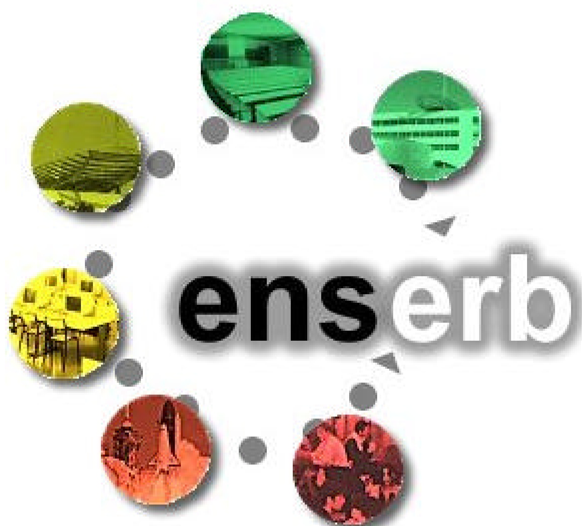
IZQUIERDO GARCIA, David
E.N.S.E.R.B.

Filière Electronique

Janvier 2000

RAPPORT DE STAGE DE TROISIEME ANNEE EFFECTUE A



école nationale supérieure d´électronique et de radioélectricité de Bordeaux

1, avenue du docteur Albert Schweitzer 33402 Talence-Cedex

téléphone : 05 56 84 65 00   fax : 05 56 37 20 23

e-mail : webmaster@enserb.u-bordeaux.fr

adresse web : http://www.enserb.u-bordeaux.fr/

Responsable de stage : **M. P. KADIONIK**

FICHE DE STAGE

Nom de l'entreprise :

ENSERB (Ecole Nationale Supérieure d'Electronique et Radioelectricité de Bordeaux).

Adresse :

ENSERB
1, avenue du Docteur Albert Schweitzer
Domaine Universitaire, 33402
TALENCE - CEDEX

Responsable de stage :

Nom : M. Kadionik

Résumé du sujet :

"Etude et mise en place d'une machine virtuelle JAVA pour le microcontrôleur Motorola 68HC11"

Mots Clés :

Microcontrôleur, Assembleur, Machine virtuelle JAVA, méthodes natives, Threads, API, Client/Serveur.

Travail effectué :

Réalisation d'une application en Java pour tester la machine virtuelle JAVA dans le microcontrôleur Motorola 68HC11 et réalisation d'une application pour pouvoir contrôler la carte microcontrôleur via le port série avec une serveur Web.

Date du stage :

Du 6 Septembre au 30 Novembre 1999.

REMERCIEMENTS

Je remercie l'ENSERB où j'ai pu effectuer mon stage et plus particulièrement:

- M. **Patrice KADIONIK**, qui a été mon tuteur de stage, et m'a aidé pendant tout le stage.
- Tout le groupe du laboratoire de Traitement du Signal et Télécommunications pour avoir accepté ma compagnie, surtout les jours quand rien ne va, spécialement à M. Yannick BERTHOMIEU et Alexandre RANDRIANTSOA.
- M. Sebastian GARCIA, qui a été en train de m'aider sur le microcontrôleur.
- ainsi que tous les membres de l'ENSERB qui, directement ou indirectement, ont contribué au bon déroulement de mon stage.

SOMMAIRE

INTRODUCTION____6

PRÉSENTATION DE L'ENSERB. ____7

OBJECTIFS ET SUJET DU STAGE. ____8

DESCRIPTION DE L'ENVIRONNEMENT MATERIELLE. _______________________9

DEVELOPPEMENT DU STAGE. ____10

1.-MICROCONTRÔLEUR DE MOTOROLA 68HC11.________________________________ 10

QU'EST CE QUE LE MICROCONTROLATUER MOTOROLA 68HC11? ____________________ 10

MIS AU POINT AVEC LE 68HC11. ____ 14

PCBUG11.____ 15

BUFFALO. ____ 16

EXÉCUTION DES PROGRAMMES. ____ 18

2.-LA MACHINE VIRTUELLE. ____ 20

EN QUOI CELA CONSISTE?____ 20

1.- Analyse des requises fonctionnelles. ____________________________________ 22

2.- Codification de l'application en Java. ___________________________________ 22

3.- Processus d'optimisation des classes Java.________________________________ 22

4.- Code des méthodes natives.____ 23

5.- Vérification du logiciel. ____ 24

L'AIDE DES THREADS.____ 25

1.- En étendant la classe Thread. ____ 26

2.- En mettant en œuvre l'interface Runnable. ________________________________ 27

MÉTHODES NATIVES. ____ 28

1.-Déclaration des méthodes natives dans la classe d'application. 29
2.-Méthodes natives en assembleur. 30
3.-Ecriture de méthodes natives en C. 31
4.-Limitations des méthodes natives. 32
PROBLÈMES RENCONTRÉS AVEC LES MÉTHODES NATIVES. 33
PAQUETAGES. 34
JAVA.LANG PAQUETAGE.- 34
HW.MOT68HC11 PAQUETAGE.- 35
RESUME DE LA MACHINE VIRTUELLE. 35
LIMITATIONS. 36
3.-LA COMMUNICATION API DE JAVA. 37
L'OBJECTIF ET LA FIN DU PROJET. 39
CONCLUSIONS 50

INTRODUCTION

Je suis un étudiant ERASMUS venant d'Espagne, de l'Université de Saragosse. J'ai eu la possibilité de faire ma dernière année de mes études d'Ingénierie de Télécommunications dans le cadre ERASMUS à l'ENSERB.

Mon stage a débuté le 6 septembre et s'est terminé le 30 novembre 1998 au sein de l'ENSERB, dans le laboratoire de Traitement de Signal et Télécommunications (Salle S016).

Le but de ce stage a été l'implémentation de la machine virtuelle de Java dans le microcontrôleur Motorola 68HC11. De cette façon nous pouvons à partir d'un ordinateur quelconque (par exemple un PC que nous avons dans notre bureau et que nous avons connecté à Internet) se connecter à un Serveur Web qui pilote la carte 68HC11 et exécuter des commandes que nous voulons (par exemple prendre la température d'un capteur).

L'objectif de ce rapport est de retracer fidèlement le déroulement de ce stage en présentant et en expliquant le sujet en lui-même, puis en présentant les outils utilisés et testés et enfin, en décrivant l'application de la machine virtuelle utilisée. Avant cela, il est nécessaire de présenter rapidement le lieu du stage.

PRÉSENTATION DE L'ENSERB.

Comme tout le monde sait, l'ENSERB (ÉCOLE NATIONALE d'INGÉNIERIE SUPERIEURE d'ELECTRONIQUE et RADIOELECTRICITÉ de BORDEAUX) est une école d'ingénieurs qui se spécialisent dans l'électronique ou dans l'informatique. Actuellement ces deux types de spécialités sont très recherchés grâce à l'essor des télécommunications (par exemple les mobiles, la télévision numérique, les réseaux, ...).

Cela fait que l'ENSERB est une école orienté vers le futur.

En plus, la disponibilité des moyens et la facilité d'accès à tous ces moyens font de cette école un centre dans lequel l'investigation et le développement de projets sont des valeurs en hausse.

OBJECTIFS ET SUJET DU STAGE.

Ce stage consiste à l'implémentation d'une machine virtuelle Java dans le microcontrôleur 68HC11 de Motorola. De cette façon, nous avons fait un programme en Java que nous avons introduit dans le microcontrôleur grâce à la machine virtuelle. Avec ce programme, nous voulons qu'au moment de donner une commande au microcontrôleur via le port RS-232 de notre ordinateur (via le port série) on exécute ce que nous avons prévu dans le programme Java. Un exemple d'utilisation est de connecter la carte microcontrôleur Motorola à un capteur de température qui nous mesure tout le temps la température, de telle sorte qu'au moment d'écrire dans l'écran de l'ordinateur le suivant instruction : "lis", le microcontrôleur comprenne grâce au programme que nous avons mis en oeuvre dans lui, qu'il doit lire la température que le capteur va lui donner et la retourner vers le port série.

Mais nous avons fait toujours un pas en plus dans la réalisation de ce projet, puisque l'objectif attendu a été de connecter la carte à un Serveur Web (un ordinateur qui est connecté à un réseau, comme par exemple Internet, qui est le plus répandu, pour donner quelque service) on peut avoir

accès à la carte dans quelconque "client" (un ordinateur qui utilise les services donnés par le Serveur) dans n'importe où, rien qu'en ayant une connexion à Internet.

Pour que la communication entre le Serveur et la carte soit possible, nous avons utilisé la "communication API" de Java, de sorte que nous contrôlons parfaitement le port RS-232 de notre Serveur qui communique avec la carte (pour l'envoi des commandes et la réception des résultats qui doivent s'imprimer dans l'écran du navigateur Web).

DESCRIPTION DE L'ENVIRONNEMENT MATERIELLE.

Ce stage a été réalisé pour l'ENSERB. Le lieu du travail se situe dans le laboratoire du traitement du signal et télécommunications de l'ENSERB, dans le pôle Electronique.

Le stage a consisté à programmer en Java, mais aussi en assembleur (assembleur du Motorola 68HC11) et en langage C.

Pendant le stage on a travaillé avec des différents logiciels :

- Assembleur du Motorola 68HC11.
- Langage C.
- JDK 1.2.
- JVM (Java Virtual Machine).

- Communication API de Java.
- PCBUG11.
- Moniteur BUFFALO, avec Hyperterminal de Windows.

L'équipement avec lequel s'est fait tout le stage a été :

- La carte avec le microcontrôleur 68HC11 avec tous ses composants auxiliaires.
- Un ordinateur connecté à la carte via la RS-232 (le port série) avec tout le logiciel qu'on a indiqué ci-dessus.

DEVELOPPEMENT DU STAGE.

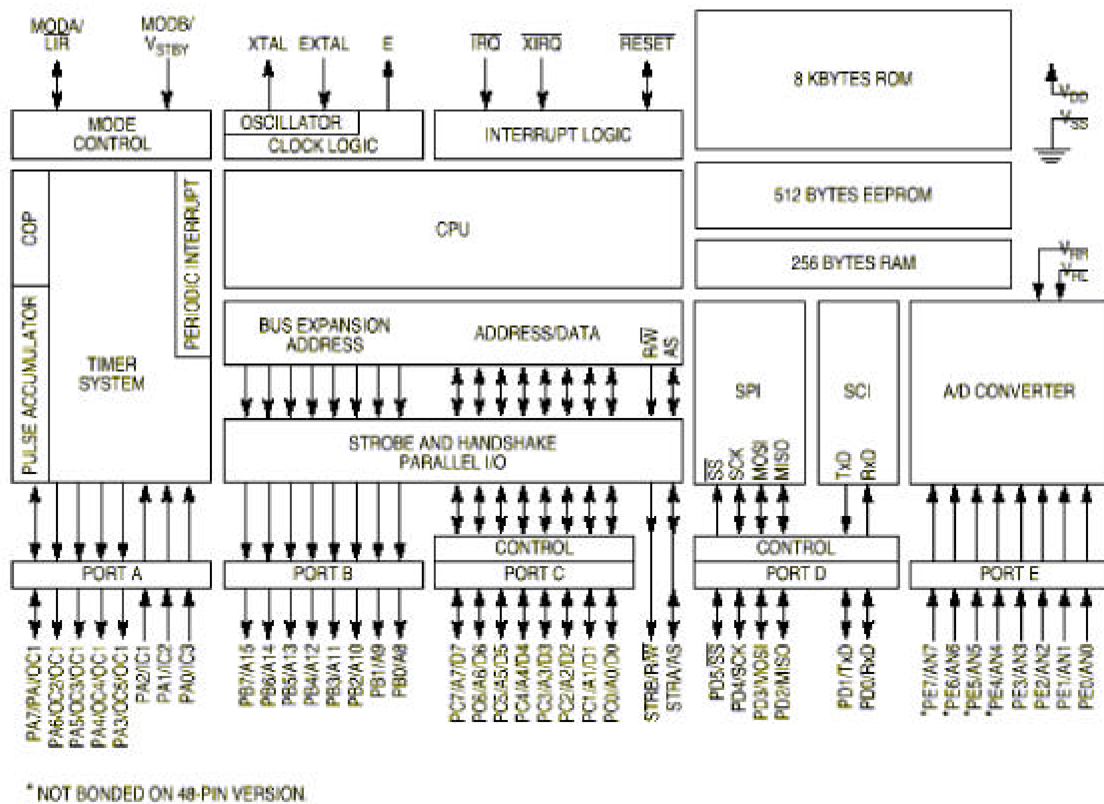
1.-MICROCONTRÔLEUR DE MOTOROLA 68HC11.

QU'EST CE QUE LE MICROCONTROLATUER MOTOROLA 68HC11?

Le microcontrôleur 68HC11 de Motorola est un microcontrôleur de 8 bits, cela semble vraiment plus puissant et dépassé par rapport aux processeurs 32 et maintenant 64 bits. Mais il trouve largement sa place dans des applications de la vie quotidienne où l'on n'a pas besoin de "MIPS" et de "MFLOPS" : micro-onde, lave vaisselle, TV, domotique..., alors leur étude est toujours d'actualité.

Un microcontrôleur est simplement un microprocesseur. Un programmeur réalise un programme qu'on mette en oeuvre dans le micro, de forme que celui-ci (le programme) quand on s'exécute, on obtient une séquence de sortie qu'on aide à porter une série d'actions simples qu'en même temps on sert d'entrée au microcontrôleur pour continuer à exécuter le programme. Et ainsi de suite.

Dans la figure adjointe on peut voir comme est le structure du microcontrôleur de Motorola 68HC11.



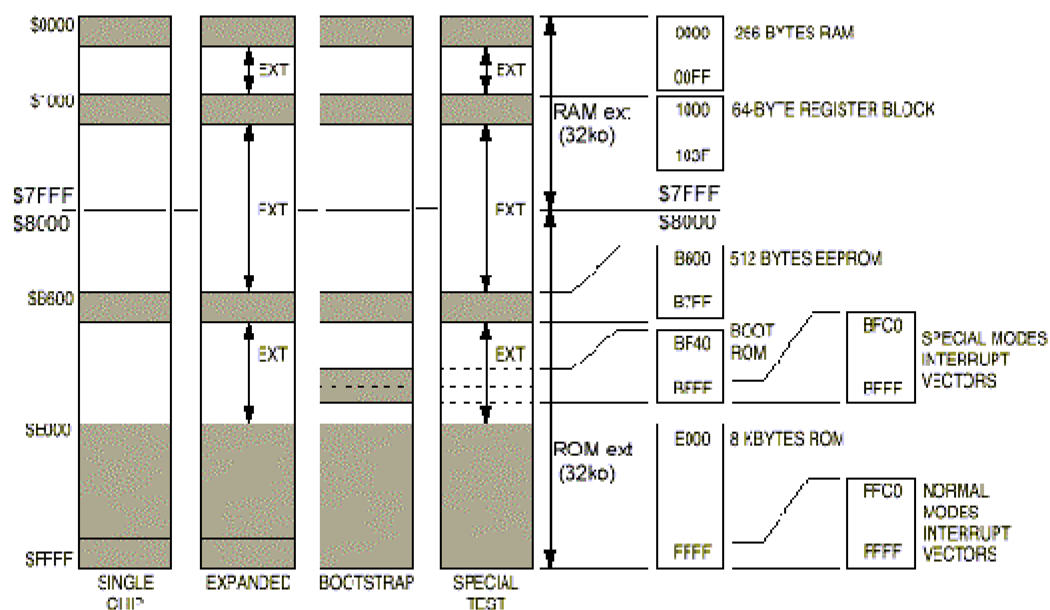
On peut souligner les éléments suivants :

- Microcontrôleur en technologie HCMOS.
- 8 Ko. de ROM.
- 512 octets d' EEPROM.
- 256 octets de RAM.
- Un timer de 16 bits (3 entrées de capture, 5 sorties de comparaison).
- 2 accumulateurs de 8 bits.
- Une liaison série asynchrone (SCI).
- Une liaison synchrone (SPI).
- Un convertisseur Analogique/Numérique 8 bits, 8 entrées multiplexées.
- Circuit d'interruption temps réel.
- Circuit oscillant externe (quartz 8MHz, dans notre application).

Les modes du fonctionnement du 68HC11 sont résumés dans le tableau suivant :

MODB	MODA	Mode sélectionné
1	0	Single Chip (Mode 0)
1	1	Expanded Multiplexed (Mode 1)
0	0	Special Bootsrap
0	1	Special test

La cartographie mémoire (mapping mémoire) du microcontrôleur 68HC11 est la suivante :



Dans notre projet, le 68HC11 a été utilisé de deux manières différentes :

- Avec le logiciel PCBUG11 de Motorola, on a travaillé en mode bootstrap. Ceci permit de déverminer (“débugger”) le code du programme à tester par l’intermédiaire d’un jeu de commandes. Le programme testé sera ensuite mis en ROM après la configuration du processeur en mode étendu lui conférant ainsi une autonomie totale.

- Avec le moniteur BUFFALO (“Bit Users’s Fast Friendly Aid to Logical Operation). On a travaillé en mode étendu puisque le moniteur est programmé dans l’EPROM externe. Le moniteur par l’intermédiaire d’un jeu de commandes permet le débuggage grâce à un simple terminal ou un émulateur de terminal sur ordinateur, comme a été notre cas, que nous avons utilisé le

Hyperterminal de Windows pour avoir un émulateur de terminal Buffalo sur l'ordinateur.

Nous allons voir un peu plus en détail comment fonctionnent ces deux logiciels.

Les autres deux formes de travail du microcontrôleur (special test et single chip) n'ont pas été utilisées pendant le stage puisqu'ils correspondent à deux formes de travail spéciaux, donc nous n'avons pas eu besoin.

MIS AU POINT AVEC LE 68HC11.

Une fois avoir présenté la carte mère du 68HC11, nous allons présenter la façon de l'utiliser.

Pour tout cela, nous avons fait les huit TP qu'on nous a proposé au début du stage (nous pouvons voir à ensuite un exemple d'un programme développé en assembleur). On nous indiquait l'utilisation du PCBUG11 et du moniteur Buffalo :

```
****************************************************************
* Programme de test d'entrée-sortie sur carte d'affichage 68HC11      *
* Configuration de la carte :                                          *
*   - PORTE : 8 bits en entées                                                         *
*   - PORTA : entées = PA0, PA1, PA2                                   *
*              sorties = PA3, PA4, PA5, PA6, PA7                       *
****************************************************************
DEBRAM   EQU        $0000
DEBPROG EQU         $B600
PORTA        EQU    $1000
PORTE        EQU    $100A

             ORG    DEBPROG
             LDS    #$FF                  * initialisation de la pile
```

```
START
          LDX      PORTA                      * selon l'état de PA0, on fait :
          BRCLR    0,X, #%00000001,chenill     * PA0=0 -> chenillard 5leds
recopi
          LDAA     PORTE                      * PA0=1 -> recopie des 5bits de
poids fort
          STAA     PORTA                      *       du PORTE sur PORTA
          BRA      START
chenill
          LDAA     #8                         * routine chenillard AR
GCHE      STAA     PORTA                      * allume la LED PA3
          JSR      TEMPO                      * routine de temporisation
          ROLA                                * rotation vers la gauche d'1 bit
          CMPA     #128                       * on regarde si on est arrive à PA7
          BNE      GCHE                       * si non on reboucle

          LDAA     #128                       * allume la LED PA7
DRTE      STAA     PORTA
          JSR      TEMPO                      * routine de temporisation
          RORA
          CMPA     #8                         * on regarde si on est arrive à PA3
          BNE      DRTE
          BRA      START                      * on a termine un cycle AR

TEMPO
          LDY      #100000                    * tempo de xx µs
          DEY
          CPY      #0
          BNE      TEMPO
          RTS
      END
```

Nous allons voir un peu plus en détail comme ces deux logiciels peuvent travailler :

PCBUG11.

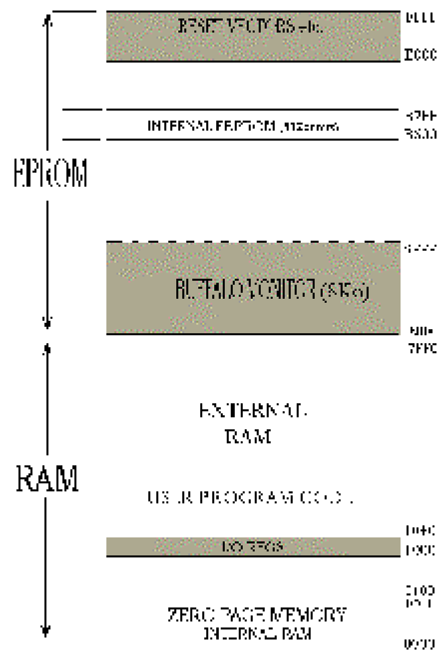
PCBUG11 est un logiciel qui permet de télécharger et débugger des programmes que l'utilisateur désire faire exécuter par le 68HC11. Mais comment procède-t-on pour télécharger son programme dans la mémoire du 68HC11?

Lors de l'écriture d'un programme en assembleur 68HC11, une directive d'assemblage (ORG + adresse en hexadécimale) permet de définir l'adresse de base du programme dans la mémoire. Il faut donc placer le programme dans une zone libre et inscriptible. Par exemple la RAM (interne ou externe) ou l'EEPROM interne.

```
PCBUG342 Ax
Total bytes loaded: $0246
Total bytes written: $0246
Total bytes loaded: $0270
Total bytes written: $002A
C000 4F              > CLRA
C001 CEF000          > LDX   #$F000
C004 2003            > BRA   $C009
C006 A700            > STAA  $00,X
C008 08              > INX
C009 8CF002          > CPX   #$F002
C00C 26F8            > BNE   $C006
C00E 8E00FF          > LDS   #$00FF
C011 BDC0A1          > JSR   >$C0A1
C014 20FE            > BRA   $C014
C016 3C              > PSHX
C017 3C              > PSHX
PC     ACCA  ACCB  X      Y      CCR (SXHINZVC)  SP         MCU:  68HC11A8
$0000  $40   $2C   $1000  $0100  $40 %.1......   $00EB      RTS Level  :ON
                                                            State:  STOPPED
                                                            Base :  HEX
restart                                                     User RST  $XXXX
loads testio2                                               User SWI  $XXXX
asm c000                                                    User XIRQ $XXXX
>>_
```

BUFFALO.

Un moniteur est un programme beaucoup plus gros (8 Ko. environ) que l'on programme dans une EPROM externe. Les avantages sont multiples puisque le moniteur occupe très peu de RAM (uniquement pour quelques variables). Dans la figure suivante nous pouvons voir la cartographie mémoire avec le moniteur BUFFALO :



Il permet de débugger le programme utilisateur grâce à des commandes intégrées et le moniteur dialogue avec l'utilisateur par l'intermédiaire d'un simple terminal (que dans notre cas va être l'émulateur Hyperterminal de Windows).

L'exécution du moniteur se fait lors du Reset en mode étendu et rend libre l'utilisation du port série après le lancement du programme utilisateur. Ici nous pouvons voir comme est la présentation du logiciel en cours d'utilisation :

```
BUFFALO 3.4 Ex (ENSERB) - Bit User Fast Friendly Aid to Logical Operation

68HC11E9 CPU
8K BUFFALO MONITOR PROGRAM E2ROM: $8000 TO $9FFF
DEFAULT INTERNAL RAM & REGISTER ALLOCATION
EEPROM: $B600 TO $B7FF
EXTERNAL RAM 32K : $0000 TO $7FFF
>

ASM [<addr>]  Line asm/disasm
  [/,=]  Same addr,       [^,-]  Prev addr,       [+,CTLJ] Next addr
  [CR]  Next opcode,                              [CTLA,.]  Quit
BF <addr1> <addr2> [<data>]  Block fill memory
BR [-][<addr>] Set up bkpt table
BULK  Erase EEPROM,                 BULKALL  Erase EEPROM and CONFIG
CALL [<addr>] Call subroutine
GO [<addr>] Execute code at addr,        PROCEED  Continue execution
EEMOD [<addr> [<addr>]] Modify EEPROM range
LOAD, VERIFY [T] <host dwnld command>  Load or verify S-records
MD [<addr1> [<addr2>]]  Memory dump
MM [<addr>] or [<addr>]/  Memory Modify
  [/,=]  Same addr,  [^,-,CTLH] Prev addr,  [+,CTLJ,SPACE] Next addr
  <addr>O Compute offset,                  [CR]  Quit
MOVE <s1> <s2> [<d>]  Block move
OFFSET [-]<arg>  Offset for download
RM [P,Y,X,A,B,C,S]  Register modify
STOPAT <addr>  Trace until addr
T [<n>]  Trace n instructions
TM  Transparent mode (CTLA = exit, CTLB = send brk)
[CTLW]  Wait,          [CTLX,DEL] Abort          [CR]  Repeat last cmd

>
```

10:02:37 connecté | ANSI | 9600 8-N-1 | Défil | Maj | Num | Capturer | Imprimer l'écho

EXÉCUTION DES PROGRAMMES.

Une fois que nous déjà connaissons un peu plus les logiciels avec les que nous devions travailler, nous présentons le conteur de l'utilisation du 68HC11.

Pour cela, nous devions créer des programmes en assembleur (avec l'extension .ASC) ou en langage c (avec l'extension .C), qu'ils correspondant au code source du programme. L'utilisation d'un langage dépendait de l'utilisation de la carte. Dans le cas du langage C, nous utilisons des

fonctions déjà définies dans une librairie (LIBHC11) qui contrôlait les entrées/sorties en même temps que la gestion de la liaison série.

Une fois que ces deux programmes sont créés, on peut les assembler (dans le cas d'utilisation de langage assembleur), ou on peut les compiler et linker. Et pour les deux, on doit créer à la fin, des fichiers Srecord (avec l'extension .S19) qui sont des fichiers qui sont chargés dans la carte du 68HC11.

Une fois que nous avons chargé ces fichiers, nous les exécutons et les déverminons (débugger) jusqu'à obtenir les résultats que nous cherchions.

2.-LA MACHINE VIRTUELLE.

EN QUOI CELA CONSISTE?

La machine virtuelle de Java pour le 68HC11 est un logiciel qui nous permet mettre en œuvre le langage Java dans le microcontrôleur. C'est un programme qui va faire les fonctions de traducteur et qui va traduire les programmes en Java que l'usager a définis, dans un code que le micro 68HC11 peut comprendre. Ce qu'on fait avec le développement de la machine virtuelle de Java est mis en œuvre au maximum le langage Java dans les possibilités du microcontrôleur. Malheureusement, pas tous les paramètres, ni toutes les classes avec les que nous pouvons être habitués à travailler en Java ne peuvent être mis en œuvre puisque le micro 68HC11 dispose de ressources limités comparés par exemple à ceux d'un PC.

Il faut tenir en compte, en plus, des limitations qui simplifient la mis en œuvre de la machine virtuelle:

- Sécurité Java: ne s'applique pas puisque nous ne pouvons pas attendre que personne puisse écrire une application qui marche. Nous supposerons qu'on aura toujours un nombre limité de programmeurs qui connaissent à fond la carte mère comme pour écrire des applications en Java qui marchent avec elle. Ces suppositions font

que nous éliminons la nécessité de vérifier toutes les classes au démarrage du système.

- I/O (entrée/sortie): dans la majorité des cas cela comprend la communication avec le port série, l'accès aux ports parallèles et à la mémoire. I/O variera d'un système à un autre, ainsi que dans la plus part de cas on nécessitera une "customization" pour la part du programmeur de l'application java.
- Long Math: les opérations en virgule flottante et en entiers en 64 bits ne sont pas implantés dans la plus part des applications avec des intégrés.
- Méthodes Natives: Son invocation se mene à bien, mais introduit certaines limitations pour réduire le temps d'invocation. Cela signifie que les méthodes natives sont limitées à recevoir et rendre des paramètres seulement de types de données primitives de Java, mais avec l'exception de la classe "String" et les "matrices unidimensionnelles", les méthodes natives ne peuvent pas invoquer des méthodes hava ni accéder directement à des champs d'une classe.

Avec tous ces éclaircissements ici exposés, un programmeur d'applications en Java pour tous ces petits systèmes d'intégrés doit réaliser un code en tenant compte toutes les limitations de la machine virtuelle. Tout ce qui peut marcher sur un PC ne peut pas marcher dans le système d'intégrés. On peut le faire mais de manière très pauvre.

Le développement d'une application Java pour un système d'intégrés dans lequel le Motorola 68HC11 soit contenu peut être divisé suivant les points suivants:

1.- Analyse des requises fonctionnelles.

Normalement le développement commence avec une analyse des contraintes fonctionnelles. Dans cette analyse le programmeur/développeur doit décider entre autres choses quels composants software doit mettre en œuvre en Java et quels doivent être programmés comme les méthodes natives. C'est évident que les actions qui sont critiques dans le temps ou qui demandent l'accès au hardware de la carte mère devraient être mis en œuvre avec des méthodes natives (nous verrons un peu plus tard que ces méthodes peuvent être écrites en assembleur 68HC11 ou en langage C).

2.- Codification de l'application en Java.

Une fois que l'analyse est complète, nous passons à la codification (écriture) de l'application java.

L'unique limitation est que le programmeur/développeur doit garder en tête que le software va s'exécuter dans un système d'intégrés de 8 bits, et une mémoire heap très limitée. C'est aussi pour ça que le nombre des classes de la librairie est limité. Pour le moment, la librairie inclut des classes dans les paquetages java.lang et hw.mot68hc11.

Le programmeur devra prendre l'option de créer des “threads” dans l'application, de sorte que chaque thread réalise une tâche isolée. De cette façon, quand un thread est en train d'attendre qu'une condition soit vraie, on appelle la méthode yield( ) de sorte qu'autre thread puisse suivre à s'exécuter. De cette façon, nous améliorons beaucoup le rendement de notre système.

Les programmes d'application peuvent être écrits et compilés avec les beaucoup outils qui ont été créés pour cela. Nous avons pris dans notre stage le SUN's JDK.

3.- Processus d'optimisation des classes Java.

Le processus d'optimisation se mène à bien avec l'outil ClassLinker et implique une lecture de toutes les classes que comprend l'application, en remplaçant les références symboliques aux objets et indices et en engendrant les fichiers de sortie. Résoudre les références symboliques simplifiera l'architecture d'exécution dans le temps de la machine virtuelle de Java, et c'est qui est le plus important, une amélioration significative du temps d'exécution du code java.

Un des fichiers de sortie créé pour l'outil ClassLinker est une "image d'application" qu'après sera chargé (avec la machine virtuelle de java et les méthodes natives) dans la mémoire du dispositif. Dépendant de l'architecture hardware, le processus de chargement peut être le chargement du programme en mode bootstrap ou la programmation dans l'EPROM.

Un autre fichier important généré pour le ClassLinker est le fichier de la table des méthodes natives (avec la liste de méthodes natives). Ses nom et position sont spécifiés dans le fichier de l'application de contrôle. Deux fichiers sont générés: la version en langage C et la version en assembleur. Le fichier de la table des méthodes natives fait la liste de toutes les méthodes natives que l'application Java référence. Les méthodes natives qui sont définies déjà dans le paquetage java.lang sont exclues et sont mis en œuvre pour la propre machine virtuelle de java.

4.- Code des méthodes natives.

La plupart de fois, les méthodes natives proportionnent accès aux périphériques hardware qui sont trouvés dans un certain système d'intégrés.

Le code des méthodes natives supporte l'écriture de sousroutines, la plupart d'elles en assembleur. Le ClassLinker engendre le fichier de la table des méthodes natives en enregistrant toutes les méthodes dont l'application a besoin. Le nom de la méthode native doit se trouver avec le nom spécifié dans ce fichier de liste/table.

Toutes les personnes qui développent une application ne doivent jamais changer l'ordre dans la liste des méthodes natives puisque ce fichier fait partie du processus de construction du code natif.

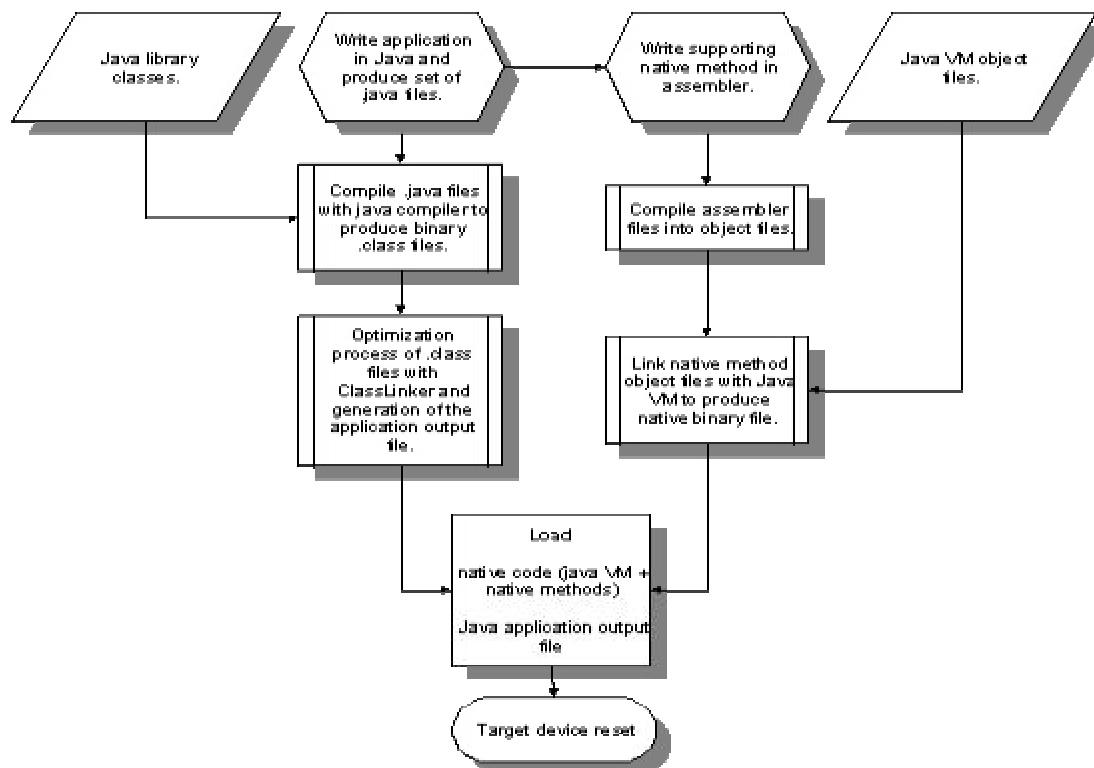
5.- Vérification du logiciel.

Beaucoup de composants d'une application Java peuvent être testés a priori en chargeant une image finale dans la mémoire du dispositif.

Le comportement de beaucoup de méthodes natives peut être simulé par logiciel. Ces techniques permettent au développeur du software de tester et débugger une application avec les outils de développement Java comme par exemple le JDK (dans notre cas). En étant capable de vérifier la plupart des fonctionnalités de l'application que l'on appelle temps de code (temps que coûte le développement de la part du code pour qu'il marche correctement) réduira surtout le temps de développement.

Tester les composants software qui ont besoin d'une interaction hardware est quelquefois plus compliqué. Le mode dans lequel le test se mene à bien dépend de la configuration hardware du dispositif.

Le schéma des étages qu'on doit suivre est le suivant.



L'AIDE DES THREADS.

Les threads de Java sont un moyen idéal pour exécuter plusieurs tâches indépendantes de façon concurrente. La façon de mettre en œuvre ces threads dépend de la plate-forme hardware sur laquelle s'exécute la machine virtuelle Java.

Le mode naturel de mise en œuvre les threads pour un système d'intégrés fondé sur le 68HC11 est d'utiliser les interruptions temps réel. Le temporisateur temps réel peut être configuré pour générer des interruptions par intervalles de 8 à 32 ms, ce qui est dans la plus part des cas suffisant pour mettre en œuvre la commutation de threads "répartis dans le temps" et avec priorité.

Tous les threads sont créés comme "non-daemon threads", ce qui signifie que la machine virtuelle Java finira l'exécution du code seulement si tous les threads sont sortis de la méthode run( ).

Dans cette mise en œuvre de la machine virtuelle Java, tous les threads ont la même priorité. Quand un thread n'a rien à faire, il devra appeler régulièrement au méthode yield( ) pour lancer la commutation de threads et ainsi permettre aux autres threads pouvoir s'exécuter plus fréquemment.

Il y a deux façons pour un programmeur de créer les threads:

1.- En étendant la classe Thread.

Les sousclasses de la classe Thread doivent ignorer la méthode run( ). Par exemple:

```
class CommThread extends Thread
{
        int baudRate;
        CommThread(int baudRate)
        {
           this.baudRate = baudRate;
        }
        public void run()
        {
           while (true)
           {
              // wait for data
              if (charIn())
              {
                 // process data
              }
              else
              {
                 // nothing to do - force thread switching
```

```
                yield();
            }
        }
    }
    private native boolean charIn();
}
```

2.- En mettant en œuvre l'interface Runnable.

Les classes qui mettent en œuvre l'interface Runnable doivent mettre en œuvre aussi la méthode run( ). Le même exemple peut être écrit comme suivant:

```
class CommRun implements Runnable
{
    int baudRate;
    CommRun(int baudRate)
    {
        this.baudRate = baudRate;
    }
    public void run()
    {
        // La même chose que dans l'exemple en haut.
        . . .
    }
    private native boolean charIn();
}
```

Pour maintenir une taille compacte de la machine virtuelle Java, on a pris quelques limitations:

- L'actuelle mise en œuvre de la machine virtuelle Java proportionne une application avec seulement le mécanisme de commutation basique de threads. La synchronisation de threads n'est pas totalement implantée. La synchronisation de méthodes dans les sections de code identifiés avec le mot clé "synchronized" ne marche pas pour l'instant.
- Les méthodes suivants de la classe java.lang.Object ne sont pas mises en œuvre :
  - notify( ).
  - notifyAll( ).
  - Toutes les formes de la méthode wait( ).

On espère que les futures versions de la machine virtuelle de Java éliminent toutes les limitations qui sont mentionnées en haut.

MÉTHODES NATIVES.

Les méthodes natives sont une part importante de l'application java. Les méthodes natives permettent au développeur du software de mettre en œuvre les sections de temps critique du code sous forme native (on soit bien en langage assembleur du 68HC11 ou C) et pourvoir un accès aux composants hardware présents dans le dispositif.

La machine virtuelle Java met en œuvre un schéma d'invocation des méthodes natives qui est rapide et n'utilise pas beaucoup de sources système.

1.-Déclaration des méthodes natives dans la classe d'application.

Dans l'exemple suivant, la méthode native charIn( ) a été déclarée. Cette méthode teste l'état du buffer de réception et elle rend vrai si le buffer contient un caractère à lire:

```
package hw.io;
    class Comm implements Runnable
    {
        public void run()
        {
            while (true)
            {
                // wait for data
                if (charIn())
                {
                    // process data
                }
                else
                {
                    // nothing to do - force thread switching
                    yield();
                }
            }
        }
        private native boolean charIn();
    }
```

2.-Méthodes natives en assembleur.

Il faut suivre les étapes suivantes:

1. Ecrire le code natif en assembleur du 68HC11. Pour l'exemple que nous avons mis en haut charIn( ) ça serai comme ça:

```
ARG2    equ   2   location of the 2nd argument on stack
LO      equ   2   position of low word in 32 bit operand
HI      equ   0   position of high word in 32 bit operand

*************************************************************
* Checks if there are any characters in the receive buffer.
* Java prototype: boolean charIn();
* Input: ARG1 (D)    - pointer to native methods arguments
* ARG2 (SP+2) - pointer to 32-bit return value
*************************************************************

hw_io_Comm_charIn:
    tsx             retrieve ptr. to return value
    ldx   ARG2,x

    clra            set default response as false
    clrb
    std   HI,x
    std   LO,x

    ldd   rx_len   check the receive buffer length
    beq   _ret
```

```
        ldd   #1      set response to true
        std   LO,x

_ret
        rts
```

2. Compiler les méthodes natives en fichiers d'objet pour pouvoir après les linker avec le code de "start-up" et avec la table de méthodes natives, pour produire l'image du code natif.
3. La convention de noms pour les méthodes natives est déterminée par le ClassLinker, qui génère le fichier de la table de méthodes natives en listant toutes les méthodes natives qui ont une référence dans l'application Java. Le nom de la méthode native, qui est spécifié dans le fichier est formé avec le nom de la classe et du paquetage (si celui-ci existe) avec un trait d'union bas (_). Par exemple, si charIn( ) est déclaré dans la classe Comm qui forme part de le paquetage hw.io, alors le ClassLinker génère le nom de la méthode native hw_io_Comm_charIn.

   L'ordre des noms dans la liste du fichier de la table des méthodes natives est très important et personne doit le changer.

3.-Ecriture de méthodes natives en C.

Le prototype de fonction C pour une méthode native est :

```
Void aNativeMéthode (long natParams[], long *returnVal);
```

Où:

- natParams: c'est un pointeur sur un tableau de 32 bits qui est passé à la méthode native quand on l'appelle.
- returnVal: c'est un pointeur sur une valeur de 32 bits. Si la méthode native doit rendre une valeur, ce paramètre indique le lieu où elle se trouve.

  Les méthodes natives ne doivent pas retourner directement une valeur.

  Alors, la méthode charIn( ) serai en C:

```
#define TRUE 1
#define FALSE 0

void hw_io_Comm_charIn(long natParams[], long *returnVal)
{
    *returnVal = (rx_len == 0) ? FALSE : TRUE;
}
```

4.-Limitations des méthodes natives.

- Seuls les types de données primitives de java (boolean, byte, short et int) sont permis pour être passés et rendus par des méthodes natives. Il y a seulement une exception, qui est le "unidimensionnel matrice" des types de données primitives et la classe String pour laquelle la machine virtuelle de Java proportionne des méthodes d'accès.
- Les méthodes natives ne peuvent pas accéder aux champs spécifiques dans le programme java directement.
- Les méthodes natives ne peuvent pas invoquer des méthodes java.
- Le programmeur doit prêter attention quand il accède aux paramètres d'une matrice ou d'une String, puisque les méthodes natives permettent l'accès direct à leurs références. Leurs références ne doivent jamais être changées.

PROBLÈMES RENCONTRÉS AVEC LES MÉTHODES NATIVES.

Pour mettre en œuvre les méthodes natives en C sous la forme que l'on m'avait proposée, nous avions traduit en langage C toutes les méthodes natives que nous avons besoin pour développer le projet, mais au moment de les compiler et ainsi obtenir les fichiers de sortie .S19 et la table de méthodes, nous avions une erreur dans la version du logiciel. Alors, nous avons envoyé un mail au département de SUN qui est responsable de la machine virtuelle Java, pour exposer le problème que nous avions détecté dans sa version.

À cause de ce problème, nous ne pouvions pas exécuter les méthodes natives en C, qui est un langage plus facile à comprendre que le langage assembleur. Voici des méthodes natives que nous avions réecrites en C :

```
#include <stdio.h>
#include "libhc11.h"

#define     TRUE  1
#define     FALSE 0

/* programme principal*/

void CommThread_byteReceived (long natParams[], long *returnVal) {

      char c;

      *returnVal = (read_com(&c)==0)?FALSE:TRUE;
}

void CommThread_getByte (long natParams[], long *returnVal) {

      char c;

      if (read_com(&c)) {
            *returnVal =c ;
      }
}

void CommThread_putByte (long natParams[], long *returnVal) {
```

```
        char c;

        c=natParams[0];
        write_com(c);
        *returnVal = TRUE;
}

void CommThread_setBaudRate(long natParams[], long *returnVal) {

        int i;

        i=natParams[0];
        baudrate(i);
}
```

PAQUETAGES.

JAVA.LANG PAQUETAGE.-

À cause de la limitation de mémoire, seulement a été mise en œuvre une part de la fonctionnalité du paquetage java.lang. Les classes et méthodes qui n’étaient pas remarquables pour le 68HC11 ont été supprimées.

Le programmeur doit tenir compte que l’application Java est compilée avec les classes java.lang qui sont incluses avec la machine virtuelle de Java pour le 68HC11. Ceci est fait, en ajoutant la variable d’environnement Classpath ou en spécifiant le classpath comme argument dans la ligne de commande.

HW.MOT68HC11 PAQUETAGE.-

Le paquetage hardware pour le 68HC11 propose une collection de méthodes basiques pour l'accès mémoire et aux registres hardware du micro 68HC11. La classe additionnelle "Timer" propose des opérations simples de "timing" (temporisation) qui peuvent être utilisées avec l'application.

RESUME DE LA MACHINE VIRTUELLE.

- Dessiné pour s'exécuter comme un petit système qui opère en Java.
- Optimisé spécialement pour s'exécuter sur systèmes avec une petite quantité de mémoire (≤ 32 KB).
- La version actuelle requiert plus ou moins 7.5 KB de mémoire pour le code et moins de 100 bytes de mémoire de données pour s'exécuter.
- On peut mettre en œuvre les threads (mais avec certains limitations), qui sont proposés via le "pre-emptive time slicing". Un CPU à 8MHZ. peut exécuter au moins 4-8 threads en même temps.
- On utilise la classe java.lang pour consommer moins de mémoire.
- On met en œuvre les objets String/StringBuffer.
- Le traitement d'exceptions est complètement mis en œuvre.
- L'interface des méthodes natives est simplifiée pour avoir un temps d'invocation plus rapide.
- On peut configurer les options de démarrage pour pouvoir spécifier la taille du "heap", la taille de la table de références, le nombre de threads, etc...

- La librairie hw.mot68hc11 propose une application Java avec l'accès basique aux quelques composants hardware, comme la mémoire et les registres du contrôle du 68HC11.
- La machine virtuelle peut être construite de façon isolée au code natif. De cette façon, on peut avoir la machine virtuelle programmée dans l'EPROM et le code natif et l'application Java chargé dans la RAM si c'est nécessaire.

LIMITATIONS.

- Le paquetage java.lang a été optimisé (limité) pour inclure seulement les classes et les méthodes qui sont importantes pour la mise en œuvre de la machine virtuelle Java.
- On ne peut pas travailler avec l'arithmétique "long" et "64 bits double".
- La division (/) et le module (%) marchent seulement avec nombres de 16 bits.
- Les threads ne proposent pas la synchronisation de méthodes synchronisées et de sections de code. La classe java.lang.Object ne met pas en œuvre les méthodes de synchronisation de threads wait( ) et notify( ).
- On ne peut pas utiliser les matrices multidimensionnelles.
- Le développement d'applications écrites en Java et exécutées avec la machine virtuelle est moins rapide que celles écrites en C. Cependant, les interruptions sont servies si tôt comme le processeur l'a enregistrée. En plus, il est toujours possible d'écrire des opérations qui sont temps critiques en code natif et sous cette forme on peut compenser la lenteur d'exécution du programme principal.

3.-LA COMMUNICATION API DE JAVA.

L'API (Application Programming Interface) de Java est une interface pour programmer des applications. Mais, qu'est ce que cela signifie? La communication API de Java peut être utilisée pour écrire des applications de communications plates-formes indépendantes, tel que le courrier de voix, fax,...

La version de la communication API de Java avec laquelle nous avons travaillé renferme un support pour le port série RS-232 et les ports parallèles IEEE-1284. Avec la fonctionnalité actuelle, on peut:

- Énumérer les ports disponibles du système
- Ouvrir et réclamer la propriété des ports.
- Résoudre la discussion entre les multiples applications sur la propriété d'un port.
- Exécuter des I/O asynchrones et synchrones avec les ports.
- Recevoir des évènements qui décrivent les changements d'état du port de communication.

Avec cette interface nous avons contrôlé le port série RS-232 de l'ordinateur (PC) avec lequel la carte mère qui contient le micro 68HC11 était connecté. De cette façon, nous configurions tous les paramètres du port, tels que la vitesse (en b/s), bits de parité, bits d'arrêt, et des autres caractéristiques qui sont nécessaires pour une parfaite communication entre les deux dispositifs. De plus, nous utilisions l'API pour pouvoir visualiser quelconque évènement qui puisse succéder dans le

port en cours d'exécution du programme, par exemple l'envoi et re-envoi (ping-pong) de caractères ou au moment d'envoyer une commande que doit exécuter le micro 68HC11.

Pour pouvoir développer tout ceci, nous nous sommes basés sur les exemples d'applications qui étaient dans la version 2.0 de la Comm API.

L'OBJECTIF ET LA FIN DU PROJET.

Une fois que nous nous étions familiarisé avec le microcontrôleur et avec tous les composants de la carte mère, ça veut dire, avec toutes les différentes mémoires qu'il y avait (ROM et RAM, externes et internes), l'échange de données entre les ports et le micro et l'interface entre l'usager et la carte (comment on peut dialoguer) et que nous avions fait les 8 TP proposés et après nous avions fait des autres programmes avec la collaboration d'un ami, qui était en train de faire un DRT et qui était en train de travailler aussi avec le Motorola 68HC11, pour bien comprendre tout le fonctionnement, nous passions à étudier le langage Java. Java a été le support principal avec la machine virtuelle pour le développement du projet. Nous avons fait des programmes en Java jusqu'à la totale compréhension du langage.

Après, nous commençons à comprendre le fonctionnement de la machine virtuelle de Java, avec toutes ses limitations, qui ont été un vrai problème à l'heure de développer les applications en Java, parce que nous ne devions pas penser seulement au langage Java, mais nous le devions faire en considérant les limitations que nous.

Au moment où tout ça a été bien compris, nous avons commencé à voir la communication API de Java. Nous avons vu comment elle travaille avec le port de communication série RS-232, et

nous avons utilisé ce logiciel pour pouvoir contrôler le port qui servait de liaison entre l'ordinateur (notre poste de travail PC) et la carte mère avec le micro de Motorola.

Une fois que nous avions étudié et bien compris tous ces logiciels, langages et les différentes formes d'opérer avec la carte, nous avons développé les applications qui avaient été établies comme objectif de ce projet. À la fin, les résultats du stage sont ces deux programmes en Java:

```
import java.lang.*;
import hw.mot68hc11.*;

class tst
{
   static void main()
   {


      // create instance of communication thread and start it
      CommThread comm = new CommThread();

      // an attempt to initialize the communication port
      try
      {
         comm.initComm(9600);
      }
      catch (Exception excep)
      {
         // fatal error - stop the application running
         return;
      }

      // transmit a welcome message
      comm.send("\r\nHello World !!! ");

      // now, start the communication thread running
      comm.start();

      // Exiting the main() will not stop your application running.
      // ExampleApp will stop running if also all exit their
      // run() method.
   }
}


/**
 * Communication thread that echoes incoming characters.
 * If a Carriage Return (0x0D) is received it transmits also
 * Line Feed (0x0A) character.
 *
 * CommInitException exception is thrown on incorrect initialization
 * parameters.
```

```
 */
class CommThread extends Thread
{
   // static initialization of exception objects
   private static String strExcep = "Wrong baud rate";
   private boolean initialised = false;
   int i;


   /**
    * Method will first check if the correct baud rate has been
    * specified, then it calls native method that performs low
    * level initialization.
    *
    * @param       Baud    Baudrate for the communication port.
    * @exception   CommInitException is thrown when incorrect
    *              baud rate was specified.
    */
   void initComm(int Baud) throws Exception
   {
      if (Baud != 9600)
      {
         throw new Exception();
      }
      setBaudRate(Baud);
      initialised = true;
   }

   /**
   * This method is started when a new instance of CommThread is
   * created. Method will in the infinite loop echo characters that
   * were received via SCI port.
   */
   public void run()
   {
      // do not start thread running until intialisation
      // is done by the initComm() method;
      while (!initialised) ;


      byte comando[] = new byte[5];
      byte cadena[] = new byte[100];
      String mot = "write";
      String apag = "erase";
      int i = 0;
      int encender = 1;
      int apagar = 1;
      int resultado=0;
      String s=new String("");

            send("\r\nWrite whatever you want: ");
      CPU led = new CPU(0x1000);
            led.writeRegByte(0x26,0x80);

      while (true)
      {
```

```
while (byteReceived())
{
   // read the received character
   byte ch = getByte();

   // echo the received character
   putByte(ch);

                   if (ch==0x0D)
           {
           i=0;

                           encender=1;
                           apagar=0;
                   while ((encender==1) && (i<5))
               {
                    if (cadena[i]==mot.charAt(i))
                               {
                                       i++;
                               }
                               else encender=0;

                           }

                           i=0;

                           if (encender==0)
                           {
                                   apagar=1;
                                   while ((apagar==1) && (i<5))
                   {
                       if (cadena[i]==apag.charAt(i))
                                       {
                                               i++;
                                       }
                                       else apagar=0;

                               }

                           }
                       send("\r\nNew Line: ");

                       if ((encender==1) && (cadena[5]==' '))
                       {

                           for(i=6; i<8; i++)
                           {
                                   switch (cadena[i])
                                   {
                                           case '0':
                                           {
                                                   if (i==6)
                                                           resultado=0;
                                                   else
                                                           resultado=resultado+0;
                                                   break;
```

```
}
case '1':
{
	if (i==6)
		resultado=16;
	else
		resultado=resultado+1;
	break;
}
case '2':
{
	if (i==6)
		resultado=16*2;
	else
		resultado=resultado+2;
	break;
}
case '3':
{
	if (i==6)
		resultado=16*3;
	else
		resultado=resultado+3;
	break;
}
case '4':
{
	if (i==6)
		resultado=16*4;
	else
		resultado=resultado+4;
	break;
}
case '5':
{
	if (i==6)
		resultado=16*5;
	else
		resultado=resultado+5;
	break;
}
case '6':
{
	if (i==6)
		resultado=16*6;
	else
		resultado=resultado+6;
	break;
}
case '7':
{
	if (i==6)
		resultado=16*7;
	else
		resultado=resultado+7;
	break;
```

```
}
case '8':
{
        if (i==6)
                resultado=16*8;
        else
                resultado=resultado+8;
        break;
}
case '9':
{
        if (i==6)
                resultado=16*9;
        else
                resultado=resultado+9;
        break;
}
case 'A':
{
        if (i==6)
                resultado=16*10;
        else
                resultado=resultado+10;
        break;
}
case 'B':
{
        if (i==6)
                resultado=16*11;
        else
                resultado=resultado+11;
        break;
}
case 'C':
{
        if (i==6)
                resultado=16*12;
        else
                resultado=resultado+12;
        break;
}
case 'D':
{
        if (i==6)
                resultado=16*13;
        else
                resultado=resultado+13;
        break;
}
case 'E':
{
        if (i==6)
                resultado=16*14;
        else
                resultado=resultado+14;
        break;
```

```
                                                  }
                                                  case 'F':
                                                  {
                                                        if (i==6)
                                                              resultado=16*15;
                                                        else
                                                              resultado=resultado+15;
                                                        break;
                                                  }

                                            }
                                      }
                                }

                            if (encender==1)
                        {
                         led.writeRegByte(0, resultado);
                            encender=1;
                                        apagar=1;
                              }
                        else if (apagar==1)
                        {
                            led.writeRegByte(0, 0x00);
                                  apagar=1;
                                        encender=1;
                        }

                                  i=0;
                  }
                  else
                  {
                      cadena[i]=ch;
                      i++;
                  }
                        if (ch=='\010')
                        {
                                  i=i-2;
                        }


              }
          }
}



  /**
   * Writes a string message to COM port.
   *
   * @param   Msg     message that will be transmitted
   * @return  true if message has been put into the transmit buffer,
   *         otherwise false is returned
   */
  boolean send(String Msg)
  {
    if (Msg == null || Msg.length() == 0)
```

```
        return false;

    for (int i=0; i < Msg.length(); i++)
    {
        if (putByte((byte)Msg.charAt(i)) == false)
            return false;
    }

    return true;
}

/**
 * Writes a character array to COM port.
 *
 * @param   Msg     Reference to character array
 * @return  true if message has been put into the transmit buffer,
 *          otherwise false is returned
 */
boolean send(char[] Msg, int Start, int Length)
{
    if (Msg == null || Msg.length == 0 || Start+Length > Msg.length)
        return false;

    for (int i=0; i < Length; i++)
    {
        if (putByte((byte)Msg[Start+i]) == false)
            return false;
    }

    return true;
}

/**
 * Writes a byte to SCI port.
 *
 * @param   Val     byte value that has to be transmitted
 * @return  true if byte has been put into the transmit buffer,
 *          otherwise false is returned
 */
private native boolean putByte(byte Val);

/**
 * Reads the byte from the receive buffer. Prior calling this
 * method programmer must make sure that there are characters
 * available in the receive buffer by calling the byteReceived
 * method.
 *
 * @return  character retrieved from the receive buffer.
 *          If there are no characters available zero will be
 *          returned.
 */
private native byte getByte();

/**
 * Checks if there are any bytes in the receive buffer.
 *
```

```
 * @return  true if there is at least one unread character,
 *         otherwise false is returned
 */
private native boolean byteReceived();

/**
 * Performs low level initialization of SCI port and sets the
 * baud rate to requested value.
 *
 * @param   Baud    SCI baud rate value
 */
private native void setBaudRate(int Baud);
}
```

et l’autre programme qui a été développé est :

```
import java.io.*;
import java.util.*;
import javax.comm.*;

public class SimpleWrite {
   static Enumeration portList;
   static CommPortIdentifier portId;
   static String messageString = "Write 55";
   static SerialPort serialPort;
   static OutputStream outputStream;

   public static void main(String[] args) {
      portList = CommPortIdentifier.getPortIdentifiers();

      while (portList.hasMoreElements()) {
         portId = (CommPortIdentifier) portList.nextElement();
         if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
            // if (portId.getName().equals("COM2")) {
            if (portId.getName().equals("/dev/term/a")) {
               try {
                  serialPort = (SerialPort)
                     portId.open("SimpleWriteApp", 2000);
               } catch (PortInUseException e) {}
               try {
                  outputStream = serialPort.getOutputStream();
               } catch (IOException e) {}
               try {
                  serialPort.setSerialPortParams(9600,
                     SerialPort.DATABITS_8,
                     SerialPort.STOPBITS_1,
                     SerialPort.PARITY_NONE);
               } catch (UnsupportedCommOperationException e) {}
               try {
                  outputStream.write(messageString.getBytes());
               } catch (IOException e) {}
            }
```

```
      }
    }
  }
}
```

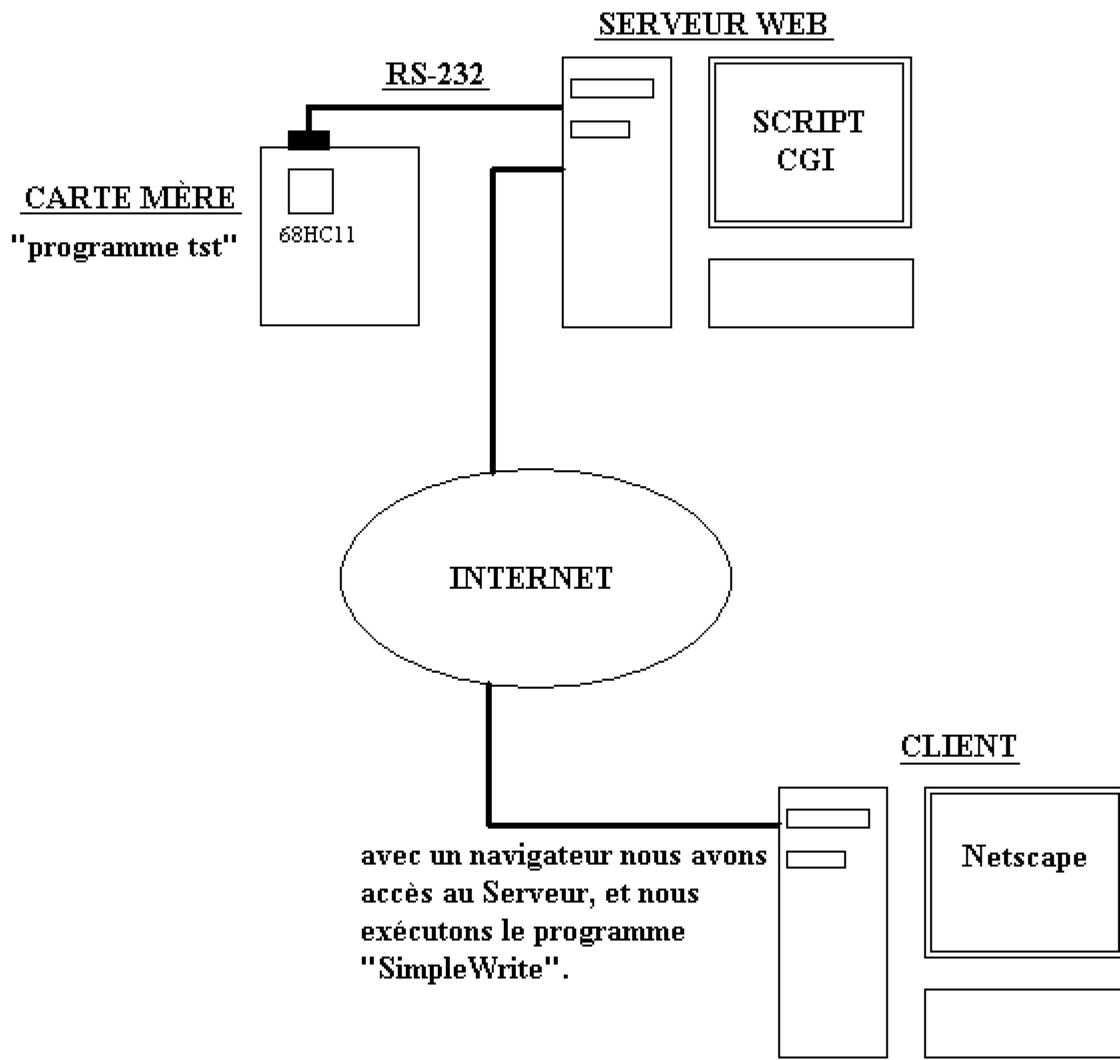
Le premier est l'application en Java qui a été inséré avec la machine virtuelle de Java dans le micro et qui est le programme principal puisqu'il exécute ce que nous voulons, ce que nous envoyons à travers des commandes qui voyagent par le port série entre le PC (qui va faire d'interface entre nous (l'usager et la carte avec le micro) et la carte mère.

Le deuxième est une application qui est fondé dans la COMM API et qui contrôle le port série (sa vitesse, les bits de parité, d'arrêt,...) et qui fait aussi l'envoi des commandes que nous voulons à travers du port.

Dans notre cas, les commandes que nous envoyons sont des commandes pour illuminer les leds qu'il y a dans la carte du micro et pour les éteindre. Ceci est seulement un exemple, parce que nous pourrions faire de même si nous avions eu un capteur de température, que le micro va acquérir. Le principe est d'avoir fait un programme qui donne un ordre au micro pour exécuter ce que nous voulons, et que cet ordre est bien accompli.

De plus, à travers une architecture client/serveur, nous avons installé notre programme d'application (celui de la COMM API) dans un serveur web qui est connecté via la RS-232 avec la carte du micro. Maintenant nous pouvons accéder avec n'importe quel ordinateur qui est connecté au réseau Internet, pour exécuter la commande que nous voulons dans la carte du microcontrôleur 68HC11.

Le schéma d'utilisation de ces logiciels et la forme de travail pourrait être le suivant:


SERVEUR WEB
RS-232
SCRIPT
CGI
CARTE MÈRE
"programme tst"
68HC11
INTERNET
CLIENT
Netscape
avec un navigateur nous avons
accès au Serveur, et nous
exécutons le programme
"SimpleWrite".

CONCLUSIONS

Comme conclusion de ce stage, je peux dire qu'il a été une expérience très bonne pour ma propre formation comme ingénieur puisque je n'ai jamais fait quelque chose comme ceci, parce qu'en Espagne il n'y a pas ces stages, et pourtant, ces trois mois que j'ai passés ici dans le laboratoire de l'ENSERB avec mon tuteur, Patrice KADIONIK, ont été très très positifs.

En plus, la façon que M. Kadionik a eu de mener à bien le projet, avec tous les étapes que nous avons fait, tous les deux, pour bien envisager toutes les difficultés qui se sont posés, et les très bons résultats que nous avons obtenus à la fin, tout cela fait beaucoup plaisir quand on regarde ce qui était fait au début du stage et que l'on voit tout ce que nous avons fait.

C'est le moment aussi de dire que toutes les choses n'ont pas été faciles. Nous avons eu beaucoup de difficultés, surtout dans tout ce qui concerne la machine virtuelle, puisque c'est un sujet très récent et ça signifie qu'il y a encore beaucoup d'erreurs (comme par exemple ce que nous avions trouvé dans les méthodes natives écrites en langage C) et beaucoup de choses de compréhension difficile, mais que grâce à notre effort et notre persévérance nous avons su les surpasser.

Le bilan que je fais de ce stage est vraiment positif et très bon, et que j'ai trouvé un projet très intéressant et très bien mis au point.

J'utilise cet espace pour remercier encore une fois à Patrice KADIONIK pour tout son temps et toute sa compréhension envers moi et pour l'opportunité qu'il m'a donnée pour faire ce joli stage.