UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et génie informatique

ENVIRONNEMENT MATÉRIEL ET LOGICIEL POUR LE DÉVELOPPEMENT DE SYSTÈMES INTELLIGENTS

Mémoire de maîtrise ès sciences appliquées
Spécialité: génie informatique

Jean-Marc Ranger

Sherbrooke (Québec), CANADA

Mai 2000

RÉSUMÉ

La recherche sur les systèmes intelligents tend à sous-estimer l'importance de la plate-forme matérielle et logicielle sur laquelle est fait le développement. Actuellement, la conception d'un nouveau robot se fait pratiquement toujours à partir de zéro, car il est plus simple de recréer les fonctionnalités des anciens robots que de récupérer les modules qui conviendraient. De la même manière, il arrive qu'un nouveau robot doive être créé simplement parce que l'ancien n'a pas la capacité de supporter les nouvelles composantes, sa capacité d'extensibilité étant trop limitée.

Le présent projet cherche à régler de tels problèmes de réutilisabilité, de reconfigurabilité et d'extensibilité, de même qu'à faciliter le développement d'applications en robotique et systèmes intelligents, par la disponibilité d'outils adaptés. La solution proposée et présentée ici est un environnement de développement matériel et logiciel multiprocesseur modulaire, baptisé EME (Environnement Multiprocesseur Embarqué).

L'environnement EME est validé au moyen d'une application simple en robotique mobile, soit un robot équipé de deux caméras. Une des caméras est pointée vers le sol et cherche un tracé à suivre; la seconde est à la recherche d'un signal lumineux par lequel le robot reçoit des directives. Les résultats montrent qu'une telle application aurait été difficile à réaliser à un coût raisonnable sans un tel environnement.

REMERCIEMENTS

Je désire tout d'abord remercier mon directeur de recherche, M. François Michaud, pour sa collaboration à la réalisation de EME. Sa compétence, sa disponibilité, sa rapidité et son souci du détail ont été des aides importantes tout au long de ce projet. Je tiens aussi à remercier les membres du jury pour leur évaluation et leurs commentaires sur ce mémoire.

Je souhaite également remercier toutes les personnes qui m'ont apporté une précieuse collaboration: M. Bruno Paillard, pour avoir suscité en moi l'intérêt pour les études supérieures, M. Philippe Mabilleau, pour ses conseils sur le Kit331, M. Serge Caron, pour son soutien technique constant et M. Louis Drolet, pour ses contributions à ce projet.

Je désire aussi souligner la contribution financière essentielle du Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG).

Finalement, je ne peux passer sous silence le support de mes proches, parents et amis, durant ces années. Je les en remercie tout spécialement, ce fut et c'est encore très apprécié.

TABLE DES MATIÈRES

1. INTRODUCTION 1

2. PRODUITS EXISTANTS POUR LA CONCEPTION DE SYSTÈMES INTELLIGENTS 3

2.1 PROCESSEURS 3
2.2 INTERFACES DE COMMUNICATION MULTIPROCESSEUR 5
2.3 CARTES, ARCHITECTURES ET IMPLANTATIONS 7
2.4 ENVIRONNEMENTS DE DÉVELOPPEMENT ET NOYAUX TEMPS RÉEL 9
2.5 SOMMAIRE 14

3. ENVIRONNEMENT EME 16

3.1 CHOIX DE CONCEPTION ET MATÉRIEL DE BASE 17
3.2 MATÉRIEL DU BUS DE COMMUNICATION EME 19
3.3 NOYAU TEMPS RÉEL ILNI 22
3.3.1 Gestionnaire des tâches 24
3.3.2 Services auxiliaires 27
3.4 INTERFACE LOGICIELLE DE COMMUNICATION INTERPROCESSEUR 28
3.4.1 Communication système – niveau 1 30
3.4.2 Format du paquet de communication 34
3.4.3 Communication utilisateur – niveau 2 35
3.4.4 Services de nom 36
3.5 SOMMAIRE 37

4. VALIDATION DE EME AVEC UN ROBOT MOBILE 39

4.1 CONFIGURATION MATÉRIELLE DU ROBOT 40
4.2 DISPOSITIF EXTERNE DE SIGNALISATION LUMINEUSE 42
4.3 TRAITEMENT DES IMAGES 44
4.4 CONTRÔLE DU ROBOT 49
4.5 SOMMAIRE 52

5. RÉSULTATS 53

5.1 PERFORMANCES DU NOYAU ILNI 53
5.2 PERFORMANCES DU BUS DE COMMUNICATION INTERPROCESSEUR 58
5.3 PERFORMANCES DU ROBOT *BIGBROTHER* 62
5.4 SOMMAIRE 66

CONCLUSION 67

ANNEXE 1: SCHÉMA ÉLECTRIQUE DE L'INTERFACE DE COMMUNICATION 71

ANNEXE 2: SCHÉMA ÉLECTRIQUE DE L'INTERFACE QUICKCAM 72

BIBLIOGRAPHIE 73

LISTE DES FIGURES

Figure 3.1: Schéma-bloc d'un système EME à deux microcontrôleurs 17

Figure 3.2: Schéma-bloc des communications dans EME, avec l'application de test 21

Figure 3.3: Modèle en couches 29

Figure 3.4: Procédure d'initialisation de l'interface de communication 31

Figure 3.5: Détection des cartes en mode d'opération 32

Figure 4.1: Robot *BigBrother* 40

Figure 4.2: *BigBrother* et l'émetteur lumineux externe 41

Figure 4.3: Émetteur lumineux 42

Figure 4.4: Effets des réglages de la caméra 1 44

Figure 4.5: Commandes du dispositif de signalisation lumineuse 44

Figure 4.6: Simplification de l'image en vecteurs 46

Figure 4.7: Effets des réglages de la caméra 2 48

Figure 4.8: Schéma d'arbitrage des moteurs 51

Figure 5.1: Utilisation du processeur en fonction du débit SPI 58

Figure 5.2: Filage du bus EME sur *BigBrother* 61

Figure 5.3: Piste d'expérimentation 64

LISTE DES TABLEAUX

Tableau 3.1: Principaux services du noyau ILNI offerts à l'application utilisateur 23

Tableau 4.1: Résultats de l'analyse de recherche de ligne... 47

Tableau 4.2: Résultats de la recherche de messages lumineux... 49

Tableau 5.1: Poids processeur des noyaux.. 55

Tableau 5.2: Taille mémoire des noyaux.. 57

Tableau 5.3: Consommation électrique.. 62

LEXIQUE

Extensibilité

Facilité avec laquelle une tierce personne peut ajouter des nouvelles capacités au matériel et au logiciel [24].

Interopérabilité

Facilité avec laquelle plusieurs machines peuvent se partager de l'information et l'utiliser de manière transparente [24].

Machine virtuelle

Une machine virtuelle peut être définie comme un pseudo-processeur ou processeur "imaginaire" [1], avec son propre jeu d'instructions qui tourne en émulation sur un processeur hôte bien réel. La machine Java (Sun) et le p-code (Newton Labs) en sont des exemples.

Machine pile

Définie comme un processeur qui ne contient aucun registre. Les données de travail sont plutôt stockées dans la pile. Ainsi, une tâche peut être définie uniquement par deux paramètres, soit le compteur ordinal [2] et la pile. Un exemple typique est la calculatrice HP48.

---

[1] Pas si imaginaire que cela puisque dans le cas de Java, des processeurs basés sur ce jeu d'instruction ont bel et bien été conçus mais ce, un certain temps après l'introduction de la machine virtuelle Java.

[2] Plus souvent identifié par son nom anglais, "*program counter*" ou "*PC*".

Multiprocesseur symétrique

Dans un système multiprocesseur symétrique, les processeurs doivent être identiques. Une tâche peut donc s'exécuter sur n'importe quel processeur - en fait, elle ne sait pas sur quel processeur elle s'exécute.

Multiprocesseur asymétrique

Un système multiprocesseur asymétrique autorise l'emploi de processeurs différents dans un même système, chacun adapté à un besoin spécifique. Il peut donc y avoir par exemple un DSP pour certaines tâches de traitement d'image et un processeur "conventionnel" pour des tâches plus simples (gestion de moteur, de capteurs simples). Par le fait même, un système asymétrique impose que chaque tâche soit assignée à un processeur spécifique.

Système (architecture) ouvert

Les caractéristiques d'un contrôleur robotisé ouvert sont qu'il doit pouvoir s'exécuter sur une variété de plates-formes matérielles disponibles dans le commerce, s'exécuter avec un système d'exploitation lui aussi disponible commercialement et être basé sur du logiciel "ouvert", qui permet l'utilisation de logiciels provenant d'autres sources [27].

Portabilité

Facilité avec laquelle une application peut être portée d'un système à un autre, en conservant ses capacités [24].

Reconfigurabilité

Facilité avec laquelle la performance d'un système existant peut être ajustée (vers le haut ou le bas) en fonction des besoins de l'application [24].

1. INTRODUCTION

La problématique entourant la notion de système intelligent implique la conception d'un système ayant des capacités particulières de perception, d'action, de traitement et de prise de décision lui permettant de s'adapter avec une certaine autonomie à son milieu d'opération afin d'y remplir efficacement son rôle. Un tel système prend habituellement la forme d'un système embarqué, composé de microprocesseurs, de capteurs, d'actionneurs, d'outils de développement logiciel (noyau temps réel, fonction de gestion de haut et de bas niveaux) et d'une architecture logicielle pour la mise en œuvre de la politique de prise de décision et de contrôle.

À la conception proprement dite du système intelligent s'ajoutent les problèmes de la réutilisabilité et de la portabilité du code, ainsi que la reconfigurabilité et l'extensibilité du système. Habituellement, chaque robot est un système unique, avec une combinaison de matériel et de logiciel qui lui est propre. Dans bien des cas, lors d'un changement de matériel, comme par exemple lors du passage de l'expérimentation à la mise en production, il est plus simple de recoder entièrement une application que de tenter d'en porter le code. On commence aussi à voir apparaître des robots multiprocesseurs mais conçus comme une seule entité et non pas par modules. Il est donc difficile, voir impossible, d'isoler et de récupérer les composantes une par une, tant en logiciel qu'en matériel, ce qui fait que dans ces cas, au lieu de régler le problème, l'aspect multiprocesseur le complique davantage en créant un système encore plus unique. Toutefois, la capacité multiprocesseur est de plus en plus recherchée, car elle permet d'accroître la puissance de calcul disponible, la capacité d'interfaçage avec des périphériques, l'extensibilité du système ou sa sécurité par de la redondance matérielle.

Il est donc souhaitable d'orienter la conception du système intelligent non seulement en direction de la fonctionnalité à atteindre, mais aussi de manière à ce que ce travail puisse être réutilisé. Ce problème de standardisation est très similaire à celui qui a donné naissance au concept de système d'exploitation et peut être résolu de manière similaire. Il existe donc actuellement un besoin et une grande utilité pour un environnement de développement à microcontrôleur conçu pour les problématiques propres aux systèmes intelligents, qui fournirait des outils de base pour répondre aux besoins en services logiciels (noyau temps réel multitâche et plus) de même qu'en matériel.

À cette fin, l'objectif du présent projet est de concevoir un environnement de développement temps réel multiprocesseur. Il est baptisé EME, pour Environnement Multiprocesseur Embarqué. L'aspect multiprocesseur est au cœur de la solution car il permet un développement modulaire et facilite l'extensibilité du système. Pour le mémoire, cet environnement est mis en œuvre avec des microcontrôleurs 68331 de Motorola, et il est validé dans une application en robotique mobile. Cette dernière démontre que l'environnement est fonctionnel, qu'il permet effectivement la conception rapide d'application complexe ainsi qu'une modularité et une extensibilité appréciables.

Ce document est organisé comme suit. D'abord, la section 2 présente les concepts et produits existants sur le marché, ainsi que les résultats de recherches antérieures dans ce domaine. La section 3 décrit les considérations de conception de l'environnement développé, ainsi que ses caractéristiques. La section 4 présente l'environnement de validation, soit un système intelligent construit au moyen de l'environnement. Les résultats et mesures de performances tant de l'environnement que de l'application de test sont présentés en section 5. Finalement, la conclusion analyse l'ensemble du projet et présente des perspectives futures.

2. PRODUITS EXISTANTS POUR LA CONCEPTION DE SYSTÈMES INTELLIGENTS

Tant pour le matériel que pour le logiciel, il existe un très grand nombre de concepts et de produits déjà disponibles sur le marché ou en cours de développement, pouvant servir à la conception de systèmes intelligents. Cette section présente les principaux, ainsi que quelques-unes de leurs caractéristiques. Lorsque disponibles, des références autres que bibliographiques sont fournies en bas de page.

2.1 Processeurs

Le premier item à choisir lors de la conception d'un système intelligent embarqué est le processeur. Voici les plus utilisés:

- *Motorola famille HC11* [3]. Ces microcontrôleurs sont simples et réputés insensibles aux interférences, mais ils sont limités par leur simplicité. Leur faible puissance de calcul, le petit nombre de registres et l'espace d'adressage (64 ko pour la plupart des versions) les limitent à des applications simples. Il existe un grand nombre de versions de HC11, avec des mémoires internes (RAM et ROM) de différentes tailles. Ils sont néanmoins utilisés comme circuits d'interface pour capteurs ou actionneurs sur certains robots (Pioneer I [4]), ou encore comme contrôleur principal dans des robots simples (RugWarrior [5]).
- *Motorola famille CPU32* [6]. Comportant notamment les microcontrôleurs 68331 et 68332, cette gamme dispose de plus de puissance de calcul que les produits de la famille HC11, ce qui

[3] http://www.mcu.motsps.com/hc11/index.html
[4] http://www.activmedia.com/robots/PioneerSpec.html
[5] http://www.tiac.net/users/akpeters/Rug-warrior.html
[6] http://www.mcu.motsps.com/33x/index.html

les rend aptes à être utilisés dans d'autres champs d'applications. L'espace d'adressage est aussi plus considérable (16 Mo). Le 68332 diffère du 68331 sur deux aspects: il dispose d'une mémoire interne de 2 ko et ses services de temporisations sont différents. Ils sont basés sur un *Time Processor Unit* (TPU), contrairement au 68331 qui utilise un module *General Purpose Timer* (GPT). Leur jeu d'instructions est une version étendue de celui du processeur 68000. Ces microcontrôleurs sont utilisés dans plusieurs robots, comme par exemple les Koala et Khepera [7].

- *Motorola familles MPC500/800* [8], *M-CORE* [9] *et Coldfire* [10]. Il s'agit ici de nouvelles familles de microcontrôleurs de Motorola. La famille MPC500/800, basée sur l'architecture PowerPC, finira par remplacer l'architecture CPU32 de la même manière que les processeurs PowerPC ont suppléé aux processeurs 68000. Il est toutefois à prévoir que les performances ne s'accroîtront pas comme dans le cas des processeurs, car même si les fréquences d'opération des microprocesseurs PowerPC sont actuellement dans les centaines de mégahertz, celles des microcontrôleurs PowerPC sont encore dans les dizaines de mégahertz. Les trois familles sont des processeurs RISC, visant des marchés différents. Motorola destine les MPC500/800 à des applications proches de l'informatique (de par leurs jeux d'instructions similaires), les M-CORE au secteur des transports (par exemple l'électronique automobile) et Coldfire aux applications "domestiques" (*consumer electronics*).

- *Intel famille 8086* [11]. Standard *de facto* dans le monde informatique, les produits Intel sont moins reconnus dans le domaine des systèmes embarqués. Bien qu'ils offrent un grand nombre de familles de microcontrôleurs (série 96, série 51, StrongARM), leur produit le plus

---

[7] http://lamiwww.epfl.ch/Khepera/produit.html
[8] http://www.mcu.motsps.com/mpc500/index.html
[9] http://www.mot.com/SPS/MCORE/
[10] http://www.mot.com/SPS/HPESD/prod/coldfire/
[11] http://developer.intel.com/design/embcontrol/

populaire même dans les systèmes embarqués demeure l'architecture 8086. Ces systèmes embarqués ne sont donc pas basés sur des microcontrôleurs, mais sont plutôt des ordinateurs complets de type "PC". Ces systèmes sont du même coup plus gros (en nombre de composantes) et plus coûteux.

- *DSP – microcontrôleur*. Concept assez récent, les grands fabriquants de processeurs de traitement de signal offrent désormais des produits intégrant les caractéristiques typiques des microcontrôleurs. Par exemple, dans le cas de la série x240 [12] de Texas Instruments, il s'agit de versions modifiées du célèbre TMS320C25. L'utilité de ce nouveau type de microcontrôleur est évidente pour les applications embarquées qui ont également à accomplir des tâches de traitement de signal, l'intégration d'un DSP à un système embarqué étant alors grandement facilitée.

2.2 Interfaces de communication multiprocesseur

Les interfaces de communication peuvent être divisées en deux grandes classes. La première, généralement associée à un système multiprocesseur symétrique, regroupe toutes les techniques visant à créer un espace d'adressage auquel plusieurs processeurs peuvent accéder. Dans ces techniques, l'emplacement de la donnée commune peut être identifié de manière unique par son adresse mémoire, adresse reconnue (avec traduction au besoin) par l'ensemble des processeurs du système. Il existe plusieurs de ces techniques, directes ou dérivées:

- *Mémoire à accès multiple*. Dans ce système, utilisé dans les premières versions de la "tête" du robot humanoïde COG du MIT [13], une même puce mémoire est directement accessible par

---

[12] http://www.ti.com/sc/docs/psheets/abstract/datasht/sprs042b.htm
[13] http://www.ai.mit.edu/projects/cog/

plusieurs processeurs du système. Il est donc possible d'y organiser une structure permettant l'échange d'informations communes.

• *Bus mémoire*. Cette catégorie comprend les VMEbus, NuBus, PCI, S-bus, PC-104 et plusieurs autres. Il s'agit d'interfaces standardisées, à moyen ou haut débit, visant à l'origine le raccordement de composantes d'ordinateurs domestiques ou industriels. Font partie de ces standards les formes et dimensions des connecteurs de raccordement, ce qui dans certain cas fixe au départ une taille minimum à l'application finale. Ces standards sont généralement assez complexes, ce qui implique parfois la conception d'un circuit dédié d'interface au bus. Aussi, ces standards sont souvent fortement couplés à une famille de processeurs précis, ce qui peut ajouter d'autres contraintes.

La seconde catégorie d'interfaces de communication interprocesseur est la grande famille des ports séries multipoints. Elle est principalement utilisée pour les systèmes asymétriques. Dans cette classe, pour aller chercher une donnée, il y a deux informations à connaître:

- l'identificateur de la carte qui contient la donnée, et
- le protocole pour la requérir, car la donnée n'est que rarement accessible directement.

Encore ici, il existe plusieurs modèles d'interface:

• *Standards "microprocesseurs"*. Le monde des ordinateurs (PC et industriel) comprend un certain nombre de standards de ce type. Le plus connu est certainement Ethernet (10 Mbit à 1 Gbit, moyenne portée). Mais de nouveaux standards sont en train de s'implanter, comme USB (12 Mbit, faible portée) ou IEEE-1394/FireWire (400 Mbit, faible portée).

• *Standards "microcontrôleurs"*. Les applications industrielles ont aussi leurs standards: SPI [14] (4 Mbit, Motorola), I2C [15] (400 kbit, Philips) ou encore CAN [16] (1 Mbit, Bosch). Chacun de ces

---

[14] http://ebus.mot-sps.com/mcu/documentation/pdf/qsmrm.pdf
[15] http://www.semiconductors.com/handbook/handbook_38.html
[16] http://www.bosch.de/de_e/productworld/k/products/prod/can/content/homepage.html

standards a ses supporters et ses opposants, mais les principales différences semblent dépendre davantage de l'âge et du domaine d'application original du standard plus que d'un réel avantage technique par rapport à un compétiteur.

2.3 Cartes, architectures et implantations

Il existe toute une variété de cartes basées sur ces différents microcontrôleurs. Chacune offre ses petites différences, comme le Vesta Board [17] (68332) qui offre des outils de développement en Forth, le L-Board [18] (68332) qui se spécialise en Lisp, la carte de Daedalus [19] (68332) conçue pour le robot du même nom, le Kit331 [20] (68331) développé pour des fins pédagogiques, ou encore ITImicro/11K [21] (68HC11) qui dispose d'un port Ethernet. En voici d'autres qui méritent qu'on s'y attarde de par leurs similitudes avec le présent projet:

- *RoboCube* [22]. Ce projet [4], basé à l'origine sur une carte Vesta Board, offre un ensemble d'outils pour le développement de robots, plus précisément les robots destinés à participer aux compétitions *RoboCup* [23]. Il s'agit d'une carte à microcontrôleur conçue de manière à avoir assez de services et de points d'interface pour combler les besoins standards d'un robot: communication série, radio et infrarouge, contrôle de moteur, convertisseurs analogiques-numériques et numériques-analogiques, entrées et sorties numériques, temporisateurs et services logiciels de base. Ce produit est très compact: 86 mm x 77 mm x 40 mm, pour trois circuits imprimés empilés. La conception semble toutefois assez fortement couplée au microcontrôleur choisi (68332). Aucune capacité multiprocesseur n'apparaît non plus dans le produit actuel.

---

[17] http://www.newtonlabs.com/arc/vesta.html
[18] http://www.isr.com
[19] http://www.gel.usherb.ca/caron/daedalus/daedalus.html
[20] http://www.gel.usherb.ca/mab/gei435/kit331.htm
[21] http://www.funet.fi/~kate/hc11.html
[22] http://arti.vub.ac.be/~thomas/robocube/overview.html
[23] http://www.robocup.v.kinotrope.co.jp/

• *Kameleon* [24]. Fabriquée par la compagnie K-Team depuis 1999, cette carte [14] est à notre connaissance la première à être conçue dès le départ comme bloc de base pour un système multiprocesseur. Elle offre d'ailleurs deux interfaces de communication entre les différents processeurs du système. La première (KNet) fonctionne à basse vitesse et semble principalement conçue pour y raccorder des capteurs intelligents, sur une interface basée sur un port SPI. Quant à la seconde (MMA/Mubus), l'interface est beaucoup plus rapide mais elle est limitée à cinq processeurs, quatre esclaves et un maître dédié. Le fonctionnement est basé sur une interconnexion des bus d'adresses et de données des microcontrôleurs, ce qui rend incertain la compatibilité avec des processeurs de modèles différents.

• *Sony MUTANT* [25]. La compagnie Sony travaille depuis plusieurs années sur un robot "jouet". Basé sur des processeurs MIPS R4000, il est utilisable tant comme animal virtuel (*robot-pet*) que comme robot pour les compétitions RoboCup. Avec ce projet, ils ont développé une architecture *plug and play* matérielle et logicielle du nom de OPENR [12] spécialement adaptée à la robotique. Il serait ainsi possible pour un robot de reconnaître directement les composantes qui sont raccordées au "corps" (par exemple des pattes ou des roues) et en faire usage sans reprogrammation. Cela a aussi comme avantage que les composantes développées selon ce standard sont utilisables sur tous les robots qui y sont conformes, contrairement à ce qui se passe aujourd'hui où le développement est fait avec un seul robot "cible" en tête.

• *Genghis, Hannibal, Attila* [26]. Ces trois robots [1, 11], tous trois en forme d'insecte à six pattes, comportent chacun entre huit et douze microcontrôleurs communiquant en permanence au moyen d'un bus série I2C. Le modèle proposé par les concepteurs est que chaque robot est

---

[24] http://www.k-team.com/boards/kameleon/index.html
[25] http://www.sony.co.jp/soj/CorporateInfo/SonyFun/topic/robot.html (malheureusement en japonais)
[26] http://www.ai.mit.edu/projects/genghis/
http://www.ai.mit.edu/projects/hannibal/hannibal.html

constitué de plusieurs "sous-robots" (patte, tête, corps) qui, puisqu'ils sont physiquement reliés, doivent collaborer pour effectuer une tâche, comme le déplacement de l'ensemble du robot. Chacun des sous-robots a un certain nombre de capteurs et d'actionneurs qui lui sont propres et il peut communiquer avec les autres sous-robots pour les aviser de ses intentions, par exemple pour empêcher que les six pattes se soulèvent en même temps. En pratique, tout le traitement se fait sur le processeur du "corps" du robot. Les processeurs des pattes ne font que lui relayer les informations de leurs capteurs et recevoir de lui les instructions requises pour les gestes à poser avec les actionneurs locaux.

2.4 Environnements de développement et noyaux temps réel

De la même façon que pour le matériel, il existe déjà un grand nombre d'environnements de développement et de noyaux temps réels [27] prêts à être utilisés dans des systèmes embarqués. En voici plusieurs, remarquables soit parce que très utilisés en industrie ou en recherche, soit parce qu'ils illustrent des concepts intéressants. Ils sont présentés avec leurs avantages et leurs inconvénients lorsqu'utilisés pour le développement de systèmes intelligents.

- *Environnements professionnels.* Les environnements professionnels les plus connus sont pSOS [28] (Integrated Systems, Inc), VxWorks [29] (Wind River Systems), RTXC [30] et RTEK [31] (Motorola, Inc.). Il s'agit d'environnements de développement puissants, utilisés par de grandes entreprises comme Nortel. Ils ont certainement de grandes qualités, mais leur prix (plus de US$ 10 000 par poste de développement) les rendent totalement inaccessibles pour des usages en recherche universitaire, tant pour la phase de développement que pour une

---

[27] http://www.cis.ohio-state.edu/hypertext/faq/usenet/realtime-computing/list/faq.html
http://www.realtime-info.be/encyc/market/rtos/rtos.htm
[28] http://www.isi.com/Products/pSOS/
[29] http://www.wrs.com/products/html/vxwks52.html
[30] http://www.spectrumdigital.com/rtxc.htm
[31] http://www.mcu.motsps.com/lit/sel_guide/rtek.htm

éventuelle mise en production.

• *UNIX temps réel*. Le principal représentant de cette catégorie est QNX [32] (QNX Software Systems Ltd.) [25]. Tout en demeurant dans la classe des environnements professionnels, le prix de ce noyau (de l'ordre de US$ 2000) rend ce produit beaucoup plus intéressant que les précédents. Ses principales caractéristiques sont les suivantes:

- noyau très petit (quelques kilo-octets), avec un minimum de services (changement de tâche, passage de messages, interruptions matérielles et quelques autres services essentiels), les services supplémentaires (communication, systèmes de fichiers, interface graphique, etc.) sont implantés à l'extérieur du noyau, au moyen de tâches systèmes;
- certifié conforme au standard de programmation temps réel Posix [33];
- exclusif à l'architecture x86.

Il existe toute une série d'autres systèmes Unix capables de fournir des services temps réel. L'un d'entre eux est RTLinux [34]. Contrairement à QNX, il s'agit d'une extension à un noyau existant et non pas d'un noyau reconçu à partir de zéro, ce qui en fait un système plein de compromis. Comme la plupart des noyaux de cette catégorie à l'exception de QNX, RTLinux manque tant de développeurs que d'utilisateurs. Le fait qu'il soit gratuit est toutefois un avantage. Il peut même être utilisé en configuration multiprocesseur, tant sur une machine comportant plusieurs processeurs que par une combinaison de plusieurs ordinateurs via un réseau formant un système. Un tel système est dit de classe Beowulf [35].

• *Java [36] (Sun Microsystems, Inc.)* [17]. Des expériences ont déjà été tentées au Département

---

[32] http://www.qnx.com/
[33] http://www.pasc.org/
[34] http://luz.cs.nmt.edu/~rtlinux/
[35] http://beowulf.gsfc.nasa.gov/, ou encore http://www.beowulf.org
http://www.linuxdoc.org/HOWTO/Beowulf-HOWTO.html
[36] http://java.sun.com
Voir aussi les travaux de Kelvin Nilsen à http://kickapoo.catd.iastate.edu/java.html

de génie électrique et de génie informatique de l'Université de Sherbrooke pour utiliser Java comme noyau temps réel sur des systèmes à base de microcontrôleur. Les résultats ont été mitigés. La majorité de ceux qui ont participé à ces projets s'entendent pour dire que même si le projet avait été mené à terme, les performances auraient été décevantes, à moins de retravailler sérieusement l'ensemble dans une optique d'optimisation des performances. Le noyau demeurerait très gros (quelques centaines de kilo-octets), ce qui encore une fois n'est pas très approprié pour les applications visées. Le document de Sun [17] le confirme: "*One could build a complete system with a total of 4 MB of ROM and 4 MB of RAM*". Il est clair toutefois que certains concepts sont à retenir. Java est un langage beaucoup plus sécuritaire que le C, dans le sens qu'il est plus difficile, par exemple, d'utiliser un pointeur en dehors de sa zone de validité. Mais cela se fait au prix d'une grande utilisation de ressources (processeur et mémoire).

- *RTEMS* [37]. Développé pour l'armée américaine, ce noyau est distribué suivant une licence inspirée de la GPL [38] de GNU. Il est disponible pour divers microprocesseurs, dont le 68020 de Motorola, en langage C et en Ada. Il s'agit ici encore d'un noyau très complet et donc assez lourd.

- *Chimera* [39] *et Echidna* [40]. Chimera (Université Carnegie Mellon, [30, 31]) est un système d'exploitation multiprocesseur symétrique, destiné principalement aux applications en robotique. Les documents disponibles indiquent qu'il est rapide et flexible. Il est basé sur un bus VME. Son développement a toutefois été abandonné suite au départ de son principal concepteur, David B. Stewart. Ce dernier, maintenant à l'Université du Maryland, travaille sur Echidna, un noyau temps réel uniprocesseur destiné aux systèmes à microcontrôleurs. Il se distingue par

---

[37] http://www.rtems.army.mil/, ou encore http://lancelot.gcs.redstone.army.mil/rtems.html
[38] http://www.gnu.org/copyleft/gpl.html
[39] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/chimera/www/home.html
[40] http://www.ece.umd.edu/serts/research/echidna/index.shtml

l'intégration des derniers concepts développés dans le domaine des systèmes d'exploitation, notamment les applications à multiples fils d'exécution parallèles (*multi-treads*).

- *Apertos* [41] *(Sony Computer Science Laboratories, Inc.)*. Ce système d'exploitation est celui que Sony utilise avec son robot MUTANT (voir section 2.3). Il est développé entièrement selon l'approche orientée objet. Il est donc possible de remplacer des composantes du système d'exploitation de la même manière que les composantes matérielles du robot. Il dispose aussi de capacités de chargement de code par objet, en plus du chargement traditionnel (par application entière).

- *Interactive C et ARCC (Newton Research Labs)* [42]. Disponible uniquement pour le 68HC11, Interactive C entre dans la catégorie des noyaux basés sur une machine virtuelle. Il bénéficie des avantages comme des inconvénients de ce type de noyau, soit:

- langage de programmation propriétaire, généralement plus fiable et plus protégé que les langages habituels (Interactive C utilise une version réduite du C, où par exemple les opérations sur les pointeurs sont très limitées);
- portabilité directe si l'émulateur est disponible sur la plate-forme cible, sinon portabilité réduite à cause du langage propriétaire;
- légère dégradation de performance à cause de l'émulateur;
- la taille du code binaire de la machine virtuelle est similaire à du code en C, mais il faut ajouter le code de l'émulateur.

Interactive C offre aussi deux services rares et très utiles pour la mise au point de programmes, soit:

- l'appel de fonctions en ligne de commande;
- le chargement manuel (sur commande de l'usager) de blocs de code.

---

[41] http://www.csl.sony.co.jp/project/Apertos/index.html
[42] http://www.newtonlabs.com/

Le premier permet d'appeler les différentes fonctions disponibles en mémoire comme s'il s'agissait de commandes ou de programmes différents, via un terminal du genre "ligne de commande" et ce, pendant l'exécution du programme principal. Le second service élargit encore cette capacité en offrant la possibilité de charger et décharger des bouts de codes (modules ou fonctions), toujours pendant que le noyau tourne. Ces fonctionnalités ne sont pas disponibles avec une méthode usuelle de développement croisé, qui requiert que le programme soit connu dans sa totalité au moment de l'édition des liens.

Pour sa part, l'environnement ARCC est une version révisée de Interactive C pour le 68332 de Motorola. La machine virtuelle a disparu dans la révision, mais les services de déverminage sont toujours présents.

Ces produits sont cependant très fortement couplés à leur matériel. Ils ont aussi été développés dans une optique uniprocesseur. Comme tous les noyaux commerciaux, ces produits ne sont pas disponibles en format "source", mais seulement sous forme d'exécutables déjà compilés [43].

- *L (IS Robotics Inc)* [7]. Il s'agit ici d'un noyau temps réel en Lisp, ce qui au point de départ ne le met pas à la portée de tout le monde. Par contre, il offre des services de déverminage très semblables à ceux de Interactive C. Il a été utilisé dans plusieurs robots basés sur des processeurs CPU32, comme par exemple le Pioneer I. Il a aussi été utilisé sur le robot COG, à l'époque où ce dernier était composé de plusieurs processeurs 68332.

- *μCOS* [44] [15]. Ce noyau a de nombreuses caractéristiques intéressantes. D'abord, il est gratuit et distribué en format source. Il a déjà été porté sur différentes architectures (dont x86, CPU32, HC11). Il offre les services de base (changement de tâche, sémaphores, passage de

---

[43] Le code source de Interactive C est public pour les versions antérieures à 3.0. Le code de ARCC n'est pas public.

[44] http://www.ucos-ii.com/

messages, etc.) et est très compact (l'ensemble du source fait moins de 100 ko). Il a par contre des limites. Il est par exemple impossible de créer simultanément plus de 64 tâches (incluant les tâches systèmes) et il est aussi impossible de créer deux tâches de même priorité. Bien entendu, il n'offre aucun service de haut niveau (communication réseau, système de fichier, interface utilisateur). L'auteur a aussi rendu public une série d'outils logiciels courants, en particulier du code d'interface avec divers périphériques d'entrée/sortie [16].

- *Ayllu* [45]. Techniquement, Ayllu n'est pas un noyau temps réel. C'est une librairie C, conçue avant tout pour servir d'environnement de développement pour les robots de la compagnie ActivMedia. Il est donc capable d'exécuter des "*lightweight processes*" (des petites tâches s'exécutant en entier dans un court laps de temps) à fréquence régulière. À ce niveau, on pourrait le comparer à un système entier, composé de tâches groupées autour d'un noyau non-préemptif, pouvant être exécuté comme une seule tâche sur un système d'exploitation habituel. Il offre aussi des services de communication parfaitement transparents pour les systèmes à plusieurs robots. Il est ainsi possible de créer une application "haut niveau" composée de plusieurs robots et d'échanger des messages d'un sous-système (un robot) à l'autre comme si le tout se passait localement. Par contre, comme la plupart des systèmes non-préemptifs, Ayllu a de la difficulté lorsque le taux d'utilisation du processeur approche de 100%.

2.5 Sommaire

De tous ces produits, plusieurs caractéristiques importantes ressortent assez clairement. D'abord, il est visible que plusieurs robots sont désormais composés de plusieurs processeurs. Les motifs diffèrent (puissance de calcul requise pour COG, nombre d'interfaces vers des capteurs pour le Pioneer I, redondance pour Hannibal), mais la solution reste la même. Il est

[45] http://www.activrobots.com/SOFTWARE/ayllu.html

aussi clair qu'il n'existe pas encore de solution intégrée unique permettant de s'adapter à toutes ces situations. Par exemple, la carte Kameleon est assez fortement liée à son matériel et exige une carte maître, et l'architecture OPENR est encore en développement et s'annonce lourde à supporter.

3. ENVIRONNEMENT EME

La recherche en robotique mobile et systèmes intelligents a tendance à négliger les considérations concernant les aspects matériels et logiciels des systèmes pour se concentrer plutôt sur la partie "intelligence" comme telle. Il est toutefois impossible de développer les algorithmes souhaités si ces considérations ne sont pas prises en compte. Pour éviter de réutiliser des composantes développées dans un projet antérieur et qui ne s'adaptent qu'à moitié au nouveau projet, il semble pertinent de développer dès le départ un ensemble d'outils flexibles et réutilisables dans une variété de contextes [3].

C'est en fait ce que nous tentons de réaliser dans ce projet avec l'environnement baptisé EME, pour Environnement Multiprocesseur Embarqué. Comme spécifications, cet environnement doit d'abord être le plus modulaire possible afin de maximiser la réutilisabilité de ses composants. Il doit aussi être peu coûteux, de manière à ne pas nuire à une éventuelle mise en production d'un système intelligent développé au moyen de EME. Il s'agit donc de limiter les coûts à tous les niveaux, tant pour les pièces que pour les licences du compilateur et les autres redevances de propriétés intellectuelles. L'extensibilité et la reconfigurabilité font aussi partie des objectifs visés.

Les quatre principaux modules de EME sont les cartes à microcontrôleurs, le bus de communication matériel, le noyau temps réel nommé ILNI ainsi que l'interface logicielle de communication interprocesseur. La figure 3.1 présente ces modules ainsi que leurs interactions. Les ovales représentent les modules logiciels, et les rectangles, les composantes matérielles. La présente section décrit en détails les objectifs de conception, ainsi que les caractéristiques des quatre modules de l'environnement développé pour y répondre.

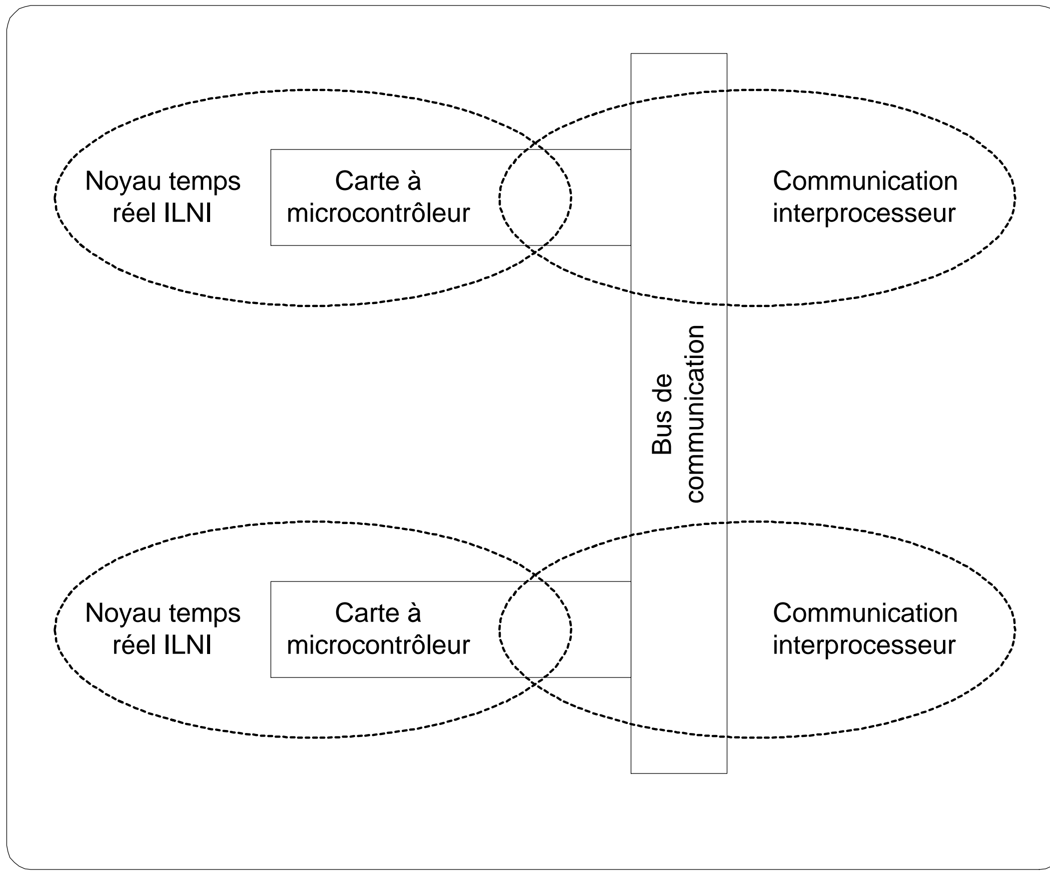


Figure 3.1: Schéma-bloc d'un système EME à deux microcontrôleurs

3.1 Choix de conception et matériel de base

Afin de rencontrer les spécifications mentionnées précédemment, un certain nombre de choix de conceptions ont été faits. Les voici, accompagnés de leurs justifications:

- *Multiprocesseur asymétrique*. Le principe fondamental de l'environnement EME est de développer des cartes propres à une application précise, en lui fournissant des moyens de s'interfacer avec d'autres, possiblement conçues initialement pour un environnement très différent. Choisir l'approche multiprocesseur symétrique aurait imposé un microprocesseur ou un microcontrôleur commun à toutes les applications basées sur EME, une contrainte que nous avons jugée inacceptable. Le système est donc de type multiprocesseur asymétrique.

- *Utilisation de microcontrôleurs.* Le développement de systèmes embarqués basés sur des microprocesseurs donne un certain avantage en terme de puissance de calcul, mais implique un coût important pour le développement ou l'intégration des interfaces avec les circuits périphériques requis pour l'application. Dans un système compact et de faible coût, les microcontrôleurs s'imposent puisque ces interfaces sont déjà intégrées. Cela n'empêche toutefois pas l'utilisation de processeurs dans des systèmes EME où le coût est moins critique que le besoin en puissance de calcul.

- *Communication par port série asynchrone SPI.* L'extensibilité et la reconfigurabilité du système sont principalement assurées par le fait qu'il est possible de lui ajouter des cartes et des processeurs selon le besoin. Cette capacité est fournie par un port d'interface matériel fonctionnant par adressage (un port pour plusieurs connexions). Le choix de l'approche multiprocesseur asymétrique, qui découple les processeurs au point d'en faire des entités autonomes, donne une latitude beaucoup plus grande quant au choix de l'interface de communication. Cette dernière peut donc devenir un échange de messages plutôt qu'un bus de données commun à tout le système. La bande passante requise diminue d'autant, mais doit tout de même permettre le passage de données de taille importante au besoin. Un débit cible de l'ordre de 500 kBaud a été fixé arbitrairement. Cette faible valeur fait en sorte que toutes les approches présentées à la section 2.2 sont envisageables. La solution la plus simple a été retenue, c'est-à-dire de se concentrer sur les ports séries multipoints déjà intégrés dans des microcontrôleurs. Cela a comme avantages l'assurance d'une compatibilité avec une large plage de contrôleurs, une interface physique simple avec peu de fils ainsi qu'un coût réduit car l'interface ne nécessite pas d'électronique en plus du microcontrôleur. La seule autre interface qui a été envisagée est Ethernet. Mais elle a rapidement été éliminée parce qu'elle nécessite environ 100ko de mémoire pour une implantation complète [28], ce qui est démesuré dans le

présent contexte. Des trois ports séries multipoints pour microcontrôleurs discutés à la section 2.2, le SPI de Motorola a été retenu pour les raisons suivantes:

- C'est celui qu'on retrouve le plus fréquemment (Motorola, Siemens, Texas Instruments). Le port I2C (Philips) devient de plus en plus fréquent mais pas encore autant que le SPI, et le port CAN (Motorola) est encore relativement rare.
- Tout comme pour I2C, il existe différents périphériques pouvant s'interfacer directement sur un port SPI, comme par exemple certaines mémoires.
- Bien que les débits diffèrent légèrement, tous sont suffisants pour la tâche.
- Une implantation I2C similaire a déjà été faite en robotique, sur les robots Genghis, Hannibal et Attila du MIT. Utiliser un autre port permet une comparaison des performances.

• *Contrôleur CPU32 et carte Kit331*. Pour le premier banc d'essai de EME, un contrôleur de puissance moyenne est approprié. Il doit contenir une interface compatible SPI intégrée. Le choix s'est porté sur la famille Motorola CPU32 et se justifie par son utilisation fréquente dans le domaine, son coût peu élevé, sa flexibilité qui le rend propre à de nombreuses applications et sa grande disponibilité. Pour la carte, de nombreux produits présents sur le marché pouvaient faire l'affaire. Vesta Board, RoboCube, Kit331, toutes pouvaient convenir. Pour des raisons de disponibilité, la carte Kit331, principalement composée d'un microcontrôleur Motorola 68331 à 16MHz, de 256 ko de RAM et de 256 ko de ROM, a été retenue.

3.2 Matériel du bus de communication EME

L'interface de communication EME est basée sur le port SPI de Motorola [22]. Il s'agit d'une interface dite "à trois fils": données en émission (TxD), données en réception (RxD) et synchronisation (CLK, ou plus précisément SCK). À cela s'ajoutent des signaux de sélection (PCS, *Peripheral chip select*).

Ce port série diffère d'un port série RS-232 (PC, Macintosh) fondamentalement de trois façons. D'abord, contrairement au RS-232, il s'agit d'un port série synchrone, c'est-à-dire qu'un signal d'horloge doit être communiqué au récepteur sur un brin séparé (ligne SCK). Ensuite, le port n'est jamais en émission ou en réception: quand il communique, c'est toujours dans les deux sens simultanément. Enfin, les pattes d'entrées/sorties (E/S) ne sont pas dédiées. Le port est en permanence dans l'un de deux états: maître ou esclave. Selon l'état dans lequel on se trouve, le rôle des pattes E/S s'inverse: l'entrée en mode maître est la sortie en mode esclave (d'où les noms MISO, *Master In Slave Out* et MOSI, *Master Out Slave In*).

Ces particularités interdisent plusieurs techniques de communications traditionnelles dans un environnement composé de plus de deux équipements. Il est par exemple impossible d'utiliser un protocole en anneau, car l'équipement qui recevrait des données d'un autre répondrait par son fil de retour à un troisième. Les problèmes de synchronisation qui en découleraient seraient colossaux.

La communication perpétuellement bidirectionnelle cause un autre problème: les multiples réponses. Si un maître communique simultanément avec plusieurs esclaves, en émission, le seul problème dont il faut se préoccuper est de ne pas surcharger la patte de sortie du maître. Mais sur le fil de retour, chacun des esclaves va vouloir communiquer, ce qui met en péril les étages de sortie de tous ces contrôleurs. La solution traditionnelle à ce problème est d'utiliser les circuits de sélection: bien que toutes les pattes de tous les circuits soient raccordées ensemble, seul un des récepteurs est sélectionné et activé. C'est la solution retenue par exemple sur la carte Kameleon. Mais cette solution requiert une seconde prémisse pour conserver son apparente simplicité: un maître de système unique. Ce maître a un contrôle total sur quel esclave est actif à quel moment. Dans le cas où on souhaite que chacune des cartes puisse devenir maître, cette solution devient vite inapplicable. En effet, pour chaque nouvelle carte ajoutée au système, il faut ajouter deux fils la reliant à chacune des autres cartes

du système, un pour «si je suis maître, je dois pouvoir te désactiver» et un autre pour «si je suis esclave, tu dois pouvoir me désactiver». Dans notre cas, la prémisse "maître unique" n'était pas acceptable car cela signifie qu'en cas d'erreur fatale sur la carte maître, l'ensemble du système tombe. Accepter cela, c'est perdre un des gros avantages des systèmes multiprocesseurs asymétriques, soit la possibilité de redondance, de surveillance mutuelle et de redémarrage forcé au besoin.

À ce problème, deux solutions sont envisageables. La première est une combinaison de matériel et de logiciel. Il serait possible de créer un protocole qui permettrait d'indiquer quel esclave doit répondre et lequel ne le doit pas. Mais pour ce faire, il faut un moyen de désactiver les transmetteurs non concernés. Il semble que cela soit possible avec les ports compatibles SPI des microcontrôleurs Siemens [13] mais que ça ne le soit pas avec ceux de Motorola [22]. La seconde solution est radicale: supprimer le fil de retour (MISO). Du même coup, la

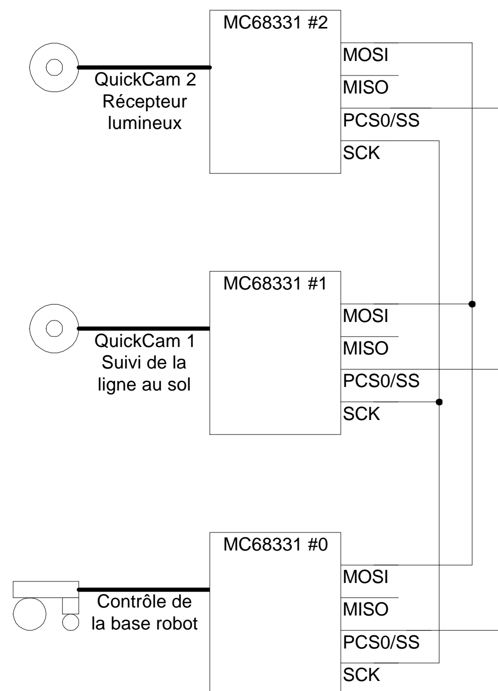


Figure 3.2: Schéma-bloc des communications dans EME, avec l'application de test

communication devient unidirectionnelle. Seul le maître garde le droit de parole, ce qui oblige maintenant à faire en sorte que toutes les cartes deviennent maître à tour de rôle. C'est cette solution qui a été retenue et implantée sur EME.

La figure 3.2 présente le schéma-bloc des communications dans l'architecture EME, telle qu'elle a été implantée dans l'application de test décrite à la section 4. Le schéma électrique détaillé est présenté en annexe 1. Les détails du protocole de communication sont présentés à la section 3.4.

3.3 Noyau temps réel ILNI

Dans les autres composantes du projet EME, seul le système d'exploitation aurait pu être acheté ou récupéré parmi les produits gratuits déjà disponibles. Nous cherchions une solution polyvalente, performante et peu coûteuse. La solution s'approchant le plus de cette description est µCOS, mais quelques irritants l'ont néanmoins écartée:

- les conditions de licence, restrictives si une production commerciale d'un produit basé sur EME devait voir le jour;
- la contrainte que toutes les tâches ont un niveau de priorité différent. Ainsi, il est impossible de partager également le temps processeur entre deux tâches qui l'utiliseraient chacune en totalité. Les tâches de faible priorité n'ont accès au processeur que dans les temps morts des tâches de priorité plus élevée, ce que nous considérons inacceptable.

La conclusion est donc qu'un système d'exploitation basé sur un noyau temps réel préemptif a été développé pour EME. Il porte le nom de ILNI (sans signification particulière). Ce noyau peut être qualifié de "machine traditionnelle", par opposition aux machines piles et aux machines virtuelles décrites à la section 2.4. En effet, ces solutions, bien qu'ayant des avantages, ont un coût important en complexité de développement, en perte de puissance de

TABLEAU 3.1: PRINCIPAUX SERVICES DU NOYAU ILNI OFFERTS À L'APPLICATION UTILISATEUR

Gestion des tâches	
`KRN_CreateTask()`	Crée une tâche.
`KRN_GetTaskID()`	Retourne le numéro d'identification de la tâche (son PID).
`KRN_Kill()`	Détruit une tâche.
`KRN_GiveCPU()`	Donne le processeur à la prochaine tâche.
`KRN_Suspend()`	Suspend la tâche pour une durée spécifiée.
Gestion des boîtes aux lettres	
`MBX_Send()`	Envoie un message.
`MBX_Rcv()`	Récupère un message.
`MBX_IsEmpty()`	Vérifie la présence d'un message.
Gestion des sémaphores	
`SEM_CreateOne()`	Initialise un sémaphore.
`SEM_Wait()`	Attend que le passage soit libre.
`SEM_Signal()`	Libère le passage.
Services de nom	
(pas de fonction spécifique, utilisés de manière transparente à travers les autres services)	
Services multiprocesseurs	
`SPI_GetBoardID()`	Retourne le numéro d'identification de la carte locale.
`MBX_Send()`	Envoie un message (version multiprocesseur).

calcul et en "isolement" (leur spécificité les place dans un monde à part, les rendant incompatibles avec presque tout ce qui existe actuellement).

Il a été envisagé au début du développement de rendre ILNI conforme à la norme de spécification µITRON [46] [28], mais il est vite devenu évident que le produit fini différerait tellement de la norme, tant par les ajouts que par les omissions, que cette conformité a été abandonnée. Il en reste néanmoins quelques traces, comme par exemple les codes de retour des fonctions.

Tout comme EME, ILNI a été réalisé en langage C, avec un minimum d'assembleur pour les rares situations incontournables comme la manipulation des registres du processeur. Le compilateur-assembleur utilisé est le *CrossCode* version 7.0 de la compagnie *Software Development Systems, Inc* [47]. Il s'agit d'un produit commercial. L'utilisation d'un produit gratuit comme le compilateur GCC de GNU [48] aurait été préférable, mais la disponibilité de SDS, la qualité de sa documentation de même que la possibilité d'utiliser une version de démonstration gratuite l'ont rendu plus intéressant.

Les prochaines sections présentent les principales caractéristiques du noyau, ainsi que les services offerts à l'application utilisant EME. Ces services sont présentés au tableau 3.1. Le détail de l'organisation interne du noyau dépasse les buts du présent document. Le lecteur intéressé devrait trouver réponse à ses questions dans le code source de EME. Les moyens de se le procurer sont présentés sur Internet [49].

3.3.1 Gestionnaire des tâches

La première composante du système d'exploitation est le gestionnaire de tâches, le noyau temps réel proprement dit. Ses principales caractéristiques sont les suivantes:

---

46 http://tron.um.u-tokyo.ac.jp/TRON/ITRON/home-e.html
47 http://www.sdsi.com
48 http://www.gnu.org/software/gcc/gcc.html
49 http://www.gel.usherb.ca/laborius/

• *Préemptif*. Par opposition à un noyau dit "coopératif", un noyau préemptif se distingue par le fait que c'est lui qui décide combien de temps chaque tâche peut disposer du processeur. C'est le noyau qui décide quand le moment d'un changement de tâche est arrivé et non pas la tâche elle-même. Dans un système coopératif, c'est cette dernière qui déciderait quand rendre le contrôle par appel à un service du noyau. Bien que plus complexe à développer et à utiliser, un noyau préemptif assure une bonne répartition du temps processeur entre les différentes tâches, un avantage que nous avons jugé indispensable.

• *Un seul niveau de priorité*. Actuellement, il est impossible de signaler à ILNI des tâches prioritaires par rapport à d'autres. Habituellement, les noyaux temps réel ont au moins quelques niveaux de priorités différents. L'avantage de n'avoir qu'un seul niveau est une simplification du code du gestionnaire des tâches. Il est toutefois prévisible que cette caractéristique sera une des premières à être modifiée dans une prochaine version de ILNI.

• *Caractéristiques de l'ordonnanceur*. L'ordonnanceur (*scheduler*) est la partie du noyau qui détermine l'ordre dans lequel les tâches accéderont au processeur. Il a été développé avec comme fondement que toutes les tâches d'un même niveau de priorité sont strictement égales. Cela trace la voie à un ordonnanceur de type *round-robin* (premier arrivé, premier servi). Le noyau étant multitâche et préemptif, l'ordonnanceur interdit qu'une tâche monopolise le processeur pendant plus d'un quantum de temps, soit une durée fixée arbitrairement à 10 ms. Les tâches disposent également d'un moyen de libérer le processeur avant la fin du quantum, soit `KRN_GiveCPU()`.

L'interruption périodique de l'ordonnanceur utilise le système *Periodic Interrupt Timer (PIT)* du 68331. Ce système permet des interruptions à une fréquence programmable, mais n'offre aucun contrôle sur l'instant du début de la période. Cela a des conséquences, par exemple dans une séquence d'événements comme celle-ci:

- l'interruption périodique donne le processeur à une tâche; l'interruption est reprogrammée pour lui enlever le processeur dans 10 ms;
- au bout de 5 ms, la tâche bloque et rend le processeur;
- un nouvelle tâche est mise sur le processeur;
- 5 ms plus tard, l'interruption périodique survient.

La seconde tâche n'aura eu le processeur que pendant 5 ms, au lieu des 10 ms prévues. Pour régler cela, l'interruption périodique est programmée pour survenir toutes les 2 ms et l'ordonnanceur ne fait effectivement un changement de tâche qu'à l'épuisement des 10 ms allouées.

• *Nombre maximal de tâches simultanées et taille des piles des processus*. Chaque tâche occupe un espace dans une liste système. Cette liste est de taille fixe, définie à la compilation, et ne peut donc accepter qu'un certain nombre de tâches simultanées (actuellement 20) [50]. À la destruction d'une tâche, son espace est rendu à nouveau disponible.

De la même manière, la pile que le système fournit à chaque tâche est créée en même temps que la liste des tâches, à l'initialisation du système. Cela implique que la taille de la pile est constante (définie à la compilation du noyau) et identique pour toutes les tâches du système.

Ces caractéristiques pourraient être modifiées aisément, car elles n'affectent que les services de création et de destruction de tâche et l'initialisation du noyau. Elles sont également propres à chaque implantation de ILNI et peuvent donc différer d'une carte à l'autre dans un même système, en fonction des besoins de chacune.

• *Suspension de la préemption*. Il est possible pour une tâche d'interdire la préemption. Cette fonction ne devrait cependant servir qu'à protéger des sections critiques, et non pas à monopoliser le processeur. Il est important de se souvenir que les tâches systèmes, même les

---

[50] Ceci est sans lien avec le numéro d'identification d'une tâche (*PID*), qui est un nombre signé de 32 bits.

tâches critiques, ont la même priorité que les tâches utilisateurs. Leur seul moyen d'obtenir le processeur est par l'ordonnanceur, comme toutes les autres tâches. Les sections critiques doivent donc être maintenues courtes. Il est aussi à noter que ILNI n'offre pas de moyen d'obtenir davantage de temps processeur comme on en rencontre sur certains noyaux, ces services étant habituellement basés sur une suspension partielle ou totale de la préemption.

- *Services de temporisateurs*. Trois approches sont proposées pour des services de délais. La première est une interruption périodique, appelée à chaque deux millisecondes, qui peut être reconfigurée pour appeler du code utilisateur. La seconde est un service de suspension de tâche. Le délai minimum de suspension est de 10 ms, mais le délai exact dépend du nombre de tâches en attente du processeur au moment de la fin de la période d'attente. La troisième consiste à utiliser l'appel système `KRN_GiveCPU()` qui garantit un certain délai d'attente, de durée variable mais courte. Cet appel va offrir le processeur séquentiellement à toutes les autres tâches avant de l'offrir à nouveau à la tâche initiale. Comme autre solution, l'utilisateur a toujours le choix d'utiliser des boucles d'attente active (`while(i++ < 1000);`), bien que la combinaison de ces boucles avec un noyau préemptif comme ILNI peut mener à des résultats inattendus. L'usage de telles boucles dans des sections à préemption supprimée est fortement déconseillée.

3.3.2 Services auxiliaires

Pour être commode, un noyau ne peut se contenter de simplement exécuter plusieurs tâches simultanément. Voici les services auxiliaires de ILNI:

- *Boites aux lettres (module MBX)*. Ce module est destiné à relayer des messages d'une tâche à une autre. Les messages n'ont pas de type défini et sont considérés être des données brutes (des chaînes d'octets de longueur connue). Seules les tâches ont besoin de savoir comment interpréter les données reçues. Il n'y a pas de contrainte sur la longueur des messages, sinon

la capacité mémoire de la carte. Les messages sont stockés dans des boîtes aux lettres, identifiées par des numéros séquentiels. Ce sont ces numéros que les tâches utilisent pour identifier les boîtes. Le module est composé de deux services principaux, `MBX_Send()` et `MBX_Rcv()`. Le premier assure l'envoi des messages. Il envoie une chaîne d'octets dont on lui indique la longueur à la boîte aux lettres spécifiée. Le second service assure la réception de message. Au cas où la boîte serait vide, selon les paramètres d'appel, la fonction retournera une erreur, ou encore elle attendra l'arrivée d'un message (fonction bloquante). Le nombre de boîtes sur une carte est aussi une donnée fixée à la compilation du noyau (actuellement 30) [51]. Le nombre de messages par boîte est limité par la capacité mémoire de la carte. Le module MBX dispose aussi d'autres services, comme par exemple `MBX_IsEmpty()`, qui retourne vrai ou faux selon l'état actuel de la boîte.

- *Sémaphores (module SEM).* Les sémaphores sont les outils idéaux de gestion des sections critiques. Ils sont de loin préférables à la suspension de la préemption [29]. Les services offerts sont les traditionnels `SEM_Wait()` et `SEM_Signal()`. Les sémaphores implantés sont du type dit "généralisé", dans le sens que le sémaphore peut être programmé pour autoriser plus d'une tâche à la fois dans la section critique, si tel est le comportement désiré.

3.4 Interface logicielle de communication interprocesseur

Bien que présentés ici dans leur version "compatible ILNI", les services de communications inter-cartes sont développés comme un module autonome, afin d'assurer une portabilité maximale permettant de les adapter à pratiquement tout noyau temps réel commercial.

---

[51] Tout comme pour les tâches, ce nombre est indépendant du numéro d'identification de la boîte, qui est un nombre signé sur 16 bits.

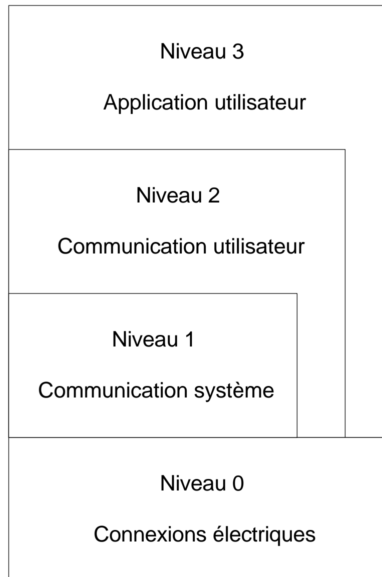


Figure 3.3: Modèle en couches

Le protocole de communication entre les cartes peut être vu de manière hiérarchique, tel qu'illustré à la figure 3.3. Ce modèle peut être vu comme une version simplifiée du modèle ISO [32], bien que la hiérarchie est légèrement différente et que les couches ne sont pas parfaitement isolées. Le niveau 0 correspond à la couche 1 (physique) du modèle ISO. Il ne contient pas de logiciel: ce ne sont que les connexions électriques, telles que décrites à la section 3.2. Le niveau 1 sert à échanger les messages requis pour la gestion de bas niveau du système de communication. Il est aussi responsable du contrôle de l'accès au bus de communication, au moyen de files d'attente. Cela correspond à une partie de la couche 2 (liaison de données) du modèle ISO. Le niveau 2 est celui où circulent les "véritables" données, soit les données utilisateurs, l'équivalent de la couche 4 (transport) dans le modèle ISO). Le niveau 3 est constitué de l'application de l'utilisateur qui utilise les services temps réel de EME, soit la couche 7 (application) du modèle ISO. Ce niveau est cependant responsable de la détection et de la correction des erreurs de communication, qui ne sont pas assurées par les niveaux inférieurs du modèle EME mais bien par l'application finale.

3.4.1 Communication système – niveau 1

Le principe de base des communications système est inspiré du protocole dit "bus à jeton" (*Token-Bus*) [32], qui a les caractéristiques suivantes:

- le bus de communication est unique, et toutes les cartes y sont connectées de la même manière;
- une carte n'a aucun moyen matériel de déterminer la présence des autres cartes;
- chaque carte possède un numéro d'identification unique;
- une seule carte a le droit d'émettre sur le bus en tout moment. C'est celle qui possède le jeton. Cette carte est aussi dite maître du bus;
- le jeton doit circuler de manière à rejoindre toute les cartes avant de retourner à une carte qui l'a déjà eu;
- la tolérance du système aux collisions sur le bus est très faible. En opération normale, de telles collisions ne devraient pratiquement jamais se produire.

Toutes ces caractéristiques s'appliquent au bus de EME. Comme autre caractéristique, EME ne contient aucun module spécifiquement dédié à la récupération des erreurs de communication. Aussi, l'utilisation d'un jeton et le fait que toutes les cartes doivent être maître de bus pour émettre rendent obligatoire la présence d'un processeur sur chaque carte, ou au moins de l'électronique capable de détecter à quel moment réémettre le jeton.

Comme dans pratiquement tous les systèmes de communications, les cartes EME ont besoin de moyens de s'identifier les unes aux autres. Dans le cas de EME, cet identificateur est un nombre compris entre 0 et 255. Tout comme en TCP/IP, certains nombres ont des significations particulières, comme pour la carte locale ou pour la diffusion générale (*broadcast*). La limite de 255 a été choisie arbitrairement. Elle peut être réduite, comme elle l'a été dans notre application de test, pour diminuer le temps de mise en route du système. Toute carte doit être en mesure de pouvoir détecter son propre numéro d'identification en appelant la fonction

SPI_GetBoardID(). Tel qu'implanté dans l'application de test, la limite a été fixée à quatre cartes, et la fonction SPI_GetBoardID() obtient l'identificateur de la carte locale par la lecture de deux broches d'entrée-sortie numériques, qui représentent le poids faible et le poids fort d'une adresse sur deux bits.

La figure 3.4 présente la procédure de démarrage d'un système EME. Elle s'effectue en trois

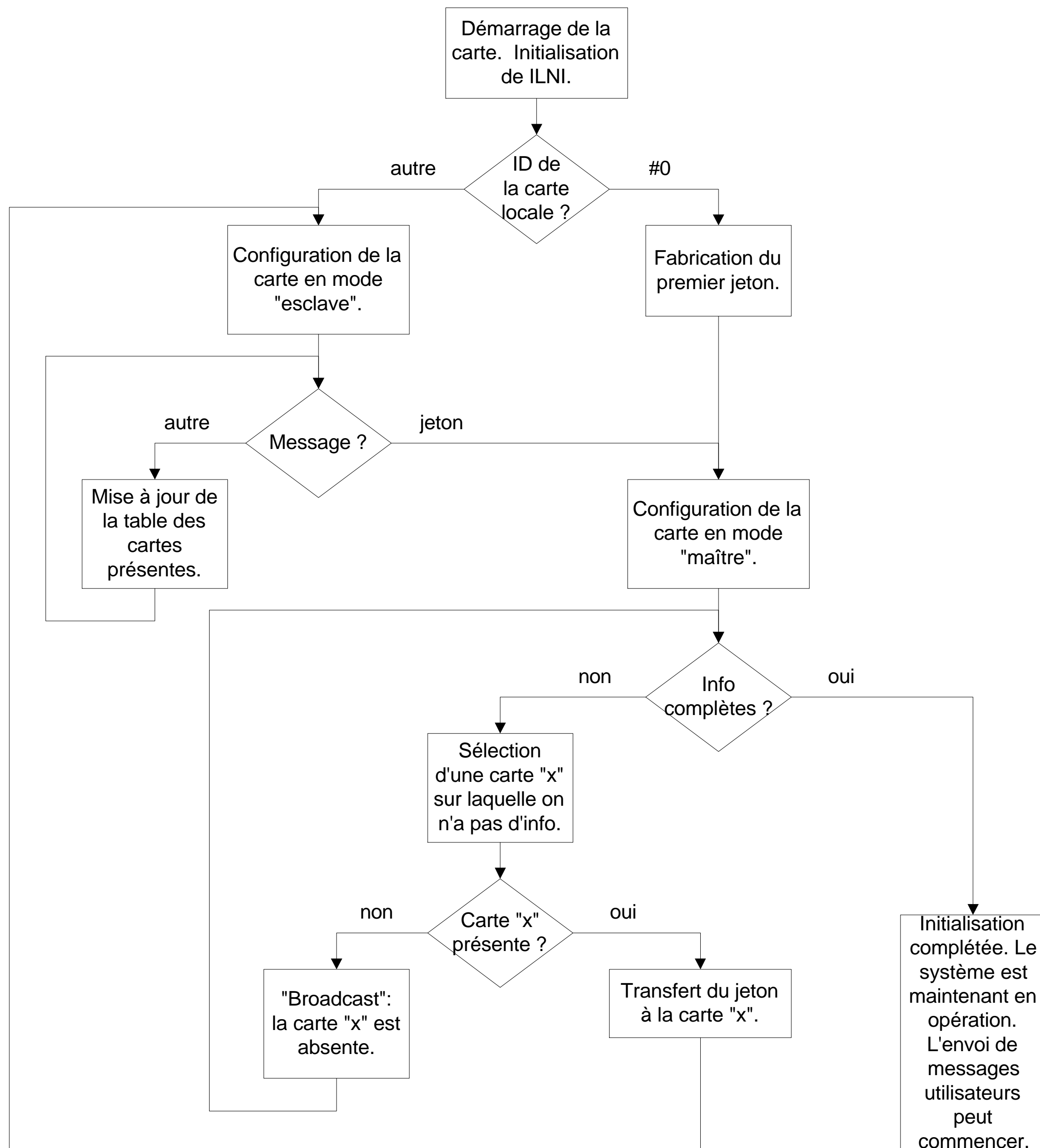


Figure 3.4: Procédure d'initialisation de l'interface de communication

étapes:

1. Création du premier jeton par la carte #0. Les autres cartes se mettent en attente. C'est le seul endroit où une carte a un comportement différent des autres.
2. Envoi du jeton à toutes les cartes, une par une, dans l'ordre numérique. Toute carte qui ne remet pas le jeton en circulation rapidement est considérée absente du système, et est marquée comme telle dans la table des cartes présentes que chaque carte maintient localement.
3. Une fois que le jeton a fait un tour complet de l'anneau virtuel, la phase d'initialisation est

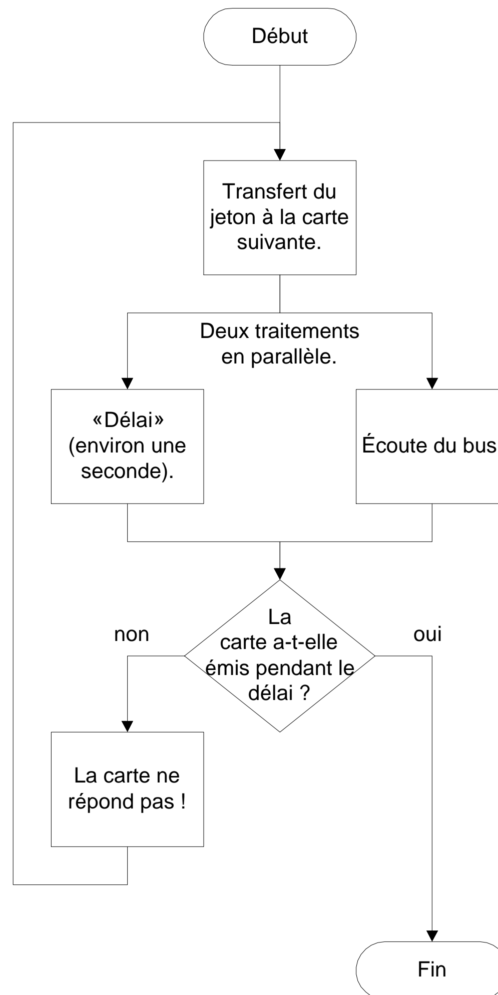


Figure 3.5: Détection des cartes en mode d'opération

complétée et l'envoi des messages de données peut débuter.

Pendant l'opération du système, le mécanisme qui a permis la détection initiale des cartes présentes demeure en service. Son fonctionnement est détaillé à la figure 3.5. Il ne vise toutefois plus à détecter de nouvelles cartes, mais plutôt à détecter rapidement le mauvais fonctionnement d'une des cartes détectée précédemment.

À l'étude de la figure 3.5, il est évident que cet algorithme ne détecte pas toutes les situations d'erreurs. Un des principaux trous dans la surveillance a été créé volontairement pour permettre une plus grande extensibilité du système. Dans les applications basées sur des microcontrôleurs, le port de communication SPI de ces derniers en est un qui est fort utilisé. Il était donc difficilement concevable qu'un système multiprocesseur n'ait aucun port SPI disponible. La solution proposée pour régler ce problème consiste à rendre temporairement inaccessible les communications de niveau 0 (figure 3.3) pour toutes les cartes EME. Ce niveau devient alors disponible pour les communications SPI traditionnelles. L'algorithme logiciel est suivant:

1. La carte reçoit le jeton; elle devient maître du bus EME.
2. Elle émet sur le bus EME pour signaler qu'elle est présente, désactivant les services de surveillance du bus.
3. Elle fait ses échanges sur le bus SPI.
4. Elle réactive le bus EME en réémettant le jeton.

Cette solution ne fonctionne que pour les cas où le microcontrôleur initie les transferts vers le périphérique SPI, ce qui est habituellement le cas. Si toutefois le périphérique peut initier des transferts, cette procédure ne fonctionne pas. La clé de la solution réside dans le fait que le bus EME, comme tous les bus SPI, est basé sur des signaux de sélection (*chip select*), contrôlé par la ligne PCS0 pour le bus EME (voir annexe 1). Ces signaux permettent de faire en sorte que les autres cartes EME, tout comme les périphériques SPI, peuvent ne pas entendre les

messages présents sur leurs pattes quand ils ne les concernent pas. Bien entendu, pendant la communication SPI, toutes les communications EME sont suspendues. Ce temps de suspension doit donc être conservé le plus court possible.

3.4.2 Format du paquet de communication

Sur le bus EME, il n'existe qu'un seul format de message. C'est le même pour tous les niveaux de communication. Il est de longueur fixe, soit 32 octets. Cette longueur a été retenue pour plusieurs raisons. D'abord, sur les processeurs CPU32, c'est le plus gros paquet qui peut être envoyé en une seule opération. Ensuite, c'est une taille commode dans le sens que, même en tenant compte des entêtes, elle permet d'envoyer une grande proportion des messages utilisateurs en un seul envoi (c'est-à-dire sans segmentation de message) tout en conservant le temps de transmission du paquet sur le bus dans des limites raisonnables. Les champs du paquet sont les suivants:

- carte source (8 bits), l'émetteur du paquet;
- carte cible (8 bits), le destinataire;
- version (8 bits), le numéro de version du paquet, au cas où le développement de nouveaux modèles de paquets serait requis;
- commande (8 bits), le type d'informations contenues dans le paquet;
- données (28 octets), les données de la communication niveau 1. Ce champ peut être utilisé directement comme il peut être redivisé par les logiciels des niveaux de communication supérieurs, selon leurs propres besoins.

Les communications de niveau 1 utilisent plusieurs commandes différentes. Le plus souvent, aucune donnée n'est associée au message, ou tout au plus une donnée qui ne déborde pas les 28 octets disponibles dans le paquet. La commande la plus typique est `CMD_GiveToken`, pour passer le jeton à une autre carte.

3.4.3 Communication utilisateur – niveau 2

La principale responsabilité de EME, dans les communications de niveau 2, est de fournir des points d'interface avec les services de ILNI et ainsi transmettre les données de l'utilisateur. Ceci peut être vu comme la zone couvrant l'intersection des deux ovales en haut ou en bas de la figure 3.1, le restant des ovales de droite représentant le niveau 1. En fait, EME n'implémente actuellement qu'un seul service de communication multicarte, soit une extension du service de boîtes aux lettres (module *MBX*). Les autres services (comme les sémaphores) n'ont pas pour l'instant d'implantation multicarte. Cette simplicité a toutefois l'avantage de mieux découpler les modules "noyau temps réel" et "communication interprocesseur" de EME, ce qui permet une plus grande réutilisabilité de chacun de ces modules.

Toujours pour assurer une portabilité maximale, l'implantation de la version multicarte du service de boîte aux lettres a été simplifiée au maximum, ce qui se traduit par des services asymétriques. En effet, il est possible d'envoyer des messages sur une autre carte avec `MBX_Send()`, mais on ne peut utiliser `MBX_Rcv()` que pour aller en chercher dans une boîte locale, i.e. sur la carte locale. En apparence [52], l'implantation des services multicartes pour ILNI s'est donc limitée à ajouter un nouveau paramètre à la fonction d'envoi de message, soit le numéro de la carte destination.

Pour acheminer les messages des communications niveau 2, trois nouveaux types de paquets sont créés: un pour les messages qui peuvent être expédiés en un seul envoi et deux pour les messages dont l'acheminement requiert plus d'un paquet sur le bus EME. Dans tous les cas, le champ "données" du paquet est redivisé de manière appropriée. Citons les

---

[52] Les autres composantes de la modification de ILNI relèvent davantage de la communication niveau 1: tâches et routines d'interruption pour la gestion du port SPI, files d'attentes de paquets reçus et à envoyer pas encore acheminés, etc.

principales nouvelles informations à transmettre (toutes ne sont pas à transmettre dans tous les cas):

- la boîte aux lettres cible [53];
- la boîte aux lettres source, ou celle à laquelle on peut répondre s'il y a lieu;
- la longueur totale du message;
- un index dans le message, pour indiquer quel segment est inclus dans le paquet;
- un identificateur de transmission, qui identifie à quel message un paquet appartient;
- et les données elles-mêmes.

En tenant compte de tous ces nouveaux champs, la longueur maximale d'un message utilisateur qui peut être envoyé en un seul envoi est de 20 octets. Dans le cas d'un message dépassant cette taille, le premier paquet contiendra 16 octets de données et tous les paquets subséquents en contiendront 22. La taille maximale des données utilisateurs (niveau 3) qui peuvent être envoyées par ce système est limitée par les champs d'index et de longueur de message. Dans la version actuelle, ces champs ont chacun 16 bits, limitant donc les messages à 64ko. Il est toujours possible de définir de nouveaux types de messages avec des limites encore plus élevées, mais il faut prendre garde de ne pas monopoliser le contrôle du bus. Le code actuel règle ce problème en n'autorisant l'envoi que d'un maximum de cinq segments de message consécutifs avant de remettre le jeton en circulation.

3.4.4 Services de nom

Comme l'a démontré l'Internet, l'utilisation de numéros pour identifier des équipements est obligatoire mais fort malcommode. Le problème est le même pour l'identification des cartes d'un système EME. Ce problème peut également être étendu aux boîtes aux lettres, aux

[53] La taille de ce champ dépend des propriétés du module *MBX*, à moins qu'on ne choisisse de rendre certaines boîtes inaccessibles aux envois inter-cartes.

sémaphores et même aux tâches d'un système EME. La solution retenue est un service de nom, fortement inspiré des *Domain Name Services (DNS)* d'Internet.

La solution consiste à créer sur chaque carte une tâche DNS. Cette tâche a pour mandat de répondre à toutes les requêtes des programmes situés sur la carte locale ainsi qu'à celles des DNS des autres cartes. Le principe est donc que toutes les requêtes des programmes sont adressées au DNS local, qui se charge d'obtenir la réponse des autres DNS si la requête ne concerne pas un item local. Pour accomplir cela, chaque DNS a en sa possession la table complète des correspondances "nom de carte" – "numéro de carte", créée à l'initialisation du système.

Dans la version actuelle de EME, le service DNS est complété pour l'identification des cartes uniquement. Les requêtes pour les autres services DNS (boîtes aux lettre, sémaphores, tâches) sont correctement reconnues mais non traitées.

3.5 Sommaire

Cette section a présenté les quatre principaux modules d'un système EME:

- des cartes à microcontrôleurs, actuellement limitées aux cartes basées sur un processeur Motorola CPU32;
- le bus de communication qui les relie, basé sur le port série multipoint SPI;
- le noyau temps réel ILNI, module autonome présent sur chaque carte et utilisable indépendamment de EME;
- et les services logiciels de communication inter-cartes, composés du logiciel de communication autonome (niveau 1) amélioré des interfaces pour ILNI (niveau 2, par l'extension du service de boîtes aux lettres *MBX*).

Lorsque tous ces modules sont initialisés, le système EME est fonctionnel. Chaque carte compte cinq tâches en service qui assurent le fonctionnement et la surveillance de l'ensemble du système.

4. VALIDATION DE EME AVEC UN ROBOT MOBILE

Pour valider l'environnement EME, nous souhaitions avoir une application qui démontre ses caractéristiques, notamment la possibilité qu'il offre de créer un système disposant d'une grande capacité d'échange de données (E/S) avec beaucoup de puissance de calcul. L'application retenue est l'utilisation de deux caméras sur un même robot mobile. C'est une application rare, les seuls cas similaires courants sont la combinaison de deux images pour fournir au robot une vision stéréoscopique. Dans notre cas, le but était plutôt orienté vers la détection et le suivi simultané de plusieurs objets. Ces objets peuvent se trouver dans des angles incompatibles pour une seule caméra et, de par leur nature ou leur position dans l'espace, requérir des réglages de caméras différents. Cette application est en fait une suite de celle réalisée par Drolet [10] et qui consiste à suivre une ligne au sol. Cette application avait montré qu'interfacer une caméra *QuickCam* de Connectix avec un microcontrôleur Motorola 68332 est relativement facile à faire et efficace, mais que le débit de transfert des images vers le contrôleur est très limité. L'ajout d'une seconde caméra contrôlée par le même processeur est physiquement possible, mais la fréquence de rafraîchissement de l'image deviendrait inacceptable. Passer à un microcontrôleur plus rapide ne serait d'aucune utilité, puisque cela n'améliorerait guère le débit disponible sur les interfaces d'échange de donnée [9]. La solution réside dans l'utilisation d'une approche multiprocesseur telle EME.

D'une manière plus précise, le robot doit se promener dans son environnement en suivant une ligne continue tracée au sol à son intention. Il doit aussi faire de son mieux pour éviter tout obstacle se trouvant sur sa route. Enfin, on espère de lui qu'il reconnaisse ses échecs au lieu de s'embourber encore plus en essayant de s'en sortir. Concrètement, cela signifie qu'au cas où il perd son tracé de vue, il doit tenter de le retrouver mais sans s'obstiner. S'il n'y arrive pas rapidement, il doit s'arrêter et demander de l'aide. C'est là une situation typique où la seconde

Figure 4.1: Robot *BigBrother*

caméra entre en jeu. Cette dernière est en permanence à la recherche d'un signal de communication lumineux, signal au moyen duquel un humain (ou éventuellement un autre robot) peut lui donner des informations ou des instructions. La communication par signalisation lumineuse n'est pour l'instant guère répandue en robotique mobile. En contrepartie de la bande passante très réduite en comparaison avec les communications radio, elle offre plusieurs avantages, spécialement celui de fournir une information sur la position de l'interlocuteur [20].

Cette section présente la configuration matérielle et logicielle de EME utilisée pour réaliser cette application. Elle commence par les aspects matériels, soit la configuration du robot et l'émetteur lumineux, et continue avec les aspects logiciels, c'est-à-dire le traitement des images provenant des caméras, puis le module de contrôle qui permet au robot de naviguer dans l'environnement selon les informations provenant des différents capteurs.

4.1 Configuration matérielle du robot

La base mobile utilisée est un robot modèle ROBUS [19]. Il est équipé de capteurs infrarouges et de collisions, et interagit avec son environnement par ses déplacements et en

émettant des sons. Pour la validation de EME, trois cartes Kit331 ainsi que les deux caméras sont installées sur cette base. Les trois cartes sont reliées par le bus EME et empilées verticalement, comme le montre la figure 4.1. L'ensemble a été nommé *BigBrother* parce que le robot est beaucoup plus haut que la base ROBUS originale et à cause des deux caméras embarquées.

Le schéma-bloc de l'application *BigBrother* est présenté à la figure 3.2. La carte dont le numéro EME est zéro est la plus basse dans la structure. C'est celle qui fait la mise en commun des données captées par l'ensemble des senseurs du robot et qui prend les décisions quant aux actions à prendre. C'est aussi sur cette carte que sont raccordés la quasi-totalité des capteurs et actionneurs de la base mobile. La carte du milieu est la carte #1. Elle est en charge de la caméra 1, celle qui suit la ligne tracée au sol. Suite au fait que la carte #0 manquait de connecteurs facilement accessibles, le haut-parleur de la base ROBUS est aussi raccordé à la carte #1. Tout en haut se trouve la carte #2, responsable de la caméra 2, le

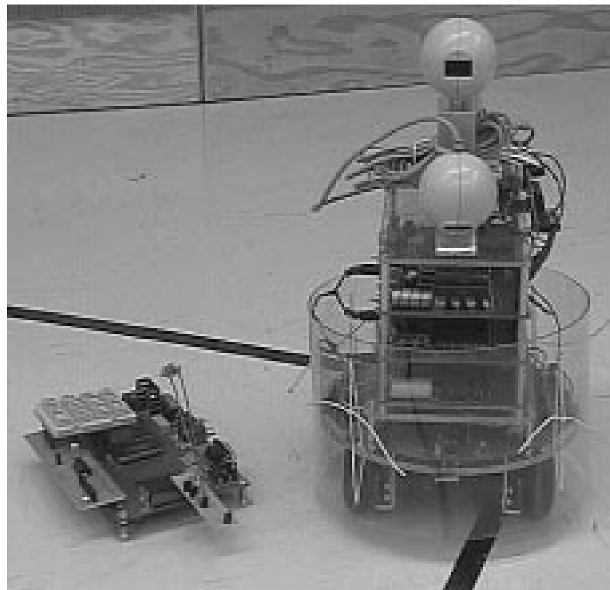


Figure 4.2: *BigBrother* et l'émetteur lumineux externe

récepteur de la communication lumineuse. L'interface de communication entre les cartes Kit331 et les caméras QuickCam a été directement récupérée du projet décrit en [10]. Le schéma électrique de cette interface est disponible à l'annexe 2. Les caméras utilisées sur *BigBrother* sont deux QuickCam noir et blanc de Connectix, avec interface pour port parallèle d'ordinateur "PC".

Le poids total de *BigBrother* (un peu plus de 3 kg) peut être considéré comme le maximum pour ROBUS, car les axes des roues commencent à plier légèrement. Il faut dire que la base a été légèrement modifiée pour l'application: les 8 batteries "AA" du ROBUS standard ont été remplacées par 8 batteries "C", compte tenu de la charge électrique de *BigBrother*.

4.2 Dispositif externe de signalisation lumineuse

Pour la communication par signalisation lumineuse, l'émetteur est réalisé sur une autre carte Kit331, externe au robot (figure 4.2). Le choix de ce matériel, largement surdimensionné pour la tâche, permettra dans l'avenir d'implanter l'émetteur sur *BigBrother* sans modifications, histoire d'obtenir une communication lumineuse bidirectionnelle. L'interface électrique,

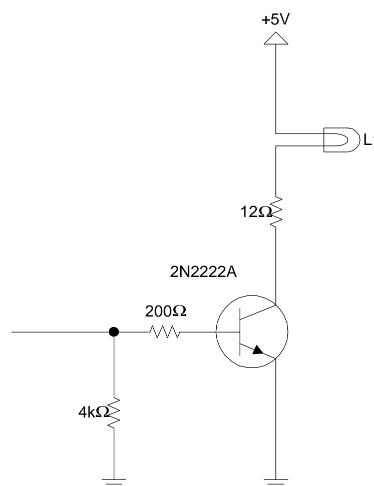


Figure 4.3: Émetteur lumineux

présentée à la figure 4.3, consiste simplement à diminuer la charge électrique sur la patte de sortie numérique du processeur. Un clavier téléphonique est aussi raccordé à cette carte, comme moyen d'entrer les données à transmettre.

Le protocole de communication est de type biphase (Manchester), c'est-à-dire un protocole où chaque bit d'information est représenté par une séquence de deux états. Dans le présent cas, le bit 0 est représenté par la séquence «lumière éteinte, puis allumée», et le bit 1 l'est par la séquence inverse «allumé puis éteint». Ce protocole a comme principal inconvénient de doubler la quantité de données à transmettre, donc dans un cas comme celui-ci où la quantité d'états reçus par seconde est limitée, de diminuer de moitié le débit. Il a par contre plusieurs avantages:

- facilité de synchronisation, le signal d'horloge étant transmis avec le message;
- détection d'interférence plus aisée: le passage d'un autre robot entre l'émetteur et le récepteur ne sera pas interprété comme un message comportant une grande suite de zéros mais bien comme une erreur de communication; de la même manière, un point très lumineux dans une pièce, comme une fenêtre, sera rapidement identifiée comme une source de bruit;
- identification de l'émetteur: il est théoriquement possible de donner une fréquence d'émission différente à chaque émetteur présent dans un certain environnement, permettant ainsi une identification instantanée du locuteur.

Toujours dans le but de pouvoir l'intégrer facilement à *BigBrother* dans l'avenir, le logiciel de la carte externe fonctionne sur le noyau ILNI. Dans ce cas-ci, c'est la version de base qui est utilisée, c'est-à-dire sans les additions pour les systèmes multiprocesseurs, ce qui démontre la modularité de EME. Le logiciel de contrôle de l'émetteur, constitué d'une seule tâche en plus de quelques routines d'interruptions, envoie des paquets de onze bits: un bit de départ (*start bit*), huit bits de données et deux bits d'arrêt (*stop bit*). Il n'y a pas de test de parité. Les données peuvent théoriquement contenir n'importe quoi, mais il est actuellement convenu d'interpréter

1 Devant à gauche	2 Devant	3 Devant à droite
4 À gauche	5 Arrêt	6 À droite
7 Derrière à gauche	8 Derrière	9 Derrière à droite
* Sans signification	0 Sans signification	# Sans signification

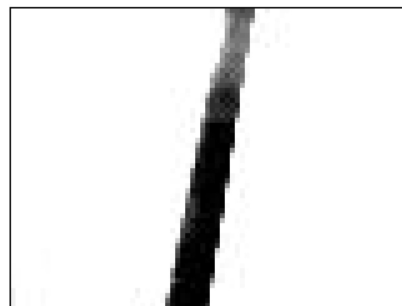
Figure 4.5: Commandes du dispositif de signalisation lumineuse

les messages comme étant des caractères ASCII. Les caractères envoyés sont ceux qui sont entrés sur le clavier téléphonique de la carte, ce qui limite aux caractères de 0 à 9, # et *. Le jeu de commandes, présenté à la figure 4.4, est actuellement uniquement composé de directives de déplacement et d'arrêt. Ces commandes peuvent servir à aider le robot à retrouver le tracé au sol ou simplement à lui donner un ordre à n'importe quel moment.

4.3 Traitement des images

Tout comme l'interface matérielle, le logiciel de bas niveau des caméras s'inspire de ce qui a été réalisé dans [10]. Excluant les fonctions d'initialisation et de calibrage, la fonction principale est celle qui renvoie un tableau contenant l'image lue. Pour *BigBrother*, le nombre d'images par

(a)



(b)

Figure 4.4: Effets des réglages de la caméra 1

seconde est plus important que la qualité de l'image. Par conséquent, pour les deux caméras, l'image récupérée de la caméra est la plus compacte autorisée par le jeu d'instructions de la caméra, soit 80x60 pixels, en 16 tons de gris.

Pour les deux caméras, le principe de traitement est similaire. Il débute par une opération de filtrage, effectuée directement au niveau de la caméra, au moyen des fonctions de calibration de cette dernière. Le but de cette opération est de rendre le plus visible possible les caractéristiques importantes de l'image et d'atténuer tous les parasites. Les étapes suivantes se déroulent sur la carte à microcontrôleur raccordée à la caméra. Le processus de traitement consiste en une cascade de tâches, qui se relaient leurs résultats au moyen des services de boîtes aux lettres. Le résultat final de ce traitement de même que les résultats intermédiaires pertinents sont renvoyés sous forme de message à la carte #0, pour interprétation finale et exécution des actions appropriées.

- *Caméra 1 – suivi du tracé au sol.* La figure 4.5 montre l'image vue par la caméra 1 avant (a) et après (b) l'opération de filtrage. À partir des images filtrées, l'information que le traitement logiciel doit trouver est:

1. Y a-t-il une ligne au sol dans le champ de vision ?
2. Où est-elle ? Centrée, ou décentrée vers la gauche ou vers la droite ?

Pour répondre à la question 1, un algorithme simple a été développé: si le nombre de pixels foncés dans l'image dépasse un certain seuil, alors une ligne est détectée. Pour la question 2, la position de l'image peut être vue comme un problème de calcul de centre de masse. Après conversion de l'image en une image noir et blanc pure, si on donne aux pixels blancs un poids de zéro (pas de ligne) et un poids de un aux pixels noirs, le centre de masse (CM) de l'objet ainsi créé correspond au centre de la partie de la ligne visible dans cette image. Pour un objet en une dimension, la formule à utiliser est l'équation 4-1, où x_i représente la coordonnée en x et $f(x_i)$, le poids à cet endroit.

$$CM_x = \frac{\sum (x_i * f(x_i))}{\sum f(x_i)} \quad (4\text{-}1)$$

En multidimension, les équations se complexifient. La simplification qui a été utilisée, illustrée à la figure 4.6, est de créer un vecteur ligne PC contenant la moyenne des poids de chacune des colonnes de l'image et un vecteur colonne PL contenant la moyenne des poids de chacune des lignes. L'application de l'équation (4-1) sur PC donne la coordonnée *x* du centre de masse de l'image; la même équation appliquée sur PL donne la coordonnée *y*. Ces coordonnées sont par la suite normalisées, le coin supérieur gauche de l'image étant désigné coordonnée (0,0) et le coin inférieur droit coordonnée (100,100). En fonction du nombre de pixels foncés dans l'image et de la coordonnée *x* du centre de masse de l'image, un des messages du tableau 4.1 est envoyé à la carte #0, indiquant le résultat de l'analyse. L'ensemble de ce traitement est réalisé par deux tâches fonctionnant en cascade.

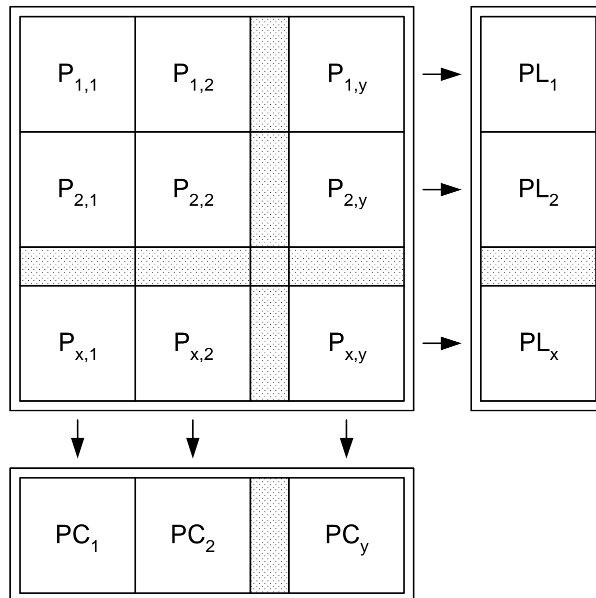


Figure 4.6: Simplification de l'image en vecteurs

TABLEAU 4.1: RÉSULTATS DE L'ANALYSE DE RECHERCHE DE LIGNE

Nom du message	CM_x	Signification
QC1Dir_None		Pas de ligne visible.
QC1Dir_FarLeft	$5 \leq x < 25$	Ligne dans l'extrême gauche de l'image.
QC1Dir_CloseLeft	$25 \leq x < 45$	Ligne légèrement à gauche.
QC1Dir_Center	$45 \leq x \leq 55$	Ligne droit devant.
QC1Dir_CloseRight	$55 < x \leq 75$	Ligne légèrement à droite.
QC1Dir_FarRight	$75 < x \leq 95$	Ligne dans l'extrême droite de l'image.
QC1Dir_Unknown		Traitement incomplet, ou erreur dans le traitement.

- *Caméra 2 – communication par signaux lumineux*. L'algorithme de gestion de la seconde caméra est une version évoluée de celui de la première. La principale différence est qu'il faut ici extraire une information temporelle, à savoir combien de temps l'émetteur est resté allumé ou éteint.

Au niveau des réglages de caméra, le filtrage est ici plus marqué. Contrairement à la caméra 1 qui scrute une surface de plancher et qui reçoit donc de la lumière réfléchie, la caméra 2 recherche une source active de lumière. Pour optimiser les lectures, des réglages de caméra extrêmes sont utilisés: contraste au maximum, décalage au minimum. L'image qui en découle est entièrement noire, sauf aux emplacements des sources lumineuses actives. La figure 4.7 illustre les images avant (a et c) et après (b et d) le filtrage pour l'émetteur éteint ainsi que pour l'émetteur allumé. La figure 4.7 (d) permet de constater que la présence de l'émetteur est beaucoup plus visible qu'avant le filtrage, même si le nombre réel de pixels blancs à l'emplacement de l'émetteur est moindre.

La seconde étape du traitement est un calcul de centre de masse en deux dimensions, avec comme seule différence par rapport à la caméra 1 que c'est le blanc qui est le cœur de l'étude.

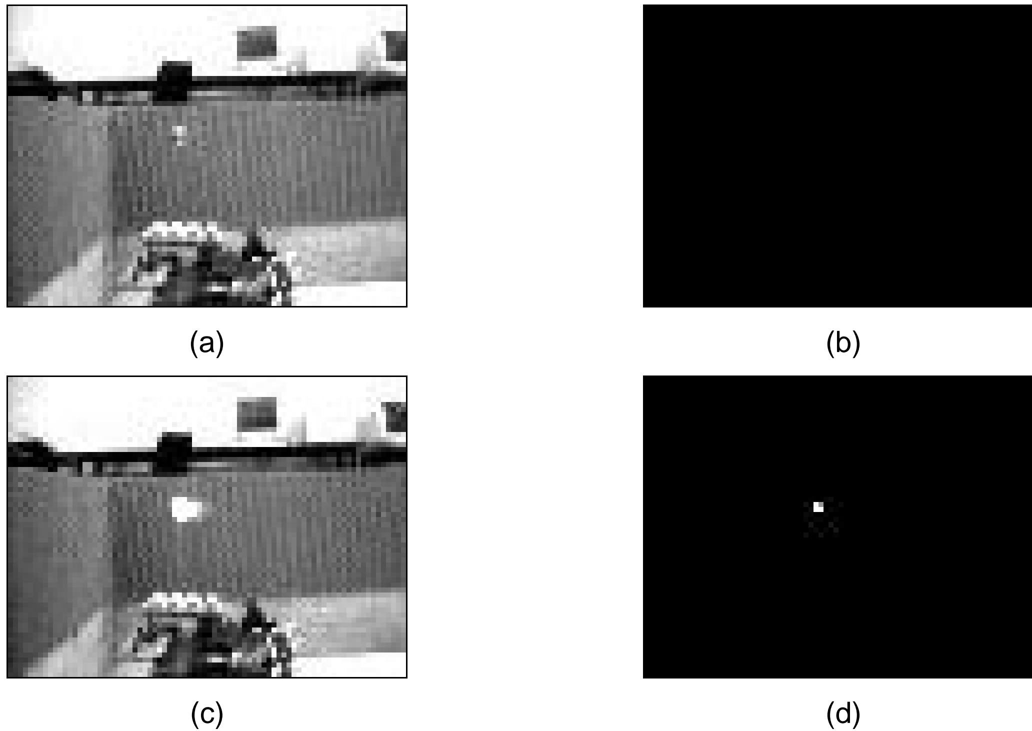


Figure 4.7: Effets des réglages de la caméra 2

Le reste du problème consiste à convertir la séquence d'images en une donnée de 8 bits, en détectant un maximum de conditions d'erreur. Les points clés de l'algorithme sont les suivants:

- La mesure du temps "émetteur allumé" ou "émetteur éteint" est réalisée par le comptage du nombre d'images où l'état est constant. C'est évidemment la résolution temporelle maximale de l'algorithme, dans le sens qu'une mesure discrète ne peut pas être plus précise qu'une unité entière. Par contre, le temps requis pour transférer une image de la caméra au processeur peut varier légèrement en fonction de la charge de ce dernier, ce qui cause une légère imprécision sur la mesure.
- L'algorithme contient une procédure d'auto-calibrage, c'est-à-dire qu'il mesure en continu la durée des signaux émis par l'ampoule, adaptation qui permet une meilleure réception et une meilleure détection des erreurs par une analyse plus précise. Le principe consiste à mesurer l'évolution dans le temps du nombre d'images par état de l'émetteur lumineux. Cet

algorithme permet du même coup une certaine adaptation à la variation du taux d'occupation du processeur.

- Il y a deux niveaux de synchronisation. Au niveau bit, le récepteur se recalibre sur chaque transition, donc au moins une fois par bit. Pour les messages entiers, la synchronisation ne se fait que sur le bit de démarrage. Elle pourrait se faire sur chaque double état (un 1 suivi d'un 0, ou le contraire, lorsque l'émetteur reste allumé ou éteint le double de la durée normale), mais cela ne s'est pas avéré nécessaire pour nos tests. Ces doubles états servent toutefois à la détection d'erreur.

L'ensemble de l'analyse est réparti sur cinq tâches. Les informations pertinentes sont relayées à la carte #0 dès qu'elles sont disponibles. La liste des messages possibles est présentée au tableau 4.2.

4.4 Contrôle du robot

Au niveau du contrôle, l'objectif est de combiner tous les modules précédents pour diriger le robot de manière à ce qu'il suive un tracé au sol, tout en évitant les obstacles autour de lui. En cas de perte de la ligne, il doit s'immobiliser. Il doit aussi en tout temps être à la recherche de signaux lumineux, être capable de les recevoir, de les interpréter et d'exécuter les commandes ainsi transmises. Pour réaliser ce contrôle, l'approche comportementale [2, 5] avec arbitrage par priorité, communément appelée *subsumption* [5], est utilisée.

TABLEAU 4.2: RÉSULTATS DE LA RECHERCHE DE MESSAGES LUMINEUX

Nom du message	Signification
`ToC_StartCom`	Émetteur détecté.
`ToC_Data`	Donnée valide reçue (message accompagné de la donnée reçue).
`ToC_EndCom`	Émetteur sorti du champ de vision.
`ToC_ProtocolError`	Erreur (toutes causes confondues).

Cette approche consiste à fournir au robot une série de modules générant des comportements particuliers, appelés "comportements". La figure 4.8 illustre les comportements de *BigBrother*. Suivre la ligne, éviter un obstacle, recevoir une commande en sont des exemples. Tous les comportements sont implantés comme des processus logiciels et s'exécutent en parallèle. Lorsque plusieurs comportements désirent simultanément prendre le contrôle du même actionneur, le processus de sélection *subsumption* entre en action. Le principe est que le contrôle de l'actionneur est donné en totalité au comportement qui a la plus haute priorité. Une liste des priorités est établie à cette fin et permet de déterminer en tout temps lequel des comportements contrôle chaque actionneur.

Sur *BigBrother*, le contrôle s'effectue au moyen de treize tâches situées sur la carte #0. L'ensemble du processus de prise de décision est basé sur les sources d'informations suivantes:

- les informations concernant la communication lumineuse, soit le début ou la fin de la communication, un message valide ou invalide reçu ainsi que le message lui-même, en provenance de la carte #2;
- la position relative de la ligne au sol ou l'absence de ligne visible, en provenance de la carte #1;
- l'état des capteurs de collisions, qui sont branchés sur la carte #0;
- la présence ou l'absence d'obstacles rapprochés, information provenant des capteurs infrarouges branchés eux aussi sur la carte #0.

À l'opposé, les actionneurs à contrôler sont les moteurs (vitesse et direction du déplacement) et le haut-parleur.

La figure 4.8 présente une vue d'ensemble de la prise de décision pour la gestion des moteurs. Il y a dans ce cas six comportements, avec une hiérarchie fixe des priorités, les comportements de haute priorité étant représentés en haut de la colonne. Les sources

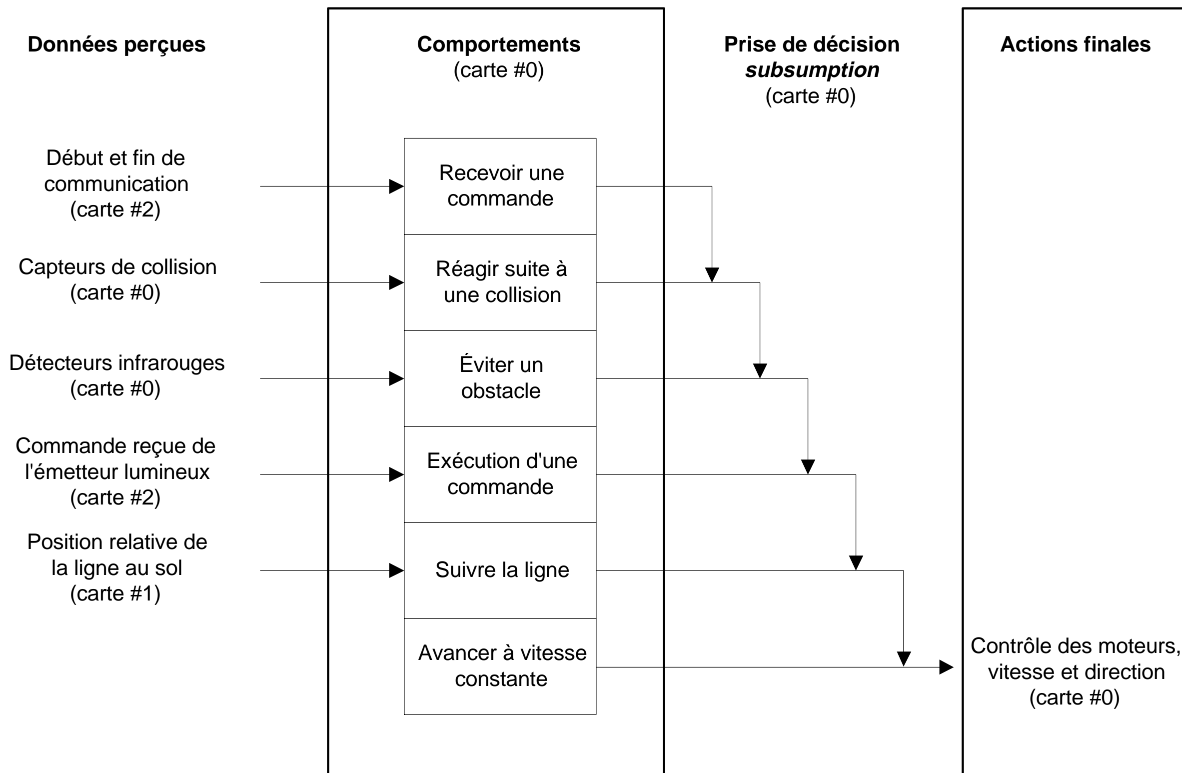


Figure 4.8: Schéma d'arbitrage des moteurs

d'informations qui influencent chaque comportement sont illustrées à gauche dans la figure et le processus d'arbitrage l'est à droite.

Les comportements "Avancer à vitesse constante", "Éviter un obstacle" et "Réagir suite à une collision" sont typiques d'un robot dont la tâche consiste à errer dans son environnement. Il avance tant qu'une information plus prioritaire ne vient pas lui dire de tourner, soit pour éviter une collision, soit parce qu'elle s'est produite. Le comportement "Suivre la ligne" ajoute des directives de faible priorité: si un tracé est visible, il est plus important de le suivre que d'avancer en ligne droite. Le comportement "Exécution d'une commande" permet à un intervenant externe d'influencer le comportement du robot. Ce comportement est plus prioritaire que le suivi de ligne de manière à ce que le robot puisse quitter le tracé au besoin, mais moins prioritaire que "Éviter un obstacle" et "Réagir suite à une collision" qui permettent une certaine

autonomie au robot. Quant au comportement "Réception de commande", son effet sur les moteurs consiste à immobiliser le robot pendant la réception d'un message par signalisation lumineuse. C'est le comportement le plus prioritaire, ce qui permet d'éviter que le déplacement du robot tel que souhaité par d'autres comportements ne perturbe la réception.

Le contrôle du haut-parleur est beaucoup plus simple. Tel que mentionné précédemment, la connexion du haut-parleur a été relocalisée sur la carte #1, pour une simple raison d'accessibilité des connecteurs. Le contrôle est assuré par une seule tâche sur cette carte, qui consiste à surveiller une boîte aux lettres et à faire jouer le message demandé. Il n'y a donc pas d'arbitrage formel, les sons sont joués dans leur ordre d'arrivée. La boîte aux lettres sert de file d'attente. Les requêtes pour des sons peuvent provenir des comportements, comme lors de la détection ou de la perte de la ligne, ou encore des tâches systèmes, par exemple au démarrage du système ou encore un son périodique indiquant le bon fonctionnement de l'ensemble.

4.5 Sommaire

Cette section a présenté l'application utilisée pour valider l'environnement EME. Il s'agit d'un robot mobile équipé de trois cartes à microcontrôleur et de deux caméras. L'intérêt de cette application est la présence de deux capteurs à haut débit, difficiles à intégrer sur un même système. L'application est composée de quatre modules principaux, soit la combinaison du matériel du robot, le dispositif de signalisation lumineuse externe, les modules logiciels de traitement des images ainsi que le logiciel de contrôle du robot.

5. RÉSULTATS

Plusieurs mesures ont été réalisées à différents niveaux sur l'environnement EME et sur *BigBrother* afin d'obtenir une vue d'ensemble de ses performances. Cette section présente les principaux résultats mesurés et les compare lorsque c'est possible.

5.1 Performances du noyau ILNI

Il existe quantité de mesures qui peuvent être faites pour qualifier les performances d'un noyau temps réel. Les développeurs de certains noyaux commerciaux vont jusqu'à fournir les temps minimum et maximum d'exécution de chacun des services du noyau. Il est toutefois important de noter qu'il est prévisible que les performances de ILNI soient moins optimales que celles de produits similaires. La raison est simple: le but du projet n'est pas de fabriquer un noyau temps réel plus performant que ce qui existe déjà puisque le dit noyau n'est qu'une des composantes du projet. Voici quelques-unes des mesures couramment utilisées:

- *Temps de changement de contexte*. Il s'agit du délai requis pour sauvegarder les registres du système, opération effectuée lors de chaque changement de tâche. Pour ILNI comme pour le noyau µCOS (version 1.08), ce délai est de l'ordre de 17 µs. Notez aussi que, bien qu'il soit régulièrement utilisé comme mesure de performance, le temps de changement de contexte brut n'est pas pertinent pour mesurer les performances d'un noyau [15], car ce délai dépend presque entièrement du nombre de registres du processeur utilisé et non pas du noyau.

- *Délais reliés aux interruptions*. Plusieurs mesures importantes sont reliées aux interruptions. Le délai entre la demande et le début de l'exécution du code d'interruption en est une, celui entre la fin du code d'interruption et le début de l'exécution de la tâche libérée par l'interruption en est une autre. La plus utilisée est la durée pendant laquelle les interruptions sont interdites

afin de protéger une section critique. Le calcul de cette valeur est ardu: il faut calculer, en nombre de coups d'horloge, le temps d'exécution de chaque instruction située dans la section critique. Pour les processeurs pipelinés (comme le CPU32), ce calcul doit tenir compte du fait que des segments de deux instructions consécutives peuvent se dérouler simultanément dans certaines conditions. Ce calcul n'a pas été fait, d'autant plus que comme dans la quasi-totalité des noyaux temps réel, le service de suspension des interruptions est offert au programmeur d'application. Il est donc possible que ce soit du code externe à ILNI qui le ralentisse le plus. Au lieu de la calculer, cette valeur a été mesurée directement à l'oscilloscope sur les différentes cartes de *BigBrother*. Cette technique a toutefois l'inconvénient de fournir une approximation du délai: une valeur particulièrement longue qui ne se produit que très rarement sera difficile à observer. Sur ILNI, les interruptions sont le plus fréquemment interdites pour une durée allant de 40 à 200 µs. Il arrive parfois que des pointes surviennent, variant de 0,5 à 1,5 ms. La source de ces longues périodes de suspension n'a pas été recherchée, mais il est présumé que ces délais pourraient facilement être optimisés. Une comparaison directe de ces valeurs est difficile. Il faudrait porter l'ensemble de l'application de test sur un autre noyau, ce qui n'est pas une mince tâche. La documentation de µCOS [15] fournit une valeur théorique, mais pour un processeur Intel 80186 16MHz. Dans ce cas, la durée de suspension des interruptions ne dépasse pas 31 µs. Il s'agit d'un processeur plus lent que le 68331 mais comportant beaucoup moins de registres, donc accélérant certaines sections critiques comme le changement de contexte. Une valeur semblable a été recherchée dans la documentation de QNX, mais on ne trouve que la mention "*in QNX, this is very small*" [25], sans valeur numérique. Il ne nous est donc pas possible d'effectuer une comparaison adéquate.

• *Efficacité globale*. Cette mesure a pour but de connaître quelle proportion du temps est consommée par l'ajout d'un noyau temps réel à une tâche simple. Dans le cas du présent test, cette tâche est tout simplement la boucle suivante:

```
int i=0;
while(i++ < 3000000);
```

Encore ici, il s'agit d'une mesure qui a ses limites, la principale étant qu'elle ne reflète que l'efficacité du code de changement de contexte; les services de messages, sémaphores et autres n'influencent pas le calcul. Elle a par contre l'avantage de donner une vision d'ensemble de la charge du système, car ce test peut être réutilisé à mesure que les tâches systèmes s'accroissent, par exemple lorsque les gestionnaires du système multicarte sont présents.

TABLEAU 5.1: POIDS PROCESSEUR DES NOYAUX

	Temps	Charge
Brut	37,3 s	0%
ILNI initial	42,1 s	11,5%
µCOS	37,8 s	1,3%
ILNI révisé	38,6 s	3,4%

Comme le montre le tableau 5.1, le temps de référence (sans noyau présent) est de 37,3 s. Le temps de ILNI en version uniprocesseur est de 42,1 s, dont 11,5% passés à exécuter le code du noyau. Quant à µCOS, il accomplit la tâche en 37,8 s, soit une efficacité de 98,7%. La principale différence entre ILNI et µCOS est une conséquence des caractéristiques de l'ordonnanceur (section 3.3.1), spécifiquement les cinq appels de l'interruption périodique entre deux changements de tâches. Si on décide d'utiliser la façon de faire traditionnelle et de programmer l'interruption périodique aux 10 ms, on obtient les chiffres dits "ILNI révisé", avec une efficacité qui passe de 88,5% à 96,6%, une amélioration de 8%. Cette version révisée n'a toutefois pas été conservée, principalement parce que le service de l'interruption périodique fournie à l'utilisateur (section 3.3.1, paragraphe *services de temporisateurs*) dépend de l'interruption de changement de contexte. La modification de la première aurait nécessité de profonds changements dans la seconde, qui est nécessaire pour le bon fonctionnement du robot.

Pour l'essentiel, la différence restante provient de la technique utilisée pour identifier la prochaine tâche à mettre sur le processeur. L'ordonnanceur de µCOS peut identifier cette tâche par une procédure séquentielle directe, alors que le code de ILNI contient une boucle pour vérifier chaque tâche potentiellement existante, ce qui est plus simple mais moins efficace. La situation présente est en fait un scénario du pire cas pour ILNI, car il n'y a qu'une tâche dans le système. L'ordonnanceur doit donc tester toutes les autres tâches potentielles de la liste statique (qui contient toujours le même nombre d'éléments – voir section 3.3.1) avant de remettre la même tâche sur le processeur.

- *Temps de changement de tâche*. Il s'agit d'une autre facette de la mesure d'efficacité, qui consiste à mesurer la durée de la fonction de changement de tâche `KRN_GiveCPU()`. Les lectures oscillaient entre 180 µs et 230 µs quatre fois sur cinq, avec une cinquième lecture de l'ordre de 330 µs. Sachant que l'interruption périodique se produisait aux 2 ms et le changement de tâche aux 10 ms, il est possible de retrouver[54] les taux d'efficacité mesurés dans la section précédente. Le temps de changement de contexte de µCOS n'a pas été mesuré, mais il est possible de calculer une durée de l'ordre de 130 µs de la même manière.

- *Espace mémoire*. L'espace mémoire requis pour le noyau seul est aussi un instrument de mesure intéressant. Le tableau 5.2 montre les valeurs approximatives pour ILNI et µCOS. Pour les deux noyaux, ces valeurs excluent une librairie commune qui fournit des services C standards (`printf`, `malloc`, `strcpy`, etc.). Cette librairie représente environ 30 ko de code et quelques dizaines d'octets de variables.

En ce qui concerne la taille du code, les deux noyaux sont comparables. µCOS offre certains services que ILNI n'offre pas, comme les queues de messages, mais ILNI en offre d'autres, par exemple la gestion des listes chaînées. En plus, même dans sa version pour systèmes

---

[54] (2 x 180 µs + 2 x 230 µs + 1 x 330 µs) / 10 ms = 11,5%

TABLEAU 5.2: TAILLE MÉMOIRE DES NOYAUX

	Code	Variables
ILNI	7.4 – 18.4 ko	~41 ko
µCOS	7.4 ko	5 – 41 ko

uniprocesseurs, ILNI conserve un surplus de code provenant de la version multiprocesseur. La taille du code de ILNI s'accroît sensiblement selon les portions de code multiprocesseur conservées et la quantité de tests de validation (défini par le niveau de DEBUG). Conserver la taille du code dans des limites raisonnables a d'autres avantages que simplement diminuer le besoin en mémoire des cartes. Par exemple, l'ensemble du projet *BigBrother* peut encore être compilé avec la version de démonstration du compilateur de SDS, qui est gratuit.

Pour la taille des variables, l'écart entre ILNI et µCOS découle de la technique de réservation de l'espace de pile des processus. ILNI crée ces piles au démarrage du système, en réservant à l'avance des piles de même taille pour l'ensemble des tâches du système (voir section 3.3.1). µCOS demande qu'on lui fournisse en paramètre (dans la fonction de création de tâche) un pointeur de pile pour la tâche. L'allocation peut donc être faite au moment opportun, avec des tailles de pile adaptées au besoin. Par contre, cela a comme conséquence qu'il est plus difficile de détecter et d'implanter un algorithme de détection de débordement de pile, le 68331 n'ayant pas le matériel requis [55] pour implanter des algorithmes complets de protection de mémoire. ILNI contient un détecteur de débordement de pile rudimentaire, mais qui a démontré plus d'une fois son utilité lors de nos tests. Il est à noter que tant pour ILNI que pour µCOS, si on exclut les piles, la taille des variables systèmes passe sous les 1 ko, valeur quasi négligeable devant les piles des tâches qui atteignent souvent 2 ko chacune.

---

[55] Chez Motorola, ce matériel se nomme *Memory Management Unit*, ou *MMU*.

5.2 Performances du bus de communication interprocesseur

Voyons maintenant les performances du matériel et du logiciel de l'interface de communication EME:

- *Débit théorique du port de communication*. Lors de la conception, un débit de 500 kBaud était visé. Selon les fiches techniques, ce débit est facile à atteindre sur un port SPI mais en pratique, cela semble irréaliste. Les expériences de Siemens [13] comme les nôtres démontrent que le débit maximal diminue rapidement à mesure que la longueur des lignes de communication augmente et que pour un bus de trois cartes ou plus, le maximum possible est de l'ordre de 115 kBaud. Aussi, le débit maximal théorique du 68331 (4 MBaud) est prévu pour une utilisation intermittente, pour l'accès à un périphérique. Dans le cas d'un bus à jeton, la communication est permanente. En conséquence, le taux d'utilisation du processeur s'élève

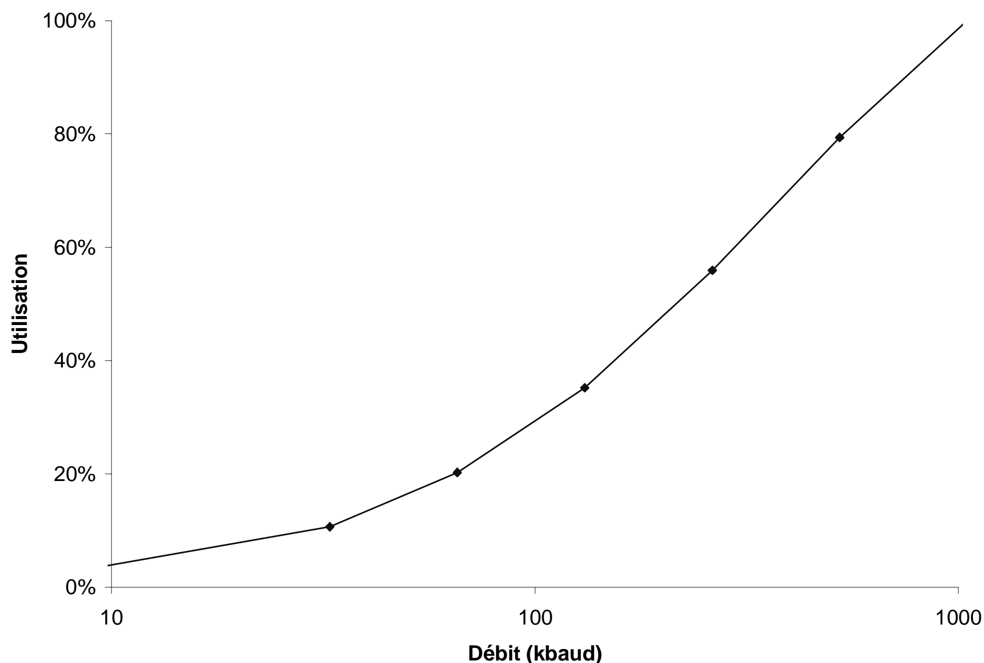


Figure 5.1: Utilisation du processeur en fonction du débit SPI

très rapidement. La figure 5.1 montre le résultat d'expériences menées avec deux cartes sur le bus EME. On y constate que à 1 MBaud, le processeur frôle les 100% d'utilisation.

Tenant compte de ces deux facteurs, EME est basé sur un bus à 65 kBaud. Cela place le taux d'utilisation du processeur à environ 20% sur chaque carte, juste pour la gestion du port de communication de EME. Lorsqu'on répète le test d'efficacité globale de la section 5.1 en ajoutant la charge de gérer le port de communication mais sans répartir la tâche de calcul entre les différentes cartes, on obtient un temps d'exécution de 52,8 s, soit une surcharge totale de 29%. Ce temps demeure pratiquement identique, qu'il soit effectué avec deux ou trois cartes sur le bus EME, ce qui s'explique par le fait que dans les deux cas, le bus travaille à temps plein pour faire circuler le jeton. Par contre, si on fait le même test mais cette fois en répartissant l'opération de calcul sur les trois cartes, le temps requis n'est plus que de 17,6 s, un peu moins de la moitié du temps de référence. Cela démontre un gain net réel dans la puissance de calcul disponible.

• *Nombre maximal de cartes*. En ce qui concerne le nombre maximal de cartes sur un seul bus, nos tests n'ont pas dépassé trois cartes. Les expériences de Siemens montrent que leur approche fonctionne à 115 kBaud avec 16 cartes à base de processeurs C167. Il s'agit ici d'une question d'électronique pure et la réponse dépend principalement de la sensibilité au bruit. Ce sera un facteur à considérer lors du port de EME sur d'autres microcontrôleurs: le plus sensible au bruit sera le maillon faible et le facteur limitatif du système.

• *Délais de communication*. Une autre donnée intéressante à connaître est le temps requis pour transmettre un message d'une carte à une autre. Les conditions du test étaient les suivantes:

- bus EME avec deux cartes présentes;
- toutes les tâches EME sont en service, mais toutes celles propres à *BigBrother* sont absentes;

- une seule tâche utilisateur sur chacune des cartes, dont le pseudo-code est le suivant:

```
while(1) {
      RéceptionMessage(bloquant);
      Détruire(message);
      EnvoiMessage(réponse);
};
```

- les tâches sur les deux cartes sont identiques, à l'exception de l'initialisation qui diffère de manière à ce qu'une des deux cartes fabrique le message initial.

L'échange de 1600 messages (800 dans chaque sens) nécessite 23 s, soit 14 ms par message. Le même test effectué en envoyant les messages à une autre tâche sur la carte locale donne un délai de 1,5 ms. Il s'agit de données intéressantes, mais difficilement comparable à quoi que ce soit, car elles dépendent de trop de facteurs comme:

- la charge du processeur sur chacune des cartes;
- le modèle des processeurs;
- le type et la vitesse du port de communication;
- l'algorithme de communication (dans le cas présent, les délais tiennent compte du temps requis pour transmettre le jeton à l'autre carte – ce sont des messages supplémentaires).

Bien que la valeur de 14 ms soit un minimum et puisse s'accroître notablement selon les tâches présentes, elle se compare favorablement aux communications inter-cartes du robot Pioneer I, qui sont normalement à intervalles fixes (100 ms) mais qui, en pratique, prennent du retard quand la charge du robot devient trop lourde.

• *Débit réel du port de communication*. Les résultats du test précédent fournissent aussi les données nécessaires pour calculer le débit pratique du port de communication. Sachant que transmettre un message de 32 octets prend 14 ms, le débit réel du port pour ce test est donc de 18 kBaud, soit une efficacité de 28%. Ce très faible taux d'efficacité a deux causes principales. D'abord, les principes de fonctionnement du bus font qu'entre chaque message, un jeton est mis en circulation. Ce jeton occupe lui aussi 32 octets, ce qui fait que même si seulement 32

octets de charge utile circulent par 14 ms, il circule de fait 64 octets sur le bus. La seconde cause est aussi reliée au jeton. Lorsqu'une carte reçoit le jeton, elle a un certain nombre d'opérations à effectuer pour devenir maître du bus de communication, opérations qui demandent un certain temps. À ce temps s'ajoute le délai requis pour que la tâche responsable de ces opérations obtienne le processeur. L'ensemble de ces facteurs fait que cette efficacité de 28% variera selon:

- la taille des données à envoyer (si le message ne contient qu'un octet utile, l'efficacité réelle diminue par un facteur de 32);
- la charge du processeur (qui influence les temps d'attente);
- le nombre de messages envoyés entre la réception et la réémission du jeton (diminution du poids relatif du jeton et des opérations de changement de maître de bus).

• *Bruit sur le bus et erreurs de communication*. L'ensemble des tests menés sur le bus EME a montré qu'il arrive occasionnellement qu'il y ait de la corruption de données pendant la

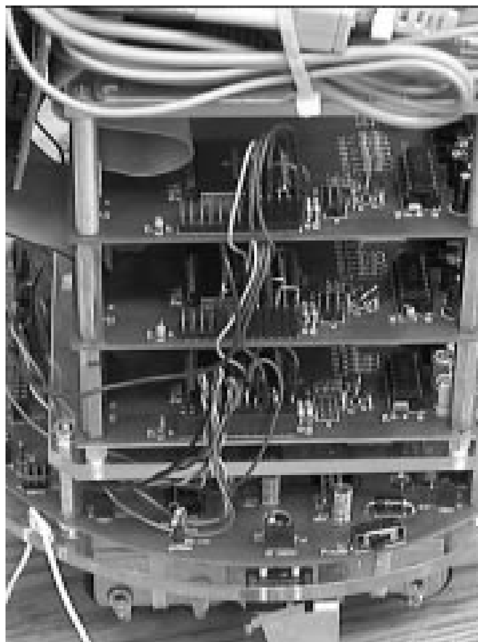


Figure 5.2: Filage du bus EME sur *BigBrother*

communication. Les données compilées sur environ deux heures d'expérimentation sur *BigBrother* rapportent un taux de corruption d'environ un bit par 10^8. La cause présumée de cette corruption est le filage qui relie les différentes cartes, illustré à la figure 5.2. Au niveau du matériel, il existe plusieurs techniques pour améliorer la fiabilité des transmissions: la conception de connecteurs spécialement adaptés, l'utilisation de tensions plus appropriées à des lignes de transmission (-12V/+12V au lieu de l'actuel 0V/5V), ou encore utiliser des lignes différentielles. Toutefois, pour une fiabilité encore plus grande, la meilleure solution serait d'implanter une procédure logicielle de retransmission des paquets corrompus. Le taux d'erreur est suffisamment faible pour que la détection puisse être faite par des méthodes simples, comme l'ajout de bits de parité.

5.3 Performances du robot *BigBrother*

Tout comme pour l'environnement EME, il y a de nombreuses mesures qualitatives et quantitatives qui peuvent être prises sur l'application de validation. Voici les performances de *BigBrother*:

- *Consommation électrique*. La première analyse porte sur la consommation électrique du robot, donnée au tableau 5.3. *BigBrother*, avec ses 8 batteries "C" (tension nominale totale de 11 V), a une autonomie de plus de 30 minutes et, selon les conditions, peut atteindre une heure

TABLEAU 5.3: CONSOMMATION ÉLECTRIQUE

Base ROBUS (1)	120 mA
Moteurs, en régime permanent (2)	160 mA
Carte Kit331 (1)	90 mA
Caméra et circuit d'interface (1)	60 mA
BigBrother entier, toutes les composantes en opération en régime permanent	680 mA

ou plus. Comme le robot n'est pas équipé d'instruments lui permettant de mesurer sa tension d'alimentation, il n'a donc aucun moyen de planifier en conséquence. La manifestation la plus commune des batteries qui faiblissent est une erreur logicielle sur une des cartes Kit331.

L'information la plus importante du tableau 5.3 est que le coût en énergie d'une carte processeur supplémentaire est de moins de 100 mA, soit le même ordre de grandeur qu'un des moteurs. Ce coût pourrait aussi être amélioré en utilisant des cartes conçues à cette fin: l'utilisation d'un seul régulateur de tension pour l'ensemble des cartes processeurs serait certainement une des premières améliorations à apporter. Avec le matériel actuel, un système composé d'une base ROBUS et d'un microcontrôleur 68331 consomme 370 mA. Ajouter un second 68331 fait passer la consommation maximale à 460 mA, une augmentation de 25%. C'est tout à fait envisageable, même sur des systèmes destinés à être commercialisés.

Il est tentant mais difficile de comparer la consommation de courant avec d'autres robots, car pratiquement chaque robot a des capacités différentes. Le seul robot pour lequel des données sont disponibles est un Pioneer 2, beaucoup plus gros que *BigBrother*. Il consomme 680 mA (tension de 13 V) pour la base seule et plus de 2,5 A lorsque l'ordinateur embarqué (Pentium 233MHz PC-104) est en service. À cela, il faut ajouter la consommation des moteurs, qui n'a pas été mesurée.

• *Caméras et traitement des images*. Les circuits utilisés pour raccorder les caméras aux cartes 68331 (basés sur des puces MC6821) sont finalement le facteur le plus limitatif de l'application. En effet, même en basse résolution, il est impossible de transférer plus de 5 images/s. Lorsque les cartes doivent en plus faire le traitement des images, ce nombre passe à entre 3 et 4 images/s, malgré que le transfert et le traitement s'effectuent en parallèle, dans des tâches séparées.

Pour le suivi de ligne effectué par la caméra 1, ce faible débit est néanmoins suffisant. Les situations où le robot perd la ligne peuvent presque toujours être associées à des conditions environnementales. Les plus fréquentes sont:

- un objet foncé (obstacle ou tache au sol) qui perturbe le calcul de centre de masse;
- un fort éclairage ambiant qui crée des ombres (extrémité droite de la figure 5.3);
- un ruban de traçage trop lisse et trop réfléchissant, qui devient invisible sous un éclairage intense (centre de la figure 5.3).

Un facteur aggravant pour le suivi de ligne est la position de la caméra sur le robot. Elle est située très haut, à près de 30 cm du sol, ce qui lui donne un champ de vision plus grand que nécessaire et donc une sensibilité à des obstacles assez éloignés.

Pour la communication par signalisation lumineuse, le principal problème demeure le faible débit de la caméra qui allonge inutilement le temps de transmission de chaque message.

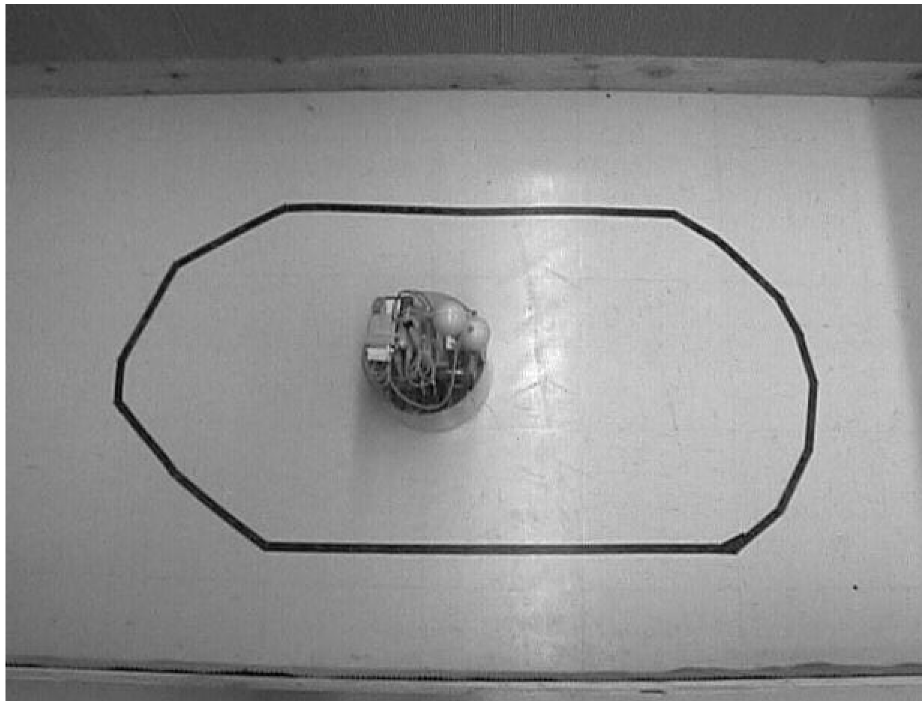


Figure 5.3: Piste d'expérimentation

L'algorithme actuel requiert la lecture de deux images consécutives pour identifier un état (lumière / pas de lumière). Chaque bit de la transmission est constitué de deux états et un message entier comporte 11 bits, dont 8 pour le champ de données. Donc dans le cas minimum, il faut 44 images pour un seul message. À 3,5 images/s, cela signifie que chaque message nécessite plus de 12 s de temps de transmission. Une solution de contournement serait de réduire le vocabulaire de la communication lumineuse. Actuellement, seulement 9 des 256 messages possibles ont une signification. Il serait donc très facile de réduire à 4 bits le champ de données, ce qui ferait passer le délai minimum à 8 s.

Quant à la réception des données, le taux de succès est de l'ordre de 90%. Il arrive à l'occasion que le récepteur se synchronise mal sur le premier bit de la communication, ce qui cause la réception d'une donnée invalide dans certain cas mais le plus souvent, cette situation est détectée par un niveau incorrect sur l'un ou l'autre des deux bits d'arrêt. Toute disparition de l'émetteur durant plus de trois fois la durée "émetteur allumé" ou "émetteur éteint" (soit 1,5 bit) est effectivement détectée et rapportée comme une erreur de communication.

- *Contrôle du robot*. Même si le bus EME final dispose d'un débit moindre qu'initialement escompté, soit 65 kbps au lieu de 500 kbps (voir section 5.2), cela n'a pas eu d'impact au niveau de l'application finale. En effet, le traitement des images est effectué localement et le bus ne sert qu'à relayer les résultats des analyses et les informations de contrôle. La mesure du temps de circulation des messages a été refaite sur *BigBrother*. Le délai est passé de 14 ms (à vide) à 36 ms (pleine charge). C'est fort raisonnable, quand on tient compte que la tâche chargée d'émettre le jeton n'est pas prioritaire par rapport aux autres tâches du système. Cela demeure également très inférieur au temps du Pioneer I.

La figure 5.3 illustre la piste sur laquelle *BigBrother* a été testé. La procédure de test consistait à faire faire cinq tours de piste au robot (un trajet de trois minutes), parfois dans un sens,

parfois dans l'autre. Une quinzaine d'expériences de ce type ont été menées. La première dizaine a permis d'identifier et de régler les problèmes suivants:

- l'extrémité droite de la piste est le point le plus sensible, en raison de la présence d'un plus grand éclairage et donc de plus de réflexions provenant du plancher blanc. Le tracé a été partiellement refait au moyen d'un marqueur moins réfléchissant pour régler un problème de perte de ligne chronique à cet endroit;
- le mur situé à l'extrémité droite de la piste a été repoussé légèrement vers l'extérieur, pour limiter les ombres à proximité du tracé;
- le mur du bas (figure 5.3) a été reculé légèrement, car lorsque le robot tournait en sens horaire, les capteurs infrarouges percevaient le mur avant même d'atteindre le milieu du virage (extrémité droite), ce qui faisait dévier le robot vers l'extérieur. Les capteurs infrarouges ont une portée variable en fonction de la couleur et de la texture des objets. Dans le cas des murs, ils sont perçus lorsque distants de moins de 50 cm du robot.

Au cours des cinq dernières expérimentations, une seule perte de ligne s'est produite, causée par une erreur logicielle (`assert(false)`), elle-même causée par une corruption de donnée dans la communication interprocesseur.

5.4 Sommaire

L'ensemble des résultats précédents nous permet de conclure que l'application *BigBrother* tout comme l'environnement EME sont fonctionnels. Il apparaît également que l'application tolère bien les petits délais imposés par les échanges de messages entre les différentes cartes. Il faut toutefois noter que ces délais ne sont guère supérieurs à ceux qui seraient imposés par un noyau temps réel chargé de quelques centaines ou de quelques milliers de tâches. La perte imposée par le délai de communication est en quelque sorte compensée par le véritable parallélisme que fournissent les multiples processeurs.

CONCLUSION

Dans ce projet de maîtrise, nous avons conçu et développé un environnement de développement matériel et logiciel multiprocesseur pour systèmes intelligents. Le principal objectif poursuivi est de fournir des outils permettant une meilleure réutilisabilité du code ainsi qu'une bonne expansibilité des plates-formes robotiques.

L'environnement, du nom de EME comporte deux modules principaux. Le premier est un noyau temps réel préemptif simple, baptisé ILNI. Le second est une interface de communication entre cartes à microcontrôleurs, basé sur le port de communication SPI. Cet environnement a été utilisé pour réaliser un robot équipé de plusieurs capteurs, dont deux à haut débit: des caméras QuickCam. Le robot devait se déplacer dans l'environnement, en fonction de l'état de ce dernier tel que perçu par une des caméras et des consignes envoyées à l'autre caméra au moyen d'un émetteur lumineux. Une telle application aurait été difficile à concevoir sans EME, la quantité de données à traiter dépassant les capacités d'un seul microcontrôleur. Les résultats démontrent que tant EME que l'application de validation fonctionnent conformément aux attentes et aux objectifs. Selon les mesures prises sur cette dernière, l'aspect multiprocesseur a comme principale conséquence l'ajout d'un délai de transmission (36 ms) lorsque l'information doit voyager d'une carte à l'autre, mais ce délai est largement compensé par la puissance de calcul accrue (un peu plus que doublée) que fournit EME.

Cette recherche a permis de démontrer que le parallélisme des microcontrôleurs est un outil de travail pratique et flexible pour la conception de systèmes intelligents. Les cartes #1 et #2 du robot *BigBrother* pourraient facilement être isolées du reste du robot et réutilisées dans une autre application. De la même manière, il serait possible et facile de créer ainsi des sous-robots pouvant être développés de façon indépendante. Comme l'interface électrique entre les cartes

ne requiert que quatre fils, la taille du connecteur d'interface ne risque pas de devenir une contrainte.

Par contre, il faut mentionner qu'en terme de puissance de calcul, une telle approche aurait beaucoup de difficulté à concurrencer une plate-forme comme PC-104. Si on utilise la même mesure d'efficacité globale que celle utilisée à la section 5.1, un ordinateur PC-104 typique (Pentium 233 MMX) donnerait un résultat brut de moins de 2 s, soit une amélioration par un facteur de presque 20 à 1. Cependant, le robot *BigBrother* entier coûte environ $825 en pièces, soit moins que juste l'ordinateur PC-104, sans base robot associée. Chaque carte processeur de *BigBrother* vaut environ $100. Il s'agit donc de deux approches complémentaires.

En comparant les performances de *BigBrother* et celles du robot hexapode Hannibal [11] basé sur dix microcontrôleurs de puissance similaire à ceux de notre robot, il est possible d'établir plusieurs comparaisons intéressantes:

- D'abord, Hannibal a un avantage au niveau de la communication. L'utilisation d'un protocole de bus à jeton dans EME a finalement un poids très lourd car le bus est perpétuellement en communication. Ceci occasionne un coût qui peut être estimé à 20% du temps CPU de chaque carte, sans compter la réduction de la fréquence d'opération du port SPI requise pour conserver le taux d'utilisation de chaque processeur à un niveau raisonnable. Hannibal utilise pour sa part une approche similaire à Ethernet, soit un protocole qui gère les collisions au lieu de les interdire. Le port SPI permet une telle approche, mais cela n'a pas été tenté dans la présente recherche. Sa mise en œuvre apporterait certainement de nouvelles données intéressantes.

- Ensuite, Hannibal, bien que comportant dix processeurs, n'en utilise qu'un pour effectuer des traitements. Les neuf autres ne sont utilisés que comme circuits d'interface pour les capteurs et les actionneurs. Cela en fait un système multiprocesseur où la tâche n'est pas distribuée. *BigBrother*, pour sa part, répartit la tâche de plusieurs façons:

- chaque carte est à la fois processeur et circuit d'interface pour capteur et actionneur;

- il n'y a pas de carte maître, ce qui assure une meilleure tolérance aux fautes et une meilleure capacité de redondance;
- l'algorithme de contrôle peut aussi être distribué lorsque l'application le justifie, par exemple en implantant le logiciel sur la carte physiquement reliée à l'actionneur à contrôler lorsqu'il y en a plusieurs, ou encore en répartissant un algorithme complexe sur plusieurs cartes.

• Également, Hannibal utilise un système d'exploitation conçu spécialement pour les approches comportementales (*Brooks's Behavior Language Operating System* [6]). Par opposition, *BigBrother* est basé sur EME, qui est fondamentalement un noyau temps réel sur lequel a été implantée une architecture de prise de décision basée sur les comportements. Cela permet une plus grande flexibilité dans le choix des algorithmes de contrôle.

Enfin, bien que EME soit fonctionnel, il est encore un environnement en cours de développement. Pour EME en tant que tel, voici les améliorations qui seraient les plus utiles:

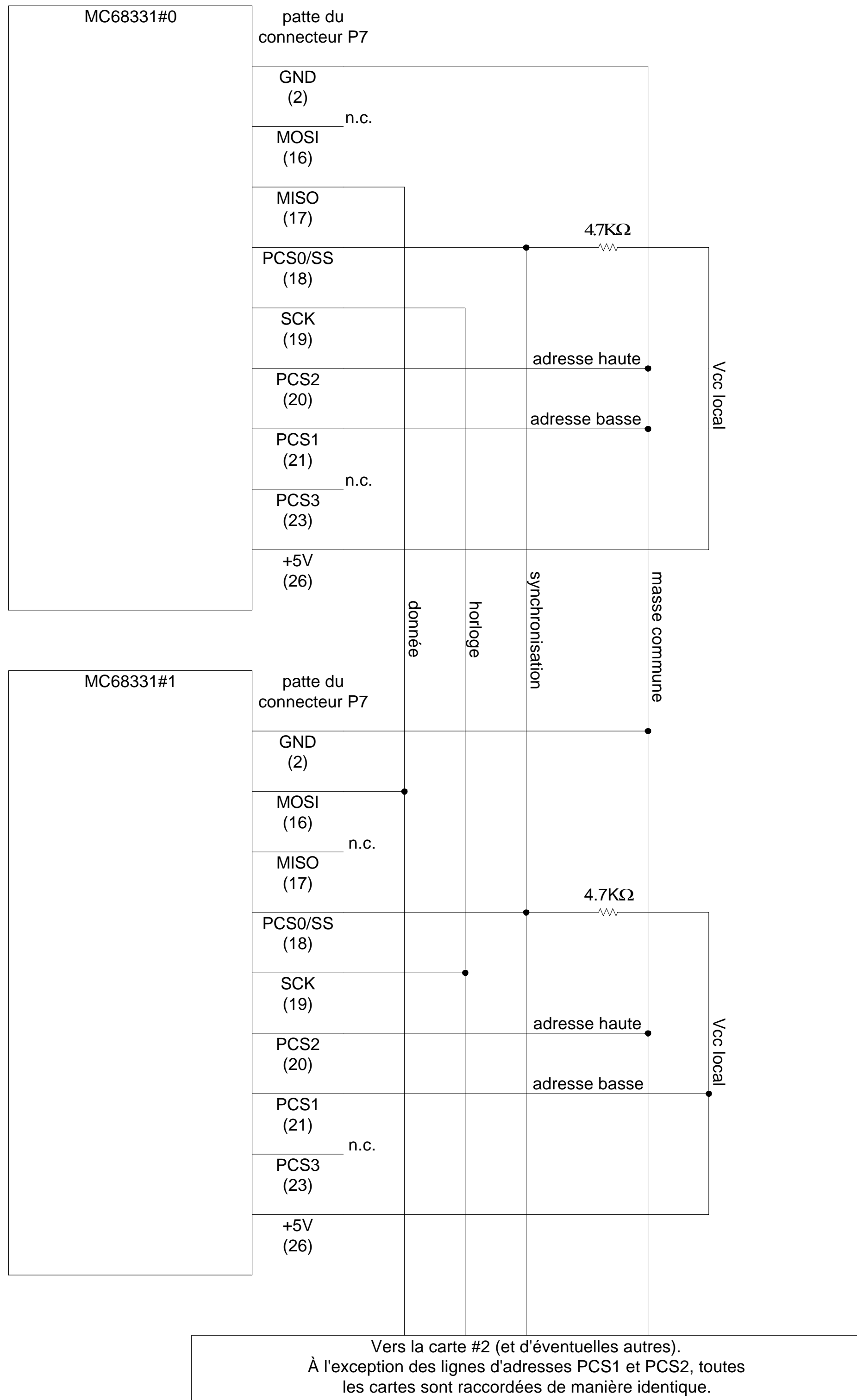
- compléter les services DNS;
- porter sur un compilateur gratuit, comme GCC;
- porter sur d'autres microcontrôleurs (ceux listés en 2.1 ou d'autres encore) et d'autres cartes;
- ajouter des services au noyau (voir [15, 16] ou les noyaux commerciaux), comme par exemple des services de communication par tube ou encore une interface *telnet* permettant de contrôler l'état du système et de lancer des tâches;
- implanter l'interface de communication sur d'autres média, comme par exemple Ethernet;
- fournir des outils pour mesurer l'usage des composantes, comme le taux d'utilisation du processeur, et ainsi valider le besoin ou non de l'ajout d'une nouvelle carte;
- modifier les techniques de détection de carte pour permettre la détection pendant l'opération du système, ce qui permettrait de réintégrer une carte qui a cessé de fonctionner un certain temps ou même à la limite d'ajouter et d'enlever des cartes pendant l'opération du système.

Pour l'application *BigBrother*, on peut aussi songer à plusieurs améliorations possibles:

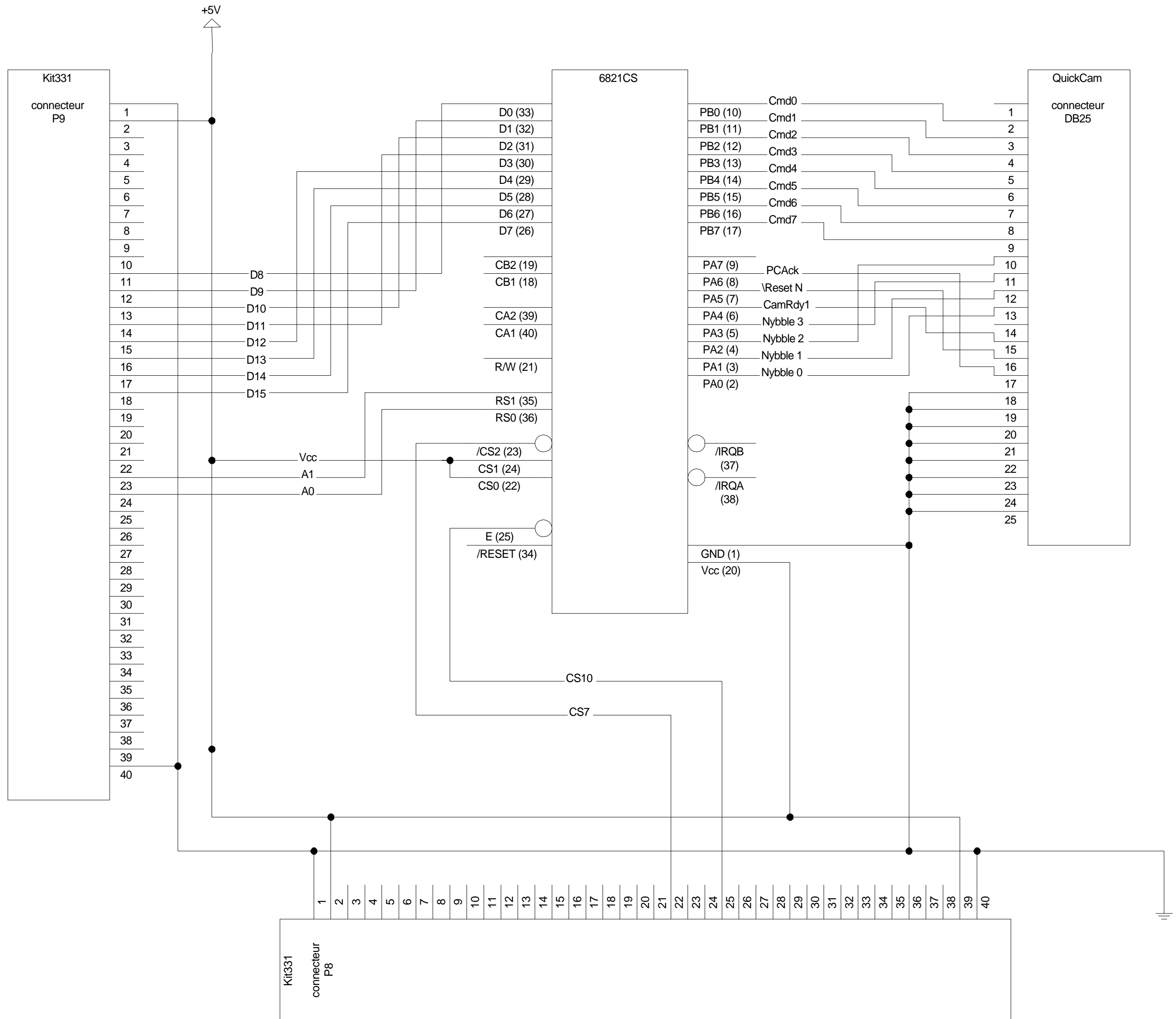
- régler le problème de débit des caméras, soit par un moyen direct (refaire la conception du circuit d'interface), soit par un moyen indirect (travailler sur des sections de l'image au lieu de l'image entière);
- réorienter le robot vers l'émetteur lumineux, ou encore utiliser une caméra motorisée, ce qui permettrait de ne pas avoir à immobiliser le robot pendant le transfert;
- utiliser un algorithme plus évolué que le centre de masse pour le suivi de la ligne au sol, comme par exemple [8] et travailler sur des portions de l'image au besoin;
- réaliser une application où une carte qui détecte qu'une autre ne fonctionne plus correctement est physiquement capable de la réinitialiser par un *reset*.

Somme toute, EME démontre que l'usage de systèmes multiprocesseurs peut être une aide précieuse au développement de systèmes intelligents, et que l'utilisation d'un système standardisé pré-développé permet une mise en route plus rapide et une meilleure réutilisation des composantes matérielles et logicielles que la technique classique, qui consiste à relier des microcontrôleurs selon une approche développée spécifiquement pour l'application. À court terme, nous espérons que EME sera utilisé dans d'autres applications, comme par exemple un robot marcheur ou encore une carte d'interface audio. On peut très bien imaginer une quatrième carte sur *BigBrother*, qui serait en charge d'un micro et d'un système de commande vocale.

ANNEXE 1: SCHÉMA ÉLECTRIQUE DE L'INTERFACE DE COMMUNICATION



ANNEXE 2: SCHÉMA ÉLECTRIQUE DE L'INTERFACE QUICKCAM



BIBLIOGRAPHIE

[1] Angle, C., "Design of an artificial creature", Mémoire de maîtrise, Department of Electrical Engineering and Computer Science, MIT, 1991, 121 p.

[2] Arkin, R.C., *Behavior-Based Robotics*, The MIT Press, 1998.

[3] Bekey, G.A., "Needs for robotics in emerging applications: a research agenda", *IEEE Robotics & Automation Magazine*, décembre 1997, p. 12-14.

[4] Birk, A., Kenn, H. et Walle, T., "RoboCube, an universal special-purpose hardware for the RoboCup small robots league", *4th International Symposium on Distributed Autonomous Robotic Systems*, 1998.

[5] Brooks, R.A., "A robust layered control system for a mobile robot", *IEEE Journal of Robotics and Automation*, vol. 2 no. 1, mars 1986, p. 14-23.

[6] Brooks, R.A., "The behavior language; user's guide", Memo 1227, MIT Artificial Intelligence Lab, avril 1990, 35 p.

[7] Brooks, R.A., *L*, Manuel utilisateur, IS Robotics, janvier 1996, 59 p.

[8] Chen, N., "A vision-guided autonomous vehicle: an alternative micromouse competition", *IEEE Transactions on Education*, 40(4), 1997.

[9] Deadkov, A.F. et Eadline, D.J., "Performance consideration for I/O-dominant applications on parallel computers", dans *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '95)*, H. R. Arabnia, ed., Athens GA, 1995.

[10] Drolet, L., "Projet de vision pour le robot mobile Daedalus", Rapport de projet de spécialité, Département de génie électrique et de génie informatique, Université de Sherbrooke, mai 1998, 17 p.

[11] Ferrell, C., "Robust agent control of an autonomous robot with many sensors and actuators", Mémoire de maîtrise, Technical report 1443, MIT Artificial Intelligence Lab, 1993, 163 p.

[12] Fujita, M., Kitano, H. et Kageyama, K., "Reconfigurable physical agents", dans *Proceedings of Second International Conference on Autonomous Agents*, mai 1998, p. 54-61.

[13] Hettman, A., "Using the SSC (SPI) interface in a multimaster system", Siemens ApNote #1632, 1996, 40 p.

[14] K-Team, *Kameleon 376 SBC User Manual*, version 1.0, Lausanne, mai 1999.

[15] Labrosse, J.J., *µCOS: The Real-Time Kernel*, 3e édition, R&D Publications, 1992, 266 p.

[16] Labrosse, J.J., *Embedded Systems Building Blocks: Complete and Ready-to-Use Modules in C*, R&D Publications, 1995, 616 p.

[17] Madany, P.W., "JavaOS™: A Standalone Java™ Environment", document de référence disponible à ftp://ftp.javasoft.com/docs/papers/JavaOS.cover.ps, mai 1996, 17 p.

[18] Michaud, F., "Nouvelle architecture unifiée de contrôle intelligent par sélection intentionnelle de comportements", Thèse de doctorat, Département de génie électrique et de génie informatique, Université de Sherbrooke, juin 1996.

[19] Michaud, F., Lucas, M., Lachiver G., Clavet A., Dirand J.-M., Boutin N., Mabilleau, P., et Descoteaux, J., "Using ROBUS in Electrical and Computer Engineering education", dans *Proceedings ASEE*, 1999.

[20] Michaud, F., Vu, M.T., "Managing robot autonomy and interactivity using motives and visual communication", dans *Proceedings Autonomous Agents*, 1999, p. 160-167.

[21] Motorola, *MC68331 User's Manual*, document MC68331UM/AD, révision 1, avril 1994.

[22] Motorola, *Queued Serial Module Reference Manual*, document QSMRM/AD, révision 1, juin 1997.

[23] Philips Semiconductors, "Using the87LPC76X in multi-master I2C applications", janvier 2000, 30 p.

[24] Proctor, F.M. et Albus, J.A., "Open-architecture controllers", *IEEE Spectrum*, juin 1997, p. 60-64.

[25] QNX Software Systems Ltd, *QNX Operating System - System Architecture - for QNX 4.24*, 2e édition, octobre 1997, 177 p.

[26] Ricard, B., Gosselin, C.M., Le-Huy, H. et Poussart, D., "On the development of a high-performance robot controller with an open architecture for research applications", *Laboratory Robotics and Automation*, vol. 6, no 6, 1994, p. 273-282.

[27] Rossol, L. et Demoe, B., "Controlling robot costs with open architecture", *Robotics World*, vol. 13 no. 1, 1995, p. 30-31.

[28] Sakamura, K., *µITRON 3.0 - An Open and Portable Real-Time Operating System for Embedded Systems - Concept and Specification*, IEEE Computer Society Press, 1998, 233 p.

[29] Silberschatz, A. et Galvin, P.B., *Operating System Concepts*, 4e édition, Addison-Wesley, 1994, 780 p.

[30] Stewart, D.B., Volpe R.A. et Khosla P.K., "Design of dynamically reconfigurable real-time software using port-based objects", Technical report CMU-RI-TR-93-11, Carnegie Mellon University, juin 1993, 24 p.

[31] Stewart D.B., Schmitz, D.E. et Khosla P.K., "The Chimera II real-time operating system for advanced sensor-based robotic applications", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no 6, novembre-décembre 1992, p. 1282-1295.

[32] Walrand, J., *Communication Networks: A First Course*, Aksen Associates, 1991, 460 p.