Mise en place d'une plateforme d'expérimentation pour systèmes temps réel

Kévin PETIT

14 juin 2005

Table des matières

Chapitre 1

Introduction

1.1 Le stage de 2^{eme} année

La deuxième année de préparation du DUT GEII est cloturée par un stage de dix semaines en entreprise. Ce dernier doit permettre à l'étudiant de mettre en application les compétences techniques et méthodologiques acquises durant l'année. L'immersion en entreprise doit également pousser l'étudiant à travailler en autonomie.

1.2 Présentation de l'entreprise

L'IRCCyN (Institut de Recherche en Communication et Cybernétique de Nantes) est une unité de recherche du CNRS. Les activités menée à l'IRCCyN sont diverses et couvrent les domaines suivants : l'automatique, *les systèmes temps réel*, l'informatique embarquée, le traitment de l'image, et bien d'autres .... Durant mon stage, je travaillais pour l'équipe système temps-réel. Cette équipe travaille actuellement sur la mise en oeuvre de moyens permettant une simulation fine (bonne précision temporelle) des systèmes distribués (plusieurs processeurs) et des problèmes d'ordonnancement.

Vous pouvez voir le batiment principal de l'IRCCyN en figure 1.1.

Fig. 1.1 – Batiment Principal de l'IRCCyN

1.3 Présentation du sujet de stage

Le travail qui m'a été attribué se divisait en deux parties : la mise en place d'une plateforme d'expérimentations à base de cartes à C167 et de cartes d'E/S et le developpement d'une application permettant le test des cartes et de l'exécutif temps réel developpé dans l'équipe : Trampoline.

1.3.1 Mise en place des cartes

Le mise en place des cartes se décompose en plusieurs étapes. Tout d'abord, une lecture des documentations et une prise en main des outils de développement s'impose. Ensuite, les carte d'E/S devront être interfacées avec les cartes à C167. Et enfin, la communication CAN entre les différentes cartes devra être rendue possible (réalisation de câbles et configuration des cartes).

1.3.2 Applications de test

Deux applications de test seront ensuite developpées : une première application simple permettant le test des cartes à C167 et des cartes d'E/S puis une deuxième application plus complète mettant en oeuvre Trampoline et la communication CAN.

1.3.3 Planning

Je travaillais du lundi au vendredi (8h30 -> 12h puis 13h30 -> 18h) soit un volume horraire de 40 heures par semaine. Ces horraires sont semblables à ceux observés dans le milieu professionnel et m'on donc permis une mise en condition réaliste.

Chapitre 2

Présentation du matériel

2.1 Introduction

Nous allons ici présenter le matériel qui va être utilisé. La plateforme d'expérimentation comportera à terme six cartes à microcontrôleur C167 reliées par un réseau CAN.

2.2 Présentation du C167CS

Le C167CS fabriqué par Infineon est un microcontrôleur 16 bits doté d'une grande varité de périphériques. La diversité des périphériques qu'il propose ainsi que les deux contrôleurs CAN qu'il embarque en font un microcontôleur idéal pour les systèmes temps réel embarqués.

Voici quelques uns de ses périphériques :

- 32 unitées de Capture/Comparaison
- Un convertisseur A/N 10bits avec 16 voies d'entrée
- Quatre modules de PWM
- Deux contrôleurs CAN
- Un bus externe
- 9 timers 16 bits
- et d'autres ...

2.3 Le phyCORE 167 et la carte phyCORE HD200

Ces cartes sont destinées à simplifier le developpement d'un système à base de C167. En effet, les boitiers du C167 et des mémoires associées (RAM + Flash) comportent beaucoup de broches et il serait difficile voire impossible de les souder « à la main », ce qui rendrait difficile la réalisation d'un prototype.

2.3.1 Le phyCORE 167

Le phyCORE 167 est un module configurable supportant un microcontrôleur C167CS, 256Ko de RAM, 256Ko de Flash, un oscillateur 20MHz, une horloge temps réel, deux tran-

ceivers CAN et un circuit permettant d'adapter les tensions de la liaison série à la norme RS-232.

Il est utilisable directement sur des cartes *utilisateur*, cependant, par souci de gain de temps, nous utiliserons la *developpement board phyCORE HD200*.

Une photo du phyCORE est disponible en figure 2.1.

FIG. 2.1 – Le phyCORE.

2.3.2 La carte phyCORE HD200

Cette carte permet d'accueillir un module phyCORE. Elle offre deux connecteurs DB9 femelles pour les liaisons séries et deux connecteur DB9 mâles pour les bus CAN. Un connecteur d'extension est également disponible (pour la carte GPIO) ainsi qu'un ensemble de cavaliers permettant de configurer la carte. On retrouve également un connecteur d'alimentation et deux boutons poussoirs (Reset et Boot).

Une photographie de cette carte est visible en figure 2.2.

2.4 La carte d'extension GPIO

La carte GPIO rassemble sur une carte au format Euro[1] divers composants. Elle comporte deux connecteurs permettant de l'interfacer à la *developpement board phyCORE HD200*. Une carte de ce genre est d'une grande utilité dans la mise au point de systèmes à base de microcontrôleurs. En effet, les composants présents sur la carte sont « précablés », et l'utilisateur de la carte n'a qu'à relier les composants de la carte avec les ports du microcontrôleur utilisés par *wrapping*. La carte GPIO supporte :

– 3 potentiomètres analogiques

[1] 160 x 100mm

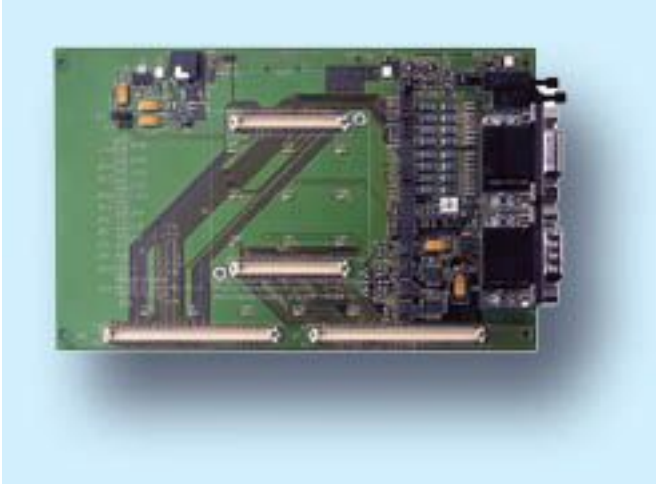

FIG. 2.2 – La carte phyCORE HD200

– 6 boutons poussoir avec anti-rebonds intégrés (monostables)
– 8 DIP switchs
– 8 LEDs
– un connecteur permettant d'acueillir un afficheur LCD
– un moteur doté d'un capteur optique pour la mesure de la vitesse
– deux connecteurs d'E/S disponibles pour l'utilisateur

Une photo de la carte GPIO est disponible en figure 2.3.

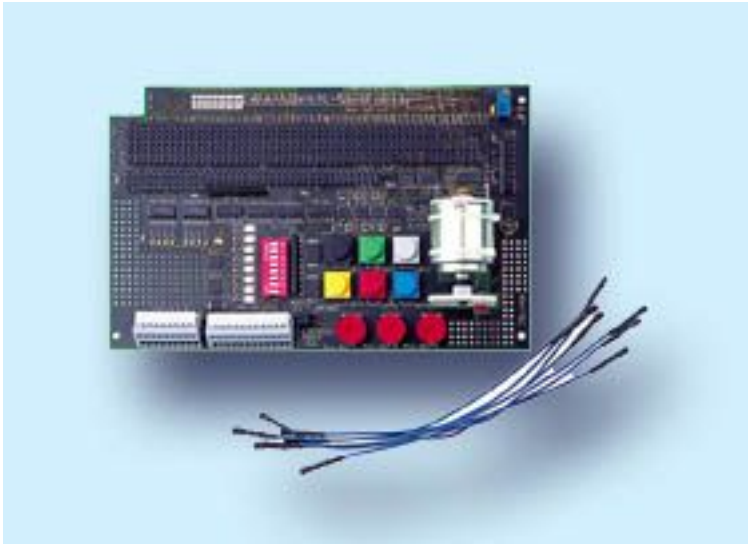

FIG. 2.3 – La Carte GPIO.

2.5 Outils pour le bus CAN

Je disposais de deux outils afin d'analyser le traffic sur le bus CAN : un module CANUSB de chez Systec et un CANscope de la société $\overrightarrow{Vector}$.

2.5.1 Le module CANUSB

Comme son nom l'indique le CANUSB est un module USB autorisant un PC sous Windows à se comporter comme un noeud CAN. Etant donné que ce module se comporte comme un noeud à part entière, son comportement sur le bus est intrusif (En effet, le module acquite les trames reçues...). Le contrôleur qu'il intègre supporte les messages standards et étendus (CAN v2.0B).

Lorque l'on démarre le logiciel PcanView (Logiciel d'exploitation du module), celui-ci demande la configuration de la vitesse des modules détectés.

Le logiciel demande ensuite à l'utilisateur de choisir le module et le masque de réception qu'il veut utliser (copie d'écran en figure 2.4).

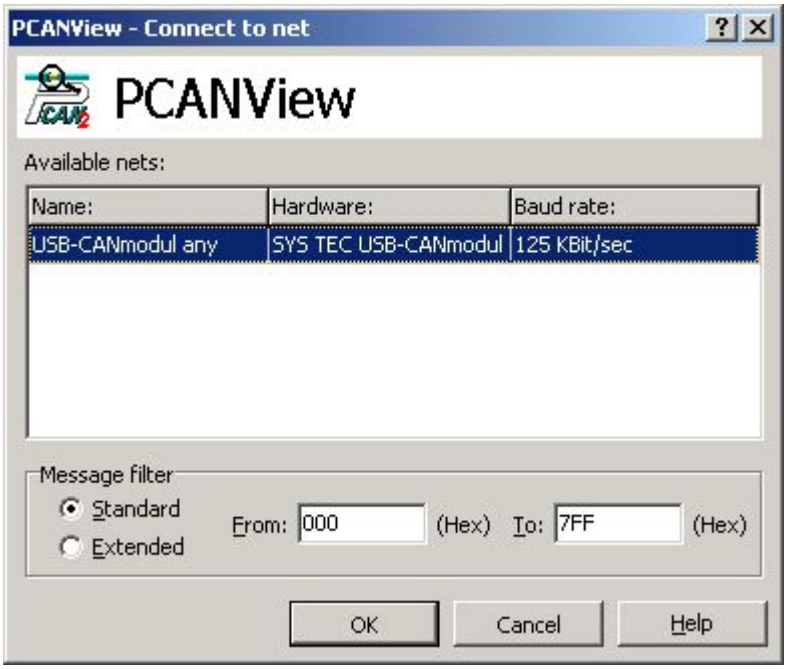

Fig. 2.4 – Choix du module et du filtre de réception

Le logiciel est maintenant démarré et l'on peut commencer à recevoir ou envoyer des messages CAN. La partie du haut permet de visualiser les messages reçus. La partie du bas, quant à elle, nous informe sur les messages envoyés ou en attente d'envoi. En plus d'afficher les ID des messages et les données qu'elles renferment, PcanView nous informe sur la périodicité des messages et le nombre de messages envoyés ou reçus. PcanView dispose d'autres fonctionnalités comme l'envoi de trames de données suite à la recption de la trame de requête correspondante. L'utilisation de ces fonctionnalités est très intuitive. Voici une copie d'écran de l'interface du logiciel (figure 2.5).

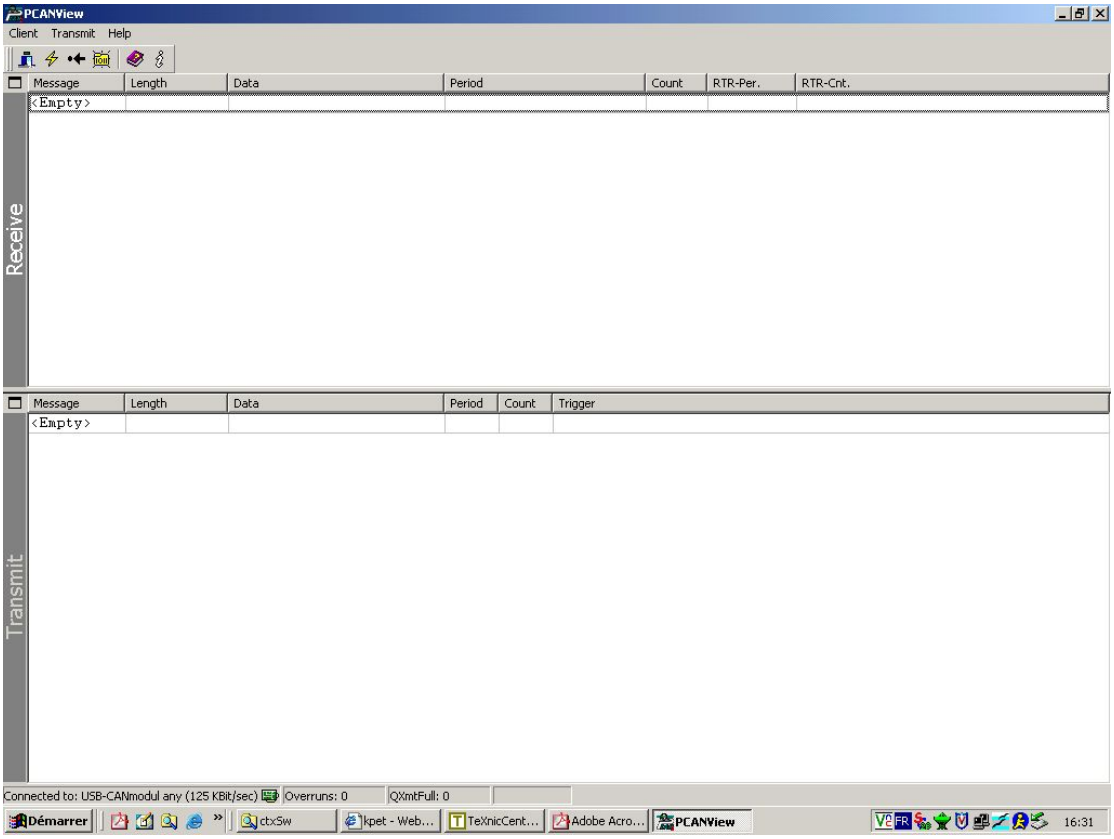

FIG. 2.5 – Interface Graphique du logiciel PcanView.

2.5.2 Le CANscope

Le CANscope est un outils très performant permettant des mesures sur le bus CAN. Il permet, à la manière d'un oscilloscope, de relever directement l'état des lignes CANH et CANL. Le CANscope est autonome et est capable d'effectuer des relevés sur le bus sans être connecté à un ordinateur. Le logiciel livré avec le CANscope permet un réglage fin des paramètres de mesure (vitesse du bus, type de trames, etc...). Les mesures peuvent être déclenchées par des évenement comme la réception d'un message prédéfini ou la présence d'erreurs sur le bus. Les informations données par le CANscope sont très détaillées et bas-niveau. J'ai peu utilisé la CANscope au profit du module CANUSB qui autorisait une vue plus globale de ce qui se passait sur le bus.

2.6 La chaîne Keil

La chaîne Keil est un ensemble d'outils permettant le développement en langage C d'applications pour microcontrôleurs. Cette chaîne se compose d'un environnement de développement intégré (EDI) et des outils spécifiques au processeur sur lequel on travaille (compilateur, assembleur, lieur, simulateur et moniteur pour débuggage, etc...). Le simulateur et le moniteur sont accessibles directement depuis l'EDI (passage en mode Debug) et autorisent une exécution pas à pas du code source en C, ce qui rend aisé la mise au point de programmes.L'EDI de Keil propose également des fonctions de gestion de projet et facilite la navigation entre les différents fichiers sources grâce à son navigateur de projet. Nous utiliserons Keil pour écrire et mettre au point nos programmes.

2.7 Phytec Flashtools 16W

Les Phytec Flashtools permettent la gestion de la mémoire flash accéssible sur le bus externe du C167 : effacement, protection en écriture, test de virginité. Ils permettent également le téléchargement d'un fichier binaire au format Intel H86 dans cette même mémoire.

Chapitre 3

Mise en place du phyCORE et de la GPIO

Nous allons ici présentation le travail réalisé pour la mise en place des cartes fournies. Dans un premier temps, nous nous intéresserons au phyCORE puis, à la carte GPIO.

3.1 Le phyCORE

Cette section va détailler les informations nécessaires à la mise en place du phyCORE (configuration et utilisation). Commençons par détailler les informations de configuration qui nous seront utiles.

3.1.1 Configuration de la carte supportant le phyCORE

La carte phyCORE HD200 est munie de cavaliers qui doivent être placés ou retirés en fonction de la configuration désirée. Nous allons détailler la configuration qui a été utilisée ici :

JP1 à JP8 : correspondent aux deuxième port série de la carte (connecteur P1B) que nous n'utiliserons pas étant donné que l'UART correspondante n'était pas présente sur la version du phyCORE que nous avions. Les cavaliers JP1 à JP8 sont donc retirés.

JP9 : en position 1-2 permet de relier le VCC à la sortie du régulateur U1 qui fournit du 3.3V à partir d'un 5V qui vient du connecteur d'alimentation. En position 2-3, il relie le VCC au 5v en provenance du connecteur d'alimentation de la carte. Nous utiliserons la position 2-3 afin d'avoir VCC = 5V.

JP10 : permet de choisir entre les signaux $BOOT$ et $\overline{BOOT}$ lorsque l'on utilise le premier connecteur de liaison série comme source pour le signal de BOOT (nous utilisons le switch S1). JP10 sera retiré.

JP11 et JP12 : permettent de choisir la source des signaux CAN dans le cas ou l'on utilise le driver intégré à la carte HD200 avec le connecteur P2A (port 4 ou port 8). JP11 et JP12 seront retirés étant donné que nous n'utilisons pas le driver intégré à la carte HD200.

JP13 : Ce cavalier permet de choisir la tension utilisée par les drivers de ligne CAN.

JP14 et JP15 : permettent le même choix mais pour le drivers de la carte HD200 relatif au connecteur P2B. Comme nous ne l'utitlisons pas, JP14 et JP15 seront retirés.

JP16 : permet de raccorder ou non VCC2 au 3.3V. Nous n'utilisons pas cette tension, JP16 sera retiré.

JP17 : permet de raccorder ou non la diode D3 de la carte HD200 au bit 0 du port 2 du C167. Nous utiliserons ce même bit pour un switch de la carte GPIO. JP17 sera retiré ou positionné en 2X (*cf* sérigraphie carte HD200).

JP18 : Ce cavalier possède un rôle similaire au cavalier JP13.

JP20, 21, 24, 25, 26 et 27 : permettent de sélectionner les signaux en provenance du premier connecteur de port série à relier au micro-contrôleur. JP20 autorise TxD0, JP21 : RI0, JP24 : DTR0, JP25 : RTS0, JP26 : DCD0 et JP27 : RxD0. Nous n'utiliserons que les signaux TxD0 et RxD0 donc JP20 et JP27 seront positionnés, et tous les autres retirés.

JP22 et 23 : permettent de rediriger les signaux DSR0 et CTS0 vers les ports du C167 en position 1-2 ou de les utiliser comme signaux de RESET et BOOT en position 2-3. Nous n'utiliserons aucune de ces deux fonctionnalités, JP22 et JP23 seront donc retirés.

JP30 : permet de relier ou non le signal TxD issu de l'UART optionnelle (non présente sur les phyCORE dont nous disposions) à la broche 8 du deuxième connecteur CAN P2B.

JP31 et JP32 : permettent de choisir quel driver CAN est utilisé pour le premier connecteur CAN (P2A). En position 1-2 ils sélectionnent les drivers SI9200EY alors que les drivers du phyCORE sont sélectionnés en position 2-3. Nous utiliserons la position 2-3 (afin de rendre le phyCORE le plus indépendant possible de la carte HD200).

JP33 et JP34 : permettent la même opération mais avec le deuxième connecteur CAN. JP33 et JP34 seront retirés étant donnée que nous n'utilisons pas le deuxième connecteur CAN.

3.1.2 Plan mémoire

Afin de pouvoir accéder aux différents périphériques raccordés au bus externe du C167, nous avons dû choisir un plan mémoire. En fait, nous avons dû choisir deux plans mémoires : un pour une utilisation avec la mémoire flash et un pour une utilisation avec le moniteur (phase de mise au point). En effet, après un reset le C167 cherchera à exécuter le code se trouvant à l'adresse 0x0000, notre programme doit donc se trouver à cette adresse (cas de la flash). Dans le cas ou l'on utilise le moniteur celui-ci se chargera en RAM et assurera l'exécution de notre code. C'est dans ce cas la RAM qui doit se trouver à 0x0000. Les plans mémoire utilisés sont à la figure 3.1.

3.1.3 Création d'un moniteur

Le création d'un moniteur spécifique consiste à déclarer les différents emplacement mémoire et la configuration qui sera utilisée. Ceci est réalisé au moyen de registres. L'initialisation de ces registres est faite automatiquement par le moniteur. Aussi, nous n'avons simplement qu'à modifier le fichier de configuration `Config.inc` et à recompiler un nouveau moniteur.

Dans ce qui suit `$KEIL$` sera le répertoire d'installation de Keil. Faire une copie du répertoire `$KEIL$/C166/MONITOR/Phytec pC167` dans `$KEIL$/C166/MONITOR`. Renomer la copie effectuée avec un nom de votre choix. Nous avons ici choisi `Phytec pC167-IRCCYN-LCD-KP`.

(a) Avec le moniteur (b) Programme en flash

FIG. 3.1 – Plans mémoire utilisés

Dans le répertoire ou se trouve la copie du moniteur ouvrir le fichier de projet (*.uv2) dans Keil. Choisir ensuite la configuration que vous désirez dans le fichier `Config.inc`. Une fois ce fichier modifié, vous devez modifier le fichier `BOOTCOPY.BAT` afin de prendre en compte le nom de votre moniteuret d'assurer son installation. La reconstruction totale du projet compilera le moniteur et l'installera. Votre nouveau moniteur est désormais utilisable.

3.1.4 Création d'un projet pour le phyCORE

Nous allons ici donner la démarche pour créer un projet pour le phyCORE, le tout pour une utilisation avec le moniteur. Commençons par créer un dossier pour notre porjet puis démarrons Keil. Dans le menu *Project* on choisit *New project...* puis on donne un nom à notre fichier de projet.

Dans la boîte de dialogue qui apparait, on choisit un processeur Infineon C167CS-LM. Dès que l'on a validé cette boite de dialogue, Keil nous demande si l'on souhaite copier le code demarrage standard pour C167 et l'ajouter à notre projet. Nous travaillerons avec le moniteur et répondrons donc non à cette question. Il nous faut maintenant configurer notre projet.

Dans le menu *Project*, choisir *Options for Target.....* Une boite de dialogue consitituée de plusieurs onglets apparait. Dans l'onglet Target, nous allons préciser les adresses de la mémoire externe. Nous utiliserons le moniteur et donc avec 265Ko de RAM à partir de l'adresse 0. Les adresses à régler sont visibles en figure 3.1.

Les 256Ko de RAM sont ainsi utilisés. Ensuite, dans l'onglet *Output*, ne pas oublier de cocher la case *Create HEX File* qui provoquera le création d'un fichier HEX (exécutable près

Type	Adresse de début	Taille
ROM	0x0	0x4000
RAM	0x4000	0x36000

Tab. 3.1 – Adresses mémoire

à être téléchargé par le moniteur). On pourra cocher l'option *Assembly code* dans l'onglet *Listing* si l'on souhaite pouvoir conulter le code assembleur généré. Dans l'onglet *L166 Misc*, nous devons maintenant spécifier au lieur les emplacements qui seront utilisés par le moniteur, et qui ne doivent donc pas être utilisés par notre application. Avec le moniteur utilisé, les emplacements à reserver (champ *Reserve*) sont : 0x8-0xB, 0xAC-0xAF, 0x6000-0x6200, 0x6B00-0x8300. Enfin, dernière chose : il nous faut informer Keil sur le moniteur que nous allons utiliser. Ceci se fait dans l'onglet *Debug*. Il faut cocher la case *Use : Keil Monitor-166 Driver*. Cliquons ensuite sur *Settings*, le moniteur à choisir est *Phytec pC167-IRCCYN-KP-LCD*. Ce moniteur utlise le plan mémoire présenté à la figure 3.1, a. Le port série et la vitesse sont à configurer en fonction de votre matériel (Utilisation jusqu'à 57600 bps sans aucun problème). Si votre application comporte des passages qui doivent être exécutés sans interruptions, vous devez cocher la case *Stop Program Execution With : NMI Only*. Vous ne pourrez plus arrêter le programme à n'importe quel moment pendant une session de débuggage mais vous aurez la possibilité de désactiver les interruptions globalement (IEN=0).

3.1.5 Utilisation du moniteur

Lorsque l'on met au point un programme à l'aide d'un moniteur, on doit définir des *points d'arrêts* (*breakpoints*). Ces points d'arrêts sont les endroits dans votre programme ou le moniteur reprendra la main, votre programme sera alors arrêter et vous pourrez observer l'état des registres ou de la mémoire avant de relancer l'exécution (jusqu'au prochain point d'arrêt ou en mode pas-à-pas).

3.2 Les LED

La carte GPIO dispose de 8 LEDs, celles-ci ont été *wrappées* sur le port P8 comme indiqué dans le tableau 3.2.

Sérigraphie GPIO n°	Bit du port 8
LED1	P8.7
LED2	P8.6
LED3	P8.5
LED4	P8.4
LED5	P8.3
LED6	P8.2
LED7	P8.1
LED8	P8.0

Tab. 3.2 – Broches utilisées pour les LED

Le port P8 est configuré en sortie avec un étage de type push/pull. Les lignes de la fonction InitGPIO() assurant cette initialisation sont en figure 3.2

```
        //Initialisation du port 8 pour les leds
        ODP8 = 0x00; //sorties types push/pull
        DP8 = 0xFF; //tous les bits en sortie
        ; //on doit attendre un cycle machine avant d'affecter une valeur
        P8 = 0x00; //valeur initiale sur les leds
```

FIG. 3.2 – Initialisations liées aux LEDs

Deux fonctions ont été écrites pour simplifier l'utilisation des LEDs. On retrouve une fonction permettant d'allumer les LEDs une à une et une autre permettant de les utiliser comme un bargraph.

3.2.1 La fonction BarGraphLED

Cette fonction permet d'utiliser les LEDs comme un bargraph. Elle doit être appellée avec deux arguments : la valeur à afficher et la valeur maximale affichable.

Le code source de cette fonction est en figure 3.3

```
void BarGraphLED(unsigned int val, unsigned int valmax){
  char nbled,nbdec;
  nbled = (val/(valmax>>3)) & 0x00FF; //nb de LED allumées
  if (!nbled){
    P8 = 0;
        }
  else {
    P8 = 1;
    for (nbdec=1;nbdec<nbled;nbdec++){
      P8 <<= 1;//on décale
      P8++;//on allume la led de poids faible
    }
  }
}
```

FIG. 3.3 – Code source de la fonction BarGraphLED

3.2.2 La fonction SetLED

Cette fonction permet d'allumer ou d'éteindre la LED de son choix. Le premier argument demandé est le numéro de la LED (1=P8.0, 2=P8.1,...), le deuxième argument demandé est ON (1) ou OFF (0).

Le code source de la fonction SetLED est en figure 3.4.

3.3 Switchs

Nous disposerons les switchs sur le port 2 afin de pouvoir disposer des unités de capure-comparaison. Pour utiliser le switch n° 1, il a fallu changer la configuration du cavalier JP17 de la carte supportant le phyCORE. Il faut en effet désactiver la diode led D3 de cette même carte car elle est commandée par P2.0.
Le brochage utilisé est visible au tableau 3.3.

Switch n°	Bit du port P2
1	P2.0
2	P2.1
3	P2.2
4	P2.3
5	P2.4
6	P2.5
7	P2.6
8	P2.7

TAB. 3.3 – Brochage Utilisé pour les Switchs

3.3.1 Les unités de capture/comparaison

Le C167CS dispose de 32 unités de capture/comparaison. Une unité de capture permet de détecter les fronts montants ou descendants (ou les deux). A chaque front détecté, un *flag* est positionné et une interruption peut être générée. Nous ne nous intéresserons qu'à la partie capture. Ces unités sont configurables via les registres CCMx. Le registre CCM0 qui permet de configurer les 4 premières unités de capture/comparaison est en figure 3.5.

Prenons l'exemple de la ligne 0. Le bit ACC0 permet de choisir le timer associé à l'unité de capture/comparaison. Etant donnée que nous n'en utiliserons pas, nous laisserons ce bit à 0. Les 3 bits CCMOD0 permettent de choisir le mode dans lequel l'unité opérera :

- 000 = désactivée
- 001 = déclenchement sur front montant
- 010 = déclenchement sur front descendant
- 011 = déclenchement sur front montant et descendant
- les autres combinaisons correspondent aux modes de comparaison que nous n'utiliserons pas ici

A chaque unité x est associé un registre de configuration d'interruption CCxIC. Le modèle d'un tel registre est visible en figure 3.6

Le bit CCxIR (bit 7) indique si l'unité de capture s'est déclenchée ou non. Ce bit est positionné par l'unité de capture et doit être remis à zéro par logiciel dans le cas ou l'on n'utilise pas de fonction d'interruption. Le bit CCxIE (bit 6) permet d'activer ou non les interruptions pour l'unité de capture concernée. Le champ ILVL (bits 5->2) contient la priorité de l'interruption parmi les autres (timer, CAN, etc...) alors que le champ GLVL (bits 1->0) définit la priorité par rapport aux autres sources de même ILVL.

3.3.2 Implementation des switchs

Les initialisations liées aux switchs sont visibles en figure 3.7.

Pour les raisons détaillées dans la section traitant des boutons poussoirs (*cf plus bas...*), il ne faudra pas oublier de déclarer les fonctions d'interruptions de la figure 3.8 associées aux unités de capture/comparaison qui ont été initialisées.

3.3.3 Utilisation des switchs

Les switchs peuvent être utilisés de deux manières. Le programme peut scruter le drapeau (bit CCxIR) indiquant une détection (*polling*) ou utiliser une fonction d'interruption.

3.4 Boutons poussoirs

Les six boutons poussoirs disponibles sur la carte GPIO seront également placés sur le port P2, toujours afin de pouvoir utiliser les unités de capture/comparaison (voir 3.3.1). Le brochage utilisé est visible à la figure 3.4.

Key n^o	Bit du port P2
1	P2.8
2	P2.9
3	P2.10
4	P2.11
5	P2.12
6	P2.13

TAB. 3.4 – Brochage Utilisé pour les Boutons Poussoirs

Le code réalisant les initialisations nécessaires à l'utilisation des boutons poussoirs est visible en figure 3.9.

Dans le code utilisateur il ne faudra pas oublier de déclarer les fonctions de la figure 3.10. Les interruptions étant activées pour les unités de capture 8 à 13, il faut associer une routine aux vecteurs d'interruptions correspondants, faute de quoi des « sauts non contrôlés »pourraient se produire. En effet, si une interruption se produit et qu'il n'y a pas de routine associée, le CPU va sauter à l'adresse du vecteur d'interruption et comme il n'y aura aucune instruction de branchement vers une routine , il continuera en séquence. Le comportement est alors plus ou moins imprévisible (surtout lors de la phase de débuggage avec utilisation d'un moniteur).

3.5 Potentiomètres

Le C167CS dispose d'un convertisseur analogique -> numérique 10 bits. Le signal à convertir peut être choisi parmi 16 voies d'entrée, toutes disponibles sur le port 5. Le tableau 3.5 indique comment seront *wrappées* les sorties des potentiomètres de la carte GPIO :

3.5.1 Le convertisseur A/N du C167

Le convertisseur A/N du C167 peut fonctionner suivant plusieurs modes différents. Il peut en effet effectuer une conversion continue ou non sur une voie d'entrée fixe ou déterminée

```
void SetLED(unsigned char no, char etat){
if (etat) //allume la led
    P8 |= (1 << (no-1));
  else
    P8 &= ~(1 << (no-1));
}
```

FIG. 3.4 – Code Source de la fonction SetLED.

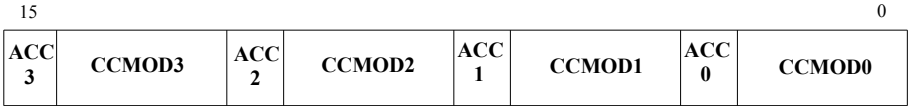

FIG. 3.5 – Registre CCM0.

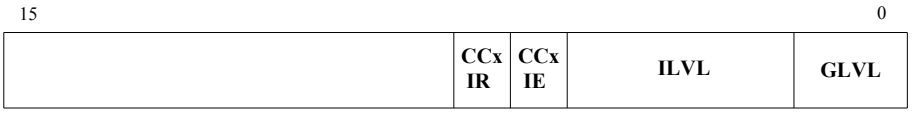

FIG. 3.6 – Registre CCxIC.

```
//Init du port 2 pour les switchs, les boutons et le codeur optique
ODP2 = 0xFFFF; //type open drain
DP2 = 0x0000; //tous les bits en entrée
...
//Initialisation des modules de capture/comparaison
CCM0 = 0x3333; //CC0->3 mode capture (fronts mont & desc), switchs 1-4
CCM1 = 0x3333; //CC4->7 mode capture (fronts mont & desc), switchs 5-8
CC0IC = 0x70; //Autorisation des IT et définition du niveau de priorité
CC1IC = 0x71; //...
CC2IC = 0x72; //...
CC3IC = 0x73; //...
CC4IC = 0x74; //...
CC5IC = 0x75; //...
CC6IC = 0x76; //...
CC7IC = 0x77; //...
```

FIG. 3.7 – Initialisations pour l'utilisation de switchs

Source	Entrée analogique de P5
Vpot1	AN0
Vpot2	AN1
Vpot3	AN2

TAB. 3.5 – Brochage Utilisé pour les Potentiomètres

```
//Fonction appellée lors d'un changement d'état du switch 1
void IntCC0(void) interrupt 0x10 {;}

//Fonction appellée lors d'un changement d'état du switch 2
void IntCC1(void) interrupt 0x11 {;}

//Fonction appellée lors d'un changement d'état du switch 3
void IntCC2(void) interrupt 0x12 {;}

//Fonction appellée lors d'un changement d'état du switch 4
void IntCC3(void) interrupt 0x13 {;}

//Fonction appellée lors d'un changement d'état du switch 5
void IntCC4(void) interrupt 0x14 {;}

//Fonction appellée lors d'un changement d'état du switch 6
void IntCC5(void) interrupt 0x15 {;}

//Fonction appellée lors d'un changement d'état du switch 7
void IntCC6(void) interrupt 0x16 {;}

//Fonction appellée lors d'un changement d'état du switch 8
void IntCC7(void) interrupt 0x17 {;}
```

FIG. 3.8 – Fonctions d'interruption associées aux switchs

```
CCM2 = 0x2222; //CC(8,9,10,11)IO en mode capture (fronts desc)
               //pour les boutons 1-4
CCM3 |= 0x0122; //CC(12,13)IO en mode capture (fronts desc)
                //pour les boutons 5-6
                //et  CC14IO (front mont) le codeur optique

CC8IC = 0x40; //Autorisation des interruption
CC9IC = 0x41; //et définition du niveau de priorité.
CC10IC = 0x42;//...
CC11IC = 0x43;//...
CC12IC = 0x44;//...
CC13IC = 0x45;//...
```

FIG. 3.9 – Initialisations liées aux boutons

automatiquement. Il dispose également d'un mode dans lequel il attend que le résultat de la conversion précédente soit lu avant d'en commencer une nouvelle. Nous utiliserons le mode simple conversion avec voie d'entrée fixe. Lorsque l'on utilise une entrée du port 5 comme entrée analogique, il vaut mieux désactiver l'étage d'entrée numérique à l'aide du registre P5DIDIS, ceci afin d'éviter que du bruit vienne parasiter le signal présent en entrée. Il y a plusieurs registres associés au convertisseur. Le registre ADCON visible en figure 3.5.1 permet d'en contrôler le fonctionnement, alors que les registres ADDAT, ADDAT2, ADCIC et ADEIC fournissent les résultats des conversions, l'état du convertisseur et permettent de configurer les interruptions qui lui sont associées.

Nous voulons utiliser le convertisseur dans le mode simple conversion sur voie d'entrée fixe(ADM=00). Le convertisseur sera par défaut initialisé pour la voie d'entrée AN0. Etant donné que les acquisitions se feront à un ryhtme peu soutenu (qq fois par seconde au max), nous n'avons pas forcement besoin d'une horloge rapide. Aussi, par souci de simplicité, nous initialiserons ADCON à 0. Les interruptions ne seront pas utilisées donc ADEIC = ADCIC = 0.

3.5.2 Utilisation des potentiomètres

Afin de simplifier l'utilisation des potentiomètres, une fonction permettant d'acquérir la valeur d'un potentiomètre a été crée. Elle est appelée avec le numéro du potentiomètre en argument et retourne une valeur correspondant à la tension présente en sortie du montage potentiométrique. La valeur retournée sera sur 8 bits, ceci afin d'en simplifier le traitement ultérieur. Ce n'est pas génant au niveau de la précision puisque les potentiomètres seront manipulés par un utilisateur qui pourra difficilement les positionner plus précisement qu'au 256^{eme} de tour près !
Le code source de la fonction GetVPot est en figure 3.12.

Cette fonction sélectionne la voie à convertir, démarre la conversion et attend la fin de la conversion. Le résultat est ensuite mis sur 8 bits. La fonction renvoie (255 - résultat) afin que la valeur minimale soit obtenue lorsque le potentiomètre est en butée à gauche (l'observateur se trouvant face à la carte GPIO).

3.6 Le moteur

Les entrées du moteur seront disposées sur le port 7 qui gère la PWM

Entrée moteur	Bit du port 7
A	P7.0
B	P7.1

Tab. 3.6 – Brochage utilisé pour le moteur

3.6.1 Les modules PWM du C167

Le C167CS dispose de 4 modules PWM configurables à l'aide de registres. Il existe quatre modes opératoires pour les modules PWM :

```
//Fonction appelée lors d'un appui sur key 1
void IntCC8(void) interrupt 0x18 {
}
//Fonction appelée lors d'un appui sur key 2
void IntCC9(void) interrupt 0x19 {
}
//Fonction appelée lors d'un appui sur key 3
void IntCC10(void) interrupt 0x1A {
}
//Fonction appelée lors d'un appui sur key 4
void IntCC11(void) interrupt 0x1B {
}
//Fonction appelée lors d'un appui sur key 5
void IntCC12(void) interrupt 0x1C {
}
//Fonction appelée lors d'un appui sur key 6
void IntCC13(void) interrupt 0x1D {
}
```

FIG. 3.10 – Fonctions d'interruption à mettre en place

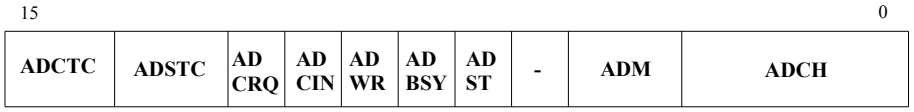

FIG. 3.11 – Registre ADCON de configuration du convertisseur A/N

```
unsigned char GetVPot(unsigned char PotNR){
        unsigned char temp;
        ADCON = (PotNR - 1);
        ADCON |= 0x80; //starts a conversion
        while(!ADCIR);
        ADCIR = 0;
        temp = 255 - ((ADDAT >> 2) & 0x00FF);
        return(temp);
}
```

FIG. 3.12 – Code source de la fonction GetVPot.

Standard PWM : Ce mode utilisable avec les quatres modules permet de générer un signal PWM « standard », c'est à dire qu'une période du signal comprend 1 temps bas x et un temps haut y.

Symmetrical PWM : Ce mode utilisable également avec les quatres modules permet de générer un signal PWM dit « symmétrique », c'est à dire que chaque période est composée d'un temp bas y, d'un temps haut $2x$ et d'un temps bas y.

Burst mode : Ce mode utilise une combinaison des modules 0 et 1 et propose un signal de sortie qui est le « ET logique » entre les sorties des modules 0 et 1.

Single shot mode : Ce mode disponible seulement sur les modules 2 et 3 permet de générer des impulsions individuellement.

Les signaux PWM sortent sur les quatres bits de poid faible de P7. La valeur présente dans les latchs de sortie de P7 influe sur la forme du signal de sortie. La valeur effectivement présente en sortie est le « OU EXCLUSIF »entre le signal généré par le module PWM et la valeur présente dans le latch de sortie.

Plusieurs registres sont associés aux modules de PWM : les regsitres associés au port P7 (ODP7, DP7 et P7), les regsitres contenant la durée de la période et la durée du temps bas (PPw et PWx), les regsitres des compteurs des modules (PTx) et enfin les registres de contrôle et de contrôle des interruptions.

Nous utiliserons le mode *Standardmode* et allons effectuer une description sommaire des registres utiles pour sa mise en oeuvre. Les registres PPx, PWx et PTx ne comportent qu'un seul champ de données de 16bits dont le contenu vient d'être décrit. Le mode d'interruption ne sera pas utilisé, aussi, nous ne parlerons pas du registre permettant de le configurer (PWMIC). Le registre de configuration PWMCON0 est visible en figure 3.6.1.

15															0
PIR 3	PIR 2	PIR 1	PIR 0	PIE 3	PIE 2	PIE 1	PIE 0	PTI 3	PTI 2	PTI 1	PTI 0	PTR 3	PTR 2	PTR 1	PTR 0

FIG. 3.13 – Le registre PWMCON0.

Les 8 bits de poids fort de ce registre sont utilisés en mode d'interruption, nous n'en parlerons pas. Chaque bit PTRx définit si le timer associé au module x est en fonctionnement ou non. Chaque bit PTIx permet de sélectionner le facteur de division de l'holorge du CPU utilisé pour obtenir l'horloge de ce même timer (0->1, 1->64). Le deuxième registre de configuration PWMCON1 est en figure 3.6.1.

15															0
PS1	PS2	-	PB 01	-	-	-	-	PM3	PM2	PM1	PM0	PEN 3	PEN 2	PEN 1	PEN 0

FIG. 3.14 – Le registre PWMCON1.

Les bits PENx de ce registre permettent d'activer le module de PWM x, PEN0 et PEN1 seront mis à 1. Les bits PMx servent à sélectionner le mode standard ou symmétrique (0->Standard, 1->Symmétrique), nous placerons PM0 et PM1 à 0. Le bit PB01 permet d'activer le burst mode, nous le laisserons à 0. Les bits PS1 et PS2 permettent d'activer le mode single shot et seront donc laissé à 0 puisque nous ne l'utilisons pas.

Les modules PWM sont initialisés comme visible à la figure 3.15.

```
#define MAXPWM 40000
...
//Initialisation du port 7 pour le moteur en PWM
ODP7 = 0x00; //sorties push/pull
DP7 = 0xFF; //tous les bits en sortie
; //P7 est toujours en entrée
P7 = 0x00; //Le signal PWM ne sera pas inversé
           //car les latchs de P7 contiennent des 0
           //(x XOR 0 = x)
...
//Initialisation des modules de PWM
PP0 = MAXPWM;
PW0 = MAXPWM;
PP1 = MAXPWM;
PW1 = MAXPWM;
PWMCON0 = 0x0000;
PWMCON1 = 0x0003;
PWMIC = 0;
```

FIG. 3.15 – Initialisation des modules PWM.

3.6.2 Les routines de commande du moteur

Une fonction permettant une commande facile du moteur a été écrite. La fonction CMDMoteur est appellée avec trois paramètre :

Le code de la fonction CMDMoteur est visible en figure 3.16.

La fonction CMDMoteur() s'appelle avec trois paramètres. Le premier paramètre qui peut prendre une valeur nulle ou non-nulle indique si le moteur doit tourner ou non. Le deuxième paramètre indique le sens de rotation : s'il est nul le moteur tournera dans un sens, s'il est non-nul le moteur tournera dans l'autre sens. Enfin, le troisième paramètre informe la fonction sur la vitesse désirée. Ce troisième paramètre peut varier de 0 à (MAXPWM / 2). Comme indiqué en figure 3.15, MAXPWM vaut 40000 par défaut.

3.7 Codeur optique et vitesse du moteur

La sortie du codeur optique a été *wrappée* sur P2.14. Nous pouvons ainsi utiliser les unités de capture/comparaison. Les initialisations faites afin de pouvoir l'utiliser sont en figure 3.17.

La fonction d'interruption associée est en figure 3.18.

La vitesse de rotation du moteur peut être mesurée de différentes façons. On peut par exemple, compter le nombre de tours en une seconde (simple mais peu précis) ou mesurer le temps mis pour faire un tour et en déduire la vitesse. L'unité GPIO ne proposera pas de service de mesure de vitesse. Celui-ci devra donc si besoin est être implémenté par l'utilisateur.

```
void CMDMoteur(int onoff, int sens, unsigned int val){
        if(onoff){ //si onoff = ON
                if (sens){
                        PTR0 = ON;
                        PTR1 = OFF;
                        PW1 = MAXPWM;
                        PW0 = MAXPWM - val;
                }
                else {
                        PTR0 = OFF;
                        PTR1 = ON;
                        PW0 = MAXPWM;
                        PW1 = MAXPWM - val;
                }
        }
        else { // si onoff = OFF
                PW0 = MAXPWM;
                PTR0 = OFF;
                PW1 = MAXPWM;
                PTR1 = OFF;
        }
}
```

FIG. 3.16 – Code Source de la fonction CMDMoteur.

```
CCM3 |= 0x0122; //CC14IO (front mont) pour le codeur optique
...
CC14IC = 0x46; //Autorisation des IT et def du niveau de priorité
```

FIG. 3.17 – Initialisations pour le codeur optique

```
//Fonction appelée à chaque tour du moteur
void IntCC14(void) interrupt 0x1E {
}
```

FIG. 3.18 – Fonction appelée à chaque tour du moteur

3.8 L'afficheur LCD

La carte GPIO founit un connecteur permettant de brancher un afficheur LCD de type x lignes de y caractères (contrôleur HD44780 ou compatible). L'afficheur LCD est connecté directement au bus externe du C167 à l'aide de quelques portes logiques. Une des broches de sélection de boitier du C167 (en l'occurence /CS2) lui a été dédié. Une plage mémoire également. Soit @LCD l'adresse de début de la plage mémoire associée au LCD (ici 0x100000). Une écriture d'un octet à @LCD provoquera l'envoi d'une instruction au LCD. De même, une écriture d'un octet à @LCD+4 provoquera l'envoi d'une donnée au LCD. La lecture d'une instruction ou d'une donnée se faisant respectivement à l'adresse @LCD+2 ou @LCD+6. Ces informations ont été prises en compte lors de la réalisation du moniteur présenté plus haut.

L'afficheur doit ensuite être « déclaré » au CPU. Cette « déclaration » se fait par l'intermédiaire des resistres ADDRSEL2 et BUSCON2. Certains champ doivent être configurés de sorte à ralentir les accès au bus externe lors de l'utilisation de la plage mémoire associée au LCD. Cette configuration est déjà faite dans le moniteur `Phytec pC167-IRCCYN-LCD-KP`. Si le LCD doit être utilisé avec un programme tournant depuis la mémoire flash, il faudra penser à faire cette déclaration (modification du fichier contenant le code de démarrage).

3.8.1 Driver de haut niveau

Plusieurs fonctions ont été écrite afin de simplifier l'utilisation de l'afficheur LCD. L'implementation de ces fonctions sere présentée ainsi que leur utilisation.

SendInsLCD

Cette fonction permet l'envoi d'une instruction au LCD. Son code source est visible en figure 3.8.1.

```
void SendInsLCD(char Instr)
{
   while ((LCD_InsRead & 0x80) != 0); /* wait until not bit 7 */
   LCD_InsWrite = Instr; /* write instruction to VFD */
}
```

FIG. 3.19 – Code source de la fonction SendInsLCD.

La fonction attend simplement que le LCD ne soit plus « busy » puis effectue l'envoi de l'instruction passée en paramètre.

SendDatLCD

Cette fonction permet l'envoi d'une donnée au LCD. Son code source est visible en figure 3.8.1.

La fonction attend simplement que le LCD ne soit plus « busy » puis effectue l'envoi de la donnée passée en paramètre.

Wait

ClearLCD

Cette fonction efface l'écran de l'afficheur en lui envoyant l'instruction `0x01`. Elle n'attend pas que l'instruction soit effectuée avant de retourner, ce n'est en effet pas nécessaire puisque les fonctions SendDatLCD et SendInsLCD qui pourraient être appellée après attendent que le LCD soit libre avant d'y écrire. Le code source de cette fonction est en figure 3.8.1.

InitLCD

setLine

putchar et utilisation de printf

La fonction `printf` permet d'afficher des chaines de caractères formattées, autorisant un affichage simple de la valeur d'une variable ou d'un pointeur. Par défaut, sur C167, la fonction `printf` utilise la liaison série pour l'envoi des chaines formattées. En fait, `printf` appelle `putchar` à chaque fois qu'un caractère doit être envoyé. Si on définit notre propre fonction `putchar` capable d'écrire un caractère sur le LCD, alors celle-ci sera appellée automatiquement par `printf` et la sortie de `printf` se fera sur le LCD. Le code source de la fonction `putchar` est visible en figure 3.22.

3.9 Connecteurs d'entrée/sortie

Ces connecteurs ne sont actuellement pas *wrappés* mais il reste de la place pour le faire. Le connecteur d'entrée pourrait être relié au port 5 tandis que le connecteur de sortie pourrait être relié au port 3.

```
void SendDatLCD (char Data)
{
   while ((LCD_InsRead &0x80) != 0); /* wait until not bit 7 */
   LCD_DataWrite = Data; /* write data to VFD */
}
```

FIG. 3.20 – Code source de la fonction SendDatLCD.

```
void ClearLCD(void){
        SendInsLCD(0x01);
}
```

FIG. 3.21 – Code source de la fonction ClearLCD.

```
signed char putchar (signed char Data)
{
        unsigned char linecounter;
          if(Data == '\r')
          {
                  SendInsLCD(lines[linecounter]); /* set DD-RAM adress by 0  Line 0 */
          }
          else if(Data == '\n'){
                  linecounter++; if (linecounter==LINES) linecounter=0;
            SendInsLCD(lines[linecounter]); /* set DD-RAM adress by 40 Line 2 */
        }
        else{
                SendDatLCD(Data);
        }
          return(Data);
}
```

FIG. 3.22 – Code Source de la fonction putchar.

Chapitre 4

Mise en place de la communication CAN

4.1 Le Bus CAN

4.1.1 Présentation

Le protocole CAN (Controller Area Network) est un protocole pour bus à communication série. Il a été developpé par Bosch dans les années 80 et est très utilisé dans l'industrie en raison de son faible coût et de sa fiabilité. C'est un bus multi-maîtres asynchrone, qui n'utilise donc pas d'horloge transmise. Chacun des noeuds possède son horloge locale et peut commencer à transmettre dès que le bus est libre. Ces horloge locales sont resynchronisées à l'aide des changement d'état du bus (fronts descendants). Des bits de bourrage (*stuffing*) sont ajoutés au milieu de séquence de plus de 5 bits identiques afin d'éviter une perte de synchronisation entre les différents noeuds. Le bus peut présenter deux états : l'état récessif correspondant au 1 logique et l'état dominant correspondant au 0 logique.

Si, à un instant donné, une station émet un bit récessif et une autre station un bit dominant, le bus présentera l'état dominant. A chaque instant, l'état du bus est le ET cablé (logique positive) entre les états que les stations veulent imposer. Le protocole CAN autorise des tranmissions jusqu'à la vitesse de 1Mbit/s sur une distance de 40m. La distance peut toutefois être augmentée en réduisant la vitesse de transmission (ex : 500m @ 125Kbit/s). Les données sont échangées sous forme de *trames*. Deux types de trames (plus trois types de trames d'erreur) peuvent circuler sur le bus :

Trame de donnée : Une trame de donnée est renferme deux informations : un identifiant et des données (8 octets max). L'identifiant est utilisé pour identifier le message ainsi que pour résoudre les problèmes de collision. La constitution d'une trame de données est visible en figure 4.1.

Trame de requête : Un noeud demande à un autre noeud l'envoi de la trame de données dont l'identifiant et le même que celui de la trame de requête. Une trame de requête est visible en figure 4.1

La taille totale d'une trame de données dépend du nombre de bits ajoutés par le *bit stuffing* et donc directement de la valeur des données transmises. On peut toutefois majorer cette taille en considérant le pire des cas. Dans une trame de donnée ou de requête, les champs ARBITRATION, CONTROL, DATA et CRC sont soumis au « *bit stuffing* ». Si n est le nombre

Nom du champ	Longueur en bits	Description
SOF (Start Of Frame)	1	Permet la détection du début de la trame (Au repos le bus est dans l'état récessif).
ARBITRATION	12	Les 11 premiers bits de ce champ représentent l'identifiant de la trame. Le dernier bit (RTR) est à 0 dans le cas d'un trame de données et à 1 dans le cas d'une trame de requête.
CONTROL	6	Les bits IDE et r0 sont à 0 tout le temps. Les 4 bits suivants représentent le champ DLC (Data Length Code). Leur valeur donne le nombre d'octets de donnée.
DATA	0 à 64	Ce champ contient les données de la trame (de 0 à 8 octets). Ce champ est vide dans le cas d'une trame de requête.
CRC (Cyclic redundancy checksum)	16	Les 15 bits de ce champ correspondent au code détecteur d'erreur et le dernier bit correspond au *CRC delimiter*, toujours récessif (à 1).
ACK (Acknoweledgement)	2	Le premier bit *ACK slot* est récessif dans le cas d'un émetteur et dominant dans le cas d'un récepteur. Le second *ACK delimiter* est toujours récessif.
EOF (End Of Frame)	7	Les sept bits de ce champ sont récessifs
INTERMISSION	3	Ces trois bits sont récessifs et font en fait partie de l'« *Interframe Space* » ou espace inter-trame. Après ces trois bits, le bus est au repos (*idle*) pour une durée qui n'est pas nécesairement multiple d'un temps bit. Lorsque le bus est libre, son état est récessif.
Total	**47 à 111**	

TAB. 4.1 – Format d'une trame CAN standard (CAN 2.0A).

d'octets de données de la trame, alors le nombre maximum de bit qui seront insérés est donné par : $(34 + 8n)/5$[1]. La taille maximum d'une trame CAN standard est donc :

$$47 + 8n + \frac{34 + 8n}{5} \tag{4.1}$$

4.1.2 Gestion des erreurs

Le protocole CAN possède une gestion très stricte des erreurs. Chaque contrôleur possède deux compteurs : TXEC (Transmission Error Count) et RXEC (Reception Error Count). En fonction des réussites ou échecs de transmission (ou réception), ces compteurs sont incrémentés (échec) ou décrémentés si non nuls (succès). Un contrôleur CAN possède trois états :

Il existe trois types de trames d'erreur :

[1] 34 est la somme des tailles des champs ARBITRATION, CONTROL et CRC ; 8n est la taille du champ DATA ; On divise par 5 car les bits de stuffing sont présents tous les 5 bits au maximum

État	Valeur des compteurs	Comportement
Error active	(TXEC<128)&&(RXEC<128)	C'est la situation à la mise sous tension, en cas d'erreur le contrôleur émet une trame d'erreur active.
Error passive	(TXEC<256)&& ((TXEC>=128)||(RXEC>=128))	Il s'est produit une erreur grâve. En cas d'erreur le contrôleur émet une trame d'erreur active puis attend durant 7 bits (émission de bits récéssifs).
Bus-off	TXEC>=256	Le contrôleur n'a plus aucune action sur le bus. Il envoie en continu des bits récessifs et ne prend pas en compte ce qu'il reçoit. Le contrôleur se place de lui même dans l'état bus-off en cas d'erreurs répétées et n'en sortira qu'à la suite d'une intervention du CPU

TAB. 4.2 – États d'un contrôleur CAN.

Trame d'erreur active : C'est la trame envoyée en cas d'erreur lorque le contrôleur est dans l'état *Error active*. Elle se compose de l'émission de 6 bits dominants suivis par l'émission d'une séquence de 6 bits récessifs interrompue dès observation d'un bit récessif sur RxD. Un *error delimiter* constitué de 7 bits récessifs est ensuite émis.

Trame d'erreur passive : C'est la trame envoyée en cas d'erreur lorsque le contrôleur est dans l'état *Error passive*. Son émission consiste en l'envoi de bits récessifs jusqu'à l'occurrence de 6 bits de même polarité puis l'émission de 6 bits récessifs interrompue dès qu'un bit récessif est observé sur RxD. Huit bits récessifs (*error delimiter* sont ensuite transmis.

Trame de surcharge : Cette trame est envoyé par tout noeud qui détecte une surcharge du bus. Elle consiste en l'émission de 6 bits dominants consécutifs, puis celle de bits récessifs jusqu'à observation d'un bit récessif sur RxD et enfin celle de 7 bits récessifs consécutifs

4.1.3 Estimation de la charge du bus

Lorsque l'on connait la taille et la fréquence d'envoi des messages pour chaque noeud ainsi que la vitesse utilisée dans l'application (V), il devient possible d'estimer par calcul la charge du bus. La première étape consiste à déterminer le nombre de messages par seconde ainsi que la pire taille possible pour chacun des messages, on utilise pour cela la formule 4.1. On multiplie ensuite la pire taille de chacun des messages par son nombre d'occurences par seconde. On fait la somme de tous les produits obtenus précedement : on obtient le nombre total de bits échangés par seconde (N_B). La charge du bus est alors : $\frac{N_B}{V} \times 100$.

Prenons l'exemple de l'application suivante qui n'est pas du tout réaliste mais permettra de bien comprendre le problème.Un batiment est equipé avec des capteurs de température de pression et des surveillants pour machines. Les 20 capteurs de température transmettent une information de 2 octets 20 fois par seconde. Les 100 capteurs de pression transmettent une information de 1 octet 10 fois par seconde. Les 10 surveillants de machine tranmettent l'état de la machine (10*8 octets, 10 fois par seconde). La transmission se fait à 500 Kbits/s.

D'après la formule 4.1, la pire taille des trames de cette exemple sont :

Nombre d'octets de données	Pire taille (bits)
1	63.4
2	73
8	130.6

A partir de ces informations, nous allons calculer la quantité de données échangées par seconde.

Le nombre total de bits envoyés par les capteurs de température est : $20 \times 20 \times 73 = 29200$ bits/s. Les capteurs de pression envoient $100 \times 10 \times 63.4 = 63400$ bits/s. Et les surveillants $10 \times 10 \times 10 \times 130.6 = 130600$ bits/s.

Le traffic total est de 223200 bits/s et la vitesse de transmission est de 500 Kbits/s, la charge du bus est :

$$\frac{223200}{500000} \times 100 = 44,64\%$$

Ce résultat est toutefois à prendre avec précautions, en effet, le calcul effectué ne tient pas compte des eventuelles collisions et trames d'erreurs. Pour conserver un fonctionnement correct, il est conseillé de ne pas dépasser une charge de 60%.

4.2 Le driver CAN utilisé

Le driver CAN utilisé provient d'une note d'application de Siemens. Il se compose d'un jeu de fonctions permettant une gestion complète et à haut niveau du contrôleur CAN embarqué dans le C167.

Voici les fonctions disponibles ainsi que leur rôle :

init_can_16x() : Cette fonction permet d'initialiser le contrôleur CAN et de paramétrer la vitesse qui sera utilisée pour la transmission ainsi que d'activer ou non la génération d'interruptions.

def_mo_16x() : Cette fonction assure l'initialisation des messages objets du contrôleur CAN :
- type d'identifiant : 11 ou 29bits
- identifiant
- objet utilisé en émission ou réception
- nombre d'octets de données
- configuration des interruptions : transmission et réception individuellement

ld_modata_16x() : Effectue le chargement des données d'un message dans le contrôleur CAN.

send_mo_16x() : L'appel de cette fonction provoque l'envoi du message CAN (préalablement définit puis chargé à l'aide des deux fonctions précédentes) dont le numéro est passé en paramètre. L'utilisation avec un message objet configuré en réception provoquera l'envoi d'une trame de requête.

check_mo_16x() : Vérifie si de nouvelles donnnées sont présentes pour le message dont le numéro est passé en paramètre

check_mo15_16x() : Même chose que la fonction précédente mais pour le message 15

rd_mo15_16x() : Permet de copier les données, l'identifiant et le DLC du message 15 en mémoire (3 pointeurs demandés en paramètre)

rd_modata_16x() : Copie les données du message dont le numéro est passe en premier paramètre à l'emplacement mémoire désigné par le pointeur passé en deuxième paramètre.

check_busoff_16x() : Renvoie une valeur non-nulle si le contrôleur est dans l'état *BusOff* puis quitte cet état, sinon renvoie une valeur nulle.

Quelques détails d'utilisation :

La fonction init_can_16x() doit être ABSOLUEMENT appellée la première. Dans le cas ou l'on active une ou plusieurs interruptions avec cette fonction, il ne faut pas oublier d'activer globalement les interruptions pour le contrôleur CAN (par exemple à l'aide du registre XP0IC pour le premier contrôleur can d'un C167CS-LM).
Les messages doivent ensuite être configurés à l'aide de la fonction def_mo_16x().
L'envoi d'un message à l'aide de la fonction send_mo_16x() sera realisé seulement si la fonction ld_modata_16x() a été appellée précédement pour le même message (même si le champ DATA de la trame est vide). Ajoutons à cela que la fonction send_mo_16x() n'est pas blocante, aussi, afin d'être sur de ne pas surcharger le contrôleur CAN de requêtes d'émission, il est préférable d'attendre l'indicateur de fin d'émission (bit TXOK du registre CSR) puis de le mettre à zéro.

4.3 Les cables

Les câbles utilisés permettent de chainer les noeuds. En effet, chaque câble possède un connecteur mâle et deux connecteurs femelles permettant ainsi d'étendre le bus jusqu'à n noeuds. Les connecteurs utilisés étant de type SUB-D 9. Le câble utilisé comportait deux âmes et une tresse de blindage. La tresse est reliée aux broches 3 et 6 des connecteurs, ce qui correspond au « GND ». Le fil marron est relié à la patte 7 des connecteurs et correspond au signal CAN-H. Le fil blanc, quant à lui, est relié à la patte 2 des connecteurs et correspond au signal CAN-L.

Chapitre 5

Présentation d'OSEK et de Trampoline

5.1 OSEK/VDX OS

Le groupe OSEK distribue gratuitement les spécifications d'un OS temps réel pour l'embarqué : OSEK/VDX OS. Un OS temps réel est un système d'exploitation adapté aux contraintes du temps réel. Il permet d'assurer l'exécution de plusieurs tâches sur un même processeur. Chaque tâche se voit attribuer un contexte d'exécution (copie en mémoire des registres et des piles). Le système permet de basculer d'un contexte à un autre et donc de l'exécution d'une tâche à une autre.

5.2 Tâches et ordonnancement

5.2.1 Les tâches

Une tâche dans OSEK/VDX OS est une fonction C standard qui est appellée sans paramètres et qui n'en retourne aucun. Chaque tâche possède une priorité statique, c'est à dire fixée à la compilation. Dans OSEK, on distingue deux types de tâche : les tâches étendues et les tâches basiques. La distinction se situe au niveau des différents états que peuvent prendre les tâches. Une tâche basique peut prendre 3 états :

RUNNING : La tâche possède le CPU et s'exécute. Une seule tâche peut être dans cet état à la fois alors que tous les autres états peuvent être attribués à plusieurs tâches simultanement.

READY : La tâche est prête à être exécutée et attend que l'ordonnanceur lui attribue le CPU.

SUSPENDED : Dans cet état, la tâche est passive et peut être activée

Une tâche étendue peut en plus prendre l'état **WAITING**. Dans cet état, la tâche est en attente d'un *évènement*

Une tâche peut effectuer des appels à des services systèmes. La spécification OSEK/VDX OS propose l'implémentation d'un certain nombre de services permettant par exmple d'activer une tâche ou de masquer les intérruptions.

5.2.2 L'ordonnancement

Chaque tâche possède un contexte d'exécution, c'est à dire un environnement dans lequel elle s'exécutera et qui lui est propre. L'ordonnancement est l'opération permettant le passage d'un contexte d'exécution à un autre et donc d'une tâche à une autre. L'ordonnanceur est appellé à chaque point de *re-scheduling* ou point de ré-ordonnacement. En pratique l'ordonnanceur sera toujours appellé explicitement par les services systèmes, le passage d'une tâche à une autre ne pourra s'effectuer que lors de l'appel à ces services.

Toutes les tâches qui sont dans l'état **READY** sont placées dans une liste ordonnée en fonction de la priorité des tâches, la plus prioritaire est étant en haut de la liste. L'ordonnanceur, lorsqu'il est appellé, compare la priorité de la tâche en cours d'exécution avec celle de la tâche qui se trouve au sommet de la liste des tâches « **READY** », il donne le CPU à la plus prioritaire des deux.

5.3 Gestion des interruptions

A la suite d'une interruption, le CPU doit être passé à une ISR (*Interrupt Service Routine*). On distingue deux types d'ISR :

Les ISR de **catégorie 1** qui n'ont pas la possibilité d'effectuer des appels systèmes. Leur exécution n'est pas contrôlée par le système et elles s'exécutent dans le contexte de la tâche ou de l'ISR de catégorie 2 interrompue (cela doit être pris en compte lors de l'attribution des tailles de piles pour les tâches et ISR2).

Les ISR de **catégorie 2** ont la possibilité d'effectuer certains appels systèmes. Elles doivent donc posséder leur propre contexte d'exécution.

5.4 Le PCP : Priority Ceiling Protocool

Ce mécannisme a été mis en place afin d'éviter les inversions de priorité entre tâches lors de la prise de ressources. En effet, imaginons le scénario suivant : une application constituée de 3 tâches, de priorités 1, 2 et 3. La tâche de priorité 1 est en cours d'exécution et les deux autres sont **SUSPENDED**. Cette même tâche prend la ressource X (*GetResource(X)*) puis active la tâche de priorité 3. La tâche de priorité 3 prend le CPU mais ne peut pas s'exécuter complètement car elle ne peut pas prendre la ressource X occupée par la tâche de priorité 1. L'ordonnanceur va donner le CPU à la tâche de priorité 2 jusqu'à la fin de celle-ci. L'ordonnanceur ne donnera le CPU à la tâche de priorité 3 que lorsque la tâche de priorité 1 aura relâché la ressource X (*ReleaseResource(X)*) et que la tâche de priorité 2 sera terminée. La tâche de priorité 2 a pris le CPU alors qu'elle était moins prioritaire.

Pour éviter ce genre de problèmes, le PCP (Priority Ceiling Protocool) a été mis en place. Avec le PCP, chaque ressource possède une priorité plafond (forcement supérieure à celle de toutes les tâches succeptibles de la prendre). Lorsque qu'une tâche prend une ressource, si sa priorité est inférieure à la priorité plafond de la ressource qu'elle est en train de prendre alors sa priorité est changée dynamiquement de façon a ce qu'elle soit égale à celle de la ressource. Reprenons notre exemple précédent en attribuant une priorité de 5 à la ressource X. Avec le PCP, lorque la tâche de priorité 1 activera la tâche de priorité 3, le CPU laissera le CPU à la tâche de priorité 1 qui sera plus prioritaire (elle a pris la priorité de la ressource). La tâche de priorité 1 relâchera la ressource X, et reprendra sa priorité originelle (1). La tâche de priorité

3 qui a été activée prendra alors le CPU avant la tâche de priorité 2. Le PCP a permis d'éviter l'exécution de la tâche de priorité 2 avant celle de la tâche de priorité 3.

Chapitre 6

Portage de trampoline

6.1 Qu'est ce qu'est le portage ?

Porter c'est adapter un code source écrit pour une architecture à une autre architecture. Certaines parties ne nécessitent aucune adaptation dans la mesure ou elle reposent pas une spécificité du matériel. De manière générale, doivent être portés :

- Les routines en assembleur
- Les routines utilisant des spécificités de la machine
- Les routines utilisant des périphériques spécifiques (cas des microncrontrôleurs)

6.2 Les fonctions spécifiques à la machine de trampoline

6.2.1 Initialisation de la machine

Cette fonction permet d'effectuer des initialisations spécifiques à la machine avant l'initialisation et le démarrage de Trampoline. Elle ne fait actuellement rien mais c'est elle qui devra assurer l'initialisation d'un timer lorsque les mécannismes d'alarme OSEK seront mis en place.

6.2.2 Initialisation des contextes

Les contextes d'exécution des différentes tâches de l'application utilisateur seront initialisées un à un par le système au démarrage. La fonction réalisant cette initialisation est `tpl_init_context()`. Comme son nom l'indique cette fonction qui est appelée par le système initialise le contexte de la tâche passée en paramètre. Chose des plus importantes, elle initialise le pointeur le pile de la tâche et place dans la pile de la tâche l'adresse du point d'entrée de celle-ci. Le passage d'une tâche à l'autre sera détaillé plus dans la sous-section suivante.

6.2.3 Changement de contexte

Le contexte d'éxecution d'une tâche représente l'environnement dans lequel elle s'éxecutera. Dans le cas du C167, cet environnement comprendra tous les registres « systèmes ». En effet, lorque l'on passera d'une tâche en cours d'éxecution à une autre, il faudra s'assurer que les données placées dans la pile par exemple seront maintenues intactes jusqu'au retour dans la

tâche interrompue. Si chaque tâche possède son propre contexte d'éxecution et qu'elle s'éxecute uniquement dans celui-ci, alors on peut préserver les données de la pile ou les registres GPR lors du passage d'une tâche à une autre.

Dans Trampoline, le changement de contexte est assuré par une fonction spécifique à la machine (écrite en assembleur pour maîtriser tout ce qui est fait). La fonction de changement de contexte est appellée par l'ordonnanceur avec deux paramètres : un pointeur vers le contexte que l'on quitte et un vers celui que l'on va mettre en place. La fonction va ensuite sauvegarder le contexte que l'on quitte (registres systèmes, piles). Les registres et les piles ne sont pas sauvegardés completement, seuls les pointeurs y autorisant l'accès sont sauvegardés. Une fois l'ancien contexte d'exécution sauvegardé, la fonction de changement de contexte va se charger de mettre en place le nouveau contexte. Les registres SFR systèmes vont tous êtres restaurés (sauf les pointeurs vers les registres : CP et la pile : SP). La sauvegarde du mot d'état puis de l'adresse (2 mots : Code Segment Pointer et Instruction Pointer) ou doit recommencer l'exécution de la tâche sont placés dans la pile. Le resgitre SP est ensuite restauré puis le registre CP. La pile et les resistres de la tâche vers laquelle on va basculer sont en places. Il reste une seule instruction assembleur à exécuter dans notre routine de changement de contexte : *RETI*. Cette instruction sert normalement à revenir d'une routine d'interruption. Nous l'utilisons ici car cette seule instruction dépile deux mots qu'elle place dans les registres CSP et IP (il s'agit de l'adresse ou le CPU continuera l'exécution de code au prochain coup d'horloge) puis dépile le mot d'état PSW. C'est en effet la seule manière de restaurer correctement PSW dans la mesure ou n'importe quelle instruction de déplacement l'affecte.

Chapitre 7

Application de démonstration

Comme evoqué lors de la présentation du sujet de stage, deux applications de démonstration seront developpées. Une première application très simple permettant de vérifier le bon fonctionnement des cartes puis une application plus complète, mettant en oeuvre la communication CAN et Trampoline.

7.1 Premiers tests

La première application de test developpée est très simple. Prévue pour tourner depuis la mémoire flash, elle permettra de vérifier rapidement le bon fonctionnement de tous les éléments de la carte GPIO (sauf LCD).

Chaque switch permet d'allumer ou d'éteindre une LED. Les boutons poussoir 1 à 3 déclenchent l'affichage en bargraph de la valeur des potentiomètres 1 à 3. Le bouton poussoir 6 permet de faire tourner ou non le moteur et les boutons 4 et 5 permettent respectivement de diminuer ou d'augmenter sa vitesse. Le code source de cette application est disponible en annexe B.1

7.2 Présentation de la deuxième application

Nous allons développer une application de démonstration basée sur trampoline afin tester ce dernier et de mettre en avant les possibilités qu'il offre. Le cas d'étude proposé est un système de pont roulant sur rail. Le pont permet de monter et descendre une charge ainsi que de la déplacer longitudinalement.

On dispose d'un moteur sur chaque rail permettant d'assurer la translation du pont roulant, et d'un troisième destiné à monter et descendre la charge. Le maintien de la perpendicularité entre le palan et les rails impose une synchronisation forte entre les deux moteurs chargés de la translation longitudinale du palan. Des capteurs de fin de course indiquent les extrémités des deux rails ainsi que la fin de la montée ou de la descente. L'utilisateur réglera la position demandée puis déclenchera un départ cycle. Un bouton d'arrêt d'urgence sera disponible.

7.3 Spécifications, cahier des charges

Ce système utilise trois microcontrôleurs *Infineon C167CS-LM* reliés par un bus CAN.
Les trois noeuds CAN auront les roles suivants :

– Un noeud de supervision
– Deux noeuds de contrôle des moteurs

Le noeud de supervision assurera la montée ou la descente de la charge.

L'architecture fonctionnelle de notre système est visible en figure 7.1. Nous voulons que les codes sources des programmes tournants sur les noeuds 1 et 2 soient les mêmes, ceci afin d'avoir seulement deux programmes à concevoir et debugger.

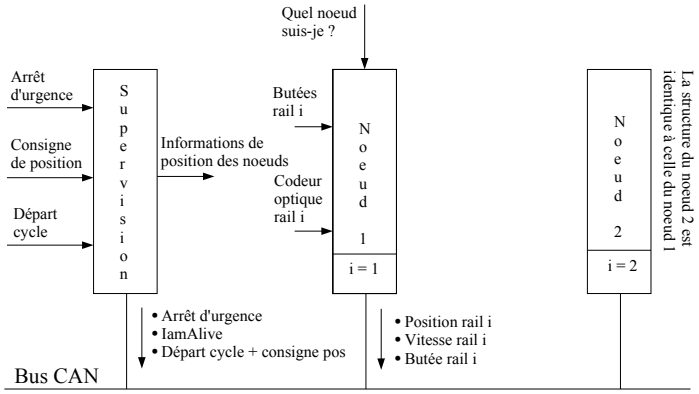

FIG. 7.1 – Architecture Fonctionnelle

Les différents messages CAN envoyés par le superviseur et les noeuds sont dans le tableau n° 7.1.

	Nom du message	ID	Nb d'octets de données
Superviseur	Arrêt d'urgence	0x000	0
	IAmAlive	0x001	0
	DCY + Consigne position	0x100	4
Noeuds	Position Noeud 1	0x20	4
	Position Noeud 2	0x21	4
	Butée Noeud 1	0x10	0
	Butée Noeud 2	0x11	0
	Vitesse Noeud 1	0x30	2
	Vitesse Noeud 2	0x31	2

TAB. 7.1 – Messages CAN envoyés par les noeuds

Les cartes GPIO seront utilisées afin de simuler le comportement du matériel.

7.4 Réalisation

La réalisation de cette application n'a pu être beaucoup avancée, pour de multiples raisons. Les divers bugs découverts dans Trampoline ainsi que le manque de temps n'ont pas permis de terminer le developpement de cette application.

Chapitre 8

Conclusion

Arrivés au terme de ce stage, nous allons effectuer un point. Ce stage en milieu professionnel m'a apporté diverses expériences tant au niveau technique que humain. Il m'aura tout d'abord permis d'avoir une expérience du travail en équipe et de ses contraintes. J'ai pu être sensibilisé par exemple aux problèmes de communication (plusieurs personnes se trouvant dans des lieux différents et qui doivent travailler ensembles) ou de synchronisation de fichiers (plusieurs personnes travaillent sur le même code source, par exemple). Ce stage m'aura également permis d'acquérir une plus grande méthode dans la résolution de problèmes et une meilleure gestion de mon temps.

Coté technique, j'ai appris de nombreuses choses comme la rédaction de documents en LaTeX 2_ε ou l'utilisation de logiciels de gestion de version. J'ai maintenant et grâce à ce stage une bonne connaissance du C167 et des outils de developpement associés. J'ai pu également avoir un apperçu du domaine des *systèmes d'exploitation temps réels* et participer au portage d'une implementation d'OSEK. Cette expérience m'a également permis d'enrichir fortement mes connaissances en programmation sur micro-contrôleur. Un nombre important de choses étudiées à l'IUT (comme le prototypage par wrapping, la programmation en langage C ou l'anglais) ont pu être mises en pratique durant ce stage et, globalement, j'en tire un bilan très positif.

Annexe A

Outils de mise en page et de dessin

A.0.1 Outils de mise en page

La mise en page de ce document a été faites avec LaTeX 2_ε. Il s'agit d'un *puissant* outil de mise en page très utilisé dans le milieu de la recherche tant pour la qualité des documents produits que pour les innombrables fonctionnalités qu'il procure. . .. Par exemple, la numérotation de titres, tableaux et autres figures est réalisé automatiquement, de même que la création d'une table des matières ou d'une liste des figures.

Grâce à LaTeX 2_ε, un simple éditeur de texte suffit pour réaliser des documents au rendu « professionnel ». Cependant, il existe des logiciels simplifiant l'écriture de documents en LaTeX 2_ε ; TeXnicCenter en est un exemple. Ce dernier a été utilisé pour la rédaction de ce document.

A.0.2 Outils de dessin

Tous les dessins, schémas et copies d'écran de ce document ont été réalisés avec Open Office Draw 1.1.4. Il s'agit d'un logiciel[1] de dessin vectoriel.

[1] libre : sous licence GPL

Annexe B

Code source de l'application simple

```
#include <C167CS.H>
#include "gpio.h"

#define MAXOF(X) ((1 << (sizeof(X) << 3))-1)

unsigned int valmoteur=2000;
char moteur_onoff=0xFF;
unsigned long nbtours=0;

/*Fonction appellée lors d'un changement d'état du switch 1 */
void IntCC0(void) interrupt 0x10 {
        SetLED(1,!SW1);
}
/*Fonction appellée lors d'un changement d'état du switch 2 */
void IntCC1(void) interrupt 0x11 {
        SetLED(2,!SW2);
}
/*Fonction appellée lors d'un changement d'état du switch 3 */
void IntCC2(void) interrupt 0x12 {
        SetLED(3,!SW3);
}
/*Fonction appellée lors d'un changement d'état du switch 4 */
void IntCC3(void) interrupt 0x13 {
        SetLED(4,!SW4);
}
/*Fonction appellée lors d'un changement d'état du switch 5*/
void IntCC4(void) interrupt 0x14 {
        SetLED(5,!SW5);
}
/*Fonction appellée lors d'un changement d'état du switch 6 */
void IntCC5(void) interrupt 0x15 {
        SetLED(6,!SW6);
}
/*Fonction appellée lors d'un changement d'état du switch 7 */
void IntCC6(void) interrupt 0x16 {
        SetLED(7,!SW7);
}
/*Fonction appellée lors d'un changement d'état du switch 8 */
void IntCC7(void) interrupt 0x17 {
        SetLED(8,!SW8);
}
```

FIG. B.1 – Code source de l'application très simple (1/2)

```
/*Fonction appelée lors d'un appui sur key 1 */
void IntCC8(void) interrupt 0x18 {
        unsigned char vpot1;
        vpot1 = GetVPot(1);
        BarGraphLED(vpot1,MAXOF(unsigned char));
}
/*Fonction appelée lors d'un appui sur key 2 */
void IntCC9(void) interrupt 0x19 {
        unsigned char vpot2;
        vpot2 = GetVPot(2);
        BarGraphLED(vpot2,MAXOF(unsigned char));
}
/*Fonction appelée lors d'un appui sur key 3 */
void IntCC10(void) interrupt 0x1A {
        unsigned char vpot3;
        vpot3 = GetVPot(3);
        BarGraphLED(vpot3,MAXOF(unsigned char));
}
/*Fonction appelée lors d'un appui sur key 4 */
void IntCC11(void) interrupt 0x1B {
        if (valmoteur <= 19900){
                valmoteur += 300;
                CMDMoteur(ON,1,valmoteur);
        }
}
/* Fonction appelée lors d'un appui sur key 5 */
void IntCC12(void) interrupt 0x1C {
        if (valmoteur >= 100){
                valmoteur -= 300;
                CMDMoteur(ON,1,valmoteur);
        }
}
/* Fonction appelée lors d'un appui sur key 6 */
void IntCC13(void) interrupt 0x1D {
        CMDMoteur(moteur_onoff,1,valmoteur);
        moteur_onoff = ~moteur_onoff;
}
/* Fonction à chaque tour du moteur */
void IntCC14(void) interrupt 0x1E {
        nbtours++;
}


void main(){
        InitGPIO();
        IEN = 1;
        while(1);
}
```

FIG. B.2 – Code source de l'application très simple (2/2)

Liste des tableaux

Table des figures