Projet TER 2015

Protocole SimpleCAN pour bus à base d'Arduino

Auteur :
Le Forestier Romain

M1 SICLE

Encadrant :
Goulven Guillou

Table des matières

1 Contexte :

1.1 Sujet du TER:

Le projet a pour objectif de définir une sur couche du protocole CAN, appelée SimpleCAN, pour faire communiquer entre eux un ensemble de nœuds composés de cartes Arduino associées à une interface CAN. L'approche devra être validée, au travers d'une application simple de visualisation et d'envoi de données via un PC, sur un matériel existant consistant en 5 nœuds : un permettant une connexion PC via USB, un destiné au décodage de données propriétaires SeaTalk (Raymarine), deux destinés à du traitement de données NMEA (issues d'un GPS et d'une centrale Tactick) et le dernier portant une centrale inertielle.

1.2 Objectif final

L'objectif du TER a été modifié, il consistait en un premier temps à évaluer l'intérêt du protocole SimpleCAN par rapport au protocole CAN standard et comparer la difficulté de mise en œuvre de ce protocole par rapport à sa mise en œuvre. Dans un second temps, il consistait à réutiliser 5 cartes Arduino et les faire communiquer via leur bus CAN intégré. Une des cartes servira d'interface USB entre le bus CAN et le pc pour récupérer les données des capteurs du réseau.

Le but de départ était d'essayer d'utiliser les travaux réaliser par Kévin Bruget, étudiant en 2013 à l'ENSTA Bretagne, pour son son projet de fin d'études pour lequel il avait étudié la possibilité de remplacer un système d'assistance à la navigation centraliser par un système réparti via un réseau CAN à base de carte Arduino.

Le projet c'est déroulé de la façon suivante, prise en main des documents fournit, le rapport du projet présentant le protocole SimpleCAN, puis la prise en main des cartes Arduino, la sélection d'une librairie CAN pour mettre en réseaux les cartes et débogage de leur fonctionnement.

Le but final était d'avoir un réseaux CAN fonctionnelle permettant de récupérer sur la carte interface USB les données des autres capteurs de façon simple et d'expliquer le fonctionnement des cartes, la méthode pour les programmés et de proposer des méthodes pour récupérer les données fournit par les capteurs.

Les codes de test pour les cartes et certain document utiliser lors du TER sont disponibles sur un [dépôt github].

2 Le protocole CAN

2.1 Historique du protocole CAN

Le protocole CAN(Controller Area Network) est un bus de terrain développer à l'origine pour le secteur de l'automobile par Bosch et l'Université de Karlsruhe dans les années 1980 et standardisé par les standards ISO au début des années 1990.

Le Développement de ce bus avait pour but de simplifier le câblage et réduire le nombre de câbles dans les voitures qui dans les années 1980 se complexifiait, principalement à cause de l'accroissement du nombre de capteur et de système électronique afin de permettre aux voitures de répondre aux exigences environnementales, de sécurité (ABS, ESP, AIR-BAG...) et une demande de confort (climatisation automatique, système de

navigation …).

En 1992 plusieurs entreprises se sont réunies pour former CAN in Automation(CIA), une organisation à but non lucratif dont l'objectif est de fournir des informations techniques et promouvoir le protocole CAN. Actuellement, 560 entreprises sont membres de cette organisation.

Le protocole CAN c'est répandu du domaine du transport pour lequel il fut développé au domaine industriel (automate, gestion de la génération de courant...), dans le bâtiment (ascenseur, porte automatique, air conditionné...), l'agriculture (machine de traite, gestion de l'alimentation des bêtes...), le domaine médical, la communication, le domaine financier(caisse enregistreuse, ATM...), le domaine du spectacle (rampe d'éclairage,robot), le domaine scientifique (équipement de laboratoire, télescope).

2.2 Principe de fonctionnement

La norme OSI qui définit un modèle basique pour l'interconnexion des systèmes ouverts divisés en services (couches) qui sont délimités par des notions de services, de protocole et d'interface, le modèle OSI est composé de 7 couches : la couche physique (1), la couche de liaison (2), la couche réseau (3), la couche de transport (4), la couche de session (5), la couche de présentation (6), la couche application (7). Le protocole CAN n'utilise que les couches physiques, de liaison et d'application.

2.2.1 Caractéristique physique du bus CAN

Le bus CAN est un bus de données série bidirectionnelle en half-duplex, ce qui signifie que la communication peut se faire dans les deux sens sur le bus, mais qu'un seul des nœuds connectés peut émettre à la fois. Le support physique est constitué de 2 fils différentiels torsadés, les fils sont dénotés CAN_L pour CAN LOW et CAN_H pour CAN HIGH. Le bus CAN étant un bus de terrain pouvant être soumis à des parasites importants, le montage différentiel permet de gommé ces perturbations, car les 2 câbles sont soumit a la même perturbation, la différence de potentiel entre les 2 câbles ne change pas voir Illustration 1. L'accès au bus suit la technique CSMA/CD se qui signifie que chaque nœud écoute le bus avant d'émettre, mais il n'y a pas de tour de parole sur le câble, la résolution des collisions est réaliser un arbitrage sur la priorité des messages.

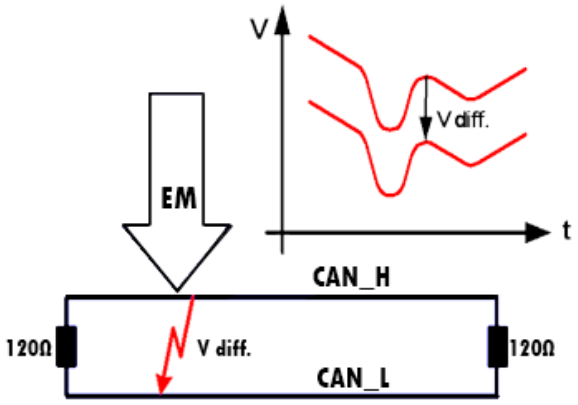

Illustration 1: conservation de la différence de potentiel quand le réseau est soumis à une perturbation électromagnétique

Il existe 2 types de câblage possible pour le protocole CAN :

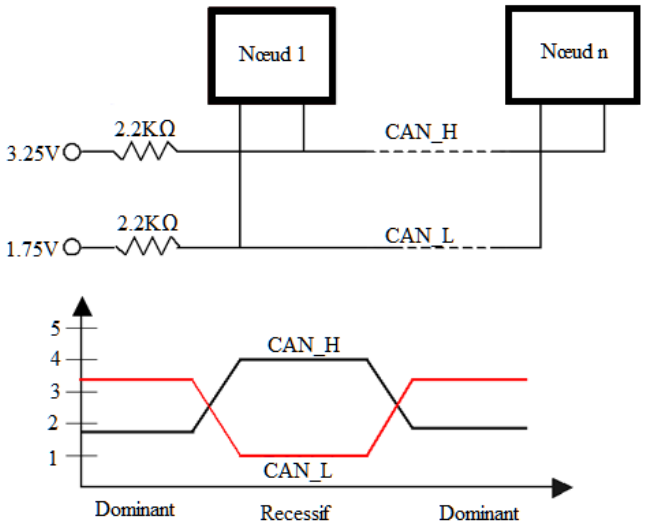

Illustration 2: ISO11898-3

Low Speed CAN < 125kb/s

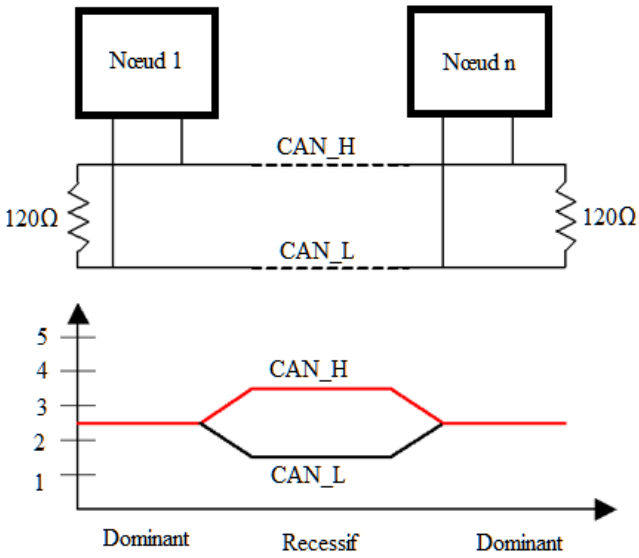

Illustration 3: ISO11898

High Speed CAN 125Kbps - 1Mb/s

La norme pour le bus ISO 11898-3 a été révisée en 2006, elle correspondait à la norme ISO 11519-2.

Les nœuds sont câblés sur le bus de façon à effectué des opérations logiques de type "ET", ce qui se correspond en cas d'émission simultanée sur le bus que la valeur 0 écrasera la valeur 1, ce qui se donne dans la terminologie CAN :

- l'état logique 0 est l'état dominant

- l'état logique 1 est l'état récessif

La longueur maximale du bus est déterminée par la vitesse de transmission utilisée :

Longueur (m)	30	50	100	250	500	1000	2500	50000
Vitesse (kb/s)	1000	800	500	250	125	62,5	20	10

2.2.2 Protocole CAN

Comme dit plus haut, le principe de communication du bus CAN est celui de la diffusion d'information(broadcast), chaque station ou nœud connectés écoute les trames transmises par les autres stations émettrices, les nœuds décident ensuite se qu'ils doivent faire de l'information reçue selon les filtres ou le programme que la station contient.

Le protocole autorise plusieurs stations a accéder au bus en même temps, c'est ensuite un procéder d'arbitrage binaire qui permet de déterminer quelle station pourra émettre sa trame.

Le moment auquel une trame sera transmise est donc aléatoire, car les émissions sur le bus doivent respecter un ordre de priorité strict qui est défini par un identifiant dans chaque message. Les priorités des messages sont définies lors de la mise en place des

stations et ne peut pas être changé dynamiquement, l'identifiant avec la plus petite valeur binaire est le plus prioritaire.

Les conflits d'accès sont résolus par un arbitrage binaire à partir des identifiants des messages envoyés par chaque station, chaque station compare son identifiant d'envoi à celui qui est transmis sur le bus et détermine s'il gagne l'arbitrage. Cela se déroule grâce au mécanisme câblé avec lequel l'état dominant écrase les états récessifs, chaque station avec une transmission récessive et une observation dominante perd l'arbitrage pour l'accès au bus. Chaque station qui perd l'arbitrage passe en mode réception pour écouter le message avec une priorité plus élevé et ne tente plus d'accéder au bus jusqu'à ce qu'il soit à nouveau disponible.

Les envois de requêtes sont gérés par ordre d'importance par le système dans leur ensemble. Ce qui est particulièrement utile dans les situations de surcharge de réseaux, car l'accès au bus étant géré par ordre de priorité des messages, il est possible de garantir une latence faible pour les systèmes temps réel, car les messages importants ne sont pas ralentis par la surcharge.

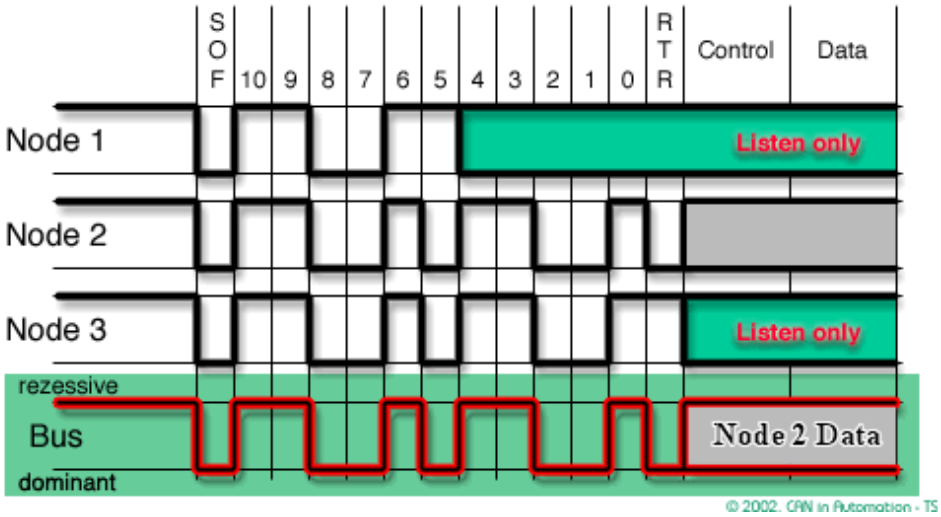

Illustration 4: Arbitrage des stations pour déterminer celle qui peut envoyer son message sur le bus

	Start Bit	Identificateur											RTR	Champ Contrôle	Données
		10	9	8	7	6	5	4	3	2	1	0			
Station 1	0	1	1	0	0	1	1	Écoute du bus							
Station 2	0	1	1	0	0	1	0	1	1	0	0	1	0	X	X
Station 3	0	1	1	0	0	1	0	1	1	0	0	1	1	Écoute du bus	
Signal sur le Bus	0	1	1	0	0	1	0	1	1	0	0	1	0	X	X

Le champ d'arbitrage débute par le bit SOF qui est un signal dominant qui informe toutes les stations du début d'une trame. Le champ d'identification est composé de 11bits d'identification pour CAN 2.0A (format standard) et de 29bits pour CAN 2.0B(format étendu).

Le champ RTR("Remote Transmission Request") permet de différencier les trames de donnée codée avec un bit dominant des trames de requête codées avec un bit récessif. Le champ RTR est codé pour les requêtes en récessif afin que si une donnée est

demandée avec un identifiant et que cet identifiant est émis en même temps, le nœud qui réclamait la donné la récupère immédiatement.

Le champ d'identification permet donc en mode standard de codé 2^{11} soit 2048 combinaisons de messages différents, et en mode étendu 2^{29} soit 536 870 912 combinaisons, le mode étendu est donc plus utile sur un réseau composer d'un grand nombre de nœuds, alors que les trames standard seront suffisantes pour de petits réseaux.

Illustration 5: Composition d'une trame CAN

La trame complète du protocole CAN est donc découpée en différents champs :

Nom du champ		Taille (bit)	
SOF		1	Indique le début d'une trame, dominant (0)
Champ d'arbitrage	Identifiant	11 | 29	Identifiant de message unique, permet de déterminer la priorité du message
	RTR	1	Doit être dominant (0) pour les trames de données et récessif (1) pour les trames de requêtes.
Champ de contrôle	Identifiant d'extension	1	Doit être dominant (0) pour les trames standard et récessives (1) pour les trames étendues.
	Champ réservé	1	bit réservé, non utilisé, doit être dominant (0)
	DLC	4	(Data Lenght Code) indique le nombre d'octets dans le champ data (0-8 octets)
Champ de données		0-64	Les données qui doivent être transmises.
Champs CRC	CRC	15	Contrôle de Redondance cyclique
	Délimiteur du CRC	1	Le bit de délimitation qui est toujours récessif (1)
Champ ACK	ACK	1	Le bit d'acquittement est toujours récessif (1) pour l'émetteur
	ACK délimiteur	1	Le bit de délimitation de ACK, récessif(1)
EOF		7	Indique la fin de la trame, tous les bits sont récessifs (1)

Le champ CRC permet de vérifier si les données transmises sont correctes, il est calculer à partir de l'ensemble des données émises avant le champ CRC, c'est-à-dire le champ SOF, le champ d'arbitrage, le champ de commande et le champ de données. Il est calculé en divisant 2^{15} par la somme des bits du message $x^{15}+x^{14}+x^{10}+x^{8}+x^{7}+x^{4}+x^{3}+1$, le reste de cette division donne la valeur du champ CRC.

La correction d'erreur de l'algorithme est basée sur la distance de Hamming, qui permet

de quantifier la différence entre deux séquences de symboles de même longueur en associant le nombre de positions où les deux suites diffèrent. La distance de Hamming pour cet algorithme est de 6, ce qui signifie que jusqu'à 5 erreurs peuvent être détectées. Grâce à ce système de détection d'erreur, le taux d'erreur moyen enregistré est très faible (inférieur a $4{,}6.10^{-11}$).

Le champ ACK permet au nœud relier au réseau d'indiqué à l'émetteur qu'au moins une station à reçus le message, si une station n'a pas reçus ou mal reçus le message, elle doit envoyer un message d'erreur.

Une trame de requête comporte un champ de moins qu'une trame de donnée, car elle n'envoie pas de données.

Entre chaque trame, il doit y avoir un espace équivalent à 3 bits récessifs, appeler espace inter trame, il permet de séparer les trames normales des trames d'erreurs et de surcharge, car elles ne sont pas précédées de ces espaces.

Les trames d'erreurs sont constituées de 2 champs, le premier champ est donné par la superposition d'ERROR FLAGS (6-10 bits dominants/récessifs) envoyer par plusieurs nœuds. Le second champ est le délimiteur d'erreur composé de 8 bits récessifs.

La trame d'erreur peut être envoyé dès qu'une erreur est détectée par le système, ce qui interrompt le message envoyé et évite que certaines stations acceptent le message, cela garantir la consistance des données dans le réseau. Après l'envoi d'une trame d'erreur, le nœud qui envoyait le message essaye de le réémettre de façon automatique. Les nœuds peuvent alors tenter de gagner le contrôle du bus, ce qui empêche qu'un message non prioritaire bloque un message prioritaire avec la réémission de sa trame qui a provoqué un message d'erreur.

La détection d'erreur se fait au niveau binaire (la couche 1 pour le modèle OSI) par 2 principes :

- La surveillance qui consiste au suivi par chaque station du bus des données circulant sur le bus, l'émetteur observe aussi le bus ce qui lui permet de détecter les différences entre les bits envoyer et ceux reçus et lui permet de déterminer si l'erreur est locale ou globale.

- L'ajout de bit, les bits envoyer par CAN utilise la méthode NRZ, soit pas de retour a zéros entre les bits, pour permettre une synchronisation et éviter qu'un message erroné par une perturbation qui forcerait le bus à un seul état, après l'envoi de 5 bits identique consécutif, l'émetteur ajoute un bit de remplissage dans le flot binaire. Ce bit de remplissage est le complémentaire du bit précédent, ce bit est ensuite supprimé par les receveurs.

Si une erreur est détectée via l'un de ces deux mécanismes par l'une des stations qui écoutent, une trame d'erreur est envoyée. Pour éviter qu'une station ne sature le bus avec des trames erronées, le protocole fait la distinction entre les erreurs sporadiques et les erreurs récurrentes provoquées par une station. Cette distinction a pour but de bloquer une station défectueuse pour l'empêcher de nuire au réseau. La distinction se fait en comptant les erreurs, via deux compteurs, le compteur TEC qui compte les erreurs à la transmission et le compteur REC qui compte les erreurs de réceptions. Selon la valeur de ces compteurs, le nœud change de mode d'émission :

-le mode d'erreur active, tant que les compteurs sont inférieurs à 127

- le mode d'erreur passive quand l'un des compteurs est entre 128 et 255

- le mode bus off quand l'un des compteurs est supérieur à 255, le nœud se déconnecte alors du bus.

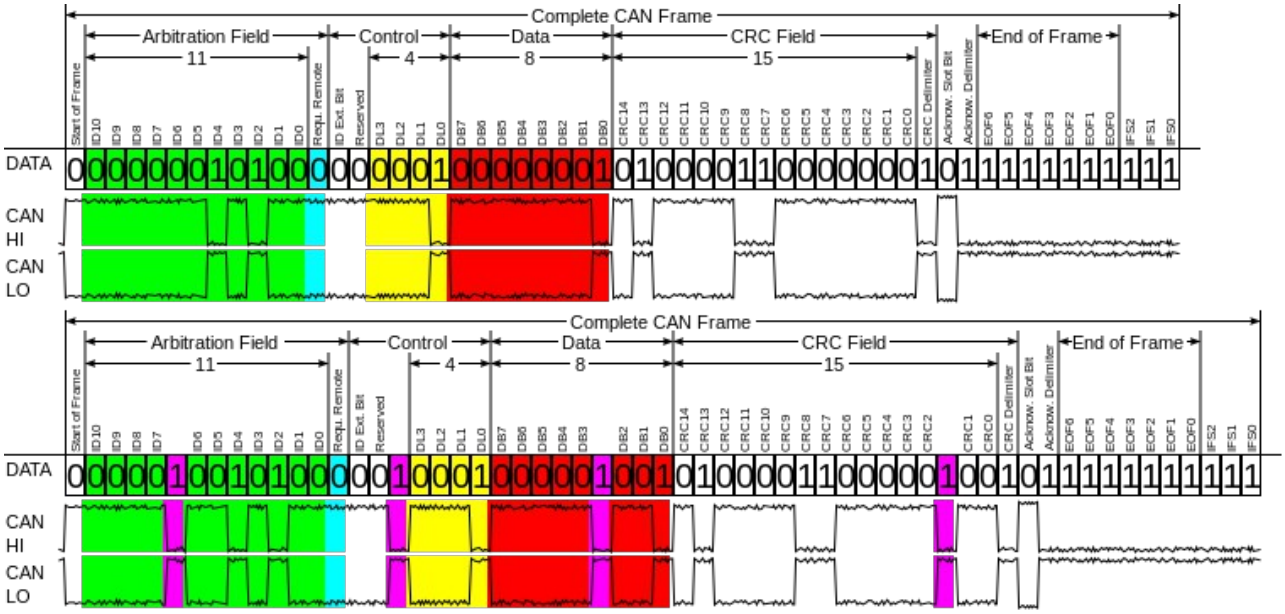

Illustration 6: exemple de tram avec les bits de remplissage en violet

Les trames de surcharge permettent aux nœuds de demander un délai avant la réception d'une nouvelle trame, elles sont envoyées dans 2 cas, quand le nœud veut demander un délai ou quand le nœud détecte un bit dominant pendant une séquence d'intertrame ce qui signifie qu'un autre nœud demande un délai. La trame de surcharge est composée de 2 parties, le drapeau de surcharge composée de 6 bits dominants, pouvant aller jusqu'à 12 bits et un délimiteur composer de 8bits récessif.

3 Les cartes utilisées pour le réseau

Le projet est basé sur un réseau de carte Arduino, les cartes Arduino étant génériques, peu coûteuses et simples à programmer.

Arduino est une plateforme open source au niveau matériel et logiciel, les cartes sont reprogrammable.

3.1 Description de la carte

Les cartes utilisées pour le projet sont basées sur l'architecture et le processeur des cartes Leonardo et intègre sur la même carte l'interface CAN basée sur une puce MCP2515 interfacer sur l'interface SPI de la carte Arduino et d'une puce émetteur-récepteur MCP 2551. La carte possède les mêmes connecteurs qu'une carte Leonardo classique avec un connecteur 5 broches pour le bus CAN. La carte utilisée permet de réduire l'espace occupé par la carte et simplifier ses connexions, car il faut utiliser un shield pour avoir un bus CAN sur la carte standard, ce dernier étant de la même taille que la carte Arduino.

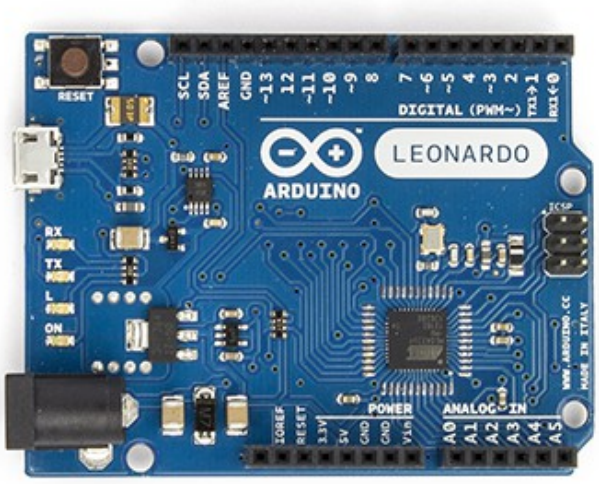

Illustration 7: Une carte Leonardo

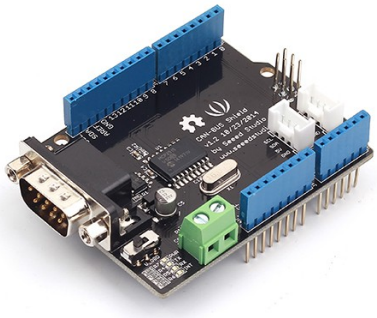

Illustration 8: Un shield CAN pour carte Arduino

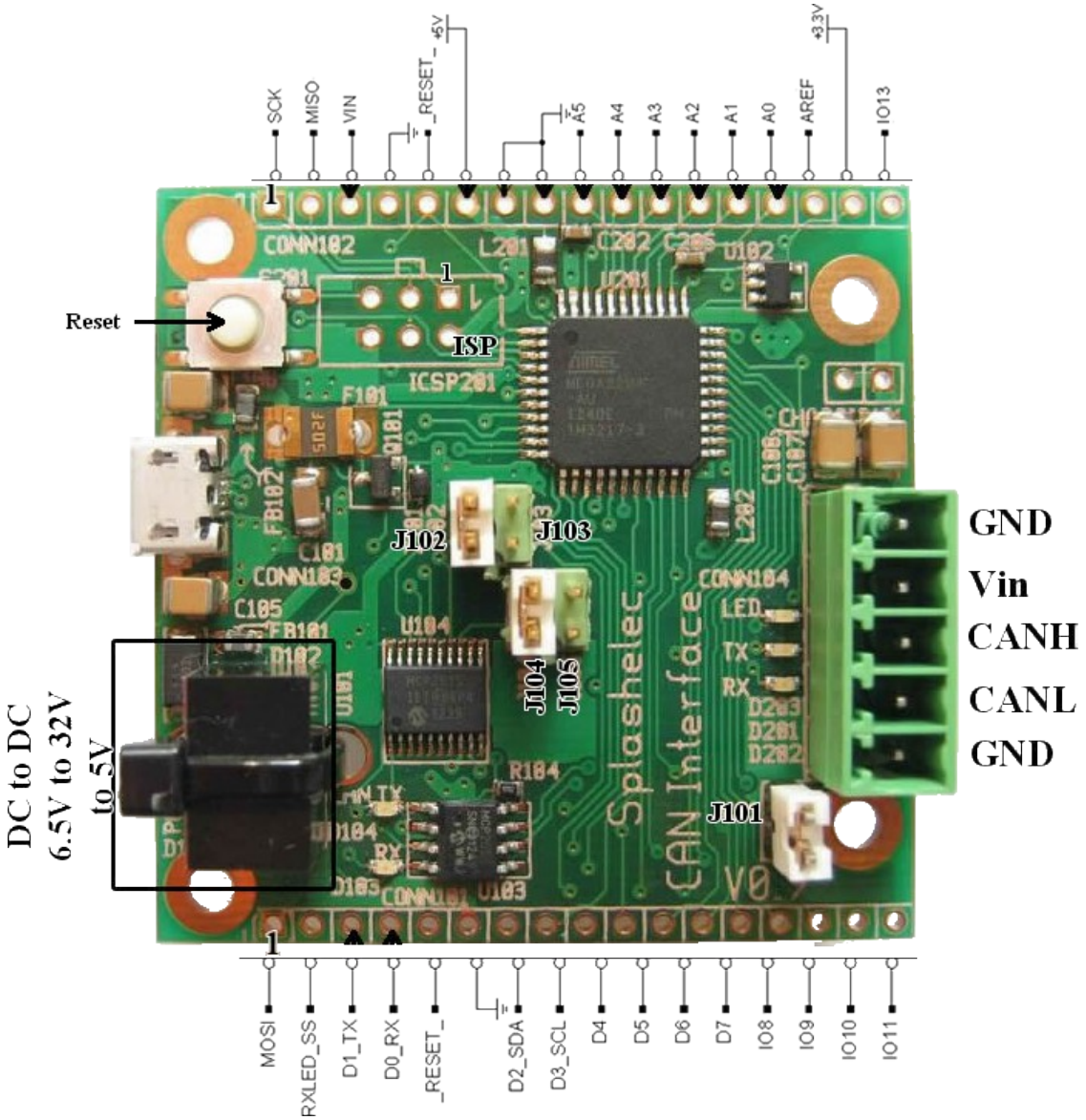

Illustration 9: La carte de base pour chaque nœud du réseau composer d'un microcontroleur identique a une carte Leonardo et de Puce CAN MCP2515 et MCP2551 que l'on retrouve sur les shield CAN

Les cartes utilisées pour le réseau permettent d'utiliser le logiciel Arduino pour les reprogrammer avec un bootloader spécifique et respectent la même dénomination pour les connecteurs de la carte que la carte Leonardo originale, ce qui permet l'utilisation de code d'exemple pour Leonardo ce qui permet de tester le fonctionnement des cartes et permet une certaine compatibilité avec les librairies déjà existantes.

La carte comporte plusieurs jumpers (connecteur) qui permettent de changer les ports auxquels le contrôleur CAN est relié. Ils permettent de changer les ports sur lesquels les entrées CS et INT du contrôleur CAN sont reliées.

L'entrée CS peut donc être connectée au port IO 9 de la carte (9 dans le logiciel Arduino) via le pont 102 (J102 sur la carte) soit au port RXLED_SS de la carte via le pont 103 (J103).

L'entrée INT peut être connectée au port D1_TX de la carte (1 dans le logiciel Arduino) via le pont 104 (J104) soit au port D3_SCL de la carte (3 dans le logiciel Arduino) via le

pont 105 (J105).

La carte comporte une résistance de 120Ω pour fermer la boucle de bus CAN, 2 nœuds du réseau doivent donc avoir le pont 101 (J101) de mis et les autres cartes du bus doivent l'avoir ouvert afin de réaliser la boucle que forme le bus CAN.

L'alimentation est sélectionnée de façon automatique entre une alimentation externe qui peut aller de 6,5V à 32V ou une alimentation stabilisée via le port micro USB.
Les ports d'alimentations:
- Vin: le port d'alimentation externe de la carte quand la source est une batterie
- 5V: L'alimentation réguler de la carte, on peut alimenter la carte en 5V via le port USB ou avec une alimentation stabilisée de 5V sur ce port, ou récupérer l'alimentation stabilisée de la carte.
- GND: la terre.

Le connecteur Vin et GND sont présent sur le connecteur du bus CAN, car l'alimentation des cartes se fera, en situation, par le biais du câble du bus CAN sur une batterie de bateaux en 12V.

La carte utilisée comporte deux bus série réelle, le bus série relier au port USB qui est appelé dans le logiciel Arduino Serial et le bus qui correspond au connecteur D0_RX et D1_TX de la carte qui est appelée Serial1 dans le logiciel Arduino, ces ports série support une connexion pouvant allé jusqu'à 115200 Bauds, les autres ports peuvent être utilisé en tant que port série via une librairie Arduino, mais à une vitesse moindre.

3.2 Programmation de la carte

Certaine carte ne peuvent pas êtres reprogrammer via le port USB, car elles ont un bootloader pour se faire reprogrammer via le bus CAN avec un script python et le protocole SimpleCAN. Pour reprogrammer les cartes, on utilisera le connecteur ICSP qui utilise le protocole ISP.

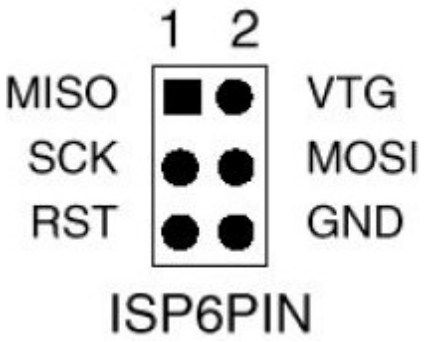

Illustration 10: Correspondance des connecteurs Arduino au connecteur ICSP

Pour reprogrammer la carte on peut utiliser un programmateur dédier, par exemple celui utiliser lors du projet, le programmateur USBtinyISP, qu'il faut sélectionner dans la liste des programmateurs dans le logiciel Arduino.

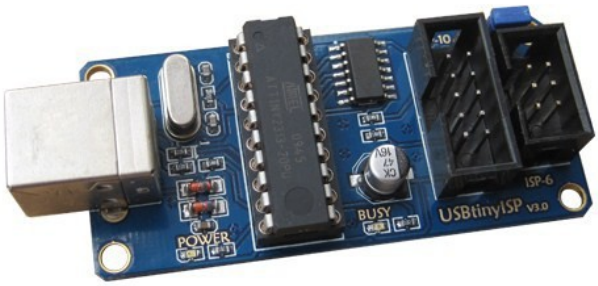

Illustration 11: Le programmateur USBtinyISP

On peut aussi utiliser une autre carte Arduino, Arduino UNO ou Mega, avec un programme fourni en exemple pour reprogrammer la carte.

Le sketch, le code pour la carte Arduino pour l'utiliser comme un programmateur ISP est disponible dans la librairie d'exemple de l'application (Fichier→Exemples→ArduinoISP), le câblage pour les cartes UNO et Mega est indiqué en commentaire au début du fichier, il faut téléverser le programme dans la carte que l'on veut utiliser en programmateur, comme indiqué dans le fichier il est conseiller d'avoir des DEL(Diode Électro-Luminescente) pour voir le fonctionnement du programme et si la programmation de la carte connecter à la carte utilisée en programmateur se déroule correctement.

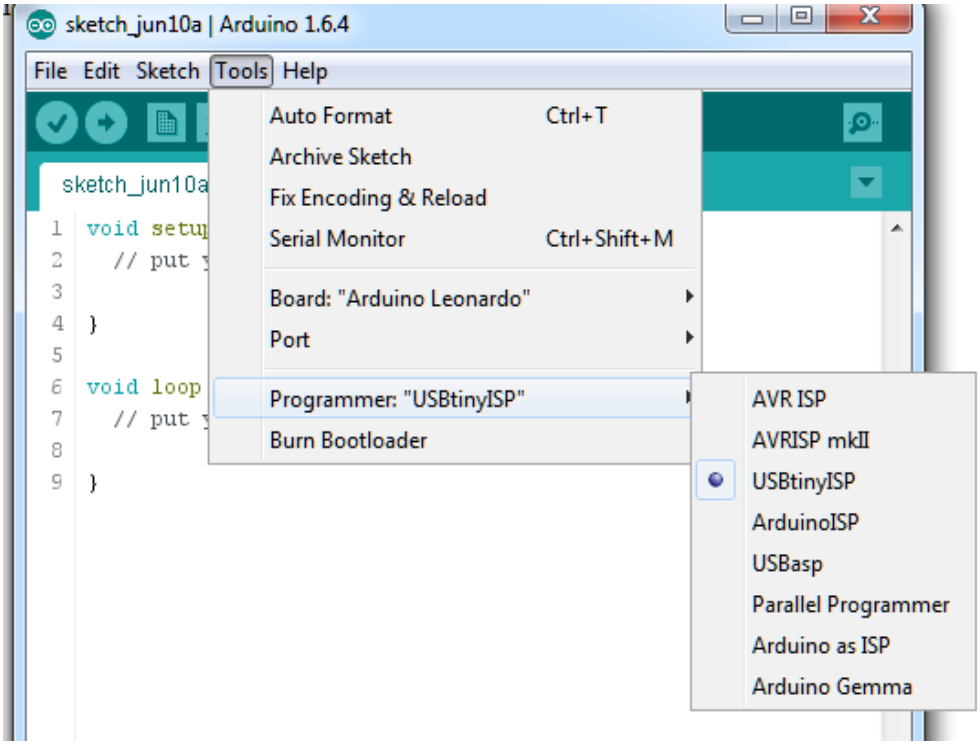

Illustration 12: Le logiciel Arduino et la sélection du programmateur

Si l'on utilise une carte Arduino comme programmateur il faut que le port série cette carte soit sélectionnée dans le logiciel et non celui de la carte que l'on veut programmer. Il faut aussi que l'architecture de la carte à programmer soit sélectionnée. Le programmateur USBtinyISP n'utilise pas de port COM, le logiciel ne tient pas compte du port sélectionné lors de la programmation d'une carte avec ce programmateur, mais tient compte de l'architecture de la carte qui est sélectionnée.

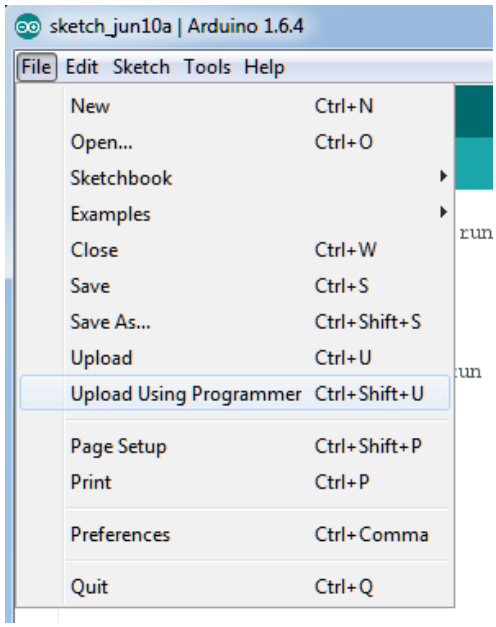

Illustration 13: le menu du programme Arduino, pour envoyer un programme sur une car via un programmateur il faut utiliser "upload using Programmer" ou "téléverser en utilisant un programmateur"

La programmation via une autre carte Arduino peut être plus complexe, car il faut être attentif à câbler correctement la carte qui va servir à programmer et le port ISCP de la carte que l'on veut programmer.

4 Programmation et librairie CAN

4.1 Le protocole SimpleCAN

4.1.1 Le protocole

Le protocole SimpleCAN a été développé par Kévin Burget pour sont projet de fin d'études, le but de se protocole était de rajouter une surcouche logicielle à CAN pour apporter par rapport a CAN une synchronisation, la gestion de leader (nœud maître), la gestion avancer des filtres.

Il existait plusieurs surcouche pour le protocole CAN tel que CANopen, CanKingdom, DeviceNet, CCP/XCP, J1939 et d'autres protocoles propriétaire. Cependant pour son projet, ces protocoles ne convenaient pas, le protocole compatible avec les processeurs AVR qui corresponde au processeur des cartes Arduino nécessite des microcontrôleurs intégrés comme l'ACT90CAN, ce qui n'est pas le cas de la carte utiliser dans le projet, les autres protocoles nécessitent un OS ou un système 32bits pour fonctionner, ce qui n'est pas le cas ici, ce sont des processeurs 8bits qui sont utiliser. Pour répondre aux besoins de son projet, compatibilité avec la carte et code open source, Kévin Bruget à donc du développer sont propre protocole.

4.1.2 La synchronisation

L'ajout de la synchronisation pour les messages CAN répondait à un besoin spécifique à sont projet, en effet, il avait besoin de récupérer des données de capteur sur une base précise et de contrôler des actionneurs de façon maîtriser et synchrone, se que ne garantit pas spécifiquement CAN. Pour ce faire il synchronise toutes les cartes toutes les 100ms.

La synchronisation se fait par le biais d'un nœud leader(maître), ce dernier envoi une trame de synchronisation, se qui permet au nœud esclave de se synchroniser sur son horloge, ce système permet aussi de garantir un timing fixe pour l'envoi des messages et que chaque nœud reçoit et envoi les donner au même moment. Cela évite qu'une carte avec un programme moins complexe n'émette plus souvent qu'une autre carte du réseau avec un programme plus complexe.

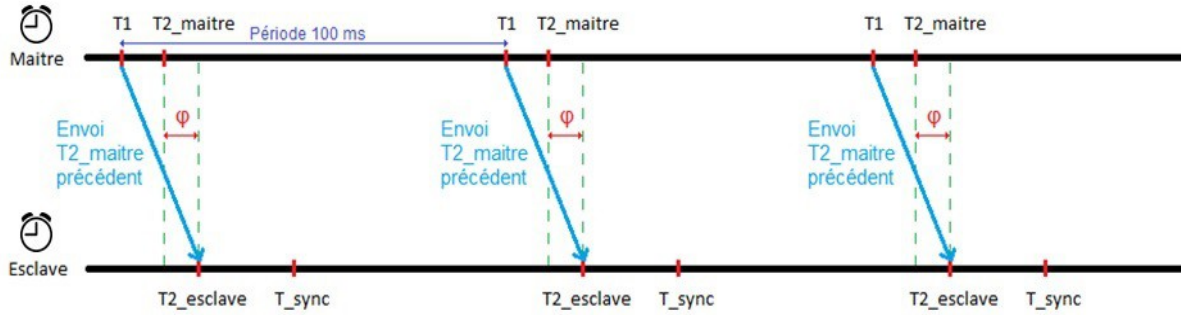

Illustration 14: La synchronisation des nœuds se fait à l'aide d'un message spécifique directement envoyer par le maître à l'instant T1, contenant la mesure T 2maitre de la période précédente. L'objectif est de faire coïncider T 2esclave avec à l'instant Tsync `a l'aide d'un régulateur PID.

Ce système est pratique pour garantir la synchronisation du système et garantir l'envoi et la réception de message dans un intervalle de temps donnés pour les contrôleurs.

Le réseau supporte très bien les cas de défaillances d'un nœud esclave, mais peut s'effondrer en cas de perte du nœud maître.

4.1.3 Élection du leader

Le nœud maître étant indispensable au bon fonctionnement du réseau et pour sa synchronisation, le protocole vise à garantir l'élection d'un nœud maître à l'initialisation du réseau et en cas de perte du nœud maître. L'élection du nœud maître est basée sur l'identifiant de la carte, ce dernier étant fixé a la programmation de la carte.

Le nœud avec l'identifiant le plus faible du réseau est désigner leader du réseau, pour ce faire chaque nœud du réseau émet sont identifiant sur le réseau et le compare a celui qu'il reçoit, s'il a l'identifiant le plus faible, il détermine qu'il est le maître, sinon il détermine qu'il est un esclave.

Ce système d'élection est très gourmand en taux d'occupation du bus, chaque nœud émettant son identifiant, cela sature le bus. Le nœud maître n'étant pas statique, cela permet en cas de détection d'une défaillance du nœud maître d'en élire un autre, ce qui permet de garantir la continuité du réseau, mais peut provoquer la perte de certains paquets.

4.1.4 Filtrage du Bus

Pour ne pas saturer le microcontrôleur utilisé, on peut utiliser les filtres disponibles sur

la puce CAN MCP2515. Les buffers de réception de la puce CAN dispose d'un masque et de 6 filtres qui permettent d'ignorer les messages qui circulent sur le bus s'il ne correspond pas au filtrage appliqué.

Mask Bit n	Filter Bit n	Message Identifier bit	Accept or Reject bit n
0	X	X	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

Note: X = don't care

Illustration 15: Principe d'utilisation du masque et des filtres pour le MCP2515

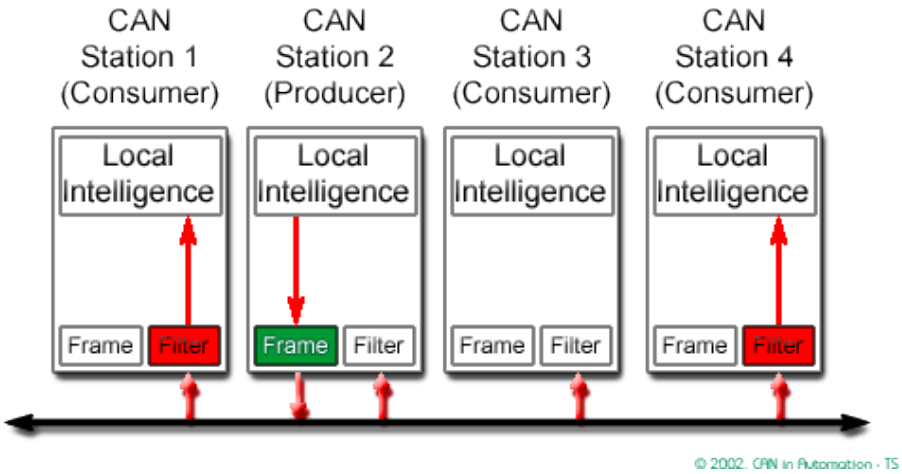

Illustration 16: Principe du filtrage de message.

SimpleCan apport un support de ces filtres, et permet de déterminer des plages de messages accepter ce qui évite de limiter à 6 messages différents par nœud, ce filtrage s'effectuant sur l'identifiant des messages de la trame CAN.

4.2 Librairie CAN

Le protocole SimpleCAN étant relativement complexe et la plupart des ajouts proposer ne correspondant pas au problème de ce TER, nous avons préférer utiliser le protocole CAN sans surcouche.

4.2.1 Choix d'une librairie

Plusieurs librairies CAN pour la puce MCP2515 existent pour Arduino, elles sont basées sur la librairie de kreatives-chaos.com [Universelle CAN Bibliothek], nous avons principalement comparé 3 librairies CAN :

- Universal CAN library for AVRs supporting AT90CAN, MCP2515 and SJA1000

- Multiplatform CAN library for Arduino supporting the MCP2515, SAM3X, and K2X controllers

- CAN BUS Shield

La première librairie avait servi de base pour le projet de Kévin Bruget, nous avons donc commencé par l'étudier, nous n'avons pas retenu cette librairie, car cette dernière n'avait pas été mise à jour

depuis février 2014 et que son utilisation était relativement complexe.

La seconde librairie était plus intéressante dans le sens où les fonctions étaient relativement simples à comprendre avec des fonctions read et write pour la gestion des messages. Cependant pour rendre compatible la librairie à la carte utilise, cela nécessitait de modifier le fichier source et la librairie n'avait pas été mise a jour depuis le 20 juin 2014.

La troisième librairie correspondait à une carte d'extension CAN pour Arduino dont les cartes utilisées pour le projet sont inspirer, la librairie était maintenue à jour et le dernier comit datant du 30 juin 2015, cette librairie comportant des fonctions de message simple (sendMsgBuf et readMsgBuf) et le choix des connecteurs étant modifiable dans le fichier de code de l'Arduino sans modifier la librairie nous avons décider de l'utiliser.

4.2.2 Librairie CAN BUS SHIELD

Cette librairie est relativement simple d'utilisation et permet d'utiliser le masque et les filtres disponibles sur la puce MCP2515 et comporte plusieurs exemples d'utilisation de ses fonctions, ce qui permet une prise en main rapide. La librairie utilise l'interface SPI(Serial Peripheriacal Interface) de l'Arduino ce qui lui permet de grandes vitesses de transfert entre le microcontrôleur et la puce MCP2515.

Cette librairie est non bloquante, l'Arduino ne sera pas bloqué si un message ne part pas ou si elle attend un message. Elle peut gérer les interruptions pour les messages entrant en utilisant les interruptions du programme Arduino.

Les fonctions disponibles sont :

```
MCP_CAN CAN(SPI_CS_PIN);
```

Création de l'objet CAN utiliser par la librairie, elle permet de définir le connecteur Arduino sur lequel la carte est connectée, le 9 dans notre cas.

```
CAN.begin(CAN_500KBPS)
```

Lancement de la puce MCP2515, plusieurs vitesses de transfert sont disponibles de 5kb/s à 1000kb/s, si la fonction se termine correctement, elle renvoie `CAN_OK`

```
CAN.init_Mask(unsigned char num, unsigned char ext, unsigned char ulData);
```

Initialise le masque de réception, il y en a 2 sur la puce MCP2515.

```
CAN.init_Filt(unsigned char num, unsigned char ext, unsigned char ulData);
```

Initialise les filtres du contrôleur.

- num : représente le registre sur lequel le masque sera appliqué les valeurs possibles sont 0 ou 1 pour le masque et 0 a 5 pour les filtres.

- ext : représente le type de trame, 0 pour les trames standard et 1 pour les trames étendues.

- ulData : représente le masque binaire.

```
CAN.checkReceive()
```

Teste si un des buffers à reçus une trame, retourne 1 si une trame est arrivée (`CAN_MSGAVAIL`), 0 sinon.

```
CAN.getCanId()
```

Récupère l'identifiant du message envoyer.

```
CAN.sendMsgBuf(INT8U id, INT8U ext, INT8U len, data_buf);
```

Envoie une trame sur le bus.

- id : représente l'identifiant du message.

- ext : représente le type de trame, 0 pour une trame standard et 1 pour une trame étendue.

- len : représente la longueur de la partie donnée, en octet, envoyé.

- data_buf : les données envoyées dans la trame, maximum 8 octets (char ou unsigned char).

```
CAN.readMsgBuf(unsigned char *len, unsigned char *buf);
```

Récupère les données disponibles sur le bus et les enregistres dans un tableau.

- len : retourne la taille en octet des données récupérer.

- buf:le tableau dans lequel les données sont stockées.

D'autres fonctions sont disponibles :

```
CAN.isRemoteRequest();
```

Teste si la trame reçue est une requête

```
CAN.isExtendedFrame();
```

Teste si la trame reçue est une trame étendue.

Ces fonctions retournent 0 pour négatif et 1 pour positif.

4.3 Les capteurs utilisés

Lors du projet nous avons eu le temps d'étudier 2 capteurs, le capteur NMEA pour les données GPS et le capteur UM6.

4.3.1 Le capteur NMEA

Le capteur que nous avons utilisé est un capteur GPS qui émet sur un port série un signal NMEA.

Le NMEA ou NMEA 183 est une spécification de communication fonder en 1957 par un groupe de vendeur de matériel électronique pour garantir une certaine compatibilité entre leur équipement. NMEA est le sigle de "National Marine Electronics Association".

Les trames NMEA sont composées de plusieurs champs séparer par des virgules, chaque trame commence par un identifiant dont le premier caractère est toujours $, suivit d'un identifiant du récepteur composer de 2 lettres :

- GP pour le GPS,

- GL pour le système GLONASS équivalent russe du GPS,

- LC pour les récepteurs Loran-C qui est un système de radio navigation,

- OM pour les récepteurs de navigation de type oméga qui est un système de radio navigation par triangulation de la distance des différentes ondes reçues émise par les stations OMEGA dont la position est connue,

- II pour instrument intégrer, par exemple AutoHelm de Seatalk système.

Certain fabricant propriétaire utilise leur propre identifiant en indiquant P pour propriétaire suivit d'un code de 3 lettres pour identifier le fabricant, par exemple Garmin donne $PGRM.

Le reste identifie la trame, pour le GPS, on trouve donc entre autres les trames :

- GGA pour GPS fixe date

- GLL pour la position géographique latitude, longitude

- GSA pour les données de précision
- GSV pour la liste des satellites en vue
- VTG pour le cap par rapport au plan terrestre et la vitesse au sol
- RMC pour les données minimales recommandées de spécification GPS.

La taille d'une trame NMEA ne peut pas dépasser 80 caractères.

Pour le projet nous nous sommes principalement intéressés à la trame $GPRMC, car elle permet de déterminer sa position et de connaître sa vitesse.

$GPRMC,hhmmss.ss,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,ddmmyy,x.x,a,m*hh
Field #
1 = UTC time of fix
2 = Data statuts (A=Valid position, V=navigation receiver warning)
3 = Latitude of fix
4 = N or S of longitude
5 = Longitude of fix
6 = E or W of longitude
7 = Speed over ground in knots
8 = Track made good in degrees True
9 = UTC date of fix
10 = Magnetic variation degrees (Easterly var. subtracts from true course)
11 = E or W of magnetic variation
12 = Mode indicator, (A=Autonomous, D=Differential, E=Estimated, N=Data not valid)
13 = Checksum

On récupère les données sous forme de caractère, il faut dans un premier temps découper la trame avec les virgules, ce qui permet de récupérer chacun des champs, il faut ensuite découper chaque champ en fonction des données que l'on veut récupérer, par exemple pour le champ d'heure du fixe, les 2 premiers caractères du champ correspondent à l'heure, les 2 suivant aux minutes, et le reste aux secondes. Il faut ensuite convertir ces tableaux de caractère en nombre pour permettre un envoi de plusieurs données. Et ensuite découper ce nombre sur plusieurs octets pour permettre sont envoi, les coordonnées GPS sont des flottants, codés sur 32 bits, il faut donc 4 octets pour envoyer 1 flottant ce qui fait que 2 données peuvent être envoyé sur une trame comportant des flottants, les entier sont codées sur 16 bits soit 2 octets ce qui permet d'envoyer 4 données entières dans un message.

4.3.2 Le capteur UM6

Le second capteur que nous avons étudié est le capteur UM6, c'est un capteur de position qui combine une un gyroscope, un accéléromètre et un magnétomètre. Ce dernier envoi les donner des capteurs via un bus série.

Les données disponibles sont :

- Angles d'Euler, représente 3 angles qui permettes de décrire l'orientation d'un solide.
- Quaternion, matrice qui permet de représenter un espace 3D.

- Les données brutes du gyroscope, de l’accéléromètre et du magnétomètre.

- L'altitude estimer par covariance.

- Les données modifiées des capteurs, données des capteurs bruts avec un facteur de recalibrage.

Le capteur est reconfigurable via un logiciel qui se connecte via la liaison série du capteur, le logiciel permet de recalibrer les capteurs et de redéfinir la vitesse de transmission ainsi que le mode de transmission des données. Par défaut le capteur communique a 115200 bauds et émet de façon automatique les données sur le bus série. Par défaut seules les données corrigées du gyroscope, de l’accéléromètre, du magnétomètre et des angles estimés sont transmises.

La communication avec le capteur se fait par l'envoi de trame.

Structure d'une trame

's'	'n'	'p'	paquet type (PT)	Adresses	Data Bytes (D0...DN-1)	Checksum 1	Checksum 0

Un nouveau paquet est défini par la séquence de caractère 's', 'n', 'p'. Suivi d'un octet contenant un "paquet type" qui décrit la fonction et la taille de la trame. L'adresse indique le registre concerné ou la commande à effectuer. La séquence d'octet dont la taille a été spécifiée dans PT. Pour finir un checksum de 2 octets pour détecter les erreurs de transmission.

Cette structure est utilisée pour tous les messages reçus et envoyée par capteur UM6.

5 Problème rencontrer

Plusieurs problèmes se sont posé pendant ce projet, le premier était le peux de documentation de départ dont je disposais sur les cartes et de leur fonctionnement. Le fait de ne pas pouvoir reprogrammer les cartes via le port USB qui était pour moi la seule méthode que je connaissais, hors, un bootloader modifier empêchait la programmation de la carte, car elle entrait en conflit avec la puce CAN de la carte.

Un autre problème pour évaluer l’efficacité du protocole SimpleCAN venait du peux de documentation sur le protocole fournit dans le rapport ou le code, se qui obligeait à relire chaque fonction pour voir ce qu'elle faisait.

La configuration de la carte a aussi posé problème, car une documentation claire sur son fonctionnement manquait, en effet seul des documents électroniques, schéma électrique, était disponible pour déterminer son fonctionnement.

Un autre problème venait du fait que tous les documents n'étaient pas a jour sur le dépôt git du projet de la carte, se qui empêchait de compiler le code d'exemple qui devait permettre de reprogrammer une carte par le bus CAN.

La programmation d'une carte par le bus CAN nécessitait de connaître l'identifiant de la carte programmer dans son bootloader, hors ce dernier n'était pas indiqué sur la carte ou dans la documentation.

Un problème matériel est aussi survenu sur une des cartes, la carte d’interface USB ce qui empêchait tout débogage de la programmation, ce problème matériel était dû à un conflit au niveau du SPI de la carte. En effet la puce CAN le MCP2515 était connecter au bus MISO et MOSI que le programmateur utilise pour reprogrammer la carte, hors le MCP2515 envoyai des données en même temps que le programmateur se qui provoquait des erreurs de transmission.

Le plus gros problème a été celui du manque de temps, car le temps de comprendre

comment la carte fonctionnait, de comprendre comment les capteurs fonctionnaient et comment interfacer les différentes cartes le projet était déjà fini, un mois étant un peu court pour pouvoir vraiment implémenter un réseau CAN fonctionnel si l'on ne connaît pas le matériel sur lequel on veut l'implémenter.

6 Conclusion

Ce projet m'a permis d'améliorer mon autonomie et améliorer mes connaissances sur les bus de terrain CAN et des cartes Arduino que je connaissais déjà un peu.

Ce projet était relativement intéressant par rapport a mon projet d'étude, car il concernait le domaine de l'embarqué et correspondait à des contraintes et un objectif concret de mise en œuvre d'un système dans son ensemble.

7 Références

[dépôt github]

https://github.com/surpriserom/Protocole_SimpleCan_Arduino

Le protocole CAN :

- technologuepro.com : Cours systèmes embarqués:Le Bus CAN

http://www.technologuepro.com/cours-systemes-embarques/cours-systemes-embarques-Bus-CAN.htm

- Wikipedia

http://fr.wikipedia.org/wiki/Controller_Area_Network

http://fr.wikipedia.org/wiki/Distance_de_Hamming

http://en.wikipedia.org/wiki/CAN_bus#Remote_frame

-CAN in automation

http://www.can-cia.org/index.php?id=systemdesign-can-physicallayer

http://www.can-cia.org/index.php?id=systemdesign-can-protocol

Les cartes utilisées pour le réseau :

- Arduino Leonardo

http://www.arduino.cc/en/Main/ArduinoBoardLeonardo

- Programmateur via Arduino

http://www.arduino.cc/en/Tutorial/ArduinoISP

- Programmateur USBtinyISP

https://perhof.wordpress.com/tag/usbtinyisp/

- Splashelec projet

http://wiki.splashelec.com/index.php/J/92s

https://github.com/splashelec/splashelec

Programmation et librairie CAN :

Rapport de projet de fin d'études de Kévin Burget

- [Universelle CAN Bibliothek]

http://www.kreatives-chaos.com/artikel/universelle-can-bibliothek

- Universal CAN library for AVRs supporting AT90CAN, MCP2515 and SJA1000

https://github.com/dergraaf/avr-can-lib

- Multiplatform CAN library for Arduino supporting the MCP2515, SAM3X, and K2X controllers

https://github.com/McNeight/CAN-Library

- CAN Bus Shield – MCP2515&MCP2551

https://github.com/Seeed-Studio/CAN_BUS_Shield

http://www.seeedstudio.com/wiki/CAN-BUS_Shield

- Arduino Interface SPI

http://www.arduino.cc/en/Reference/SPI

- Norme NMEA 183

http://www.nmea.org/content/about_the_nmea/about_the_nmea.asp

- Wikipedia

http://en.wikipedia.org/wiki/NMEA_0183

http://en.wikipedia.org/wiki/GLONASS

http://fr.wikipedia.org/wiki/LORAN

http://en.wikipedia.org/wiki/Omega_%28navigation_system%29

- Les trames NMEA

http://www.prism-astro.com/fr/aide/Menu-Telescope/TELESCOPE-GPS/renseignements.htm

- Arduino et GPS

http://blog.iteadstudio.com/play-arduino-with-global-positioning-system-gps/

- utilisation du protocole CAN avec un GPS

http://savvymicrocontrollersolutions.com/arduino.php?article=adafruit-ultimate-gps-shield-seeedstudio-can-bus-shield

- conversion de la longitude et latitude NMEA en degré

http://www.csgnetwork.com/gpscoordconv.html

Capteur UM6

- UM6 Datasheets

p21 ch 10.1.1UART Serial Packet Structure

- Angles d'Euler

http://fr.wikipedia.org/wiki/Angles_d%27Euler

- Quaterion

http://en.wikipedia.org/wiki/Quaternion

8 Annexe

Programmation d'un réseaux CAN avec 3 nœuds composé de la carte interface USB, la carte avec la capteur UM6 et une carte interface NMEA 183. Pour tester ce réseaux, l'interface USB sera relier à un PC et enverra sur la console les messages circulant sur le bus, la carte avec le capteur NMEA enverra des données GPS àà partir d'un trame enregistrer et le capteur UM6 enverra les données en direct pour le gyroscope et les angles Phi, Theta et PSI correspondant aux Roll, Pitch, Yaw.

8.1 Câblage

Le câblage du réseaux peut êtres l'un des suivants :

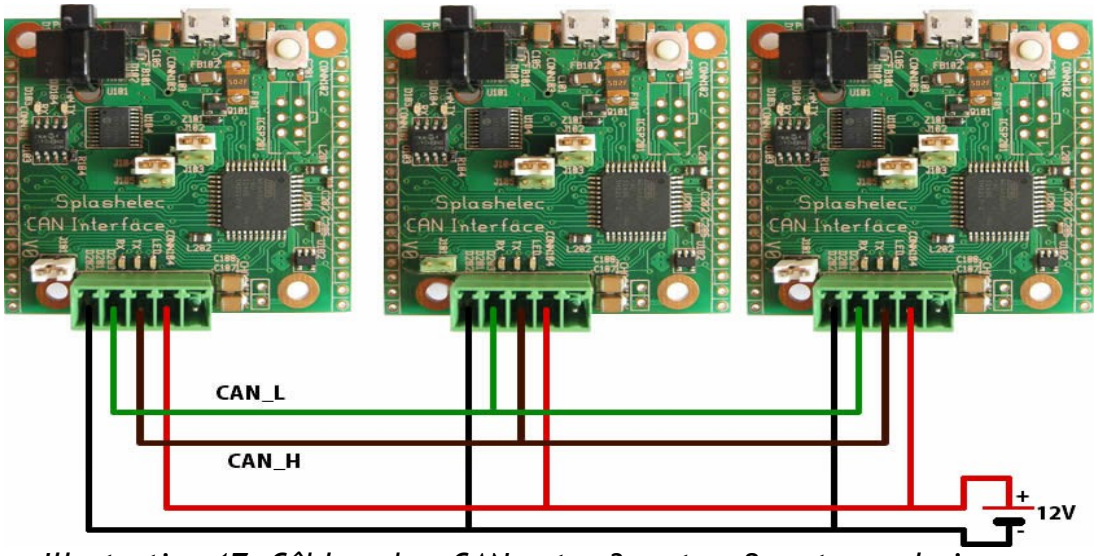

Illustration 17: Câblage bus CAN entre 3 cartes, 2 cartes on le jumper J101 pour la résistance de 120Ω du bus en mode high speed

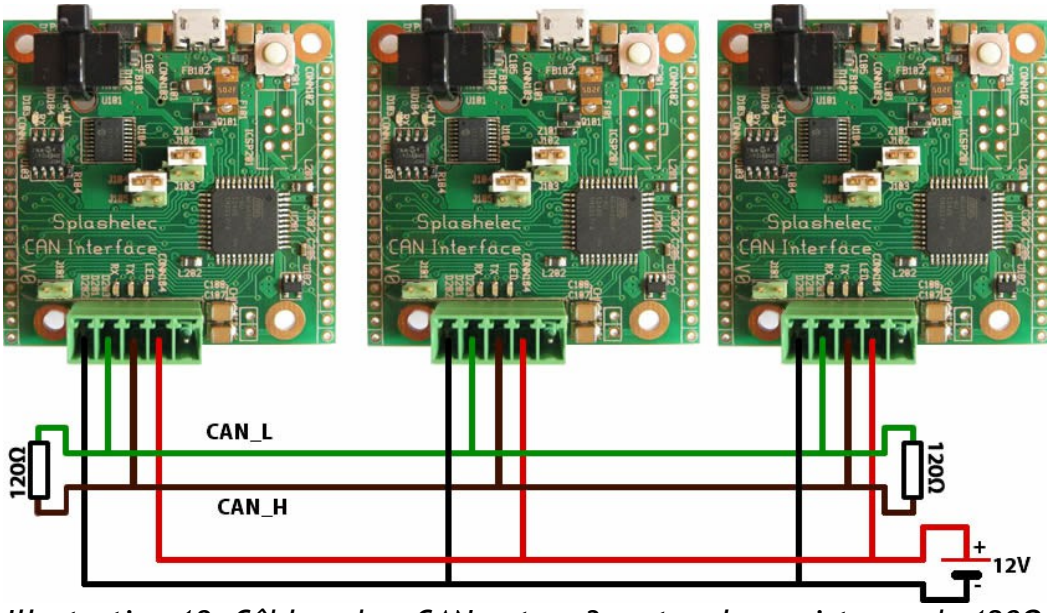

Illustration 18: Câblage bus CAN entres 3 cartes, les resistance de 120Ω sont installer sur le bus et non sur les cartes

Pour que la librairie utilisée lors des tests fonctionne, le jumper J102 doit êtres mis sur toutes les cartes, le 2ème câblage permet de déconnecté une carte du bus sans se soucier de savoir si les 2 résistances de 120Ω son bien installées. L'alimentation des

cartes peut se faire soit en 12V par le bus comme montré sur les schéma soit en alimentant individuellement chaque carte via le port USB, la carte interface USB étant alimenter en 5V. Les cartes Arduino sélectionnes leurs entrées d'alimentation automatiquement en privilégiant l'entrée 5V.

8.2 Programme

Les programmes testés sont disponibles sur le [dépôt github] dans le dossier Programme_Test. Les programme sont : receive_gps_accelero_ex, send_nmea_GPS_ex, UM6_CAN_ex.

8.2.1 Programme receveur

```
#include <SPI.h>
#include "mcp_can.h"
//#include "gps_parser.h" //non utilise dans l'exemple
#include "parseCan.h"


const int SPI_CS_PIN = 9;
const int led = 13;
boolean state = false;

ParseCan parser(true);

MCP_CAN CAN(SPI_CS_PIN);                    // Set CS pin

void setup()
{
   Serial1.begin(115200);

     while (!Serial1) {
       ; // wait for Serial1 port to connect. Needed for Leonardo only
      }

  //set the test signal led
  pinMode(led, OUTPUT);
  digitalWrite(led, LOW);

START_INIT:

   if(CAN_OK == CAN.begin(CAN_500KBPS))              // init can bus : baudrate = 500k
   {
      Serial1.println("CAN BUS Shield init ok!");
   }
   else
   {
      Serial1.println("CAN BUS Shield init fail");
      Serial1.println("Init CAN BUS Shield again");
      delay(100);
      goto START_INIT;
   }
}

void loop()
{
   unsigned char len = 0;
   char buf[80];
   static unsigned long time = 0;
   unsigned long time2 = 0;
```

```
  if(CAN_MSGAVAIL == CAN.checkReceive())            // check if data coming
  {
    CAN.readMsgBuf(&len,(unsigned char *) buf);    // read data,  len: data length, buf: data buf

    unsigned char canId = CAN.getCanId();

    switch(canId)
    {
     case MSG_GPRMC_LAT_LONG :
          Serial1.println("MSG_GPRMC_LAT_LONG");
          Serial1.print("lat:");
          Serial1.print(parser.ucharToFloat((unsigned char *)buf,0));
          Serial1.print(" long:");
          Serial1.println(parser.ucharToFloat((unsigned char *)buf,4));
       break;
     case MSG_GPRMC_VIT_DATE :
          Serial1.println("MSG_GPRMC_VIT_DATE");
          Serial1.print("vitesse:");
          Serial1.print(parser.ucharToFloat((unsigned char *)buf,0));
          Serial1.print("noeud, date:");
          Serial1.print((unsigned char)buf[4],DEC);
          Serial1.print("/");
          Serial1.print((unsigned char)buf[5],DEC);
          Serial1.print("/");
          Serial1.println((unsigned char)buf[6],DEC);
       break;
     case MSG_GYRO_X_Y_Z :
          parser.set_int_GYRO((unsigned char *) buf);
          Serial1.println("MSG_GYRO_X_Y_Z");
          Serial1.print("GYRO x:");
          Serial1.print(parser.get_int_GYRO_X());
          Serial1.print(" y:");
          Serial1.print(parser.get_int_GYRO_Y());
          Serial1.print(" z:");
          Serial1.println(parser.get_int_GYRO_Z());
       break;
     case MSG_IMU_PHI_THETA_PSI :
          Serial1.println("MSG_IMU_PHI_THETA_PSI");
          Serial1.print("IMU phi:");
          Serial1.print(parser.ucharToInt((unsigned char *) buf,0));
          Serial1.print(" theta:");
          Serial1.print(parser.ucharToInt((unsigned char *) buf,2));
          Serial1.print(" psi:");
          Serial1.println(parser.ucharToInt((unsigned char *) buf,4));
       break;
       default: //par defaut on affiche le code hexa que l'on a reçus
         Serial1.print("recus id: ");
         Serial1.println(canId,HEX);
       break;
    }
  }

  time2 = millis();
  if(time2 > time )
  {
   //creer un clignotement asynchrone pour avoir une information visuel de debogage
   state = !state;
   digitalWrite(led, (state ? HIGH : LOW));
   time = time + (state ? 500 : 1000);
  }

}
```

Ce programme attend une série de message dont il connaît l'identifiant sur le bus CAN, dès qu'il reçois un message, il récupère l'identifiant du message et le compare avec un switch au message qu'il connaît. Aucun filtre n'a été initialiser donc il peut recevoir des message qu'il ne connaît pas, pour cela un cas default affiche l'identifiant du message qu'il n'a pas dans sa liste. Ce programme utilise la librairie ParseCan développé pour le projet, elle contient des identifiants pour certain message et des fonctions pour décoder et encoder des "int" et des "float" dans un tableau d'octet.
Voici un exemple de ce qu'affiche la console du logiciel quand les 3 nœud sont connecter :

```
MSG_GPRMC_LAT_LONG
lat:48.40 long:-4.51
MSG_GPRMC_VIT_DATE
vitesse:0.45noeud, date:12/6/15
MSG_IMU_PHI_THETA_PSI
IMU phi:21 theta:10 psi:36
MSG_GYRO_X_Y_Z
GYRO x:146 y:36 z:-80
MSG_IMU_PHI_THETA_PSI
IMU phi:48 theta:4 psi:57
MSG_GYRO_X_Y_Z
GYRO x:-35 y:-29 z:58
MSG_IMU_PHI_THETA_PSI
IMU phi:5 theta:-1 psi:49
MSG_GYRO_X_Y_Z
GYRO x:-78 y:-30 z:-70
MSG_GPRMC_LAT_LONG
lat:48.40 long:-4.51
MSG_GPRMC_VIT_DATE
vitesse:0.77noeud, date:12/6/15
MSG_IMU_PHI_THETA_PSI
IMU phi:5 theta:-12 psi:15
MSG_GYRO_X_Y_Z
GYRO x:92 y:-13 z:-85
```

8.2.2 Programme NMEA

```
#include "mcp_can.h"
#include <SPI.h>
#include "gps_parser.h"
#include "parseCan.h"

// the cs pin of the version after v1.1 is default to D9
// v0.9b and v1.0 is default D10
const int SPI_CS_PIN = 9;
const int led = 13;
boolean state = false;

const int sentenceSize = 80;

unsigned char staticGPRMC[] = "$GPRMC,023934.00,A,4823.97002,N,00430.49225,W,0.636,,120615,,,D*64";

unsigned char sentence[sentenceSize];

GPS_PARSER parserGps(true);
ParseCan parserCan(true);

MCP_CAN CAN(SPI_CS_PIN); // Set CS pin
```

```
void setup()
{
   Serial1.begin(9600);

   while (!Serial1) {
      ; // wait for Serial1 port to connect. Needed for Leonardo only
   }

 //set the test signal led
 pinMode(led, OUTPUT);
 digitalWrite(led, LOW);

START_INIT:

   if(CAN_OK == CAN.begin(CAN_500KBPS))                    // init can bus : baudrate = 500k
   {
      digitalWrite(led, HIGH);
   }
   else
   {
      state = !state;
      digitalWrite(led, (state ? HIGH : LOW));
      delay(150);
      goto START_INIT;
   }
}

void loop()
{
   static unsigned long time1 = 0;
   static unsigned long time2 = 0;
   volatile unsigned long time3 = millis();
   static int i = 0;

   //if (Serial1.available()>0) //commenté car on utilise une trame déjà enregistrer
   if(time3 > time2)
   {
    time2 += 300; //timer ajouter pour ralentir le programme, le programme n'enverra
                  //un message sur le bus que toute les 300ms
    /* //commente pour test
    unsigned char ch = Serial1.read();
    if (ch != '\n' && i < sentenceSize)
    {
     sentence[i] = ch;
     i++;
    }
    else
    {
     sentence[i] = '\0';
     if(parserGps.isGPRMC((char *) sentence))
     {
      */
       GPRMC_frame frame;
       GPRMC_data data;
            unsigned char buff1[8];
            unsigned char buff2[8];

          //  parserGps.parseGPRMC((char *) sentence, &frame); //commenté pour le teste
            parserGps.parseGPRMC((char *) staticGPRMC, &frame);
                  parserGps.convertGprmcFrame(&frame, &data);
       if(frame.valide == 'A')
       {
                  parserCan.floatToUChar(buff1,0,data.latitude);
```

```
                    parserCan.floatToUChar(buff1,4,data.longitude);
                    CAN.sendMsgBuf(MSG_GPRMC_LAT_LONG, 0, 8, buff1);
                    parserCan.floatToUChar(buff2,0,data.speed);
                    buff2[4] = data.day;
                    buff2[5] = data.month;
                    buff2[6] = data.year;
                    CAN.sendMsgBuf(MSG_GPRMC_VIT_DATE, 0, 8, buff2);
        }
        else
        {
        }
     /* } //commenté pour test
      i = 0;
     }*/
   }

   if(time3 > time1 )
   {
     state = !state;
     digitalWrite(led, (state ? HIGH : LOW));
     time1 = time1 + (state ? 500 : 1000);
   }
}
```

Ce programme récupère un trame NMEA, la trame GPRMC et envoi les information si les données sont valide, pour le teste, la réception du signal NMEA qui se fait par le Serial1 à été commenté car on utilise une trame GPRMC enregistrée. Un timer à aussi été ajouté car plusieurs trame NMEA sont transmise normalement en plus de la trame GPRMC, le timer vise à ralentir l'envoi des données sur le bus CAN. Ce programme utilise la librairie GPSParser implémenter pour le projet pour décoder une trame NMEA et plus particulièrement la trame GPRMC et la librairie ParseCan développé pour le projet, elle contient des identifiants pour certain message et des fonctions pour décoder et encoder des "int" et des "float" dans un tableau d'octet.

8.2.3 Programme UM6

```
//modification du code du char a voile pour émettre sur le bus CAN

#define STATE_ZERO 0
#define STATE_S 1
#define STATE_SN 2
#define STATE_SNP 3
#define STATE_PT 4
#define STATE_READ_DATA 5

#define STATE_CHK1 6
#define STATE_CHK0 7

#define PT_HAS_DATA 0b10000000
#define PT_IS_BATCH 0b01000000
#define PT_COMM_FAIL 0b00000001

#define UM6_EULER_PHI_THETA 0x62 // Data register for angles
#define UM6_GYRO_PROC_XY 0x5c // Data register for angular rates
#define GYRO_SCALE_FACTOR 0.0610352 // Scale factor to convert register data to Degrees per Second
#define EULER_SCALE_FACTOR 0.0109863 // Scale factor to convert register data to Degrees

#define DATA_BUFF_LEN 16

#define BAUD 115200
```

```
#include "mcp_can.h"
#include <SPI.h>
#include "parseCan.h"

byte aPacketData[DATA_BUFF_LEN];

byte c = 0;

int nState = 0;
int n = 0;
int nDataByteCount = 0;

struct IMU{
        int in;
}phi, theta, psi; //Roll, pitch and yaw, respectively.

struct GYRO{
 int in;
}girX, girY, girZ; // angular rates relative to the axes X, Y and Z, respectively.

typedef struct {
 boolean HasData;
 boolean IsBatch;
 byte BatchLength;
 boolean CommFail;
 byte Address;
 byte Checksum1;
 byte Checksum0;
 byte DataLength;
} UM6_PacketStruct ;


const int SPI_CS_PIN = 9;
const int led = 13;
boolean state = false;

UM6_PacketStruct UM6_Packet;
MCP_CAN CAN(SPI_CS_PIN);
ParseCan parser(true);

void setup(){
 Serial1.begin(BAUD); // initialize serial port 1

   while (!Serial1) {
     ; // wait for Serial1 port to connect. Needed for Leonardo only
   }
  //set the test signal led
  pinMode(led, OUTPUT);
  digitalWrite(led, LOW);

START_INIT:

   if(CAN_OK == CAN.begin(CAN_500KBPS))    // init can bus : baudrate = 500k
   {
     digitalWrite(led, HIGH);
   }
   else
   {
     state = !state;
     digitalWrite(led, (state ? HIGH : LOW));
     delay(100);
     goto START_INIT;
   }
```

```
}


void loop(){
 unsigned long time2 = 0;
static unsigned long time = 0;
n = Serial1.available();
if (n > 0){
        c = Serial1.read();
        switch(nState){
                case STATE_ZERO : // Begin. Look for 's'.
                        Reset();
                        if (c == 's'){ //0x73 = 's'
                                nState = STATE_S;
                        } else {
                                nState = STATE_ZERO;
                        }
                break;

                case STATE_S : // Have 's'. Look for 'n'.

                        if (c == 'n'){ //0x6E = 'n'
                                nState = STATE_SN;
                        } else {
                                nState = STATE_ZERO;

                        }
                        break;
                 case STATE_SN : // Have 'sn'. Look for 'p'.
                        if (c == 'p'){ //0x70 = 'p'
                                nState = STATE_SNP;
                        } else {
                                nState = STATE_ZERO;
                        }
                break;
                case STATE_SNP : // Have 'snp'. Read PacketType and calculate DataLength.
                        UM6_Packet.HasData = 1 && (c & PT_HAS_DATA);
                        UM6_Packet.IsBatch = 1 && (c & PT_IS_BATCH);
                        UM6_Packet.BatchLength = ((c >> 2) & 0b00001111);
                        UM6_Packet.CommFail = 1 && (c & PT_COMM_FAIL);
                        nState = STATE_PT;

                        if (UM6_Packet.IsBatch){
                                UM6_Packet.DataLength = UM6_Packet.BatchLength * 4;
                        } else {
                                UM6_Packet.DataLength = 4;
                        }
                break;
                case STATE_PT : // Have PacketType. Read Address.
                        UM6_Packet.Address = c;
                        nDataByteCount = 0;
                        nState = STATE_READ_DATA;
                break;
                case STATE_READ_DATA : // Read Data. (UM6_PT.BatchLength * 4) bytes.
                        aPacketData[nDataByteCount] = c;
                        nDataByteCount++;
                        if (nDataByteCount >= UM6_Packet.DataLength){
                                nState = STATE_CHK1;
                        }
                break;
                case STATE_CHK1 : // Read Checksum 1
                        UM6_Packet.Checksum1 = c;
                        nState = STATE_CHK0;
```

```
                    break;
                    case STATE_CHK0 : // Read Checksum 0
                            UM6_Packet.Checksum0 = c;
                            nState = STATE_ZERO;
                            ProcessPacket(); // Entire packet consumed. Process packet
                    break;
                    }
            }

   time2 = millis();
   if(time2 > time )
   {
    state = !state;
    digitalWrite(led, (state ? HIGH : LOW));
    time = time + (state ? 500 : 1000);
   }

}

void ProcessPacket(){
        switch(UM6_Packet.Address){ // Get the angles values (phi, theta and psi)
                    case UM6_EULER_PHI_THETA :
                            phi.in = ((aPacketData[0] << 8) | aPacketData[1]) * EULER_SCALE_FACTOR;
                            theta.in = ((aPacketData[2] << 8) | aPacketData[3]) * EULER_SCALE_FACTOR;
                            psi.in = ((aPacketData[4] << 8) | aPacketData[5]) * EULER_SCALE_FACTOR;
                    break;
                    case UM6_GYRO_PROC_XY : // Get the angular rates
                            if (UM6_Packet.HasData && !UM6_Packet.CommFail){
                                    girX.in = ((aPacketData[0] << 8) | aPacketData[1]) * GYRO_SCALE_FACTOR;
                                    girY.in = ((aPacketData[2] << 8) | aPacketData[3]) * GYRO_SCALE_FACTOR;
                                    if (UM6_Packet.DataLength > 4){
                                            girZ.in = ((aPacketData[4] << 8) | aPacketData[5]) *
GYRO_SCALE_FACTOR;
                                    }
                            }
                    break;
        }
        SendData();
}

void Reset(){
 UM6_Packet.HasData = false;
 UM6_Packet.IsBatch = false;
 UM6_Packet.BatchLength = 0;
 UM6_Packet.CommFail = false;
 UM6_Packet.Address = 0;
 UM6_Packet.Checksum1 = 0;
 UM6_Packet.Checksum0 = 0;
 UM6_Packet.DataLength = 0;
}

void SendData(){
      unsigned long time2 = millis();
      static unsigned long time = 0;
       unsigned char buff[8];
       unsigned char buff2[8];
          if(time  < time2)
      {
        time += 300;
          parser.intToUChar(buff, 0, phi.in);
          parser.intToUChar(buff, 2, theta.in);
          parser.intToUChar(buff, 4, psi.in);
          CAN.sendMsgBuf(MSG_IMU_PHI_THETA_PSI, 0, 8, buff);
```

```
            parser.intToUChar(buff2, 0, girX.in);
            parser.intToUChar(buff2, 2, girY.in);
            parser.intToUChar(buff2, 4, girZ.in);
            CAN.sendMsgBuf(MSG_GYRO_X_Y_Z, 0, 8, buff2);
    }
}
```

Ce programme est une adaptation du code du projet de char a voile de Youcef HADJ KACI pour exploiter le capteur UM6. La partie pour contrôler le servo moteur a été enlever et la partie de débogage a été remplacer par le code pour envoyer les message CAN. Ce programme utilise la librairie ParseCan développé pour le projet, elle contient des identifiants pour certain message et des fonctions pour décoder et encoder des "int" et des "float" dans un tableau d'octet.

Un timer sert à ralentir la vitesse d'envoi des message en mettant 300ms entre chaque envoi de message.

8.2.4 Librairies ParseCan

```
#ifndef _PARSECAN_
#define _PARSECAN_

//liste des identifiant can
//on peut donner des identifiants plus grand pour les tram de données
//ainsi les tram qui envoi des commande seront prioritaire sur le bus
#define MSG_GPRMC_LAT_LONG              0x40 //identifiant pour une tram avec la latitude et la longitude
#define MSG_GPRMC_VIT_DATE              0x41 //identifiant pour une trame avec la vitesse et la date_order
#define MSG_GPGGA_ALT_PREC              0x42 //identifiant pour une trame avec l'altitude et la precision
#define MSG_IMU_PHI_THETA_PSI           0x43 //identifiant pour une trame avec Roll, pitch and yaw
#define MSG_GYRO_X_Y_Z                  0x44 //identifiant avec angular rates relative to the axes X, Y and
Z, respectively.

#define MSG_test            0xFE    // un message de test

class ParseCan
{
   private:
      bool init;
    public:
            ParseCan(bool val);//inutile, sert a compiler
                    //convertie 2 char d'un tableau en int
            int ucharToInt(unsigned char buff[], int offset);
                    //convertie 4 char
            float ucharToFloat(unsigned char buff[], int offset);
                    //buff tableau de char
                    // offset position du msb

                    //convertie un int en 2 char
            void intToUChar(unsigned char buff[], int offset, int val);
                    //convertie un float en 4 char
            void floatToUChar(unsigned char buff[], int offset, float val);
};

#endif
```

Cette librairie va servir à découper les "int" et les "float" dans un tableau de "unsigned char" qui composent les messages CAN et permet de les recoller pour récupérer ces données à la réception. La librairie contient aussi une liste d'identifiant pour les messages CAN, cette liste contient les identifiants utilisés pour les programmes d'exemple.

8.2.5 Librairies GpsParser

```
#ifndef GPSPARSER_h
#define GPSPARSER_h

#include <Arduino.h>

#define NMEALenght 80 //nmea by design is 80 char max per line

/*
* $GPRMC,225446,A,4916.45,N,12311.12,W,000.5,054.7,191194,020.3,E*68
field Name -> 0 -> $GPRMC
field Heure Fixe -> 1 -> 225446 -> 22H 54mn 46s
field gps signal valide -> 2 -> A -> A = ok | V = Warning
field Latitude -> 3 -> 4916.45 -> 49 deg. 16.45 min
field -> 4 -> N -> N = north
field lognitude -> 5 -> 12311.12 -> 123 deg. 11.12 min
field -> 6 -> W -> W = west
field vitesse -> 7 -> 000.5 -> 000.5 = vitesse sol, Knots
field cap -> 8 -> 054.7 -> cap (vrai)
field date fixe -> 9 -> 191194 -> Date du fix 19 Novembre 1994
field -> 10 -> 020.3 -> Déclinaison Magnetique 20.3 deg Est
field cheksum -> 11 -> E*68 -> cheksum 68
*/
typedef struct {
  char valide;
  //latitude data
  char latInd;     //indicateur de latitude N=nord, S=sud
  char latDeg[3] = "00";  //degree from 0 to 90
  char latMn[8] = "0000000";   //minute

  //longitude data
  char longInd;  //indicateur de longitude E=est, W=ouest
  char longDeg[4] = "000";  //degree from 0 to 180
  char longMn[8] = "0000000";   //minute

  //date
  char day[3] = "00";
  char month[3] = "00";
  char year[3] = "00"; //seule les 2 dernier chiffres de l'annee

  //time utc
  char hour[3] = "00";
  char minute[3] = "00";
  char second[3] = "00";

  char speed[6] = "00000"; //vitesse en noeud
  }GPRMC_frame;

  //valeur a 0 par defaut au cas ou la trame est non valide
typedef struct {
  bool valide;
  //latitude data
  float latitude=0.0;

  //longitude data
  float longitude=0.0;

  //date
  unsigned char day = 0;
  unsigned char month = 0;
  unsigned char year = 0;
```

```
  //time utc
  unsigned char hour= 0;
  unsigned char minute = 0;
  unsigned char second= 0;

  float speed = 0.0; //vitesse en noeud
  }GPRMC_data;

/*
*  $GPGGA,123519,4807.038,N,01131.324,E,1,08,0.9,545.4,M,46.9,M, , *42
*
* 01 -> 123519 = Acquisition du FIX à 12:35:19 UTC
* 02 -> 4807.038 = Latitude 48 deg 07.038'
* 03 -> N = north
* 04 -> 01131.324 = Longitude 11 deg 31.324'
* 05 -> E = East
* 06 -> 1 = Fix qualification : (0 = non valide, 1 = Fix GPS, 2 = Fix DGPS)
* 07 -> 08 = Nombre de satellites en pousuite.
* 08 -> 0.9 = Précision horizontale ou DOP (Horizontal dilution of position) Dilution horizontale.
* 09 -> 545.4= Altitude, au dessus du MSL (mean see level) niveau moyen des Océans.
* 10 -> M = en metre
* 11 -> 46.9 = Correction de la hauteur de la géoïde en Metres par raport à l'ellipsoîde WGS84 (MSL).
* 12 -> M = en metre
* 13 -> (Champ vide) = nombre de secondes écoulées depuis la dernière mise à jour DGPS.
* 14 -> (Champ vide) = Identification de la station DGPS.
* 15 -> 42 = Checksum
*  Non représentés CR et LF.
*/
  typedef struct {
  char valide;
  //latitude data
  char latInd;     //indicateur de latitude N=nord, S=sud
  char latDeg[3] = "00";  //degree from 0 to 90
  char latMn[8] = "0000000";   //minute

  //longitude data
  char longInd;  //indicateur de longitude E=est, W=ouest
  char longDeg[4] = "000";  //degree from 0 to 180
  char longMn[8] = "0000000";   //minute

  //time utc
  char hour[3] = "00";
  char minute[3] = "00";
  char second[3] = "00";

  char nbSat[3] = "00"; //Satellites are in view
  char accuracy[4] = "000"; //Relative accuracy of horizontal position

  char altitude[6] = "00000"; // altitude above mean sea level
  char altitudeUnite; // altitude unite mesure (meter)

  }GPGGA_frame;


class GPS_PARSER
{
            public:
      GPS_PARSER(boolean init);
                  void parseGPRMC(const char buffer[], GPRMC_frame *data); //parse a gprmc sentence
                  void parseGPGGA(const char buffer[], GPGGA_frame *data);//parse a gpgga sentence
                  void getNemaField(const char buffIn[], char field[], const unsigned char fieldNb); // get the field
number from NMEA sentence
                  boolean isGPRMC(const char buffer[]);
```

```
        boolean isGPGGA(const char buffer[]);

        void convertGprmcFrame(GPRMC_frame *data, GPRMC_data *val);

    private:
        boolean init;
};

#endif
```

Cette librairie permet de récupérer les données d'une trame NMEA, elle est orienté pour traité les trame GPRMC et GPGGA, la fonction getNemaField permet de traité toute les trame NMEA en se basant sur le format d'une trame NMEA où chaque champ est séparer par une virgule. On peut récupérer les données d'une trame GPRMC ou GPGGA et les récupérer dans les structures correspondante, ce qui permet une exploitation plus simple des données de ces trames. La fonction convertGprmcFrame permet de convertir les données d'une structure GPRMC_frame stockée sous forme de chaîne de caractères en une structure GPRMC_data qui stocke les données sous forme de "int" et de "float".