Rapport de projet
Licence Robotique

Coupe E=M6 2004

Robot Principal

Matthieu Simon ***Iut De Cachan***

Sommaire

- Glossaire **Page 1**
- Résumé **Page 2**
- Abstract **Page 3**
- Remerciements **Page 4**

- Star12 & *µC/OS-II*
1\ Carte CPU Star12 **Page 6**
2\ *µC/OS-II* **Page 8**
2.1\ Organisation de la mémoire Page 8
2.2\ L'OS et la mémoire Page 10
2.3\ Sections Critiques et mémoire paginée Page 10

- Compas
1\ Module boussole *CMPS03* **Page 11**
2\ Bus I2C **Page 12**
3\ Communication avec le compas **Page 13**
4\ « Asservissement de cap » **Page 13**

- Propulsion
1\ Le type char **Page 15**
2\ Description de la plateforme **Page 15**
3\ Carte d'asservissement **Page 16**
3.1\ Le LM629 Page 16
3.2\ Etude électronique Page 17
3.2.1\ Carte hacheurs Page 17
3.2.2\ Commande des LM629 Page 18
3.2.3\ Connecteurs Star12 Page 19
3.2.4\ Réalisation Page 19
3.3\ Librairie d'utilisation des LM629 Page 20
3.3.1\ Fonctions bas niveau Page 20
3.3.1.1\ *LM629_ReadStatus* Page 22
3.3.1.2\ *LM629_ReadDataWord* Page 23
3.3.1.3\ *LM629_WriteCmd* Page 24
3.3.1.4\ *LM629_WriteDataWord* Page 25
3.3.2\ Commandes LM629 Page 26
3.3.2.1\ Liste des fonctions Page 26
3.3.2.2\ Description des fonctions Page 28
4\ Les premiers tours de roues **Page 51**

- Synthèse du projet **Page 53**

- Bibliographie **Page 55**

- Annexes

Glossaire

- ***µC/OS-II :*** Système temps réel multitâches préemptif, développé par J. Labrosse, très réputé dans le domaine de l'embarquée (certifié par la *Federal Aviation Administration*). Il a été porté sur un bon nombre d'architecture depuis la version 2, on peut le trouver sur PIC, HC11-HC12, 80x86, et bien d'autres encore.

- ***OS :*** *Operating System* : Système d'exploitation (e.g. Windows, MacOS, Unix, …)

- ***HC12* :** Microprocesseur 16 bits Motorola : M68HC12, remplaçant du M68HC11.

- ***MC9S12DP256B :*** Microcontrôleur Motorola basé sur le M68HC12.

- ***PIC :*** Microcontrôleur 8 bits de chez *Microchip*.

- ***Flash EEPROM :*** Nouvelle génération de mémoire non volatile programmable et effaçable électriquement. Plus rapide que la précédente.

- ***Timers* :** Modules que l'on trouve dans les microcontrôleurs, ils permettent, par exemple, de générer des signaux TTL où de mesurer des fréquences.

- ***PWM :*** *Pulse Width Modulation* : Modulation de largeur d'impulsions.

- ***ADC** : Analog – Digital Converter* : convertisseur analogique - numérique (CAN)

- ***I2C :*** *IIC* : *Inter-IC Bus* : Bus de transmission série.

- ***CAN*** : *Controller Area Network* : Réseau de terrain à débit moyen (1Mbits/s max) inventé par *Bosch* en 1986.

Résumé

La formation en Licence Professionnelle de Robotique à l'Iut de Cachan impose de réaliser un projet de robotique. Depuis 2003, le projet consiste à réaliser un robot pour participer à la coupe de France de robotique (Coupe E=M6).
Cette année, on joue au *Coconut Rugby*, des petits ballons de rugby sont disposés au sols et sur des cocotiers, le but étant de ramasser les balles et de marquer des essais ou/et des transformations, comme au rugby. Le règlement autorise deux robots par équipe, un robot principal et un robot secondaire, on les distingue surtout par leur hauteur maximum, 40cm pour le premier et 20cm pour le second. Mes activités peuvent se décomposer en :

- Développer la base informatique pour les deux robots.
- Réaliser une partie de l'électronique du robot principal.

Ce document présente une partie du travail effectué. Il reste beaucoup de chose à faire avant de pouvoir participer à la coupe mais nous avons toutes nos chances.

Abstract

In order to be graduate with the *Licence Professionnelle de Robotique* diploma (Bachelor grade), we need to design a robotic project. Since 2003, this project is to make a robot for the French Robotic Cup (Eurobot).
This year we play "*Coconut Rugby*", some little rugby balls are on the table and on coconuts, the robot must catch the balls and make points with. Two robots per team can be designed, a principal robot and a secondary robot, the biggest difference is the height of the robots, 40cm for the first and 20cm for the second. My activities can be decomposed in two parts:

- Develop the base code for the two robots.
- Design some PCBs for the principal robot.

This document presents a part of the things I did. A lot of others need to be done before being able to do the cup but we hope we will.

Remerciements

Je remercie :

- M. Hugues Angelis, professeur à l'Iut de Cachan.
- M. Nicolas Gandolfo, électronique et informatique au CRIC.
- M. Nicolas Gossart, mécanique au CRIC.
- M. Jacques-Olivier Klein, maître de conférence à l'Iut de Cachan et chercheur à l'IEF Orsay.
- M. Thomas Labois, mécanique et électronique au CRIC.
- M. Bertrand Manuel, responsable du CRIIP et professeur à l'Iut de Cachan.
- M. Pascal Martinelli, mon tuteur de projet et professeur à l'Iut de Cachan.
- M. Franck Martinet, mécanique au CRIC.
- M. Christophe Montaron, technicien au CRIIP.
- M. Nicolas Pietu, mécanique au CRIC.
- M. Jean-Luc Ricard, responsable de la section Robotique et professeur à l'Iut de Cachan.
- Les étudiants de la Ménagerie, autre projet de la Licence de Robotique.
- Les étudiants du CRAC, projet E=M6 de GMP2.
- Le département GE1 de l'Iut de Cachan.
- Le département GMP de l'Iut de Cachan.

Star12 & *µC/OS-II*

Pour cette année 2004, nous avons décidé de doter nos deux robots de cartes microcontrôleurs. Les années précédentes, sauf 2000 et 2001, les robots étaient équipés d'une carte PC embarquée. Cette architecture est très différente de celle de cette année, une carte PC nécessite obligatoirement un système d'exploitation, plus de mémoire, …, cependant la puissance de calcul et la possibilité de compiler et debugger sur cible fait la force de ce type de système. Les robots de cette année n'ont pas ce besoin, il est préférable niveau complexité de mise en œuvre et coûts d'utiliser la même plateforme basé sur un microcontrôleur sur les deux robots. Deux possibilités se sont présentées, la première étant de développer une carte comprenant un microcontrôleur PIC et la deuxième d'utiliser la plateforme pédagogique du département GE1 pour tout les TP et projets d'informatique industrielle, la Star12. Nous avons opté pour la seconde, les performances du PIC étant plus faible.

1\ Carte CPU Star12 :

Cette carte est donc la plateforme pédagogique de l'IUT de Cachan développée par J.O. Klein, G. Raynaud et C. André. Le microcontrôleur Motorola MC9S12DP256B (architecture HC12) intègre tous les composants nécessaire pour un fonctionnement basique : timers, sorties PWM, ADC, … .
Le tout est géré par un OS temps réel multitâches préemptif : *µC/OS-II*.

Le MC9S12DP256B est un microcontrôleur 16 bits Motorola basé sur un « Core » M68HC12. Il est spécialement conçu pour les applications d'informatique embarquée et de robotique mobile, il est très utilisé dans les voitures par exemple.
Il comporte, dans les grandes lignes :

- 256k octets de Flash EEPROM.
- 12k octets de RAM.
- 4k octets d'EEPROM.
- 2 interfaces de communications série asynchrones (SCI)
- 8 timers 16bits.
- 2 fois 8 canaux ADC de 10bits.
- 8 modules PWM 8bits.
- 5 contrôleurs CAN.
- 1 contrôleur I2C.
- 3 contrôleurs SPI.
- …
- Et enfin une liaison BDM (*Background Debug Mode*) pour la programmation du circuit et le debug en pas à pas.

Dans notre application, la fréquence de l'oscillateur utilisé est de 50MHz. La tension d'alimentation du circuit est de 5V DC.

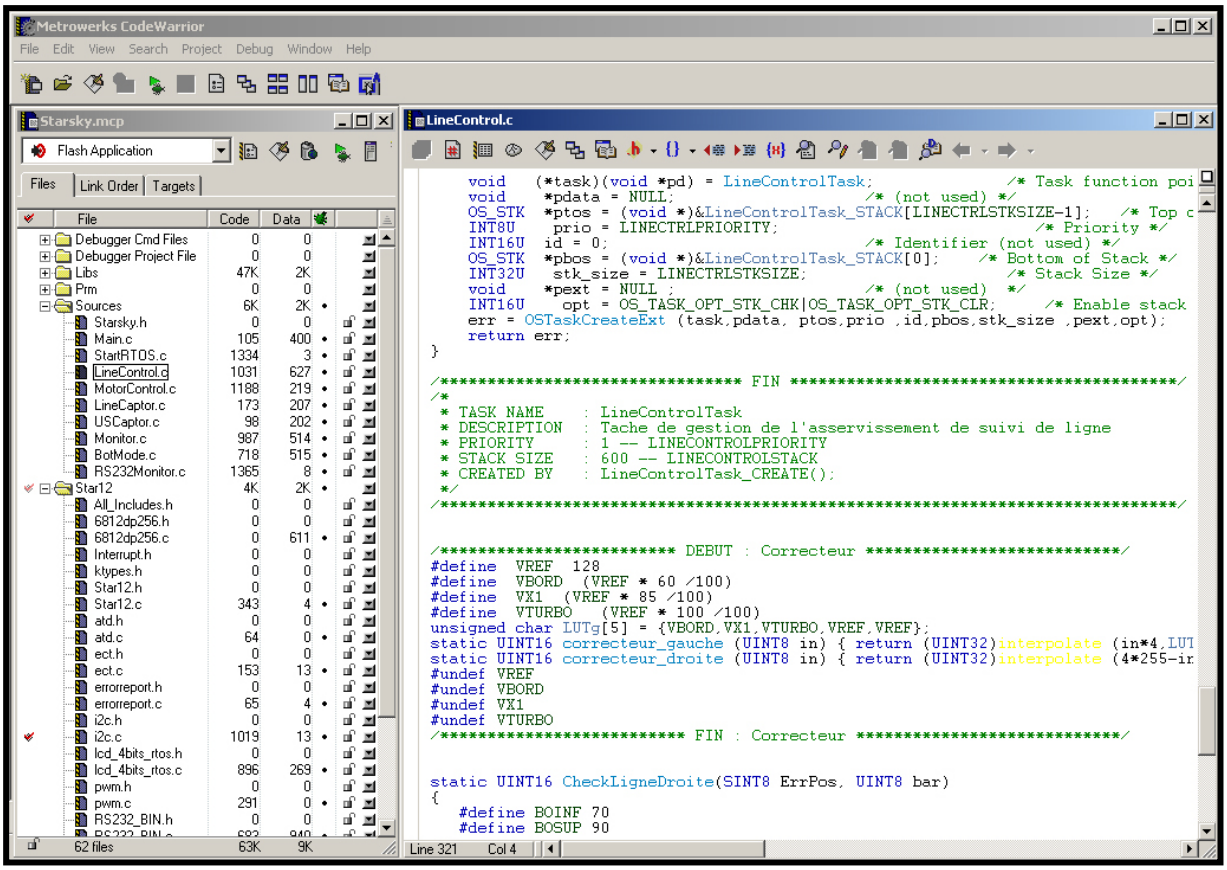
Configuration des connecteurs d'extension : Cf. Documentation Star12

Plateforme de développement : *Metrowerks CodeWarrior HC12*

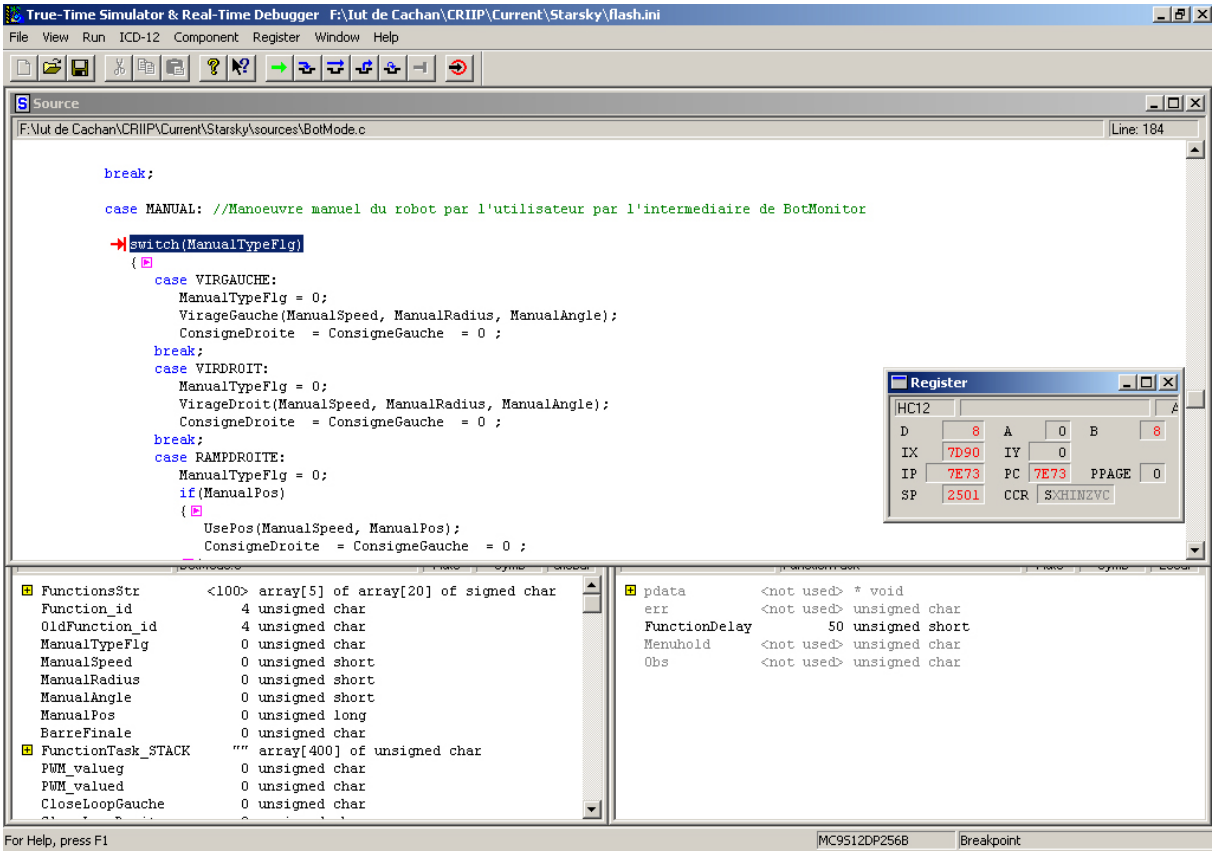
Elle se compose d'un éditeur, d'un compilateur, assembleur, d'un linker et d'un programmateur-debuggeur.

- Editeur : Il offre toutes les capacités d'un bon éditeur de code : une coloration de la syntaxe, des liens vers les définitions des fonctions et des variables, …
- Compilateur : C'est un compilateur évolué ANSI-C. Il offre par défaut une précompilation et compilation ANSI-C ainsi que la librairie ANSI-C (qui a du être modifié par J.O. Klein).
- Assembleur : Permet d'assembler du code assembleur HC12.
- Editeur de liens : Génère le code exécutable sur le MC9S12DP256B.
- Programmateur : Programmation du code en mémoire Flash, EEPROM ou RAM.
- Débuggeur : Debug du C et de l'assembleur en pas à pas, surveillance des registres processeur, …
- Des modules développés à l'Iut : Pour la plupart des modules du microcontrôleur.

Metrowerks CodeWarrior IDE



HI-WAVE



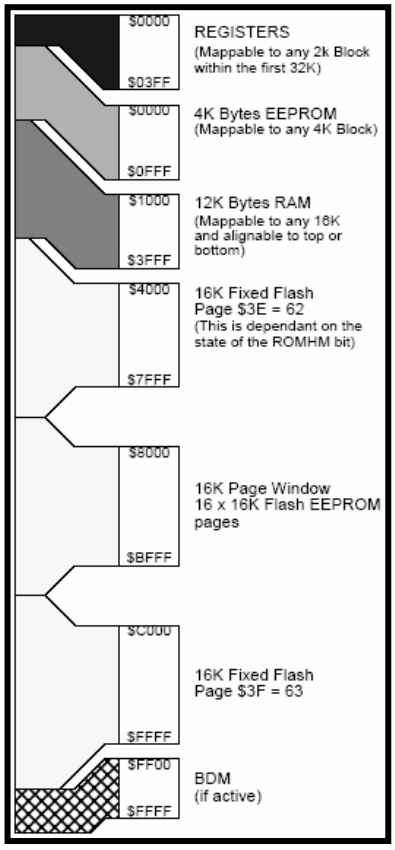
2\ *µC/OS-II* :

La plateforme logicielle intègre un système d'exploitation offrant une multitude de possibilités. Cet « OS » est multitâches, l'illusion de parallélisme permet de décomposer l'application en plusieurs éléments distincts rendant le travail plus simple à réaliser à plusieurs. Le concept de système multitâches temps réel préemptif n'est pas simple et il est difficile de s'en approcher dans ce rapport, je vous invite par contre à consulter le document édité par J.O. Klein *RTOS_UCOSII_S12.pdf*. Un exemple de projet réalisé avec la version 2.0 de *µC/OS-II* s'exécutant en mémoire Flash non paginée est fourni dans l'environnement logiciel de la Star12.

2.1\ Organisation de la mémoire :

L'adressage absolu maximum d'un microprocesseur dépend de la largeur du bus d'adresse, sur le HCS12 il y a 16 bits d'adresse donc 2^16 = 64k octets adressable. Ces 64ko sont répartis en quatre blocs de 16k octets chacun attribués au moment de la programmation du circuit (fichier *xxxx.prm* dans un projet *CodeWarrior*). On trouve d'abord les registres du micro, puis les 4ko d'EEPROM suivi des 12ko de RAM et enfin les trois derniers blocs sont pour la mémoire Flash. (Les derniers 256 octets sont réservés pour le BDM si actif)

A droite ce trouve l'organisation de la mémoire du *9S12DP256* (64ko réellement adressable), rappelons que ce circuit possède 256ko de mémoire Flash or nous avons vu que la capacité d'adressage maximum était de 64ko dont 48 sont réservés pour la Flash. Comment utiliser toute la mémoire disponible ? En réalité, le microcontrôleur propose deux modes de stockage en mémoire Flash, les modes « paginé » et « non paginé ».



Mémoire non paginée : Seules les blocs 1, 2 et 4 sont accessible donc 32k octets de mémoire Flash.

Mémoire paginée : Le microcontrôleur utilise un système pour étendre la capacité d'adressage. Les 256k octets de Flash EEPROM sont en fait divisés en 16 pages de 16k octets chacune sélectionnées à l'aide d'un registre, le registre *PPAGE*. Le troisième bloc est alors actif et une des 14 pages libre (les blocs 2 et 4 pointent toujours sur les deux dernières pages) est alors « mappée » en fonction du registre *PPAGE*. La difficulté de ce mode est qu'il n'utilise pas le même code assembleur que le mode non paginée, les routines d'interruptions doivent aussi être situées dans une zone non paginée, … Cela permet quand même de passer de 32ko à 256ko de mémoire non volatile.

Il a donc fallu trouver un moyen d'utiliser l'OS dans ce mode de fonctionnement.

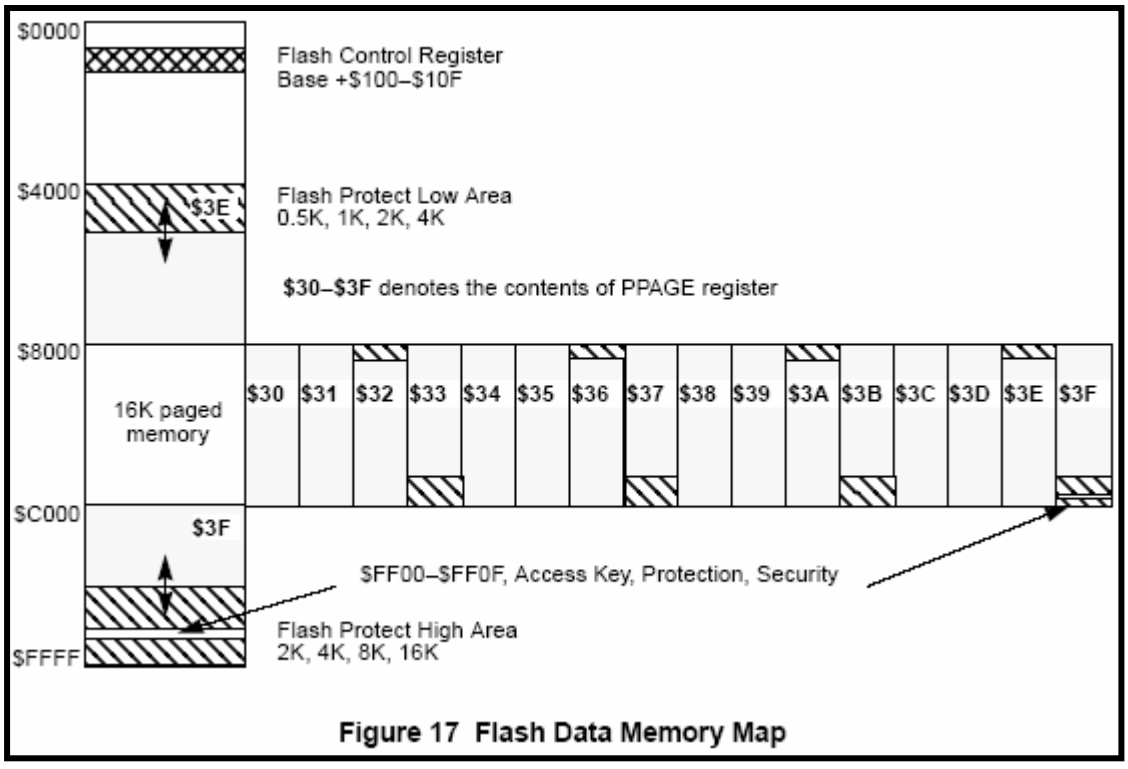


Figure 17 Flash Data Memory Map

2.2\ L'OS et la mémoire :

Le système exécute des changements de contexte afin de « sauter » de tâches en tâches (et autres routines d'interruptions). Ces *Context Switch* sont réalisés par des routines assembleur propre à chaque microprocesseur. Lors de ces procédures, l'OS sauvegarde les registres micro dans la pile de la tâche préemptée et restaure le « contexte » de la tâche de plus haute priorité prête à être exécutée. En mode non paginé, le compteur ordinal (*Program Counter – PC*) est simplement sauvegardé dans la pile, en mode paginée, il faut aussi indiquer la page (*PPAGE Register*) dans laquelle se trouve les instructions valides. Le projet fourni ne fonctionne donc pas en mémoire paginée, après quelques recherches sur Internet, une version 2.7 de l'OS utilisant ce mode a pu nous permettre de l'implémenter sur notre plateforme (Nous nous sommes doté légalement de la version 2.61).

2.3\ Sections Critiques et mémoire paginée :

La version 2.61 de *µC/OS-II* corrige quelques erreurs au niveau de la gestion des sections critiques (*Critical Sections*), il a donc fallu fixer ces bugs dans notre application en suivant les modifications de l'auteur. Lors d'une section critique, le code en train d'être exécuté ne doit pas être préempté afin de garantir l'exclusivité et la validité des ressources partagées. Pour cela il suffit d'indiquer explicitement à l'OS (via les macros *OS_ENTER_CRITICAL()* et *OS_EXIT_CRITICAL()*) de désactiver les interruptions et de les réactiver en fin de section critique. Une des méthodes consiste à sauvegarder l'état du *Code Condition Register*, de désactiver les interruptions et enfin de restaurer ce registre.

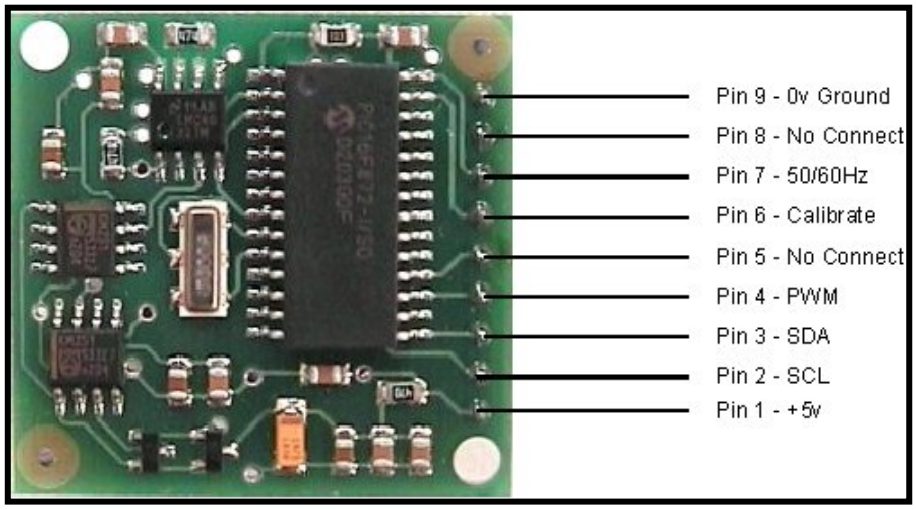
```
#define OS_ENTER_CRITICAL() { // Début de section critique
   OSCPUSaveSR   (&cpu_sr);   // Sauvegarde le CCR
}
#define OS_EXIT_CRITICAL()  { // Fin de section critique
   OSCPURestoreSR( cpu_sr);   // Restaure le CCR
}

void OSCPUSaveSR(OS_CPU_SR *os_cpu_sr)
{
   __asm
   {
      pshd
      tpa          // Copie le CCR dans le registre A.
      ldx   0, sp
      staa  0, x   // Sauvegarde A dans la pile.
      sei          // Désactive les interruptions.
      puld
   }
}

void OSCPURestoreSR(OS_CPU_SR os_cpu_sr)
{
   __asm
   {
      ldaa os_cpu_sr // Copie le paramètre dans le registre A
      tap            // Restaure le CCR avec la valeur continue dans A
   }
}
```

Compas

Une des nombreuses difficultés de la robotique mobile est de se situer dans son environ. Pour la coupe de France de robotique 2004, le robot doit évoluer sur une table sur laquelle se trouve trois autres robots, deux « cocotiers », des balles, des rebords et des zones d'embut. Notre robot n'a pas besoin de connaître à tout moment sa position exacte sur la table, il doit par contre savoir dans quelle direction se trouve les deux zones d'embut et savoir les différencier. Nous avons donc opté pour une solution qui nous fourni à tout moment l'orientation du robot par rapport à une référence connu, l'angle de départ. Les premiers essais ont été réalisés avec un gyroscope *ADXRS300* de chez *Analog Devices*, ils furent peu concluants du fait de la difficulté de mise en œuvre. En effet, ce circuit nécessite une électronique de filtrage et de conversion importante à réaliser, sans cette structure il n'est pas possible de retenir une information fiable. Ce gyroscope délivre une tension analogique variable en fonction de la vitesse angulaire qu'on lui applique, il est donc nécessaire de réaliser une intégration discrète enfin d'obtenir la position angulaire courante. Les bruits électriques et de quantification doivent donc être très faible, ce qui est difficile dans un robot basé sur un petit microcontrôleur.
Nous nous sommes ensuite tourné vers un module compas comprenant une boussole et un microcontrôleur PIC.

1\ Module boussole *CMPS03* :

Ce module bon marché se compose d'un capteur de champs magnétique terrestre Philips KMZ51, d'une électronique d'acquisition et d'un microcontrôleur PIC réalisant un prétraitement de l'information.



Deux modes de communications sont possibles, le premier utilise un signal PWM image de l'orientation du compas et le second est basé sur une communication I2C.

Sortie PWM :

L'information peut être récupéré en mesurant le rapport cyclique du signal avec un timer ou encore en réalisant un filtre passe bas très rapide afin d'extraire la composante continue du signal puis de la convertir en numérique.

Réseau I2C :

Cette méthode est la plus simple à mettre en œuvre sur notre plateforme, la Star12 dispose d'un contrôleur I2C et d'une librairie prête à l'emploi. Le compas fonctionne alors en mode « esclave », le contrôleur maître peut alors envoyer des requêtes et recevoir les valeurs des différents registres du circuit.

Remarque :

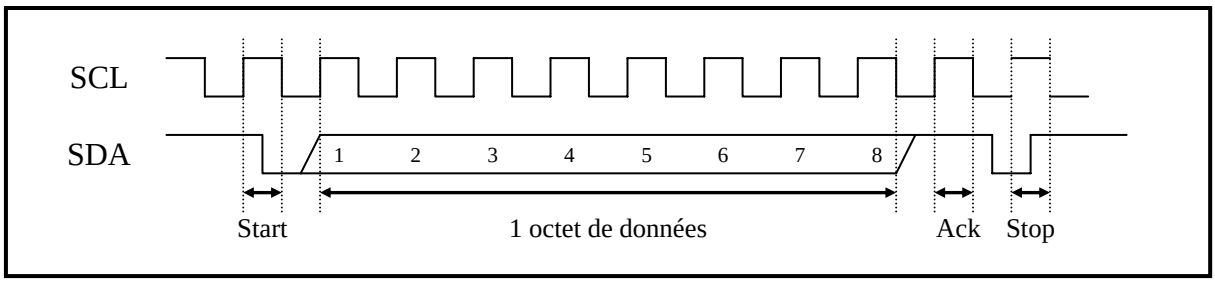
Le module possède aussi une procédure de calibration accessible par l'intermédiaire de la broche 6 (Calibrate) active à l'état bas ou de l'I2C. Pour calibrer le compas, il faut juste pointer les quatre points cardinaux, en fixant la broche 6 au niveau logique bas, ou écrivant 0xFF dans le registre 0x0F en I2C, à chaque fois.

2\ Bus I2C :

Il est né en 1992 quand sa première version fut normalisée par l'IEEE. Ce bus, à l'origine conçu pour l'interfaçage des périphériques d'un calculateur, est une "évolution" du bus parallèle IEEE 488.

Le principe du bus I2C est de transmettre via une liaison trifilaire des informations numériques sur des distances de quelques mètres, il est donc volontairement très limité dans ses capacités de transfert. Ainsi, une trame I2C élémentaire est composée des champs suivants :

- Un délimiteur de début de trame ;
- Un champ de donnée de 8 bits ;
- Un bit de validation ;
- Un délimiteur de fin de trame.



Du niveau de la Star12, trois fonctions ont été définies afin de créer une couche supérieure accessible directement par l'utilisateur :

```
void I2C_IT_Init(void);
void I2C_IT_MasterTx(UINT8 address, UINT8 size,UINT8 * buffer);
void I2C_IT_MasterRx(UINT8 address, UINT8 size,UINT8 * buffer);
```

I2C_IT_Init : Initialise le contrôleur.
I2C_IT_MasterTx : Transmission de données.
I2C_IT_MasterRx : Réception de données.

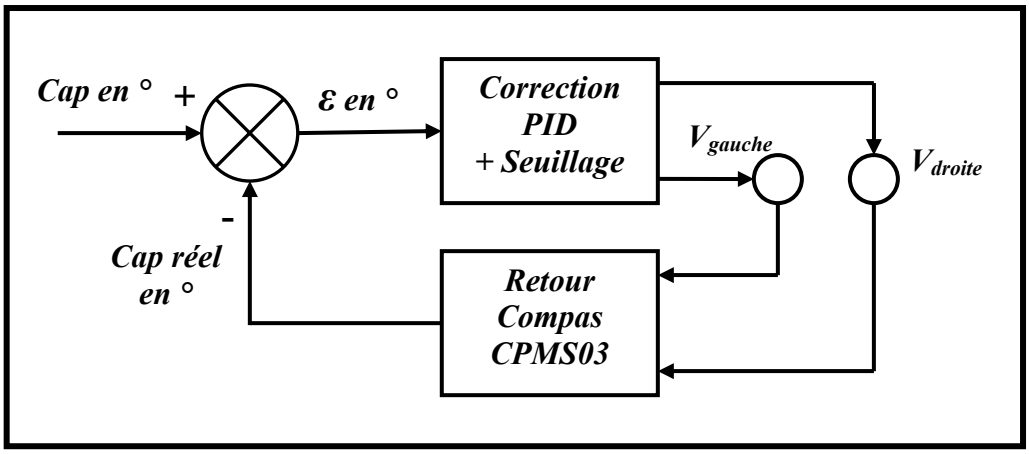
3\ Communication avec le compas :

Le circuit retourne une valeur comprise entre 0 et 3599 représentant l'orientation. Ceci équivaut à un angle de 0° à 359.9°. Pour récupérer cette valeur via le bus I2C, il faut écrire la commande 0x02 (Cf. doc. *CMPS03*) puis lire 2 octets à la suite, un octet de poids fort puis un octet de poids faible. On peut donc définir une fonction permettant de récupérer cette valeur :

```
UINT16 GetAngleFromCompas(void)
{
   UINT16 nValue = 0x0200;                        //Commande 0x02
   I2C_IT_MasterTx(0xC0, 1, (UINT8 *)&nValue); // Ecrit la commande.
   I2C_IT_MasterRx(0xC0, 2, (UINT8 *)&nValue); // Lit 2 trames de 8bits
   return nValue;                                 // Retourne l'angle.
}
```

4\ « Asservissement de cap » :

Le robot doit être capable de suivre un cap qui est susceptible de varier. Cela revient en fait à réaliser un pilote automatique. Le système reçoit un cap à suivre en consigne et agit sur la vitesse des deux moteurs de propulsion afin de garder cette direction.



En implémentant en langage C on obtient :

```
 while(TRUE)
 {
    OSSemPend(CompasSem, 0, NULL);           // Synchronisation
    nRealCap = GetAngleFromCompas();         // Cap en cours via I2C
    nErreur  = (SINT16)(nCap - nRealCap);    // Calcul de l'erreur
    FindAngleModulo(&nErreur);               // Modulo 2PI de l'erreur
    nIntegrale += (SINT32)(nErreur);         // Calcul de l'intégrale
    nDerivee = (SINT32)(nErreur) - nDerivee; // Calcul de la dérivée
...
```

```
...
    nCorr  = nErreur    * COMPAS_KP;  // Correction Proportionnelle
    nCorr += nIntegrale * COMPAS_KI;  // Correction Intégrale
    nCorr += nDerivee   * COMPAS_KD;  // Correction Dérivée
    nCorr >>= 2;                      // Coefficient division par 4

    VitesseLM2 =  nCorr + COMPAS_SPD; // Variation vitesse gauche
    VitesseLM1 = -nCorr + COMPAS_SPD; // Variation vitesse droite
    if(VitesseLM1 > MAX_SPD)          // Seuillage Droit supérieur
       VitesseLM1 = MAX_SPD;          // Si V>Vmax alors V=Vmax
    else if(VitesseLM1 < -MAX_SPD)    // Seuillage Droit inférieur
       VitesseLM1 = -MAX_SPD;         // Si V<-Vmax alors V=-Vmax
    if(VitesseLM2 > MAX_SPD)          // Seuillage Gauche supérieur
       VitesseLM2 = MAX_SPD;          // Si V>Vmax alors V=Vmax
    else if(VitesseLM2 < -MAX_SPD)    // Seuillage Gauche inférieur
       VitesseLM2 = -MAX_SPD;         // Si V<-Vmax alors V=-Vmax
 }
```

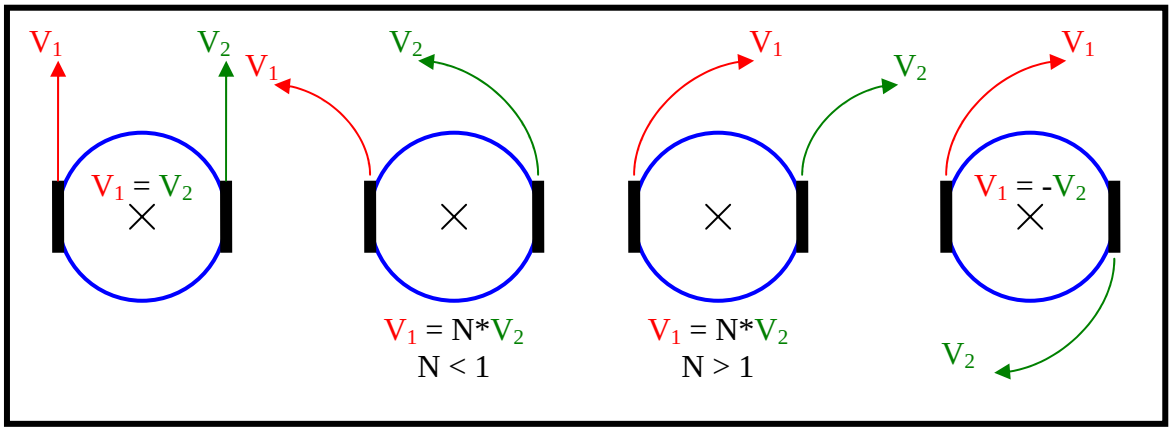
Les essais sont convaincants, le robot suit bien le cap qu'on lui impose. Les premiers réglages sont grossier, les termes Ki et Kd donnent trop d'influence au système qui rentre tout de suite en oscillation. Avec un faible coefficient proportionnel, l'asservissement est souple et d'une précision raisonnable pour notre application, la rapidité n'est pas une chose recherchée. Cependant il faudrait affiner l'influence de la correction qui devient vite trop importante et sature les moteurs.

Propulsion

Le robot doit pouvoir évoluer facilement sur l'aire de jeu, en l'occurrence la table, pour cela nous avons opté pour une propulsion de type char, nous verrons pourquoi ceci est, pour nous, le meilleur compromis entre simplicité de conception et performance. Il doit pouvoir aussi répondre avec une certaine précision aux commandes du « maître », le programme dans notre cas, il a donc fallu étudier, réaliser et implémenter un asservissement.

1\ Le type char :

Ce mode de propulsion se compose de deux roues et d'au moins un patin, ou roue folle, dans le cas d'un char d'assaut il s'agit de deux chenilles uniquement. Les roues sont placées de part et d'autre du véhicule et leur vitesse fixe sa direction, la même vitesse de chaque coté le fera avancer tout droit, les virages sont réalisés en fixant des vitesses différentes sur les deux roues.



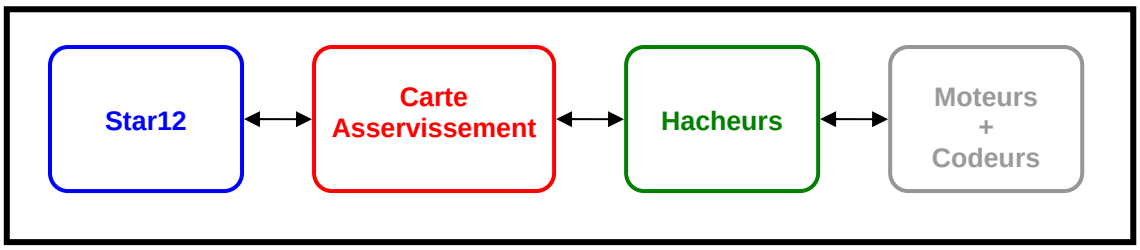
Propulsion de type char. V_1 et V_2 sont des vitesses, N est un facteur.

Cette solution est adéquate dans notre cas car elle nécessite une mécanique simple composée de deux motoreducteurs et de deux roues, et que nous disposons du savoir faire des années précédentes. En effet, les robots de 1998, 1999, 2002 et 2003 fonctionnent avec ce type de propulsion.

2\ Description de la plateforme :

Notre plateforme mobile se compose donc de deux motoreducteurs Maxon et de deux roues usinées dans l'atelier. Pour réaliser l'asservissement que nous verrons par la suite, nous avons ajouté aux moteurs des codeurs optiques Agilent pour le retour d'information. Il faut bien entendu de l'électronique pour piloter les moteurs, celle-ci se constitue d'une partie puissance et d'une partie commande. On distingue, pour la puissance, deux hacheurs quatre cadrans disposés sur une même carte (constructeur *Mesa*), la commande est réalisée par une carte d'asservissement fabriquée à l'Iut et d'une carte microcontrôleur, la Star12 (Cf. doc. Star12).

Chaîne de propulsion :



3\ Carte d'asservissement :

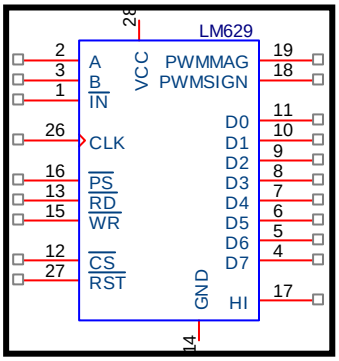
Comme son nom l'indique, son rôle est d'asservir en vitesse et en position les deux moteurs en fonction des paramètres que lui fournis la Star12. Réaliser une électronique d'asservissement performante est long et fastidieux, profitant de l'expérience des robots précédents nous avons choisi d'utiliser la même électronique basée sur des composants bien connus que sont les LM629. Les années passées, les robots utilisant des LM629 avaient tous des cartes PC embarquées possédant un bus PC104 (bus ISA dont le connecteur est différent), l'asservissement était alors réalisé via une carte industrielle placée sur ce bus. L'architecture du microcontrôleur de la Star12 étant différente, il a fallu étudier une nouvelle solution afin de piloter ces composants.

3.1\ Le LM629 :

Ce composant de chez *National Semiconductors* fut longtemps réputé dans le monde de l'électronique, il commence peu à peu à devenir obsolète face à la concurrence tel que la famille des HCTL de chez *Agilent* (ex *Hewlett-Packard*). Cependant cela n'enlève rien de la puissance qu'on lui connaît, ce petit composant (format DIP28) est en réalité un microcontrôleur qui peut gérer l'asservissement en vitesse et en position d'un moteur. Il possède diverses entrées-sorties réparties en deux groupes, d'un coté il récupère les 3 signaux logiques issus des codeurs optiques (1 signal d'index et 2 signaux en quadrature de phase) et commande le hacheur avec une PWM et un signal de direction, et de l'autre une interface de communication avec le contrôleur maître. Ces composants sont destinés à être interfacé avec des systèmes à microprocesseur, ces systèmes sont constitués généralement d'un microprocesseur, de mémoire RAM et ROM et de périphériques. On distingue principalement deux bus dans ces systèmes, le bus de donnée qui permet le transit des données entre les composants, et le bus d'adresse qui permet la sélection des composants qui sont tous reliés à ces deux bus. Typiquement, le microprocesseur sélectionne les composants qu'il veut adresser et ensuite soit il écrit, soit il lit sur le bus de donnée. Le LM629 fonctionne sur ce principe.

Ci-contre la représentation des broches du composant, il en possède 28 qui ne sont pas toutes connectées.

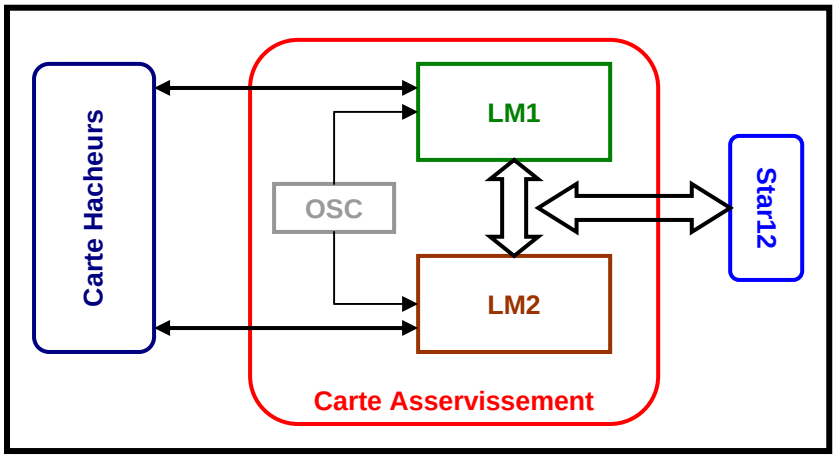
- 14 et 28 sont l'alimentation (5V).
- 1, 2, 3, 18 et 19 sont pour le codeur et le hacheur.
- 4 jusqu'à 11 sont les 8 bits de données.
- 12, 13, 15 et 16 pour la sélection du composant.
- 17 est pour la gestion des interruptions (IRQ).
- 26 est l'horloge du circuit (8MHz).
- 27 est pour le reset.



La carte Star12 utilisée cette année ne permet pas de commander le LM629 de cette façon. En effet, le microcontrôleur de type HC12, ne propose que des ports d'entrées-sorties d'une largeur de 8 bits. Il est donc possible de le piloter mais différemment des années précédentes et cela nécessite l'étude d'une nouvelle carte et d'une librairie la contrôlant.

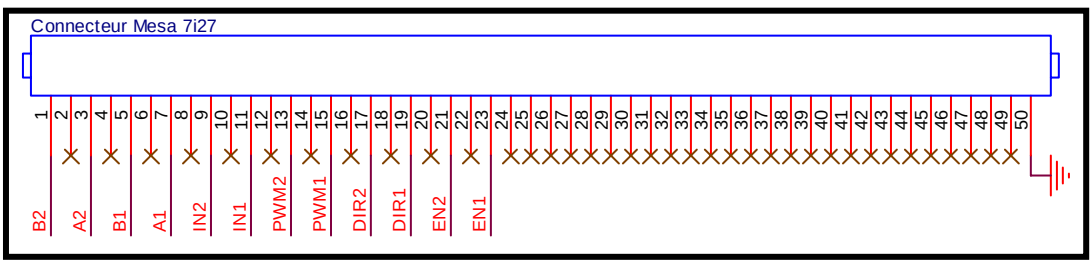
3.2\ Etude électronique :

Comme vu plus haut, le LM629 permet d'asservir un moteur, il faut donc en câbler deux. Pour optimiser le nombre d'entrées-sorties nécessaire il est évident que l'on doit utiliser les propriétés énumérées précédemment, il s'agit donc de connecter les 2 circuits sur les mêmes bus. Nous avons par ailleurs besoin de pouvoir les remettre à zéros et surveiller le niveau d'interruption indépendamment l'un de l'autre. Les 10 signaux pour les 2 hacheurs, quand à eux, doivent se trouver sur un connecteur compatible. Un oscillateur et un connecteur pour la Star12 sont aussi nécessaires.



3.2.1\ Carte hacheurs :

La carte hacheur utilisée est la même que sur le robot 2003, c'est une carte industrielle du constructeur *Mesa* (modèle *7i27*). Elle possède 2 hacheurs 4 cadrans pouvant accepter 10A maximum en régime permanent, et l'électronique nécessaire pour la mise en forme des signaux codeurs. Elle propose aussi une protection en température et une position marche arrêt pour chaque voie.
Le connecteur est un HE10 50 points sur lequel seulement quelques broches sont câblées :



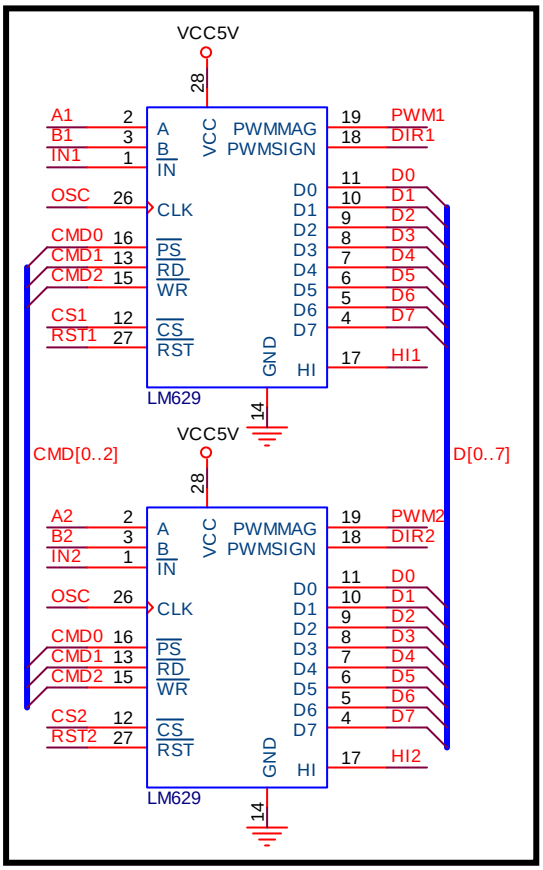
Remarque : Les robots de 1998, 1999 et 2002 ont un petit tic nerveux lorsque qu'on les allume. Cela se produit car au démarrage les LM629 commandent un cours instant les moteurs au maximum de leur vitesse, le hacheur étant en permanence activé, le robot roule un peu. En 2003 et cette année, avec la carte *Mesa*, il y a possibilité de désactiver le hacheur avec EN1 et EN2.

3.2.2\ Commande des LM629 :

Afin de restreindre le nombre d'entrées-sorties nécessaire sur la Star12 il est nécessaire de coupler le plus possible les broches des deux circuits ensemble. Ils peuvent communiquer de quatre façons différentes, ces modes se sélectionnent à l'aide des trois entrées active à l'état bas PS, RD et WR. Ces trois sorties et les 8 bits du bus de données bidirectionnel (D0-D7) ne sont active uniquement lorsque l'entrée CS du circuit est au niveau logique bas. Lorsque cette entrée est au niveau logique haut, les 3 entrées de sélection de mode sont sans effet et les 8 bits de données se placent en sortie haute impédance (HI-Z).

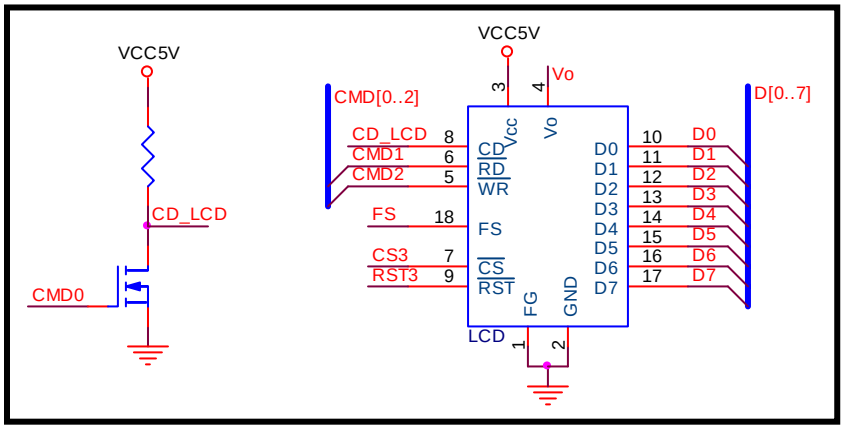
Remarque : Il est théoriquement possible d'écrire la même chose sur les deux circuits en même temps (CS1 = 0 et CS2 = 0), dans ce cas il est judicieux d'agir sur les deux « Chip Select » indépendamment :

CS1	CS2	Chip
0	0	LM1+LM2
0	1	LM1
1	0	LM2
1	1	NONE



Remarque sur le LCD :

L'écran LCD que nous utilisons cette année fonctionne de la même façon, c'est-à-dire qu'il possède un bus de données d'une largeur de 8 bits, 3 entrées de sélection du mode de communication et une entrée de sélection du composant. Le nombre d'entrées-sorties de la Star12 étant restreint il a été nécessaire de placer le LCD sur les mêmes bus. Il possède aussi les quatre mêmes modes, les broches RD et WR sont identiques mais la broche CD est inversée par rapport à la broche PS du LM629, c'est pourquoi il a fallu ajouter un petit circuit inverseur composé d'une résistance et d'un transistor MOS.



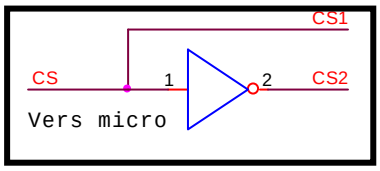
3.2.3\ Connecteurs Star12 :

Il faut maintenant déterminer le nombre de ports nécessaire, leur type et leur utilisation. Le bus de données à besoin d'un port complet (8 bits D0-D7), ce port est amené à fonctionner en entrée et en sortie logique. Le bus de commande (CMD0-CMD2) nécessite trois bits fonctionnant uniquement en sortie logique, tout comme les trois broches « Chip Select » (CS1-CS3), les trois broches de « Reset » (RST1-RST3), les deux broches d'activation des hacheurs (EN1-EN2), et la broche FS (Font Select) du LCD, qui permet de sélectionner la taille de la police sur l'afficheur. Et enfin les deux LM629 ont besoin de deux entrées logique sous interruption (HI1-HI2). Pour des raisons de câblage de la Star12 les ports du microcontrôleur MC9S12DP256B ont été attribué comme suit :

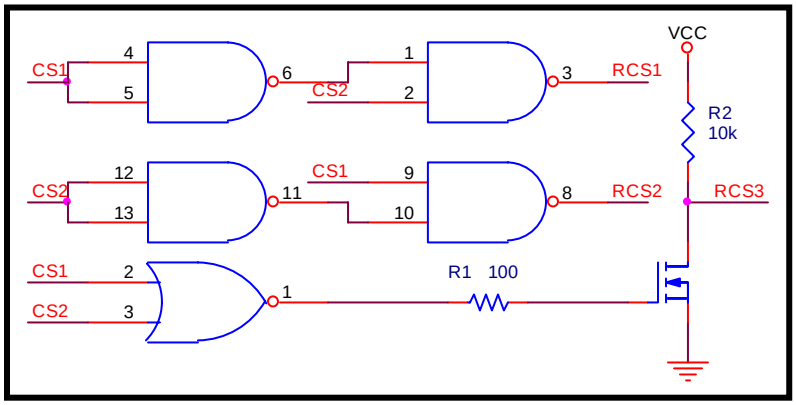
Périphériques	Connecteurs	MC9S12DP256B
D0-D7	HE10 I/O	PORTB : PB0-PB7
HI1-HI2	HE10 I/O	PORTH : PH4-PH5
RST1-RST2	HE10 I/O	PORTH : PH6-PH7
CS1-CS3	DIP20 BarGraph	PORTA : PA0-PA2
CMD0-CMD2	DIP20 BarGraph	PORTA : PA3-PA5
EN1-EN2	DIP20 BarGraph	PORTA : PA6-PA7
FS	DIP20 BarGraph	PORTM : PM6
RST3	DIP20 BarGraph	PORTM : PM7

3.2.4\ Réalisation : (Cf. Schéma structurel et routage CarteAsser en annexe)

Plusieurs versions ont vu le jour, la première réalisation fut simple mais efficace, elle comprenait uniquement les deux LM629 (sans le LCD) dont les broches « Chip Select » de l'un était l'inverse de l'autre. Ceci ne nécessite donc qu'un seul bit de sélection mais permet uniquement de sélectionner soit l'un soit l'autre.
Cette solution a permis d'élaborer la première version de la librairie sur Star12, ensuite il a fallut ajouter le LCD donc réaliser une nouvelle carte. La sélection des composants ce faisait cette fois avec 2 bits, 2^2 = 4 positions :



CS1	CS2	RCS1 (LM1)	RCS2 (LM2)	RCS3 (LCD)
0	0	1	1	0
0	1	0	1	1
1	0	1	0	1
1	1	1	1	1



Au jour où ces lignes sont écrites, en fonction des changements récents de la mécanique (l'électronique les subis forcement), le nombre d'entrées-sorties disponible a augmenté pour donner le tableau du 3.2.3. Ceci permet désormais d'utiliser la fonctionnalité d'écriture simultanée sur les deux LM629. Chaque circuit est désormais sélectionné indépendamment des autres :

CS1	CS2	CS3	Circuit(s)
0	0	0	Ne pas utiliser
0	0	1	LM629-1 & 2
0	1	0	Ne pas utiliser
0	1	1	LM629-1
1	0	0	Ne pas utiliser
1	0	1	LM629-2
1	1	0	LCD
1	1	1	Aucun

Les portes logiques utilisées dans la version précédente augmentaient la constante de temps du système à cause de leur temps de propagation. Un retard était alors visible entre les entrées WR, RD, CD et l'entrée CS de chaque circuit. Désormais ces entrées commutent exactement au même moment et perturbe le fonctionnement global du système.
Il a donc fallu modifier les fonctions bas niveau de l'interface logicielle déjà bien avancée pour donner ce qui suit :

3.3\ Librairie d'utilisation des LM629 : (Cf. Fichiers Sources *LM629.h* et *LM629.c*)

Elle constitue la plus grosse partie du travail, la gestion du LCD sera vue autre part. Il s'agit de commander les deux LM629 avec la Star12. Comme vu plus haut, ce composant communique de quatre façons différentes qui sont :

- Lire l'octet de statut du circuit.
- Lire un mot de 2 octets.
- Ecrire un mot de 2 octets.
- Ecrire une commande.

Nous avons vu qu'il suffit d'agir sur les entrées PS, RS et WR pour choisir un des quatre modes. Ces bits partagent le PORTA avec d'autres entrées (CS1-CS3 et EN1-EN2), pour cela un champs de bits a été défini (Cf. Fichier Source *ExtraIO.h*)

```
#define BUSCTRL     PORTA
#define BUSCTRLTRIS DDRA
static volatile union {
   struct {
      unsigned CS:3; // MSB : CS3\ CS2\ CS1\ : LSB
      unsigned CM:3; // MSB :  WR\  RD\  CD\ : LSB
      unsigned EN:2; // MSB : EN2\ EN1\      : LSB
   }CTRL_BITS;
   UINT8 CTRL_BYTE;
}*CTRLEXT = (UINT8*)&BUSCTRL;
```

3.3.1\ Fonctions bas niveau :

Il y a quatre fonctions bas niveau pour chaque mode de communication avec les circuits. Ces fonctions sont internes aux fonctionnement de la librairie, elles sont définies afin d'améliorer la portabilité sur d'autre plateformes, l'utilisateur n'est pas censé les utiliser. Elles réalisent le lien entre hardware et software, on peut parler dans ce cas de driver bas niveau. Elles utilisent le champ de bits défini ci-dessus, il n'est donc pas nécessaire de toucher au code si l'on change de port ou de bits.

La sélection du composant et du mode se fait en écrivant sur le port de contrôle l' « adresse » du circuit et l'état des entrées de sélection.

```
// "Adresse" des circuits
#define LM1             0x06
#define LM2             0x05
#define LMS             0x04
#define NONE            0x07

// Etats des entrees de selection (Cf. datasheet LM628.pdf)
#define WAIT_CMD        0x06  // WR\=1 RD\=1 CD\=0
#define READSTATUS_CMD  0x04  // WR\=1 RD\=0 CD\=0
#define READDATA_CMD    0x05  // WR\=1 RD\=0 CD\=1
#define WRITECMD_CMD    0x02  // WR\=0 RD\=1 CD\=0
#define WRITEDATA_CMD   0x03  // WR\=0 RD\=1 CD\=1

// Exemple : selection du second LM en mode "Read Status"
CTRLEXT->CTRL_BITS.CS = LM2;
CTRLEXT->CTRL_BITS.CM = READSTATUS_CMD;
```

Les pages suivantes présentent l'implémentation propre aux quatre fonctions bas niveaux.

3.3.1.1\ *LM629_ReadStatus* :

Le composant possède un mode dans lequel il renseigne le maître de son état courant. Cet octet de statut se lit à l'aide de la constante *READSTATUS_CMD*.

Le chronogramme suivant est donné dans la documentation technique de *National* :

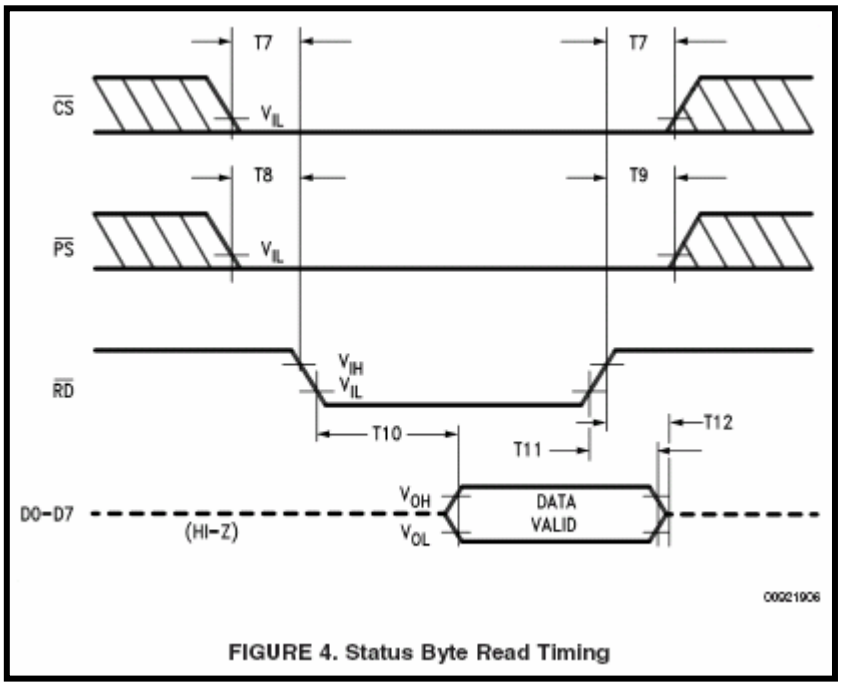


FIGURE 4. Status Byte Read Timing

En traduisant en langage C on obtient :

```
UINT8 LM629_ReadStatus(UINT8 lm)
{
   UINT8 ret;
   OS_CPU_SR cpu_sr;
   OS_ENTER_CRITICAL();                    // section critique
   CTRLEXT->CTRL_BITS.CS = lm;             // selection du LM
   CTRLEXT->CTRL_BITS.CM = READSTATUS_CMD; // selection du mode
   __asm nop;                              // Temps T10
   __asm nop;                              // (Cf. datasheet LM628.pdf)
   BUSDATATRIS = BUSINPT;                  // bus de donnees en entree
   ret = BUSDATA;                          // lecture du bus
   CTRLEXT->CTRL_BITS.CM = WAIT_CMD;       // mode attente (BUS HI-Z)
   CTRLEXT->CTRL_BITS.CS = NONE;           // deselection du LM
   OS_EXIT_CRITICAL();                     // fin de section critique
   return ret;                             // retourne l'octet de statut
}
```

3.3.1.2\ *LM629_ReadDataWord :*

Certaines commandes permettent de récupérer des informations comme la vitesse ou encore la position. Les données transitent uniquement sous forme de mots de 16 bits scindés en un octet de poids fort et un octet de poids faible.
Un mot se lit à l'aide de la constante *READDATA_CMD*.
Le chronogramme suivant est donné dans la documentation technique de *National* :

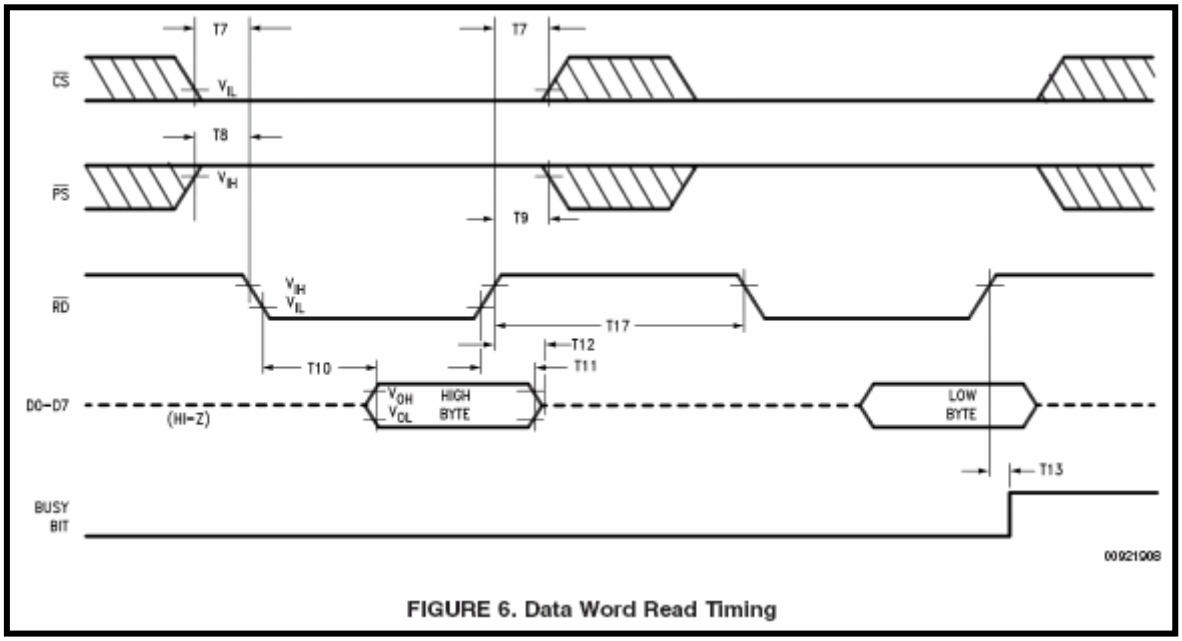


FIGURE 6. Data Word Read Timing

En traduisant en langage C on obtient :

```
UINT16 LM629_ReadDataWord(UINT8 lm)
{
   UINT16 word;
   OS_CPU_SR cpu_sr;
   LM629_Busy(lm);                        // circuit ready ?
   OS_ENTER_CRITICAL();                   // section critique
   BUSDATATRIS = BUSINPT;                 // bus de donnees en entree
   // Octet de poids fort
   CTRLEXT->CTRL_BITS.CS = lm;            // selection du LM
   CTRLEXT->CTRL_BITS.CM = READDATA_CMD; // selection du mode
   __asm nop;                             // Temps T10
   __asm nop;                             // (Cf. datasheet LM628.pdf)
   word = BUSDATA<<8;                     // lecture du MSByte
   CTRLEXT->CTRL_BITS.CM = WAIT_CMD;      // mode attente (BUS HI-Z)
   // Octet de poids faible
   CTRLEXT->CTRL_BITS.CM = READDATA_CMD; // selection du mode
   __asm nop;                             // Temps T10
   __asm nop;                             // (Cf. datasheet LM628.pdf)
   word |= BUSDATA;                       // lecture du LSByte
   CTRLEXT->CTRL_BITS.CM = WAIT_CMD;      // mode attente (BUS HI-Z)
   CTRLEXT->CTRL_BITS.CS = NONE;          // deselection du LM
   OS_EXIT_CRITICAL();                    // fin de section critique
   return word;                           // retourne le mot de 16bits
}
```

3.3.1.3\ *LM629_WriteCmd* :

Le circuit défini un jeu de commande de taille fixe (un octet) que l'on utilise via cette fonction.
Une commande s'écrit à l'aide de la constante *WRITECMD_CMD*.

Le chronogramme suivant est donné dans la documentation technique de *National* :

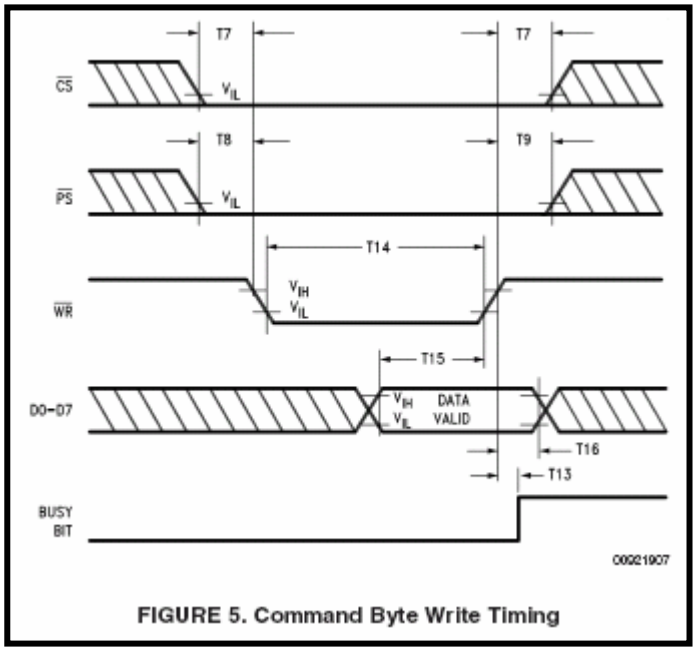


FIGURE 5. Command Byte Write Timing

En traduisant en langage C on obtient :

```
void LM629_WriteCmd(UINT8 lm, UINT8 cmd)
{
   OS_CPU_SR cpu_sr;
   LM629_Busy(lm);                          // circuit ready ?
   OS_ENTER_CRITICAL();                     // section critique
   BUSDATATRIS = BUSOUTP;                   // bus de donnees en sortie
   BUSDATA = cmd;                           // ecriture de la commande
   CTRLEXT->CTRL_BITS.CS = lm;              // selection du LM
   CTRLEXT->CTRL_BITS.CM = WRITECMD_CMD;    // selection du mode
   __asm nop;                               // Temps T14
   __asm nop;                               // (Cf. datasheet LM628.pdf)
   CTRLEXT->CTRL_BITS.CM = WAIT_CMD;        // mode attente (BUS HI-Z)
   CTRLEXT->CTRL_BITS.CS = NONE;            // deselection du LM
   OS_EXIT_CRITICAL();                      // fin de section critique
}
```

Remarque :

Les commandes du LM629 sont définies dans le fichier d'entête *LM629.h*.
Chaque commande est déclarée sous forme d'une « constante manifeste » comprise entre 0 et 255 (8bits). Ces constantes sont destinées à être passé en paramètre de *LM629_WriteCmd*.
(Cf. Documentation technique du LM629 : *LM628.pdf*)

3.3.1.4\ *LM629_WriteDataWord* :

Il est parfois necessaire d'envoyer des données pour configurer les registres du LM629. Ces données transitent uniquement sous forme de mots de 16 bits scindés en un octet de poids fort et un octet de poids faible.
Un mot s'écrit à l'aide de la constante *WRITEDATA_CMD*.
Le chronogramme suivant est donné dans la documentation technique de *National* :

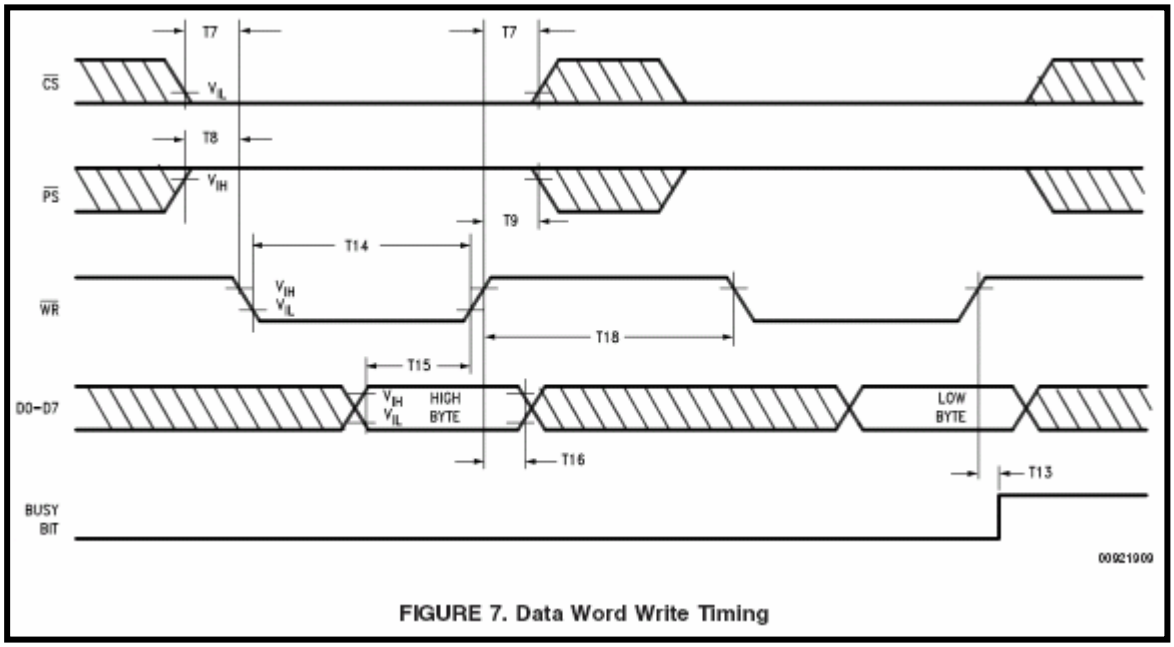


FIGURE 7. Data Word Write Timing

En traduisant en langage C on obtient :

```
void LM629_WriteDataWord(UINT8 lm, UINT16 word)
{
   OS_CPU_SR cpu_sr;
   LM629_Busy(lm);                          // circuit ready ?
   OS_ENTER_CRITICAL();                     // section critique
   BUSDATATRIS = BUSOUTP;                   // bus de donnees en sortie
   // Octet de poids fort
   BUSDATA = (word>>8);                     // ecriture du MSByte
   CTRLEXT->CTRL_BITS.CS = lm;              // selection du LM
   CTRLEXT->CTRL_BITS.CM = WRITEDATA_CMD;   // selection du mode
   __asm nop;                               // Temps T14
   __asm nop;                               // (Cf. datasheet LM628.pdf)
   CTRLEXT->CTRL_BITS.CM = WAIT_CMD;        // mode attente (BUS HI-Z)
   // Octet de poids faible
   BUSDATA = (UINT8)word;                   // ecriture du LSByte
   CTRLEXT->CTRL_BITS.CM = WRITEDATA_CMD;   // selection du mode
   __asm nop;                               // Temps T14
   __asm nop;                               // (Cf. datasheet LM628.pdf)
   CTRLEXT->CTRL_BITS.CM = WAIT_CMD;        // mode attente (BUS HI-Z)
   CTRLEXT->CTRL_BITS.CS = NONE;            // deselection du LM
   OS_EXIT_CRITICAL();                      // fin de section critique
}
```

Ces quatre routines de bas niveau sont maintenant définies, les fonctions de haut niveau les utilisant peuvent maintenant être étudiées. Il s'agit maintenant d'étudier la documentation technique du circuit et d'utiliser les quatre routines ci-dessus pour chaque commande.

3.3.2\ Commandes LM629 :

Le circuit dispose de son propre jeu de commandes régissant son fonctionnement. Ci-dessous la liste exhaustive des commandes (Cf. *LM628.pdf*):

```
#define LM_RESET  0x00 // Reset LM628
#define LM_PORT8  0x05 // Select 8-Bit Output
#define LM_PORT12 0x06 // Select 12-Bit Output
#define LM_DFH    0x02 // Define Home
#define LM_SIP    0x03 // Set Index Position
#define LM_LPEI   0x1B // Interrupt on Error
#define LM_LPES   0x1A // Stop on Error
#define LM_SBPA   0x20 // Set Breakpoint, Absolute
#define LM_SBPR   0x21 // Set Breakpoint, Relative
#define LM_MSKI   0x1C // Mask Interrupts
#define LM_RSTI   0x1D // Reset Interrupts
#define LM_LFIL   0x1E // Load Filter Parameters
#define LM_UDF    0x04 // Update Filter
#define LM_LTRJ   0x1F // Load Trajectory
#define LM_STT    0x01 // Start Motion
#define LM_RDSIGS 0x0C // Read Signals Register
#define LM_RDIP   0x09 // Read Index Position
#define LM_RDDP   0x08 // Read Desired Position
#define LM_RDRP   0x0A // Read Real Position
#define LM_RDDV   0x07 // Read Desired Velocity
#define LM_RDRV   0x0B // Read Real Velocity
#define LM_RDSUM  0x0D // Read Integration Sum
```

Ces commandes permettent de surveiller, écrire et lire (sur) les registres du circuit. Un jeu de fonction est défini pour quasiment toutes les commandes. Il serait trop long et ennuyeux d'expliquer toutes les définitions des fonctions dans ce rapport, son but est plutôt d'expliquer comment utiliser cette librairie. Pour cela chaque fonction est décrite, dans les pages suivantes, paramètre par paramètre, valeur de retour et quelques remarques spécifique à propos de *µC/OS-II* ou de la *Star12*. (Cf. *LM629.h* et *LM629.c*)

3.3.2.1\ Listes des fonctions :

```
/*********************************************************************/
/*                    Prototypes Fonctions HARDWARE                  */
/*********************************************************************/
void LM629_CardReset        ( void     );
void LM629_EnablePower      ( void     );
void LM629_DisablePower     ( void     );
void LM629_InitConfig       ( void     );
void LM629_EnableInterrupts ( UINT8 lm );
void LM629_DisableInterrupts( UINT8 lm );
```

```
/*****************************************************************/
/*                Prototypes Fonctions BAS NIVEAU LM629          */
/*****************************************************************/
void LM629_Busy ( UINT8 lm );
void LM629_Reset( UINT8 lm );

/*****************************************************************/
/*                  Prototypes Fonctions CORRECTEUR              */
/*****************************************************************/
void LM629_SetPID( UINT8  lm,
                   UINT16 Kp, UINT16 Ki, UINT16 Kd, UINT16 LimInt );

/*****************************************************************/
/*                  Prototypes Fonctions TRAJECTOIRE             */
/*****************************************************************/
void LM629_SetAcceleration( UINT8 lm, UINT32 nAcc, BOOL bRelative );
void LM629_SetVelocity    ( UINT8 lm, UINT32 nSpd,
                            BOOL  bRelative, BOOL bOnly, BOOL bSense );
void LM629_SetPosition    ( UINT8 lm, UINT32 nPos, BOOL bRelative );
void LM629_DefineHome     ( UINT8 lm );
void LM629_Start          ( UINT8 lm );
void LM629_Stop           ( UINT8 lm, BOOL bSmooth, BOOL bMotorOff );
void LM629_LoadPositionErrorStop( UINT8 lm, UINT16 nPos );

/*****************************************************************/
/*                Prototypes Fonctions INTERRUPTIONS             */
/*****************************************************************/
void LM629_ResetInterrupts( UINT8 lm, UINT16 nBits );
void LM629_MaskInterrupts ( UINT8 lm, UINT16 nBits );
void LM629_SetBreakPoint  ( UINT8 lm, UINT32 nPos, BOOL bRelative );
void LM629_LoadPositionErrorInt( UINT8 lm, UINT16 nPos );

/*****************************************************************/
/*                  Prototypes Fonctions MONITORING              */
/*****************************************************************/
UINT32 LM629_GetDesiredVelocity( UINT8 lm );
UINT32 LM629_GetRealVelocity   ( UINT8 lm );
UINT32 LM629_GetDesiredPosition( UINT8 lm );
UINT32 LM629_GetRealPosition   ( UINT8 lm );

/*****************************************************************/
/*                   Prototypes Fonctions ODOMETRIE              */
/*****************************************************************/
typedef enum {
   GAUCHE = 1,
   DROITE = 2,
} VIRAGE;
void LM629_Avance     ( UINT32 nAcc, SINT32 nSpeed, SINT32 nPos );
void LM629_Avance_Wait( UINT32 nAcc, SINT32 nSpeed, SINT32 nPos );
void LM629_Virage     ( VIRAGE vVir, UINT32 nAcc,   UINT32 nSpeed,
                                     UINT16 nRayon, SINT16 nAngle );
void LM629_Virage_Wait( VIRAGE vVir, UINT32 nAcc,   UINT32 nSpeed,
                                     UINT16 nRayon, SINT16 nAngle );
```

```
/*****************************************************************/
/*          Prototypes Fonctions ASSERVISSEMENT DE VITESSE       */
/*****************************************************************/
void LM629_SetVelocityOnly( UINT8 lm, SINT16 nSpeed );
```

3.3.2.2\ Description des fonctions :

LM629_CardReset

Réalise un "reset" matériel sur les deux LM629.

```
void LM629_CardReset( void );
```

Paramètres

Aucun.

Valeur de retour

Aucune.

Remarques

- Cette fonction est bloquante pendant 20ms.

LM629_EnablePower

Active la carte hacheur *Mesa 7i27*.

```
void LM629_EnablePower( void );
```

Paramètres

Aucun.

Valeur de retour

Aucune.

Remarques

- Valable uniquement avec la carte *Mesa 7i27*.

LM629_DisablePower

Désactive la carte hacheur *Mesa 7i27*.

```
void LM629_DisablePower( void );
```

Paramètres

Aucun.

Valeur de retour

Aucune.

Remarques

- Valable uniquement avec la carte *Mesa 7i27*.

LM629_InitConfig

Initialisation des LM629 (Reset et Correcteurs) et des constantes mécaniques.

```
void LM629_InitConfig( void );
```

Paramètres

Aucun.

Valeur de retour

Aucune.

Remarques

- A appeler pendant l'initialisation de la carte,avant le demarrage de *µC/OS-II* .

LM629_EnableInterrupts

Active les interruptions venant des LM629.

```
void LM629_EnableInterrupts( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1*, *LM2* ou *LMS*

Valeur de retour

Aucune.

Remarques

- Spécifique à la *Star12* .

LM629_DisableInterrupts

Désactive les interruptions venant des LM629.

```
void LM629_DisableInterrupts( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1*, *LM2* ou *LMS*

Valeur de retour

Aucune.

Remarques

- Spécifique à la *Star12* .

LM629_Busy

Attend que le LM629 soit prêt (bit Busy a l'état bas).

```
void LM629_Busy( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

Aucune.

Remarques

- ATTENTION : Cette fonction est bloquante !

LM629_Reset

Réalise un "reset" logiciel d'un des LM629.

```
void LM629_Reset( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

Aucune.

Remarques

- Cette fonction est bloquante pendant 2ms.

LM629_SetPID

Charge les paramètres du correcteur PID d'asservissement.

```
void LM629_SetPID( UINT8  lm,
                   UINT16 Kp,
                   UINT16 Ki,
                   UINT16 Kd,
                   UINT16 LimInt );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Kp :
Coefficient - Proportionnel (16 bits non signés)

Ki :
Coefficient - Intégrale (16 bits non signés)

Kd :
Coefficient - Dérivée (16 bits non signés)

LimInt :
Limitation de l'Intégrale (16 bits non signés)

Valeur de retour

Aucune.

Remarques

- Cette fonction fixe aussi la fréquence d'échantillonnage de la dérivée (Cf. *LM629.h*).

LM629_SetAcceleration

(Re)Charge l'accélération sur un LM629 (génération de la loi de commande trapézoïdale).

```
void LM629_SetAcceleration( UINT8  lm,
                            UINT32 nAcc,
                            BOOL   bRelative );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nAcc :
La nouvelle accélération en *PasCodeur/s²* sur 32bits non signés.
Pour une accélération en *mm/s²* :
*nAcc = MyAccInMmPerSquaredSecond * AccFactor*

bRelative :
FALSE : Mode Absolu : *NewAcc = nAcc*
TRUE : Mode Relatif : *NewAcc = nAcc + OldAcc*

Valeur de retour

Aucune.

Remarques

- Ne peut être appelée lorsque le LM629 éxécute une trajectoire.

LM629_SetVelocity

(Re)Charge la vitesse sur un LM629 (génération de la loi de commande trapézoïdale ou libre).

```
void LM629_SetVelocity( UINT8  lm,
                        UINT32 nSpd,
                        BOOL   bRelative,
                        BOOL   bOnly,
                        BOOL   bSense );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nSpd :
La nouvelle vitesse en *PasCodeur/s* sur 32bits non signés.
Pour une vitesse en *mm/s* :
*nSpd = MySpdInMmPerSecond * SpeedFactor*

bRelative :
FALSE : Mode Absolu : *NewSpd = nSpd*
TRUE : Mode Relatif : *NewSpd = nSpd + OldSpd*

bOnly :
FALSE : Asservissement de position.
Génération de la loi de commande trapézoïdale.
TRUE : Asservissement de vitesse.
En réalité le LM629 fait galoper la consigne de position.

bSense :
Valide Uniquement si *bOnly == TRUE*.
FALSE : NewSpd = -nSpd
TRUE : NewSpd = nSpd

Valeur de retour

Aucune.

Remarques

- Peut être appelé à la volée (*On The Fly*). Nécessite l'appel à *LM629_Start* pour mettre à jour les registres.

LM629_SetPosition

(Re)Charge la position sur un LM629 (génération de la loi de commande trapézoïdale).

```
void LM629_SetPosition( UINT8  lm,
                        UINT32 nPos,
                        BOOL   bRelative );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nPos :
La nouvelle position en *PasCodeur* sur 32bits non signés.
Pour une position en *mm* :
*nPos = MyPosInMm * MmPerCount*

bRelative :
FALSE : Mode Absolu : *NewPos = nPos*
TRUE : Mode Relatif : *NewPos = nPos + OldPos*

Valeur de retour

Aucune.

Remarques

- Peut être appelé à la volée (*On The Fly*). Nécessite l'appel à *LM629_Start* pour mettre à jour les registres.

LM629_DefineHome

Remise A Zéro de la position, la position actuelle devient la position de référence.

```
void LM629_DefineHome( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

Aucune.

Remarques

Aucune.

LM629_Start

Démarre l'exécution d'un profil généré.

```
void LM629_Start( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

Aucune.

Remarques

- En fonction de l'accélération, de la vitesse et de la position, le LM629 génére un profil trapézoidal, cette fonction indique au LM629 de démarrer la loi de commande.

LM629_Stop

Stop l'exécution du profil généré (et les moteurs).

```
void LM629_Stop( UINT8 lm,
                 BOOL  bSmooth,
                 BOOL  bMotorOff );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

bSmooth :
FALSE : Arrêt brutal. (Arrêt d'urgence)
TRUE : Arrêt Souple : suivant le profil d'acceleration défini.

bMotorOff :
FALSE : Moteur asservi sur la position finale.
TRUE : Moteur éteint. (Roue libre)

Valeur de retour

Aucune.

Remarques

- ATTENTION : Lors des arrets brutaux, le moteur est instantanément asservi sur la position actuelle : cela provoque une accélération qui tend vers l'infinie ce qui n'est pas bon du tout pour la mécanique.
- ATTENTION : Eviter les arrêts en Roue Libre à pleine vitesse, l'inertie du système pouvant être importante il est possible qu'il se deplace encore moteurs éteints.

LM629_LoadPositionErrorStop

Charge un seuil sur l'erreur de position, en cas de dépassement le moteur s'arrête.

```
void LM629_LoadPositionErrorStop( UINT8  lm,
                                  UINT16 nPos );
```

Paramètres

lm :

Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nPos :

La valeur absolue de l'erreur de position maximum en *PasCodeur* sur 16bits non signés.

Pour une position en *mm* :

*nPos = MyPosInMm * MmPerCount*

Valeur de retour

Aucune.

Remarques

Aucune.

LM629_ResetInterrupts

Réinitialise les interruptions LM629.

```
void LM629_ResetInterrupts( UINT8  lm,
                            UINT16 nBits ) ;
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nBits :
Ecrire '0' pour reset l'interruption correspondante. (Cf. *LM628.pdf*)

Valeur de retour

Aucune.

Remarques

- Utiliser les masques d'interruptions déclarés dans le fichier *LM629.h*.
  Ex : *LM629_ResetInterrupts(LM2, ~TRAJ_COMPLETE_INT);*

LM629_MaskInterrupts

Active les interruptions LM629.

```
void LM629_MaskInterrupts( UINT8  lm,
                           UINT16 nBits ) ;
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nBits :
Ecrire '1' pour activer l'interruption correspondante. (Cf. *LM628.pdf*)

Valeur de retour

Aucune.

Remarques

- Utiliser les masques d'interruptions déclarés dans le fichier *LM629.h*.
  Ex : *LM629_MaskInterrupts(*
  LM2, TRAJ_COMPLETE_INT | POSITION_ERROR_INT);

LM629_SetBreakPoint

Charge un point d'arrêt, le LM629 déclenchera une interruption lorsque le point d'arrêt sera atteint.

```
void LM629_SetBreakPoint( UINT8  lm,
                          UINT32 nPos,
                          BOOL   bRelative );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nPos :
La position du point d'arrêt en *PasCodeur* sur 32bits non signés.
Pour une position en *mm* :
*nPos = MyPosInMm * MmPerCount*

bRelative :
FALSE : Mode Absolu : *NewBP = nPos*
TRUE : Mode Relatif : *NewBP = nPos + OldPos*

Valeur de retour

Aucune.

Remarques

Aucune.

LM629_LoadPositionErrorInt

Charge un seuil sur l'erreur de position, en cas de dépassement une interruption se déclenché.

```
void LM629_LoadPositionErrorInt( UINT8  lm,
                                 UINT16 nPos );
```

Paramètres

lm :

Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nPos :

La valeur absolue de l'erreur de position maximum en *PasCodeur* sur 16bits non signés.

Pour une position en *mm* :

*nPos = MyPosInMm * MmPerCount*

Valeur de retour

Aucune.

Remarques

Aucune.

LM629_GetDesiredVelocity

Récupère la vitesse théorique en cours.

```
UINT32 LM629_GetDesiredVelocity( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

La vitesse théorique courante en *PasCodeur/s* sur 32bits non signés.

Remarques

Aucune.

LM629_GetRealVelocity

Récupère la vitesse réelle en cours.

```
UINT32 LM629_GetRealVelocity( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

La vitesse réelle courante en *PasCodeur/s* sur 32bits non signés.

Remarques

- ATTENTION : le LM629 retourne en fait uniquement les 16bits de poids fort d'où une faible precision.

LM629_GetDesiredPosition

Récupère la position théorique en cours.

```
UINT32 LM629_GetDesiredPosition( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

La position théorique courante en *PasCodeur* sur 32bits non signés.

Remarques

Aucune.

LM629_GetRealPosition

Récupère la position réelle en cours.

```
UINT32 LM629_GetRealPosition( UINT8 lm );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

Valeur de retour

La position réelle courante en *PasCodeur* sur 32bits non signés.

Remarques

Aucune.

LM629_Avance

Génération d'un profil trapézoïdal pour chaque moteur.

```
void LM629_Avance( UINT32 nAcc,
                   SINT32 nSpeed,
                   SINT32 nPos );
```

Paramètres

nAcc :
L'accélération en *mm/s²* sur 32bits non signés.
nSpeed :
La vitesse en *mm/s* sur 32bits signés.
nPos :
La position en *mm* sur 32bits signés.
Si '0' alors asservissement de vitesse.

Valeur de retour

Aucune.

Remarques

- Fonction non bloquante.

LM629_Avance_Wait

Génération d'un profil trapézoïdal pour chaque moteur et attend la fin de trajectoire.

```
void LM629_Avance_Wait ( UINT32 nAcc,
                         SINT32 nSpeed,
                         SINT32 nPos );
```

Paramètres

nAcc :
L'accélération en *mm/s²* sur 32bits non signés.
nSpeed :
La vitesse en *mm/s* sur 32bits signés.
nPos :
La position en *mm* sur 32bits signés.
Si '0' alors asservissement de vitesse.

Valeur de retour

Aucune.

Remarques

- Fonction semi bloquante : le retour se fait une fois la trajectoire finie. Fonctionne à l'aide des interruptions et des évenements offert par *µC/OS-II*.

LM629_Virage

Réalise un arc de cercle.

```
void LM629_Virage( VIRAGE vVir,
                   UINT32 nAcc,
                   UINT32 nSpeed,
                   UINT16 nRayon,
                   SINT16 nAngle ) ;
```

Paramètres

vVir :
Coté du virage : *GAUCHE* ou *DROITE*.
nAcc :
L'accélération en *mm/s²* sur 32bits non signés.
nSpeed :
La vitesse en *mm/s* sur 32bits signés.
nRayon :
Le rayon de courbure en *mm* sur 16bits non signés.
nAngle :
L'angle à parcourir en *degrés* sur 16bits signés.

Valeur de retour

Aucune.

Remarques

- Fonction non bloquante.

LM629_Virage_Wait

Réalise un arc de cercle et attend la fin de trajectoire.

```
void LM629_Virage_Wait ( VIRAGE vVir,
                         UINT32 nAcc,
                         UINT32 nSpeed,
                         UINT16 nRayon,
                         SINT16 nAngle ) ;
```

Paramètres

vVir :
Coté du virage : *GAUCHE* ou *DROITE*.
nAcc :
L'accélération en *mm/s²* sur 32bits non signés.
nSpeed :
La vitesse en *mm/s* sur 32bits signés.
nRayon :
Le rayon de courbure en *mm* sur 16bits non signés.
nAngle :
L'angle à parcourir en *degrés* sur 16bits signés.

Valeur de retour

Aucune.

Remarques

- Fonction semi bloquante : le retour se fait une fois la trajectoire finie. Fonctionne à l'aide des interruptions et des évenements offert par *µC/OS-II.*

LM629_SetVelocityOnly

Fixe une vitesse sur l'un des LM629.

```
void LM629_SetVelocityOnly( UINT8  lm,
                            SINT16 nSpeed );
```

Paramètres

lm :
Le LM629 : *LM1* ou *LM2* (Ne pas utiliser *LMS*)

nSpeed :
La vitesse en *mm/s* sur 16bits signés.

Valeur de retour

Aucune.

Remarques

- Fonctionne en asservissement de vitesse.
- La commande *Start* est incluse.

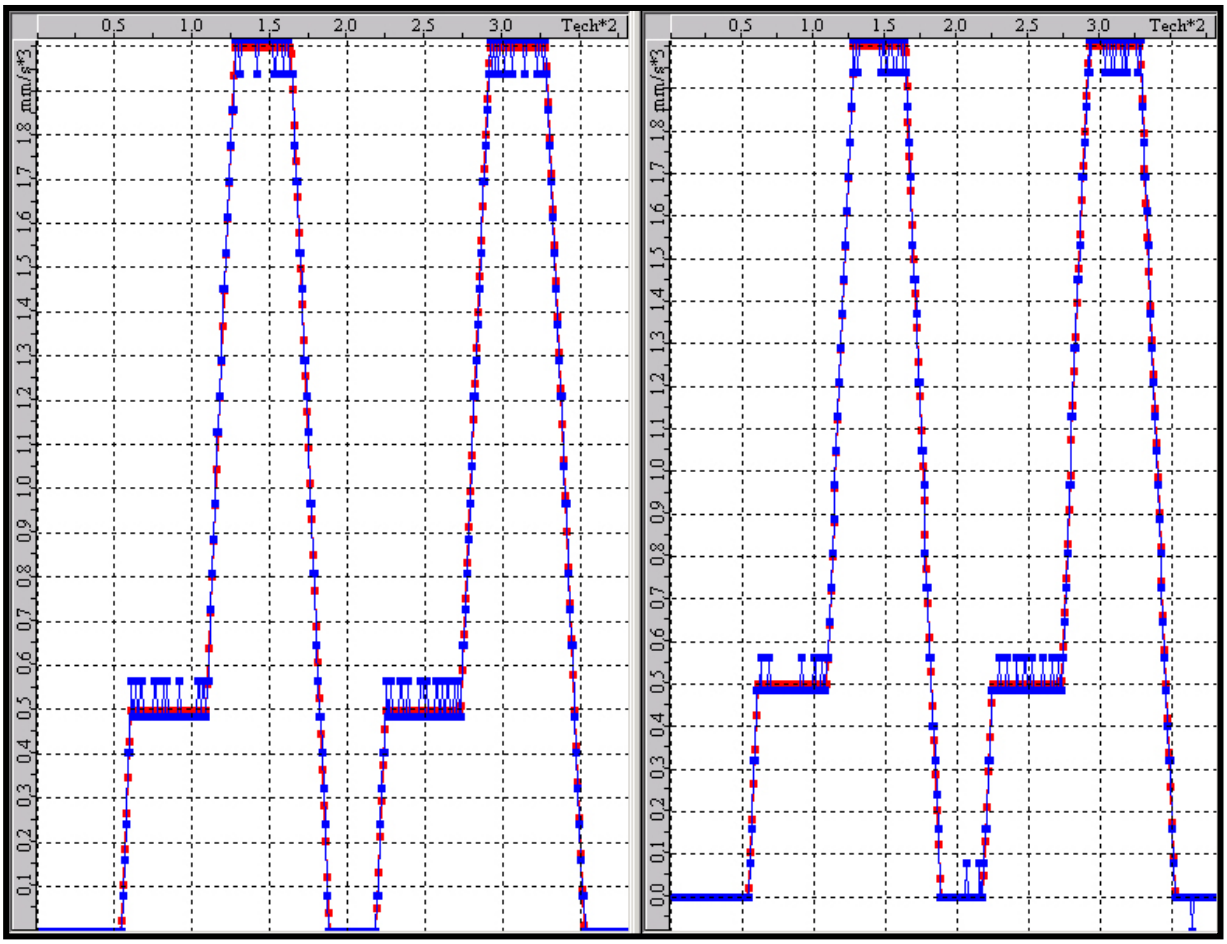
4\ Les premiers tours de roues :

La carte et la librairie sont maintenant opérationnelles, les premiers essais ont était réalisé, bien avant que la mécanique soit fabriquée, avec une petite maquette simple sur laquelle était disposée les deux moteurs, la carte hacheurs, la carte LM629 et la Star12. A partir de la librairie a évoluée rapidement sans encombre, des tests de clothoïdes pré calculées ont même été effectué.
L'intérêt d'un tel circuit, le LM629, est sans aucun doute la performance, réaliser un asservissement d'aussi bonne qualité est difficile de long à mettre en œuvre, ce qui explique peut être le coup du composant, un peu moins de 100€.

Ci-dessous, une des nombreuses courbes de vitesses relevées lors des essais :

Courbe Rouge : Consigne de vitesse en *mm/s*.
Courbe Bleue : Vitesse réelle en *mm/s*.
Base de temps : T/Tech.

Remarque : Ces courbes sont construites à partir des données LM629 récupérées avec les fonctions de « Monitoring », ce qui explique les oscillations lorsque la consigne est stable. (Cf. *GetRealVelocity*) Elles sont dues au quantum énorme provoqué par la faible précision de l'information (Quantum = 2^16 *PasCodeur* car nous avons seulement les 16 bits de poids fort).



Synthèse du projet

Les 7 mois passés sur ce projet passionnant m'ont permis d'améliorer énormément mais compétences techniques en électronique, en informatique industrielle, en mécanique, d'approfondir le langage *VisualC++*, que je pratique comme intéressement personnel, et bien d'autres encore.

Cela m'a aussi permis de me faire une idée de ce qu'est un projet professionnel, des contraintes industrielles de qualité, de temps, de coûts, de rendements, … La nécessité de fournir un travail propre et suivi pour la formation en Licence de Robotique tout comme réaliser un robot performant en vue de coupe E=M6.

Ce dernier mois de Mars fut difficile sur le point du travail à fournir et du stress. Il était prévu de présenter nos deux robots lors des journées portes ouvertes du 27 Mars 2004, ce que nous avons réussi à faire juste dans les temps. Nous avons alors pu voir le robot secondaire tirer ces premières balles, malheureusement le robot principal n'a pas pu faire de démonstration, un problème d'adhérence au niveau des roues motrices l'empêche d'évoluer comme prévu sur la table.

La date butoir de la coupe de France de Robotique est encore loin, il est donc encore temps de trouver le problème et de le contourner. Pour ma part je vais commencer mon stage en laissant plus ou moins la main au suivant tout en gardant un œil sur le robot.

Je vous dis donc à bientôt, le 19 Mai à la Ferté Bernard, pour admirer le travail fini.

Bibliographie

[1] – **Motorola**
MC9S12DP256 Advance Information
2000

[2] – **Jean J. Labrosse**
MicroC/OS-II, The Real Time Kernel, Second Edition
2002

[3] – **Brian W. Kernighan, Denis M. Ritchie**
Le langage C, Norme ANSI, 2e édition
2000

[4] – **National Semiconductor**
LM628/LM629 Precision Motion Controller
2003

[5] – **Toshiba**
Application Notes for the T6963C, LCD Graphics Controller Chip
1995

[6] – **J.O. Klein**
Carte de TP microcontrôleur STAR12, Manuel de référence
2002