- ENSEIRB -

ÉCOLE NATIONALE SUPÉRIEURE
ÉLECTRONIQUE, INFORMATIQUE & RADIOCOMMUNICATIONS
BORDEAUX

ETUDE ET MISE EN ŒUVRE DU MICROCONTROLEUR 68HC11

Patrice KADIONIK
kadionik@enseirb.fr
www.enseirb.fr/~kadionik

TABLE DES MATIERES

AVANT-PROPOS

Ces Travaux Pratiques sont avant tout destinés à tous ceux qui désirent en savoir plus sur le développement d'applications basées sur l'utilisation du microcontrôleur 8 bits 68HC11. Un microcontrôleur 8 bits, cela semble vraiment peu puissant et dépassé par rapport aux processeurs 32 et maintenant 64 bits. Mais ils trouvent largement leur 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, voiture, domotique…leur étude est donc toujours d'actualité et reste un bon tremplin pour l'étude de microcontrôleurs plus puissants (ColdFire de Motorola…)…

Durant votre carrière d'ingénieur, vous serez amené(e) à travailler sur des documents écrits dans leur grande majorité dans la langue de Shakespeare. Ce TP se base sur de nombreux documents écrits en anglais que vous retrouverez dans les annexes et essaye de vous placer dans ce contexte professionnel. Ce que vous avez pu voir sur le microprocesseur 68000 vous aidera sûrement et toute analogie possible sera bénéfique…

Nous verrons quelques moyens de développement d'applications 68HC11 : en assembleur, avec le langage C, l'utilisation de bibliothèques d'I/O et la programmation multitâche avec un noyau temps réel (µC/OS II) en 3ème année.

Ce TP requiert une connaissance de base sur le langage C et l'architecture d'un microprocesseur (68000 à l'ENSEIRB)

Ce texte a été rédigé à partir d'un rapport de stage IUT de Y. Benaben (1998/99) que je tiens ici à remercier et qui est sorti de l'ENSEIRB en 2002.

P. Kadionik
04/03/03

PARTIE I

- PRESENTATION GENERALE -

1. PRESENTATION DE LA CARTE MERE ENSEIRB 68HC11

1.1. Caractéristiques du microcontrôleur 68HC11

Le processeur employé est un microcontrôleur 68HC11A1 8 bits de Motorola dont les caractéristiques essentielles sont résumées sur le schéma suivant (voir annexe 2) :

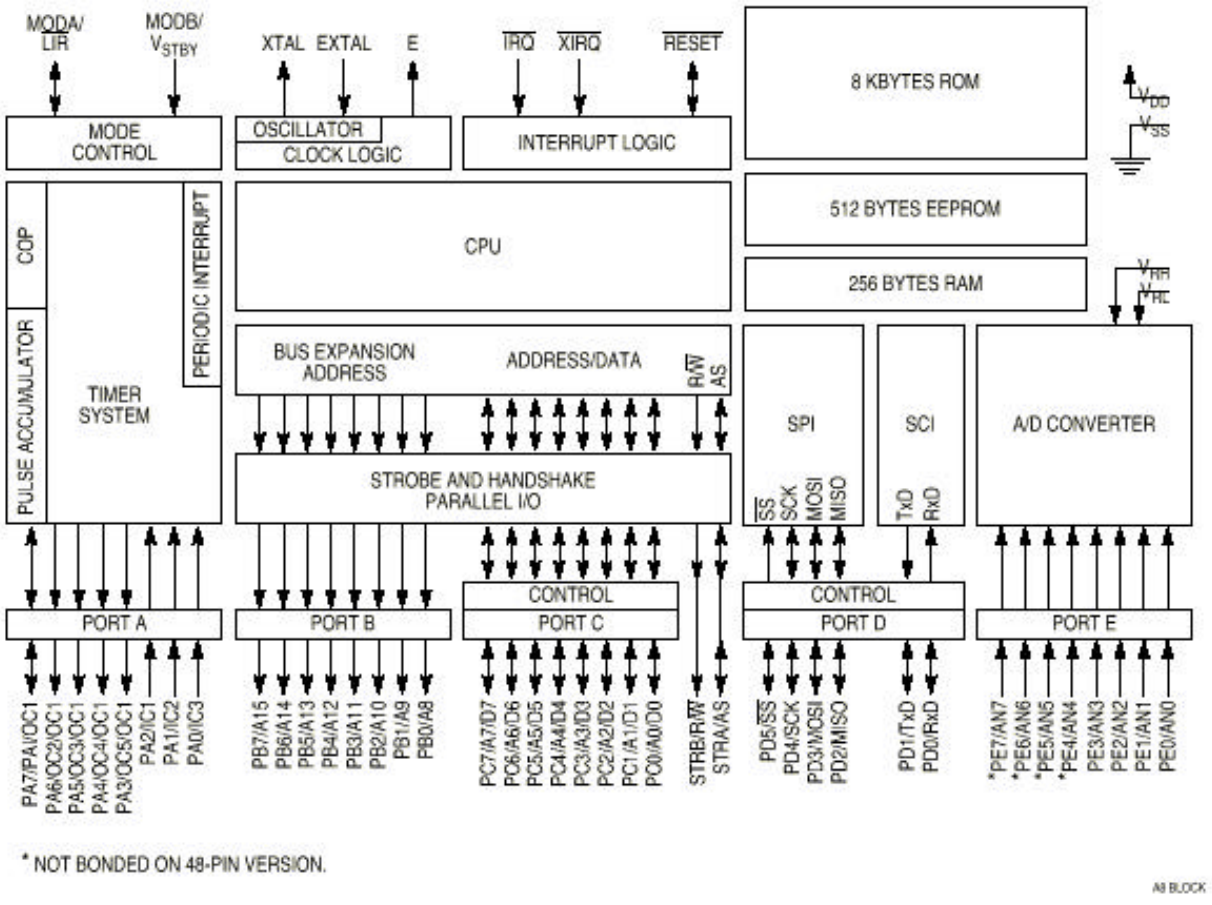


Figure 1 : Architecture interne du 68HC11.

Les caractéristiques de l'ensemble des microcontrôleurs de la famille 68HC11 sont présentées dans le document donné en annexe 1. On y trouvera aussi la liste exhaustive de l'ensemble des documents de référence.

On peut souligner les éléments suivants :

- microcontrôleur en technologie HCMOS.
- 8Ko de ROM.
- 512 octets de EEPROM.

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

Le document donné en annexe 3 précise ce qu'est un microcontrôleur. Après lecture de ce document, vous devez être capable de faire la différence entre un microcontrôleur (68HC11) et un microprocesseur (68000).

Les modes de fonctionnement du 68HC11 sont résumés dans le tableau suivant (voir annexe 5) :

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

La carte mère ENSEIRB réalisée permet l'utilisation du microcontrôleur 68HC11 dans tous les modes grâce à la présence de **2 interrupteurs MODA et MODB** qui fixent la valeur de MODA et MODB vus précédemment.

1.2. Schéma électrique de la carte mère

Le schéma électrique de la carte mère est donné ci-après.

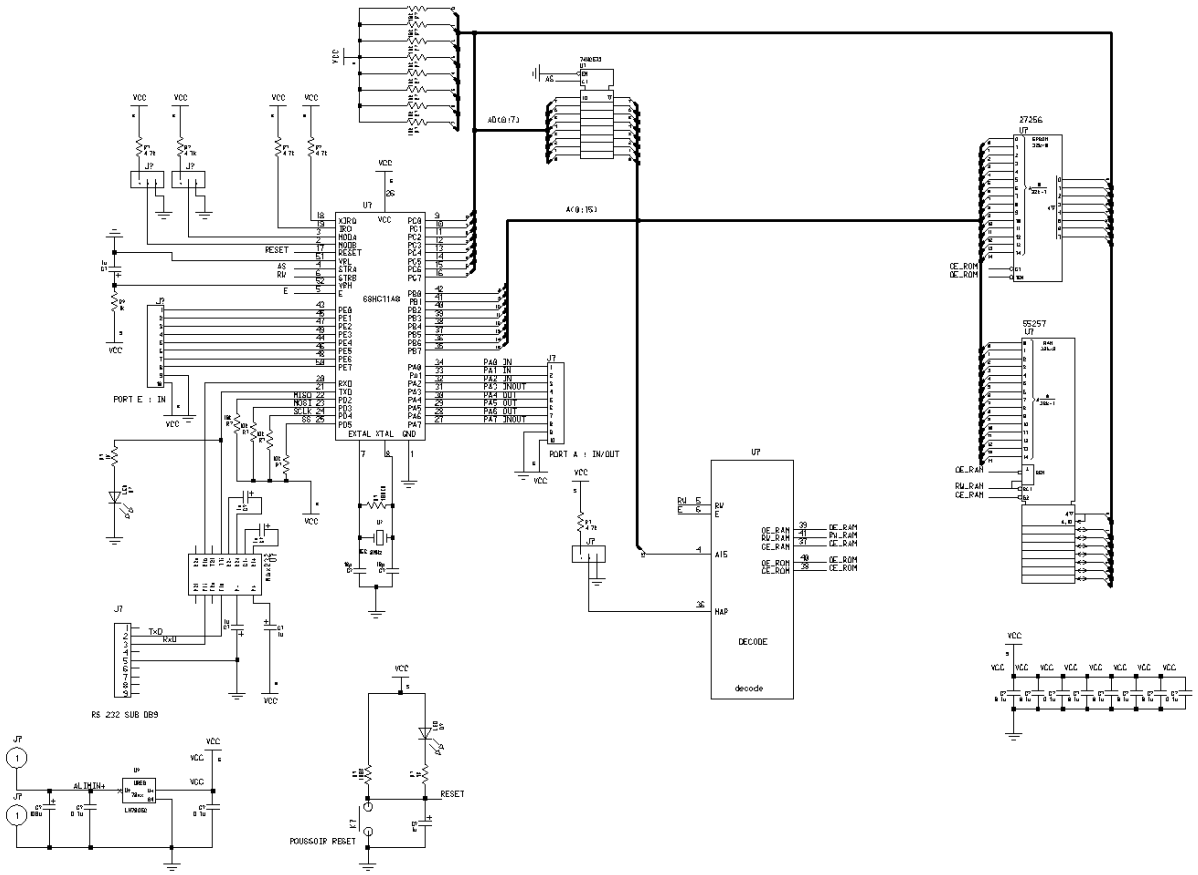


Figure 2 : Schéma électrique de la carte mère

Si le microcontrôleur travaille en mode étendu, l'utilisateur dispose des ressources externes qui sont une RAM et une ROM externes de 32 Ko. Des ports du 68HC11 sont alors attribués aux fonctions d'interfaçage de composants extérieurs :

- Le PORT C est multiplexé : selon la valeur du signal AS, le PORT C est soit le bus bidirectionnel des données (D0…D7) soit le poids faible du bus d'adressage 16 bits (A0…A7). Le démultiplexage est ici confié au circuit 74LS573.
- Le PORT B est le poids fort du bus d'adressage (A8…A15).

L'adressage sur 16 bits permet donc d'adresser directement 64 Ko.

1.3. Décodage d'adresse

Le décodage d'adresse est réalisé par un circuit programmable ALTERA dont le schéma symbolique est le suivant :

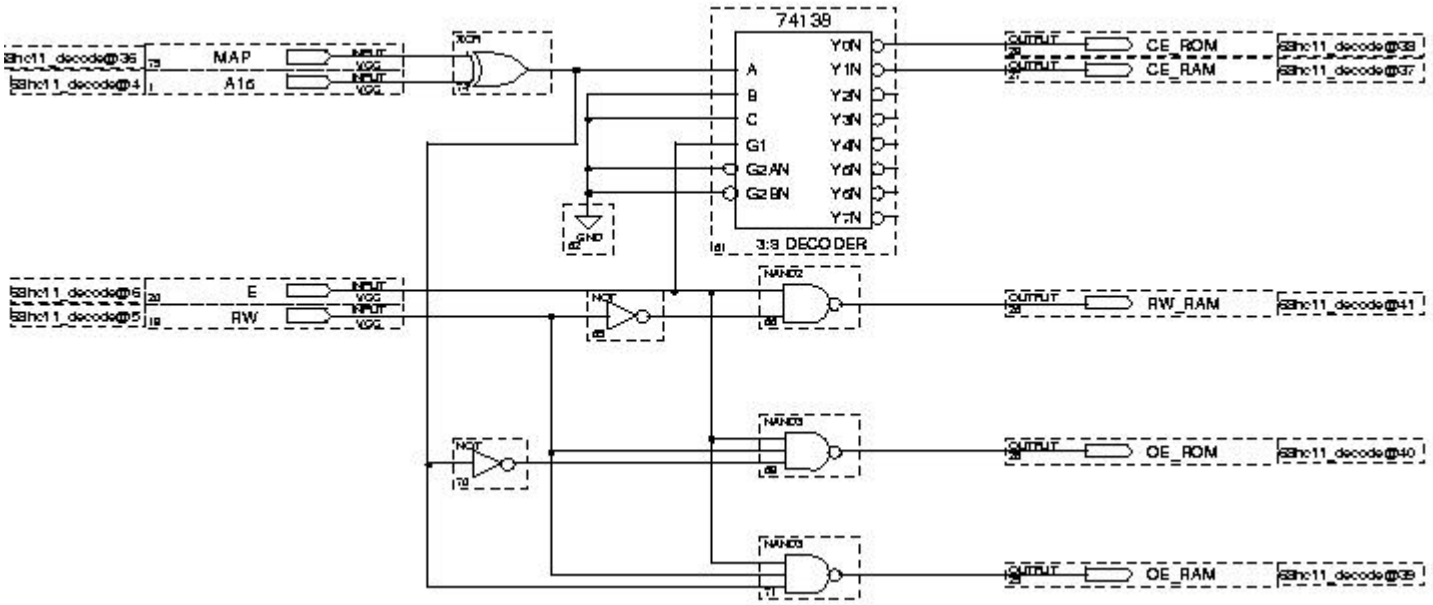


Figure 3 : Schéma du décodage d'adresse

On peut remarquer que le schéma comporte un décodeur d'adresse 74LS138 ainsi qu'un circuit logique destiné à activer les CS des boîtiers mémoire et la broche R/W de la RAM. On remarque également qu'une entrée supplémentaire appelée MAP reliée à une porte XOR a été rajoutée afin de remplacer la mémoire RAM par la mémoire ROM et vice-versa selon le programme à implanter dans la page d'adresse $8000-$FFFF. La valeur de MAP est fixée par un **interrupteur MAP** sur la carte mère.
Un composant programmable a été introduit ici pour pouvoir le reprogrammer par la suite car la carte mère est utilisée en projet de fin d'étude, la « glue logique » de la partie spécifique du projet pouvant alors naturellement s'incorporer dans le composant.

1.4. Cartographie mémoire de la carte mère

La cartographie mémoire (mapping mémoire) du microcontrôleur 68HC11Ax (A0, A1 et A8) est la suivante :

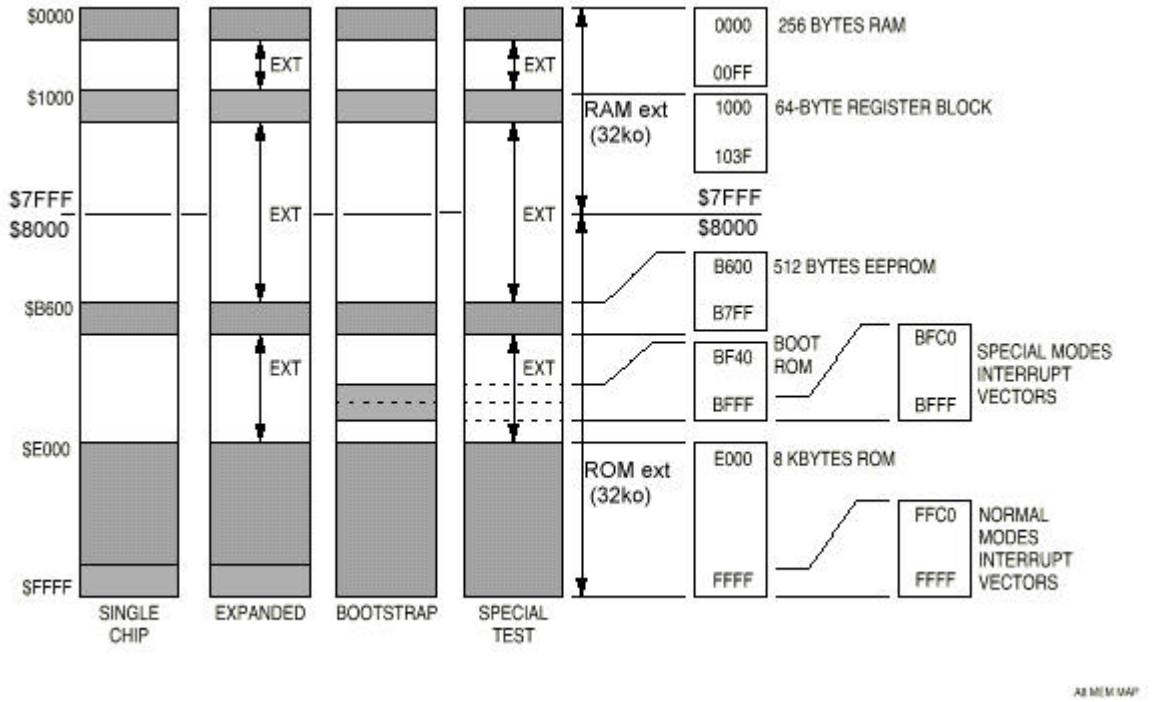


Figure 4 : Cartographie mémoire du 68HC11 dans tous les modes de fonctionnement

Remarque : *il est important de noter que les accès aux ressources internes (RAM interne, EEPROM, registres...) sont prioritaires vis-à-vis des accès externes. De plus, la configuration d'origine du bit IRV du registre HPRIO permet de rendre "invisibles" les accès internes non reflétées sur le bus externe d'adresses.*

Le 68HC11A1 est ici utilisé de 2 manières :

- Avec le logiciel **PCBUG11** de Motorola, on travaille en mode bootstrap puis en mode special test. Ceci permet de déverminer (« débugger ») le code du programme à tester par l'intermédiaire d'un jeu de commandes (voir annexe 6). 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 User's Fast Friendly Aid to Logical Operation »), on travaille en mode étendu puisque le moniteur est programmé dans l'EPROM externe. Le moniteur par l'intermédiaire d'un jeu de commandes (voir annexe 7), permet alors le débuggage grâce à un simple terminal ou un émulateur de terminal sur ordinateur.

Nous verrons plus en détail comment nous utilisons ces 2 logiciels par la suite.
Il faut noter que le jeu de commandes de ces 2 logiciels est semblable à celui du kit 68000. Motorola , quel que soit le type de ses processeurs, propose un moniteur avec les mêmes commandes, ce qui facilite la vie de l'utilisateur en passant d'un processeur à un autre…

2. PRESENTATION DE LA CARTE D'ENTREES/SORTIES

Le microcontrôleur 68HC11Ax en mode étendu ne possède que 2 ports libres : le PORT A et le PORT E. Le PORT E ne pose pas de problème particulier puisque tous les bits du PORT E sont accessibles en entrée (la fonction de conversion A/N n'est pas utilisée ici). Le PORT A nécessite une attention plus particulière notamment au niveau du registre de contrôle PACTL qui configure le TIMER partagé avec le PORT A. Nous choisissons alors d'utiliser toutes les broches possibles en sortie. La carte d'E/S est figée de la manière suivante :

PORT E :	**entrées**	**= PE0 à PE7**
PORT A :	**entrées**	**= PA0, PA1, PA2**
	sorties	**= PA3, PA4, PA5, PA6, PA7**

Le schéma électrique est donné ci-dessous : on retrouve les 2 connecteurs du PORT A et PORT E. Le PORT E est interfacé avec un boîtier DIL de 8 micro-switchs dont les sorties sont reliées à des résistances de pull down de 10 KOhms. Le PORT A est partagé en 2 : Les 3 bits de poids faible sont reliés de la même manière que le PORT E grâce à un boîtier micro-switch de 4 interrupteurs, les 5 bits de poids fort sont reliés à un circuit buffer 74HC245 qui alimente 5 LEDs de visualisation.

Il faut en outre noter que la sortie PA3 est utilisée par le moniteur BUFFALO et par le noyau temps réel µC/OS II et est ainsi dans ces 2 cas indisponible.

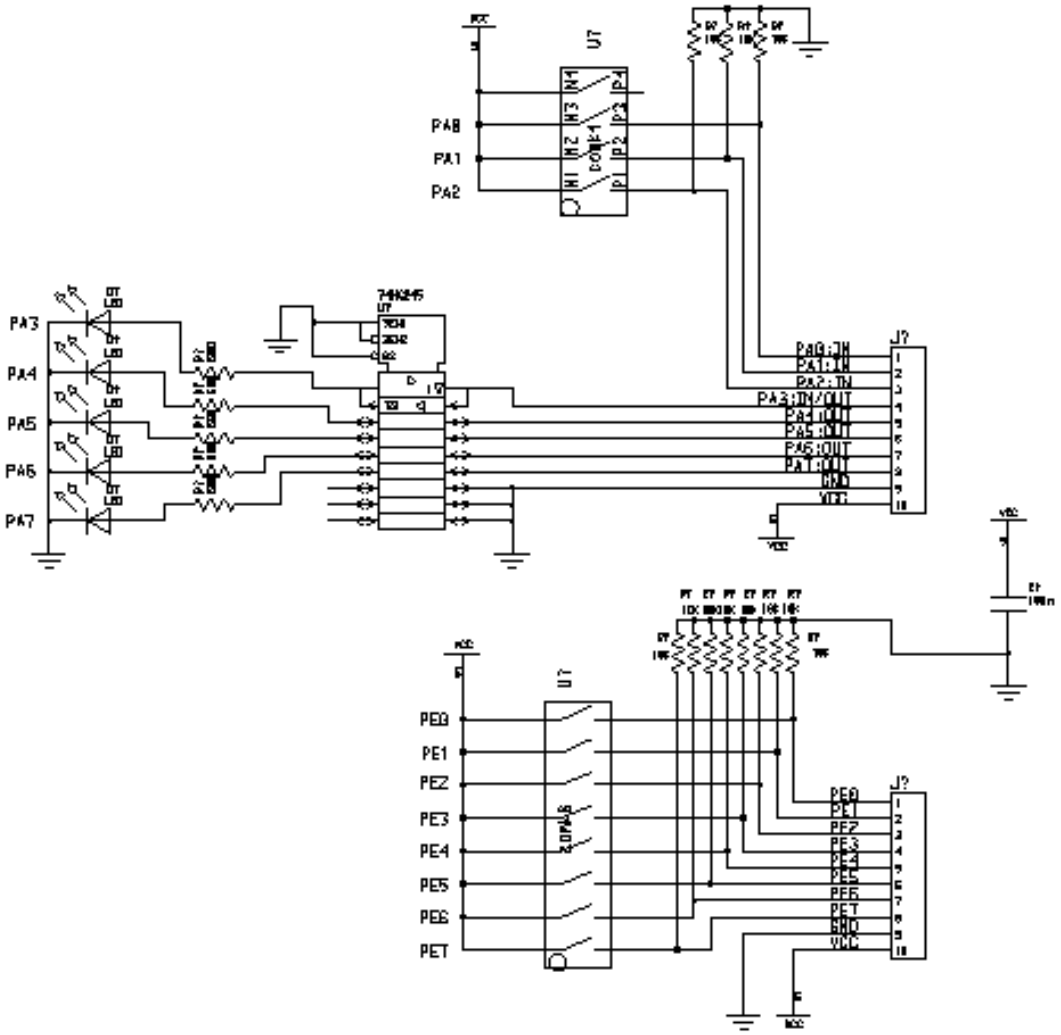


Figure 5 : Schéma électrique de la carte d'E/S

3. UTILISATION DE PCBUG11

3.1. Fonctionnement du 68HC11 en mode bootstrap avec PCBUG11

Avant d'aborder la programmation du microcontrôleur 68HC11 nous allons voir un peu plus en détail le fonctionnement du logiciel PCBUG11.

PCBUG11 est un petit 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 hexadécimal) permet de définir l'adresse de base du programme dans la mémoire. Il faut donc placer son programme dans une zone libre et inscriptible. C'est le cas de la RAM interne ou externe et de l'EEPROM interne. L'EEPROM nécessite un protocole de programmation un peu plus complexe que la RAM mais l'avantage est que le code du programme sera conservé même si l'alimentation du microcontrôleur est coupée.
Le cas d'un programme écrit en langage évolué type C peut se ramener au cas précédent sachant que l'on précise l'adresse de base de téléchargement lors de la phase d'édition de liens.

Pour la suite, il faut examiner le fonctionnement du microcontrôleur. En mode « bootstrap », il apparaît une « boot ROM » en $BF40-$BFFF, zone de mémoire morte à laquelle le processeur saute après le reset. Cette mémoire de boot présente dans la cartographie uniquement en mode « bootstrap » et « special test », contient une petite routine qui autorise le téléchargement de 256 octets depuis le port série vers la RAM à partir de l'adresse $0000. En effet, ce programme en « boot ROM » configure la liaison série à 9600 b/s, 1 bit de start, 8 bits de données, 1 bit de stop (pas de contrôle de flux ni de parité).
Le programme de boot offre également la possibilité de télécharger les 256 octets à une vitesse de 1200 b/s par l'intermédiaire d'une routine interne de la ROM appelée autobaud qui détecte la vitesse du téléchargement grâce à la réception du caractère $FF. C'est donc à ce moment là que le logiciel PCBUG11 envoie un petit programme de 192 octets environ qui est appelé TALKER. Comme son nom l'indique, ce petit programme va permettre de communiquer avec l'ordinateur hôte et ainsi d'exécuter les commandes de PCBUG11 (voir annexe 5)

Le logiciel PCBUG11 présente l'interface graphique suivante :

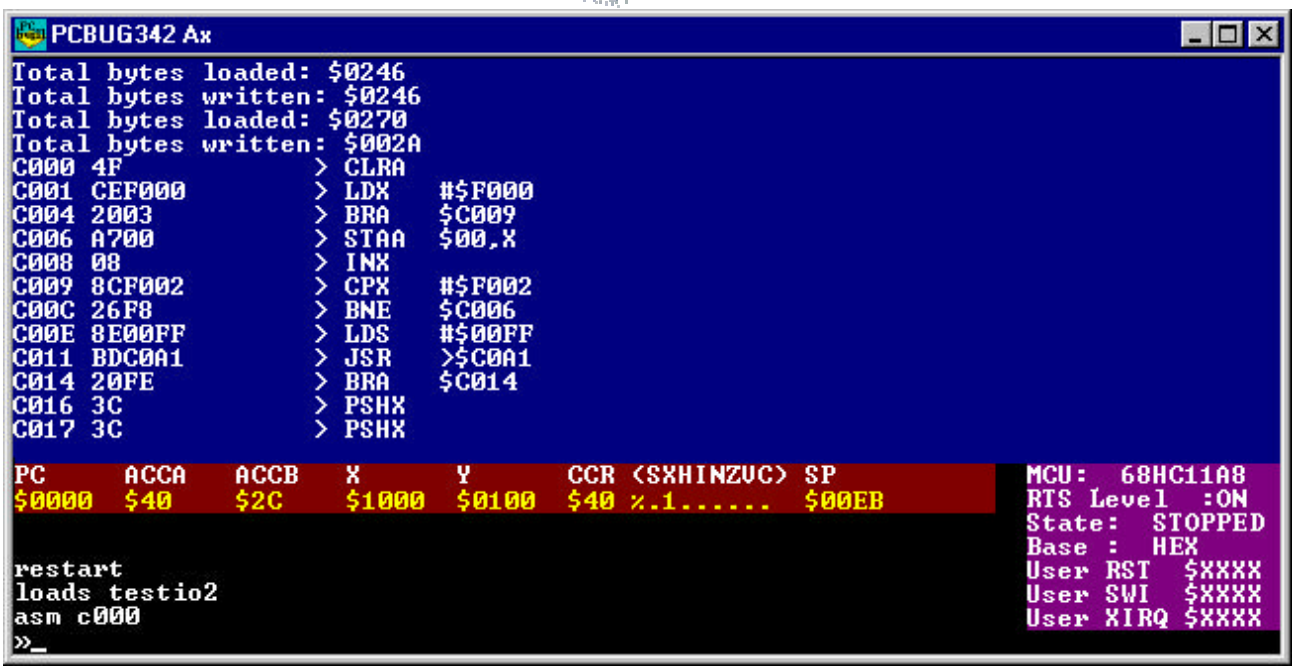


Figure 6 : Logiciel PCBUG11 en cours d'utilisation

La zone supérieure est la zone de dialogue où s'affiche le résultat des opérations lancées par l'utilisateur.
La zone centrale au milieu montre l'état des registres du 68HC11 et à droite le type de processeur ainsi que le mode de fonctionnement de PCBUG11 (Running, Stopped, Trace).
La partie inférieure est réservée à l'utilisateur pour rentrer les commandes de PCBUG11.

Voici une brève description des commandes essentielles de PCBUG11 (voir annexe 6) :

- *ASM addr* : permet le désassemblage/assemblage en ligne du code en mémoire à l'adresse addr.
- *BF addr1 addr2 byte|word* : permet de remplir un bloc de mémoire commençant à addr1 et terminant par addr2 avec la valeur byte / word.
- *BR addr [macroname]* : cette fonction permet de placer des points d'arrêt autorisant ainsi à l'utilisateur de débugger son programme en pas à pas. Cette fonction est utilisable seulement si l'on a répondu No à la question "Do you wish use the XIRQ interrupt ?" au lancement de PCBUG11. Le programme doit être également implanté dans une zone de mémoire accessible en écriture puisque PCBUG11 place une instruction SWI (interruption logicielle) à l'adresse du point d'arrêt décalant ainsi tout le reste du programme en mémoire. Le paramètre [macroname] est facultatif, il permet d'exécuter une macro préalablement chargée lors d'un point d'arrêt.
- *EEPROM 0|addr1 addr2* : permet de supprimer une plage EEPROM (paramètre 0) ou bien de la configurer en spécifiant l'adresse de départ et d'arrivée.
- *EEPROM ERASE [bulk]* : permet d'effacer partiellement ou dans son intégralité l'EEPROM dont l'étendue est définie grâce à EEPROM addr1 addr2.

- *G addr* : exécute le programme à l'adresse addr.
- *HELP* : affiche toutes les fonctions de PCBUG11 ainsi qu'un commentaire.
- *LOADS filename* : permet de télécharger un fichier SRecord (.S19) en mémoire.
- *MD startaddr [endaddr]* : visualise la plage de mémoire s'étendant de startaddr à endaddr.
- *MM addr* : modifie le contenu de la case mémoire à l'adresse addr.
- *NOBR addr* : supprime le point d'arrêt placé à l'adresse addr.
- *QUIT Y* : quitte le logiciel PCBUG11.
- *RD* : rafraîchit l'affichage des registres dans la fenêtre centrale.
- *RESTART* : permet de relancer PCBUG11 si une erreur de communication est apparue (on peut aussi utiliser CONTROL R soit ^R).
- *^B* : permet d'envoyer un BREAK sur la liaison série.

La liste complète de toutes les instructions ainsi que leur mode d'emploi est disponible dans le manuel de PCBUG11 (cf. bibliographie  et l'annexe 6).

3.2. Téléchargement d'un programme objet avec PCBUG11

Lorsque l'utilisateur désire télécharger un programme dans la RAM ou l'EEPROM de la carte mère, il doit procéder comme suit :

- Effectuer un RESET sur la carte cible.
- Lancer PCBUG11.

 ***Remarque*** *: le fonctionnement de PCBUG11 sous DOS ne nécessite pas de configuration particulière de l'ordinateur . En revanche l'utilisation sous Windows 95 / 98 nécessite les configurations suivantes :* ***pas de contrôle de flux matériel*** *(pour le port de communication COM1 ou COM2) et* ***sensibilité d'attente basse*** *(dans l'onglet propriété « divers » du programme PCBUG11).*

Dans le cas d'un téléchargement en EEPROM, taper au préalable les commandes suivantes :

- *EEPROM $B600 $B7FF* : si l'EEPROM est située de l'adresse $B600 à $B7FF.

➔*EEPROM ERASE BULK* : permet d'effacer l'EEPROM. Nous pouvons ainsi vérifier si l'opération a été réussie en tapant :

➔*MD $B600 $B7FF* : on ne doit voir que des octets à $FF.

 ***Remarque importante*** *: si la programmation de la mémoire EEPROM n'est pas correcte, il est possible, soit qu'elle ne se trouve pas aux adresses $B600 à $B7FF, soit que le registre de protection en écriture BPROT (présent sur les 68HC11Ex, F1...) ait été configuré pour protéger l'EEPROM. Pour résoudre ce problème, il faut, après avoir lancé PCBUG11, taper* ➔*MM $1035 00 : où $1035 est l'adresse du registre BPROT.*
Si l'EEPROM n'est pas présente dans la cartographie mémoire, c'est que le registre CONFIG est mal configuré : attention, ce registre se programme comme une case de mémoire EEPROM... (voir le manuel "M68HC11 REFERENCE MANUAL", page 3-5, paragraphe 3.2.2. Le bit concerné est dénommé EEON et permet d'activer ou non l'EEPROM).

➔*LOADS filename* : filename est le nom du fichier (sans l'extension) SRecord de Motorola (.S19) qui se trouve ici dans le même répertoire que PCBUG11 (utiliser sinon la commande ➔*DOS CD c:\...* pour changer de répertoire). Les fichiers S19 seront créés avec le programme assembleur **ASMHC11.EXE** ou bien par un compilateur croisé C (C Cosmic).

➔*ASM point_d_entrée* : vérifie que le programme est bien implanté en mémoire en déassemblant/assemblant en ligne (point_d_entrée=$B600 si le programme est en EEPROM).

➔*G point_d_entrée* : exécution du programme ligne (point_d_entrée=$B600 si le programme est en EEPROM).

➔*S* : le programme est stoppé, le 68HC11 revient au TALKER.

La note d'application AS-67 de Motorola indique comment on peut accéder aux ressources externes tout en utilisant PCBUG11. En effet, PCBUG11 ne marche qu'en mode « bootstrap ». En revanche, il est possible de basculer en mode « special test » sous PCBUG11 en écrivant $E5 à l'adresse $103C (adresse du registre HPRIO). On positionne ainsi les bits SMOD et MODA à 1 (« special test ») et le bit IRV reste à 0.

➔***BF 103C E5 :*** **validation des accès externes depuis le mode bootstrap sous PCBUG11.**
A partir de ce moment, il est possible d'accéder aux ressources externes du 68HC11 et donc de télécharger des programmes en RAM externes depuis PCBUG11.

PCBUG11 permettra le téléchargement des fichiers SRecord en mémoire de manière simple. En revanche, PCBUG11 a quelques inconvénients tels que l'occupation d'une grande partie de la RAM interne ainsi que l'obligation de démarrer le 68HC11 en mode bootstrap.

De plus **l'utilisation de la liaison série par le programme utilisateur rompt la liaison** entre le TALKER et PCBUG11. **On ne peut dans ce cas faire des opérations d'E/S sur la liaison série avec PCBUG11**. Dans ce dernier cas, il vaut mieux utiliser le moniteur BUFFALO.

4. OUTIL JBUG11

L'outil JBug11 est une version améliorée de PCBUG11 fonctionnant sous Windows. Il possède donc une interface graphique conviviale et permet de faire les mêmes choses que PCBUG11.
L'interaction avec JBug11 se fait par les menus ou à partir de la fenêtre de commandes que l'on peut taper.

Il est à noter que JBug11 est compatible au niveau des commandes en ligne (BF, MD,…)

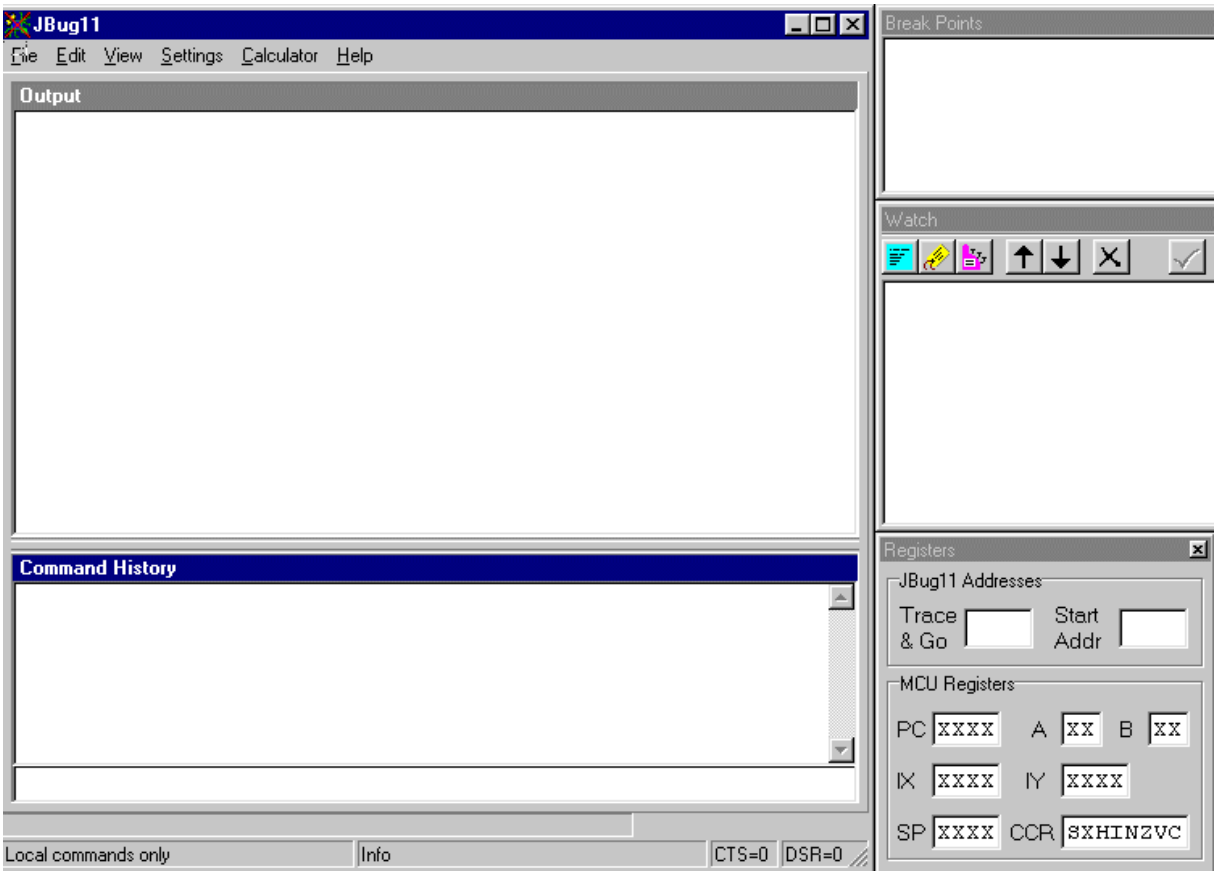


Figure 7 : Logiciel JBug11

5. REALISATION DE PROGRAMMES ASSEMBLEUR ET C

5.1. Programmation en assembleur 68HC11

Le jeu d'instructions est présenté dans le manuel « M68HC11 REFERENCE MANUAL - APPENDIX A ». Il ressemble beaucoup à celui du microprocesseur 68000. On se réfèrera au document donné en annexe 4 pour une description plus complète de la programmation en assembleur 68HC11 (ainsi que l'annexe 2)…

La production d'un objet à partir d'un programme assembleur 68HC11 est la suivante :

- Edition du fichier source : extension du fichier **.ASC** obligatoirement. Une directive d'assemblage ORG doit être impérativement utilisée.
- Assemblage à l'aide de la commande ASMHC11.EXE .
  ex : c:\> ASMHC11 TOTO.ASC
- Le fichier SRecord a été créé (pour l'exemple : TOTO.S19) ainsi qu'un fichier listing pour localiser les erreurs éventuelles (pour l'exemple : TOTO.LST).
- Téléchargement dans la carte mère.

Un exemple de fichier écrit en assembleur 68HC11 est donné ci-après. On notera les fortes analogies avec l'assembleur 68000.

```
PORTA   EQU     $1000       * port A data
PORTE   EQU     $100A       * port E data
PACTL   EQU     $1026       * Pulse Acc. Control Register
                          *  bit  7  :  DDRA7  0  :  input,  1  :
ouput
        ORG     $B600
        LDS     #$FF        * initialisation de la pile
        LDAA    #$80        * DDRA7 (PA7) : sortie
        STAA    PACTL

CHENILLE
        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 a
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 a
PA3
        BNE     DRTE
        BRA     CHENILLE    * on a termine un cycle AR

TEMPO
        LDY     #20000      * temporisation
BCLE    DEY
        CPY     #0
        BNE     BCLE
        RTS

        END
```

Figure 9 : Exemple d'un programme assembleur 68HC11

5.2. Programmation en langage C

❖ Généralités

Le compilateur C croisé de Cosmic Software est ici utilisé. Il est à noter que l'on peut aussi assembler des programmes écrits en assembleur mais pour des raisons de commodité, la solution précédente est préférée.

La production d'un objet à partir d'un programme C 68HC11 est la suivante :

- Edition du fichier source : extension du fichier **.c** obligatoirement.
- Compilation du fichier source à l'aide de la commande **CX6811.EXE**
  ex : c:\> CX6811 -v –e libhc11.c
- Le fichier objet a été créé (pour l'exemple : libhc11.o). Un fichier d'erreur est éventuellement créé (pour l'exemple : libhc11.err).
- Edition de liens avec les autres fichiers objets éventuels. On créera pour cela un fichier de commandes (commande **CLNK.EXE**, extension du fichier de commandes : **.LKF)**
- Production du fichier SRecord final (commande **CHEX.EXE**).
- Téléchargement dans la carte mère.

Un exemple complet est donné ci-après. On se réfèrera à l'annexe 8 pour de plus amples explications.

❖ Bibliothèque LIBHC11 de fonctions C de contrôle de la carte d'entrées/sorties

Une bibliothèque de fonctions C de contrôle de la carte d'entrées/sorties ainsi que de gestion de la liaison série a été écrite.

Cela permet d'écrire des programmes plus clairs et mieux structurés. Les spécificités matérielles sont aussi gommées ; c'est à dire que cette bibliothèque joue un peu le rôle de « driver ».

Voici la description des fonctions et leur prototype :

```
/*************************************************************/
/*************************************************************/
/* Fonction: tempo()                                         */
/*             entree(s) : duree de la tempo en ms           */
/*             sortie(e) : rien                              */
/* Description :                                             */
/* Temporisation en ms. Utilise la sortie Output Compare  1  */
/*************************************************************/
/*************************************************************/
void tempo (unsigned int);


/*************************************************************/
/*************************************************** ****/
/* Fonction : get1A()                                        */
/*          entree(s) : numero du bit a lire du port A       */
/*          sortie(e) : valeur du bit (autres bits mis a 0)  */
/* Description :                                             */
/* Lecture de l'etat d'un bit du port A (entree)             */
/*************************************************************/
/*************************************************************/
unsigned char get1A(unsigned char);


/*************************************************************/
/*************************************************************/
/* Fonction : get1E()                                        */
/*          entree(s) : numero du bit a lire du port E       */
/*          sortie(e) : valeur du bit  (autres bits mis a 0)*/
/* Description :                                             */
/* Lecture de l'etat d'un bit du port E (entree)             */
/*************************************************************/
/*************************************************************/
unsigned char get1E(unsigned char);
```

```
/*************************************************************/
/*************************************************************/
/* Fonction : get8E()                                        */
/*              entree(s) : rien                             */
/*              sortie(e) : valeur du port E                 */
/* Description :                                             */
/* Lecture de l'etat du port E (entree 8 bits)               */
/*************************************************************/
/*************************************************************/
unsigned char get8E();

/*************************************************************/
/*************************************************************/
/* Fonction : put8A()                                        */
/*              entree(s) : valeur a ecrire dans le port A   */
/*              sortie(e) : rien                             */
/* Description :                                             */
/* Ecriture dans le port A (sortie 8 bits)                   */
/*************************************************************/
/*************************************************************/
void put8A(unsigned char);


/*************************************************************/
/*************************************************************/
/* Fonction : put1A()                                        */
/*        entree(s) : valeur d'un bit a ecrire dans le port A*/
/*        sortie(e) : rien                                   */
/* Description :                                             */
/* Ecriture d'un bit dans le port A (sortie 8 bits)          */
/*************************************************************/
/*************************************************************/
void put1A(unsigned char, unsigned char);


/*************************************************************/
/*************************************************************/
/* Fonction : baudrate()                                     */
/*              entree(s) : valeur du baud rate de la RS.232 */
/*              sortie(e) : rien                             */
/* Description :                                             */
/* Configuration du port serie : vitesse max =9600bps,       */
/* vitesse min=1200bps                                       */
/*************************************************************/
/*************************************************************/
void baudrate(unsigned int);
```

```
/*************************************************************/
/*************************************************************/
/* Fonction : read_com()                                     */
/*        entree(s) : valeur du caractere recu sur la RS.232 */
/*        sortie(e) : caractere recu = 1, 0 si rien          */
/* Description :                                             */
/* Lecture d'un octet sur le port serie                      */
/*************************************************************/
/*************************************************************/
unsigned char read_com (unsigned char *);
```

```
/*************************************************************/
/*************************************************************/
/* Fonction : write_com()                                    */
/*   entree(s) : valeur du caractere a emettre sur la RS.232 */
/*   sortie(e) : rien                                        */
/* Description :                                             */
/* Ecriture d'un octet sur le port serie                     */
/*************************************************************/
/*************************************************************/
void write_com (unsigned char);
```

```
/*************************************************************/
/*************************************************************/
/* Fonction : AppTickInit ()                                 */
/*   entree(s) : rien                                        */
/*   sortie(e) : rien                                        */
/* Description :                                             */
/* Initialisation de l'interruption tick timer pour le noyau */
/* uC/OS II                                                  */
/*************************************************************/
/*************************************************************/
extern void AppTickInit (void);
```

L'utilisation de la bibliothèque LIBHC11 nécessite l'inclusion du fichier « header » libhc11.h et l'édition de liens avec le fichier objet de la bibliothèque LIBHC11 libhc11.o.

Le squelette du programme C est ainsi le suivant :

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

void main (void) {
int i = 0;


baudrate(9600);
…
}
```

❖ Exemple de programme C et d'utilisation de l'environnement Cosmic C

Le corps du programme source C (fichier ping.c) est donné ci-après. Il faut noter l'utilisation ici de la bibliothèque LIBHC11.

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

void main (void) {
int i = 0;

baudrate(9600);
…
}
```

Figure 10 : Squelette du fichier ping.c

Les étapes de compilation, éditions de liens et génération du fichier SRecord sont réalisées par le lancement d'un fichier DOS batch .BAT ping.bat.

```
@echo off
cx6811 -v -e ping.c
if errorlevel 1 goto bad
:clink
echo.
echo Linking ...
clnk -o ping.h11 -m ping.map ping.lkf
if errorlevel 1 goto bad
:chexa
echo.
echo Converting ...
chex -o ping.s19 ping.h11
if errorlevel 1 goto bad
:cobj
echo.
```

```
echo Section size ...
cobj -n ping.h11
if errorlevel 1 goto bad
echo.
echo.
echo        OK.
goto sortie
:bad
echo.
echo.
echo        BAD.
:sortie
```

Figure 11 : Fichier ping.bat

Au vu de ce qui est écrit dessus le fichier de commandes pour l'édition de liens s'appelle ping.lkf.

```
#       link command file for test program
#       Copyright (c) 1995 by COSMIC Software
#
+seg .text -b 0x2000 -n .text   # program start address
+seg .const -a .text            # constants follow program
+seg .data -b 0x3000            # data start address bss
ping.o                          # application program
libhc11.o                       # 68hc11 enseirb lib
d:/logiciel/68hc11/cx32/lib/libi.h11          # C library
d:/logiciel/68hc11/cx32/lib/libm.h11               #  machine
library
```

Figure 12 : Fichier ping.lkf

6. UTILISATION DU MONITEUR BUFFALO

6.1. Qu'est ce qu'un moniteur ?

Nous avons vu que le logiciel PCBUG11 télécharge un petit programme de 192 octets environ en mémoire RAM. Un moniteur est un programme beaucoup plus gros (8 Ko environ) que l'on programme dans une EPROM externe et qui permet donc de remplacer le TALKER de PCBUG11. Les avantages sont multiples puisque le moniteur occupe très peu de RAM (uniquement pour quelques variables)
Il permet de «débugger » le programme utilisateur grâce à des commandes intégrées que nous verrons plus loin et le moniteur dialogue avec l'utilisateur par l'intermédiaire d'un simple terminal (ou l'émulateur HYPERTERMINAL Windows).

L'exécution du moniteur se fait lors du RESET en mode ETENDU et rend libre l'utilisation du port série après le lancement du programme utilisateur.

6.2. Cartographie mémoire avec le moniteur BUFFALO

Voici le mapping mémoire de la carte mère avec le moniteur BUFFALO version 3.4 intégré :

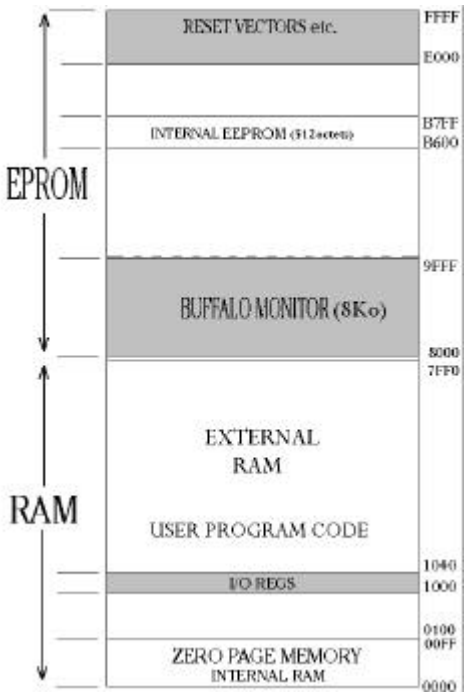


Figure 13 : Cartographie mémoire en mode étendu avec le moniteur BUFFALO 3.4

Elle est identique à celle que nous avons déjà vue, une zone de 8 Ko étant réservée au code du moniteur.

En revanche l'EPROM utilisée est de 32 Ko permettant ainsi de mettre d'autres programmes en EPROM. L'initialisation de la table des vecteurs doit se faire lors de la programmation de l'EPROM.

6.3. Commandes du moniteur BUFFALO

Les commandes du moniteur BUFFALO sont similaires à celles utilisées par PCBUG11 et sont accessibles dès la mise en route du moniteur en tapant *entrée* ou *help* : la capture d'écran suivante montre l'ensemble des commandes dont le rôle est assez intuitif.

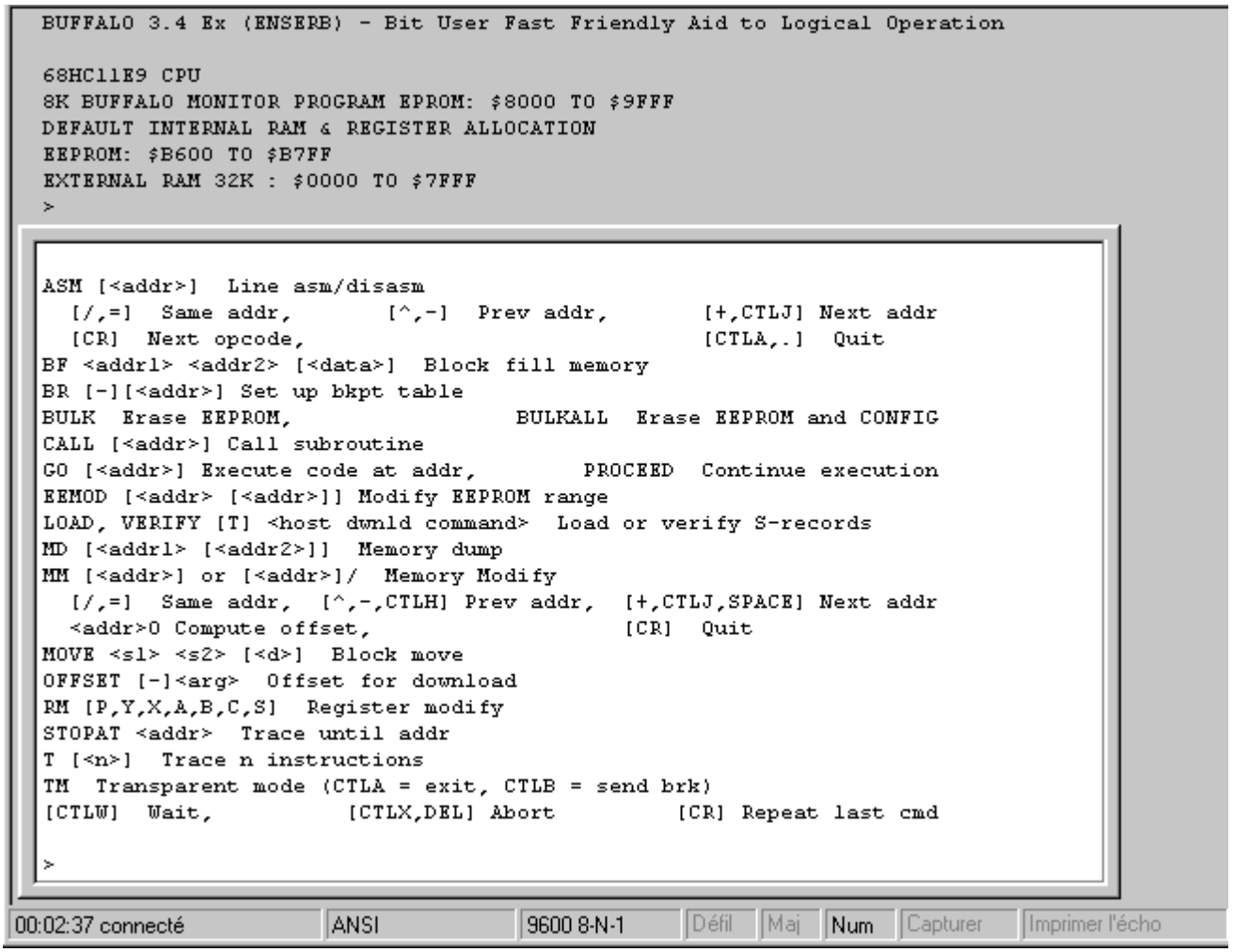


Figure 14 : Moniteur BUFFALO en cours d'utilisation

Le programme source du moniteur BUFFALO est disponible chez MOTOROLA pour différentes versions du 68HC11. (voir bibliographie).

La configuration du moniteur consiste à spécifier le début de la zone où le moniteur sera implanté (EPROM en général), le début de la RAM interne, la EEPROM et les registres. L'assemblage se fait avec ASMHC11.EXE

La description complète des commandes du moniteur BUFFALO est donnée dans l'annexe 7.

7. UTILITE D'UN EMULATEUR

L'émulateur CT68HC11 est un appareil conçu pour émuler la série de microcontrôleurs 68HC11. Il pilote en temps réel l'ensemble des signaux du microcontrôleur et permet donc d'exécuter et de débugger un programme téléchargé dans l'émulateur ou sur la carte cible. C'est un allié lors de la mise au point car il évite par exemple à programmer à chaque fois une EPROM avec la nouvelle version de programme à tester.
Il peut être aussi passif et déassembler en ligne le code exécuté par le microcontrôleur.
En général, c'est l'outil qui va permettre de savoir si l'on a sur une carte un problème hard ou soft.
Il existe autant d'émulateur qu'il y a de processeurs existants. C'est un appareil cher (> 150 KF) mais dont la rentabilité est faite en cas d'un problème pointu (par exemple dans un système embarqué pour les télécommunications, plantage au bout de quelques jours en endurance sans raison apparente (expérience personnelle : plus d'un mois pour « fixer le bug » ! ! !).

Pour l'émulateur 68HC11, toutes les fonctions d'émulation se font en temps réel et l'on peut configurer l'émulateur dans tous les modes : « bootstrap », «special test », « single chip » et « extended ». L'utilisateur peut également choisir d'utiliser l'horloge de la carte cible (Target) ou bien l'horloge interne de l'émulateur. Voici ci-dessous l'ensemble des configurations que l'on peut effectuer sur le type de processeur :

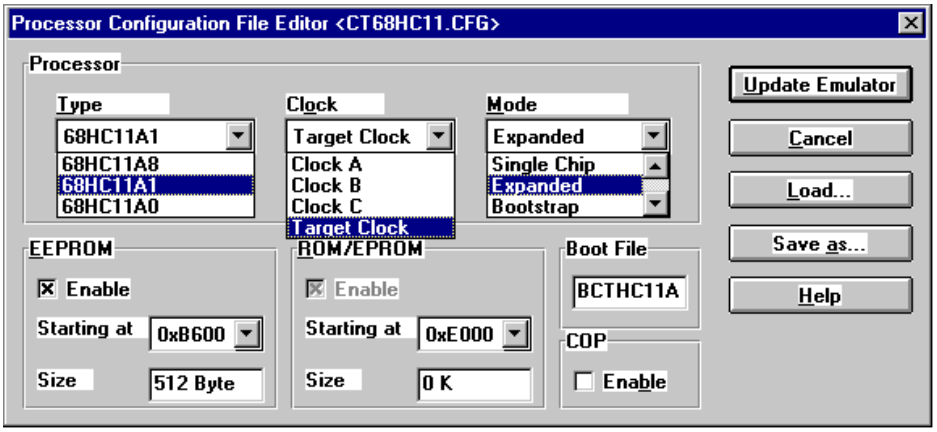


Figure 15 : Ecran de configuration de l'émulateur CT68HC11

L'émulateur 68HC11 possède 64 Ko de mémoire programme et supporte une fréquence d'horloge jusqu'à 24 MHz (soit 6 MHz interne).

L'émulateur permet grâce à un logiciel spécifique de visualiser simultanément dans des fenêtres séparées l'état du microprocesseur, la mémoire, les registres internes, le code désassemblé, la pile (pointeur + contenu) et la ligne de commande. L'émulateur permet également de visualiser le code source, les variables du programme ainsi que l'exécution pas à pas en mode Trace.

Une manipulation intéressante consiste à émuler un 68HC11Ax en mode «bootstrap » (mémoire émulateur) avec le contrôle de PCBUG11 sur le port série. On visualise ainsi sur la capture d'écran ci-dessous le programme du TALKER qui a été téléchargé en RAM par PCBUG11.

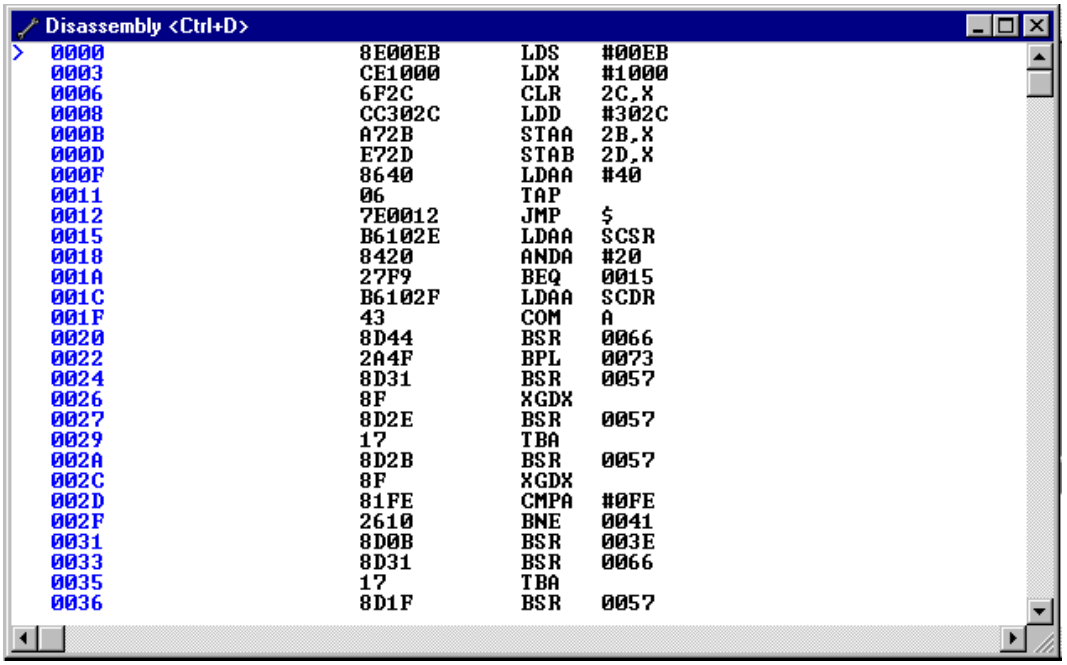


Figure 16 : TALKER de PCBUG11 visualisé depuis l'émulateur

L'utilisation de l'émulateur 68HC11 ne sera pas abordée durant le TP.

8. DERNIERES MINUTES

Il faut noter que le microcontrôleur 68HC11 version A1 n'est plus fabriqué par Motorola. Il a été remplacé par le 68HC11 version E1 compatible broche à broche.

Des différences existent néanmoins au niveau des fonctionnalités internes résumées ci-après :

- 512 octets de RAM au lieu de 256.
- Une ROM interne contenant le moniteur BUFFALO est intégrée et visible de $D000 à $FFFF quand son utilisation a été validée. Cela permettrait sur la carte mère de remplacer le moniteur en ROM externe par le code d'une autre application, ce que l'on ne fera pas ici. **On utilisera donc le moniteur BUFFALO en ROM externe.**
- Protection des écritures en EEPROM par validation de bits du registre BPROT.
- La broche 3 du port A est maintenant bidirectionnelle.
- Dans le processus de "bootstrap", le JMP $0 est fait après 512 octets chargés en RAM interne ou après avoir reçu 4 caractères IDLE (ligne à l'état 1 durant la durée de réception de 4 caractères).

TRES IMPORTANT :

Il est possible que le moniteur BUFFALO en ROM interne soit validé par une fausse manipulation sur les registres du 68HC11. On peut s'en apercevoir en autre par la bannière :

```
BUFFALO 3.2 (int) - Bit User Fast Friendly Aid to Logical Operation
```

Au lieu de :

```
BUFFALO 3.4 - (ENSEIRB pk/yb) - Bit User Fast Friendly Aid to Logical
Operation
68HC11 CPU A & E series
8K BUFFALO MONITOR PROGRAM EPROM: $8000 TO $9FFF
EXTERNAL RAM 32K : $0000 TO $7FFF
EEPROM: $B600 TO $B7FF
DEFAULT INTERNAL RAM & REGISTER ALLOCATION
```

Si tel est le cas, prévenir aussitôt l'enseignant qui dévalidera le moniteur BUFFALO en ROM interne pour pouvoir faire les TP sinon certaines manipulations seront impossibles.

L'outil PCBUG11 ne marche pas sur des PC de fréquence supérieure à 300 MHz et sous Windows NT. En conséquence, on utilisera l'outil JBug11 qui possède une interface Windows, faisant la même chose que PCBUG11 et compatible au niveau des commandes en ligne.

PARTIE I I

- MANIPULATIONS -

9. EX 0 : QUESTIONS DE SYNTHESE

A l'issue de la lecture de ce TP, vous devriez pouvoir répondre à ces quelques questions…

1. Préciser les différences existant entre un microprocesseur et un microcontrôleur.
2. Préciser les différents modes de configuration du microcontrôleur 68HC11.
3. Le 68HC11 étant en mode bootstrap, comment peut-on autoriser les accès externes ?
4. Qu'est ce que le format SRecord ?
5. Quels sont les différentes méthodes de téléchargement d'un objet SRecord en RAM interne ou externe ou en EEPROM ?
6. Comment peut-on définir la notion de temps réel ?
7. Qu'est ce qu'un noyau temps réel ? Quel est son utilité ?

10. EX 1 : ANALYSE D'UN SOURCE ASM 68HC11. COMPARAISON AVEC L'ASSEMBLEUR 68000

Récapitulatif :

BUT	**TSTIO - Comparaison ASM 68HC11/ASM 68000. EEPROM**
LANGAGE	**Assembleur 68HC11**
MODE 68HC11	**Etendu**
RAM EXTERNE	**LOW ($0000-$7FFF)**
MONITEUR	**BUFFALO**
REPERTOIRE	**..\TP_68HC11\EX1**
AUTRE	**HYPERTERMINAL**

1. Editer le fichier *tstio.asc*. Comparer les mnémoniques des instructions 68HC11 avec celles des instructions 68000. Vous semble-t-il facile de maîtriser l'ASM 68HC11 connaissant l'ASM 68000 ? Où sont les difficultés ?

2. Quelle est l'adresse de base de téléchargement du code objet en mémoire ? Quel est ainsi le type de la mémoire adressée ? Le programme est-il perdu après une remise sous tension ? Que fait ce programme ? Remapper le programme source en mémoire RAM à l'adresse de base $2000.

3. Assembler le programme source avec l'exécutable ASMHC11.EXE (exécuter le batch *go.bat*). Télécharger le SRecord généré avec le moniteur BUFFALO. Lancer le programme. Remarquer que la sortie PA3 est utilisée par BUFFALO**. Pour télécharger le fichier Srecord, on utilisera la commande BUFFALO "LOAD T" (réception du fichier sur la liaison série) et pour le programme HYPERTERMINAL, le menu Transfert>Envoyer un fichier texte (émission du fichier sur la liaison série).**

Remarque :

Il est possible de lancer automatiquement le programme en EEPROM au reset de la carte :

- ✓ Au niveau hard : en reliant les signaux en sortie du microcontrôleur Rx et Tx de la liaison série ensemble par une résistance de pull up
- ✓ Au niveau soft : au reset en mode bootstrap, envoyer le caractère BREAK au microcontrôleur. Normalement, le microcontrôleur s'attend à recevoir le caractère $FF pour télécharger dans sa RAM interne à partir de l'adresse $0000, 256 octets avant de faire un « JMP $0 ». S'il reçoit le caractère BREAK, il fait dans ce cas un « JMP $B600 », c'est à dire un saut au début du programme en EEPROM.

4. A quoi correspond le caractère BREAK au niveau de la liaison série RS.232 ?

11. EX 2 : ECHO SUR LA RS.232. UTILISATION DE LA BIBLIOTHEQUE LIBHC11

Récapitulatif :

BUT	**PING1 - Echo sur la liaison RS.232**
LANGAGE	**C**
MODE 68HC11	**Etendu**
RAM EXTERNE	**LOW ($0000-$7FFF)**
MONITEUR	**BUFFALO**
REPERTOIRE	**..\TP_68HC11\EX2**
AUTRE	**HYPERTERMINAL**

Le but de ce TP est d'écrire un programme C qui réalise l'écho de tout ce qui est reçu sur la RS.232. On utilisera la bibliothèque de fonctions LIBHC11 pour cela.

1. Créer le fichier *ping1.c* dont la trame vous est donnée. En utilisant la bibliothèque LIBHC11, réaliser le programme d'écho sur la RS.232.

2. Regarder le fichier *ping1.bat*, fichier batch pour créer le fichier SRecord *ping1.s19*. Expliquer les différentes étapes exécutées.

3. Analyser le fichier de commandes *ping1.lkf* pour l'édition de liens. Quelles sont les adresses de base pour le code, les données initialisées et les données non initialisées ? Exécuter le fichier *ping1.bat*. Après correction des erreurs éventuelles, vous devez avoir les fichiers *ping1.map, ping1.s19* (*ping1.o, ping1.h11*). Les erreurs éventuelles sont consignées dans le fichier *ping1.err*.

4. Analyser le fichier de mapping *ping1.map* donnant entre autre la table de symboles. Quel est le point d'entrée du programme ?

5. Regarder le fichier SRecord *ping1.s19* et retrouver les différents champs du fichier.

6. A l'aide de BUFFALO, télécharger le fichier SRecord *ping1.s19*. Vérifier le bon fonctionnement du programme **PING1**.

12. EX 3 : ECHO SUR LA RS.232. UTILISATION DE LA BIBLIOTHEQUE D'E/S COSMIC

Récapitulatif :

BUT	**PING2 - Echo sur la liaison RS.232.**
LANGAGE	**C**
MODE 68HC11	**Etendu**
RAM EXTERNE	**LOW ($0000-$7FFF)**
MONITEUR	**BUFFALO**
REPERTOIRE	**..\TP_68HC11\EX3**
AUTRE	**HYPERTERMINAL**

Le but de ce TP est de réécrire le programme C précédent en utilisant la bibliothèque de fonctions d'E/S de Cosmic pour cela.

1. Créer le fichier *ping2.c* à partir du fichier précédent *ping1.c*. En utilisant la bibliothèque d'E/S Cosmic, réaliser le programme d'écho sur la RS.232.

2. Créer le fichier *ping2.bat à partir de ping1.bat.*

3. Créer le fichier de commandes *ping2.lkf* pour l'édition de liens à partir de *ping1.lkf*. Quelles sont les adresses de base pour le code, les données initialisées et les données non initialisées. Exécuter le fichier *ping2.bat*. Après correction des erreurs éventuelles, vous devez avoir les fichiers *ping2.map, ping2.s19* (*ping2.o, ping2.h11*). Les erreurs éventuelles sont consignées dans le fichier *ping2.err*.

4. Analyser le fichier de mapping *ping2.map* donnant entre autre la table des symboles. Quel est le point d'entrée du programme ?

5. A l'aide de BUFFALO, télécharger le fichier SRecord *ping2.s19*. Vérifier le bon fonctionnement du programme **PING2**.

6. Le programme *ping2.c* est-il portable dans un environnement standard UNIX ?

13. EX 4 : ECHO SUR LA RS.232. EXECUTION DIRECTE DU PROGRAMME AU RESET EN MODE ETENDU

Récapitulatif :

BUT	**PING3 - Echo sur la liaison RS.232. Exécution directe au reset**
LANGAGE	**C**
MODE 68HC11	**Bootstrap avec validation des accès externes puis étendu**
RAM EXTERNE	**HIGH ($8000-$FFFF)**
MONITEUR	**JBUG11**
REPERTOIRE	**..\TP_68HC11\EX4**
AUTRE	**HYPERTERMINAL**

Le but de ce TP est de reprendre le programme C ping1.c du TP1 en le faisant exécuter directement au reset. La RAM externe est en position RAM HIGH (interrupteur MAP) pour pouvoir configurer la table des vecteurs. Il faut noter ici qu'en version finale, le programme est mis en EPROM, ce qui reviendrait à remplacer l'EPROM du moniteur BUFFALO par celle contenant le programme. L'utilisation de la RAM ici facilite la mise au point…

1. Recopier sous D:\TP_68HC11\EX4 le fichier *ping1.c* sous le nom *ping3.c*

2. Analyser le fichier *crts.s*. Quel est son rôle ? Quel est l'étiquette de point d'entrée du programme **PING3** ?

3. Analyser le fichier batch *ping3.bat*.

4. Regarder le fichier d'initialisation de la table des vecteurs *vector.c*. Retrouver le point d'entrée du programme **PING3**.

5. Analyser le fichier de commandes d'édition de liens *ping3.lkf*. Quelles sont les adresses de base pour le code, les données initialisées et les données non initialisées ? Quelle est l'adresse de base du tableau const _vectab[]. A quelle adresse se situe alors le point d'entrée au reset ? Est-ce normal ?

6. L'ordre de chargement des objets .o est-il important dans le fichier précédent ?

7. Exécuter le fichier *ping3.bat*.

8. Analyser le fichier de mapping *ping3.map*. Quelle est l'adresse du point d'entrée du programme ?

9. Configurer la carte en mode bootstrap. Utiliser JBUG11. Valider les accès externes afin de pouvoir accéder à la RAM externe de $8000 à $FFFF. Télécharger le fichier SRecord *ping3.s19*.
10. Vérifier par dump la table des vecteurs et notamment les adresses $FFFE-$FFFF qui contiennent le point d'entrée du programme **PING3**.

11. Reconfigurer la carte en mode étendu et faire un reset. Le programme **PING3** doit alors s'exécuter. On utilisera pour le vérifier le programme Windows HYPERTERMINAL.

PARTIE III

- ANNEXES -

- ANNEXE 1 -
Commandes de PCBUG11

Use the host-computer keyboard to edit the PCbug11 command line. A recall buffer holds the last 16 commands entered. The following table lists the edit keys.

```
left arrow     - Moves cursor back one character
right arrow    - Moves cursor forward one character
Home           - Moves cursor to first character
End            - Moves cursor to last character
delete left    - Deletes to the left of the cursor
Del            - Deletes at the cursor position
<CTRL> End     - Deletes from cursor position to the end of line
Ins            - Inserts at cursor position (cursor changes to blocked
                 cursor)
up arrow       - Recalls previous commands, in reverse order
down arrow     - Recalls last command in recall buffer
Esc            - Clears command line or terminates most commands in
progress
```

The four lines above the command line serve as a trace of the last four commands. The fifth line above the command line shows breakpoints and error codes.

Possible error codes are:

```
     0 :        No error
     1 :        VERF error
     2 :        MS or BF error
     3 :        Talker communication failure.
```

For MS-DOS batch files, an error code can be checked via ERRORLEVEL after PCbug11 terminates.

Monitor Commands

The following is a summary of PCbug11 commands.

COMMAND	DESCRIPTION
ASM addr [mne|dir]	Call symbolic macro line assembler, with option to auto insert mnemonic or directive
BAUD [rate]	Display or set serial baud rate
BF addr1 [addr2] byte|word	Block fill memory with byte or word
BL	Display breakpoints
BR [addr [macroname]]	Display/Set break point [with optional macro execution]
CALL addr	Execute the subroutine at addr
CLRM	Clear all command macros
CLS	Clear main window
CONTROL	Display or set CONTROL parameters
CONTROL BASE BIN|HEX|DEC	Change default number base

```
|CONTROL BIOS                  | Access serial COM port through BIOS calls
|CONTROL COLOR                 | Set PCbug11 display colors
|CONTROL COLOUR                | Set PCbug11 display colours
|CONTROL COM1                  | Use COM1 port
|CONTROL COM2                  | Use COM2 port
|CONTROL EPROG address         | Change the EPROM register address
|CONTROL ERRMSG option         | Enable or disable display of error
messages
|CONTROL HARDWARE              | Access COM port directly through hardware
|CONTROL LAST                  | Toggle the last error msg window on/off
|CONTROL LOG CLOSE             | Close the open command log file
|CONTROL LOG OFF               | Suspend logging to command log file
|CONTROL LOG ON                | Enable logging to the command log file
|CONTROL LOG OPEN              | Open the command log file
|CONTROL PPROG address         | Change the EEPROM register address
|CONTROL PROTECT               | Use the RTS to provide memory protection
|CONTROL RTS                   | Control the RTS line directly
|CONTROL RTS ON|OFF            | Set RTS line high or low
|CONTROL TIMEOUT [value]       | Display or set the value of serial
|                              | COM timeout during input
-------------------------------------------------------------------------
|DASM addr1 [addr2]            | Disassemble from addr1 [to addr2]
|DB startaddr [endaddr]        | Display MCU Memory
|DEBUG                         | Reserved word
|DEFINE symbol value|address   | Define a symbol
|DEFM macrnam|TRACE|AUTOSTART  | Define a command, trace or autostart macro
|DELM macrnam|TRACE|AUTOSTART  | Delete a command, trace or autostart macro
|DIR [mask]                    | Display disk directory
|DOS [command]                 | Shell to DOS or execute DOS command
-------------------------------------------------------------------------
|EDITM macrnam                 | Edit a macro
|EEPROM [startaddr [endaddr]]  | Display, clear, set EEPROM addr range(s)
|EEPROM DELAY option           | Set EEPROM erase or write programming time
|EEPROM ERASE [option]         | Display or chg EEPROM erase-before-write
|ENV                           | Define Environment save options
|EPROM [startaddr [endaddr]]   | Display, clear, set EPROM address range(s)
|EPROM DELAY option            | Set EPROM erase or write programming time
-------------------------------------------------------------------------
|FIND byte|word addr1 addr2    | Find all occurrences of byte or word
|                              | between addr1 & addr2
|FIND mnemonic addr1 addr2     | Find all occurrences of mnemonic
|                              | between addr1 & addr2
-------------------------------------------------------------------------
|G [addr]                      | Start user code execution
-------------------------------------------------------------------------
|HELP [command]                | Display help information
|HELP MCU [topic]              | Display help on specific MCU topic
-------------------------------------------------------------------------
|KLE                           | Kill last error message
-------------------------------------------------------------------------
|LOADM [filename] [macroname]  | Load macro des from default or user
|                              | file
|LOADS filename [loadaddr]     | Load S-record file into MCU memory
|LS symbol                     | Display symbols
|LSTM [mname|TRACE|AUTOSTART]  | Display macro names or definitions
-------------------------------------------------------------------------
```

MD startaddr [endaddr]	Display MCU memory
MM addr	Modify memory from addr
MOVE addr1 addr2 addr3	Move MCU memory between addr1 & addr2 to addr3
MS addr byte|word [byte|word]	Set MCU memory byte(s) or word(s)
MSG [string]	Display message in main window
NOBR [address]	Remove all or specified breakpoints
PAUSE [ms]	Wait for any key press or delay time
PRINT	Display PCbug11 version number
PROTECT [startaddr [endaddr]]	Display, clear or set write-protected address range(s)
QUIT [Y]	Terminate PCbug11 session [without confirming]
RD [T]	Display ot trace MCU registers
RESET [addr]	MCU Hardware Reset with Existing or new reset vector.
RESTART [option]	Restart PCbug11 with same or new option.
RM	Modify MCU registers in window
RS register value	Set value of MCU register
S	Stop user code execution
SHELL ["command" ;P]	Shell to DOS or execute DOS command
SAVEM [filename]	Save macro definitions in default or user file
T [addr]	Trace user code
TERM [X1 Y1 X2 Y2]	Simple windowed terminal emulator
TYPE filename	Display disk file in main window
UNDEF symbol	Undefine a symbol
VER	Display version number
VERF ERASE addr1 [addr2]	Verify that memory contains $FF
VERF SET addr1 addr2 value	Verify that memory contains the value
VERF filename [memaddr]	Verify S-record disk file against memory
WAIT [ms]	Wait for ms

Special Key operations:

<CTRL> B	Send break on COM channel
<CTRL> P	Toggle MCU memory write protect/RTS line
<CTRL> R	Attempt to re-synchronise talker

EVS BUFFALO Compatible commands

These commands operate in a similar manner to the command of the same name on the Motorola 68HC11 EVS systems.

ASM, BF, BR, G, HELP, MD, MM, NOBR, RD, RM

- ANNEXE 2 -

Présentation de Cosmic C

C Cross Compiler User's Guide for MC68HC11 (Résumé)

TUTORIAL INTRODUCTION

This chapter will demonstrate, step by step, how to compile, assemble and link the example program acia.c, which is included on your distribution media. Although this tutorial cannot show all the topics relevant to the COSMIC tools, it will demonstrate the basics of using the compiler for the most common applications.

In this tutorial you will find information on the following topics:

Default Compiler Operation

Compiling and Linking

Generating Hex File

Linking Your Application

Generating Automatic Data Initialization

Specifying Command Line Options

Acia.c, Example file
The following is a listing of acia.c. This C source file is copied during the installation of the compiler:

```
/* EXAMPLE PROGRAM WITH INTERRUPT HANDLING */
#include <io.h>

#define SIZE    128                     /* buffer size */
#define TDRE    0x80                    /* transmit ready bit */

/*      Authorize interrupts. */
#define cli() _asm("cli\n")

/*      Some variables */
char buffer[SIZE];                      /* reception buffer */
char * ptlec;                           /* read pointer */
char * ptecr;                           /* write pointer */

/*      Character reception.
 *      Loops until a character is received.
 */
char getch(void)
        {
        char c;                         /* character to be returned */

        while (ptlec == ptecr)          /* equal pointers => loop */
                ;
        c = *ptlec++;                   /* get the received char */
        if (ptlec >= &buffer[SIZE])     /* put in in buffer */
                ptlec = buffer;
        return (c);
        }

/*      Send a char to the SCI. */
void outch(char c)
        {
```

```
        while (!(SCSR & TDRE))          /* wait for READY */
                ;
        SCDR = c;                       /* send it */
        }

/*      Character reception routine. This routine is called on interrupt.
 *      It puts the received char in the buffer.
 */
@interrupt void recept(void)
        {
        SCSR;                           /* clear interrupt */
        *ptecr++ = SCDR;                        /* get the char */
        if (ptecr >= &buffer[SIZE])     /* put it in buffer */
                ptecr = buffer;
        }

/*      Main program.
 *      Sets up the SCI and starts an infinite loop of receive transmit.
 */
void main(void)
        {
        ptecr = ptlec = buffer;         /* initialize pointers */
        BAUD = 0x30;                    /* initialize SCI */
        SCCR2 = 0x2c;                   /* parameters for interrupt */
        cli();                          /* authorize interrupts */
        for (;;)                                /* loop */
                outch(getch());         /* get and put a char */
        }
```

Default Compiler Operation

By default, the compiler compiles and assembles your program. You may then link object files using clnk to create an executable program.

As it processes the command line, **cx6811** echoes the name of each input file to the standard output file (your terminal screen by default). You can change the amount of information the compiler sends to your terminal screen using command line options, as described later.

According to the options you will use, the following files, recognized by the COSMIC naming conventions, will be generated:

file.s	Assembler source module
file.o	Relocatable object module
file.h11	input (*e.g.* libraries) or output (*e.g.* absolute executable) file for the linker

Compiling and Linking

To compile and assemble acia.c using default options, type:

```
cx6811 acia.c
```

The compiler writes the name of the input file it processes:

```
acia.c:
```

The result of the compilation process is an object module named acia.o produced by the assembler. We will, now, show you how to use the different components.

Step 1: Compiling

The first step consists in compiling the C source file and producing an assembly language file named acia.s.

```
cx6811 -s acia.c
```

The -s option directs cx6811 to stop after having produced the assembly file acia.s. You can then edit this file with your favorite editor. You can also visualize it with the appropriate system command (type, cat, more,...). For example under MS/DOS you would type:

```
type acia.s
```

If you wish to get an interspersed C and assembly language file, you should type:

```
cx6811 -l acia.c
```

The **-l** option directs the compiler to produce an assembly language file with C source line interspersed in it. Please note that the C source lines are commented in the assembly language file: they start with ';'.

As you use the C compiler, you may find it useful to see the various actions taken by the compiler and to verify the options you selected. The **-v** option, known as verbose mode, instructs the C compiler to display all of its actions. For example if you type:

```
cx6811 -v -s acia.c
```

the display will look like something similar to the following:

```
acia.c:
        cp6811 -o ctempc.cx1 -i\cx\h6811 -u acia.c
        cg6811 -o ctempc.cx2 ctempc.cx1
        co6811 -o acia.s ctempc.cx2
```

The compiler runs each pass:

cp6811 - the C parser
cg6811 - the assembly code generator
co6811 - the optimizer

Step 2: Assembler
The second step of the compilation is to assemble the code previously produced. The relocatable object file produced is acia.o.

```
cx6811 acia.s
```

or

```
ca6811 -i\cx\h6811 acia.s
```

if you want to use directly the macro cross assembler.

The cross assembler can provide, when necessary, listings, symbol table, cross reference and more. The following command will generate a listing file named acia.ls that will also contain a cross reference:

```
ca6811 -c -l acia.s
```

Step 3: Linking
This step consists in linking relocatable files, also referred to as object modules, produced by the compiler or by the assembler (<files>.o) into an absolute executable file: **acia.h11** in our example. Code

and data sections will be located at absolute memory addresses. The linker is used with a command file (acia.lnk in this example).

An application that uses one or more object module(s) may require several sections (code, data, interrupt vectors, etc.,...) located at different addresses. Each object module contains several sections. The compiler creates the following sections:

.text - code (or program) section (*e.g.* ROM).
.const - constant section (*e.g.* ROM)
.data - all static initialized data (*e.g.* RAM).
.bss - all non initialized extern data (*e.g.* RAM)
.bsct - all data in the first 256 bytes (see @dir in chapter 3), also called **zero page**.

In our example, and in the test file provided with the compiler, the *acia.lnk* file contains the following information:

```
line 1 # LINK COMMAND FILE FOR TEST PROGRAM
line 2 # Copyright (c) 1995 by COSMIC Software
line 3 #
line 4 +seg .text -b 0xe000 -n .text     # program start address
line 5 +seg .const -a .text              # constant after program
line 6 +seg .data -b 0x2000              # data start address
line 7 crts.o                            # startup routine
line 8 acia.o                            # application program
line 9 \cx\lib\libi.h11                  # C library (if needed)
line 10 \cx\lib\libm.h11                 # machine library
line 11 +seg .const -b0xffd6             # vectors start address
line 12 vector.o                             # interrupt vectors file
line 13 +def __memory=@.bss              # symbol used by startup
line 14 +def __stack=0x00ff              # stack pointer initial value
```

You can create your own link command file by modifying the one provided with the compiler.

Here is the explanation of the lines in acia.lnk:

lines 1 to 3: These are comment lines. Each line can include comments. They must be prefixed by the "#" character.

line 4: **`+seg .text -b0xe000 -n .text`** creates a text (code) segment located at e000 (hex address), and sets the segment name to **`.text`**.

line 5: **`+seg .const -a .text`** creates a constant segment located after the previous text segment

line 6: **`+seg .data -b0x2000`** creates a data segment located at 2000 (hex address)

line 7: **`crts.o`** runtime startup code. It will be located at 0xf000 (code segment)

line 8: **`acia.o`**, the file that constitutes your application. It follows the startup routine for code and data

line 9: **`libi.h11`** the integer library to resolve references

line 10: **`libm.h11`** the machine library to resolve references

line 11: **`+seg .const -b0xffd6`** creates a new constant segment located at ffd6 (hex address)

line 12: **`vectors.o`** interrupt vectors file

line 13: **+def __memory=@.bss** defines a symbol *__memory* equal to the value of the current address in the *.bss* segment. This is used to get the address of the end of the *bss*. The symbol *__memory* is used by the startup routine to reset the *bss*.

line 14: **+def __stack=0x00ff** defines a symbol *__stack* equal to the absolute value 0x00ff (hex value). The symbol *__stack* is used by the startup routine to initialize the stack pointer.

By default, and in our example, the *.bss* segment follows the .data segment.

The crts.o file contains the runtime startup that performs the following operations:

* initialize the bss, if any
* initialize the stack pointer
* call main() or any other chosen entry point.

For more information, see Chapter 3, “Programming for MC68HC11 Environment, Modifying the Runtime Startup”.

After you have modified the linker command file, you can link by typing:

```
clnk -o acia.h11 acia.lnk
```

Step 4: Generating S-Records file

Although *acia.h11* is an executable image, it may not be in the correct format to be loaded on your target. Use the **chex** utility to translate the format produced by the linker into standard formats. To translate *acia.h11* to Motorola standard S-record format:

```
chex acia.h11 > acia.hex
```

or

```
chex -o acia.hex acia.h11
```

acia.hex is now an executable image in Motorola S-record format and is ready to be loaded in your target system.

Linking Your Application

You can create as many text, data and bss segments as your application requires. For example, assume we have one bss, two data and two text segments. Our link command file will look like:

```
+seg .bsct -b0x0                    # zpage start address
var_zpage.o                     # file containing zpage variable
+seg .text -b 0xe000 -n .text    # program start address
+seg .const -a .text             # constant follow program
+seg .data -b 0x2000             # data start address
+seg .bss -b 0x2500              # bss start address
crts.o                          # startup routine
acia.o                          # main program
module1.o                       # application program
+seg .text -b 0xf000             # start new text section
module2.o                       # application program
module3.o                       # application program
\cx\lib\libi.h11                    # C library (if needed)
\cx\lib\libm.h11                    # machine library
+seg .const -b0xffd6             # vectors start address
vector.o                            # interrupt vectors
+def __memory=@.bss              # symbol used by startup
```

```
+def __stack=0x00ff             # stack pointer initial value
```

In this example the linker will locate and merge crts.o, acia.o and module1.o in a text segment at 0xe000, a data segment at 0x2000 and a bss segment, if needed at 0x2500. zero page variables will be located at 0x0. The rest of the application, module2.o and module3.o and the libraries will be located and merged in a new text segment at 0xf000 then the interrupt vectors file, vector.o in a constant segment at 0xffd6.

Specifying Command Line Options
You specify command line options to cx6811 to control the compilation process.

To compile and get a relocatable file named *acia.o*, type:

```
cx6811 acia.c
```

The **-v** option instructs the compiler driver to echo the name and options of each program it calls. The **-l** option instructs the compiler driver to create a mixed listing of C code and assembly language code in the file acia.ls.

To perform the operations described above, enter the command:

```
cx6811 -v -l acia.c
```

When the compiler exits, the following files are left in your current directory:

the C source file acia.c
the C and assembly language listing acia.ls
the object module acia.o

It is possible to locate listings and object files in specified directories if they are different from the current one, by using respectivally the **-cl** and **-co** options:

```
cx6811 -cl\mylist -co\myobj -l acia.c
```

This command will compile the acia.c file, create a listing named acia.ls in the \mylist directory and an object file named acia.o in the \myobj directory.

cx6811 allows you to compile more than one file. The input files can be C source files or assembly source files. You can also mix all of these files.

If your application is composed with the following files: two C source files and one assembly source file, you would type:

```
cx6811 -v start.s acia.c getchar.c
```

This command will assemble the *start.s* file, and compile the two C source files.

PROGRAMMING FOR MC68HC11 ENVIRONMENTS

Modifying the Runtime Startup
The runtime startup module performs many important functions to establish a runtime environment for C. The runtime startup file included with the standard distribution provides the following:

* Initialization of the bss section if any,

* ROM into RAM copy if required,

* Initialization of the stack pointer,

* _main or other program entry point call, and

* An exit sequence to return from the C environment. Most users must modify the exit sequence provided to meet the needs of their specific execution environment.

The following is a listing of the standard runtime startup file **crts.h11** included on your distribution media. It does not perform automatic data initialization. A special startup program is provided, **crtsi.h11**, which is used instead of *crts.h11* when you need automatic data initialization. The runtime startup file can be placed anywhere in memory. Usually, the startup will be “linked” with the **RESET** interrupt, and the startup file may be at any convenient location.

Description of Runtime Startup Code

```
1  ;    C STARTUP FOR MC68HC11
2  ;    Copyright (c) 1995 by COSMIC Software
3  ;
4       xdef    _exit, __stext
5       xref    _main, __memory, __stack
6  ;
7       switch  .bss
8  __sbss:
9       switch  .text
10 __stext:
11      clra                    ; reset the bss
12      ldx     #__sbss         ; start of bss
13      bra     loop            ; start loop
14 zbcl:
15      staa    0,x             ; clear byte
16      inx                     ; next byte
17 loop:
18      cpx     #__memory       ; up to the end
19      bne     zbcl            ; and loop
20      lds     #__stack                ; initialize stack pointer
21      jsr     _main           ; execute main
22 _exit:
23      bra     _exit           ; and stay here
24 ;
25      end
```

_main is the entry point into the user C program.

__memory is an external symbol defined by the linker as the end of the bss section. The start of the **bss** section is marked by the local symbol **__sbss**.

__stack is an external symbol defined by the linker as an absolute value.

Lines 11 to 19 reset the *bss* section.

Line 20 sets the stack pointer. You may have to modify it to meet the needs of your application.

Line 21 calls *main()* in the user's C program.

Lines 22 to 23 trap a return from *main()*. If your application must return to a monitor, for example, you must modify this line.

Performing Input/Output in C
You perform input and output in C by using the C library functions *getchar, gets, printf, putchar, puts* and *sprintf*.

The C source code for these and all other C library functions is included with the distribution, so that you can modify them to meet your specific needs. Note that all input/output performed by C library functions is supported by underlying calls to *getchar* and *putchar*. These two functions provide access to all input/output library functions. The library is built in such a way so that you need only modify *getchar* and *putchar*, the rest of the library is independent of the runtime environment.

Function definitions for *getchar* and *putchar* are:

```
char getchar(void);
char putchar(char c);
```

Accessing Internal Registers
The type modifier **@port** may be used on a data object in conjunction with an absolute address to improve access to an internal register. These input-output registers are seen as memory locations, but bit instructions are inoperative on extended addressing mode. The *@port* type modifier instructs the compiler to load the base address indirectly, allowing efficient bit instructions to be used. When the absolute address is specified, the base address loaded is obtained from the upper bits of the full address (0x1000 for an address equal to 0x1021). The **PORTB** register may be declared:

```
@port char PORTB @0x1004;
```

The *@port* modifier may be omitted when the register is accessed as bytes rather than bits.

All registers are declared in the **io.h** file provided with the compiler. This file should be included by a:

```
#include "io.h"
```

in each file using the input-output registers. Three separate files **ioc0.h**, **iof1.h**, and **iok4.h** are provided for the special **68HC11C0**, **68HC11F1** and **68HC11K4** processors. They do not use the same same of registers than the standard family. All the register names are defined by assembly *equates* which are made *public*. This allows any assembler source to use directly the input-output register names by defining them with an *xref* directive. All those definitions are already provided in the **io.s** file which may be included in an assembly source by a:

```
include "io.s"
```

All these header files assume a default location for the input-output registers depending on the actual target. This default value may be changed by defining the C symbol **_BASE** by a **#define** directive before the header file # include:

```
#define _BASE 0x1000
#include <io.h>
```

The default value of **0x1000** for the register starting address as defined by the file *<io.h>* is changed to **0**.

NOTE: The *@port* modifier is an extension to the ANSI standard.

Inserting Inline Assembly Instructions

The _asm() function inserts inline assembly code in your C program. The syntax is:

```
_asm("string constant", arguments...);
```

The "*string constant*" argument is the assembly code you want embedded in your C program. "*arguments*" follow the standard C rules for passing arguments. The string you specify follows standard C rules. For example, carriage returns can be denoted by the '\n' character. For example, to produce the following assembly sequence:

```
xgdx
addd #1000h
xgdx
txs
jsr _main
```

you would write

```
_asm("xgdx\naddd #1000h\nxgdx\ntxs\njsr _main\n");
```

NOTE: The argument string must be shorter than 512 characters. If you wish to insert longer assembly code strings you will have to split your input among consecutive calls to *_asm()*.

To copy a value in the condition register, you write:

```
_asm("tba\ntap\n", varcc);
```

The *varcc* variable is passed in the **d** register, as a first argument. The *_asm* sequence then transfers the low byte from the **b** register to the **a** register then to the condition register.

_asm() does not perform any checks on its argument string. Only the assembler can detect errors in code passed as argument to an *_asm()* call.

_asm() can be used in expressions, if the code produced by *_asm* complies with the rules for function returns. For example:

```
if (_asm("tpa\ntab\n") & 0x010)
```

allows to test the overflow bit. That way, you can use *_asm()* to write equivalents of C functions directly in assembly language.

NOTE: The argument string is added as is to the code during the compilation. The optimizer ***does not*** modify the specified instructions, unless the **-a** option is specified to the code generator.

Writing Interrupt Handlers

A function declared with the type qualifier **@interrupt** is suitable for direct connection to an interrupt (hardware or software). *@interrupt* functions may not return a value. *@interrupt* functions are allowed to have arguments, although hardware generated interrupts are not likely to supply anything meaningful. When you define an *@interrupt* function, the compiler uses the "**rti**" instruction for the return sequence.

You define an *@interrupt* function by using the type qualifier *@interrupt* to qualify the type returned by the function you declare. An example of such a definition is:

```
@interrupt void it_handler(void)
```

```
{
...
}
```

You cannot call an *@interrupt* function directly from a C function. It **must** be connected with the interrupt vectors table.

NOTE: The *@interrupt* modifier is an extension to the ANSI standard.

Placing Addresses in Interrupt Vectors
You may use either an assembly language program or a C program to place the addresses of interrupt handlers in interrupt vectors. The assembly language program would be similar to the following example:

```
                switch .const
                xref handler1, handler2, handler3
vector1:                dc.w handler1
vector2:                dc.w handler2
vector3:                dc.w handler3
                end
```

where *handler1* and so forth are interrupt handlers.

A small C routine that performs the same operation is:

```
extern void handler1(), handler2(), handler3();
void (* const vector[])() =
        {
        handler1,
        handler2,
        handler3,
        };
```

where *handler1* and so forth are interrupt handlers. Then, at link time, include the following options on the link line:

```
+seg .const -b0xffd6 vector.o
```

where *vector.o* is the file which contains the vector table. This file is provided in the compiler package.

Interfacing C to Assembly Language
The C cross compiler translates C programs into assembly language according to the specifications described in this section.

You may write external identifiers in both uppercase and lowercase. The compiler prepends an underscore '_' character to each identifier.

The compiler places function code in the **.text** section. Function code is not to be altered or read as data. External function names are published via **xdef** declarations.

Literal data such as strings, float or long constants, and switch tables, are normally generated into the **.const** section. An option on the code generator allows such constants to be produced in the **.text** section.

The compiler generates initialized data into the **.data** section. External data names are published via **xref** declarations. Data you declare to be of "*const*" type by adding the type qualifier *const* to its base type is normally generated into the *.const* section. Data declared with the **@dir** space modifier will be

generated into the **.bsct** section. Uninitialized data are normally generated into the **.bss** section, unless forced to the **.data** section by the compiler option **+nobss**.

Function calls are performed according to the following:

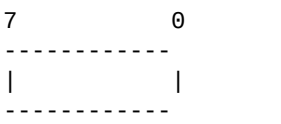
1) Arguments are moved onto the stack from right to left. Unless the function returns a double or a structure, the first argument is stored in the **d** register if its size is less than or equal to the size of an int, or in **d** and **2,x** if its type is long or unwidened float.

2) A data space address is moved onto the stack if a structure or double return area is required.

3) The function is called via a **jsr** _**func** instruction. If the called is an @**far** function, the calling sequence is different. It is detailed in the next paragraph.

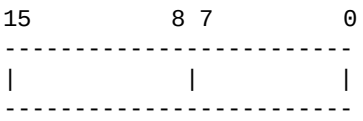
4) The arguments to the function are popped off the stack.

Data Representation
Data objects of type char are stored as one byte:



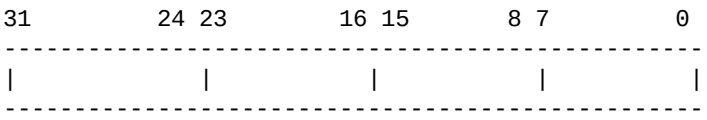
Char representation

Data objects of type *short*, *int* and *16 bits pointers* are stored as two bytes, more significant byte first.



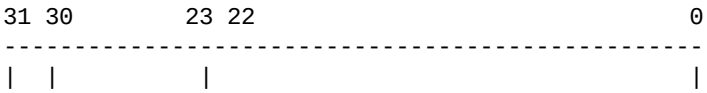
Short, Int, 16 bits pointer

Data objects of type *long* and *32 bits pointers* are stored as four bytes, in descending order of significance.



Long, 32 bits pointers

Data objects of type *float* and *double* are represented as for the proposed IEEE Floating Point Standard; four bytes (for float) or height bytes (for double) stored in descending order of significance. The IEEE representation is: most significant bit is one for negative numbers, and zero otherwise; the next eight bits are the characteristic, biased such that the binary exponent of the number is the characteristic minus 116 (for float) or 1022 (for double); the remaining bits are the fraction, starting with the weighted bit. If the characteristic is zero, the entire number is taken as zero, and should be all zeros to avoid confusing some routines that do not process the entire number. Otherwise there is an assumed 0.5 (assertion of the weighted bit) added to all fractions to put them in the interval [0.5, 1.0). The value of the number is the fraction, multiplied by -1 if the sign bit is set, multiplied by 2 raised to the exponent.



```
-------------------------------------------------
 ^        ^                       ^
 |_ Sign  |_ Characteristic       |_ Mantissa
```

Float representation

```
63 62        52 51                                              0
-------------------------------------------------------------------
|  |          |                                                 |
-------------------------------------------------------------------
 ^        ^                       ^
 |_ Sign  |_ Characteristic       |_ Mantissa
```

Double representation

USING THE COMPILER

This chapter explains how to use the C cross compiler to compile programs on your host system. It explains how to invoke the compiler, and describes its options. It also describes the functions which constitute the C library. This chapter includes the following sections:

Invoking the Compiler

File Naming Conventions

Generating Listings

C Library Support

Invoking the Compiler
To invoke the cross compiler, type the command **cx6811**, followed by the compiler options and the name(s) of the file(s) you want to compile. All the valid compiler options are described in this chapter. Commands to compile source files have the form:

```
cx6811 [options] <files>.[c|s]
```

cx6811 is the name of the compiler. The option list is optional. You must include the name of at least one input file <file>. <file> can be a C source file with the suffix '**.c**', or an assembly language source file with the suffix '**.s**'. You may specify multiple input files with any combination of these suffixes in any order.

If you do not specify any command line options, *cx6811* will compile your <files> with the default options. It will also write the name of each file as it is processed. It writes any error messages to STDERR.

The following command line:

```
cx6811 acia.c
```

compiles and assembles the acia.c C program, creating the relocatable program **acia.o**.

If the compiler finds an error in your program, it halts compilation. When an error occurs, the compiler sends an error message to your terminal screen unless the option **-e** has been specified on the command line. In this case, all error messages are written to a file whose name is obtained by replacing

the suffix .c of the source file by the suffix .err. An error message is still output on the terminal screen to indicate that errors have been found. If one or more command line arguments are invalid, *cx6811* processes the next file name on the command line and begins the compilation process again.

The example command above does not specify any compiler options. In this case, the compiler will use only default options to compile and assemble your program. You can change the operation of the compiler by specifying the options you want when you run the compiler.

To specify options to the compiler, type the appropriate option or options on the command line as shown in the first example above. Options should be separated with spaces. You must include the '-' or '+' that is part of the option name.

Compiler Command Line Options
The cx6811 compiler accepts the following options:

```
cx6811 [options] <files>
        -a*>  assembler options
        -cl*  path for listings
        -co*  path for objects
        -d*>  define symbol
        -ex   prefix executable
        -e    create error file
        -f*   configuration file
        -g*>  code generator options
        -i*>  path for include
        -l    create listing
        -no   do not use optimizer
        -o*>  optimizer options
        -p*>  parser options
        -s    create only assembler file
        -t*   path for temporary files
        -v    verbose
        -x    do not execute
        +*>   select compiler options
```

File Naming Conventions
The programs making up the C cross compiler generate the following output file names, by default. See the documentation on a specific program for information about how to change the default file names accepted as input or generated as output.

program	input file name	output file name
C Compiler Passes		
cp6811	<file>.c	<file>.1
cg6811	<file>.1	<file>.2
co6811	<file>.2	<file>.s
error listing	<file>.c	<file>.err
assembler listing	<file>.[c|s]	<file>.ls
C header files	<file>.h	
Assembler		
ca6811	<file>.s	<file>.o
source listing	<file>.s	<file>.ls
Linker		
clnk	<file>.o	name required
Programming Support Utilities		

chex	<file>	STDOUT
clabs	<file.h11>	<files>.la
clib	<file>	name required
cobj	<file>	STDOUT

Generating Listings
You can generate listings of the output of any (or all) the compiler passes by specifying the **-l** option to **cx6811**. You can locate the listing file in a different directory by using the **-cl** option.

The example program provided in the package shows the listing produced by compiling the C source file acia.c with the **-l** option:

```
cx6811 -l acia.c
```

Generating an Error File
You can generate a file containing all the error messages output by the parser by specifying the **-e** option to cx6811. For example, you would type:

```
cx6811 -e prog.c
```

The error file name is obtained from the source filename by replacing the *.c* suffix by the *.err* suffix.

Return Status
cx6811 returns success if it can process all files successfully. It prints a message to STDERR and returns failure if there are errors in at least one processed file.

Examples
To echo the names of each program that the compiler runs:

```
cx6811 -v file.c
```

To save the intermediate files created by the code generator and halt before the assembler:

```
cx6811 -s file.c
```

C Library Support
This section describes the facilities provided by the C library. The C cross compiler for MC68HC11 includes all useful functions for programmers writing applications for ROM-based systems.

How C Library Functions are Packaged
The functions in the C library are packaged in three separate sub-libraries; one for machine-dependent routines (the machine library), one that does not support floating point (the integer library) and one that provides full floating point support (the floating point library). If your application does not perform floating point calculations, you can decrease its size and increase its runtime efficiency by including only the integer library.

Inserting Assembler Code Directly
Assembler instructions can be quoted directly into C source files, and entered unchanged into the output assembly stream, by use of the *_asm()* function. This function is not part of any library as it is recognized by the compiler itself.

Linking Libraries with Your Program
If your application requires floating point support, you must specify the floating point library **before** the integer library in the linker command file. Modules common to both libraries will therefore be loaded from

the floating point library, followed by the appropriate modules from the floating point and integer libraries, in that order.

Integer Library Functions

The following table lists the C library functions in the integer library.

```
_asm            iscntrl         memmov      strcpy
abort           isdigit         memset      strcspn
abs             isgraph         printf      strlen
atoi            islower         putchar     strncat
atol            isprint         puts        strcmp
calloc          ispunct         rand        strncpy
div             isspace         realloc     strpbrk
eepcpy          isupper         sbreak      strrchr
eepera          isxdigit        scanf       strspn
eepset          labs            setjmp      strstr
exit            ldiv            sprintf     strtol
free            longjmp         srand       strtoul
getchar         malloc          sscanf      tolower
gets            memchr          strcat      toupper
isalnum         memcmp          strchr      vprintf
isalpha         memcpy          strcmp      vsprintf
```

Floating Point Library Functions

```
acos            cosh            log         sinh
asin            exp             log10       sprintf
atan            fabs            modf        sqrt
atan2           floor           pow         sscanf
atof            fmod            printf      strtod
ceil            frexp           scanf       tan
cos             ldexp           sin         tanh
```

Common Input/Output Functions

Six of the functions that perform stream output are included in both the integer and floating point libraries. The functionalities of the versions in the integer library are a subset of the functionalities of their floating point counterparts. The versions in the integer library cannot print or manipulate floating point numbers. These functions are: *printf*, *scanf*, *sprintf*, *sscanf*, *vprintf* and *vsprintf*.

Functions Implemented as Macros

Five of the functions in the C library are actually implemented as “macros”. Unlike other functions, which (if they do not return int) are declared in header files and defined in a separate object module that is linked in with your program later, functions implemented as macros are defined using #define preprocessor directives in the header file that declares them. Macros can therefore be used independently of any library by including the header file that defines and declares them with your program, as explained below. The functions in the C library that are implemented as macros are: *max*, *min*, *va_arg*, *va_end* and *va_start*.

Including Header Files

If your application calls a C library function, you must include the header file that declares the function at compile time, in order to use the proper return type and the proper function prototyping, so that all the expected arguments are properly evaluated. You do this by writing a preprocessor directive of the form:

```
#include <header_name>
```

in your program, where *<header_name>* is the name of the appropriate header file enclosed in angle brackets. The required header file should be included before you refer to any function that it declares.

The names of the header files packaged with the C library and the functions declared in each header are listed below.

<**assert.h**> - Header file for the assertion macro: *assert*.

<ctype.h> - Header file for the character functions: *isalnum, isalpha, iscntrl, isgraph, isprint, ispunct, isspace, isxdigit, isdigit, isupper, islower, tolower* and *toupper*.

<**float.h**> - Header file for limit constants for floating point values.

<**io.h**> - Header file for input-output registers. Each register has an upper-case name which matches the standard Motorola definition. They are mapped at a base address defaulted to 0x1000. Specifics I/O header files are provided for the MC68HC11K4, MC68HC11F1 and MC68HC11C0 respectively called iok4.h, iof1.h and ioc0.h.

<**limits.h**> - Header file for limit constants of the compiler.

<**math.h**> - Header file for mathematical functions: *acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan* and *tanh*.

<**setjmp.h**> - Header file for nonlocal jumps: *setjmp* and *longjmp*

<**stdarg.h**> - Header file for walking argument lists: *va_arg*, *va_end* and *va_start*. Use these macros with any function you write that must accept a variable number of arguments.

<**stddef.h**> - Header file for types: *size_t*, *wchar_t* and *ptrdiff_t*.

<**stdio.h**> - Header file for stream input/output*: getchar, gets, printf, putchar, puts* and *sprintf*.

<**stdlib.h**> - Header file for general utilities: *abs, abort, atof, atoi, atol, div, exit, labs, ldiv, rand, srand, strtod, strtol* and *strtoul*.

<**string.h**> - Header file for string functions: *memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn* and *strstr*.

Functions returning int - C library functions that return **int** and can therefore be called without any header file, since **int** is the default function return type that the compiler assumed by default, are: *isalnum, isalpha, iscntrl, isgraph, isprint, ispunct, isspace, isxdigit, isdigit, isupper, islower, sbreak, tolower* and *toupper*.

USING THE PROGRAMMING SUPPORT UTILITIES AND DEBUGGING SUPPORT

This chapter describes each of the programming support utilities packaged and the debugging support available with the C cross compiler targeting MC68HC11. The following utilities are available:

chex - translates object module format
clabs - generates absolutes listing
clib - builds and maintains libraries
cobj - examines objects modules

The description of each utility tells you what tasks it can perform, the command line options it accepts, and how you use it to perform some commonly required operations. At the end of the chapter are a series of examples that show you how to combine the programming support utilities to perform more complex operations.

The chex Utility

You use the **chex** utility to translate executable images produced by clnk to one of several hexadecimal interchange formats. These formats are: Intel standard hex format, and Motorola S-record format. You can also use chex to override text and data biases in an executable image or to output only a portion of the executable.

The executable image is read from the input file <file>.

Command Line Options
chex accepts the following command line options, each of which is described in detail below:

```
chex [options] file
```

-a## the argument file is a considered as a pure binary file and ## is the output address of the first byte.

-f? define output file format. If ‘i’ is specified (-fi), Intel hex format is produced. If ‘m’is specified (-fm), Motorola format is produced. If ‘2’ is specified (-f2), Motorola format with S2 records is produced. Default is to produced Motorola S-Records (-fm). Any other letter will select the default format.

-h do not output the header sequence if such a sequence exists for the selected format.

+h* insert * in the header sequence if such a sequence exists for the selected format.

-m# output # maximum data bytes per line. Default is to output 32 bytes per line.

-n*> output only segments whose name is equal to the string *. Up to twenty different names may be specified on the command line. If there are several segments with the same name, they will all be produced.

-o* write output module to file *. The default is STDOUT.

Return Status
chex returns success if no error messages are printed; that is, if all records are valid and all reads and writes succeed. Otherwise it returns failure.

Examples

The file hello.c, consisting of:

```
char *p = {"hello world"};
```

when compiled produces the following the following Motorola S-record format:

```
chex hello.o
S00A000068656C6C6F2E6F44
S1110000020068656C6C6F20776F726C640090
S9030000FC
```

and the following Intel standard hex format:

```
chex -fi hello.o
:0E000000020068656C6C6F20776F726C640094
:00000001FF
```

The clabs Utility

clabs processes assembler listing files with the associated executable file to produce listing with updated code and address values.

clabs decodes an executable file to retrieve the list of all the files which have been used to create the executable. For each of these files, *clabs* looks for a matching listing file produced by the compiler (“.ls” file). If such a file exists, *clabs* creates a new listing file (“.la” file) with absolute addresses and code, extracted from the executable file.

To be able to produce any results, the compiler must have been used with the ‘-l’ option.

Command Line Options
clabs accepts the following command line options, each of which is described in detail below.

```
clabs [options] file
```

-l process files in the current directory only. Default is to process all the files of the application.

-s* specifies the output suffix, including the dot ‘.’ character if required. Default is “.la”

-v be verbose. The name of each module of the application is output to STDOUT.

<*file*> specifies one file, which must be in executable format.

Return Status
clabs returns success if no error messages are printed; that is, if all reads and writes succeed. Otherwise it returns failure.

Examples
For example, the following command line:

```
clabs -v acia.h11
```

will output:

```
crts.ls
acia.ls
```

vector.ls

and creates the following files:

crts.la
acia.la
vector.la

The following command line:

```
clabs -a.lx acia.h11
```

will generate:

crts.lx
acia.lx
vector.lx

The clib Utility

clib builds and maintains object module libraries. clib can also be used to collect arbitrary files in one place. <library> is the name of an existing library file or, in the case of replace or create operations, the name of the library to be constructed.

Command Line Options
clib accepts the following command line options, each of which is described in detail below:

```
clib [options] <library> <files>
```

-c creates a library containing <files>. Any existing <library> of the same name is removed before the new one is created.

-d deletes from the library the zero or more files in <files>.

-i* takes object files from a list *. You can put several files per line or put one file per line. Each lines can include comments. They must be prefixed by the '#' character. If the command line contains <files>, then <files> will be also added to the library.

-l when a library is built with this flag set, all the modules of the library will be loaded at link time. By default, the linker only loads modules necessary for the application.

-r in an existing library, replaces the zero or more files in <files>. If no library <library> exists, create a library containing <files>. The files in <files> not present in the library are added to it.

-s lists the symbols defined in the library with the module name to which they belong.

-t lists the files in the library.

-v be verbose

-x extracts the files in <files> that are present in the library into discrete files with the same names. If no <files> are specified, all files in the library are extracted.

At most one of the options -[c r t x] may be specified at the same time. If none of these is specified, the -t option is assumed.

Return Status
clib returns success if no problems are encountered. Otherwise it returns failure. After most failures, an error message is printed to STDERR and the library file is not modified. Output from the **-t**, **-s** options, and verbose remarks, are written to STDOUT.

Examples
To build a library and check its contents:

```
clib -c libc one.o two.o three.o
clib -t libc
```

will output:

```
one.o
two.o
three.o
```

To build a library from a list file:

```
clib -ci list libc six.o seven.o
```

where list contains:

```
# files for the libc library
one.o two.o three.o
four.o
five.o
```

The cobj Utility

You use cobj to inspect relocatable object files or executable. Such files may have been output by the assembler or by the linker. cobj can be used to check the size and configuration of relocatable object files or to output information from their symbol tables.

Command Line Options
cobj accepts the following options, each of which is described in detail below.

```
cobj [options] file
```

<file> specifies a file, which must be in relocatable format or executable format.

-d output in hexadecimal the data part of each section.

-h display all the fields of the object file header.

-n display the name, size and attribute of each section.

-r output in symbolic form the relocation part of each section.

-s display the symbol table.

-x display the debug symbol table.

If none of these options is specified, the default is **-hns**.

Return Status
cobj returns success if no diagnostics are produced (*i.e.* if all reads are successful and all file formats are valid).

Examples
For example, to get the symbol table:

```
cobj -s alloc.o

symbols:

_main:     0000003e section .text defined public
_outch:    0000001b section .text defined public
_buffer:   00000000 section .bss defined public
_ptecr:    00000000 section .bsct defined public zpage
_getch:    00000000 section .text defined public
_ptlec:    00000002 section .bsct defined public zpage
_recept:   00000028 section .text defined public
```

Fonction C Cosmic getchar()

```
/*    GET CHARACTER FROM SCI INPUT
 *    Copyright (c) 1995 by COSMIC Software
 */
#include <io.h>

#define RDRF      0x20

/*    read a character with echo */
int getchar(void)
      {
      char c;

      while (!(SCSR & RDRF))
            ;
      c = SCDR;
      if (c == '\r')
            c = '\n';
      return (putchar(c));
      }
```

Fonction C Cosmic putchar()

```
/*    PUT A CHARACTER TO SCI OUTPUT
 *    Copyright (c) 1995 by COSMIC Software
 */
#include <io.h>

#define TRDE      0x80

/*    output a character
 */
int putchar(char c)
      {
      for (;;)
            {
            while (!(SCSR & TRDE))
                  ;
            SCDR = c;
            if (c == '\n')
                  c = '\r';
            else
                  break;
            }
      return (c);
      }
```

- ANNEXE 3 -

Présentation du format SRecord

FORMAT MOTOROLA :

An S-record file consists of a sequence of specially formatted ASCII character strings. An S-record will be less than or equal to 78 bytes in length.

The order of S-records within a file is of no significance and no particular order may be assumed.

The general format of an S-record follow:

```
+------------------//------------------//----------------------+
| type | count | address  |            data           | checksum |
+------------------//------------------//----------------------+
```

type	A char[2] field. These characters describe the type of record (S0, S1, S2, S3, S5, S7, S8, or S9).
count	A char[2] field. These characters when paired and interpreted as a hexadecimal value, display the count of remaining character pairs in the record.
address	A char[4,6, or 8] field. These characters grouped and interpreted as a hexadecimal value, display the address at which the data field is to be loaded into memory. The length of the field depends on the number of bytes necessary to hold the address. A 2-byte address uses 4 characters, a 3-byte address uses 6 characters, and a 4-byte address uses 8 characters.
data	A char [0-64] field. These characters when paired and interpreted as hexadecimal values represent the memory loadable data or descriptive information.
checksum	A char[2] field. These characters when paired and interpreted as a hexadecimal value display the least significant byte of the ones complement of the sum of the byte values represented by the pairs of characters making up the count, the address, and the data fields.

Each record is terminated with a line feed. If any additional or different record terminator(s) or delay characters are needed during transmission to the target system it is the responsibility of the transmitting program to provide them.

S0 Record The type of record is 'S0' (0x5330). The address field is unused and will be filled with zeros (0x0000). The header information within the data field is divided into the following subfields.

```
                              mname       is char[20] and is the
                                          module name.
                              ver         is char[2] and is the
                                          version number.

                              rev         is char[2] and is the
                                          revision number.
                              description is char[0-36] and is a
                                          text comment.
```

Each of the subfields is composed of ASCII bytes whose associated characters, when paired, represent one byte hexadecimal values in the case of the version and revision numbers, or represent the hexadecimal values of the ASCII characters comprising the module name and description.

S1 Record The type of record field is 'S1' (0x5331). The address field is interpreted as a 2-byte address. The data field is composed of memory loadable data.

S2 Record The type of record field is 'S2' (0x5332). The address field is interpreted as a 3-byte address. The data field is composed of memory loadable data.

S3 Record The type of record field is 'S3' (0x5333). The address field is interpreted as a 4-byte address. The data field is composed of memory loadable data.

S5 Record The type of record field is 'S5' (0x5335). The address field is interpreted as a 2-byte value and contains the count of S1, S2, and S3 records previously transmitted. There is no data field.

S7 Record The type of record field is 'S7' (0x5337). The address field contains the starting execution address and is interpreted as 4-byte address. There is no data field.

S8 Record The type of record field is 'S8' (0x5338). The address field contains the starting execution address and is interpreted as 3-byte address. There is no data field.

S9 Record The type of record field is 'S9' (0x5339). The address field contains the starting execution address and is interpreted as 2-byte address. There is no data field.

EXAMPLE

Shown below is a typical S-record format file.

```
S00600004844521B
S1130000285F245F2212226A000424290008237C2A
```

```
S113001000020008000826290018538123410018 13
S113002041E900084E42234300182342000824A952
S107003000144ED492
S5030004F8
S9030000FC
```

The file consists of one S0 record, four S1 records, one S5 record and an S9 record.

The S0 record is comprised as follows:

S0	S-record type S0, indicating it is a header record.
06	Hexadecimal 06 (decimal 6), indicating that six character pairs (or ASCII bytes) follow.
00 00	Four character 2-byte address field, zeroes in this example.
48	ASCII H, D, and R - "HDR".
1B	The checksum.

The first S1 record is comprised as follows:

S1	S-record type S1, indicating it is a data record to be loaded at a 2-byte address.
13	Hexadecimal 13 (decimal 19), indicating that nineteen character pairs, representing a 2 byte address, 16 bytes of binary data, and a 1 byte checksum, follow.
00 00	Four character 2-byte address field; hexidecimal address 0x0000, where the data which follows is to be loaded.
28 5F 24 5F 22 12 22 6A 00 04 24 29 00 08 23 7C	Sixteen character pairs representing the actual binary data.
2A	The checksum.

The second and third S1 records each contain 0x13 (19) character pairs and are ended with checksums of 13 and 52, respectively. The fourth S1 record contains 07 character pairs and has a checksum of 92.

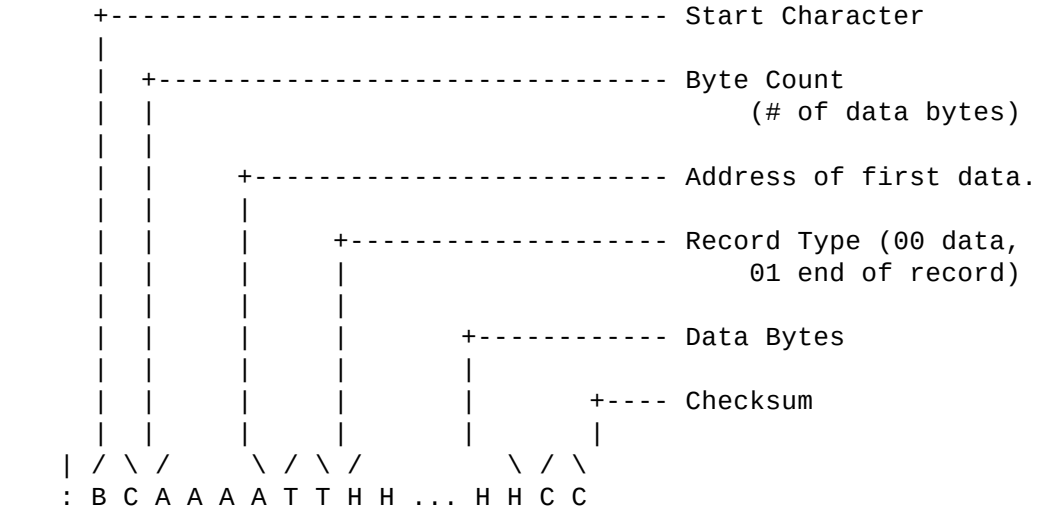
The S5 record is comprised as follows:

S5	S-record type S5, indicating it is a count record indicating the number of S1 records.
03	Hexadecimal 03 (decimal 3), indicating that three character pairs follow.
00 04	Hexadecimal 0004 (decimal 4), indicating that there are four data records previous to this

```
            record.
   F8       The checksum.

The S9 record is comprised as follows:

   S9       S-record type S9, indicating it is a termination
            record.
   03       Hexadecimal 03 (decimal 3), indicating that three
            character pairs follow.

   00 00    The address field, hexadecimal 0 (decimal 0)
            indicating the starting execution address.
   FC       The checksum.
```

FORMAT INTEL :

Intel HEX-ASCII format takes the form:



An examples:

```
:10000000DB00E60F5F1600211100197ED300C3004C
:1000100000000101030307070F0F1F1F3F3F7F7FF2
:01002000FFE0
:00000001FF
```

This information comes from _Microprocessors and Programmed Logic_, Second Edition, Kenneth L. Short, 1987, Prentice-Hall, ISBN 0-13-580606-2.

Provisions have been made for data spaces larger than 64 kBytes. The above reference does not discuss them. I suspect there is a start of segment type record, but I do not know how it is implemented.

14. BIBLIOGRAPHIE

Hardware et software MOTOROLA :

- Page www : www.enseirb.fr/~kadionik/68hc11.html
- Manuel « M68HC11 REFERENCE MANUAL » (M68HC11RM/AD)
- Guide « M68HC11A8 PROGRAMMING REFERENCE GUIDE » (MC68HC11A8RG/AD)
- « Les microcontrôleurs HC11 et leur programmation » de C. Cazaubon
  Edition Masson. Enseignement de l'Electronique
  ISBN 2-225-85527-7
- Manuel « Technical Summary 8-bit Microcontrollers MC68HC11A8, A1, A0 »
- Note d'application AN1060 pour le fonctionnement en mode bootstrap
- Application Snapshot AS-67 pour le fonctionnement en mode spécial test sous PCBUG11
- Logiciel PCBUG11 version 3.42 et son manuel complet « M68HC11 PCBUG11 USER'S MANUAL » (M68PCBUG11/D)
- Manuel « HC11 EVALUATION SYSTEM OPERATIONS MANUAL » qui donne la liste des commandes du moniteur BUFFALO
- Programme BUFFALO disponible pour toutes les versions du 68HC11 téléchargeable à l'adresse suivante :

Software langage C :

- Version d'évaluation et manuel complet « C Cross Compiler User's Guide for Motorola MC68HC11 » de COSMIC Software téléchargeables à l'adresse suivante : http://www.cosmic-us.com
- Note d'application « COSMIC V4.1x 68HC11 C Compiler Package » .

Emulateur Pathfinder 11 :

- Manuel « Embedded development systems USER MANUAL » de ASHLING Microsystems Ltd, l'adresse du site internet est http://www.ashling.com (rubrique CT68HC11 Microprocessor Development System for windows).