure mise en place correspond à un système d'exploitation primitive où une petite partie du temps est occupé au traitement des événements utilisateur et la grande partie restante est dédié aux tâches en background de gestion des périphériques.

**Schéma fonctionnel de la structure**

![Figure 58: Structure de VersaCAN](image)
8.4. Implémentation de VersaCAN monitor

VersaCAN monitor est une application C++ très simple qui a été développée à partir d'une application d'exemple fournie par FTIDchip. Cette application communique avec le contrôleur USB travers les API de FTDI. Ces API sont très simples à comprendre, les fonctions plus utiles sont:

- FT_Open: ouvre le périphérique
- FT_Close: ferme le périphérique
- FT_Read: lit des données depuis le périphérique
- FT_Write: envoie des données vers le périphérique

Dans VersaCAN Monitor un timer de 50 ms a été mis en place, chaque ms le timer s'active, lit les valeurs depuis les périphériques et affiche à l'écran les valeurs mises à jour.

VersaCAN Monitor est implémenté sur un normal PC Windows, vu l'énorme différence de performance par rapport à un simple microcontrôleur, l'implémentation de VersaCAN Monitor n'as pas été une tâche très compliquée. En plus cette tâche a été ultérieurement allégée grâce à l'environnement de développement offert par Borland, très performant et facile à utiliser. On conseille pourtant de lire directement le code C++ avec les commentaires pour comprendre en détail le fonctionnement de VersaCAN Monitor.
9. Logiciels de développements et outils hardware

9.1. WinAVR

WinAVR (prononcé “whenever”) est une collection d’outils de développement open source pour la série AVR de microcontrôleurs Atmel. Le développement du code C, est assuré par PN (Programmer Notepad) (en figure 59) un éditeur de texte très convivial. Depuis PN on peut lancer l’exécution d’un fichier Makefile qui va utiliser les différents outils de WinAVR pour compiler, debugger et programmer le microcontrôleur. Le compilateur inclus en WinAVR est avr-gcc, un compilateur complètement gratuit et open source. Pour la plateforme AVR sont disponibles des autres compilateurs commerciaux, mais une approche open source a été gardée, quand c’était possible, le long de tout le projet.

![Figure 59: WinAVR](image)

9.2. Atmel AVR Studio 4

AVR Studio 4 (en figure 60) est un environnement de développement (IDE) utilisé pour écrire et debugger des applications AVR en environnement Windows. AVR Studio 4 est fourni gratuitement par Atmel, le fabricant des microcontrôleurs AVR. Un assemblateur et un simulateur y sont inclus. Les suivantes périphériques hardware de développement sont supporté:

- ICE 50
- ICE 40
- JTAGICE
- ICE200
- STK500/501/502
- AVRISP

AVR Studio a été très utile pendant les premières phases du projet. Il a permit de tester les librairies de base en conjonction avec une carte de développement STK500.
9.3. PonyProg 2000

PonyProg 2000 (en figure 61) est un simple mais très puissant logiciel qui permet de programmer aisément un nombre impressionnant de mémoires EEPROMs et microcontrôleurs du type AVR et PIC. Le logiciel est disponible pour Windows et Linux et il supporte différents types d’interfaces hardware pour la programmation. Dans le cadre de ce projet, l’interface parallèle de programmation ISP STK300 a été utilisé.
9.4. Interface de programmation ISP

Le laboratoire LSM disposait d’un nombre important de kits STK-300 de développement pour la famille de microcontrôleurs AVR. Il était assez logique d’incorporer sur la carte VersaCAN la même interface de programmation ISP (In Circuit Programming). En plus cette interface était supportée par le logiciel de programmation gratuit PonyProg. Il faut remarquer que Atmel ne donne pas d’information sur les composants utilisés pour cette interface, ils ont même limé le texte d’identification sur le boîtier du circuit intégré. Heureusement sur le site web de PonyProg ils ont fourni le schéma pour la réalisation directe de cette adaptateur (en figure 62).

![Interface de programmation ISP](image)

Figure 62: Interface de programmation ISP

9.5. Peak-System CAN-USB Interface

L’interface CAN de Peak-System (en figure 63) permet une connexion simple et économique du PC à un réseau CAN. Grâce à l’utilisation de la porte USB une alimentation externe du module n’est pas nécessaire. Cet adaptateur USB supporte les spécifications CAN 2.0A et 2.0B avec une vitesse maximale de transmission des données de 1Mbit/s.

Détails techniques:

- Equipé avec le contrôleur CAN SJA1000 de Philips
- Equipé avec le transceiver CAN 82C251 de Philips
- Possibilité d’exécuter un reset hardware depuis le logiciel de contrôle
- Connexion via une porte 9-pôles DIN selon les recommandations CiA DS 102
Avec l'adapteur hardware Peak-System fournit:

- Un driver pour Windows et Linux
- Le logiciel PCANView (en figure 64)
- Des exemples de code source pour implémenter l'interface CAN-USB

PCANView est une version réduite du logiciel PCANExplorer 2.0 de Peak-System qui permet de:

- Visualiser les messages reçus par l'adapteur
- Envoyer des messages à intervalles périodiques
- Envoyer des messages manuellement

Même si les possibilités d'utilisation de ce logiciel ne soient pas très grandes, PCANView c'est démontré amplement suffisante pour développer et tester la carte VersaCAN.
9.6. Borland C++ Builder 5

Borland C++ Builder 5 (en figure 65) est un environnement RAD (Rapid Application Development) pour Windows. Cet environnement gère une bonne partie de l'infrastructure du code à développer (API, gestion d'événement, gestion des objets) en permettant de se concentrer sur les parties essentielles du code.

![Figure 65: Borland C++ Builder 5](image)

L'efficacité des produits Borland avait déjà été testée pendant des projets antécédents et l'utilisation pour ce projet de diplôme a été une confirmation. En partant d'une application d'exemple d'interface USB, donnée en open source par le fabricant du chip USB (FTDI), une application d'exemple a été très rapidement développée de façon aisée.

9.7. MProg 1.0

MProg (en figure 66) est une petite utilitaire développé par FTDI, les fabricants du chip USB FT245. MProg peut programmer l'EEEPROM optionnelle connecté au chip USB. Avec une interface très simple, la sauvegarde des différents réglages du chip USB (vendor ID, product ID …), est très facile. Pour la série BM, MProg est aussi capable de programmer jusqu'à 16 périphériques en même temps. Il faut remarquer que MProg ne peut pas tester la EEPROM, mais seulement notifier si la EEPROM a été programmée.
9.8. Protel 99 SE

Protel 99 SE (en figure 67) a été utilisé comme environnement de design du PCB. Protel dispose d'un puissant éditeur de schématique, une très grande librairie de composants et d'un éditeur de PCB avec un très sophistiqué mécanisme de placement et routage.
10. Conclusions

Les objectifs du projet ont été atteints. Un réseau de carte VersaCAN a été développé et implémenté. Les cartes communiquent sans problèmes et il a été possible aussi connecter ce réseau à un ordinateur standard travers la porte USB établir une connexion.

L’application implémentée pour la démonstration des fonctionnalités du système développé est liée aux capteurs de l’entreprise LEM. Mais le système peut être facilement adapté pour des autres applications, qui ne sont pas nécessairement liés à ce domaine, on peut penser par exemple à la domotique et le contrôle des actuateurs.

Le CAN bus, implémenté dans ce projet, c’est démontré une solution idéale pour des systèmes où il faut envoyer et recevoir des petites quantités de données entre différents nœuds. Quelques caractéristiques: transmission flexible et robuste des messages, coût relativement bas d’implémentation hardware et software, grand choix de solutions hardware.

La solution USB fournie par FTDI: contrôleurs, drivers, applications d’exemple, c’est démontrée très efficace. L’implémentation USB, dans ce cas, a été très ciblée à niveau de l’application finale, étant les détails de communication USB au bas niveau gérés totalement par le contrôleur USB FTDI.
11. Développements futurs

Le prototype développé est fonctionnel, quand même c'est difficile de considérer un projet comme terminé. Plusieurs tâches peuvent encore être accomplis:

- Créer des librairies supplémentaires pour la porte d'expansion, on peut penser à différents types de capteurs, cartes de contrôle d'actuateurs, relais et servomoteurs
- Implémenter la mise à jour du firmware de la carte travers la porte USB
- Implémenter un firmware pour transformer VersaCAN en un adaptateur universel CAN <-> USB (similaire à l'adaptateur de Peak-System)
- Implémenter un logiciel PC permettant de connecter VersaCAN à Internet, en commandant par exemple le réseau CAN par une page Web.

Celles-ci sont seulement quelques propositions données à titre d'exemple. Clairement les possibilités de développements futurs sont vraiment nombreuses.

Il faut remarquer que des questions de temps ont obligé de lancer la production des 10 cartes VersaCAN sans pouvoir produire un PCB d'essai. Des petites erreurs ont été détectées et corrigées à niveau du schématique et du PCB. En annexe les deux versions ont été incluses. S'il y avait l'idée de lancer la production d'autres PCB la version corrigée est vivement conseillée.

12. Remerciements

Dans la réalisation de ce travail de diplôme, je tiendrais à remercier mon assistant Serge Lopez pour l'aide au développement de ce projet et pour la confiance qui m'a donné. Un remerciement aussi à Sylvain Hauser et Georges Vaucher, pour l'aide précieux qui m'ont donné pendant la conception et la réalisation du PCB. Un remerciement aussi à toute l'équipe du laboratoire LSM pour l'accueil chaleureux, j'ai bien aimé pouvoir travailler avec vous.
13. Bibliographie


Lausanne, le 19 février 2004

Davide Rivola
Annexes