Installation de capteurs et centralisation par microcontrôleur pour le robot Bipède KONDO

INRIA Rhone-Alpes - Service Support des Expérimentations et de Developpement logiciel (SED)
IUT Clermont-Ferrand - Licence Professionnelle Informatique Embarquée et Robotique

Maitres de stage : Pissard-Gibollet Roger, Cuniberto Jean-François
Examinateurs : KAUFMANN, LE HEN

Avril 2006 - Juillet 2006

BRUNEAUX Jérôme

Remerciements

Je remercie chaleureusement Roger Pissard-Gibollet et Jean-François Cuniberto pour leur accueil au sein de l'INRIA, leurs précieux conseils, leur gentillesse, et leur amitié.

Je remercie aussi toutes les personnes du service SED qui m'ont accueillies au sein de leur équippe : Roger Pissard-Gibollet, Jean-François Cuniberto, Claudie Marchand, Gérard Baille, Soraya Arias, Laurence Boissieux, Eric Ragusi, Hervé Mathieu et Nicolas Turro.

Je remercie toutes les personnes qui m'ont directement ou indirectement aidé par leurs conseils ou connaissances, mais aussi tout ceux qui ont contribué à rendre ce stage très intéressant, instructif et à en faire une expérience humaine.

Table des matières

Première partie

Introduction

Ce rapport présente le stage éffectué au sein de l'INRIA Rhône-Alpes dans le cadre de l'obtention de la Licence Professionnelle Informatique Embarquée et Robotique. J'ai choisi de réaliser ce stage car il me permettait de mettre en oeuvre à la fois mes compétences nouvelles acquises au cours de la licence professionnelle (notemment au niveau de la programmation des microcontroleurs) ainsi que mes compétences acquisent au cours de ma formation DUT Génie Electrique et Informatique Industrielle (au niveau électronique principalement). De plus, le monde de la robotique est très intéressant et en plein essor.

Le stage proposé par l'INRIA portait sur l'intégration de capteurs sur un robot humanoïde. Ce robot, le Kondo KHR-1, doit permettre d'illustrer certaines fonctionnalités du logiciel HuMAnS (logiciel de modélisation physique de robots). Le robot exécute des trajectoires, générées par HuMAnS, en boucle ouverte sans utiliser de capteurs. L'intégration de capteurs sur le tobot tels que capteurs de pression ou accéléromètre permettera entre autres d'utiliser le robot en boucle fermée et donc de créer des asservissements pour éviter une chute par exemple.

Afin de mieux connaitre l'INRIA, le service au sein duquel j'ai été accueilli et le robot bipède Kondo KHR-1, le chapitre 'Stage' présente l'INRIA et ses différentes unités, l'unité de recherche Rhône-Aples ainsi que le service qui m'as accueilli et pour finir, une brève introduction au robot et ses fonctionnalités.

Pour atteindre les objectifs du stage, j'ai commencé par réaliser la base de l'extension, à savoir l'intégration matérielle des capteurs sur le robot. L'intégration matérielle est présentée dans la partie 'Extension Matérielle'. Les différentes étapes de la réalisation y sont décrites, à commencer par la réalisation des semelles présentées dans la section 'Semelles'. Puis, j'ai continué par l'étude et la réalisation de la carte embarquant un microcontrôleur qui nous permettera de centraliser les données. Cette carte est présentée dans le chapitre 'Carte d'extension'. Au cours de cette étude, il a été décidé d'ajouter un capteur sur le robot. Ce nouveau capteur est décrit dans le chapitre 'Capteur CEA'.

Une fois la partie matérielle réalisée, il a fallu créer le programme qui sera embarqué sur la carte d'extension. Cette partie du stage est présentée dans la partie 'Logiciel'. La communication avec le robot étant primordiale dans ce projet, il a fallu étudier la programmation du robot. Cette étude est rapportée dans le chapitre 'API Kondo'. Le programme embarqué sur la carte d'extension est expliqué dans le chapitre 'Code embarqué'. Une interface logicielle graphique était nécessaire afin de pouvoir facilement utiliser cette carte d'extension. Cette'interface logicielle crée est présentée par le chapitre 'Interface Grpahique'. Enfin, pour finir, quelques résultats expérimentaux seront présentés, en collaboration avec un autre stagiaire. Ces résultats sont expliqués par le chapitre 'Résultats expérimentaux'.

Chapitre 1

Contexte du stage

1.1 Institut National de Recherche en Informatique et Automatique

L'INRIA (Institut National de Recherche en Informatique et Automatique) est un etablissement public à caractère scientifique et technologique qui mène des recherches avancées dans le domaine des sciences et technologies de l'information et de la communication. Ce domaine inclut l'informatique et l'automatique, mais aussi les télécommunications et le multimédia, la robotique, le traitement du signal et le calcul scientifique.

Les recherches de l'INRIA sont réparties sur cinq unités (plusieurs projets sans lieu géographique sont regroupés dans l'unité INRIA Futurs) qui sont :
- INRIA Lorraine
- INRIA Rhône-Alpes
- INRIA Rennes
- INRIA Rocquencourt
- INRIA Sophia Antipolis

1.2 Unité de recherche INRIA Rhône-Alpes

L'unité de recherche INRIA Rhône-Alpes est composé d'environ 500 personnes réparties dans 26 équipes de recherches :
- 159 chercheurs et enseignants-chercheurs
- 60 ingénieurs, techniciens, administratifs
- 154 doctorants et post-doctorants dont 33% étrangers
- 35 ingénieurs en développment
- 130 stagiaires accueillis régulièrement

L'unité de recherche est aussi impliquée dans des projets internationaux avec 8 équipes associées avec des laboratoires étrangers (USA, Canada, Brésil, Pays Bas, Suisse), 26 contrats de

recherche européens en cours et la participation à 10 réseaux de recherche européens.

Coté économique, l'INRIA Rhône-Alpes avait un budget de 6,2 Meuros en 2005 dont 3,2 Meuro en ressources contractuelles, 271 contrats actifs dont 53 contrats de recherche directs avec l'industrie et 14 start-up ont été créées depuis 1998, dont Polyspace et Kelkoo, ayant généré plus de 560 emplois.

1.3 Service SED

Le SED, service de support des expérimentations et de developpement logiciel, a une importance capitale dans le bon déroulement des recherches de l'INRIA. En effet, c'est cette équipe qui crée et développe tous les outils utiles aux expérimentations. Elle gére également la bonne utilisation de la halle robotique.

Ce service apporte le support à trois plates-formes expérimentales :
- Les plates-formes Robotique & Vision : Composée d'une Halle robotique regroupant les véhicules Cycab, les robots bipèdes et les ateliers mécanique/électronique.
- La plate-forme Réalité-Virtuelle : Plate-forme de 160m basée sur un couple de supercalculateurs graphiques SGI et composée d'une salle d'immersion plein-pied (écan cylindrique), d'un plateau de prise de vue avec fond bleu et d'une salle de manipulation.
- Les plates-formes Grappes : Composée d'une grappe d'Itanium2 (i-cluster2) et d'une grappe de PC connecté sur un mur d'images interactif (GrImage).

Ce service est en charge de mon accueil à l'INRIA, plus précisément sur la plate-forme Robotique & Vision, et met à ma disposition le robot Kondo, humanoïde fabriqué au Japon.

1.4 Robot Kondo KHR-1

Cette section présente le robot Kondo KHR-1 autour duquel le stage s'est déroulé. En effet, c'est sur ce robot que les capteurs devront être installés.

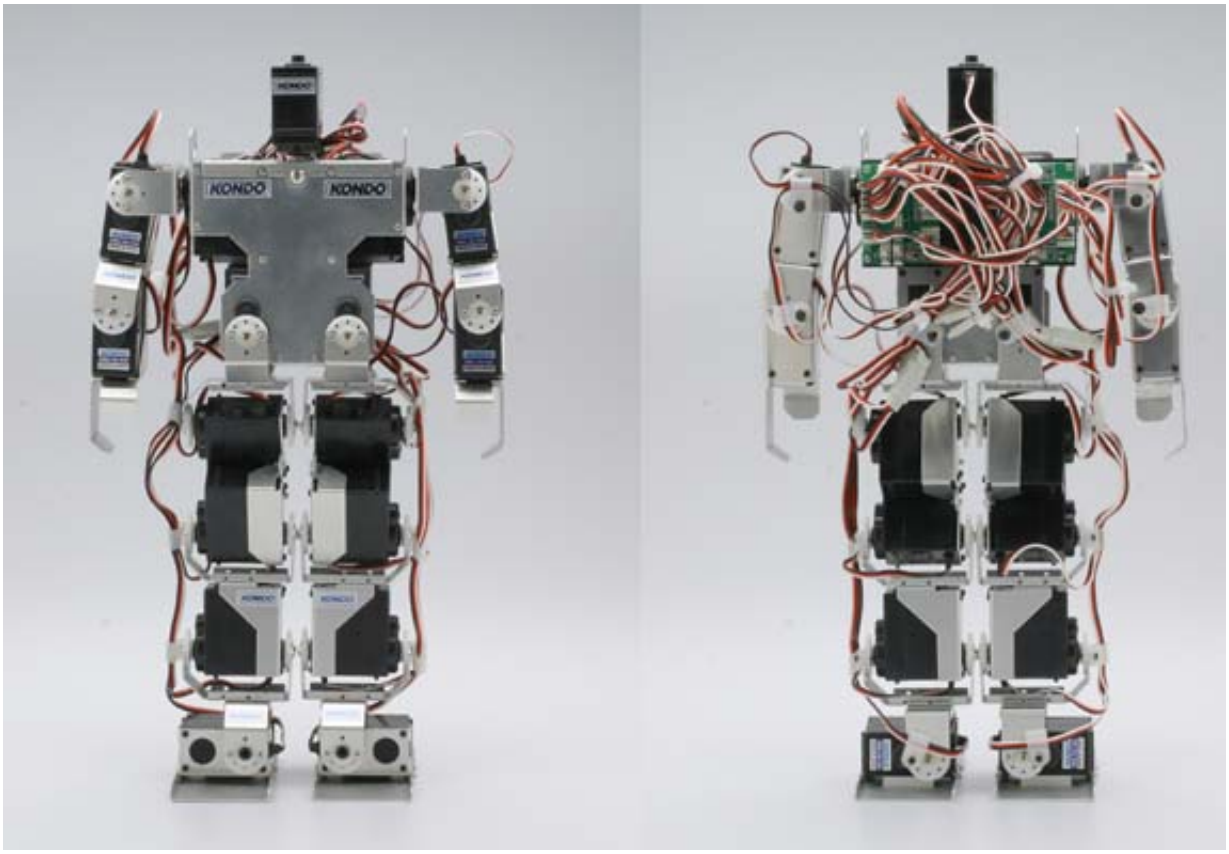


FIG. 1.1 – Robot Kondo KHR-1

1.4.1 Architecture du robot

Le robot KHR-1 repose sur 17 servo moteurs KRS-768ICS (fabriqués par la société Kondo), dont 8 par coté et 1 pour la tête. Ces moteurs sont contrôlés par les deux cartes RCB-1 installées au dos du robot. L'une d'entre elle contrôle les servos des bras et du corps. L'autre carte contrôle les servos des jambes et de la tête.

Ces cartes embarquent chacune un microcontrôleur PIC et une mémoire. Le microcontrôleur permet d'asservir les moteurs en positions, d'assurer les communications inter-RCB-1 et aussi les communications RS-232. Elles possèdent aussi une mémoire qui permet de stocker des séquences de mouvements (appelées Motions) ainsi que des scénarios (suite de séquences de mouvements).

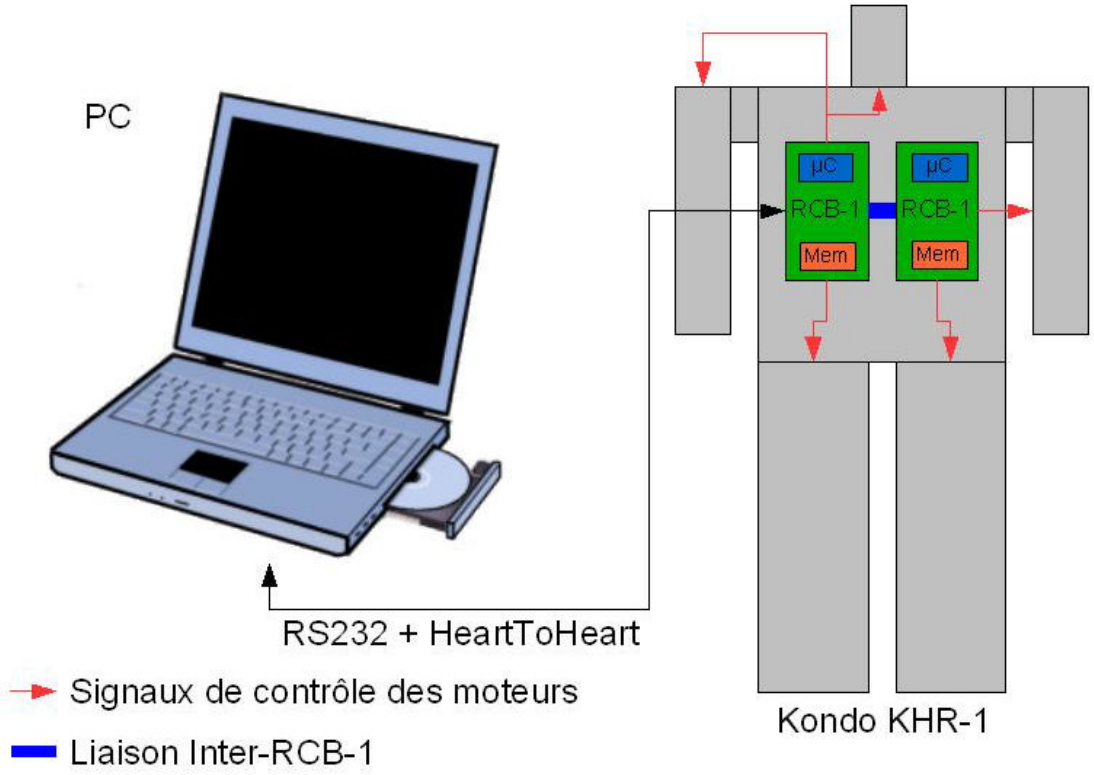


FIG. 1.2 – Architecture du Robot Kondo KHR-1

1.4.2 Principe de base du robot

Grâce au logiciel HeartToHeart fourni avec le robot, nous pouvons programmer le robot (motions et scénarios) et le régler (0 des moteurs, identifiants des cartes RCB-1, ...). Ce logiciel permet aussi de régler la position Home. Cette position de repos est la position que le robot prends automatiquement lors de sa mise en marche. Celle-ci est règlée un première fois et stockée dans la mémoire des cartes RCB-1.

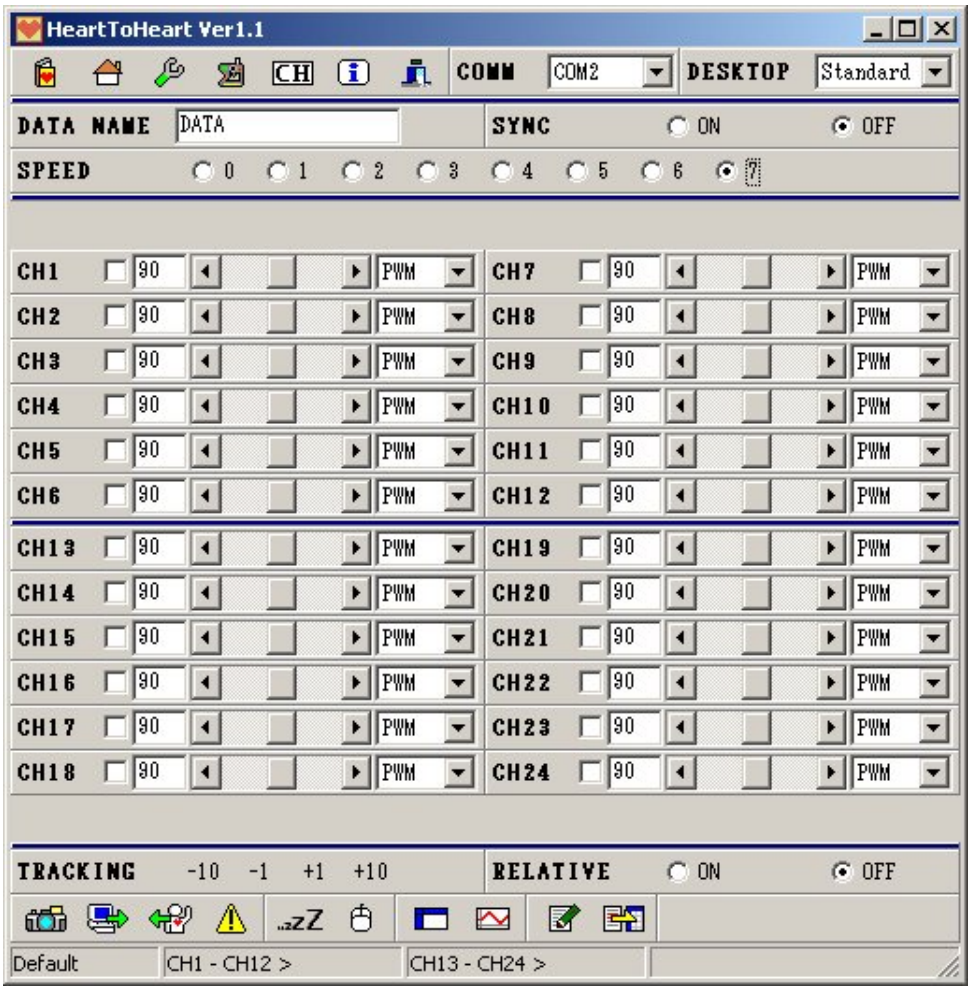


FIG. 1.3 – Fenêtre principale du logiciel HeartToHeart

Quelques exemples de mouvements sont fournis avec le logiciel. Ces mouvements (motions) sont stockés dans des fichiers CSV. Les mouvements sont en fait une suite de positions des servos qui sont lues successivement. Selon la vitesse réglée pour chaque positions, celles-ci sont atteintes plus ou moins rapidement. Cette vitesse varie entre 0 (mouvement très rapide) et 7 (mouvement lent).

La lecture du [Manuel HeartToHeart] permet d'acquérir les connaissances de bases du fonctionnement du logiciel HeartToHeart ainsi que les principes de bases du robot.

Chapitre 2

Déroulement du Stage

Dès la deuxième semaine de stage, après avoir pris contact avec le matériel et commencé l'apprentissage de la manipulation du robot, nous avons établi avec mes maitres de stage un planning prévisionnel afin de bien délimiter le travail. Ce planning permet aussi de se donner des objectifs à réaliser et permet de ne pas se lancer dans plusieurs tâches en même temps.

Tâche	***Mois, # Semaine***
Installation et prise en main du sujet	Avril, 14
Evaluation des capteurs FSR	Avril, 15-16
Semelles à base de FSR et acquisition analogique sur PC via I2C	Avril, 17
Prise en main du micro-controleur	Mai, 18
Acquisition analogique micro-controleur via I2C	Mai, 19
Communication RS232 entre PC et la carte	Mai, 20-21
Communication avec les micro-controleurs du Kondo	Juin, 22-23
Interface Logicielle et Tests	Juin, 24-25
Rédaction du rapport, Documentation, Présentation	Juillet

Chapitre 3

Compte rendu d'expérience professionnelle

Ce stage de fin d'études m'as beaucoup apporté, tant sur le point technique que sur le point humain.

Tout d'abord, il m'as permis de me conforter dans mon orientation professionnelle. En effet, ce stage ayant fait appel à mes compétences acquisent lors de ma formation GEII (Génie Electrique et Informatique Industrielle) et celles acquisent lors de ma licence professionnelle Informatique Embarquée et Robotique. Ainsi, j'ai pu constater que ces deux formations se complètent et permettent de mener à bien un projet tel que celui qui m'as été confié, de la réalisation matérielle à la réalisation logicielle associée. Par ailleurs, lors de ce stage, il m'as fallu programmer un microcontrolleur différent de ceux que j'avais eu l'occasion d'étudier, ainsi il a donc fallu que j'apprenne à manier les outils de développement associés, le matériel, ...Cela m'as permis de constater que je pouvais adapter techniquement mes connaissances à un matériel différent. Aussi, ce stage m'as permi de tester mes aptitudes à m'adapter à un environnement de travail professionnel mais aussi les limites de mon autonomie dans ce domaine professionnel.

Pour finir, ce stage m'as permis de constater que l'on pouvais allier de bonnes conditions de travail (ambiance détendue au sein de l'équipe, temps de travail aménageables, ...) avec une bonne productivité, voire même que ces conditions de travail l'améliore en donnant une meilleure motivation aux membres de l'équipe.

Deuxième partie

Extension matérielle

Cette partie a pour but de présenter les différentes éléments qui constituent l'extension matérielle, ainsi que les différentes étapes qui ont menées à la réalisation de ces éléments. Pour commencer, nous allons voir la réalisation de semelles equipées de capteurs de pression.

Chapitre 4

Semelles

Les principaux capteurs qu'il a fallu intégrer sur le robot sont des capteurs permettant de connaitre le point d'appui sur les pieds du robot. Pour ce faire, il a fallu créer des semelles intégrant des capteurs de pression.

La partie mécanique des semelles (parties métalliques supérieure et inférieure) a été réalisée par Jean-Francçois Cuniberto, mon maitre de stage. Les capteurs de pression à intégrer sont des capteurs de type FSR (*force sensing resistor*, de l'anglais).

4.1 Capteurs FSR

4.1.1 Principe de fonctionnement

Les capteurs de pression FSR sont des capteurs dont la résistance varie en fonction de la pression exercée sur ceux-ci. Ces capteurs ont une plage de mesure de 10g à 10Kg selon le branchement et l'actionneur. La résistance varie entre 2MΩet 1kΩ(10Kg). Leur durée de vie est d'environ 1,000,000 d'actionnements.

FIG. 4.1 – Différents capteurs FSR

Sur ces capteurs, la surface active est constituée par la zone striée (circulaire ou rectangulaire et de taille variable). La taille de cette zone et sa forme n'influencent pas la variation de résistance. Les différentes formes permettent d'adapter la surface active au montage mécanique.

Pour récupérer les informations des capteurs, nous pouvons soit travailler en courant (I) ou en tension (V). Dans le cas de notre application, le travail en tension à été choisi car les informations des capteurs doivent être transmisent par I2C, or le composant utilisé pour cette communication est un convertisseur Analogique/Numérique.

4.1.2 Tests

Avant d'utiliser les capteurs, j'ai du tester leur réponse sur la plage de mesure qui nous concerne, à savoir 0g jusqu'au poids du robot, soit 1.2kg environ. J'ai donc utilisé une plateforme qui actionne notre capteur et sur laquelle on place des poids calibrés.

Voici le schéma de branchement du capteur pour les tests :

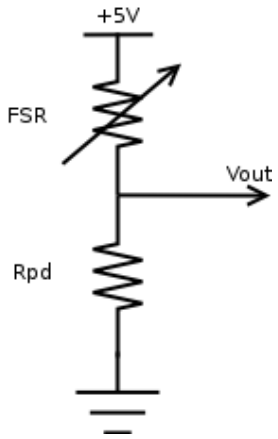


FIG. 4.2 – Branchement basique des FSR

Le tableau ci-dessous présente les résulatats de 3 expériences successives.

Pièce	***Poids (g)***	***Tension (mV)***	***Tension (mV)***	***Tension (mV)***	***Moyenne (mV)***
Vide	0	3,4	3,4	3,5	3,43
Vide + Plateau	55	260,4	223	335	272,8
Plateau + Vis	99	605	590	552	582,33
Plateau + Pièce 1	305	1550	1430	1530	1503,33
Plateau + Pièce 2	375	1710	1730	1810	1750
Plateau + Pièce 3	595	2420	2360	2450	2410
Plateau + Pièce 4	720	2530	2470	2560	2520
Plateau + Pièce 5	1039	2950	2920	2970	2946,67
Plateau + Pièce 3 *2	1145	3000	3030	3050	3026,67

Les courbes ci dessous sont une représentation graphique des données du tableau précédent.

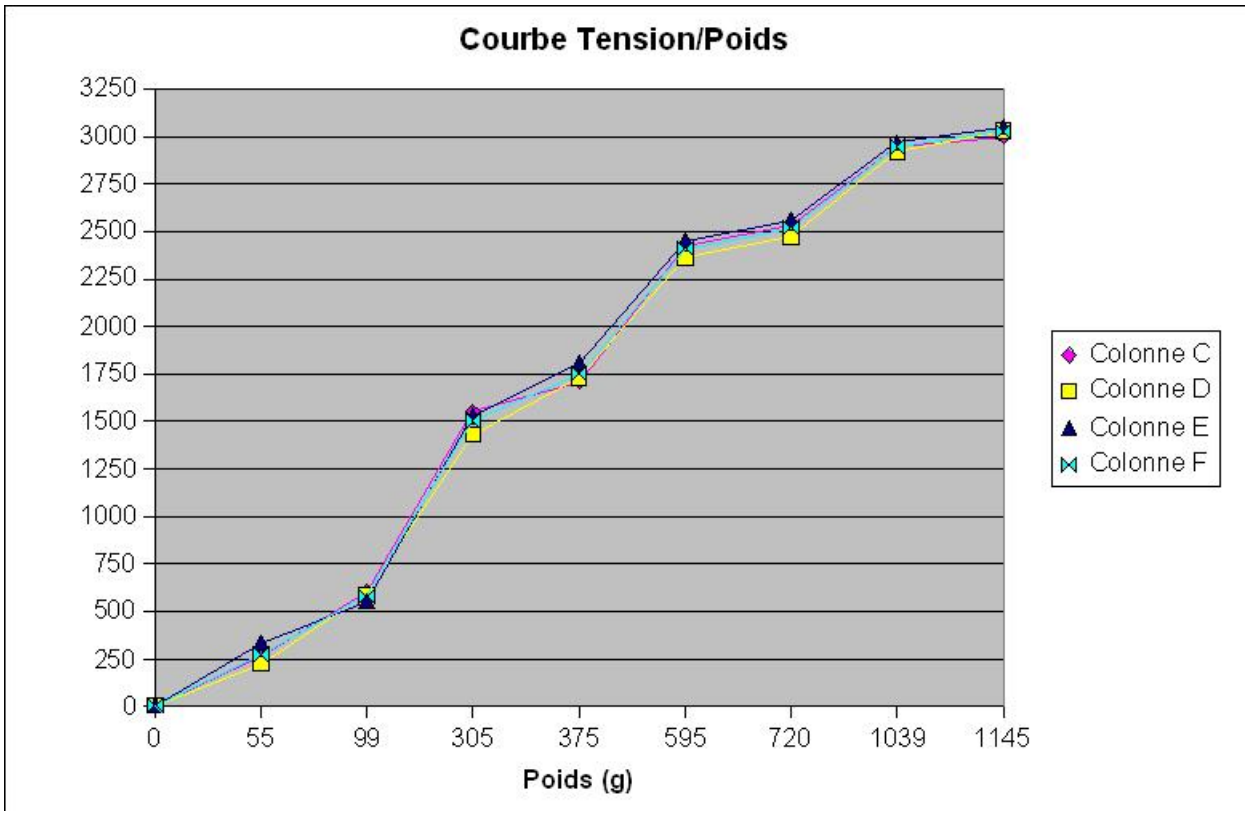


FIG. 4.3 – Réponse des capteurs FSR

Selon le montage, les capteurs peuvent avoir différentes utilisations. Quelques montages testés sont présentés dans la section suivante.

4.2 Montages

4.2.1 Placement des capteurs

Le plus important pour l'utilisation des capteurs FSR est l'installation mécanique du capteur. En effet, je me suis rendu compte que selon l'appui sur la zone active du capteur, les valeurs récupérées peuvent fortement varier. Il a donc fallu trouver une solution mécanique qui actionne uniformément toute la surface active. Le deuxième point important est le nombre et la répartition des capteurs. Il a été décidé d'intégrer 4 capteurs par semelles afin d'obtenir une bonne visualisation de la répartition de l'appui sur la semelle :

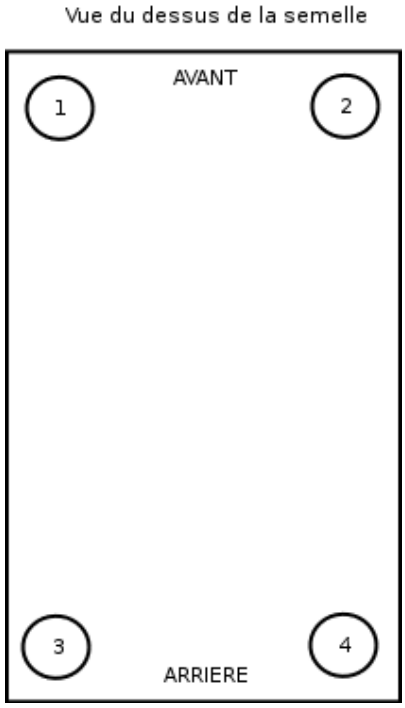


FIG. 4.4 – Répartition des 4 capteurs FSR

Les différents montages mécaniques que j'ai testés sont présentés dans la section suivante.

4.2.2 Différents montages

Voici les différentes solutions de montages mécaniques testées pour l'actionnement des capteurs.

Montage 1 :

FIG. 4.5 – Montage 1

-> Problème de hauteur et de fixation (les patins glissent les uns sur les autres donc l'appui varie).

Montage 2 :

FIG. 4.6 – Montage 2

-> Problème de hauteur et de fixation du capteur (le capteur bouge entre les patins).

Montage 3 :

FIG. 4.7 – Montage 3

-> Problème d'appui (la surface des patins n'est pas forcément plane).

Après plusieurs essais de montages avec de petits patins en caoutchouc, je me suis rendu compte que certains de ces patins etaient légèrement plus petit que d'autres, causant donc les problèmes de hauteur. De plus, leur surface etait parfois légèrement oblique, causant ainsi une mauvaise répartition de l'appui et donc un mauvais actionnement des capteurs.

La solution finalement adoptée est la suivante :

FIG. 4.8 – Montage final

Les vis étant calibrées, le problème de hauteur irrégulière est résolu. L'appui se fait uniformément sur le capteur. Les capteurs sont fixés sur la base de la semelle entre deux couches de double face. Ceci permet de fixer le capteur et d'éviter les glissement des têtes de vis sur la surface active des capteurs. Les deux parties métallisque de la semelle sont maintenues par deux vis et un système d'écrou/contre-écrou évite que la semelle ne se décolle.

Avant de pouvoir exploiter les données des capteurs, il a fallu réaliser un étage électronique pour amplifier et récupérer les tensions.

4.2.3 Electronique

Les tensions récupérées varient entre 0V à vide et 0.4V environ en appui maximal (robot appuyé sur un seul capteur). Afin d'avoir une meilleure précision, il faut amplifier ces tensions. Une amplification de 10 permet d'obtenir une bonne précision et une tension variant entre 0V à vide et 4V. Cette amplification est réalisée avec un circuit intégré comportant 4 amplificateurs opérationnels.

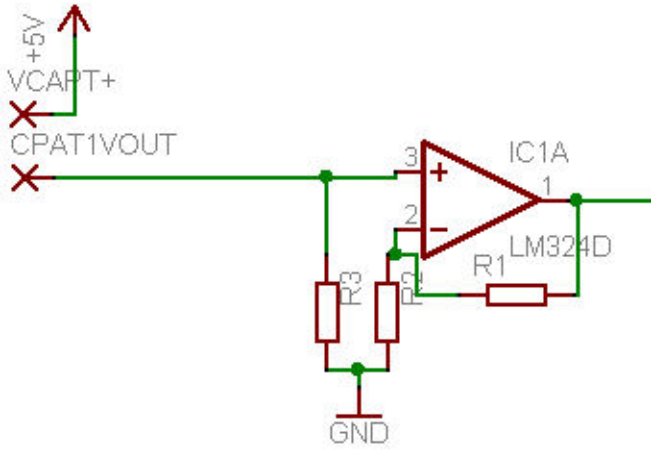


FIG. 4.9 – Amplification d'un capteur FSR

Dans une première version du traitement des données des capteurs, seule la détection de l'appui ou non du pied sur un des capteurs est réalisée. Les capteurs sont donc utilisés en tout ou

rien. Le principe pour utiliser les capteurs en tout ou rien est d'utiliser ceux-ci dans un montage comparateur qui comparera la tension issue du capteur et une tension seuil réglable. La tension de seuil correspond à la tension issue du capteur FSR non sollicité. Lorsque la tension issue du capteur est supérieure à cette tension seuil, alors la tension de sortie du comparateur passe à +5V.

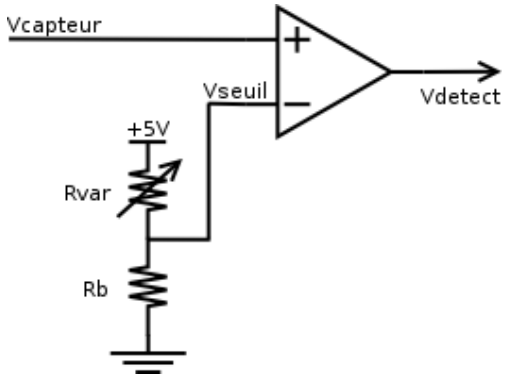


FIG. 4.10 – Montage Tout ou Rien

Un étage logique suit l'étage comparateur et permet ainsi de détecter 6 positions d'appui de la semelle :

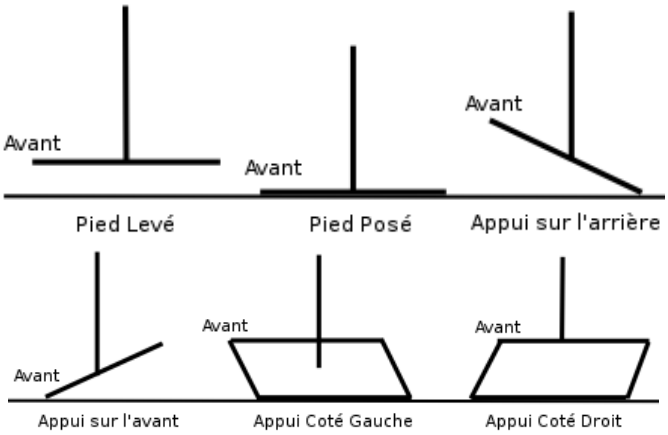


FIG. 4.11 – Positions détectées

Schéma de l'étage logique :

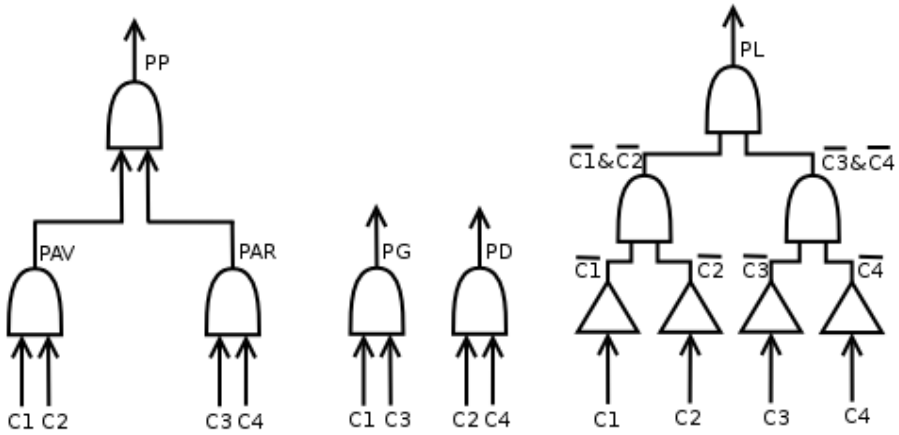


FIG. 4.12 – Diagramme logique

Cette version permettait donc de valider les montages des capteurs. Une nouvelle version plus précise était nécessaire et il a donc fallu passer à un traitement numérique des données. Ainsi, dans la version finale, les tensions amplifiées sont converties en valeur numérique sur 8 bit (0 à 255) par le convertisseur analogique/numérique 4 voies Philips PCF8591. La communication avec le composant se fait par bus I2C. Ce bus présente l'avantage de réduire le nombre de fils à 2 pour communiquer. Il a donc fallu tester la communication I2C sur PC pour vérifier le bon fonctionnement du montage.

Pour tester la communication I2C sur PC, une interface sur port parallèle m'as été fournie, ainsi qu'un code source réalisant les bases du protocole I2C (envoi de la trame 'Start', 'Stop', ...). Partant de ce code, j'ai réalisé une première application qui permet de communiquer via cette interface avec le composant PCF8574 (8 entrées/sorties numériques qui branché sur l'étage comparateur précédement décrit, permet de détecter sans circuits logiques les positions du pied) et avec le composant PCF8591 (convertisseur analogique/numerique 4 voies). (Voir figure 'Logiciel de test I2C' page suivante)

Après tests, le composant PCF8591 et la communication I2C marchent bien. Cette solution de conversion et communication est donc confirmée pour le montage final. Un circuit imprimé intégrant l'amplification et le composant PCF8591 pour chaque semelle à été réalisé.

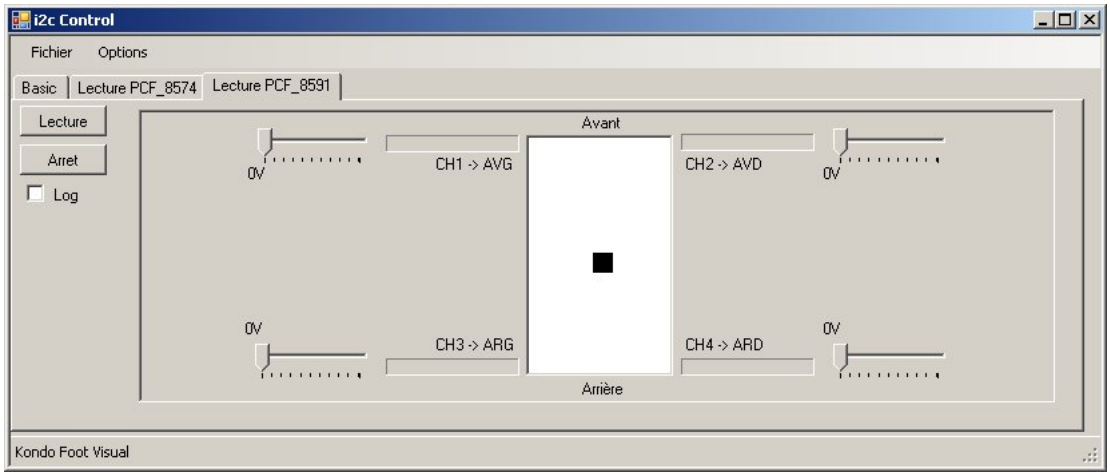


FIG. 4.13 – Logiciel de test I2C

La prochaine étape du projet est de centraliser les données des capteurs et de communiquer avec le robot. Pour ce faire, il a fallu utiliser une carte embarquant un microcontrôleur. Cette carte est expliquée dans le chapitre suivant.

Chapitre 5

Carte d'extension

L'idée de la carte d'extension est de centraliser toutes les informations du robot et d'offrir une seule interface entre le PC et les différents composants (capteurs et cartes RCB-1) :

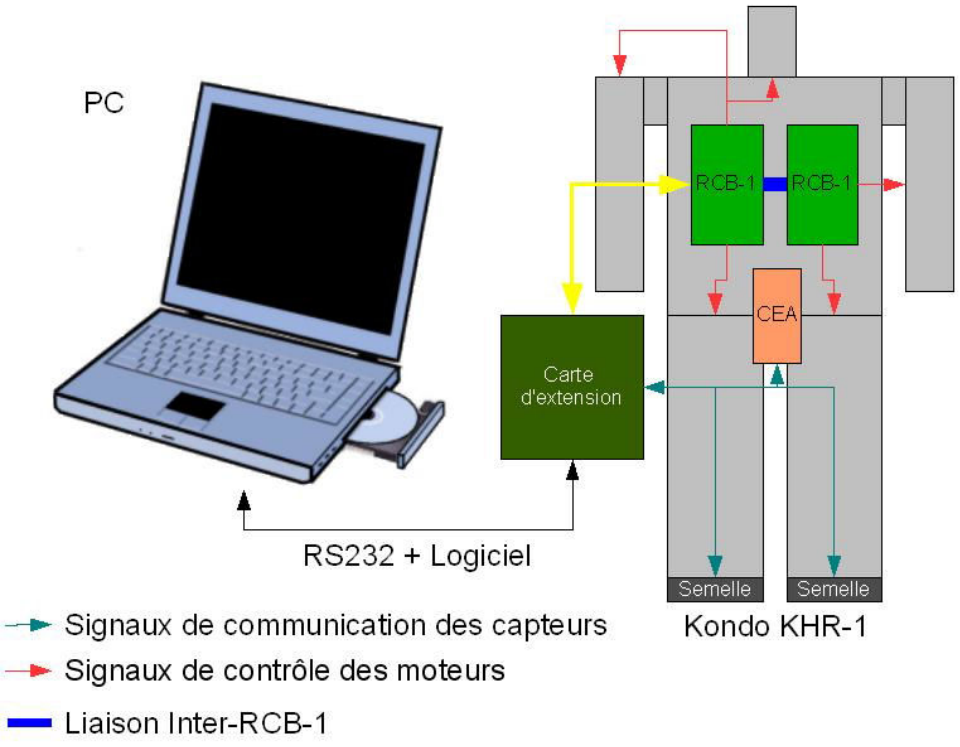


FIG. 5.1 – Robot avec extension

Le microcontrôleur devait présenter les caractéristiques suivantes :
- Possibilité de communication série avec deux ports série (un port pour communiquer avec le PC, l'autre pour communiquer avec le robot).
- Communication par bus I2C
- Convertisseurs Analogiques/Numériques (pour l'ajout de capteurs comme le capteur CEA présenté dans le chapitre suivant).
- Puissance assez bonne
- Evolutivité (possibilité d'ajouter de nouvelles commandes, de nouveaux capteurs, un asservissement, ...)

Pendant la recherche du microcontrôleur, un membre de l'équipe E-motion de l'INRIA nous a conseillé l'utilisation du microcontrôler AtMega128 qu'eux même utilisaient pour leur Robot. De plus, ils nous ont fourni une carte avec ce microcontrôleur afin que l'on puisse tester et développer notre application. Cette carte présentais l'avantage de fournir tous les composants et connecteurs nécessaires (deux ports série, deux connecteurs pour le bus I2C, ports du microcontrôleur répoliqué sur broches, ...). Le seul inconvénient de cette carte était sa taille mais ne nécéssitait que quelques modifications pour l'adapter à notre projet.

Le schéma de la carte finale est donné a la fin de ce chapitre.

5.1 Spécification microcontrôleur AtMega128

Caractéristiques principales du microcontrôleur AtMega128 :
- Alimentation : 4.5 - 5.5 Volt
- Mémoire : 128-Kbyte self-programming Flash Program Memory, 4-Kbyte SRAM, 4-Kbyte EEPROM
- Vitesse : 0 - 16MHz
- Interfaces de programmation : JTAG, SPI

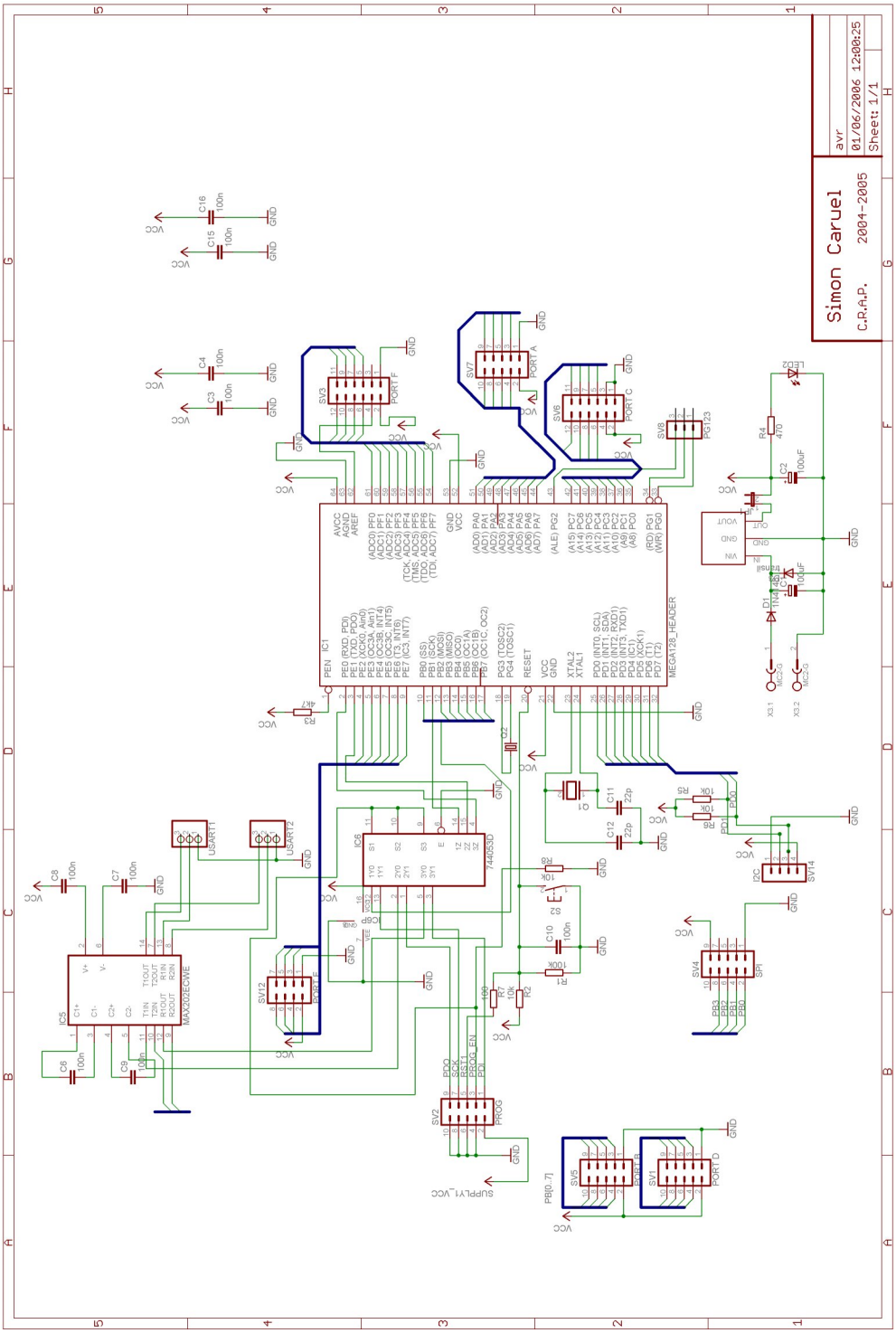
Principaux périphériques intégrés :
- 53 Entrées / Sorties
- 2 timers/compteurs 8bit
- 2 timers/compteurs 16bit
- 6 voies PWM (Pulse Width Modulation)
- 8 convertisseurs Analogique-Numérique 10 Bits
- Bus I2C
- 2 USART programmables

5.2 Spécifications de la carte

Caractéristiques principales de la carte :

– Dimensions : 67mm x 67mm
– Alimentation : 6V à 12V
– Led de contrôle d'alimentation
– Programmation par SPI (Serial Programming interface) et JTAG
– 2 ports USARTS avec MAX202 intégré (Voir brochage)
– 1 bus I2C (sur connecteur indépendant) (Voir brochage)
– Ports A, B, C, D, E et F répliqués sur connecteurs Males

La documentation technique complète de la carte est disponible en Annexe A.


Simon Caruel
C.R.A.P.
2004-2005
avr
01/06/2006 12:00:25
Sheet: 1/1
MEGA128_HEADER
MAX202ECWE
74ABT638
USART1
USART2
SPI
I2C
PORT A
PORT C
PORT D
PORT F
PORT G
PROG
PG123
SUPPLY1_VCC
VCC
GND

Chapitre 6

Autre capteurs

6.1 CEA

Les capteurs CEA sont des capteurs développés par le Commissariat aux Energies Atomiques. Les informations détaillées sur ce capteur sont confidentielles et seules quelques informations sont disponibles. Ainsi, nous savons seulement que les capteurs CEA sont constitués de 6 capteurs répartis sur 3 axes formant un repère orthonormé. Chaque axe est équippé d'un accéléromètre et d'un magnétomètre. Pour récupérer les valeurs de chaque capteurs, nous utilisans 6 convertisseurs analogiques/numériques présents sur la carte d'extension.

6.2 Moteurs

Afin de pouvoir controler l'état du robot, il est important de connaitre la position des moteurs en temps réel. La solution a été de récupérer ces positions en les lisant dans les carte RCB-1. En effet, une fonction de ces cartes permet de récupérer instantannément la position des moteurs.

Une fois les parties matérielles crées et testées, il a fallu créer les parties logicielles.

Chapitre 7

Photos du robot équippé du kit d'extension

Voici quelques photos du robot équippé de la carte d'extension, du capteur CEA et des semelles.

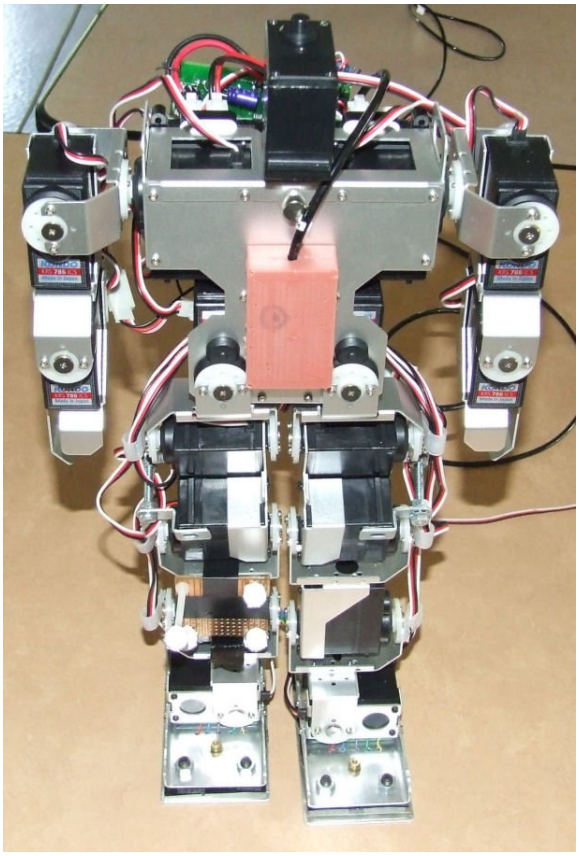


FIG. 7.1 – Vue de la face avant du robot équipé du capteur CEA (Bloc orange)

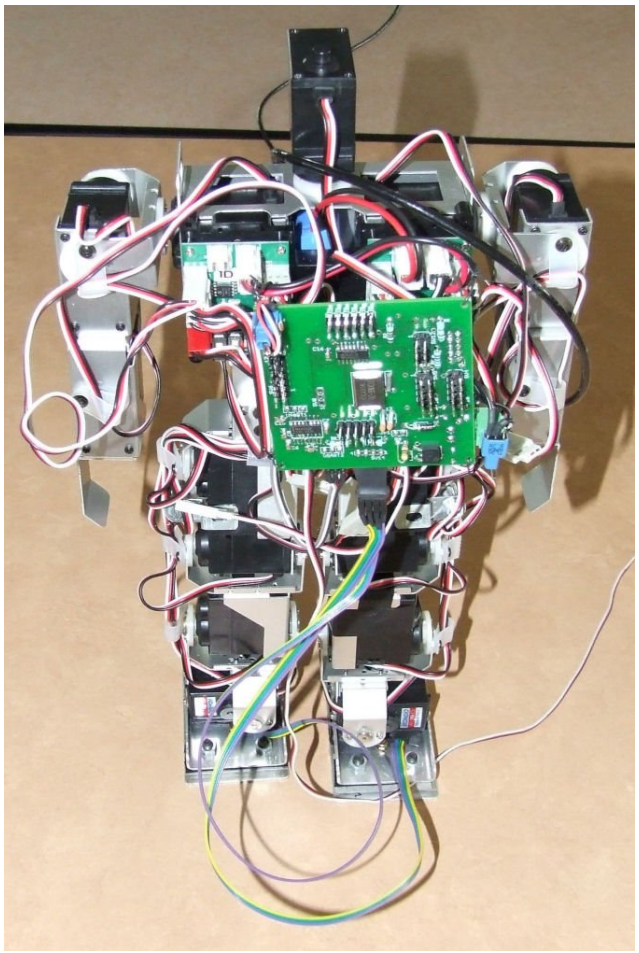


FIG. 7.2 – Vue de la face avant du robot équipé de la carte d'extension et des semelles

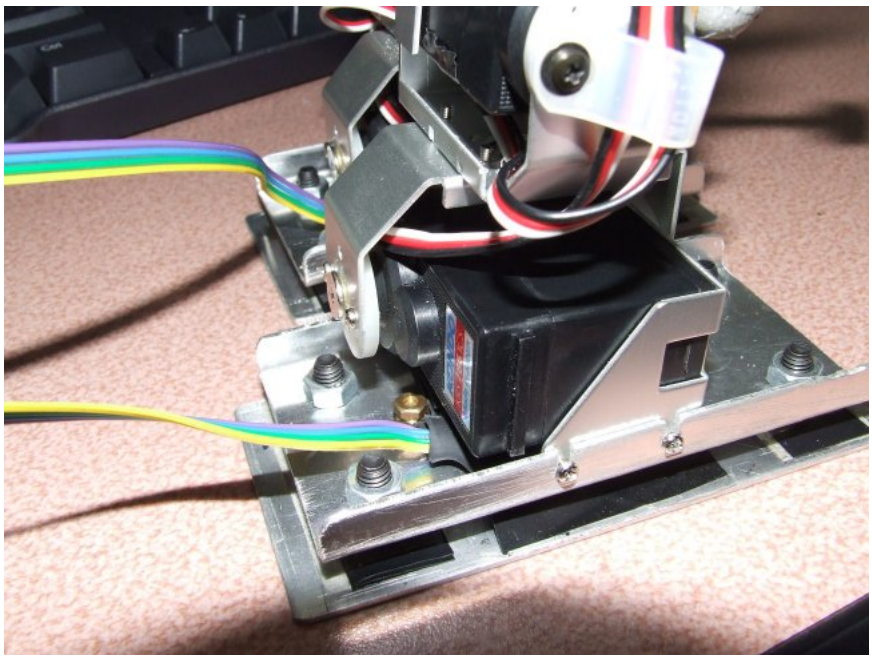


FIG. 7.3 – Vue arrière des semelles

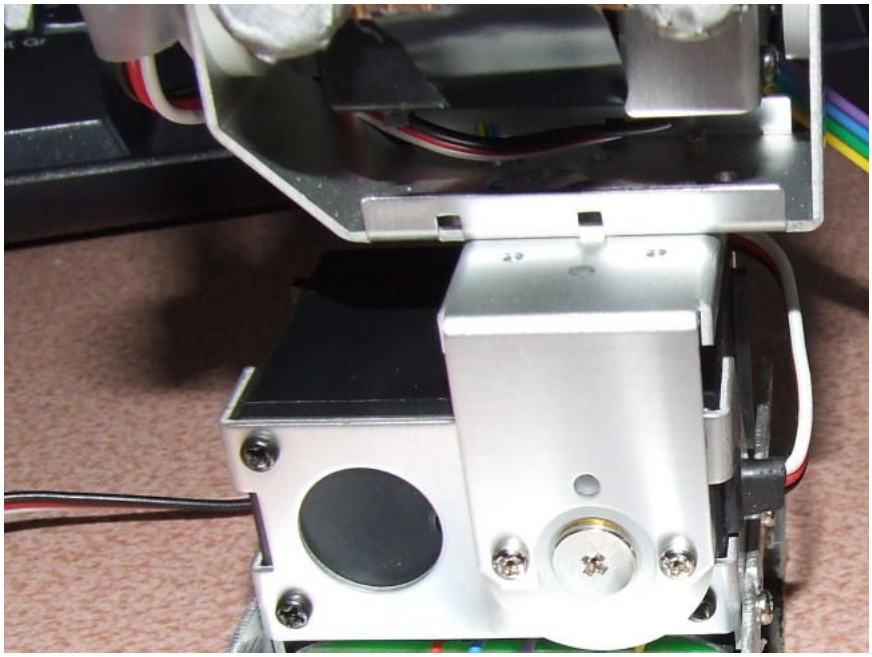


FIG. 7.4 – Vue de la face avant du robot équipé de la carte d'extension et des semelles

Troisième partie

Interface Logicielle

Cette partie à pour but de décrire les différentes étapes de la création de la partie logicielle du projet. Cette partie est très importante car elle permet de créer une évolutivité de la carte d'extension. Il a donc fallu définir certains paramètres tels que portabilité ou langages avant de se lancer dans la réalisation.

La première chose à faire était de créer une 'bibliothèque' de communication avec le robot. Cette bibliothèque est décrite dans le chapitre suivant.

Chapitre 8

API Kondo

La communication avec le robot se fait avec les cartes RCB-1 installées sur celui-ci. Ces cartes intègrent un microcontrôleur qui permet de controler cette carte et d'obtenir certaines informations. La communication avec les cartes RCB-1 se fait par liaison série. Après quelques recherches, j'ai trouvé sur un site (http://www.robosaavy.com/) un document décrivant le format des données à envoyer pour communiquer avec les cartes. Les données à envoyer et celles émises par la carte en réponse sont décrites dans ce document : [RCB-1 Communication Protocol].

L'API (Application Programming Interface) du robot doit permettre de communiquer avec ces cartes sur différentes plateformes (Linux, Windows et AVR au moins). Ainsi, j'ai décidé de séparer l'API en deux parties. La partie commune met en forme la trame, l'envoi et éventuellement, analyse la réponse. La partie dépendante de la plateforme d'utilisation contient les fonctions d'ouverture/fermeture de port, d'envoi de commande d'écriture et d'envoi de commande de lecture.

Lors de la compilation, la plateforme doit être sélectionnée afin de compiler le code avec les fonctions correspondantes.

La documentation complète de l'API est fournie dans en Annexe C.

Chapitre 9

API carte d'extension

La communication avec la carte d'extension de fait par liaison série. Afin de simplifier la communication avec la carte d'extensions, celle-ci est basée sur des commandes formattées permettant ainsi d'être facilement utilisées par la carte et d'ajouter facilement de nouvelles fonctions sur cette carte.

Voici le format générique des commandes de l'API de la carte d'extension :
[TX]
CMDP CMD_LENGTH CMD1 CMD2 ... CMD_LAST CHKSUM

Paramètres :

Paramètre	Valeur	Description
CMDP	DDh	Commande AtMega128 API
CMD_LENGTH	00h - FFh	Longeur Commande *
CMD1 - CMD_LAST	00h - FFh	Paramètres de la commande
CHKSUM **	00h - 7Fh	(CMDP + CMD_LENGTH + CMD1 + ... + CMD_LAST) & 7Fh

* La longeur de la commande est le nombre de bits envoyés - 2
** & est un ET logique, + est une addition
exemple de chksum : (FFh + 5Ah + 2h) & 7f = 5Bh

Cette API à pour but de facilement pouvoir récupérer les données que la carte centralise. Les principales fonctions de cette API permettent de récupérer les valeurs des capteurs et de communiquer basiquement avec le robot (mise en position Home, lecture de la position Home, lecture d'une motion).

Le document [Protocol de communication de la carte d'extension] nous donne le format des commandes.

La documentation complète de l'API est disponible en Annexe D.

Chapitre 10

Code Embarqué

Le code embarqué dans la carte joue un rôle important. En effet, la carte doit communiquer avec le PC, le robot et les capteurs. Aussi, la difficulté principale était de prévoir un code permettant une évolutivité de la carte. Il etait donc nécessaire de bien définir la communication. Les fonctions de communication sur port série sont disponibles dans les bibliothèques fournies par l'équipe d'E-motion et n'ont nécessité qu'une légère modification afin de pouvoir utiliser ces fonctions sur les deux ports.

10.1 Communication avec les capteurs

Les fonctions de communication sur bus I2C ainsi que les fonctions de conversion Analogique/Numérique sont présentes dans des bibliotheques du pack WinAVR (pack d'outils et de bibliothèques pour les microcontôleurs AVR). Leur mise en place à donc été simplifiée ainsi que leur utilisation.

10.2 Communication avec le robot

Pour communiquer avec le robot, il a fallu inclure l'API du robot au programme en compilant cette API pour la plateforme AVR (AVR le type de microcontrôleur de l'AtMega128).

10.3 Communication avec le PC

La communication avec le PC se faisant par liaison série et utilisant des commandes au format prédéfini, il a fallu mettre en place une analyse des trames reçues. Une fois analysée, la fonction correspondant à la commande est appellée et, si besoin est, le résultat est envoyé. Si aucun résultat n'est à envoyer, la carte emet une trame d'acquittement pour confirmer la bonne exécution de la commande. Les commandes et résultats renvoyés sont vérifiables grâce à la présence d'un checksum : lors de l'émission, cette valeur est calculée selon les données émises, lors de la réception, la valeur de ce checksum est comparée à la valeur calculée à partir des

données recues. Voir le document [Protocol de communication de la carte d'extension] pour le calcul de ce checksum. Ceci permet d'éviter de traiter les données érronnées.

Chapitre 11

Interface Graphique

L'interface graphique créée permet de contrôler facilement et visuellement la carte d'extension. Elle permet aussi de capturer les données reçues et de les enregistrer pour les exploiter avec un logiciel extérieur (Scilab par exemple). Une partie visualisation des données est aussi incluse permettant de voir graphiquement le point d'appui sur chaque semelle et la vision 3D du robot. Aussi, cette interface reprends les majeures fonctions du logiciel HeartToHeart fourni avec le robot. Ainsi, il permet l'édition des motions et des scénarios.

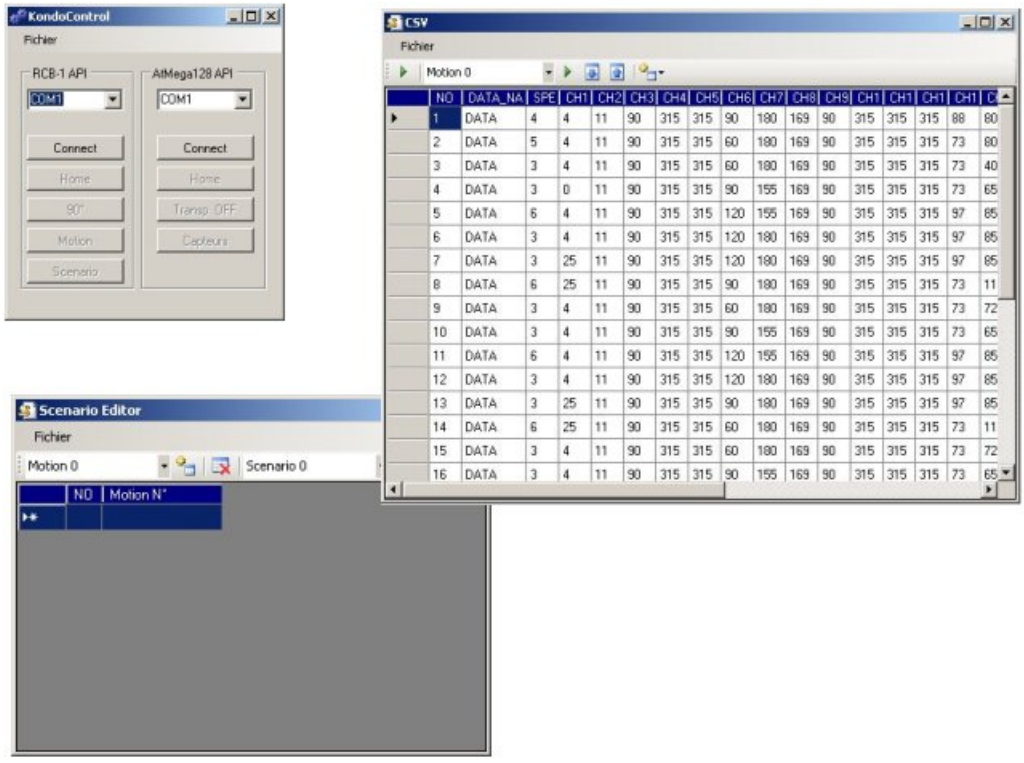


FIG. 11.1 – Fonction du logiciel HeartToHeart

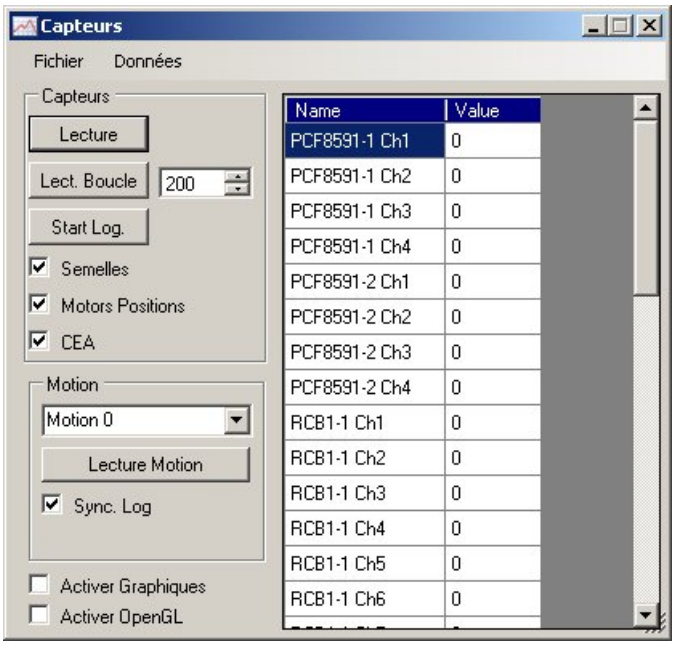


FIG. 11.2 – Fenêtre principale de contrôle des capteurs

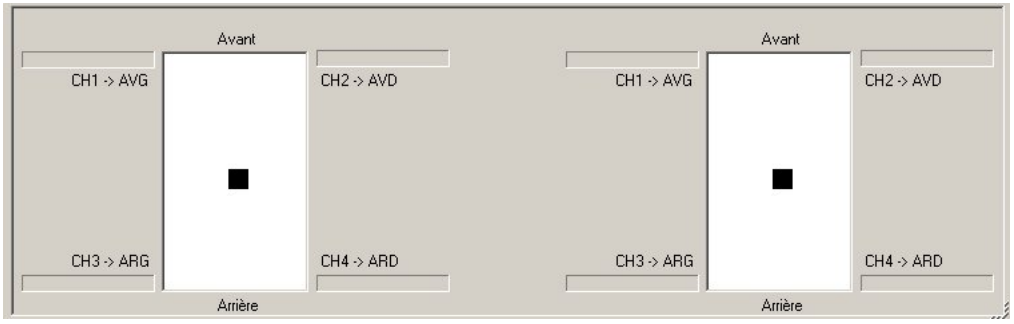


FIG. 11.3 – Visualisation graphique des points d'appuis

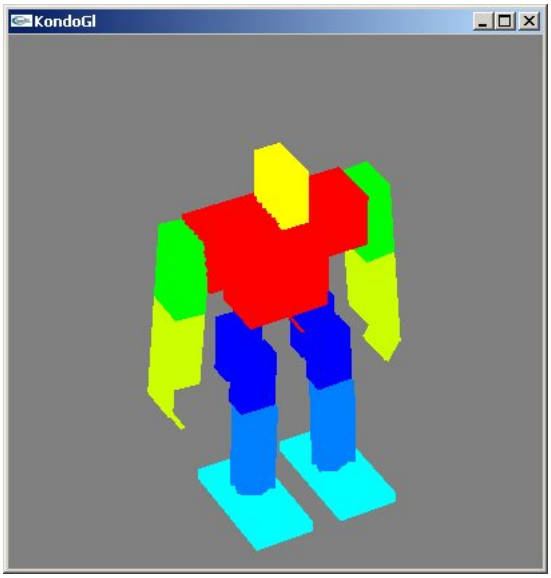


FIG. 11.4 – Visualisation 3D du robot

La documentation complète du logiciel est fournie en Annexe A.

Chapitre 12

Tests et Résultats Expérimentaux

Après plusieurs tests du logiciel, quelques bugs ont été résolus, principalements dûs à des erreurs d'indices dans les envois/réceptions de commandes avec le robot. Aussi, j'ai changé de méthode de codage pour les fonctions d'envoi/réceptions de commandes afin d'accélérer le programme (suppression des temps morts imposés remplacés par des boucles avec limite de temps équivalente au temps morts précédements imposés). Ceci à eu pour effet d'améliorer la rapidité d'éxécution et donc la possibilité de capture des données (fréquence d'acquisition supérieure). On note cependant un ralentissement du programme lorsque tous les capteurs sont capturés en même temps. De même, l'interface graphique ralentit l'éxécution du programme (un logiciel plus rapide pourrait être créé en utilisant un autre logiciel de développement ou en créant une simple interface 'texte'. Aussi, la gestion du temps étant plus précise sous linux, il serais utile de développer une version de l'interface pour linux. Ceci est faisable étant donné la portabilité des API.

Pour completer les tests logiciels, j'ai réalisé, en collaboration avec un collègue chargé de la modélisation géométrique et dynamique du robot, une comparaison entre les données récupérées par la carte (dans ce cas là, la position des moteurs), le résultat théorique du mouvement effectué et un troisième système de capture appelé Cyclope (donnée optique).

Voici les trois courbes des résultats (courbe bleue = Carte, courbe jaune = HuMAnS, courbe verte = Cyclope) :

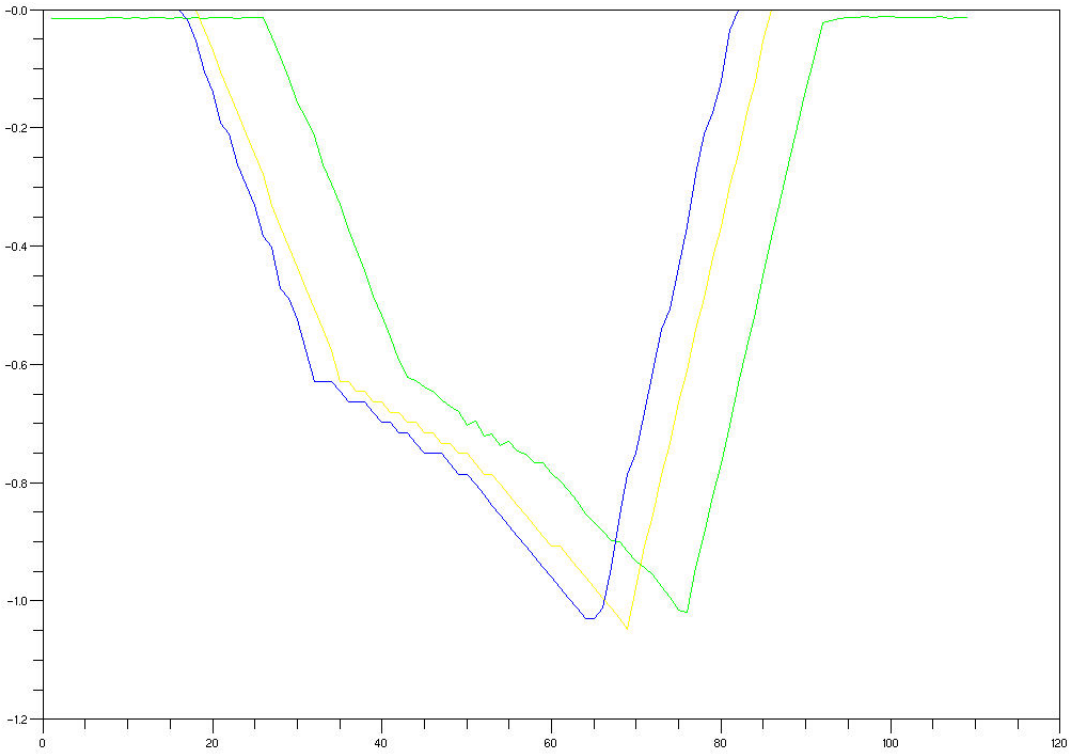


FIG. 12.1 – Courbes des trois résultats

On constate que les trois courbes sont de forme similaire. Seules quelques petites erreurs sont notables (zone de pic inférieure).

Quatrième partie

Conclusion

Le stage s'est globalement bien déroulé en respectant les temps fixés par le planning prévisionnel. Malgré un retard pris lors de la réalisation des semelles au début du stage, le travail est quasiment complet. Le semelles sont opératives bien que certains ajustements ne soient certainement nécessaires au niveau mécanique ou en post-traitement des données. La carte d'extension est opérative et évolutive grâce au systeme de commandes formattées utilisé. L'API du robot est fonctionnelle sous Windows et sur les microcontrôleurs AVR, mais il reste à écrire les fonctins d'ecriture/lecture du port série pour linux afin de pouvoir l'utiliser sur cette plateforme. L'API de la carte d'extension n'est pas encore portée sous une autre plateforme que Windows mais cela peut se faire simplement en créant les fonctions d'écriture/lecture sur port série pour la plateforme désirée.

Après quelques test primaire du kit d'extension, on constate que l'ensemble fonctionne mais que certaines améliorations peuvent être apportées à l'interface graphique (notamment une amélioration au niveau de la lenteur d'éxécution, majoritairement dûe aux composant utilisés par Visual Studio).

Cinquième partie

English Summary

I made my training in the INRIA (Institut National de Recherche en Informatique et Automatique) at Grenoble (38). This Institute is involved in a lot of research projects, especially in robotics projects. The aim of my training project was to add some sensors to the Kondo KHR-1, a humanoïd robot.

In this objective, I first started to choose sensors for the foot. I used FSR sensors, which are sensors that would best suit the application I had to do because they have a quite good precision and a quite fast response. Those sensors are very thin so they can be used in a lot of applications. The only major problem of those sensors is that the actuators are very important. If two actuators are a little bit different (shape or size), the response will be totally different. As soon as I fully tested the sensors, I made a first version of the robot soles with a logical circuit which was detecting six positions of the foot (foot not laying on the floor, foot fully layed on the floor, foot layed on the left side, right side, front side and back side). This version was not precise enought so I had to do a new circuit with a microcontroller. Using a microcontroller would permit to work with analog values of the sensors to have a better precision but also, it permits us to add some functions to the extension. The major new feature was to embed the KHR-1 communication protocol on the microcontroller so that we would not need a computer to control the robot (it opens the way to automation, autocorrection of position if robot is going to fall, ...). Also, we have add an other sensor named CEA which includes 6 sensors (one accelerometer and one magnetometer per axe, on three othonormal axes). I made the microcontroller code in a way that permit to add some more functions or sensors for the next developpers.

To control the card from a computer, I had to made a software which would be able to communicate with it. This software was also made to capture the sensors values so that they can be analysed. Thus, motors positions, CEA values and foot sensors values can be captured independently and saved in independent files. Some extra features are also provided as 3D real time view of the robot position (according to the motors positions), mass repartition on each foot.

As a conclusion, I would say that this training was very interesting, even if I ain't had time to accomplish all the work I wanted to do, especially, optimising the software to get better capture speed and the hardware to get more accuracy on the foot sensors.

Table des figures

Bibliographie

[Manuel HeartToHeart] Kondo, *HeartToHeart Operations Manual*, Kondo Kagaku CO. LTD, 2004

[RCB-1 Communication Protocol] http://robosavvy.com/Support/images/RCB1stdraft.doc, *RS232 command reference. Communications between PC and RCB-1.*, Dan Albert of sjrobotics.org, 2005

[Protocol de communication de la carte d'extension] BRUNEAUX Jérôme, *Protocol de communication de la carte d'extension*, INRIA, 2006

Sixième partie

Annexe A : Documentation Technique de la carte d'extension

Introduction

Ce document à pour but de présenter la carte d'interface utilisée pour le projet d'intégration de capteurs sur le robot bipède Kondo KHR-1. Cette carte est basée sur un microcontrôleur AtMega128 fabriqué par ATMEL. Elle est donc facilement évolutive car facilement programmable. Un logiciel accompagne cette carte afin d'avoir une interface visuelle pour récupérer les informations en provenance de cette carte (informations sur le robot et les capteurs).

Pré-Requis

L'interface logicielle AtMega128 nécessite un PC avec :
- Windows XP
- Microsoft .NET Framework Version 2.0 Redistributable Package (http://www.microsoft.com/downloads/)
- OpenGL Glut DLL (http://www.xmission.com/~nate/glut.html)
- Port Serie

L'interface matérielle AtMega128 requiert :
- Alimentation DC 6V à 12V
- Cable série PC (voir brochage dans chapitre 'Description Matérielle' section 'Brochages')
- Cable série Robot (voir brochage dans chapitre 'Description Matérielle' section 'Brochages')

Architecture

Robot sans extension

Le schéma ci-dessous représente l'architecture du robot tel qu'il est utilisable sans extension. Les deux cartes RCB-1 installées sur le robot communiquent d'une part avec les moteurs, d'autre part avec le PC par liaison RS-232 grâce au logiciel HeartToHeart.

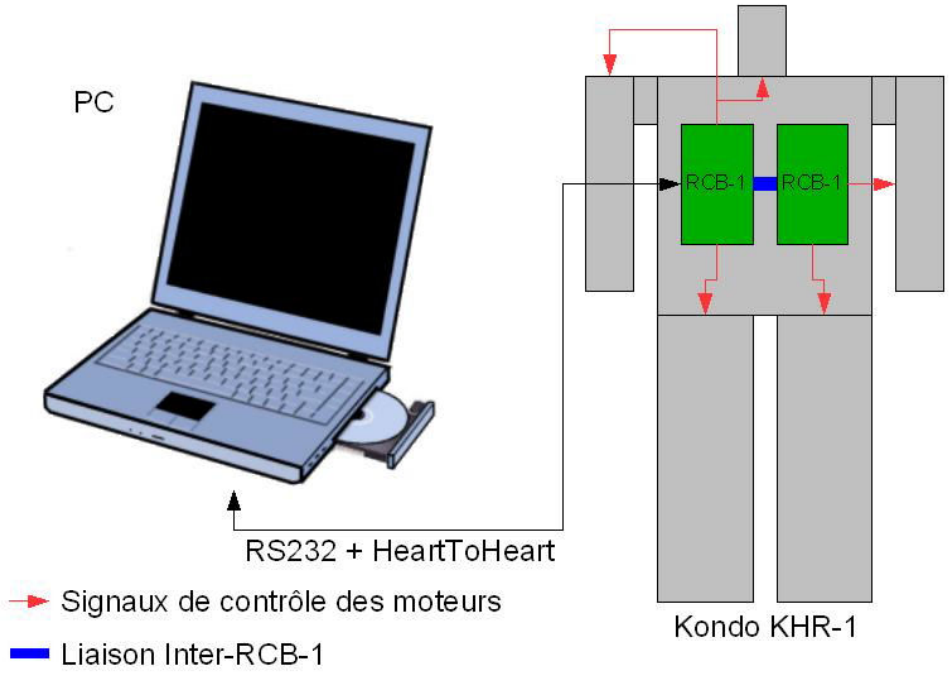


Fig. 12.2 – Robot sans extension

Robot avec extension

Le schema ci dessous représente l'architecture du robot avec le kit d'extension. Ce kit comprends la carte d'extension, les deux semelles équippées de capteur et un capteur CEA. Cette carte est branchée en intermédiaire entre le PC et le robot. Cette carte communique avec les cartes RCB-1 et le PC par liaison RS-232. Elle communique aussi par i2c avec les semelles pour récupérer les données des capteurs installés sur celles-ci. Les données du capteur CEA sont récupérées par la carte d'extension par convertisseur Analogique/Numérique.

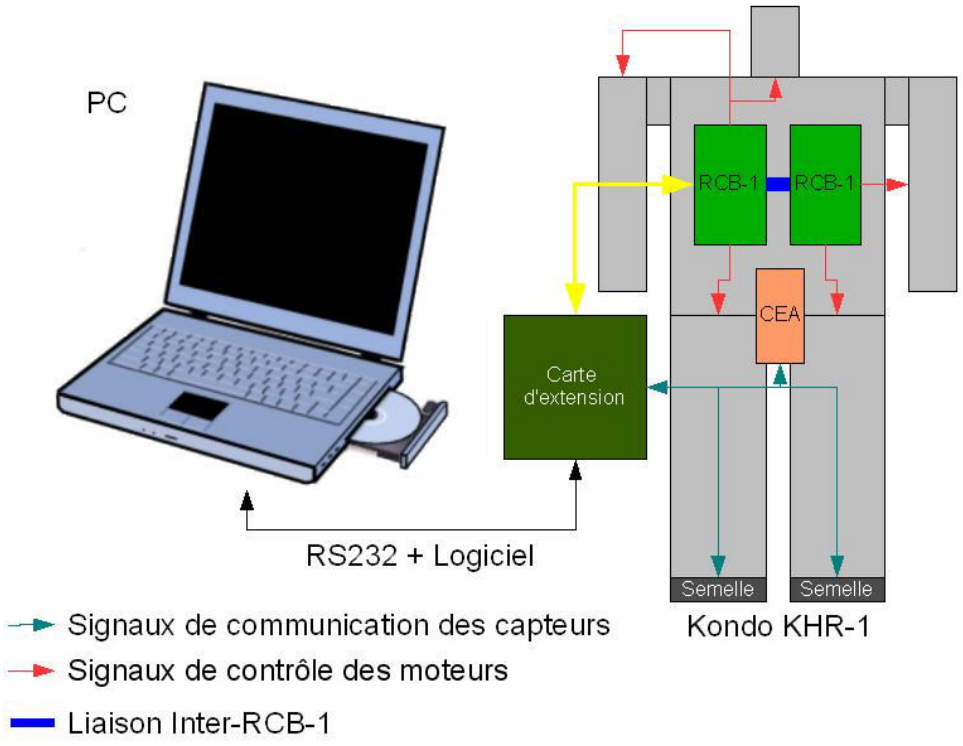
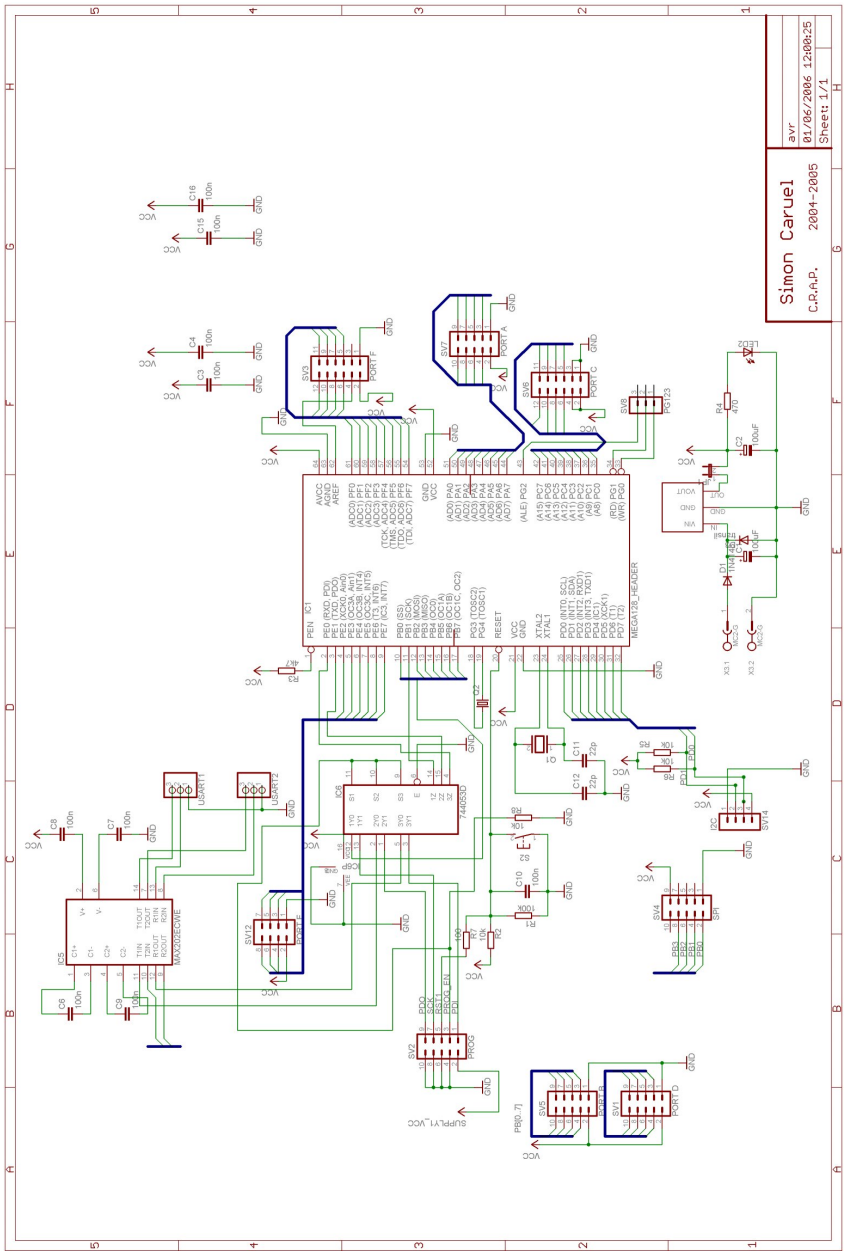


FIG. 12.3 – Robot avec extension

Description matérielle

Ce chapitre à pour but de présenter la carte AtMega128. Cette carte recoit un micro-contrôleur AtMega128 dont les fonctionnalités sont décrites dans la partie 3.2.1 . Les différents périphériques et connecteurs qui la composent sont également décrits afin que l'utilisateur puisse facilement ajouter de nouveaux périphériques ou capteurs.
La carte à été réalisée à l'aide du logiciel Eagle Layout Editor.

Schema de la carte



Fonctionnalités de la carte

12.0.1 Spécification microcontrôleur AtMega128

Caractéristiques principales du microcontrôleur AtMega128 :
- Alimentation : 4.5 - 5.5 Volt
- Mémoire : 128-Kbyte self-programming Flash Program Memory, 4-Kbyte SRAM, 4-Kbyte EEPROM
- Vitesse : 0 - 16MHz
- Interfaces de programmation : JTAG, SPI

Principaux périphériques intégrés :
- 53 Entrées / Sorties
- 2 timers/compteurs 8bit
- 2 timers/compteurs 16bit
- 6 voies PWM (Pulse Width Modulation)
- 8 convertisseurs Analogique-Numérique 10 Bits
- Bus I2C
- 2 USART programmables

12.0.2 Spécifications de la carte

Caractéristiques principales de la carte :
- Dimensions : 67mm x 67mm
- Alimentation : 6V à 12V
- Led de contrôle d'alimentation
- Programmation par SPI (Serial Programming interface) et JTAG
- 2 ports USARTS avec MAX202 intégré (Voir brochage)
- 1 bus I2C (sur connecteur indépendant) (Voir brochage)
- Ports A, B, C, D, E et F répliqués sur connecteurs Males

12.0.3 Dénomination des connecteurs

Cette figure indique le placement des différents connecteurs et leurs fonctions.

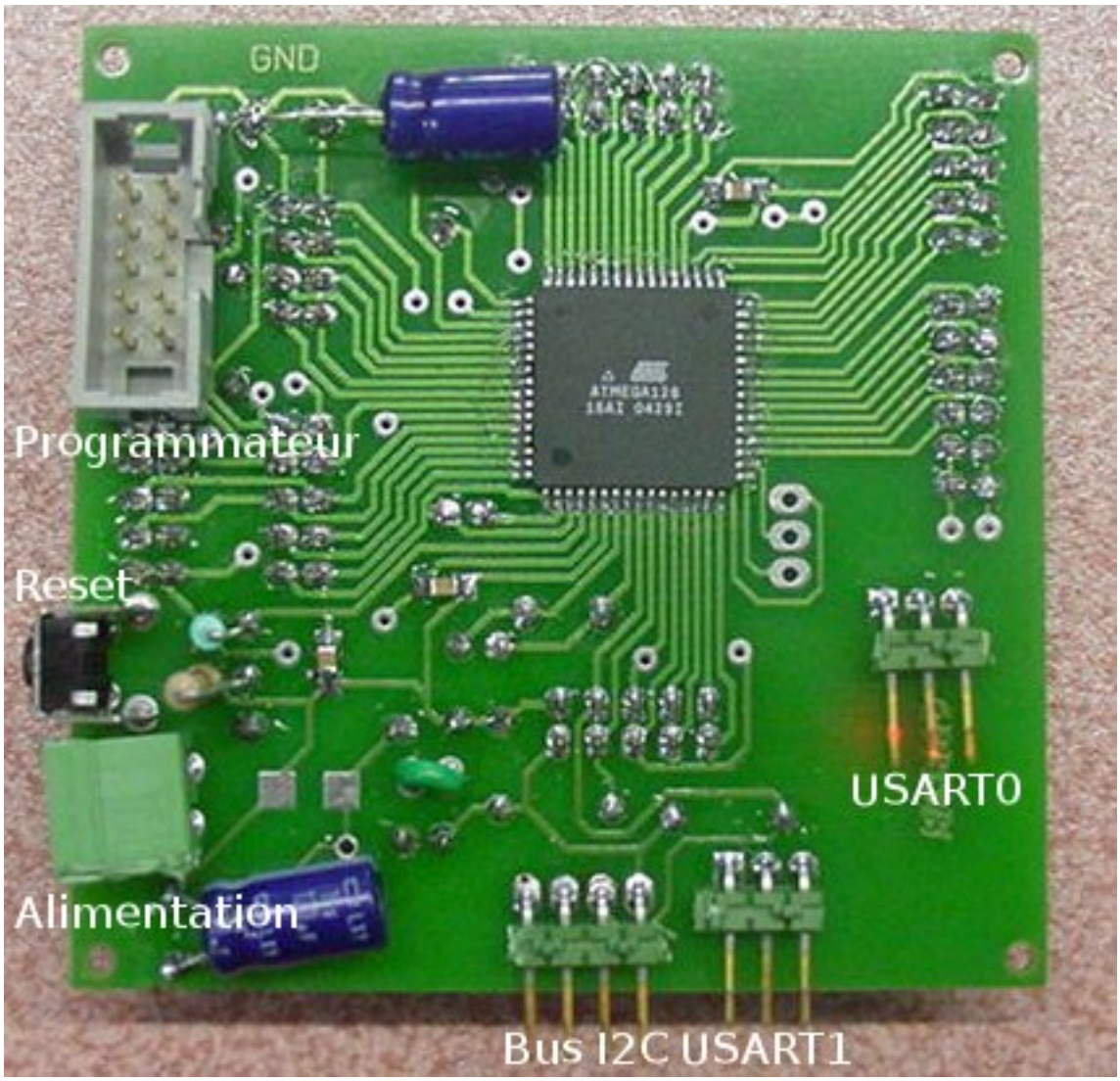


FIG. 12.4 – Connecteurs situés sur le dessous de la carte AtMega128

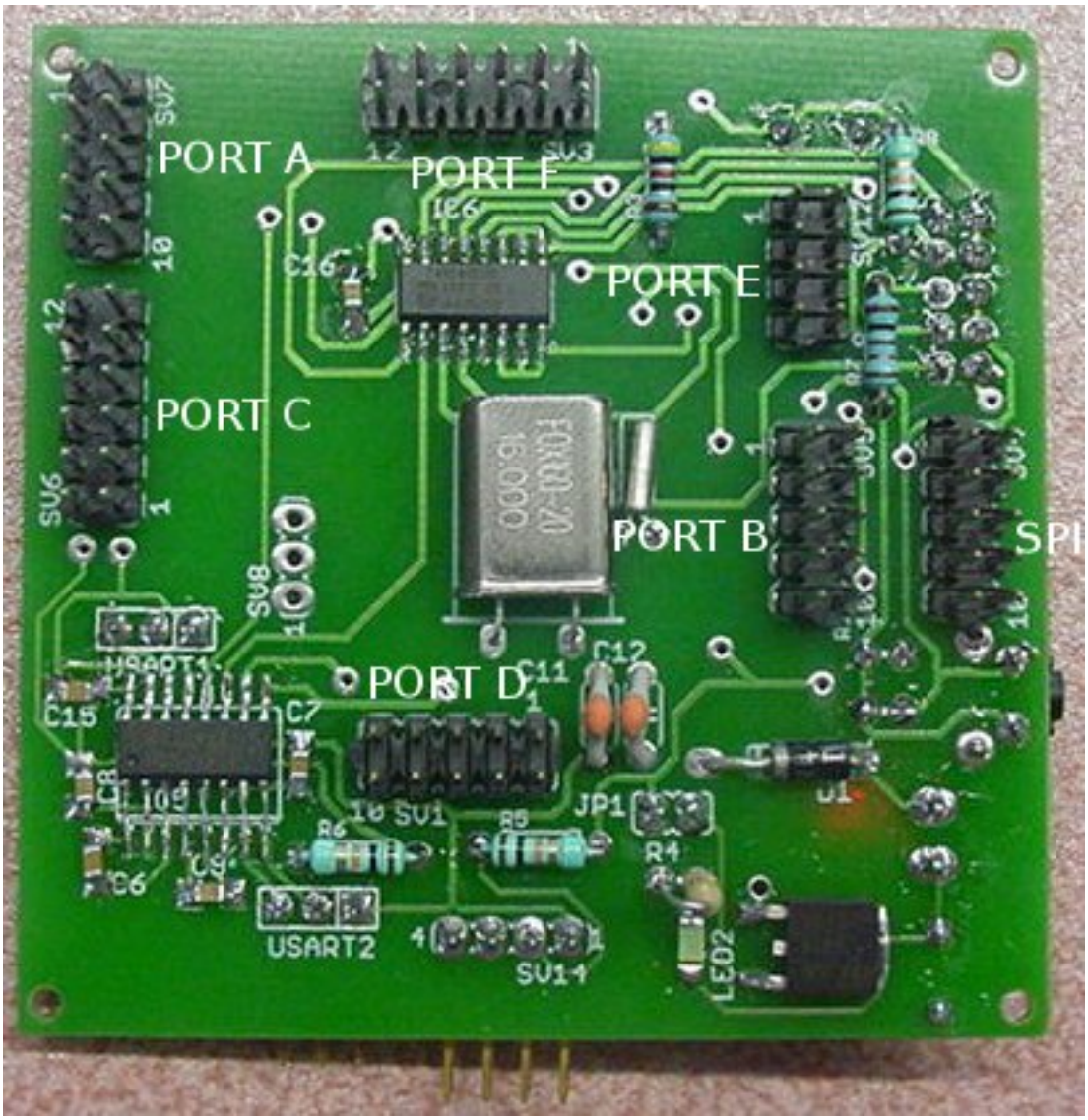


FIG. 12.5 – Connecteurs situés sur le dessus de la carte AtMega128

Brochages & Cables

12.0.4 Brochage Alimentation

Brochage du connecteur d'alimentation

FIG. 12.6 – Brochage du connecteur d'alimentation

12.0.5 Brochage Ports USART

Brochage des ports USART0 et USART1. Les trois fils correspondent à la masse, RX et TX.

FIG. 12.7 – Brochage des connecteurs des ports USART0 et USART1

12.0.6 Cable Série PC

Schéma de brochage du cable série reliant le PC à la carte AtMega128. Ce cable est composé d'une prise DB-9 et d'un connecteur 3 broches femelles au pas de 2.54mm. Les trois fils correspondent a la masse, au TxD et au RxD.

FIG. 12.8 – Brochage du cable série PC

12.0.7 Cable Série Robot

Schéma de brochage du cable série reliant la carte AtMega128 au robot. Ce cable est composé de deux connecteurs trois broches femelles au pas de 2.54mm. Les trois fils correspondent a la masse, au TxD et au RxD.

FIG. 12.9 – Brochage du cable série Robot

12.0.8 Brochage I2C

Schéma de brochage du bus I2C. Les quatres fils correspondent à la masse, +5V, SDA et SCL.

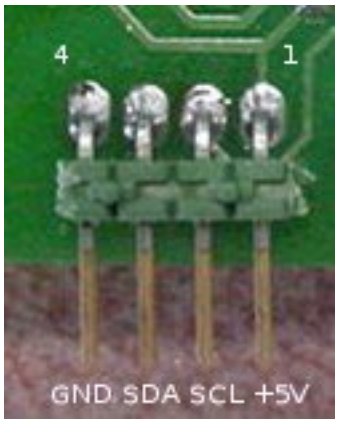


FIG. 12.10 – Brochage du connecteur du bus I2C

12.0.9 Ports

Voir schéma de la carte en annexe A.

12.0.10 Interface de programmation

Schéma de brochage de l'interface de programmation.

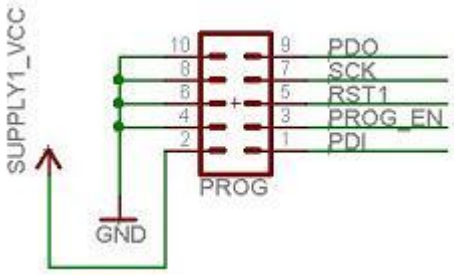


FIG. 12.11 – Brochage du connecteur de l'interface de programmation

Branchement des capteurs

12.0.11 Semelles I2C

Les capteurs des semelles I2C sont reliés à la carte d'extension par liaison I2C. La liaison I2C permet de connecter jusqu'à 128 périphériques sur le bus. Le branchement se fait en parallèle.

Un dédoubleur d'entrée I2C est fourni afin de pouvoir brancher les deux semelles. Sur les semelles, le fil jaune correspond à l'alimentation +5V, le fil vert correspond la ligne SDA, le fil bleu correspond à la ligne SCL et le fil violet correspond à la masse.

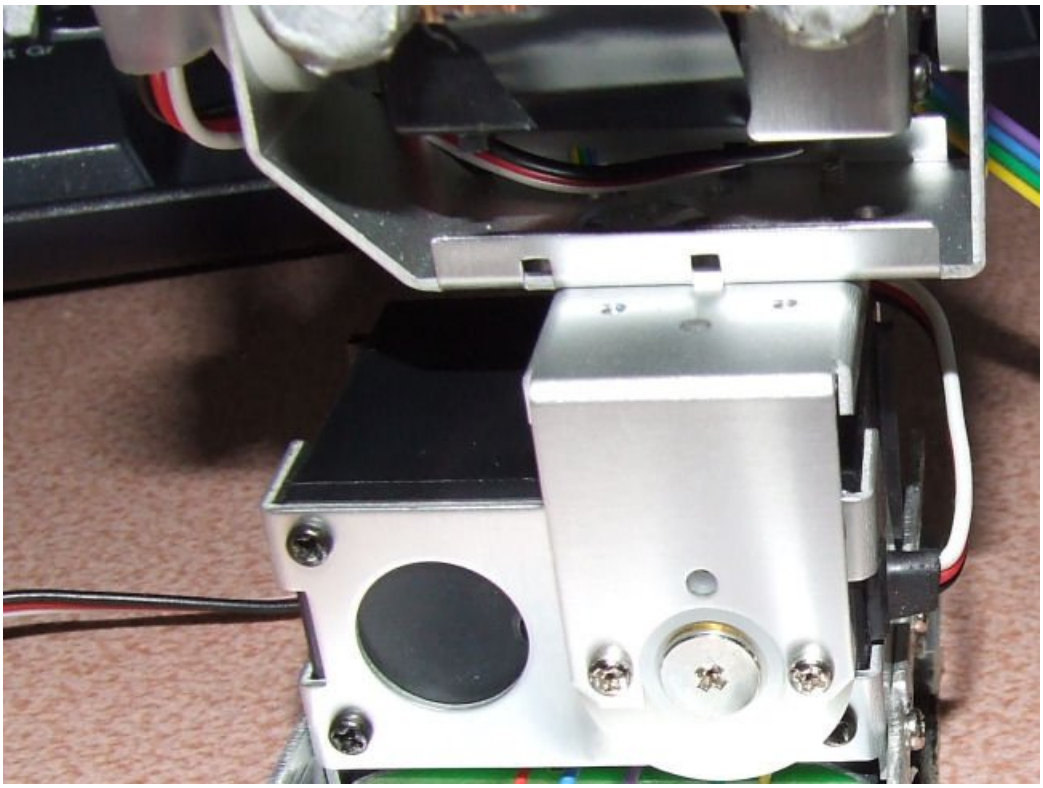


FIG. 12.12 – Semelles vues de face

12.0.12 Capteur CEA

Le capteur CEA se branche sur le port F. Ce port correspond aux convertisseurs analogiques/numériques du microcontrôleur et possède aussi une alimentation +5V pour le capteur. L'accéléromètre X est branché à la broche 0 du port F et le magnétomètre est branché sur la broche 1 du port F. L'accéléromètre Y est branché à la broche 2 du port F et le magnétomètre est branché sur la broche 3 du port F. L'accéléromètre Z est branché à la broche 4 du port F et le magnétomètre est branché sur la broche 5 du port F.

Le capteur CEA se branche sur les broches encadrées sur le schéma ci dessous.

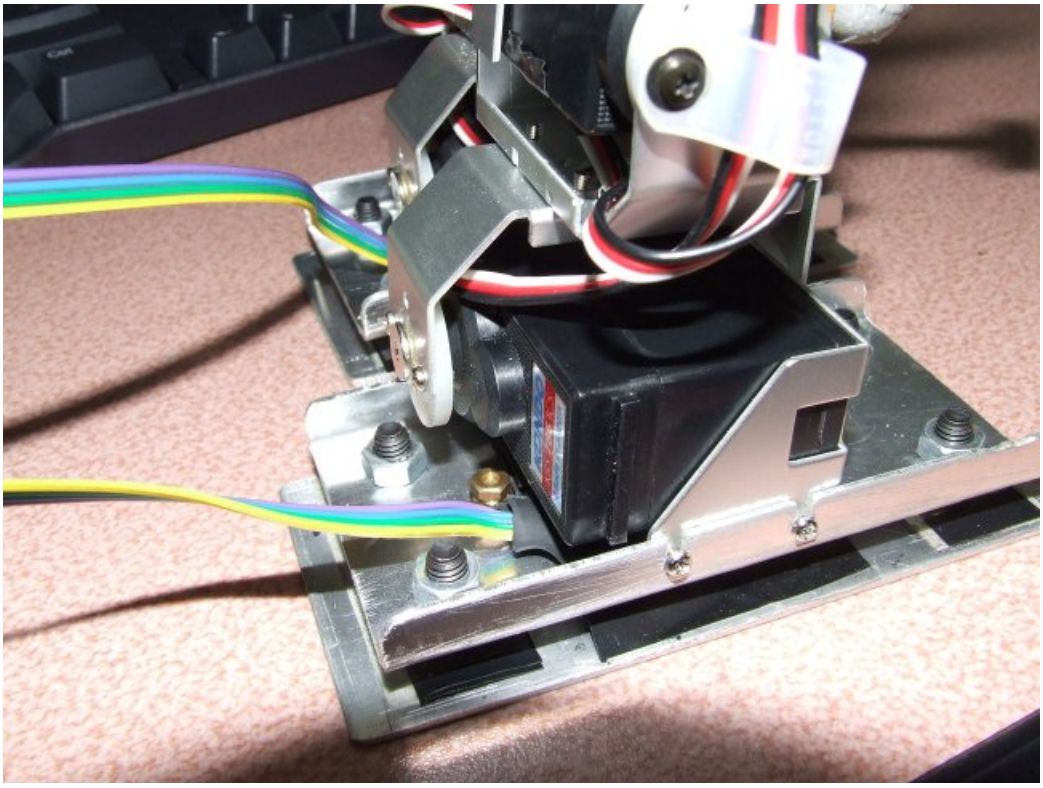


Fig. 12.13 – Semelles vues de derrière

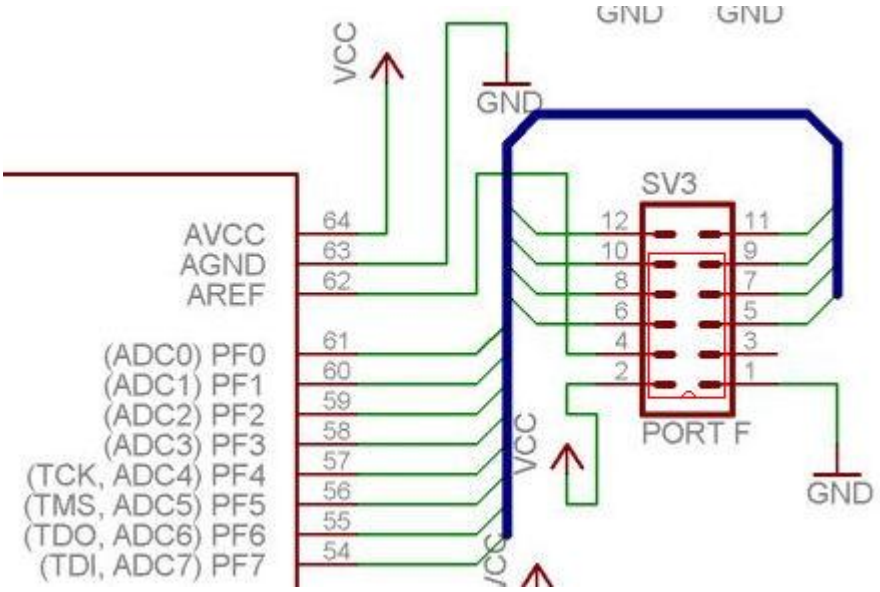


Fig. 12.14 – Branchement capteur CEA

Manuel d'utilisation du logiciel

Ce chapitre a pour but de présenter l'interface logicielle créée pour communiquer aisément avec la carte AtMega128 présentée au chapitre précédent. Afin de mieux comprendre son utilisation, il est conseillé d'avoir un minimum de connaissance du fonctionnement du robot (cartes de contrôle RCB-1, principes de mouvements et de scénarios, ...). La lecture du document [Manuel HeartToHeart] apporte ces notions.
Cette interface logicielle va nous permettre de récupérer les informations des différents capteurs installés sur le robot, mais aussi de communiquer directement avec le robot pour le programmer ou encore obtenir des informations sur l'état de ses moteurs. Le logiciel fournit aussi une interface visuelle pour certains des capteurs.

Fenêtre d'accueil

La fenêtre principale du logiciel est séparée en deux parties : une partie communiquant avec le robot en utilisant l'API (Application Programming interface) des cartes RCB-1 et une partie communiquant avec le robot en utilisant l'API de la carte AtMega128. Ces deux API sont décrites dans le chapitre « Référence de programmation ».

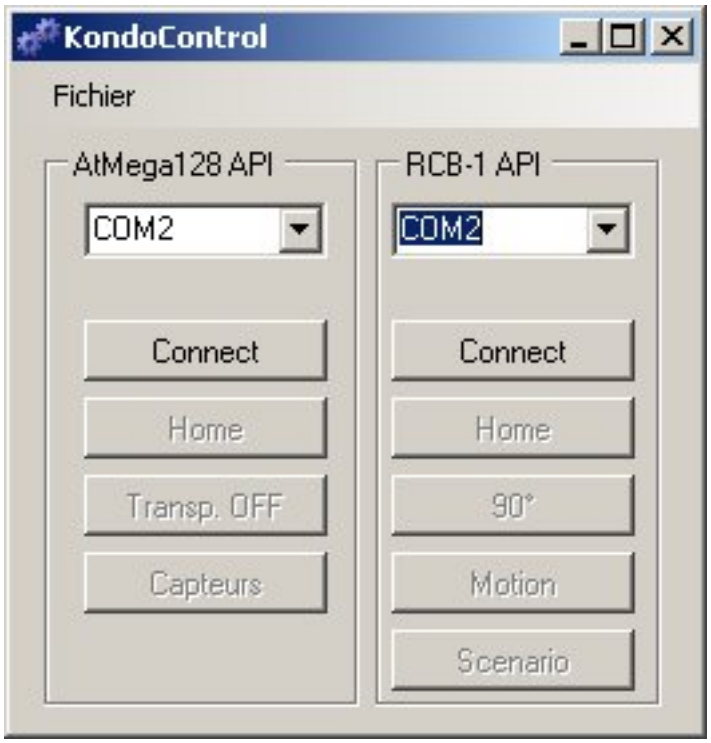


Fig. 12.15 – Fenêtre principale du logiciel

Section AtMega128

Cette section permet la communication avec la carte d'interface AtMega128.

- Le bouton « Connect » permet de se connecter à la carte AtMega128 par l'intermédiaire du port de communication sélectionné dans la liste déroulante située au dessus de ce bouton. Une fois connecté, le bouton est renommé « Disconnect » et permet donc de se déconnecter de la carte.
- Le bouton « Home » permet de placer le robot dans sa position de repos (le microcontrôleur lit la position « Home » stockée dans le robot et l'applique ensuite aux moteurs).
- Le bouton « Transp. OFF » permet de passer la carte en mode transparent. Ce mode renvoit chaque donnée reçue sur le port USART0 de la carte AtMega sur le port USART1 de celle-ci et inversement. Pour désactiver ce mode, il faut cliquer sur le bouton « Transp. ON » qui aura remplacé le bouton « Transp. OFF ».
- Le bouton « Capteurs » permet d'ouvrir la fenêtre de contrôle des capteurs. Cette fenêtre est décrite dans la partie « Fenêtre principale d'interface capteurs ».

12.0.13 Fenêtre principale d'interface capteurs

Cette fenêtre permet de récupérer les informations des capteurs installés sur le robot et la position des moteurs. Une fois capturées, ces données peuvent être enregistrées pour être traitées ou visualisées graphiquement.

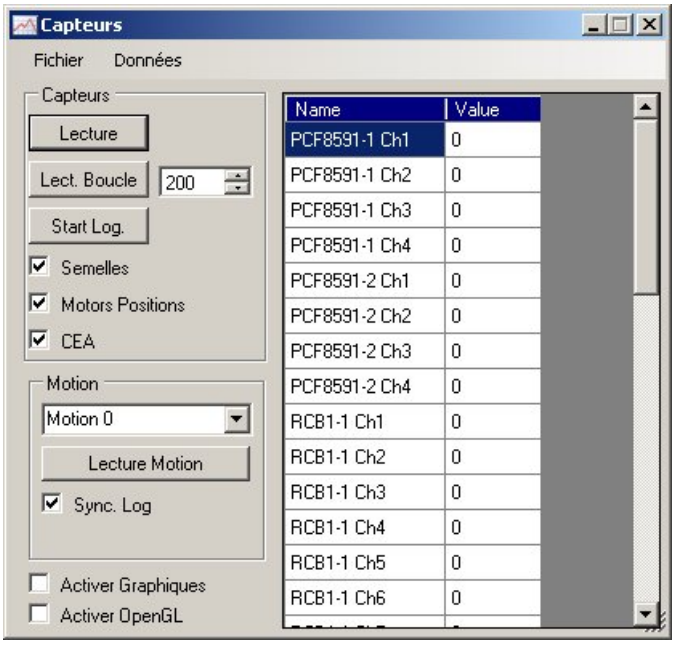


Fig. 12.16 – Fenêtre d'interface des capteurs

Cette fenêtre est composée d'une partie commande et d'une partie visualisation des résultats.

La partie commande est constituée par les sous-ensembles « Capteurs » et « Motion ».

Le sous-ensemble « Capteurs » est constituée de 3 boutons et 3 boîtes à cocher :

- La boîte « I2C » permet d'activer la récupération des valeurs des 8 capteurs situés sous les semelles du robot lors de l'acquisition.
- La boite « Motors Positions » permet d'activer la récupération des positions des moteurs en temps réel lors de l'acquisition.
- La boite « CEA » permet d'activer la récupération des 6 valeurs du capteur CEA lors de l'acquisition.
- Le bouton « Lecture » permet de faire une seule acquisition des capteurs sélectionnés par les boîtes.
- Le bouton « Lect. Boucle » permet de faire une acquisition en boucle à intervalles réguliers (l'intervalle est fixé en millisecondes à droite du bouton « Lect. Boucle »). Le bouton sera alors renommé « Stop » et permettra d'arrêter l'acquisition des données.
- Le bouton « Start Log. » permet de logger les données acquises pour qu'elles puissent ensuite être sauvegardées. Le bouton sera alors renommé en « Stop Log. » et permettra d'arrêter le logging.

Le sous-ensemble « Motion » permet la lecture d'une motion sur le robot. Si la boîte « Sync. Avec motion » est cochée alors les capteurs sélectionnés dans la section « Capteurs » seront acquis et loggés dès que la motion sera lancée.
Pour lancer une motion, il faut sélectionner la motion dans la liste déroulante et cliquer sur le bouton « Lecture Motion ».

La partie visualisation est constituée de la grille de résultats et des deux options « Graphiques » et « OpenGL ». Ces deux options sont décrites ci-après. Les données récupérées lors de chaque acquisition sont affichées dans la grille de résultat. Une fois acquises et loggées, les données peuvent être enregistrées par le biais du menu « Données » puis « Enregistrer ». Ce menu propose alors 3 enregistrements correspondants à chacune des 3 données acquises.

12.0.14 Visualisation graphique des points d'appuis

Une visualisation du point d'appui sur chaque semelle est donnée par les deux graphiques dans la partie inférieure de l'application lorsque la boite « Activer Graphiques » est cochée. Le carré noir représente le point d'appui.

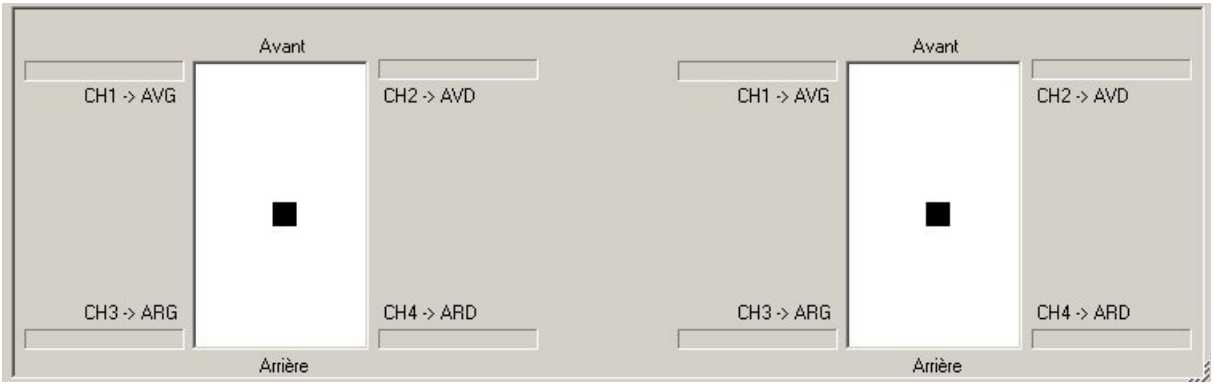


FIG. 12.17 – Visualisation graphique des points d'appuis

12.0.15 Visualisation graphique de l'état robot

Une visualisation 3D de l'état du robot est affichée lorsque la boite « Activer OpenGL » est cochée. La position de chaque moteur est récupérée et appliquée au modèle.

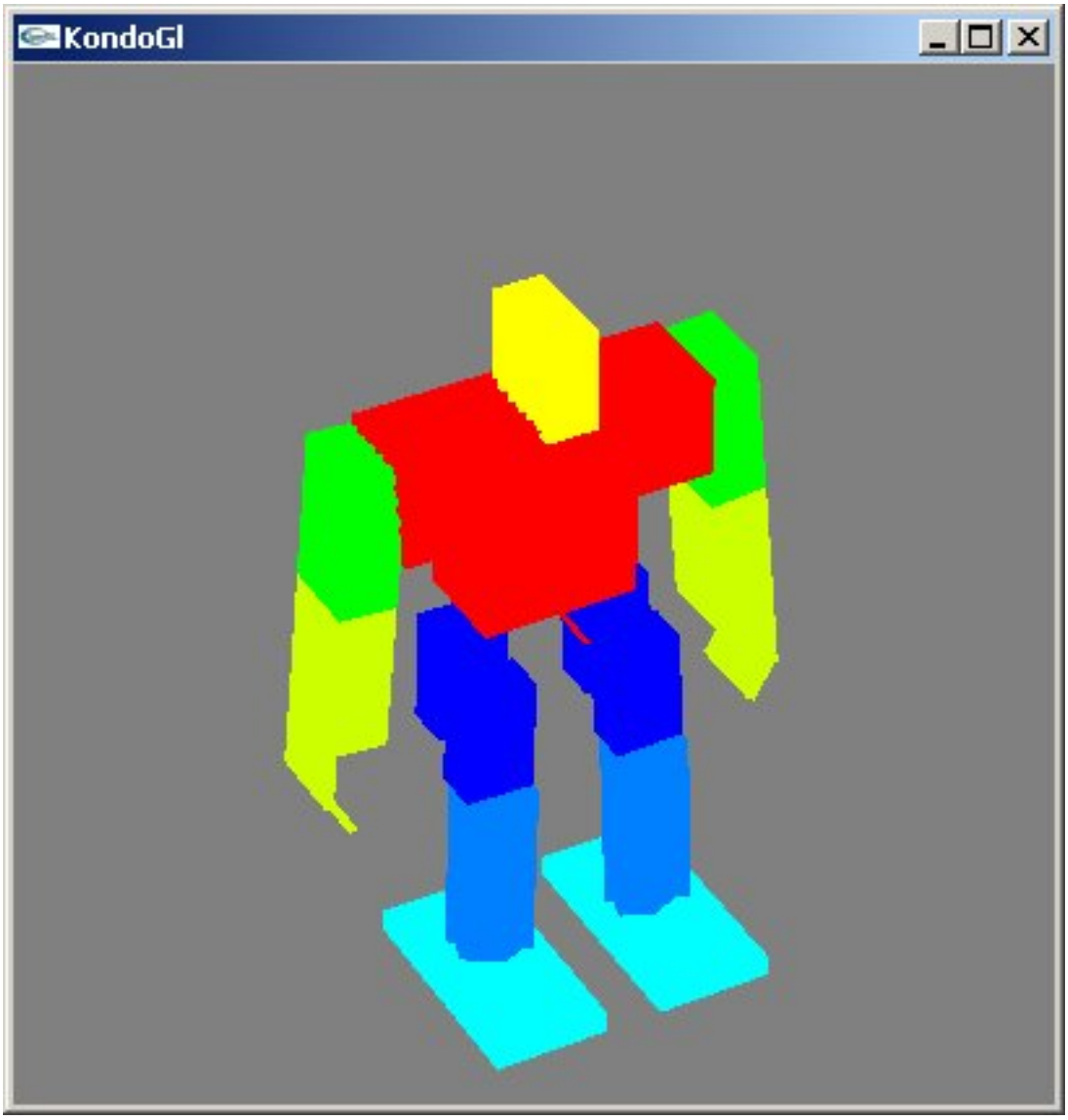


FIG. 12.18 – Visualisation 3D du robot

Section RCB-1

Cette partie permet de controler directement le robot sans utiliser la carte AtMega128 (Voir encadré). Elle reprend les principales fonctions du logiciel HeartToHeart.

- Le bouton « Connect » permet de se connecter au robot par l'intermédiaire du port de communication sélectionné dans la liste déroulante située au dessus de ce bouton. Une fois connecté, le bouton est renommé « Disconnect » et permet donc de se déconnecter du robot.
- Le bouton « Home » permet de placer le robot dans sa position de repos (le logiciel récupère la position « Home » stockée dans le robot et l'applique ensuite aux moteurs).
- Le bouton « 90° » place tous les moteurs à l'angle 90°
- Le bouton « Motion » permet d'éditer les séquences de mouvements (appellées motions) du robot et de les stocker sur les cartes RCB-1. La fenêtre d'édition de motion est décrite dans la partie « Editeur de Motion ».
- Le bouton « Scenario » permet d'éditer les scénarios du robot et de les stocker sur les cartes RCB-1. Un scénario est une suite de motions. La fenêtre d'édition de scénario est décrite dans la partie « Editeur de Scenario ».

Pour utiliser ces fonctions, le robot doit être directement connecté au port COM sélectionné, sinon il sera nécessaire de passer la carte AtMega128 en mode Transparent. Pour passer la carte en mode transparent, il faut se connecter à celle-ci et cliquer sur le bouton « Tranp. ON » puis se déconnecter de la carte.

12.0.16 Editeur de Motion

Cette fenêtre permet d'editer les motions utilisables par le robot, de les stocker dans les cartes RCB-1, de les récupérer à partir de ces cartes pour les modifier ou les enregistrer sur le PC. Elle permet aussi de lire une motion directement à partir du PC sans avoir à la stocker sur le robot.

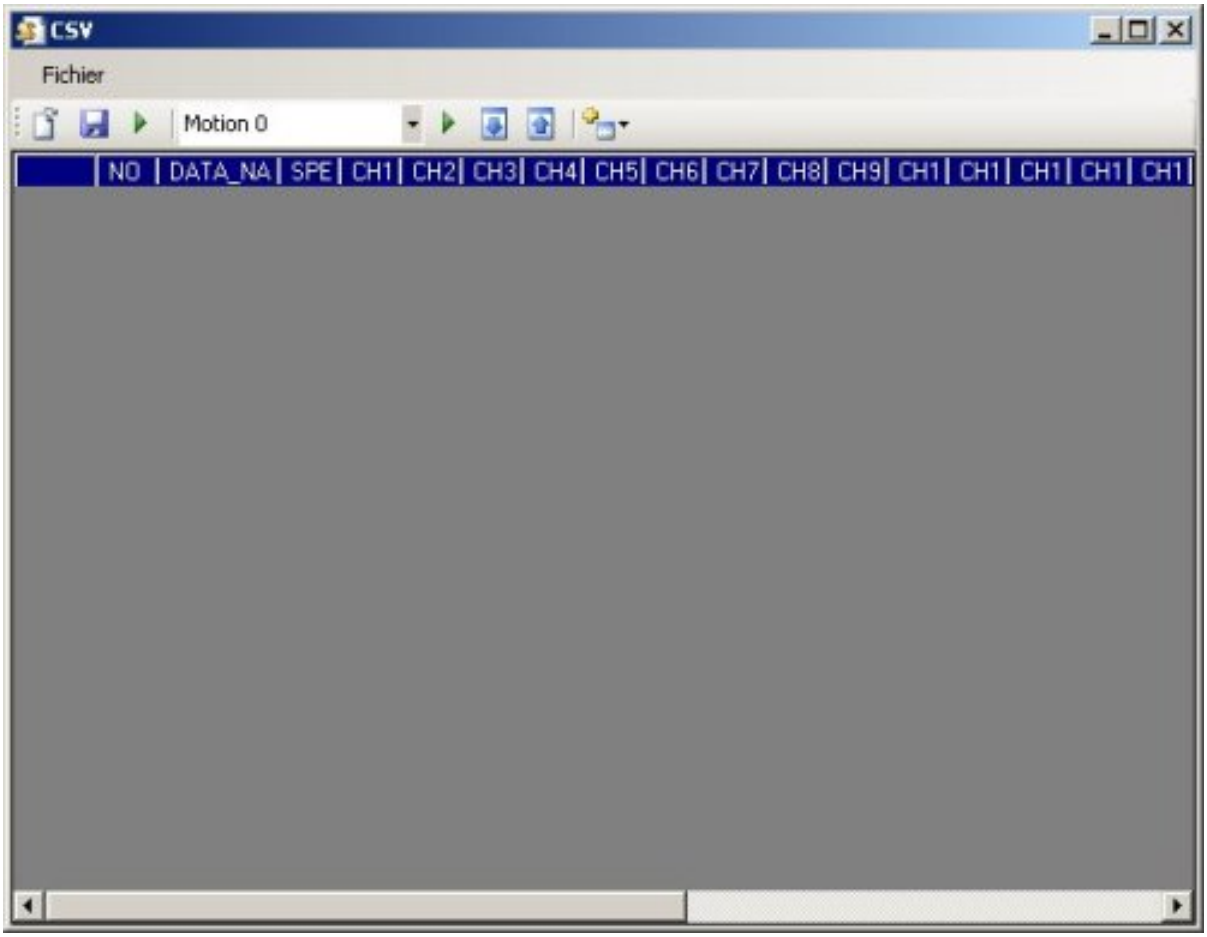
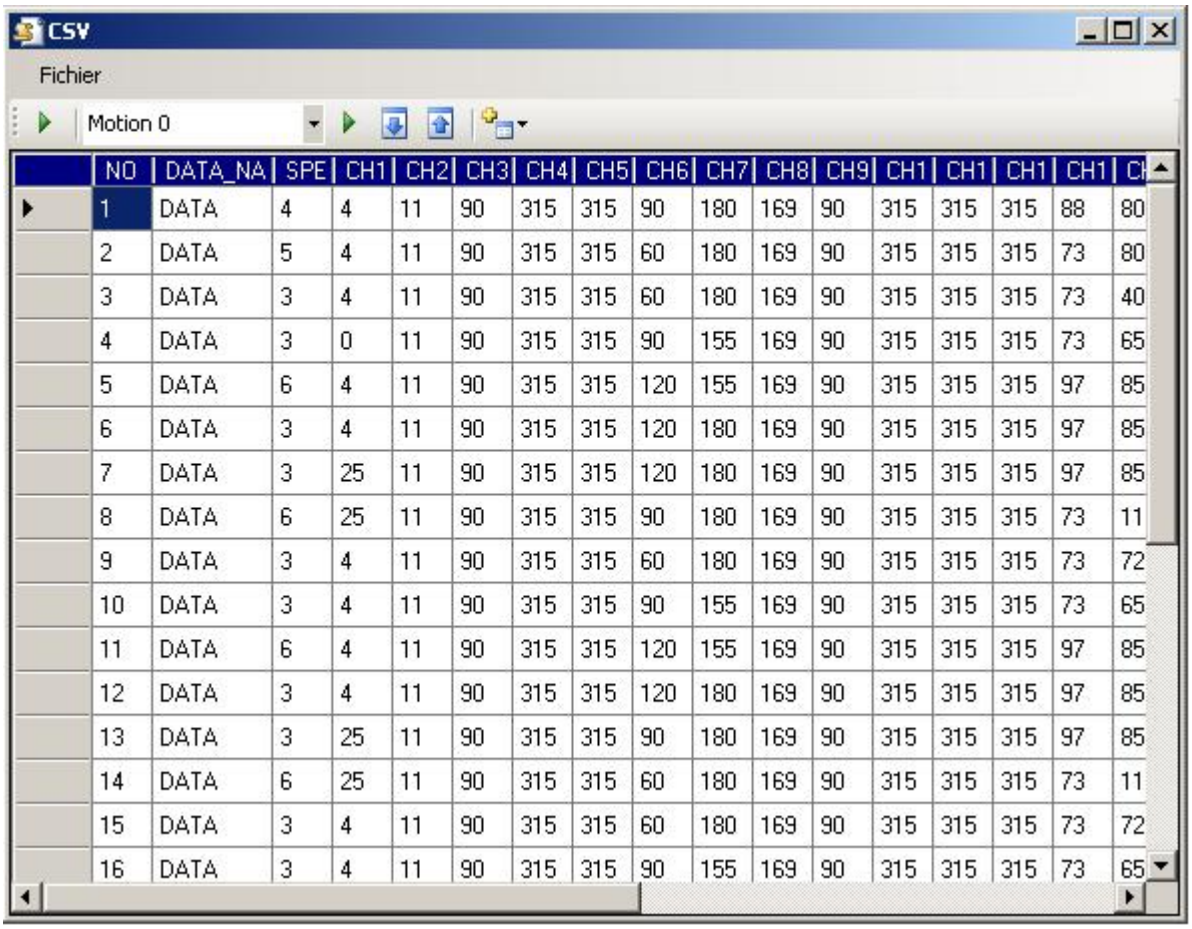


FIG. 12.19 – Fenêtre d'édition de motion

Le logiciel peut charger une motion à partir d'un fichier CSV (en cliquant sur le bouton ou en passant par le menu « Fichier » puis « Ouvrir »), ou en la récupérant dans les cartes RCB-1 (en selectionnant la motion voulue et en cliquant sur).



NO	DATA_NA	SPE	CH1	CH2	CH3	CH4	CH5	CH6	CH7	CH8	CH9	CH1	CH1	CH1	CH1	CH
1	DATA	4	4	11	90	315	315	90	180	169	90	315	315	315	88	80
2	DATA	5	4	11	90	315	315	60	180	169	90	315	315	315	73	80
3	DATA	3	4	11	90	315	315	60	180	169	90	315	315	315	73	40
4	DATA	3	0	11	90	315	315	90	155	169	90	315	315	315	73	65
5	DATA	6	4	11	90	315	315	120	155	169	90	315	315	315	97	85
6	DATA	3	4	11	90	315	315	120	180	169	90	315	315	315	97	85
7	DATA	3	25	11	90	315	315	120	180	169	90	315	315	315	97	85
8	DATA	6	25	11	90	315	315	90	180	169	90	315	315	315	73	11
9	DATA	3	4	11	90	315	315	60	180	169	90	315	315	315	73	72
10	DATA	3	4	11	90	315	315	90	155	169	90	315	315	315	73	65
11	DATA	6	4	11	90	315	315	120	155	169	90	315	315	315	97	85
12	DATA	3	4	11	90	315	315	120	180	169	90	315	315	315	97	85
13	DATA	3	25	11	90	315	315	90	180	169	90	315	315	315	97	85
14	DATA	6	25	11	90	315	315	60	180	169	90	315	315	315	73	11
15	DATA	3	4	11	90	315	315	60	180	169	90	315	315	315	73	72
16	DATA	3	4	11	90	315	315	90	155	169	90	315	315	315	73	65

FIG. 12.20 – Fenêtre d'édition de motion avec motion chargée

La motion en cours d'édition peut être enregistrée dans un fichier CSV (en cliquant sur le bouton ou en passant par le menu « Fichier » puis « Enregistrer Sous »), ou stockée dans les cartes RCB-1 (en cliquant sur le bouton).

Le bouton permet d'ajouter des positions dans le mouvement en cours d'édition. Trois modes d'ajout sont proposés :

- Manuel : Chaque valeur de moteur est reglée manuellement. Le résultat peut-être visualisé directement sur le robot en activant l'option « Synchronisation ».
- Capture Once : Le logiciel capture l'état courant des moteurs. La vitesse peut-être sélectionnee dans le menu.
- Motion Capture : Le logiciel récupère la valeur des moteurs continuellement à intervalle donné. L'intervalle peut-être réglé en millisecondes dans le sous menu Fréquence.

La motion en cours d'édition peut être prévisualisée sur le robot en utilisant le premier bou-

ton  situé à gauche. Une motion stockée dans les cartes RCB-1 peut être lue en sélectionnant cette motion dans la liste déroulante et en cliquant sur le bouton  situé à droite de la liste.

12.0.17 Editeur de Scenario

Cette fenêtre permet d'éditer les scénarios utilisables par le robot, de les stocker dans les cartes RCB-1 ou de les récupérer à partir de ces cartes pour les modifier.

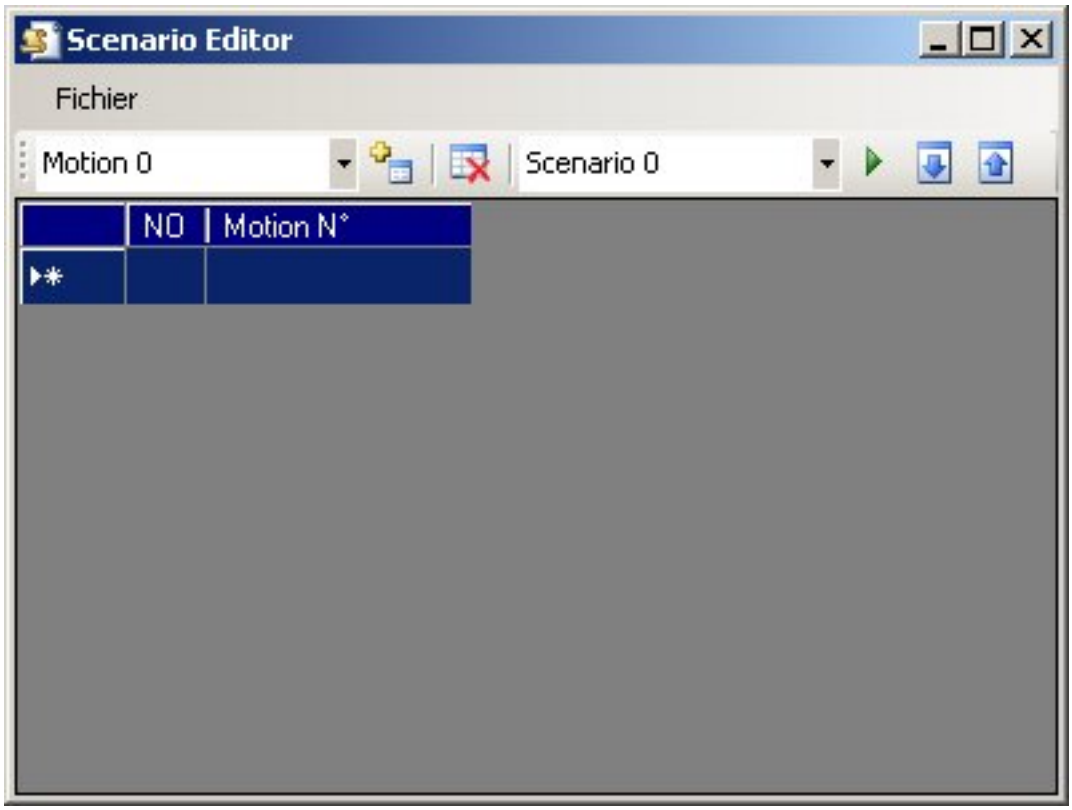


FIG. 12.21 – Fenêtre d'édition de scenario

Pour éditer un scenario, le logiciel peut charger un scenario précédemment stocké dans les cartes RCB-1 (en selectionnant le scenario voulu dans la liste déroulant et en cliquant sur ), ou l'éditer à partir d'un scenario vide.

Le bouton  permet d'ajouter la motion sélectionnée dans la liste déroulante de gauche dans le scenario en cours d'édition.

Le scenario en cours d'édition peut être stockée dans les cartes RCB-1 en cliquant sur le bouton.  .

Référence de programmation

Ce chapitre à pour but de présenter la communication entre le PC, le microcontrôleur et le robot afin que l'utilisateur puisse facilement ajouter ses propres fonctions à la carte. Ainsi, nous verrons la communication entre le PC et la carte AtMega128, la communication entre la carte AtMega128 ou le PC et le Robot.

Architecture du programme

Le diagramme ci-dessous présente l'architecture de programmation de la carte d'extension.

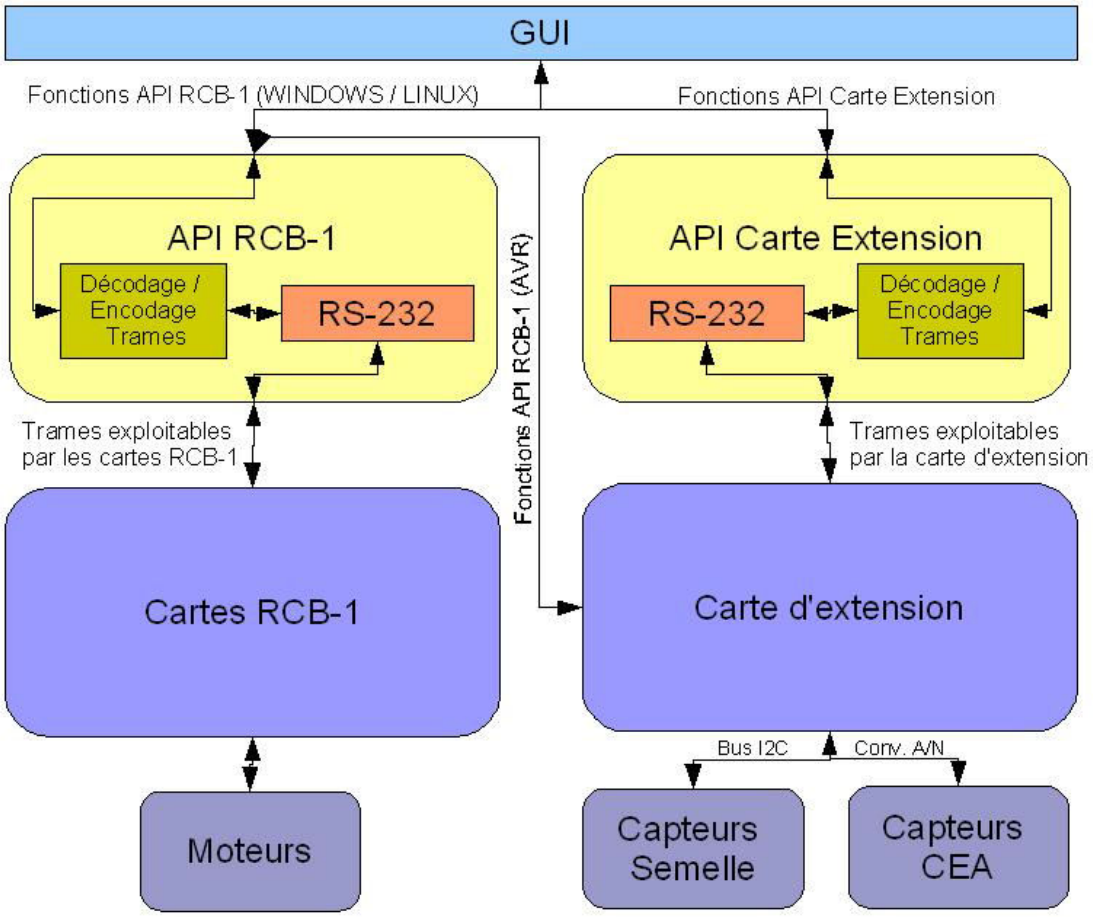


Fig. 12.22 – Architecture de programmation

Les deux principaux composants sont l'API RCB-1 et l'API de la carte d'extension.

L'API RCB-1 permet la communicatin avec les cartes RCB-1 installées sur le Robot. Elle est utilisable sur différentes plateformes (à l'heure actuelle, Windows, Linux et microcontrôleur AVR) et est portable sur toute autre plateforme supportant les communication séries RS-232. Seule les fonctions d'envoi et de réception sur port série sont alors à créer pour la plateforme utilisée. Cette API est décrite dans la section « Communication avec le Robot ».

L'API de la carte d'extension permet la communication avec la carte d'extention. Cette carte embarque l'API des cartes RCB-1 permettant à celle-ci de communiquer avec le robot. L'API de la carte d'extension est actuellement utilisable seulement sur plateforme windows

mais peut aussi être facilement portée sur autres plateformes en créant les fonctions d'envoi et de réception sur port série pour la plateforme utilisée. Cette API est décrite dans la section suivante.

La carte d'extension embarquant l'API RCB-1, il est possible de communiquer avec le robot sans intervenir sur le PC. Ainsi, on peut facilement intégrer un asservissement sur cette carte. La communication avec les capteurs est réalisée par le microcontrôleur. Seules les informations utiles sont renvoyée au PC par liaison RS-232.

L'API RCB-1 est constituée par les fichiers KondoApi.h et KondoApi.c . L'API de la carte d'extension est constituée par les fichiers ATmegaApi.h et ATmegaApi.c .

Communication PC - Carte AtMega128

La communication entre le PC et le robot se fait par trames envoyées sur la liaison série. Ces trames ont un format prédéfini qui permet un traitement simple et rapide. Aussi, ce format permet d'ajouter facilement de nouvelles fonctions à la carte tels qu'ajout de capteurs, traitement arithmetiques, asservissements, ...

Le port série est configuré avec les paramètres suivants :
- Vitesse de communication : 115000 Bauds
- Bits de données : 8
- Parité : Aucune
- Bit de stop : 1
- Contrôle de flux : Aucun

12.0.18 Format générique de trame

Format générique de trame de communication envoyée par le PC à la carte AtMega128 (hors trames pour Mode Tranparent) :

[TX]
CMDP CMD_LENGTH CMD1 CMD2 ... CMD_LAST CHKSUM

Paramètres :

Paramètre	Valeur	Description
CMDP	DDh	Commande AtMega128 API
CMD_LENGTH	00h - FFh	Longeur Commande *
CMD1 - CMD_LAST	00h - FFh	Paramètres de la commande
CHKSUM **	00h - 7Fh	(CMDP + CMD_LENGTH + CMD1 + ... + CMD_LAST) & 7Fh

* La longeur de la commande est le nombre de bits envoyés - 2
** & est un ET logique, + est une addition
exemple de chksum : (FFh + 5Ah + 2h) & 7f = 5Bh

12.0.19 Mode Transparent ON

Cette trame permet de passer la carte en mode transparent. La carte emet une trame d'acquitement pour confirmer la bonne exécution de la commande. Pour repasser la carte en mode normal, il faudra alors envoyer la trame « Mode Transparent OFF » décrite ci-après ou redémarrer la carte.

[TX]
CMD

[RX]
CMD ACK

Paramètres :

Paramètre	Valeur	Description
CMD	DAh	Commande Mode Transparent ON
ACK	06h	Acquittement

12.0.20 Mode Transparent OFF

Cette trame permet de passer la carte en mode normal. La carte emet une trame d'acquitement pour confirmer la bonne exécution de la commande.

[TX]
CMD

[RX]
CMD ACK

Paramètres :

Paramètre	Valeur	Description
CMD	DBh	Commande Mode Transparent OFF
ACK	06h	Acquittement

12.0.21 Statut Mode Transparent

Cette trame permet de connaitre l'état du mode Transparent de la carte. La carte emet une trame donnant l'état de ce mode.

[TX]
CMD

[RX]
STA

Paramètres :

Paramètre	Valeur	Description
CMD	DCh	Commande Statut Mode Transparent
STA	DAh - DBh	Statut (DA = ON, DB = 0FF)

12.0.22 Lecture valeurs PCF8591 sur I2C

Cette trame permet de récupérer les 4 valeurs de chacun des 2 convertisseurs Analogiques Numériques 4 voies PCF8591 branchés sur le bus I2C.

[TX]
CMDP 2 CMD CHKSUM

[RX]
V1 V2 V3 V4 V5 V6 V7 V8 CHKSUM

Paramètres :

Paramètre	Valeur	Description
CMDP	DDh	Commande AtMega128 API
CMD	DEh	Commande
TX CHKSUM	00h - 7Fh	(CMDP + 2 + CMD) & 7Fh
V1 - V8	00h - FFh	Valeurs des 8 voies converties sur 8bit
RX CHKSUM	00h - 7Fh	(V1 + V2 + ... + V8) & 7Fh

12.0.23 Mettre robot en position Home

Cette trame permet de passer le robot dans sa position Home.

[TX]
CMDP 2 CMD CHKSUM

[RX]
CMD ACK

Paramètres :

Paramètre	Valeur	Description
CMDP	DDh	Commande AtMega128 API
CMD	DFh	Commande
CHKSUM	00h - 7Fh	(CMDP + 2 + CMD) & 7Fh

12.0.24 Lire les positions moteurs du robot

Cette trame permet de lire les positions des 24 moteurs à partir des cartes RCB-1 du robot.

[TX]
CMDP 2 CMD CHKSUM

[RX]
CH1 CH2 ... CH24 CHKSUM

Paramètres :

Paramètre	Valeur	Description
CMDP	DDh	Commande AtMega128 API
CMD	E0h	Commande
TX CHKSUM	00h - 7Fh	(CMDP + 2 + CMD) & 7Fh
CH1 - CH24	00h - 5Ah	Angle des 24 moteurs (00h = 0°, 5Ah = 180°)
RX CHKSUM	00h - 7Fh	(CH1 + CH2 + ... + CH24) & 7Fh

12.0.25 Lire une motion du robot

Cette trame permet de lire une motion stockée sur les cartes RCB-1 du robot.

[TX]
CMDP 3 CMD MOTION# CHKSUM

[RX]
CMD ACK

Paramètres :

Paramètre	Valeur	Description
CMDP	DDh	Commande AtMega128 API
CMD	E1h	Commande
MOTION#	00h-27h	Motion à lire
CHKSUM	00h - 7Fh	(CMDP + 3 + CMD + MOTION#) & 7Fh

Communication avec le Robot

La communication avec le robot se fait par liaison série paramétrée comme suit :
- Vitesse de communication : 115000 Bauds
- Bits de données : 8
- Parité : Aucune
- Bit de stop : 1
- Contrôle de flux : Aucun

Le PC communique alors avec les deux cartes RCB-1 installées sur le Robot. Chacune des cartes est identifiée par une ID. Pour communiquer avec ces cartes, nous réutilisons les trames décrites dans le document « RCB-1 Command Reference » disponible sur le site http://www.robosaavy.com. L'api de communication avec les cartes RCB-1 à été réécrite indépendemment du reste du programme afin de pouvoir être utilisée avec d'autres programmes mais aussi sur différentes plateformes (Windows, Linux, AVR, ...).

Bibliographie

[Manuel HeartToHeart] Kondo, *HeartToHeart Operations Manual*, Kondo Kagaku CO. LTD, 2004

Septième partie

Annexe B : Guide de programmation AVR

Pré-Requis

Les programmes utilisés pour ce manuel requièrent :
- Windows 98/NT/2000/XP
- Pack d'outils WinAVR (http://winavr.sourceforge.net/)
- Pack de librairies AvrLib (http://hubbard.engr.scu.edu/avr/avrlib/)
- Atmel AVR Studio 4 (http://www.atmel.com/)
- SP12 AVR Programmer (http://www.xs4all.nl/~sbolt/e-spider_prog.html)

La programmation du microcontrôleur requiert un programmateur parallèle dont le schéma est donné en annexe A.

Installation des outils

Ordre d'installation

L'installation doit se faire de préférence dans l'ordre donné ci-dessous afin que les librairies et outils soient automatiquement reconnus pas Atmel AVR Studio.

- Pack d'outils WinAVR
- Pack de librairies AvrLib
- SP12 AVR Programmer
- Atmel AVR Studio 4

Installation

Lors de l'installation, regrouper tous les outils dans un même répertoire (ex : c :\AVR) Ainsi, lors de la configuration, il est plus aisé de retrouver les différents composants.

L'installation des outils se fait de façon automatique et ne requiert que le réglage du répertoire d'installation pour chacun des outils. SP12 requiert le redémarrage impératif de l'ordinateur pour permettre l'installation d'un pilote d'accès au port parallèle sous Windows 2000/NT/XP.

Création d'un projet

Lorsque l'installation des logiciels set terminée, vous pouvez alors utiliser AVR Studio pour commencer un nouveau projet.

12.0.26 Lancement de AVR Studio

Au lancement de AVR Studio, vous obtenez la fenêtre de la figure 2.1.

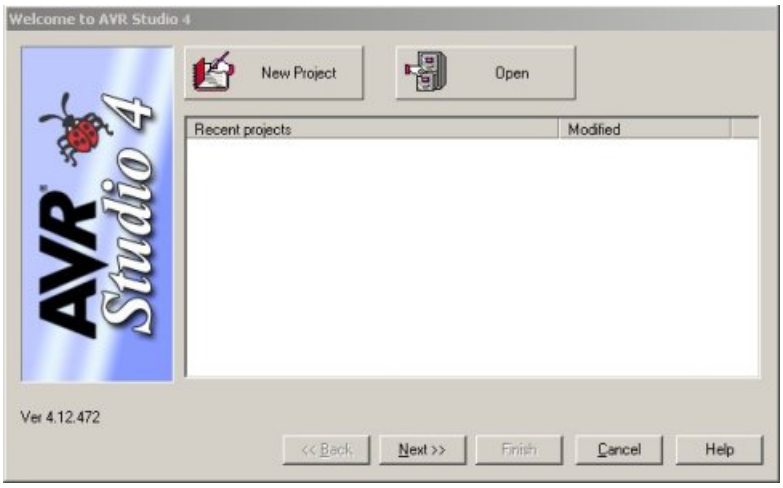


FIG. 12.23 – Fenêtre d'accueil de AVR Studio

Lorsque vous aurez créé un projet, celui-ci apparaitra dans cette fenêtre au lancement et pourra ainsi facilement être rechargé.

12.0.27 Création d'un nouveau projet

Cliquez sur 'New Project' pour démarrer un nouveau projet. Vous obtiendrez alors la fenêtre suivante :

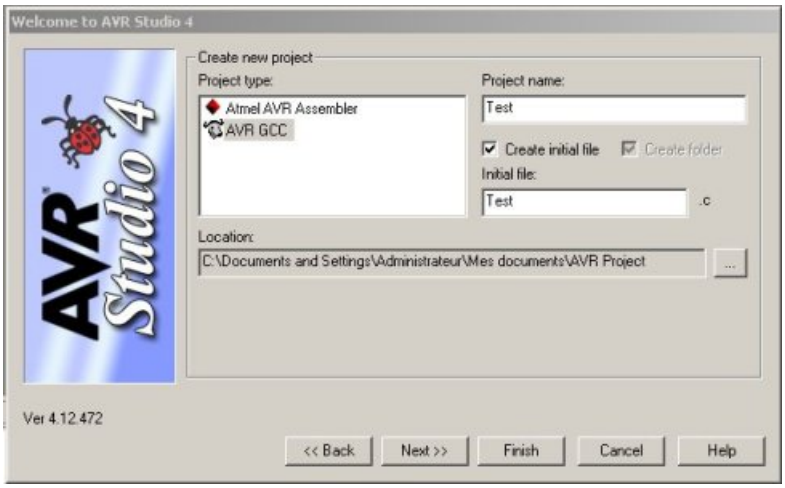


FIG. 12.24 – Sélection du type de projet

Sélectionnez 'AVR GCC' dans la liste 'Project Type'. Remplissez le champ 'Project name' et choisissez le répertoire du projet dans le champ 'Location'. Une fois ces renseignements complétés, cliquez sur 'Next'.

Vous obtiendrez la fenêtre suivante :

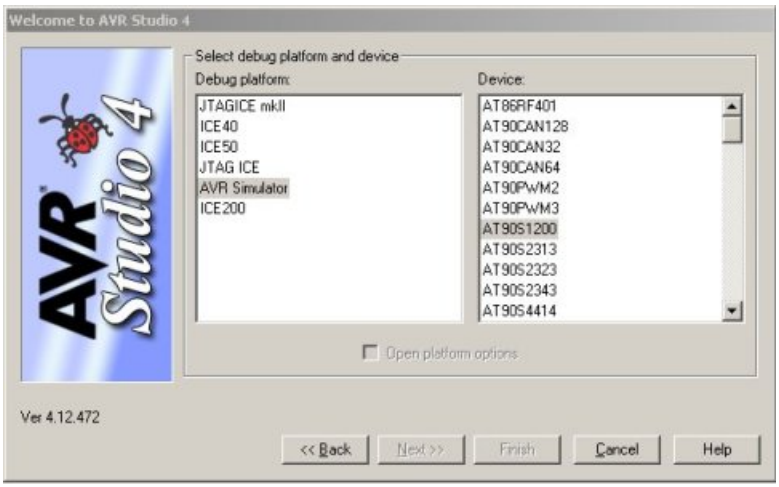


FIG. 12.25 – Sélection de la plateforme de déboguage

Dans cette fenêtre, sélectionnez 'AVR Simulator' dans la liste 'Debug Platform'. Dans la liste de droite, sélectionnez 'ATmega128' et cliquez sur le bouton 'Finish'.

Vous obtiendrez la fenêtre suivante :

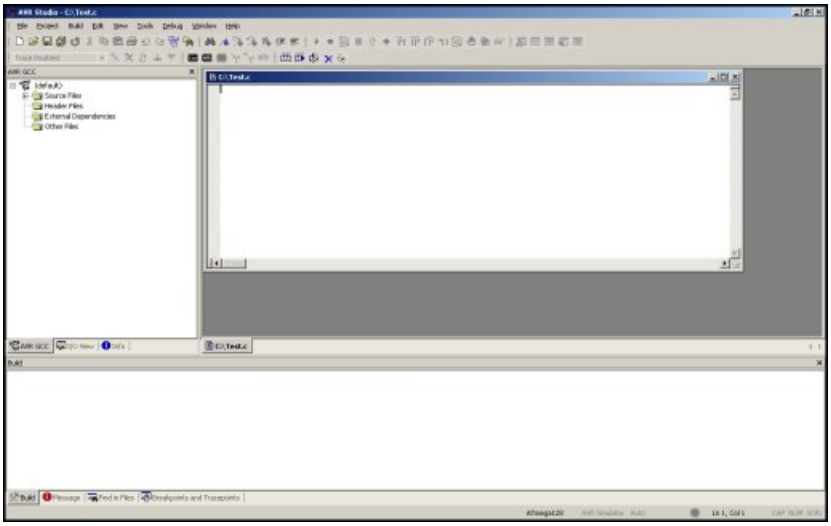


FIG. 12.26 – Fenêtre principale de AVR Studio

La création du projet est maintenant terminée. Pour tester les outils de compilations, entrez les lignes suivantes dans le fichier c :

```
int main(void)
{
    return 0;
}
```

Pour compiler ce programme, appuyez sur la touche F7 ou allez dans le menu 'Build' puis 'Build'. Si les outils sont biens intallés, la fenêtre 'Build' doit afficher ceci :

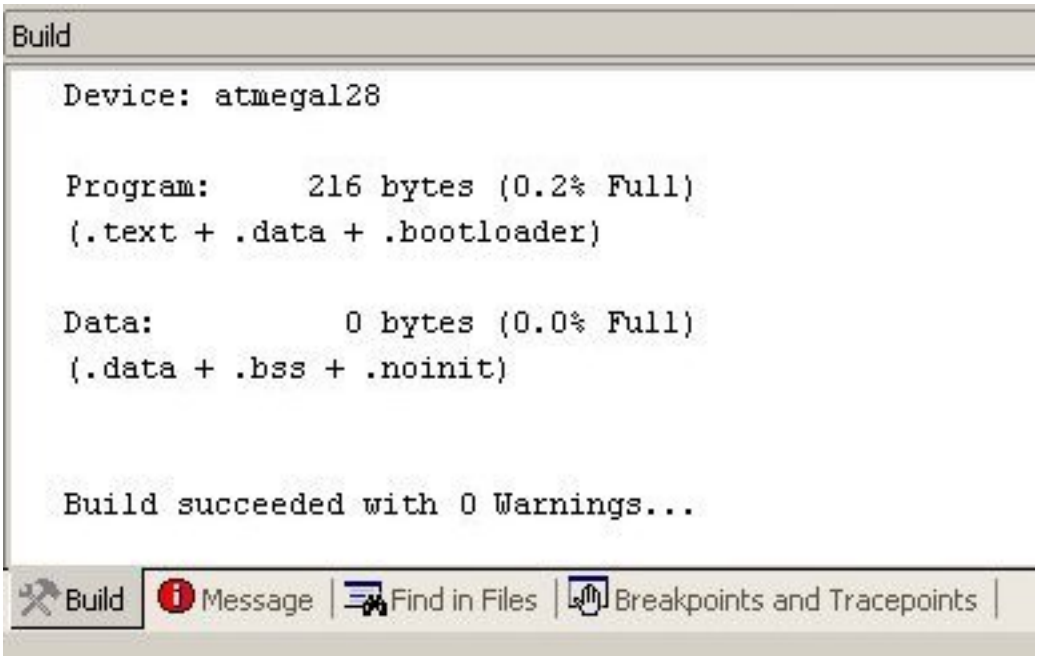


FIG. 12.27 – Log de la compilation

Ce log de compilation résume les principales informations du programme (taille, pourcentage d'utilisation de la mémoire, erreurs, avertissements, ...). Les erreurs de compilation y seront affichées de même que les avertissements.

Programmation du microcontrôleur

Lorsque le programme est complètement réalisé, il peut être programmé sur le microcontrôleur. Lors de la première programmation du microcontrôleur AtMega128, il faut régler les fusibles internes afin de permettre au microcontrôleur d'exécuter le programme.

12.0.28 Paramétrage des fusibles internes

Les paramètres des fusibles internes sont à déterminer à partir de la documentation technique du microcontrôleur ATmega128. Se référer au tableaux ci dessous reprenant ces réglages :

Lock Fuse	Byte Bit No.	Description	Default Value
BLB12	5	Boot lock bit	1 (unprogrammed)
BLB11	4	Boot lock bit	1 (unprogrammed)
BLB02	3	Boot lock bit	1 (unprogrammed)
BLB01	2	Boot lock bit	1 (unprogrammed)
LB2	1	Lock bit	1 (unprogrammed)
LB1	0	Lock bit	1 (unprogrammed)

Tab. 4.1 - Paramètres Lock Fuses

Low Fuse	Byte Bit No.	Description	Default Value
BODLEVEL	7	Brown out detector trigger level	1 (unprogrammed)
BODEN	6	Brown out detector enable	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	1 (unprogrammed)
SUT0	4	Select start-up time	0 (programmed)
CKSEL3	3	Select Clock source	0 (programmed)
CKSEL2	2	Select Clock source	0 (programmed)
CKSEL1	1	Select Clock source	0 (programmed)
CKSEL0	0	Select Clock source	1 (unprogrammed)

Tab. 4.2 - Paramètres Low Fuses

High Fuse	Byte Bit No.	Description	Default Value
OCDEN(4)	7	Enable OCD	1 (unprogrammed, OCD disabled)
JTAGEN(5)	6	Enable JTAG	0 (programmed, JTAG enabled)
SPIEN(1)	5	Enable Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)
CKOPT(2)	4	Oscillator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size	0 (programmed)
BOOTSZ0	1	Select Boot Size	0 (programmed)
BOOTRST	0	Select Reset Vector	1 (unprogrammed)

Tab. 4.3 - Paramètres High Fuses

Extended Fuse	Byte Bit No.	Description	Default Value
-	7	-	1
-	6	-	1
-	5	-	1
-	4	-	1
-	3	-	1
-	2	-	1
M103C	1	ATmega103 compatibility mode	0 (programmed)
WDTON	0	Watchdog Timer always on	1 (unprogrammed)

Tab. 4.4 - Paramètres Extended Fuses

Les paramètres des fusibles à appliquer dans le cas d'une application générale sont :
- Lock Fuses : L[111111]
- Low Fuses : F[11101111]
- High Fuses : H[11011001]
- Extended fuses : X[11111111]

Pour appliquer ces paramètre, il faut utiliser SP12. Ouvrir une invite de commande puis aller dans le répertoire de SP12.
Executer la commande :

sp12 -wL111111 -wF11101111 -wH11011001 -wX0xFF

Cette commande va paramétrer les fusibles du microcontroleur et permettre l'exécution des programmes.

12.0.29 Programmation des mémoires Flash et EEPROM

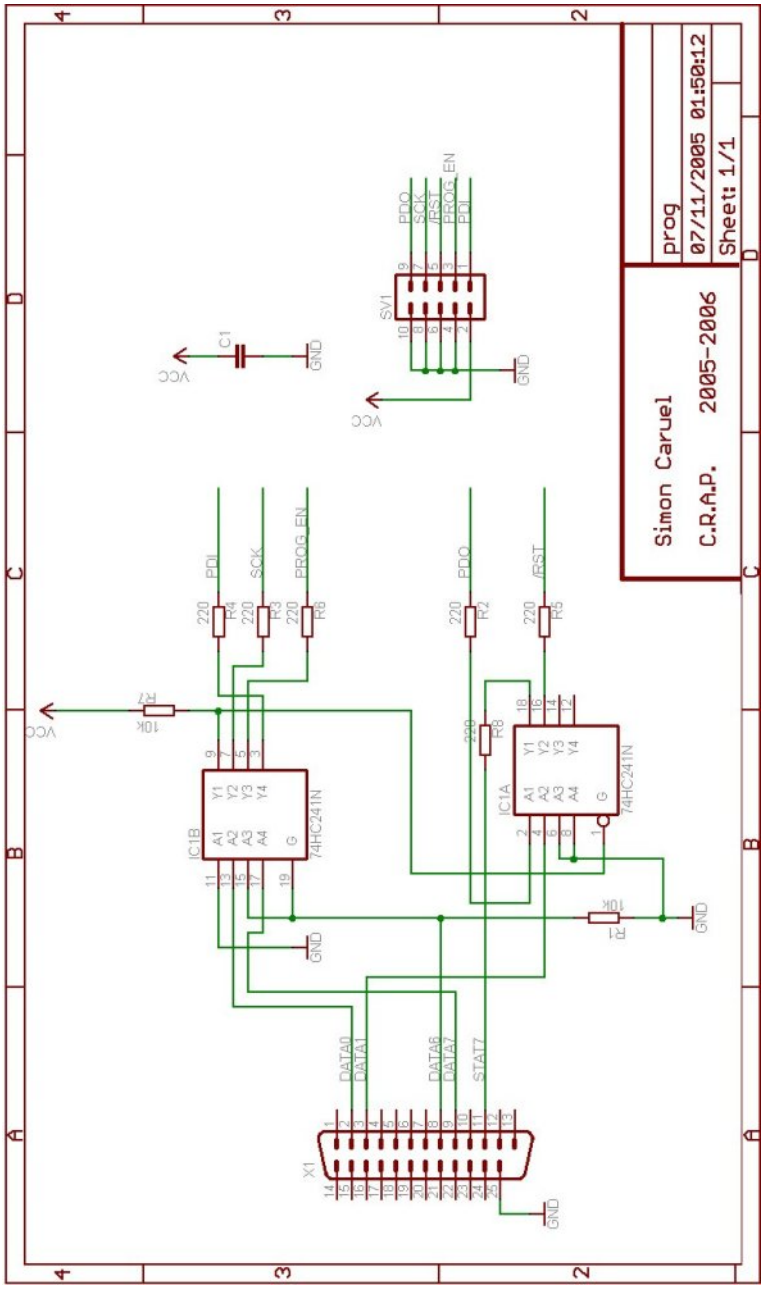
Une fois le programme achevé et compilé, nous obtenons deux fichiers qui serviront à programmer le microcontrôleur. Ces fichiers sont situés dans le sous-répertoire 'Default' du répertoire du projet. Ces deux fichiers sont les fichiers %NomduProjet%.hex et %NomduProjet%.eep.

Le fichier %NomduProjet%.hex est la partie Flash du programme du microcontrôleur. Le fichier %NomduProjet%.eep est la partie EEPROM du programme du microcontrôleur.

La programmation se fait par SP12. Ouvrir une invite de commande puis aller dans le répertoire de SP12.
Executer les commandes :

sp12 -wpf %RepertoireDuProjet%\Default\%NomduProjet%.hex
sp12 -wef %RepertoireDuProjet%\Default\%NomduProjet%.eep

Schéma du programmeur SP12



Huitième partie

Annexe C : Documentation API Carte RCB-1

Api_Kondo Class Index

Api_Kondo Class List

Here are the classes, structs, unions and interfaces with brief descriptions :
– defmotor_t

Api_Kondo File Index

Api_Kondo File List

Here is a list of all files with brief descriptions :
- API Kondo : Kondo Api.cpp
- API Kondo : Kondo Api.h

Api_Kondo Class Documentation

defmotor_t Struct Reference

`#include <KondoApi.h>`

Public Attributes

- int **ch1**
- **int ch2**
- **int ch3**
- **int ch4**
- **int ch5**
- **int ch6**
- **int ch7**
- **int ch8**
- **int ch9**
- **int ch10**
- **int ch11**
- **int ch12**

Detailed Description

Definition at line 120 of file KondoApi.h.

Member Data Documentation

int defmotor_t : :ch1

Definition at line 121 of file KondoApi.h.

Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch10

Definition at line 121 of file KondoApi.h.
Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch11

Definition at line 121 of file KondoApi.h.
Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch12

Definition at line 121 of file KondoApi.h.
Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch2

Definition at line 121 of file KondoApi.h.
Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch3

Definition at line 121 of file KondoApi.h.
Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch4

Definition at line 121 of file KondoApi.h.
Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch5

Definition at line 121 of file KondoApi.h.

Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch6

Definition at line 121 of file KondoApi.h.

Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch7

Definition at line 121 of file KondoApi.h.

Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch8

Definition at line 121 of file KondoApi.h.

Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

int defmotor_t : :ch9

Definition at line 121 of file KondoApi.h.

Referenced by RCB_GetHomePositions(), RCB_GetMotionPosition(), RCB_GetMotorPositions(), RCB_GetTrimPositions(), RCB_SetMotionPosition(), RCB_SetMotorPositions(), RCB_SetTrimPositions(), and Set90().

The documentation for this struct was generated from the following file :

– Bureau/INRIA/FINAL/Doc/API_Kondo/**KondoApi.h**

Api_Kondo File Documentation

KondoApi.cpp File Reference

```
#include "KondoApi.h"
```

Functions

- int **RCB_GetID (int *reply)**
- **int RCB_GetSwitchBits (int id, int *reply)**
- **int RCB_SetSwitchBits (int id, int switchValue)**
- **int RCB_GetHomePositions (int id, motors *motorsdata)**
- **int RCB_SetHomePositions (int id)**
- **int RCB_GetMotorPositions (int id, motors *motorsdata)**
- **int RCB_SetMotorPositions (int id, int speed, motors motorsdata)**
- **int RCB_GetTrimPositions (int id, motors *motorsdata)**
- **int RCB_SetTrimPositions (int id, motors motorsdata)**
- **int RCB_GetMotionPosition (int id, int motion, int position, motors *motorsdata, int *speed)**
- **int RCB_SetMotionPosition (int id, int motion, int posNumber, int speed, motors motorsdata)**
- **int RCB_GetMotionPositionCount (int id, int motion, int *reply)**
- **int RCB_SetMotionPositionCount (int id, int motion, int posNumber)**
- **int RCB_PlayMotion (int id, int motion)**
- **int RCB_PlayScenario (int id, int scenario)**
- **int RCB_GetScenarioMotion (int id, int scenario, int motion, int *reply)**
- **int RCB_SetScenarioMotion (int id, int scenario, int motionIndex, int motion)**
- **int RCB_GetScenarioMotionCount (int id, int scenario, int *reply)**
- **int RCB_SetScenarioMotionCount (int id, int scenario, int motionNumber)**
- **int SetHome (void)**
- **int Set90 (void)**

Function Documentation

• **int RCB_GetHomePositions (int *id*, motors * *motorsdata*)**

Routine to get the Home Position

Parameters :

← ***id*** Specify Board ID

→ ∗***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 555 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().

Referenced by SetHome().

• **int RCB_GetID (int * *reply*)**

Specific LINUX declarations :
TODO : Communication functions, uint8_t definition, print() and print_uint8() functions
Definition at line 478 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetMotionPosition (int *id*, int *motion*, int *position*, motors * *motorsdata*, int * *speed*)**

Routine to get the Motors Position of the Motion Position Index

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

← ***position*** Specify the motion position index

→ ∗***motorsdata*** Stores Motors Position Values

→ ∗***speed*** Stores Motors Speed Value

Returns : 1 : successful completion

0 : error

Definition at line 870 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().

• **int RCB_GetMotionPositionCount (int *id*, int *motion*, int * *reply*)**

Routine to get the number of positions in the motion

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

→ ***reply*** Store the number of positions in the motion

Returns : 1 : successful completion

0 : error

Definition at line 942 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetMotorPositions (int *id*, motors * *motorsdata*)**

Routine to get the Motors Actual Position

Parameters :

← ***id*** Specify Board ID

→ ∗***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 661 of file KondoApi.cpp.
References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().

• **int RCB_GetScenarioMotion (int *id*, int *scenario*, int *motion*, int * *reply*)**

Routine to get the motion Index of the scenario's motion number motion

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario number

← ***motion*** Specifiy the motion number

→ ***reply*** Store the motion index

Returns : 1 : successful completion

0 : error

Definition at line 1007 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetScenarioMotionCount (int *id*, int *scenario*, int * *reply*)**

Routine to get the number of motions in the scenario

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario

→ ***reply*** Store the number of motions in the scenario

Returns : 1 : successful completion

0 : error

Definition at line 1047 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetSwitchBits (int *id*, int * *reply*)**

Routine to get the Board Software Switch Bits

Parameters :

← ***id*** Specify Board ID

→ ***reply*** Board Software Switch Value

Returns : 1 : successful completion

0 : error

Definition at line 513 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetTrimPositions (int *id*, motors * *motorsdata*)**

Routine to get the Motors Trim Position

Parameters :

← ***id*** Specify Board ID

→ ∗***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 767 of file KondoApi.cpp.
References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().

• **int RCB_PlayMotion (int *id*, int *motion*)**

Routine to play a motion

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number to play

Returns : 1 : successful completion

0 : error

Definition at line 981 of file KondoApi.cpp.
References SendSetCom().

• **int RCB_PlayScenario (int *id*, int *scenario*)**

Routine to play a scenario

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario number to play

Returns : 1 : successful completion

0 : error

Definition at line 994 of file KondoApi.cpp.
References SendSetCom().

• **int RCB_SetHomePositions (int *id*)**

Routine to set the Home Position

Parameters :

← ***id*** Specify Board ID

Returns : 1 : successful completion

0 : error

Definition at line 623 of file KondoApi.cpp.
References Console, and SendSetCom().

• **int RCB_SetMotionPosition (int *id*, int *motion*, int *posNumber*, int *speed*, motors *motorsdata*)**

Routine to set the Motors Position of the Motion Position Index

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

← ***posNumber*** Specify the motion position index

← ***speed*** Stores Motors Speed Value

← ***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 910 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and SendSetCom().

• **int RCB_SetMotionPositionCount (int *id*, int *motion*, int *posNumber*)**

Routine to set the number of positions in the motion

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

← ***posNumber*** Number of positions in the motion

Returns : 1 : successful completion

0 : error

Definition at line 967 of file KondoApi.cpp.

References SendSetCom().

• **int RCB_SetMotorPositions (int *id*, int *speed*, motors *motorsdata*)**

Routine to set the Motors Position

Parameters :

← ***id*** Specify Board ID

← ***speed*** Motors Speed

← ***motorsdata*** Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 729 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and SendSetCom().

Referenced by Set90(), and SetHome().

• **int RCB_SetScenarioMotion (int *id*, int *scenario*, int *motionIndex*, int *motion*)**

Routine to set the motion Index of the scenario's motion number motion

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario number

← ***motionIndex*** Specifiy the motion number in the scenario

← ***motion*** Specify the motion number of the scenario motion index

Returns : 1 : successful completion

0 : error

Definition at line 1032 of file KondoApi.cpp.
References SendSetCom().

• **int RCB_SetScenarioMotionCount (int *id*, int *scenario*, int *motionNumber*)**

Routine to set the number of motions in the scenario

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specify the scenarion

← ***motionNumber*** Specify the number of motions in the scenario

Returns : 1 : successful completion

0 : error

Definition at line 1071 of file KondoApi.cpp.
References SendSetCom().

• **int RCB_SetSwitchBits (int *id*, int *switchValue*)**

Routine to set the Board Software Switch Bits

Parameters :

← ***id*** Specify Board ID

← ***switchValue*** Software Switch Value

Returns : 1 : successful completion

0 : error

Definition at line 541 of file KondoApi.cpp.
References SendSetCom().
Referenced by SetHome().

• **int RCB_SetTrimPositions (int *id*, motors *motorstrim*)**

Routine to set the Motors Trim Position

Parameters :

← ***id*** Specify Board ID

← ***motorstrim*** Motors Trim Values

Returns : 1 : successful completion

0 : error

Definition at line 835 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and SendSetCom().

• **int Set90 (void)**

Routine to set all servos to 90°

Returns : 1 : successful completion

0 : error

Definition at line 1158 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and RCB_SetMotorPositions().

• **int SetHome (void)**

Routine to set robot to its Home position (Read Home position from Robot and set motors to this position)

Returns : 1 : successful completion

0 : error

Definition at line 1085 of file KondoApi.cpp.

References Console, RCB_GetHomePositions(), RCB_SetMotorPositions(), and RCB_SetSwitchBits().

KondoApi.h File Reference

Classes

– struct **defmotor_t**

Defines

- #define **Debug 0**
- **#define Console 0**

Typedefs

- typedef **defmotor_t motors**

Functions

- int **RCB_GetID (int *reply)**
- **int RCB_GetSwitchBits (int id, int *reply)**
- **int RCB_SetSwitchBits (int id, int switchValue)**
- **int RCB_GetHomePositions (int id, motors *motorsdata)**
- **int RCB_SetHomePositions (int id)**
- **int RCB_GetMotorPositions (int id, motors *motorsdata)**
- **int RCB_SetMotorPositions (int id, int speed, motors motorsdata)**
- **int RCB_GetTrimPositions (int id, motors *motorsdata)**
- **int RCB_SetTrimPositions (int id, motors motorstrim)**
- **int RCB_GetMotionPosition (int id, int motion, int position, motors *motorsdata, int *speed)**
- **int RCB_SetMotionPosition (int id, int motion, int posNumber, int speed, motors motorsdata)**
- **int RCB_GetMotionPositionCount (int id, int motion, int *reply)**
- **int RCB_SetMotionPositionCount (int id, int motion, int posNumber)**
- **int RCB_PlayMotion (int id, int motion)**
- **int RCB_PlayScenario (int id, int scenario)**
- **int RCB_GetScenarioMotion (int id, int scenario, int motion, int *reply)**
- **int RCB_SetScenarioMotion (int id, int scenario, int motionIndex, int motion)**
- **int RCB_GetScenarioMotionCount (int id, int scenario, int *reply)**
- **int RCB_SetScenarioMotionCount (int id, int scenario, int motionNumber)**
- **int SendSetCom (unsigned char *cmd, int cmdlength, int id)**
- **int SendGetCom (unsigned char *cmd, int cmdlength, int replength)**
- **int SetHome (void)**
- **int Set90 (void)**

Define Documentation

#define Console 0

If set to 1, received datas will be displayed with format
Definition at line 99 of file KondoApi.h.

Referenced by RCB_GetHomePositions(), RCB_GetID(), RCB_GetMotionPosition(), RCB_-GetMotionPositionCount(), RCB_GetMotorPositions(), RCB_GetScenarioMotion(), RCB_Get-ScenarioMotionCount(), RCB_GetSwitchBits(), RCB_GetTrimPositions(), RCB_SetHomePositions(), and SetHome().

#define Debug 0

If set to 1, sent and received Comport datas will be displayed
Definition at line 87 of file KondoApi.h.

Typedef Documentation

typedef struct defmotor_t motors

Function Documentation

• **int RCB_GetHomePositions (int *id*, motors * *motorsdata*)**

Routine to get the Home Position

Parameters :

← ***id*** Specify Board ID

→ ∗***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 555 of file KondoApi.cpp.
References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().
Referenced by SetHome().

• **int RCB_GetID (int * *reply*)**

Specific LINUX declarations :
TODO : Communication functions, uint8_t definition, print() and print_uint8() functions
Definition at line 478 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetMotionPosition (int *id*, int *motion*, int *position*, motors * *motorsdata*, int * *speed*)**

Routine to get the Motors Position of the Motion Position Index

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

← ***position*** Specify the motion position index

→ ∗***motorsdata*** Stores Motors Position Values

→ ∗***speed*** Stores Motors Speed Value

Returns : 1 : successful completion

0 : error

Definition at line 870 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().

• **int RCB_GetMotionPositionCount (int *id*, int *motion*, int ∗ *reply*)**

Routine to get the number of positions in the motion

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

→ ***reply*** Store the number of positions in the motion

Returns : 1 : successful completion

0 : error

Definition at line 942 of file KondoApi.cpp.

References Console, and SendGetCom().

• **int RCB_GetMotorPositions (int *id*, motors ∗ *motorsdata*)**

Routine to get the Motors Actual Position

Parameters :

← ***id*** Specify Board ID

→ ∗***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 661 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().

• **int RCB_GetScenarioMotion (int *id*, int *scenario*, int *motion*, int * *reply*)**

Routine to get the motion Index of the scenario's motion number motion

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario number

← ***motion*** Specifiy the motion number

→ ***reply*** Store the motion index

Returns : 1 : successful completion

0 : error

Definition at line 1007 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetScenarioMotionCount (int *id*, int *scenario*, int * *reply*)**

Routine to get the number of motions in the scenario

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario

→ ***reply*** Store the number of motions in the scenario

Returns : 1 : successful completion

0 : error

Definition at line 1047 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetSwitchBits (int *id*, int * *reply*)**

Routine to get the Board Software Switch Bits

Parameters :

← ***id*** Specify Board ID

→ ***reply*** Board Software Switch Value

Returns : 1 : successful completion

0 : error

Definition at line 513 of file KondoApi.cpp.
References Console, and SendGetCom().

• **int RCB_GetTrimPositions (int *id*, motors * *motorsdata*)**

Routine to get the Motors Trim Position

Parameters :

← ***id*** Specify Board ID

→ ∗***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 767 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, Console, and SendGetCom().

• **int RCB_PlayMotion (int *id*, int *motion*)**

Routine to play a motion

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number to play

Returns : 1 : successful completion

0 : error

Definition at line 981 of file KondoApi.cpp.

References SendSetCom().

• **int RCB_PlayScenario (int *id*, int *scenario*)**

Routine to play a scenario

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario number to play

Returns : 1 : successful completion

0 : error

Definition at line 994 of file KondoApi.cpp.

References SendSetCom().

• **int RCB_SetHomePositions (int *id*)**

Routine to set the Home Position

Parameters :

← ***id*** Specify Board ID

Returns : 1 : successful completion

0 : error

Definition at line 623 of file KondoApi.cpp.
References Console, and SendSetCom().

• **int RCB_SetMotionPosition (int *id*, int *motion*, int *posNumber*, int *speed*, motors *motorsdata*)**

Routine to set the Motors Position of the Motion Position Index

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

← ***posNumber*** Specify the motion position index

← ***speed*** Stores Motors Speed Value

← ***motorsdata*** Stores Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 910 of file KondoApi.cpp.
References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and SendSetCom().

• **int RCB_SetMotionPositionCount (int *id*, int *motion*, int *posNumber*)**

Routine to set the number of positions in the motion

Parameters :

← ***id*** Specify Board ID

← ***motion*** Specify the motion number

← ***posNumber*** Number of positions in the motion

Returns : 1 : successful completion

0 : error

Definition at line 967 of file KondoApi.cpp.
References SendSetCom().

• **int RCB_SetMotorPositions (int *id*, int *speed*, motors *motorsdata*)**

Routine to set the Motors Position

Parameters :

← ***id*** Specify Board ID

← ***speed*** Motors Speed

← ***motorsdata*** Motors Position Values

Returns : 1 : successful completion

0 : error

Definition at line 729 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and SendSetCom().

Referenced by Set90(), and SetHome().

• **int RCB_SetScenarioMotion (int *id*, int *scenario*, int *motionIndex*, int *motion*)**

Routine to set the motion Index of the scenario's motion number motion

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specifiy the scenario number

← ***motionIndex*** Specifiy the motion number in the scenario

← ***motion*** Specify the motion number of the scenario motion index

Returns : 1 : successful completion

0 : error

Definition at line 1032 of file KondoApi.cpp.

References SendSetCom().

• **int RCB_SetScenarioMotionCount (int *id*, int *scenario*, int *motionNumber*)**

Routine to set the number of motions in the scenario

Parameters :

← ***id*** Specify Board ID

← ***scenario*** Specify the scenarion

← ***motionNumber*** Specify the number of motions in the scenario

Returns : 1 : successful completion

0 : error

Definition at line 1071 of file KondoApi.cpp.

References SendSetCom().

• **int RCB_SetSwitchBits (int *id*, int *switchValue*)**

Routine to set the Board Software Switch Bits

Parameters :

← ***id*** Specify Board ID

← ***switchValue*** Software Switch Value

Returns : 1 : successful completion

0 : error

Definition at line 541 of file KondoApi.cpp.
References SendSetCom().
Referenced by SetHome().

• **int RCB_SetTrimPositions (int *id*, motors *motorstrim*)**

Routine to set the Motors Trim Position

Parameters :

← ***id*** Specify Board ID

← ***motorstrim*** Motors Trim Values

Returns : 1 : successful completion

0 : error

Definition at line 835 of file KondoApi.cpp.
References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and SendSetCom().

• **int SendGetCom (unsigned char * *cmd*, int *cmdlength*, int *replength*)**

Routine to send a 'get' command

Parameters :

← ∗***cmd*** Command to send

← ***cmdlength*** Command Length

← ***replength*** Reply Length

Returns : 1 : successful completion

0 : error

Referenced by RCB_GetHomePositions(), RCB_GetID(), RCB_GetMotionPosition(), RCB_GetMotionPositionCount(), RCB_GetMotorPositions(), RCB_GetScenarioMotion(), RCB_GetScenarioMotionCount(), RCB_GetSwitchBits(), and RCB_GetTrimPositions().

• **int SendSetCom (unsigned char * *cmd*, int *cmdlength*, int *id*)**

Routine to send a 'set' command

Parameters :

← ∗***cmd*** Command to send

← ***cmdlength*** Command Length

← ***id*** Specify Board ID

Returns : 1 : successful completion

0 : error

Referenced by RCB_PlayMotion(), RCB_PlayScenario(), RCB_SetHomePositions(), RCB_SetMotionPosition(), RCB_SetMotionPositionCount(), RCB_SetMotorPositions(), RCB_SetScenarioMotion(), RCB_SetScenarioMotionCount(), RCB_SetSwitchBits(), and RCB_SetTrimPositions().

• **int Set90 (void)**

Routine to set all servos to 90°

Returns : 1 : successful completion

0 : error

Definition at line 1158 of file KondoApi.cpp.

References defmotor_t : :ch1, defmotor_t : :ch10, defmotor_t : :ch11, defmotor_t : :ch12, defmotor_t : :ch2, defmotor_t : :ch3, defmotor_t : :ch4, defmotor_t : :ch5, defmotor_t : :ch6, defmotor_t : :ch7, defmotor_t : :ch8, defmotor_t : :ch9, and RCB_SetMotorPositions().

• **int SetHome (void)**

Routine to set robot to its Home position (Read Home position from Robot and set motors to this position)

Returns : 1 : successful completion

0 : error

Definition at line 1085 of file KondoApi.cpp.

References Console, RCB_GetHomePositions(), RCB_SetMotorPositions(), and RCB_SetSwitchBits().

Neuvième partie

Annexe D : Documentation API Carte Extension

Api_CarteExtension File Index

Api_CarteExtension File List

Here is a list of all files with brief descriptions :
- API Carte Extension Atmega : Api.cpp
- API Carte Extension Atmega : Api.h

Api_CarteExtension File Documentation

AtmegaApi.cpp File Reference

```
#include "AtmegaApi.h"
```

Functions

- int **AtMega_RCB_Home (void)**
- **int AtMega_RCB_PlayMotion (int motion)**
- **int AtMega_I2c_ReadPCF8591 (int Rep[])**
- **int AtMega_GetHomePositions (int Rep[])**
- **int AtMega_GetMotorsPositions (int Rep[])**
- **int AtMega_ReadCEA (int Rep[])**

Function Documentation

• **int AtMega_GetHomePositions (int *Rep*[])**

Routine to get the Home positions of robot

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion

0 : error

Definition at line 284 of file AtmegaApi.cpp.
References TIMEOUT_COMKHR1.

• **int AtMega_GetMotorsPositions (int *Rep*[])**

Routine to get motors positions from robot

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion
0 : error

Definition at line 327 of file AtmegaApi.cpp.
References TIMEOUT_COMKHR1.

- **int AtMega_I2c_ReadPCF8591 (int *Rep*[ ])**

Routine to read the 2 PCF8591 over I2C

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion
0 : error

Definition at line 239 of file AtmegaApi.cpp.
References TIMEOUT_COM.

- **int AtMega_RCB_Home (void)**

Routine to set the Kondo robot to Home position

Returns : 1 : successful completion
0 : error

Definition at line 179 of file AtmegaApi.cpp.

- **int AtMega_RCB_PlayMotion (int *motion*)**

Routine to play a motion on the robot

Parameters :

← ***motion*** Motion to play

Returns : 1 : successful completion
0 : error

Definition at line 203 of file AtmegaApi.cpp.
References TIMEOUT_COM.

- **int AtMega_ReadCEA (int *Rep*[ ])**

Routine to read the 6 values of the CEA sensor

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion
0 : error

Definition at line 370 of file AtmegaApi.cpp.
References TIMEOUT_COM.

AtmegaApi.h File Reference

Defines

- #define **TIMEOUT_COM 3000**
- **#define TIMEOUT_COMKHR1 15000**

Functions

- int **AtMega_RCB_Home (void)**
- **int AtMega_RCB_PlayMotion (int motion)**
- **int AtMega_I2c_ReadPCF8591 (int Rep[])**
- **int AtMega_GetHomePositions (int Rep[])**
- **int AtMega_GetMotorsPositions (int Rep[])**
- **int AtMega_ReadCEA (int Rep[])**

Define Documentation

#define TIMEOUT_COM 3000

Definition at line 41 of file AtmegaApi.h.
Referenced by AtMega_I2c_ReadPCF8591(), AtMega_RCB_PlayMotion(), and AtMega_ReadCEA().

#define TIMEOUT_COMKHR1 15000

Definition at line 42 of file AtmegaApi.h.
Referenced by AtMega_GetHomePositions(), and AtMega_GetMotorsPositions().

Function Documentation

• **int AtMega_GetHomePositions (int *Rep*[])**

Routine to get the Home positions of robot

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion
0 : error

Definition at line 284 of file AtmegaApi.cpp.
References TIMEOUT_COMKHR1.

• **int AtMega_GetMotorsPositions (int *Rep*[])**

Routine to get motors positions from robot

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion
0 : error

Definition at line 327 of file AtmegaApi.cpp.
References TIMEOUT_COMKHR1.

• **int AtMega_I2c_ReadPCF8591 (int *Rep*[])**

Routine to read the 2 PCF8591 over I2C

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion
0 : error

Definition at line 239 of file AtmegaApi.cpp.
References TIMEOUT_COM.

• **int AtMega_RCB_Home (void)**

Routine to set the Kondo robot to Home position

Returns : 1 : successful completion
0 : error

Definition at line 179 of file AtmegaApi.cpp.

• **int AtMega_RCB_PlayMotion (int *motion*)**

Routine to play a motion on the robot

Parameters :

← ***motion*** Motion to play

Returns : 1 : successful completion
0 : error

Definition at line 203 of file AtmegaApi.cpp.
References TIMEOUT_COM.

• **int AtMega_ReadCEA (int *Rep*[ ])**

Routine to read the 6 values of the CEA sensor

Parameters :

→ ***Rep*** Reply from the board

Returns : 1 : successful completion

0 : error

Definition at line 370 of file AtmegaApi.cpp.
References TIMEOUT_COM.

Installation de capteurs et centralisation par microcontrôleur pour le robot Bipède KONDO

INRIA Rhone-Alpes - Service Support des Expérimentations et de Developpement logiciel (SED)
IUT Clermont-Ferrand - Licence Professionnelle Informatique Embarquée et Robotique

Maitres de stage : Pissard-Gibollet Roger, Cuniberto Jean-François
Examinateurs : KAUFMANN, LE HEN

Avril 2006 - Juillet 2006

Ce rapport présente le travail réalisé au cours du stage de fin d'année de Licence professionnelle Informatique Embarquée et Robotique de l'IUT de Clermont Ferrand. Ce stage à été éffectué à l'Institut National de Recherche en Informatique et Automatique (INRIA) dans l'unité de recherche Rhône-Alpes. Ce stage consistait à ajouter des capteurs sur un robot humanoïde, le Kondo KHR-1. De plus, il fallais réaliser une carte d'extension évolutive qui permetterais de récupérer diverses informations sur les capteurs déja présents tout en laissant la possibilité d'ajouter de nouveaux capteurs. Enfin, un logiciel permettant de capturer les données issues de cette carte à été créé pour simplifier l'utilisation du kit (interface graphique).