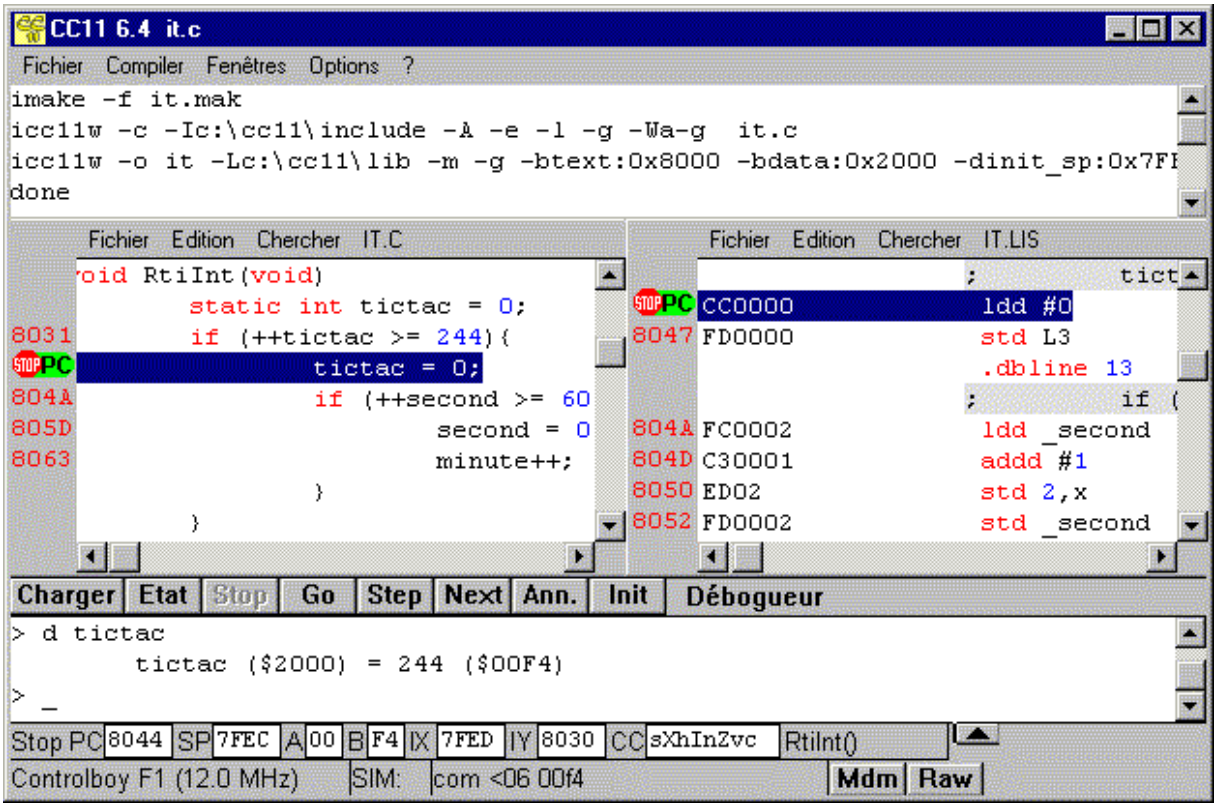
Documentation Controlboy,
Programmation en CC11, Basic11, Assembleur, Prototypage Rapide
Débogueur, Simulateur,
Unité centrale 68HC11, Entrées et Sorties

Controlord, 484, Avenue des Guiols, F 83210 La Farlède. France
Tél. (0033) 04 94 48 71 74 Fax (0033) 04 94 33 41 47
controlord@controlord.fr http://www.controlord.fr

Informations générales



La surface de programmation permet d'écrire, de compiler, de charger et de déboguer des programmes en Assembleur, en Basic ou en C. Le programme vous présente après le lancement deux fenêtres. La fenêtre en haut est la fenêtre principale. Elle vous permet de lancer un éditeur pour éditer le programme source et compiler le programme. La fenêtre en bas contient le débogueur qui vous permet de communiquer avec la carte cible à base du 68HC11. Dans le menu FICHIER vous trouvez les outils pour ouvrir et traiter le programme source principal. COMPILER lance le compilateur. Le curseur d'attente s'affiche pendant la compilation. Après on trouve la sortie des programmes de compilation.

!E <fichier>(<ligne>)

Une telle ligne affiche une erreur. Cliquer deux fois sur cette ligne ouvre une fenêtre pour éditer ce fichier et positionne la source sur la ligne erronée. Le menu FENETRES affiche les fichiers source et tous les fichiers inclus dans ces fichiers par la directive #include. Cliquer sur un fichier ouvre une fenêtre pour éditer ce fichier. Le dialogue OPTIONS vous permet de spécifier les options pour compiler le programme: la commande du compilateur, les options de compilation, les fichiers à compiler, et les répertoires des fichiers sources.

Cliquer deux fois sur une ligne d'un programme Basic ou C affiche les lignes en assembleur généré par le compilateur pour cette ligne en langage de haut niveau. Cliquer deux fois sur une ligne assembleur affiche la ligne en Basic ou C.

Dans la fenêtre du débogueur vous communiquez avec la cible. (Points d'arrêt, pas à pas, table de symboles). Si le débogueur atteint un point d'arrêt (également pour les pas à pas), il déplace la source et sélectionne la ligne de l'instruction. Si votre programme est écrit en Basic ou en C, vous voyez le programme source et le programme assembleur créé par le compilateur. Le débogueur a deux pas à pas: L'un entre dans les sous-programmes, l'autre n'entre pas. Il débogue même un programme qui tourne: Lire et écrire la mémoire, mettre des points d'arrêt.

Préparation du logiciel pour la carte cible

Paramètres standards

Cliquez sur FICHIER, ensuite CONFIGURATION pour configurer le débogueur et le talker de la cible.

Port: Port pour communiquer avec la cible. Vous pouvez choisir entre COM1, COM2, COM3, COM4 et SIM pour travailler avec le simulateur.

Hardware: Choisissez entre plusieurs descriptions de matériel. Si vous avez une autre cible, vous pouvez laisser le champ vide. Tous les paramètres sont alors enregistrés dans le fichier ENV.TXT. Pour plusieurs cibles donnez le nom de la cible, par exemple CIBLE1. Dans le fichier ENV.TXT se trouvera une ligne BOARD=CIBLE1 et les données spécifiques de la cible seront enregistrées dans le fichier CIBLE1.CNF.

Quartz: Sélectionnez la vitesse du 68HC11: Vous pouvez choisir de 1 Mhz à 24 Mhz.

Baud: Sélectionnez la vitesse de la communication avec la cible. Les vitesses sont proposées selon le Quartz de 300 à 57600 baud.

Communication timeout: Temps pour le déboguer pour attendre une réponse de la cible. Si la cible ne répond pas dans ce temps, le déboguer affiche 'Cible ne répond pas'. Un temps de 2000 ms est conseillé. Il faut augmenter ce temps pour des lentes vitesses de transmission.

Talker: Le nom du talker. Sélectionnez un talker. Voir examples suivants.

RAM: Indiquez la taille et l'adresse de la mémoire vive.

EEPROM: Indiquez la taille et l'adresse de la mémoire morte.

Un changement des paramètres sensibles demande un remplacement du talker dans la cible.

Les Talkers

Le déboguer utilise un programme qui s'appelle talker et qui tourne dans la cible.

Fichier	nom interne	reside	Adresse	travaile sur
talkboy.a11	cboy	EEPROM	FE80	EEPROM du 68HC811E2 (Controlboy 1)
talkram.a11	ram	RAM	0	EEPROM interne du 68HC11A1,E1,F1,..
talkxico.a11	xicor	EEPROM	FE80	EEPROM compatible XICOR en mode protection
talkxram.a11	xram	RAM/EEPROM	FE80	RAM ou EEPROM compatible XICOR sans protection
talkslic.a11	slic	EEPROM	FE80	EEPROM Xicor X68C75 (Controlboy 2, 3)
talkcf1.a11	cboyf1	EEPROM	FE00	EEPROM compatible XICOR(Controlboy F1)

Le talker peut être adapté à la cible. Le talker inclut entre autres l'algorithme pour écrire dans l'EEPROM. La commande INITTALKER du débogueur charge le talker dans la cible. Les talkers du CC11 ont l'extension .s.

Le logiciel utilise la ligné série du 68HC11 pour communiquer avec la cible. Les pattes PD1 et PD0 du port D servent comme Transmit Data et Receive Data et doivent être transférés aux normes RS232.

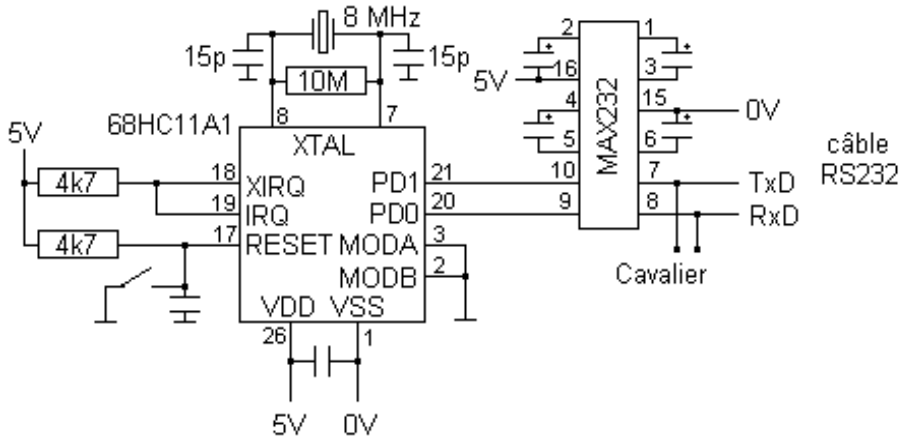
Les déclarations des pointeurs sont essentielles pour le comportement du logiciel.

En Basic11 ces déclarations se trouvent au début du programme. On utilise souvent le fichier START.BAS qui contient ces déclarations. Il faut adapter ces déclarations à la carte cible. Il faut aussi vérifier les définitions des ports qui se trouvent dans ce fichier. Les déclarations des ports B et C dépendent du matériel choisi.

En C on trouve les déclarations des pointeurs dans le OPTIONS, fichier.MAK. Les déclarations des port se trouvent dans le fichier HC11.H.

Enfin il faut s'assurer que dans la version finale c'est bien votre programme qui sera lancé automatiquement après le RESET.

Voici une carte cible minimum à base de 68HC11A1.



Le câble cible - P.C.

68HC11	Max232	PC COM port	
		9 broches	25 broches
PD1 (TxD)	10 - 7	2 (RxD)	3 (RxD)
PD0 (RxD)	9 - 8	3 (TxD)	2 (TxD)
0V		5	7

Dès que vous avez choisi la bonne configuration pour votre matériel, il reste à établir la communication entre le débogueur qui tourne sur le P.C. et votre cible. Reliez donc votre carte cible au port série du P.C., alimentez la carte correctement, adressez-vous à la fenêtre du débogueur et suivez les instructions suivantes selon les caractéristiques de votre cible.

Controlboy 1, 2k EEPROM, 256 RAM

Il suffit de choisir le matériel pendant l'installation. Le talker CBOY est déjà dans l'EEPROM de la carte. Le talker utilise l'EEPROM de FE80 à FFFF et la RAM de 00E9 à 00FF. Après le RESET le talker examine l'entrée D5 (touche T1). Si celle-ci est à zéro, le talker attend une commande par la ligne série. Si celle-ci est à 1, le talker saute à l'adresse $F800 pour exécuter directement l'application.

Basic11	ProgramPointer	$F800	DataPointer	$0002	StackPointer	$00E8
CC11	text	0xF800	data	0x0002	init_sp	0x00E8

Controlboy 2, Controlboy 3, 8k EEPROM, 512 RAM

Il suffit de choisir le matériel pandant l'installation. Le talker SLIC est déja dans l'EEPROM de la carte. Le talker utilise l'EEPROM de FE80 à FFFF et la RAM de 00E9 à 00FF. Après le RESET le talker examine l'entrée D5 (touche T1). Si celle-ci est à zéro, le talker attend une commande par la ligne série. Si celle-ci est à 1, le talker saute à l'adresse $E000 pour exécuter directement l'application.

Basic11	ProgramPointer	$E000	DataPointer	$0002	StackPointer	$01FF
CC11	text	0xE000	data	0x0002	init_sp	0x01FF

Controlboy F1, 32k EEPROM, 32k RAM

Il suffit de choisir le matériel pandant l'installation. Le talker CBOYF1 est déja dans l'EEPROM de la carte. Le talker utilise l'EEPROM de FE00 à FFFF et la RAM de 00E9 à 00FF. Après le RESET le talker examine l'entrée G1 (touche T1). Si celle-ci est à zéro, le talker attend une commande par la ligne série. Si celle-ci est à 1, le talker saute à l'adresse $8000 pour exécuter directement l'application.

Basic11	ProgramPointer	$8000	DataPointer	$2000	StackPointer	$7FFF
CC11	text	0x8000	data	0x2000	init_sp	0x7FFF

68HC11A1, 512 EEPROM, 256 RAM, sans mémoire externe

Il faut charger le talker RAM après chaque RESET à laide de la commande INITTALKER du débogueur dans la RAM de la carte cible.

Basic11	ProgramPointer	$B600	DataPointer	$0002	StackPointer	$00EA
CC11	text	0xB600	data	0x0002	init_sp	0x00EA

Le microprocesseur tourne toujours en mode BOOTSTRAP. Pour que votre programme chargé à l'adresse B600 dans l'EEPROM démarre automatiquement après le RESET, il faut relier Transmit Data à Receive Data du microprocesseur. La mémoire est partagée entre le talker et votre programme.

0000 - 0021	RAM libre pour l'application
0022 - 00CB	Talker
00CC - 00EA	Pile 31 octets
00EB - 00FF	Talker

68HC11E1, 512 EEPROM, 512 RAM, sans mémoire externe

Comme le 68HC11A1. Mais avec de la RAM en plus, on peut réserver les 256 premiers octets au talker.

Basic11	ProgramPointer	$B600	DataPointer	$0100	StackPointer	$01FF
CC11	text	0xB600	data	0x0100	init_sp	0x01FF

68HC11F1, 512 EEPROM, 1k RAM, sans mémoire externe

Il faut charger le talker RAM après chaque RESET à laide de la commande INITTALKER du débogueur dans la RAM de la carte cible. Les 256 premiers octets de la RAM sont réserves au talker.

Basic11	ProgramPointer	$FE00	DataPointer	$0100	StackPointer	$03FF
CC11	text	0xFE00	data	0x0100	init_sp	0x03FF

Le microprocesseur tourne en mode BOOTSTRAP pour charger et déboguer votre programme. Pour que votre programme chargé à l'adresse $FE00 démarre automatiquement après le RESET, il faut une fois initialiser le vecteur du RESET qui se trouve à l'adresse $FFFE dans l'EEPROM. On écrit donc à l'invité du débogueur.

```
mset -w FFFE = FE00
```

Le microprocesseur va donc executer le programme à l'adresse FE00 àpres le RESET en mode SINGLE CHIP.

68HC811E2, 2k EEPROM, 256 RAM, sans mémoire externe

Il faut charger le talker CBOY une fois à laide de la commande INITTALKER du débogueur dans l'EEPROM de la carte cible. Le talker utilise l'EEPROM de FE80 à FFFF et la RAM de 00E9 à 00FF. Après le RESET le talker examine l'entrée D5. Si celle-ci est à zéro, le talker attend une commande par la ligne série. Si celle-ci est à 1, le talker saute à l'adresse $F800 pour exécuter directement l'application.

Basic11	ProgramPointer	$F800	DataPointer	$0002	StackPointer	$00E8
CC11	text	0xF800	data	0x0002	init_sp	0x00E8

68HC11 avec de la mémoire externe, le talker

Vous devrez faire une copie du fichier d'un talker pour l'adapter à votre carte cible.

Sous Basic11 les sources des talkers se trouvent dans le sousrépertoire TALKER11. Les fichiers ont l'extension A11.

Sous CC11 les sources des talkers se trouvent dans le sous-répertoire TALKERS. Les fichiers ont l'extension S.

Le talker talkxico.a11 (ou talkxico.s pour CC11) est le talker le plus probable pour votre cible.

La première parti d'un talker est un preloader d'une taille de $100 octets. La commande INITTALKER va vous guider plus tard de mettre le 68HC11 en mode bootstrap, et le 68HC11 va charger le preloader dans la RAM de $0000 à $00FF. Ensuite le 68HC11 saute à l'adresse $0000. Le preloader va mettre le microprocesseur en mode étendu pour avoir accès à la mémoire externe. Ensuite il charge par la liaison série le talker et il va le programmer dans l'EEPROM externe à la fin de la mémoire. Il attend une EEPROM, compatible XICOR à cet endroit. Notez, que pratiquement toutes les EEPROM sont compatibles de ce type.

Le preloader et le talker se trouvent dans le même fichier source. Vous devrez peut-être adapter le preloader à ce qui est spécifique de votre carte pour accéder à la mémoire externe après l'étiquette PRESTART avant que le preloader charge le talker. Vous devrez également adapter le talker. Après un RESET, le talker saute à l'étiquette TALKSTART. Tout se trouve dans les premiers vingt instructions âpres cette étiquette. Le talker allume un instant une LED à la sortie D3. Le talker examine un interrupteur à l'entrée D5. Si celle-ci est à zéro, le talker attend une commande par la ligne série. Si celle-ci est à 1, le talker saute au début de l'EEPROM pour exécuter directement l'application.

Apres l'adaptation vous devrez compiler le talker.
Sous Basic11, cliquez sur COMPILER.

Sous CC11, il existe un fichier TALKERS.MAK pour compiler tous les talkers. Vous devrez éditer ce fichier pour ajouter votre talker pour le compiler correctement ou editer un makefile pour votre talker. Un talker doit être compilé comme suivant:

```
talkxxx.s19: talkxxx.s
        icc11w -m -R -btext:0 -bdata:0 -dinit_sp:0 -dheap_size:0 talkxxx.s
```

Le preloader doit avoir une taille exacte de $100 octets. Si vous ajouter des octets, il reste des octets libres vers la fin du preloader avant l'étiquette PREFREEBYTES. Enlevez quelques octets. Le talker doit finir à l'adresse $FFFF. Si vous ajouter des octets, il reste des octets libres vers la fin du talker avant l'étiquette TALKFREEBYTES. Notez également, que les lignes marquées POKE indiquent des données qui seront modifiées par le débogueur avant de charger le programme. Le Basic11 vous affiche une erreur, si ce n'est pas le cas. Avec CC11 vous devrez vérifier ces données vous-même: Ouvrez le fichier LIS et vérifiez les lignes marquées #assert.

La configuration du débogueur vous propose les talkers qu'elle a trouvé dans le sous-répertoire. Choisissez votre talker. Ensuite vous devrez effectuer un INITTALKER. Dans le champ de communication le débogueur vous affiche le roulement de l'opération.
Si le chargement du talker s'arrête dès qu'il touche la mémoire externe

- Vérifiez que la mémoire ROM n'est pas allumée. Registre CONFIG, bit 1 ROMON doit être 0. Chargez le talker RAM et tapez >d CONFIG. Vous pouvez changer le registre CONFIG par >CONFIG=$13, il faut un RESET pour que le changement soit pris en compte.
- Vérifiez que votre mémoire externe est accessible. Charger le talker RAM et tapez >HPRIO=$25 pour passer en mode étendu. Vérifiez l'accès à la mémoire externe par la commande MEM du débogueur.

Après cette opération le débogueur doit vous afficher CIBLE STOP.
Ouff.

Pour toutes les cibles

Votre programme devrait exécuter l'instruction assembleur CLI pour permettre la communication entre le débogueur et le talker sur la cible.

Votre programme peut bien sûr écraser les données du talker. Vous pouvez donc charger le programme à l'aide du déboguer mais vous ne pouvez le débogueur.

Enfin, une fois que vous avez initialisé le talker et avant de charger votre programme dans la cible il faut que le débogueur affiche CIBLE STOP ou CIBLE TOURNE.

Vitesse de base

La vitesse du 68HC11, le quartz, influence la vitesse de communication.

Quartz	Vitesse du Bus	Baud	Baud Inittalker
1.000.000	250.000	1200,600,300,150	150
2.000.000	500.000	2400, 1200, 600, 300, 150	300
2.457.600	614.400	19200, 9600, 4800, 2400 1200, 600, 300, 150	2400
4.000.000	1.000.000	4800, 2400, 1200, 600, 300, 150	600
4.915.200	1.228.800	19200, 9600, 4800, 2400, 1200 600, 300, 150	4800
8.000.000	2.000.000	9600, 4800, 2400, 1200, 600, 300, 150	1200
9.830.400	2.457.600	19200, 9600, 4800, 2400, 1200, 600, 300	9600
16.000.000	4.000.000	19200, 9600, 4800, 2400, 1200 600, 300, 150	2400

Unité centrale

Modèle de programmation

La donnée de base de 68HC11 est l'octet. l'octet à 8 bits. Le bit 0 est le bit de poids faible et le bit 7 est le bit de poids fort.

De plus, le 68HC11 connaît des mots de 16 bits. Le bit 0 est aussi le bit de poids faible et le bit 15 est le bit de poids fort. Dans la mémoire l'octet de poids fort se trouve devant l'octet de poids faible.

L'unité centrale connaît trois présentations des nombres:

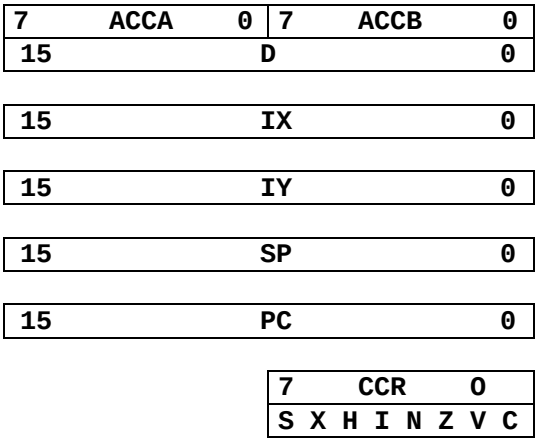
Nombres entiers non signés (Unsigned Integer): Un octet peut avoir des valeurs entre 0 et 255 ($FF), un mot entre 0 et 65535 ($FFFF).

Nombres entiers signés complément à 2 (Two's complement). Un octet peut avoir des valeurs entre -128 ($80) et +127 ($7F), un mot entre -32768 ($8000) et 32767 ($7FFF). Le bit de poids fort indique toujours le signe.

Décimal codé binaire (BCD Binary Coded Decimal): Un octet contient deux chiffres décimaux. Les bits 7,6,5,4 contiennent le chiffre de poids fort, les bits 3,2,1,0 contiennent le chiffre de poids faible. Un octet peut donc avoir des valeurs entre 0 ($00) et 99 ($99). Cette présentation est peu utilisée.

Des **adresses** ont 16 bits et adressent des octets. L'espace d'adressage de l'unité centrale comprend donc 2^{16} = 65536 octets ($0000..$FFFF). On trouve toutes les mémoires comme la RAM, la ROM, l'EEPROM, et les registres d'E/S dans l'espace d'adressage.

Registres



C = Carry
V = Overflow
Z = Zero
N = Negativ
I = Interrupt mask
H = Half Carry
X = X Interrupt mask
S = Stop disable

ACCA et **ACCB** sont les deux accumulateurs A et B. Chaque registre comprends 8 bits. le registre double **D** a 16 bits et contient les deux registres A et B.

```
ldaa    #$12        charger la valeur $12 dans le registre A
ldab    #$34        charger la valeur $34 dans le registre B
ldd     #$1234      charger $12 dans le registre A et $34 dans B
```

Les registres **IX** et **IY** servent comme registres d'index pour adresser la mémoire. Il y a peu d'instructions pour changer ces registres. Mais beaucoup d'instructions les utilisent pour adresser la mémoire. L'utilisation du registre IX peut améliorer un programme, par exemple

```
ldaa    $1033       Charger le registre d'E/S
staa    $1034       Enregistrer dans le registre d'E/S
```

Au lieu de cette séquence on peut écrire

```
ldx     #$1000      Le registre IX adresse maintenant l'espace d'E/S
ldaa    $33,x       Charger le registre d'E/S
staa    $34,x       Enregistrer dans le registre d'E/S
```

Le pointeur de pile **SP** adresse toujours le premier octet, qui n'est pas encore réservé sur la pile.

Le registre d'instructions **PC** adresse l'instruction à exécuter.

Codes conditions

Le registre des codes conditions **CCR** contient huit bits. Beaucoup d'opérations changent ces bits. Quelques opérations changent leur comportement selon ces bits. Par exemple

decb	diminuer de 1 le registre B change le CCR bits N, Z et V
beq	brancher si Z=1, ce qui veut dire si le registre B est égal à 0

Les bits du registre CCR ont les significations suivantes:

Z Zéro: Le résultat de la dernière opération est zéro. Après une comparaison, zéro indique que les deux opérandes ont la même valeur.

N Négatif: Le résultat de la dernière opération est négatif. Seulement pour des nombres entiers signés en complément à 2.

V Overflow, Débordement: Le résultat de la dernière opération a causé un débordement. Seulement pour des nombres entiers signés en complément à 2.

C Carry, Retenue: Le résultat de la dernière opération a causé une retenue. Le bit permet des opérations sur des opérandes plus longs que 16 bits.

H Half Carry, Demi-retenue: Ce bit permet des opérations sur des opérandes en présentation décimal codé binaire. Des opérations comme ADD positionnent ce bit et l'opération suivant DAA l'utilise pour corriger le résultat.

I Masque d'interruption: Quand le bit est mis à 1, les interruptions sont ignorées. Quand une interruption est acceptée par l'unité centrale, le bit est automatiquement mis à 1 pour éviter des interruptions récursives.

X Masque d'interruption: Quand le bit est mis à 1, les interruptions XIRQ sont ignorées.

S Ignorer Stop: Quand le bit est mis à 1, l'instruction STOP est ignorée.

Modes d'adressage

Le 68HC11 connaît cinq modes d'adressage.

Immédiat (Immediate): C'est l'adressage le plus facile. L'opérande se trouve directement dans le programme derrière le code de l'instruction. Au niveau assemblage on exprime ce mode d'adressage avec le symbole dièse (#).

```
86  7F          ldaa    #127            charger 127 dans ACCA
8B 10           adda    #$10            additionner 16
CE 10 00        ldx     #$1000          charger l'adresse $1000 dans le registre IX
```

Direct 16 bits: l'adresse de l'opérande se trouve dans les deux octets suivants le code de l'instruction. L'opérande peut être un octet ou un mot de 16 bits.

```
B6 10 33        ldaa    $1033           charger un octet situé à l'adresse $1033
FF 11 00        stx     $1100           enregistrer le registre IX à l'adresse $1100:1101
BD F8 77        jsr     $F877           sauter au sous-programme à l'adresse $F877
```

Direct 8 bits: l'adresse de l'opérande se trouve dans l'octet suivant le code de l'instruction. L'opérande peut être un octet ou un mot de 16 bits. L'adressage permet d'adresser les octets aux adresses $0000..$00FF, dans la première page de la mémoire vive. Au niveau d'assemblage on ne voit pas la différence entre cet adressage et l'adressage de 16 bits. C'est l'assembleur qui prend automatiquement cet adressage lorsque c'est possible.

```
96  33          ldaa    $0033           charger un octet de l'adresse $0033
DF 80           stx     $0080           enregistrer le registre IX à l'adresse $0080:0081
9D B0           jsr     $00B0           sauter au sous-programme à l'adresse $00B0
```

Relatif PC: Cet adressage est réservé aux instructions de branchement. L'adresse du branchement est calculée par l'adresse de l'instruction diminué jusqu'à -128 ou augmenté jusqu'à +127 par l'octet qui se trouve derrière le code de l'instruction. Au niveau de l'assembleur on indique l'adresse absolue et c'est l'assembleur qui calcule l'adressage.

```
1F 08 20 FC     brclr   $08,x  $20  *   attendre que le bit $20 atteigne la valeur 1
```

Comme cet adressage est réservé à un espace très étroit, l'assembleur remplace automatiquement une instruction qui se rend à une adresse trop loin comme

```
        2C ??           bge     troploin        brancher si plus grand ou égal
par deux instructions
        2D 03           blt     L               brancher si plus petit
        7E FA 00        jmp     troploin        sauter si plus grand ou égal
                    L   equ     *
```

Indexé (IX, IY) : L'adresse est calculé par le contenu du registre IX ou IY en ajoutant une valeur entre $00 et $FF qui se trouve dans l'octet suivant le code de l'instruction. Le résultat est sur 16 bits et adresse donc la totalité d'espace d'adressage. Cet adressage est variable et chaque sous-programme qui veut adresser n'importe quelle adresse doit s'en servir. Le programme suivant efface la mémoire vive de l'adresse $0040 à $004F

```
CE 00 40        ldx     #$0040          Commencer à l'adresse $0040
86 10           ldaa    #16             ACCA compteur: 16 octets
6F 00       L:  clr     0,x             Effacer l'octet à l'adresse dans IX
08              inx                     Augmenter de 1 l'adresse
4A              deca                    Diminuer de 1 le compteur
26 FA           bne     L               Continuer si c'est pas encore fini
```

Vue d'ensemble du jeu d'instructions

Mnémonique	Opérandes	CCR	Opération
Charger et Enregistrer			
LDAA/B	#, dir, ext, ind	NZV	ACCx = M
LDD/S/X/Y	##, dir, ext, ind	NZV	ACCD/SP/IX/IY = M:M+1
PSHA/B/X/Y		-	push A/B/IX/IY
PULA/B/X/Y		-	pop A/B/IX/IY
STAA/B	dir, ext, ind	NZV	M = ACCx
STD/S/X/Y	dir, ext, ind	NZV	M:M+1 = ACCD/SP/IX/IY
TAB		NZV	ACCB = ACCA
TAP		tous	CCR = ACCA
TBA		NZV	ACCA = ACCB
TPA		-	ACCA = CCR
TSX/Y		-	IX/IY = SP+1
TXS/TYS		-	SP = IX/IY - 1
XGDX/Y		-	ACCD <=> IX/IY
Arithmétique			
ABA		HNZV C	ACCA += ACCB
ADCA/B	#, dir, ext, ind	HNZV C	ACCx += M + C
ADDA/B	#, dir, ext, ind	HNZV C	ACCx += M
ADDD	##, dir, ext, ind	NZVC	ACCD += M:M+1
ANDA/B	#, dir, ext, ind	NZV	ACCx &= M
CBA		NZVC	Comparer: ACCA-ACCB
CLRA/B/m	ext, ind	NZVC	ACCx/M = 0
CMPA/B	#, dir, ext, ind	NZVC	ACCx - M
COMA/B/m	ext, ind	NZVC	ACCx/M = ~ ACCx/M
CPD	##, dir, ext, ind	NZVC	ACCD - M:M+1
DAA		NZVC	régler ACCA pour BCD
DECA/B/m	ext, ind	NZV	ACCx/M --
EORA/B	#, dir, ext, ind	NZV	ACCx ^= M ou exclusif
FDIV		ZVC	IX,ACCD = ACCD/IX fractional
IDIV		ZVC	IX,ACCD = ACCD/IX non signé
INCA/B/m	ext, ind	NZV	ACCx/M ++
MUL		C	ACCD = ACCA * ACCB
NEGA/B/m	ext, ind	NZVC	ACCx/M = 0 - ACCx/M
ORAA/B	#, dir, ext, ind	NZV	ACCx |= M
SBA		NZVC	ACCA -= ACCB
SBCA/B	#, dir, ext, ind	NZVC	ACCx -= M + C
SUBA/B	#, dir, ext, ind	NZVC	ACCx -= M
SUBD	##, dir, ext, ind	NZVD	ACCD -= M
TSTA/B/m	ext, ind	NZVC	ACCx/M - 0
Décalage, Rotation			
ASLA/B/m	ext, ind	NZVC	ACCx/M arithm. shift left 1 bit
ASLD		NZVC	ACCD arithm. shift left double 1 bit
ASRA/B/m	ext, ind	NZVC	ACCx/M arithm. shift right 1 bit
LSLA/B/m	ext, ind	NZVC	ACCx/M shift left 1 bit
LSLD		NZVC	ACCD shift left double 1 bit
LSRA/B/m	ext, ind	NZVC	ACCx/M shift right 1 bit; bit7=0
LSRD		NZVC	ACCD shift right 1 bit; bit15=0
ROLA/B/m	ext, ind	NZVC	ACCx/M rotate left 1 bit thru Carry
RORA/B/m	ext, ind	NZVC	ACCx/M rotate right 1 bit thru Carry

Opérations sur bits			
BCLR	dir, ind mask	NZV	M &= ~mask
BSET	dir, ind mask	NZV	M |= mask
BITA/B	#, dir, ext, ind	NZV	Bit test: ACCx & M
Registres d'index			
ABX/Y		-	IX/Y += ACCB
CPX/Y	##, dir, ext, ind	NZVC	Comparer IX/Y - M:M+1
DES		-	SP --
DEX/Y		Z	IX/Y --
INS		-	SP++
INX/Y		Z	IX/Y++
Branchement, Contrôle			
BCC,BCS,BEQ,BGE, BGT,BHI,BHS,BLE, BLO,BLS,BLT,BMI, BNE,BPL,BVC,BVS	rel	-	Brancher conditional
BRA	rel	-	Brancher
BSR	rel	-	Brancher au sous-programme
BRCLR	dir, ind mask rel	-	Brancher si les bits sont à 0
BRSET	dir, ind mask rel	-	Brancher si les bits sont à 1
JMP	ext, ind	-	Sauter
JSR	dir, ext, ind	-	Sauter au sous-programme
RTI		tous	Retour d'interruption
RTS		-	Retour de sous-programme
STOP		-	
SWI		I	Interruption logicielle
WAI		-	Attendre une interruption
Opérations sur CCR			
CLC/I/V		CIV	Mettre à 0 Carry / Int / Overflow
SEC/I/V		CIV	Mettre à 1 Carry / Int / Overflow

Opérands	**Syntaxe**	
#	#<$00..$FF>	imédiat 8 bits
##	#<$00..$FFFF>	imédiat 16 bits
dir	<$00..$FF>	adressage direct 8 bits
ext	<$0000..$FFFF>	adressage direct 16 bits
ind	<$00..$FF>,X	indexé par IX
	<$00..$FF>,Y	indexé par IY
rel	<-128..+127>	relatif PC
mask	<$00..$FF>	Masque, 2. opérand

ABA

Add ACCB to ACCA
Addition ACCB à ACCA

ABA

Additionne les accumulateurs A et B et place le résultat dans l'accumulateur A

CCR	H	mis pour supporter la présentation BCD par l'opération DAA
	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si le résultat a causé un retenue

Notation	1B	ABA

ABX/Y

Add ACCB to IX/IY
Addition ACCB à IX/IY

ABX/Y

Additionne l'accumulateur B au registre IX ou IY. Le contenu de l'accumulateur B est traité comme un nombre non signé ($00..$FF).

CCR	non affecté	

Notation	3A	ABX
	18 3A	ABY

ADC

Add with Carry
Addition avec retenue

ADC

Additionne l'accumulateur, l'opérande et le bit C du registre CCR et place le résultat dans l'accumulateur.

CCR	H	mis pour supporter la présentation BCD par l'opération DAA
	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si le résultat a causé une retenue

Notation	89 xx	ADCA	#<valeur 8 bits>
	C9 xx	ADCB	#<valeur 8 bits>
	99 xx	ADCA	<adresse 8 bits>
	D9 xx	ADCB	<adresse 8 bits>
	B9 xx xx	ADCA	<adresse 16 bits>
	F9 xx xx	ADCB	<adresse 16 bits>
	A9 xx	ADCA	<décalage 8 bits>,X
	E9 xx	ADCB	<décalage 8 bits>,X
	18 A9 xx	ADCA	<décalage 8 bits>,Y
	18 E9 xx	ADCB	<décalage 8 bits>,Y

ADD

Add without Carry
Addition sans retenue

ADD

Additionne l'accumulateur, l'opérande et place le résultat dans l'accumulateur.

CCR	H	mis pour supporter la présentation BCD par l'opération DAA
	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si le résultat a causé un retenue

Notation	8B xx	ADDA	#<valeur 8 bits>
	CB xx	ADDB	#<valeur 8 bits>
	9B xx	ADDA	<adresse 8 bits>
	DB xx	ADDB	<adresse 8 bits>
	BB xx xx	ADDA	<adresse 16 bits>
	FB xx xx	ADDB	<adresse 16 bits>
	AB xx	ADDA	<décalage 8 bits>,X
	EB xx	ADDB	<décalage 8 bits>,X
	18 AB xx	ADDA	<décalage 8 bits>,Y
	18 EB xx	ADDB	<décalage 8 bits>,Y

ADDD

Add Double Accumulator
Addition registre double

ADDD

Additionne le registre double, l'opérande et place le résultat dans le registre double.

CCR	N	mis à 1 si le bit 15 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si le résultat a causé une retenue

Notation	C3 xx xx	ADDD	#<valeur 16 bits>
	D3 xx	ADDD	<adresse 8 bits>
	F3 xx xx	ADDD	<adresse 16 bits>
	E3 xx	ADDD	<décalage 8 bits>,X
	18 E3 xx	ADDD	<décalage 8 bits>,Y

AND

Logical AND
ET logique

AND

Effectue un ET logique de l'accumulateur et de l'opérande et place le résultat dans l'accumulateur. Chaque bit du résultat sera à 1 si les deux bits des deux opérandes sont à 1.

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 0

Notation			
	84 xx	ANDA	#<valeur 8 bits>
	C4 xx	ANDB	#<valeur 8 bits>
	94 xx	ANDA	<adresse 8 bits>
	D4 xx	ANDB	<adresse 8 bits>
	B4 xx xx	ANDA	<adresse 16 bits>
	F4 xx xx	ANDB	<adresse 16 bits>
	A4 xx	ANDA	<décalage 8 bits>,X
	E4 xx	ANDB	<décalage 8 bits>,X
	18 A4 xx	ANDA	<décalage 8 bits>,Y
	18 E4 xx	ANDB	<décalage 8 bits>,Y

ASL

Arithmetic Shift Left
Décalage arithmétique à gauche

ASL

Effectue un décalage d'un bit de l'opérande. Le bit 0 est mis à 0. Le bit de poids fort est transféré dans le bit C du CCR. L'opération correspond à une multiplication par 2.

CCR	N	mis à 1 si le bit de poids fort du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si le bit de poids fort est à 1 avant l'opération

Notation			
	48	ASLA	
	58	ASLB	
	78 xx xx	ASL	<adresse 16 bits>
	68 xx	ASL	<décalage 8 bits>,X
	18 68 xx	ASL	<décalage 8 bits>,Y
	05	ASLD	

ASR

Arithmetic Shift Right
Décalage arithmétique à droite

ASR

Effectue un décalage d'un bit de l'opérande. Le bit de poids fort reste inchangé. Le bit 0 est transféré dans le bit C du CCR. L'opération correspond à une division par 2.

CCR	N	mis à 1 si le bit de poids fort du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si C n'est pas égal à N avant l'opération
	C	mis à 1 si le bit 0 est à 1 avant l'opération

Notation	47	ASRA	
	57	ASRB	
	77 xx xx	ASR	<adresse 16 bits>
	67 xx	ASR	<décalage 8 bits>,X
	18 67 xx	ASR	<décalage 8 bits>,Y

Bcc

Branch conditional
Branchement conditionnel

Bcc

Branche selon la condition.

CCR non affecté

				Branche si
Notation	20 xx	BRA	<-128..+127>	toujours
	21 xx	BRN	<-128..+127>	jamais
	24 xx	BCC	<-128..+127>	C du CCR à 0
	25 xx	BCS	<-128..+127>	C du CCR à 1
	27 xx	BEQ	<-128..+127>	Z du CCR à 1
	2B xx	BMI	<-128..+127>	N du CCR à 1
	26 xx	BNE	<-128..+127>	Z du CCR à 0
	2A xx	BPL	<-128..+127>	N du CCR à 0

Après une comparaison des nombres signés

2C xx	BGE	<-128..+127>	1.opérande >= 2.opérande
2E xx	BGT	<-128..+127>	1.opérande > 2.opérande
2F xx	BLE	<-128..+127>	1.opérande <= 2.opérande
2D xx	BLT	<-128..+127>	1.opérande < 2.opérande

Après une comparaison des nombres non signés

22 xx	BHI	<-128..+127>	1.opérande > 2.opérande
24 xx	BHS	<-128..+127>	1.opérande >= 2.opérande
25 xx	BLO	<-128..+127>	1.opérande < 2.opérande
23 xx	BLS	<-128..+127>	1.opérande <= 2.opérande

Après des opérations avec des nombres signés

28 xx	BVC	<-128..+127>	Débordement à 0
29 xx	BVS	<-128..+127>	Débordement à 1

BCLR

Clear Bits in Memory
Mis à 0 des bits en mémoire

BCLR

Met à 0 un ou plusieurs bits en mémoire.

CCR
N mis à 1 si le bit 7 du résultat est à 1
Z mis à 1 si le résultat est zéro
V mis à 0

Notation

15 xx xx	BCLR	<adresse 8 bits> <masque 8 bits>
1D xx xx	BCLR	<décalage 8 bits>,X <masque 8 bits>
18 1D xx xx	BCLR	<décalage 8 bits>,Y <masque 8 bits>

BIT

Bit Test
Test bit

BIT

Effectue un ET logique entre l'accumulateur et l'opérande. Le résultat positionne les bit du CCR. L'accumulateur n'est pas changé.

CCR
N mis à 1 si le bit 7 est à 1
Z mis à 1 si le résultat est zéro
V mis à 0

Notation

85 xx	BITA	#<valeur 8 bits>
C5 xx	BITB	#<valeur 8 bits>
95 xx	BITA	<adresse 8 bits>
D5 xx	BITB	<adresse 8 bits>
B5 xx xx	BITA	<adresse 16 bits>
F5 xx xx	BITB	<adresse 16 bits>
A5 xx	BITA	<décalage 8 bits>,X
E5 xx	BITB	<décalage 8 bits>,X
18 A5 xx	BITA	<décalage 8 bits>,Y
18 E5 xx	BITB	<décalage 8 bits>,Y

BRCLR

Branch if Bits Clear
Branchement si bits à 0

BRCLR

Effectue un ET logique entre l'octet en mémoire et le masque. Branche si le résultat est zéro, ça veut dire si tous les bits qui sont à 1 dans le masque touchent des bits à 0.

CCR
non affecté

Notation

13 xx xx xx	BRCLR	<adresse 8 bits> <masque 8 bits> <-128..+127>
1F xx xx xx	BRCLR	<décalage 8 bits>,X <masque 8 bits> <-128..+127>
18 1F xx xx xx	BRCLR	<décalage 8 bits>,Y <masque 8 bits> <-128..+127>

BRSET

Branch if Bits Set
Branchement si bits à 1

BRSET

Effectue un ET logique entre le complément de l'octet en mémoire et le masque. Branche si le résultat est zéro, ça veut dire si tous les bits qui sont à 1 dans le masque touchent des bits à 1.

CCR non affecté

Notation			
	12 xx xx xx	BRSET	<adresse 8 bits> <masque 8 bits> <-128..+127>
	1E xx xx xx	BRSET	<décalage 8 bits>,X <masque 8 bits> <-128..+127>
	18 1E xx xx xx	BRSET	<décalage 8 bits>,Y <masque 8 bits> <-128..+127>

BSET

Set Bits in Memory
Mis à 1 des bits en mémoire

BSET

Met à 1 un ou plusieurs bits en mémoire.

CCR		
	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 0

Notation			
	14 xx xx	BSET	<adresse 8 bits> <masque 8 bits>
	1C xx xx	BSET	<décalage 8 bits>,X <masque 8 bits>
	18 1C xx xx	BSET	<décalage 8 bits>,Y <masque 8 bits>

BSR

Branch to Subroutine
Branchement au sous-programme

BSR

Sauvegarde sur la pile l'adresse de l'instruction suivante. Branche à l'adresse indiquée.

CCR non affecté

Notation			
	8D xx	BSR	<-128..+127>

CBA

Compare Accumalators
Comparaison des accumulateurs

CBA

Compare les accumulateurs A et B et positionne les bits du CCR.

CCR		
	N	mis à 1, si (A-B) est négatif
	Z	mis à 1, si A et B sont égaux.
	V	mis à 1, si (A-B) cause un débordement.
	C	mis à 1, si la valeur absolue d'A est plus petite que la valeur absolue de B

Notation		
	11	CBA

CLC/I/V

Clear Condition Code Bits
Mise à 0 des bits du CCR

CLC/I/V

Met à 0 des bits du CCR

CCR	C	mis à 0 par l'instruction CLC
	I	mis à 0 par l'instruction CLI
	V	mis à 0 par l'instruction CLV

Notation	0C	CLC
	0E	CLI
	0A	CLV

CLR

Clear
Mise à 0

CLR

Met à 0 l'opérande.

CCR	N	0
	Z	1
	V	0
	C	0

Notation	4F	CLRA	
	5F	CLRB	
	7F xx xx	CLR	<adresse 16 bits>
	6F xx	CLR	<décalage 8 bits>,X
	18 6F xx	CLR	<décalage 8 bits>,Y

CMP

Compare
Comparaison

CMP

Compare l'accumulateur et l'opérande et positionne les bits du CCR.

CCR	N	mis à 1, si (l'accumulateur - le 2. opérande) est négatif
	Z	mis à 1, si l'accumulateur et le 2. opérande sont égaux.
	V	mis à 1, si (l'accumulateur - le 2. opérande) cause un débordement.
	C	mis à 1, si la valeur absolue d'accumulateur est plus petits que la valeur absolue du 2. opérande

Notation	81 xx	CMPA	#<valeur 8 bits>
	C1 xx	CMPB	#<valeur 8 bits>
	91 xx	CMPA	<adresse 8 bits>
	D1 xx	CMPB	<adresse 8 bits>
	B1 xx xx	CMPA	<adresse 16 bits>
	F1 xx xx	CMPB	<adresse 16 bits>
	A1 xx	CMPA	<décalage 8 bits>,X
	E1 xx	CMPB	<décalage 8 bits>,X
	18 A1 xx	CMPA	<décalage 8 bits>,Y
	18 E1 xx	CMPB	<décalage 8 bits>,Y

COM

Complement
Complément

COM

Remplace l'opérande par son complément. Chaque bit à 1 est mis à 0, est chaque bit à 0 est mis à 1.

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 0
	C	mis à 1

Notation	43	COMA	
	53	COMB	
	73 xx xx	COM	<adresse 16 bits>
	63 xx	COM	<décalage 8 bits>,X
	18 63 xx	COM	<décalage 8 bits>,Y

CPD/X/Y

Compare Double Registre
Comparaison registre double

CPD/X/Y

Compare l'accumulateur et l'opérande et positionne les bits du CCR.

CCR	N	mis à 1, si (le registre - le 2. opérande) est négatif
	Z	mis à 1, si le registre et le 2. opérande sont égaux.
	V	mis à 1, si (le registre - le 2. opérande) cause un débordement.
	C	mis à 1, si la valeur absolue du registre est plus petits que la valeur absolue du 2. opérande

Notation	1A 83 xx xx	CPD	#<valeur 16 bits>
	8C xx xx	CPX	#<valeur 16 bits>
	18 8C xx xx	CPY	#<valeur 16 bits>
	1A 93 xx	CPD	<adresse 8 bits>
	9C xx	CPX	<adresse 8 bits>
	18 9C xx	CPY	<adresse 8 bits>
	1A B3 xx xx	CPD	<adresse 16 bits>
	BC xx xx	CPX	<adresse 16 bits>
	18 BC xx xx	CPY	<adresse 16 bits>
	1A A3 xx	CPD	<décalage 8 bits>,X
	AC xx	CPX	<décalage 8 bits>,X
	1A AC xx	CPY	<décalage 8 bits>,X
	CD A3 xx	CPD	<décalage 8 bits>,Y
	CD AC xx	CPX	<décalage 8 bits>,Y
	18 AC xx	CPY	<décalage 8 bits>,Y

DAA

Decimal Adjust ACCA
Correction de l'accumulateur A

DAA

Corrige le résultat dans l'accumulateur A après une opération ABA, ADD ou ADC. Si les opérandes étaient en présentation BCD, le résultat sera aussi en présentation BCD.

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si l'opération à causé une retenue

Notation	19	DAA	

DEC

Decrement
Diminution de 1

DEC

Diminue l'opérande de 1.

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)

Notation	4A	DECA	
	5A	DECB	
	7A xx xx	DEC	<adresse 16 bits>
	6A xx	DEC	<décalage 8 bits>,X
	18 6A xx	DEC	<décalage 8 bits>,Y

DES/X/Y

Decrement Double Register
Diminution de 1 d'un registre double

DESX/Y

Diminue le registre double de 1.

CCR	Z	mis à 1 si le résultat est zéro
		DES ne change pas le CCR

Notation	34	DES
	09	DEX
	18 09	DEY

EOR

Exclusive OR
OU exclusif logique

EOR

Effectue un OR exclusif entre l'accumulateur et l'opérande et place le résultat dans l'accumulateur. Chaque bit du résultat sera à 1 si un de deux bits des opérandes est à 1.

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	0

Notation	88 xx	EORA	#<valeur 8 bits>
	C8 xx	EORB	#<valeur 8 bits>
	98 xx	EORA	<adresse 8 bits>
	D8 xx	EORB	<adresse 8 bits>
	B8 xx xx	EORA	<adresse 16 bits>
	F8 xx xx	EORB	<adresse 16 bits>
	A8 xx	EORA	<décalage 8 bits>,X
	E8 xx	EORB	<décalage 8 bits>,X
	18 A8 xx	EORA	<décalage 8 bits>,Y
	18 E8 xx	EORB	<décalage 8 bits>,Y

FDIV

Fractional Divide
Division du reste

FDIV

Divise le reste dans le registre D par le dénominateur dans le registre IX. Place le quotient dans le registre IX et le reste dans le registre D. Le numérateur in D dois être plus petit que le dénominateur in IX. Le quotient dans IX représente une valeur entre 0,000015 ($0001) et 0,99998 ($FFFF). FDIV résout le reste d'une opération IDIV ou FDIV.

CCR	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si D est plus grand que IX
	C	mis à 1 si le dénominateur dans IX est à 0

Notation	03	FDIV

IDIV

Integer Divide
Division

IDIV

Divise le numérateur dans le registre D par le dénominateur dans le registre IX. Place le quotient dans le registre IX et le reste dans le registre D.

CCR	Z	mis à 1 si le résultat est zéro
	V	0
	C	mis à 1 si le dénominateur dans IX est à 0

Notation	02	IDIV

INC

Increment
Augmentation de 1

INC

Augmente l'opérande de 1.

CCR	N	mis à 1 si le bit 7 du résultat est à 1	
	Z	mis à 1 si le résultat est zéro	
	V	mis à 1 si le résultat a causé un débordement (nombres signés)	
Notation	4C	INCA	
	5C	INCB	
	7C xx xx	INC	<adresse 16 bits>
	6C xx	INC	<décalage 8 bits>,X
	18 6C xx	INC	<décalage 8 bits>,Y

INS/X/Y

Increment Double Register
Augmentation de 1 d'un registre double

INS/X/Y

Augmente le registre double de 1.

CCR	Z	mis à 1 si le résultat est zéro
		INS ne change pas le CCR

Notation	31	INS
	08	INX
	18 08	INY

JMP

Jump
Saut

JMP

Saute à l'adresse.

CCR	non affecté		
Notation	7E xx xx	JMP	<adresse 16 bits>
	6E xx	JMP	<décalage 8 bits>,X
	18 6E xx	JMP	<décalage 8 bits>,Y

JSR

Jump to Subroutine
Saut au sous-programme

JSR

Sauvegarde sur la pile l'adresse de l'instruction suivante et saute à l'adresse.

CCR	non affecté		
Notation	9D xx	JSR	<adresse 8 bits>
	BD xx xx	JSR	<adresse 16 bits>
	AD xx	JSR	<décalage 8 bits>,X
	18 AD xx	JSR	<décalage 8 bits>,Y

LDA

Load Accumulator
Chargement de l'accumulateur

LDA

Charge l'accumulateur avec l'opérande

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 0

Notation	86 xx	LDAA	#<valeur 8 bits>
	C6 xx	LDAB	#<valeur 8 bits>
	96 xx	LDAA	<adresse 8 bits>
	D6 xx	LDAB	<adresse 8 bits>
	B6 xx xx	LDAA	<adresse 16 bits>
	F6 xx xx	LDAB	<adresse 16 bits>
	A6 xx	LDAA	<décalage 8 bits>,X
	E6 xx	LDAB	<décalage 8 bits>,X
	18 A6 xx	LDAA	<décalage 8 bits>,Y
	18 E6 xx	LDAB	<décalage 8 bits>,Y

LDD/S/X/Y

Load Double Register
Chargement d'une registre double

LDD/S/X/Y

Charge le registre double D, 0SP, IX ou IY avec l'opérande.

CCR	N	mis à 1 si le bit 15 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 0

Notation	CC xx xx	LDD	#<valeur 16 bits>
	DC xx	LDD	<adresse 8 bits>
	FC xx xx	LDD	<adresse 16 bits>
	EC xx	LDD	<décalage 8 bits>,X
	18 EC xx	LDD	<décalage 8 bits>,Y
	8E xx xx	LDS	#<valeur 16 bits>
	9E xx	LDS	<adresse 8 bits>
	BE xx xx	LDS	<adresse 16 bits>
	AE xx	LDS	<décalage 8 bits>,X
	18 AE xx	LDS	<décalage 8 bits>,Y
	CE xx xx	LDX	#<valeur 16 bits>
	DE xx	LDX	<adresse 8 bits>
	FE xx xx	LDX	<adresse 16 bits>
	EE xx	LDX	<décalage 8 bits>,X
	CD EE xx	LDX	<décalage 8 bits>,Y
	18 CE xx xx	LDY	#<valeur 16 bits>
	18 DE xx	LDY	<adresse 8 bits>
	18 FE xx xx	LDY	<adresse 16 bits>
	1A EE xx	LDY	<décalage 8 bits>,X
	18 EE xx	LDY	<décalage 8 bits>,Y

LSL

Logical Shift Left
Décalage logique à gauche

LSL

Effectue un décalage d'un bit de l'opérande. Le bit 0 est mis à 0. Le bit de poids fort est transféré dans le bit C du CCR. L'opération correspond à une multiplication par 2.

CCR	N	mis à 1 si le bit de poids fort du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si le bit de poids fort est à 1 avant l'opération

Notation	48	LSLA	
	58	LSLB	
	78 xx xx	LSL	<adresse 16 bits>
	68 xx	LSL	<décalage 8 bits>,X
	18 68 xx	LSL	<décalage 8 bits>,Y
	05	LSLD	

LSR

Logical Shift Right
Décalage logique à droite

LSR

Effectue un décalage d'un bit de l'opérande. Le bit de poids fort est mis à 0. Le bit 0 est transféré dans le bit C du CCR.

CCR	N	mis à 0
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le bit 0 est à 1 avant l'opération
	C	mis à 1 si le bit 0 est à 1 avant l'opération

Notation	44	LSRA	
	54	LSRB	
	74 xx xx	LSR	<adresse 16 bits>
	64 xx	LSR	<décalage 8 bits>,X
	18 64 xx	LSR	<décalage 8 bits>,Y
	04	LSRD	

MUL

Multiply
Multiplication

MUL

Multiplie l'accumulateur A et l'accumulateur B et pose le résultat dans le registre D.

CCR	C	mis à 1, si le bit 7 du résultat est à 1

Notation	3D	MUL

NEG

Negate
Négation

NEG

Remplace l'opérande par son complément à 2.

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement ($80).
	C	mis à 1 si l'opérande n'est pas zéro.

Notation	40	NEGA	
	50	NEGB	
	70 xx xx	NEG	<adresse 16 bits>
	60 xx	NEG	<décalage 8 bits>,X
	18 60 xx	NEG	<décalage 8 bits>,Y

NOP

No Operation
Farniente

NOP

CCR	non affecté	
Notation	01	NOP

ORA

Inclusive OR
OU inclusif logique

ORA

Effectue un OR inclusif entre l'accumulateur et l'opérande et place le résultat dans l'accumulateur. Chaque bit du résultat sera à 1 si au moins un des bits des opérandes est à 1.

CCR	N	mis à 1 si le bit 7 du résultat est à 1
	Z	mis à 1 si le résultat est zéro
	V	0

Notation	8A xx	ORAA	#<valeur 8 bits>
	CA xx	ORAB	#<valeur 8 bits>
	9A xx	ORAA	<adresse 8 bits>
	DA xx	ORAB	<adresse 8 bits>
	BA xx xx	ORAA	<adresse 16 bits>
	FA xx xx	ORAB	<adresse 16 bits>
	AA xx	ORAA	<décalage 8 bits>,X
	EA xx	ORAB	<décalage 8 bits>,X
	18 AA xx	ORAA	<décalage 8 bits>,Y
	18 EA xx	ORAB	<décalage 8 bits>,Y

PSH

Push Register onto Stack

Sauvegarde d'un registre sur la pile

PSH

Sauvegarde le registre sur la pile. Le pointeur de pile SP est diminué de 1 ou de 2 selon la taille du registre.

CCR non affecté

Notation		
	36	PSHA
	37	PSHB
	3C	PSHX
	18 3C	PSHY

PUL

Pull Register from Stack

Récupération d'un registre de la pile

PUL

Récupère le registre de la pile et augmente le pointeur de pile SP selon la taille du registre.

CCR non affecté

Notation		
	32	PULA
	33	PULB
	38	PULX
	18 38	PULY

ROL

Rotate Left

Rotation à gauche

ROL

Effectue un décalage d'un bit de l'opérande. Place le bit C du CCR dans le bit 0. Le bit de poids fort est transféré dans le bit C du CCR.

CCR		
	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si C=1 et N=0 ou C=0 et N=1
	C	mis à 1 si le bit 7 est à 1 avant l'opération

Notation			
	49	ROLA	
	59	ROLB	
	79 xx xx	ROL	<adresse 16 bits>
	69 xx	ROL	<décalage 8 bits>,X
	18 69 xx	ROL	<décalage 8 bits>,Y

ROR

Rotate Right
Rotation à droite

ROR

Effectue un décalage d'un bit de l'opérande. Place le bit C du CCR dans le bit 7. Le bit 0 est transféré dans le bit C du CCR.

CCR	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si C=1 et N=0 ou C=0 et N=1
	C	mis à 1 si le bit 0 est à 1 avant l'opération

Notation	46	RORA	
	56	RORB	
	76 xx xx	ROR	<adresse 16 bits>
	66 xx	ROR	<décalage 8 bits>,X
	18 66 xx	ROR	<décalage 8 bits>,Y

RTI

Return from Interrupt
Retour d'interruption

RTI

Effectue un retour d'interruption avec restauration des registres CCR, B, A, IX, IY, PC. Récupère les registre de la pile et augmente le pointeur de pile SP de 9.

CCR	est récupéré de la pile	
Notation	3B	RTI

RTS

Return from Subroutine
Retour de sous-programme

RTS

Effectue un retour de sous-programme. Récupère le registre PC de la pile et augmente le pointeur de pile SP de 2.

CCR	non affecté	
Notation	39	RTS

SBA

Subtract ACCB from ACCA
Soustraction

SBA

Soustrait l'accumulateur B de l'accumulateur A et pose le résultat dans l'accumulateur A.

CCR	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si le résultat a causé un retenue

Notation	10	SBA

SBC

Subtract with Carry
Soustraction avec retenue

SBC

Soustrait l'opérande et le bit C du CCR de l'accumulateur et place le résultat dans l'accumulateur.

CCR	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si la valeur absolue du 2. opérande est plus grande que l'accumulateur

Notation	82 xx	SBCA	#<valeur 8 bits>
	C2 xx	SBCB	#<valeur 8 bits>
	92 xx	SBCA	<adresse 8 bits>
	D2 xx	SBCB	<adresse 8 bits>
	B2 xx xx	SBCA	<adresse 16 bits>
	F2 xx xx	SBCB	<adresse 16 bits>
	A2 xx	SBCA	<décalage 8 bits>,X
	E2 xx	SBCB	<décalage 8 bits>,X
	18 A2 xx	SBCA	<décalage 8 bits>,Y
	18 E2 xx	SBCB	<décalage 8 bits>,Y

SEC/I/V

Set Condition Code Bits
Mis à 1 des bits du CCR

SEC/I/V

Met à 1 des bits du CCR.

CCR	C	mis à 1 par l'instruction SEC
	I	mis à 1 par l'instruction SEI
	V	mis à 1 par l'instruction SEV

Notation	0D	SEC
	0F	SEI
	0B	SEV

STA

Store Register
Mise en mémoire d'un registre

STA

Stocke le registre dans la mémoire.

CCR	N	mis à 1 si le bit 7 est à 1 (nombres signés)
	Z	mis à 1 si le registre est zéro
	V	0

Notation			
	97 xx	STAA	<adresse 8 bits>
	D7 xx	STAB	<adresse 8 bits>
	B7 xx xx	STAA	<adresse 16 bits>
	F7 xx xx	STAB	<adresse 16 bits>
	A7 xx	STAA	<décalage 8 bits>,X
	E7 xx	STAB	<décalage 8 bits>,X
	18 A7 xx	STAA	<décalage 8 bits>,Y
	18 E7 xx	STAB	<décalage 8 bits>,Y
	DD xx	STD	<adresse 8 bits>
	FD xx xx	STD	<adresse 16 bits>
	ED xx	STD	<décalage 8 bits>,X
	18 ED xx	STD	<décalage 8 bits>,Y
	9F xx	STS	<adresse 8 bits>
	BF xx xx	STS	<adresse 16 bits>
	AF xx	STS	<décalage 8 bits>,X
	18 AF xx	STS	<décalage 8 bits>,Y
	DF xx	STX	<adresse 8 bits>
	FF xx xx	STX	<adresse 16 bits>
	EF xx	STX	<décalage 8 bits>,X
	CD EF xx	STX	<décalage 8 bits>,Y
	18 DF xx	STY	<adresse 8 bits>
	18 FF xx xx	STY	<adresse 16 bits>
	1A EF xx	STY	<décalage 8 bits>,X
	18 EF xx	STY	<décalage 8 bits>,Y

STOP

Stop Processing
Stop processeur

STOP

Le processeur arrête tout travail pour réduire la consommation. Les registres et les sorties restent inchangés. Lorsque un RESET, un IRQ ou un XIRQ arrivent le processeur continue le travail.

CCR	non affecté	
Notation	CF	STOP

SUB

Subtract
Soustraction

SUB

Soustrait l'opérande de l'accumulateur et place le résultat dans l'accumulateur.

CCR	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si la valeur absolue du 2. opérande est plus grande que l'accumulateur

Notation	80 xx	SUBA	#<valeur 8 bits>
	C0 xx	SUBB	#<valeur 8 bits>
	90 xx	SUBA	<adresse 8 bits>
	D0 xx	SUBB	<adresse 8 bits>
	B0 xx xx	SUBA	<adresse 16 bits>
	F0 xx xx	SUBB	<adresse 16 bits>
	A0 xx	SUBA	<décalage 8 bits>,X
	E0 xx	SUBB	<décalage 8 bits>,X
	18 A0 xx	SUBA	<décalage 8 bits>,Y
	18 E0 xx	SUBB	<décalage 8 bits>,Y

SUBD

Subtract Double Accumulator
Soustraction du registre double

SUBD

Soustrait l'opérande du registre double et place le résultat dans le registre double.

CCR	N	mis à 1 si le bit 15 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 1 si le résultat a causé un débordement (nombres signés)
	C	mis à 1 si la valeur absolue du 2. opérande est plus grande que le registre

Notation	83 xx xx	SUBD	#<valeur 16 bits>
	93 xx	SUBD	<adresse 8 bits>
	B3 xx xx	SUBD	<adresse 16 bits>
	A3 xx	SUBD	<décalage 8 bits>,X
	18 A3 xx	SUBD	<décalage 8 bits>,Y

SWI

Software Interrupt
Interruption logicielle

SWI

Interrompt le programme, sauve les registres PC, IY, IX, A, B, CRR sur la pile et continue l'exécution à l'adresse qui se trouve à l'adresse $FFF6. L'instruction se comporte comme une interruption. Le talker utilise cette instruction et elle est donc réservé.

CCR	I	mis à 1

Notation	3F	SI

TAB/TBA

Transfer Accumulator
Transfère d'Accumulateur

TAB/TBA

Transfère l'accumulateur à l'autre.

CCR	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 0

Notation	16	TAB
	17	TBA

TAP/TPA

Transfer Condition Codes
Transfère CCR

TAP/TPA

Transfère l'accumulateur A au registre CCR (TAP) ou le registre CCR à l'accumulateur A (TPA).

CCR	L'instruction TAP positionne tous les bits du CCR
	L'instruction TPA n'affecte pas les bits du CCR

Notation	06	TAP
	07	TPA

TST

Test
Test

TST

Teste l'opérande et positionne les bit du CCR selon le résultat.

CCR	N	mis à 1 si le bit 7 du résultat est à 1 (nombres signés)
	Z	mis à 1 si le résultat est zéro
	V	mis à 0
	C	mis à 0

Notation	4D	TSTA	
	5D	TSTB	
	7D xx xx	TST	<adresse 16 bits>
	6D xx	TST	<décalage 8 bits>,X
	18 6D xx	TSTA	<décalage 8 bits>,Y

TSX/Y TX/YS

Transfer Stack Pointer
Transfère du pointeur de pile

TSX/Y TX/YS

Transfère le pointeur de pile augmenté de 1 au registre IX (TSX) ou IY (TSY) , ou transfère le registre IX (TXS) ou IY (TYS) diminué de 1 au pointeur de pile.

CCR non affecté

Notation		
	30	TSX
	18 30	TSY
	35	TXS
	18 35	TYS

WAI

Wait for Interrupt
Attention d'une interruption

WAI

Sauve les registres PC, IY, IX, A, B, CCR sur la pile et attend une interruption. Le processeur entre dans le mode WAIT où la consommation est réduite. Les sorties est entrées continuent à fonctionner.

CCR non affecté

Notation		
	3E	WAI

XGDX/Y

Exchange Double Accumalator and Index Register
Changement du registre double avec un registre d'index

XGDX/Y

Echange le registre double D avec le registre IX ou IY.

CCR non affecté

Notation		
	8F	XGDX
	18 8F	XGDY

Pile

Les opérations BSR et JSR sauvegardent sur la pile l'adresse de l'instruction suivante, pour que le RTS puisse retourner au programme principal. L'adresse a 16 bits et prend donc deux octets dans la pile. Les opérations BSR et JSR diminuent le pointeur de pile. Ansi un sous-programme peut appeler un autre sous-programme et revenir correctement.

Au debut le pointeur de pile est à $E8.

```
F850        BSR   SousA
F852        ...
```

Le programme appelle le sous-programme SousA, l'adresse de retour $F852 se trouve maintenant sur la pile et le pointeur de pile est diminué de 2.

```
F941        JSR   SousB
F944        ...
```

Le sous-programme SousA appelle un autre sous-programme SousB et stocke son adresse de retour $F944 sur la pile.

```
F890        RTS
```

Le sous-programme SousB effectue le RTS qui récupère l'adresse de la pile. Le pointeur de pile est augmenté de 1. Le programme continue son travail dans le sous-programme SousA à l'adresse $F944.

```
F950        RTS
```

Le sous-programme SousA achève son travail et en effectuant un RTS on retourne au programme principal. Le PC est mis à $F852, est le pointer de pile à $E8.

Les interruptions utilisent le même principe de stockage sur la pile. Une interruption sauvegarde sur la pile tous les registres. Pour revenir au programme interrompu il faut exécuter l'opération RTI au lieu de RTS.

Un programme peut utiliser la pile pour stocker des données. L'opération PSH stocke un registre sur la pile et l'opération PUL récupère les données.

```
pshx        sauvegarde IX sur la pile
.....       utilisation du registre IX pour des autres tâches
pulx        récupère IX
```

Un programme doit toujours prévoir assez de place pour la pile.

Interruptions

Des interruptions arrivent au programme par des événements extérieurs. Le programme qui est produit par le générateur de programme utilise l'interruption de l'horloge temps réel. On peut interdire ou autoriser les interruptions par

```
cli     autorise (enable) toutes interruptions.
sei     interdit (disable) toutes interruptions.
```

Le programme qui traite l'interruption doit s'assurer que le programme interrompu reprend son travail avec

```
rti     Return from Interrupt
```

Quand une interruption arrive, l'unité centrale sauvegarde tous les registres sur la pile. Ensuite elle charge selon l'interruption une adresse du programme pour traiter l'interruption.

	Adresses	Interruptions
*	FFD6	SCI Asynchronous Serial Interface (RS232)
	FFD8	SPI
	FFDA	Counter / Timer PA Input Edge
	FFDC	Counter / Timer PA Overflow
	FFDE..FFEE	Timer
	FFF0	Real Time Interrupt
	FFF2	Parallel I/O Handshake
	FFF4	XIRQ
*	FFF6	SWI Software interrupt
*	FFF8	Illegal opcode
	FFFA	COP Failure, Chien de garde
	FFFC	Clock Monitor Fail
*	FFFE	Reset

Les interruptions marquées par * sont prises par le talker.
Par exemple, si une interruption de l'horloge temps réel arrive, l'unité centrale lit à l'adresse $FFF0 l'adresse du sous-programme qui traite cette interruption. Il faut donc écrire dans le programme

```
        org     $FFF0
        fdb     rtiint
```

et ailleurs le sous-programme lui même

```
rtiint: ...
        rti
```

Le 68HC11 sauvegarde tous les registres sur la pile avant de commencer ce sous-programme. L'opération RTI restaure la pile et continue le programme interrompu.

Mémoire avant	SP-8	SP-7	SP-6	SP-5	SP-4	SP-3	SP-2	SP-1	SP
Registre sur la Pile	CCR	ACCB	ACCA	IX poids fort	IX poids faible	IY poids fort	IY poids faible	PC poids fort	PC poids faible
Mémoire après	SP+1	SP+2	SP+3	SP+4	SP+5	SP+6	SP+7	SP+8	SP+9

Les entrées et les sorties

La programmation des entrées et sorties comprend la configuration des ports, l'entrée des données, la sortie des données, la traitement des erreurs et les interruptions. Toutes ces fonctions sont traitées par des registres E/S qui se trouvent dans l'espace d'adressage de $1000 à $105F.

Si dans les chapitres suivants un bit dans un registre et indiqué à 0, il doit toujours être programmé à 0. Si un bit est indiqué à 1, il doit être toujours mis à 1. Si un bit est indiqué à *, il est expliqué dans un autre chapitre.

Aucune borne du 68HC11 ne doit jamais être exposée à une tension inférieure de la masse GND du 68HC11 ou supérieure à l'alimentation VCC 5V.

Port A: entrées numériques et sorties numériques

Le port A contient trois entrées digitales (A0, A1, A2), trois sorties digitales (A4, A5, A6) et deux bornes qui peuvent être programmées comme entrées ou comme sorties.

Les bornes A3 et A7 doivent être déclarées comme entrée (0) ou comme sortie (1) avant l'utilisation dans le registre PACTL. A7 est branché au compteur.

Pour programmer une sortie il suffit d'écrire la valeur dans le registre PORTA. Une programmation d'un bit à 0 met une tension de 0V à la borne. Une programmation d'un bit à 1 met une tension de 5V à la borne.

En lisant le registre PORTA on reçoit pour une sortie la dernière valeur écrite et pour une entrée la valeur da la tension qui se trouve momentanément à la borne. Un bit mis à 0 correspond à une tension entre 0V et 1V, un bit mis à 1 correspond à une tension entre 3,5V et 5V.

Les Timer du temps réel et le compteur utilisent également le registre PACTL.

7	6	5	4	3	2	1	0	
A7	A6	A5	A4	A3	A2	A1	A0	1000 PORTA
DDA7	*	*	*	DDA3	0	*	*	1026 PACTL

68HC11F1: Les bornes du port A d'un 68HC11F1 peuvent être déclarées comme entrée (0) ou sortie (1)

7	6	5	4	3	2	1	0	
DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	1001 DDRA

Port B: sorties numériques

Le port B n'est pas disponible que dans le mode SINGLE CHIP. Le port B a huit sorties digitales. PORTB est programmé comme décrit pour les sorties du port A.

7	6	5	4	3	2	1	0	
B7	B6	B5	B4	B3	B2	B1	B0	1004 PORTB

Port C: entrées numériques et sorties numériques

Le port C n'est pas disponible que dans le mode SINGLE CHIP. Les bornes du port C peuvent être déclarées comme entrée ou comme sortie. Chaque borne doit être déclarée comme entrée (0) ou comme sortie (1) avant l'utilisation dans le registre DDRC. On programme les sorties et les entrées du registre PORTC comme ceux du registre PORTA.

7	6	5	4	3	2	1	0	
C7	C6	C5	C4	C3	C2	C1	C0	1003 PORTC
DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	1007 DDRC

68HC11F1: Le port PORTC se trouve à l'adresse 1006.

7	6	5	4	3	2	1	0	
C7	C6	C5	C4	C3	C2	C1	C0	1006 PORTC

Port B et C sur le X68C75

Si on utilise le X68C75 comme EEPROM, le 68HC11 tourne dans le mode EXPANDED MODE et les ports B et C servent de bus externe. Les deux ports du X68C75 remplacent les ports B et C internes. En mettant dans le registre SCONF le bit DIRB à 0, le port B a 8 entrées. En lisant le registre PORTBI on reçoit la valeur de la tension qui se trouve à la borne. En mettant DIRB à 1, le port B a 8 sorties. Ces sorties sont à collecteur ouvert (BWO=1) ou des sorties normales CMOS (BWO=0). Pour programmer une sortie on écrit la valeur dans le registre PORTB. En mettant le bit DIRC à 0, le port C a 8 entrées. En lisant le registre PORTCI on reçoit la valeur de la tension qui se trouve à la borne. En mettant DIRC à 1, le port C a 8 sorties. Ces sorties sont à collecteur ouvert (CWO=1) ou des sorties normales CMOS (CWO=0). Pour programmer une sortie on écrit la valeur dans le registre PORTC.

7	6	5	4	3	2	1	0	
	1	BWO	CWO	DIRB	DIRC			0420 SCONF
B7	B6	B5	B4	B3	B2	B1	B0	0410 PORTB
BI7	BI6	BI5	BI4	BI3	BI2	BI1	BI0	0430 PORTBI
C7	C6	C5	C4	C3	C2	C1	C0	0408 PORTC
CI7	CI6	CI5	CI4	CI3	CI2	CI1	CI0	0428 PORTCI

Port D: entrées numériques et sorties numériques

Les six bornes du port D peuvent être déclarées comme entrée ou comme sortie. Les bornes D0 et D1 sont prises par la ligne série. Les autres bornes sont programmées comme ceux du port C.

7	6	5	4	3	2	1	0	
		D5	D4	D3	D2	D1	D0	1008 PORTD
		DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	1009 DDRD

Port E: entrées analogiques ou numériques

Le port E contient des entrées analogiques.

Pour protéger les entrées et surtout l'unique convertisseur analogique/numérique il est conseillé de brancher un signal extérieur par une résistance de 1 kΩ à la borne de la puce. Une résistance supérieure à 10 kΩ peut déformer le résultat. Le convertisseur compare les entrées avec les tensions VREFL et VREF (normalement 0V et 5V).

Avant d'utiliser le port, il faut mettre le bit ADPU du registre OPTION à 1 et attendre 100 µs. Pour entamer une conversion on écrit dans le registre ADCTL. Les bits 0,1 et 2 choisissent l'entrée à convertir. Les autres bits sont à 0. Il faut attendre la fin de la conversion qui est signalée par le bit CCF. Ensuite on trouve le résultat dans le registre ADR. Une valeur de 0 ($00) correspond à une tension de 0V, une valeur de 255 ($FF) à une tension de 5V.

Port E peut également servir comme port parallèle avec 8 entrées digitales à PORTE.

7	6	5	4	3	2	1	0	
ADPU	0	0	1	0	0	0	0	1039 OPTION
CCF	0	0	0	0	CC	CB	CA	1030 ADCTL
ADR7	ADR6	ADR5	ADR4	ADR3	ADR2	ADR1	ADR0	1031 ADR
E7	E6	E5	E4	E3	E2	E1	E0	100A PORTE

Port F: sorties numériqued

Le port F existe sur quelques types de 68HC11 comme le 68HC11F1. Il n'existe pas dans la serie E, 68HC11E0, E1, E2. Le port F n'est pas disponible dans le mode EXPANDED. Le port a huit sorties digitales.

7	6	5	4	3	2	1	0	
F7	F6	F5	F4	F3	F2	F1	F0	1005 PORTF

Port G: entrées numériques et sorties numériques

Le port G existe sur quelques types de 68HC11 comme le 68HC11F1. Il n'existe pas dans la serie E, 68HC11E0, E1, E2. Les bornes du port C peuvent être déclarées comme entrée ou comme sortie. Chaque borne doit être déclarée comme entrée (0) ou comme sortie (1) avant l'utilisation dans le registre DDRG. Les bornes G7,G6,G5 et G4 peuvent être utilisées comme CSPROG, CSGEN, CSIO1, CSIO2.

7	6	5	4	3	2	1	0	
G7	G6	G5	G4	G3	G2	G1	G0	1002 PORTG
DDG7	DDG6	DDG5	DDG4	DDG3	DDG2	DDG1	DDG0	1003 DDRG

Interface série RS232

La ligne série connecte la cible au grand frère. Les bornes D0 et D1 du port D servent comme Transmit Data et Receive Data et doivent être transférés aux normes RS232.

Configuration

Pour communiquer avec le déboguer le talker fait déjà la configuration avant que le programme commence. Il utilise une configuration de 8 bit, sans parité, 1 bit de stop. La vitesse est variable. Le registre BAUD permet de sélectionner la vitesse de transmission. La vitesse est calculée par

Q / 64 / X / Y

avec Q la fréquence du quartz. X et Y sont déterminé par le registre BAUD comme indiqué ci-dessous.

SCP1	SCP0	X =
0	0	1
0	1	3
1	0	4
1	1	13

SCR2	SCR1	SCR0	Y =
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Exemple: Q = 8 Mhz, BAUD = 0x30. Vitesse = 8.000.000 / 64 / 13 / 1 = 9615 baud.

Le registre SCCR1 doit être mis à 0.

Si le bit TIE (Transmitter Interrupt Enable) du registre SCCR2 est mis à 1, l'unité centrale déclenche une interruption quand le caractère écrit dans le registre SCDR est complètement transféré sur la ligne série. C'est à votre programme d'installer un sous-programme pour traiter cette interruption.

Si le bit RIE (Receiver Interrupt Enable) du registre SCCR2 est mis à 1, l'unité centrale déclenche une interruption quand un caractère est complètement reçu de la ligne série. Cette interruption est normalement traitée par le talker.

Le talker met les registres suivant le quartz et la vitesse configurés.

7	6	5	4	3	2	1	0	
0	0	1	0	0	SCR2	SCR1	SCR0	102B BAUD
0	0	0	0	0	0	0	0	102C SCCR1
TIE	0	RIE	0	1	1	0	0	102D SCCR2

Transfert des données

Si le bit TTRE (Transmit Data Register Empty) est à 1, un caractère peut être mis dans le registre SCDR pour l'envoyer sur la ligne série.

Si le bit RDRF (Receive Data Register Full) est à 1, le 68HC11 a reçu un caractère. Les bits OR, NF, FE vous donnent des informations supplémentaires sur la réception.

OR (Overrun Error) indique un débordement du récepteur, c'est à dire un caractère est arrivé alors que le précédent n'était pas encore lu. On a donc perdu au moins un caractère.

NF (Noise Flag) indique du bruit sur la ligne série pendant la réception. Le caractère est peut être déformé.

FE (Framing Error) indique une erreur du format pendant la réception. Le récepteur n'a pas identifié le bit de stop. Le caractère est peut être déformé.

Pour recevoir un caractère il faut lire le registre SCSR pour obtenir les erreur et après le registre SCDR pour obtenir le caractère reçu. Les erreurs sont automatiquement mises à 0.

7	6	5	4	3	2	1	0	
TDRE		RDRF		OR	NF	FE		102E SCSR
SCD7	SCD6	SCD5	SCD4	SCD3	SCD2	SCD1	SCD0	102F SCDR

Timer du temps réel

Le timer du temps réel peut servir à interrompre l'unité centrale périodiquement est sert donc de base pour une horloge. Un programme qui veut installer une telle horloge doit déclarer à l'adresse $FFF0 l'adresse du sous-programme à exécuter quand l'interruption du timer arrive.

Les bits RTR1 et RTR0 du registre PACTL précise la vitesse du timer. Pour déclencher le timer il faut mettre le bit RTII (Real Time Interrupt Enable) dans le registre TMSK2 à 1. Le sous-programme qui traite l'interruption doit mettre le bit RTIF (Real Time Interrupt Flag) dans le registre TFLG2 à 1 pour permettre une autre interruption.

Le port A utilise également le registre PACTL. Le compteur utilise également les registres PACTL, TMSK2 et TFLG2. Un programme créé par le générateur de programme utilise le timer du temps réel à 150 Hz.

RTR1	RTR0	RTI Q = 4,9152 Mhz	RTI Q = 8 Mhz
0	0	150 Hz 6,7 ms	244,14 Hz 4,1 ms
0	1	75 Hz 13,3 ms	122,07 Hz 8,2 ms
1	0	37,5 Hz 26,7 ms	61,03 Hz 16,4 ms
1	1	18,75 Hz 53,3 ms	30,52 Hz 32,8 ms

7	6	5	4	3	2	1	0	
0	**RTII**	*	*	**0**	**0**	**0**	**0**	**1024 TMSK2**
0	**RTIF**	*	*	**0**	**0**	**0**	**0**	**1025 TFLG2**
*	*	*	*	*	**0**	**RTR1**	**RTR0**	**1026 PACTL**

Compteur / Timer

Le Pulse Accumulateur PA est en effet un registre qui peut servir comme compteur ou comme timer. Il utilise comme entrée la borne A7 qui doit être programmée comme entrée, DDA7 mis à 1 dans le registre PACTL.

Comme compteur le registre compte les transitions appliquées à la borne. Comme timer il mesure le temps ou l'entrée est mis à 1 (ou à 0 selon le mode). Le registre est alors périodiquement augmenté de 1. La période se calcule après Q / 256 soit pour un quartz de 8 Mhz 31,25 kHz ou 32 µs et pour un quartz de 4,9152 Mhz 19,2 kHZ ou 52 µs. Les bits PMOD et PEDG déterminent le mode de travail.

PMOD	PEDG	
0	0	Compteur. Augmente de 1 quand A7 change de 0 à 1.
0	1	Compteur. Augmente de 1 quand A7 change de 1 à 0.
1	0	Timer. Mesure le temps quand A7 est mis à 1.
1	1	Timer. Mesure le temps quand A7 est mis à 0.

Il faut mettre PAEN (PA enable) dans le registre PACTL à 1 pour faire travailler le compteur / timer. Le résultat se trouve dans le registre PACNT, qui peut être lu et écrit.

Le port A et le timer du temps réel utilisent également le registre PACTL.

7	6	5	4	3	2	1	0	
DDA7	PAEN	PMOD	PEDG	*	0	*	*	1026 PACTL
PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0	1027 PACNT

Compteur / Timer contrôle et interruptions

Si le compteur change de $FF à $00 il met dans le registre TFLG2 le bit POVF (PA Overflow Flag) à 1. Pour mettre le bit à 0, il faut écrire $20 au registre. Si le bit POVI (PA Overflow Interrupt Enable) dans le registre TMSK2 est mis à 1, le compteur déclenche une interruption quand le bit POVF est mis à 1. L'adresse du sous-programme qui traite l'interruption doit être déclarée à l'adresse $FFDC. Ce sous-programme met le bit POVF à 0 pour permettre une autre interruption.

Cette interruption permet de compter plus que 256 événements. Si le sous-programme augmente un compteur dans la mémoire vive à chaque interruption, les deux compteurs de 8 bits présentent en effet un compteur de 16 bits. Elle permet également d'être interrompu après un certain nombre d'événements (par exemple 3). Il faut écrire le complément à 2 du nombre (-3 = $FD) dans le registre PACNT.

Si la borne A7 change de 0 à 1 et PEDG est égal à 0 ou A7 change de 1 à 0 et PEDG est égal à 1, le compteur met le bit PAIF(PA Input Edge Flag) dans le registre TFLG2 à 1. Pour mettre le bit à 0, il faut écrire $10 au registre. Si le bit PAII (PA Input Edge Interrupt Enable) dans le registre TMSK2 est mis à 1, le compteur déclenche une interruption quand le bit PAIF est mis à 1. L'adresse du sous-programme qui traite l'interruption doit être déclarée à l'adresse $FFDA. Ce sous-programme met le bit PAIF à 0 pour permettre une autre interruption.

Cette interruption peut signaler la fin d'une impulsion. Elle permet également de traiter la borne A7 comme une entrée d'interruption masquable externe.

Le timer du temps réel utilise également le registre TMSK2 et TFLG2.

7	6	5	4	3	2	1	0	
0	*	POVI	PAII	0	0	0	0	1024 TMSK2
0	*	POVF	PAIF	0	0	0	0	1025 TFLG2

CC11 Compilateur C

Introduction

Le compilateur vous permet d'écrire des programmes en C sur le système PC hôte pour des cibles à base de MC68HC11 de Motorola.

Le langage C a été créé par Dennis M. Ritchie en 1972 dans les laboratoires d'AT&T pour écrire le système d'exploitation UNIX et pour écrire des applications sur UNIX. Le langage a évolué depuis et 1988 ANSI (American National Standards Institute) a créé la norme ANSI C. C'est ce langage ANSI C qui a gagné une popularité écrasant sur des machines UNIX comme sur les P.C. et pour des systèmes embarqués. C n'est ni un langage très haut ni un langage très lourd. Tout en offrant les éléments des langages de haut niveau, C reste proche du niveau de la machine. On peut facilement mélanger des programmes C avec des programmes assembleur. C est donc parfaitement adapté pour la programmation pour les microprocesseurs sur des systèmes embarqués.

Le compilateur CC11 utilise un langage qui est très proche de la norme ANSI. Cependant quelques options ne sont pas raisonnables sur un système embarqué. Ces changements limitent l'espace du code généré et adaptent le langage à la vitesse limité d'un microprocesseur de 8 bits.

Le compilateur exécute plusieurs optimisations du code pendant la compilation pour créer un programme rapide et limité en espace. Des variables déclarées volatiles sont exclues des toutes optimisations pour éviter les mauvaises surprises d'un programme qui travaille directement sur les ports périphériques.

Le compilateur donne comme résultat un fichier format S records de Motorola.

Ce manuel ne vous informe pas sur le langage C. Il y a des livres pour apprendre la programmation en C. Ce manuel est un complément pour les éléments du langage qui sont spécifique pour la programmation pour des cibles sur système embarqué à base de MC68HC11. Il vous informe également sur l'utilisation du compilateur.

Installation

Il faut exécuter le programme **`install.exe`** qui se trouve sur le CD-ROM et entrer le code qui vous est communiqué normalement sur la facture lors de l'achat.

Ceci installe le logiciel sur le disque dans le répertoire **`\cc11`**. Les exemples suivants supposent que vous avez installé le logiciel dans ce répertoire.

Répertoires des Fichiers

Dans le répertoire **`\cc11`** vous trouvez les sous-répertoires suivants.

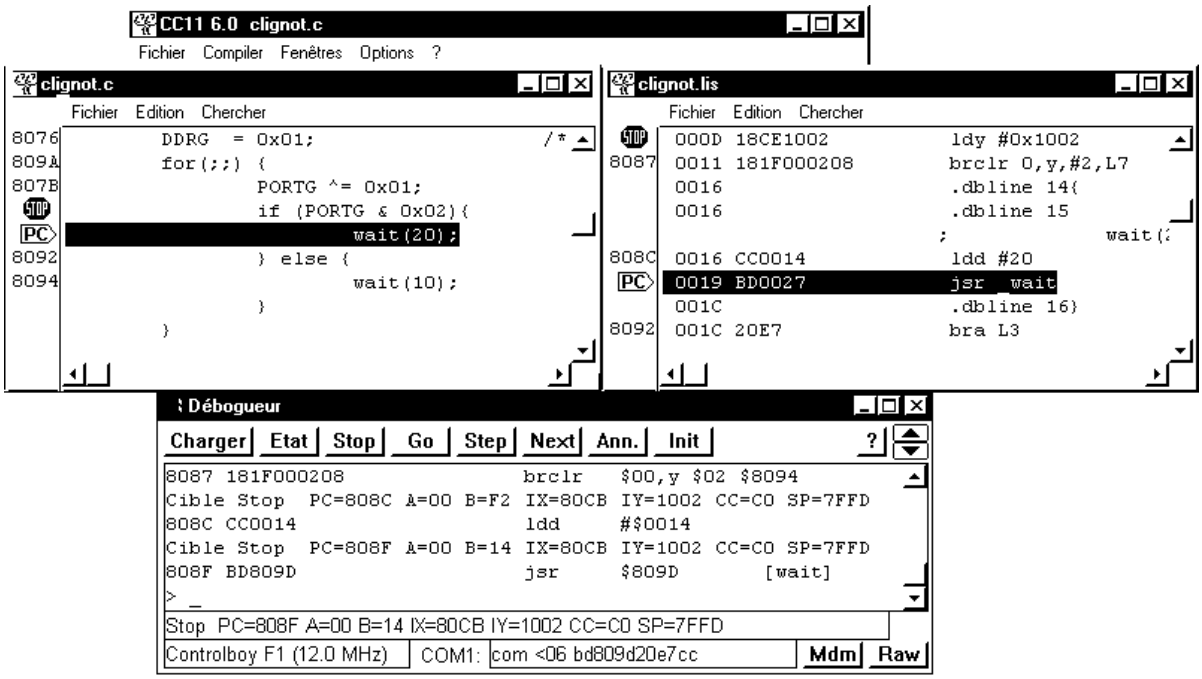
`Bin`	les programmes à exécuter
`Include`	les fichiers en-tête (fichiers #include)
`Lib`	les fichiers de la bibliothèque
`Libsrc`	les fichiers source de la bibliothèque
`Exemples`	les exemples de programme
`Talkers`	les sources de talkers pour le débogueur

Dans le répertoire BIN vous trouvez les fichiers suivants:

`Compile.exe`	Programme DOS	Exécute des programmes DOS à partir de WINDOWS
`Env.txt`	Fichier de données	Données de l'utilisateur
`Ias6811w.exe`	Programme DOS	Assembleur
`Icc11w.exe`	Programme DOS	Pilote de compilateur
`Iccom11w.exe`	Programme DOS	Compilateur C
`Icppw.exe`	Programme DOS	Préprocesseur
`Ilibw.exe`	Programme DOS	Gestionnaire de bibliothèque
`Ilinkw.exe`	Programme DOS	Editeur de liens
`Imake.exe`	Programme DOS	Make
`Ledit.dll`	WINDOWS DLL	Editeur de programme source
`Wcc11.exe`	Programme WINDOWS	Programme principale

Vous travaillez avec le programme WCC11.EXE sous Windows. Ce programme lance les programmes sous DOS pour compiler vos programmes C.

Présentation du logiciel



Cette surface permet d'écrire, de compiler, de charger et de déboguer des programmes en C. Le programme vous présente après le lancement deux fenêtres. La fenêtre en haut est la fenêtre principale. Elle vous permet de lancer un éditeur pour éditer le programme source et compiler le programme. La fenêtre en bas contient le débogueur qui vous permet de communiquer avec la carte cible à base du 68HC11. Dans le menu FICHIER vous trouvez les outils pour ouvrir et traiter le programme source principal. COMPILER lance le compilateur. Le curseur d'attente s'affiche pendant la compilation. Après on trouve la sortie des programmes de compilation.

Si le compilateur vous affiche une erreur, il suffit de cliquer deux fois sur le message d'erreur pour ouvrir une fenêtre pour éditer le fichier et positionner la source sur la ligne erronée. Le menu FENETRES affiche les fichiers source. Cliquer sur un fichier ouvre une fenêtre pour éditer ce fichier. Le dialogue OPTIONS vous permet de spécifier les options pour compiler le programme: Les fichiers sources, les options de compilation, les options de link, le fichier make.

Il est très simple de naviguer du C à l'assembleur correspondant.
Cliquer deux fois sur une ligne d'un programme C affiche les lignes en assembleur généré par le compilateur pour cette ligne en langage de haut niveau. Cliquer deux fois sur une ligne assembleur affiche la ligne C.

Dans la fenêtre du débogueur vous communiquez avec la cible. (Points d'arrêt, pas à pas, table de symboles). Si le débogueur atteint un point d'arrêt (également pour les pas à pas), il déplace la source et affiche le PC (Program Counter) dans la marge gauche. Vous voyez le programme source C et le programme assembleur créé par le compilateur. Le débogueur a deux pas à pas: L'un entre dans les sous-programmes, l'autre n'entre pas. Il débogue même un programme qui tourne: Lire et écrire la mémoire, mettre des points d'arrêt.

Utilisation du Compilateur
Exemple 1: Faire clignoter une LED

Le programme dans le répertoire **/cc11/exemples** s'appelle **clin.c**. Voilà le programme source C.

```
#include <hc11.h>

void wait(int cnt);
int x;

void
main(void)
{
      DDRG  = 0x01;                 /* PG1 = sortie */
      for(;;) {
            PORTG ^= 0x01;                /* basculer led */
            wait(20000);
            x++;
      }
}

void
wait(int cnt)
{
      for (;cnt>0; cnt--);

}
```

Cliquez sur FICHIER, OUVRIR et ouvrez le programme. Le programme bascule régulièrement la DEL L2 sur une carte Controlboy F1. Vous devrez adapter le programme si vous travaillez sur une autre carte cible. Cliquez sur OPTIONS, CLIN.MAK et vérifiez les paramètres pour la mémoire de votre cible.

-BTEXT	début de l'EEPROM de votre cible	0x8000 pour Controlboy F1
-BDATA	début de la RAM	0x2000 pour Controlboy F1
-Binit_sp	pointeur de pile, fin de la RAM	0x7FFF pour Controlboy F1

Si les paramètres sont corrects, fermez le dialogue et cliquez sur COMPILER dans la fenêtre principale. Le logiciel sous Windows va lancer le IMAKE sous DOS qui va interpréter le fichier **clin.mak** pour lancer le compilateur ICC11W pour compiler et linker le programme. Les messages de ces programmes sous DOS sont affichés dans la fenêtre principale. Si la compilation se termine sans erreur, ce vous donne le fichier objet **clin.s19**. C'est le programme en format S records de Motorola prêt à charger dans l'EEPROM de la cible. Vous pouvez visualiser ce fichier dans FENETRES, CLIN.S19.

Il faut que le débogueur dans la fenêtre en bas vous affiche une cible dans l'état STOP. Si la cible tourne, il suffit de cliquer sur STOP pour arrêter la cible. Sinon adressez-vous au chapitre Préparation du logiciel pour la carte cible. Cliquez sur CHARGER pour charger le programme dans la mémoire de la cible. Cliquez sur GO pour lancer votre programme.

La DEL doit maintenant clignoter.

Vous pouvez cliquer sur STOP pour arrêter le programme. Deux fenêtres vous affichent la source C de votre programme et le fichier listing du programme assembleur. C'est le programme créé par le compilateur. Dans chaque fenêtre vous trouvez sur la marge gauche un symbole PC qui correspond à l'endroit ou se trouve le registre d'instructions PC de votre programme. Tapez à l'invite du débogueur

```
d x
```

pour afficher la valeur de la variable.

Exemple 2: Printf sur afficheur LCD

Le deuxième exemple dans le répertoire **/cc11/exemples** s'appelle **print.c**. Le programme utilise la fonction **printf()** de la bibliothèque standard. Cette fonction est basée comme bien d'autres fonctions de la bibliothèque sur la sortie standard de C. Comme il n'y a pas de sortie standard, par exemple l'écran d'un ordinateur, il faut donner cette sortie aux fonctions de la bibliothèque. Toutes ces fonctions s'appuient effectivement sur une seule fonction: **putchar(char c)**. En réalisant cette fonction qui affiche un seul caractère pour l'afficheur cristaux-liquide LCD, on réalise la totalité de la sortie standard de C sur cet afficheur. Le programme **print.c** contient donc la fonction principale **main()** qui utilise **printf()** de la bibliothèque standard, et la fonction **putchar()** pour l'afficheur LCD.

```
// Programme sur Controlboy F1: printf sur LCD

#include "hc11.h"
#define PORTM     *(unsigned char *)(_IO_BASE + 0x62)
#define PORTN     *(unsigned char *)(_IO_BASE + 0x63)

#include <stdio.h>

void lcdinit(void);

void
main(void)
{     lcdinit();
      printf(" Controlboy F1   ");
}

int
putchar(char c)               /* library function     */
{
...
```

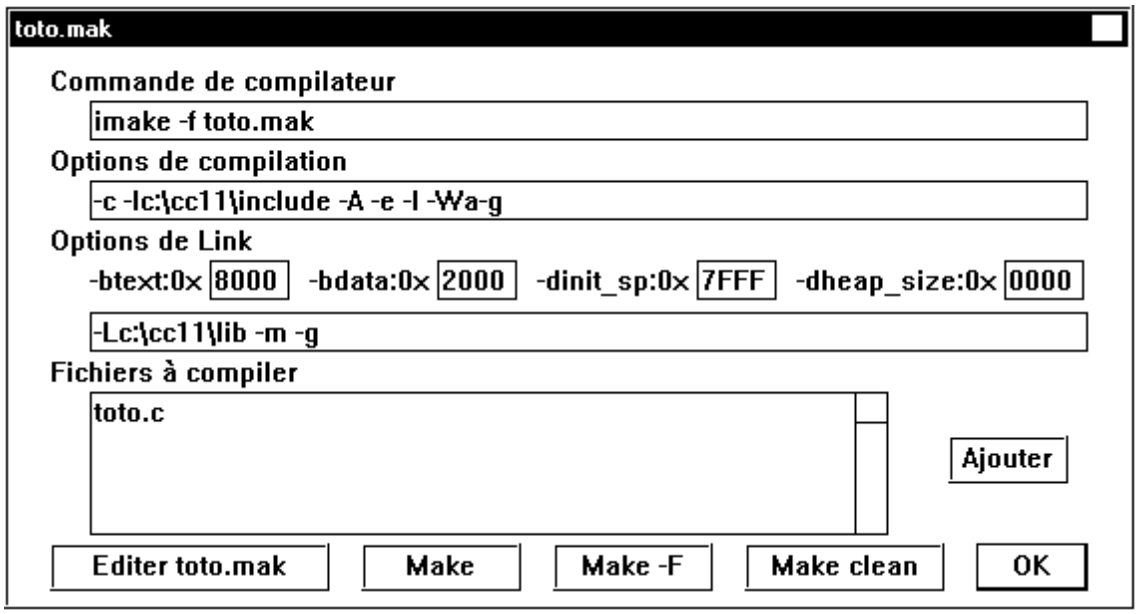
Exemple 3: Virgule flottante

Le troisième exemple dans le répertoire **/cc11/exemples** s'appelle **float.c**. Le programme calcule la racine carrée de 2 et l'affiche sur l'afficheur LCD. Notez qu'on doit ajouter **-lfp11** dans la ligne OPTIONS DE LINK dans la fenêtre FLOAT.MAK.

```
// Programme sur Controlboy F1: printf virgule flottante sur LCD

#include "hc11.h"
#define PORTM     *(unsigned char *)(_IO_BASE + 0x62)
#define PORTN     *(unsigned char *)(_IO_BASE + 0x63)

#include <stdio.h>
#include <math.h>
void lcdinit(void);

double e;

void
main(void)
{     lcdinit();
      e = sqrt(2.0);
      printf("sqrt(2)=%f", e);
}
...
```

Options du compilateur, Make

CC11 utilise l'utilitaire IMAKE pour construire le fichier exécutable à partir de vos fichiers source. Dès que vous avez créé un fichier source **`toto.c`**, le CC11 vous crée automatiquement le fichier **`toto.mak`**. Cliquez sur OPTIONS, TOTO.MAK. Vous pouvez changer les paramètres de ce fichier ou directement éditer le fichier **`toto.mak`**.



Vous pouvez cliquer sur EDITER TOTO.MAK pour visualiser et changer le fichier **`toto.mak`**. En cliquant sur MAKE vous lancez Imake pour créer le fichier exécutable. En cliquant sur MAKE -F Imake va recréer le fichier objet à partir de zéro. Toutes les cibles seront mises à jour. En cliquant sur MAKE CLEAN Imake efface tous les fichiers sauf les fichiers sources.

Personnalisation

Crt11.s, End.s

Les fichiers **crt11.s** et **end.s** se trouvent dans le répertoire **/cc11/libsrc**. Le fichier **cc11/lib/crt11.o** est inclus dans le programme par le linker tout au début. Vous devrez peut-être adapter le programme aux caractéristiques de la carte cible. **Crt11.s** contient quelques déclarations concernant les mémoires à utiliser. Le programme dans ce fichier est exécuté même avant de lancer le programme C. Il est responsable de donner un environnement propre pour que le programme C puisse tourner. Il initialise le pointeur de pile. Il charge le segment DATA depuis des données stockées dans l'EEPROM et met le segment BSS à zéro.

Après il lance le programme C **main().** Notez que le compilateur C met toujours un blanc souligné avant le nom. Il appelle donc **_main**. Quand le programme **main** est terminé ou quand le programme C appelle **exit()**, le programme s'arrête dans la boucle sans fin. Ce programme contient également quelques routines de base pour le roulement du programme.

Le fichier **end.s** est inclus dans le programme à la fin. Il contient quelques déclarations pour calculer les tailles des segments.

Vous pouvez changer le programme **crt11.s**. dans le répertoire **/cc11/libsrc**. Il y a un fichier **crt11.mak** pour compiler et installer le fichier **crt11.o** dans **/cc11/lib**.

Hc11.h

Le fichier **hc11.h** se trouve dans le répertoire **/cc11/include**. Ce fichier contient des déclarations pour les registres du 68HC11. Il faut peut-être changer ce fichier selon les caractéristiques de la carte cible. On peut inclure ce fichier dans un programme C à l'aide de

```
#include <hc11.h>
```

Le fichier contient des lignes de déclaration.

```
#define _IO_BASE  0x1000
#define PORTA     *(unsigned char volatile *)(_IO_BASE + 0x00)
```

Ces déclarations vous permettent d'écrire dans un programme les opérations suivantes.

`PORTA = 0x08;`	mettre le port à 08
`i = PORTA;`	lire le port
`PORTA |= 0x08;`	mettre le bit 3 à 1
`PORTA &= ~0x08;`	mettre le bit 3 à 0
`if (PORTA & 0x08)`	examiner le bit 3

Putchar

Un programme peut utiliser la fonction **`printf()`** de la bibliothèque standard. Cette fonction est basée comme bien d'autres fonctions de la bibliothèque sur la sortie standard de C. Comme il n'y a pas de sortie standard, par exemple l'écran d'un ordinateur, il faut donner cette sortie aux fonctions de la bibliothèque. Toutes ces fonctions s'appuient effectivement sur une seule fonction:

```
int putchar(char c)
```

En réalisant cette fonction qui affiche un seul caractère, on réalise la totalité de la sortie standard de C.

Voici l'exemple des fonctions **`putchar()`** et **`getchar()`** pour l'interface série RS232.

```
#include <hc11.h>

#define bit(x)    (1 << (x))
#define RDRF      bit(5)
#define TDRE      bit(7)

int getchar()
      {

      while ((SCSR & RDRF) == 0)
            ;
      return SCDR;
      }

void putchar(unsigned char c)
      {

      if (c == '\n')
            putchar('\r');
      while ((SCSR & TDRE) == 0)
            ;
      SCDR = c;
      }
```

Caractéristiques du compilateur

Le compilateur accepte des noms jusqu'à 32 caractères. Il met un blanc souligné avant le nom dans le programme assembleur.

Sections

Le compilateur stocke des données dans trois segments.

- Le segment TEXT contient les instructions du programme. Ce segment sera chargé dans l'EPROM ou dans l'EEPROM de la cible.
- Le segment DATA contient des données initialisées. Ce segment se trouve dans la RAM de la cible. Les données sont stockées dans un segment IDATA qui se trouve dans l'EEPROM. Avant de lancer le programme C, ces données sont copiées de l'EEPROM dans la mémoire vive.
- Le segment BSS contient les données non initialisées. Ce segment se trouve également dans la RAM de la cible. Ce segment est mis à zéro avant de lancer le programme C.

Il faut également prévoir dans la cible assez de place pour la pile du programme.

Si on utilise l'allocation dynamique de mémoire de la bibliothèque, il faut prévoir de la mémoire pour le tas.

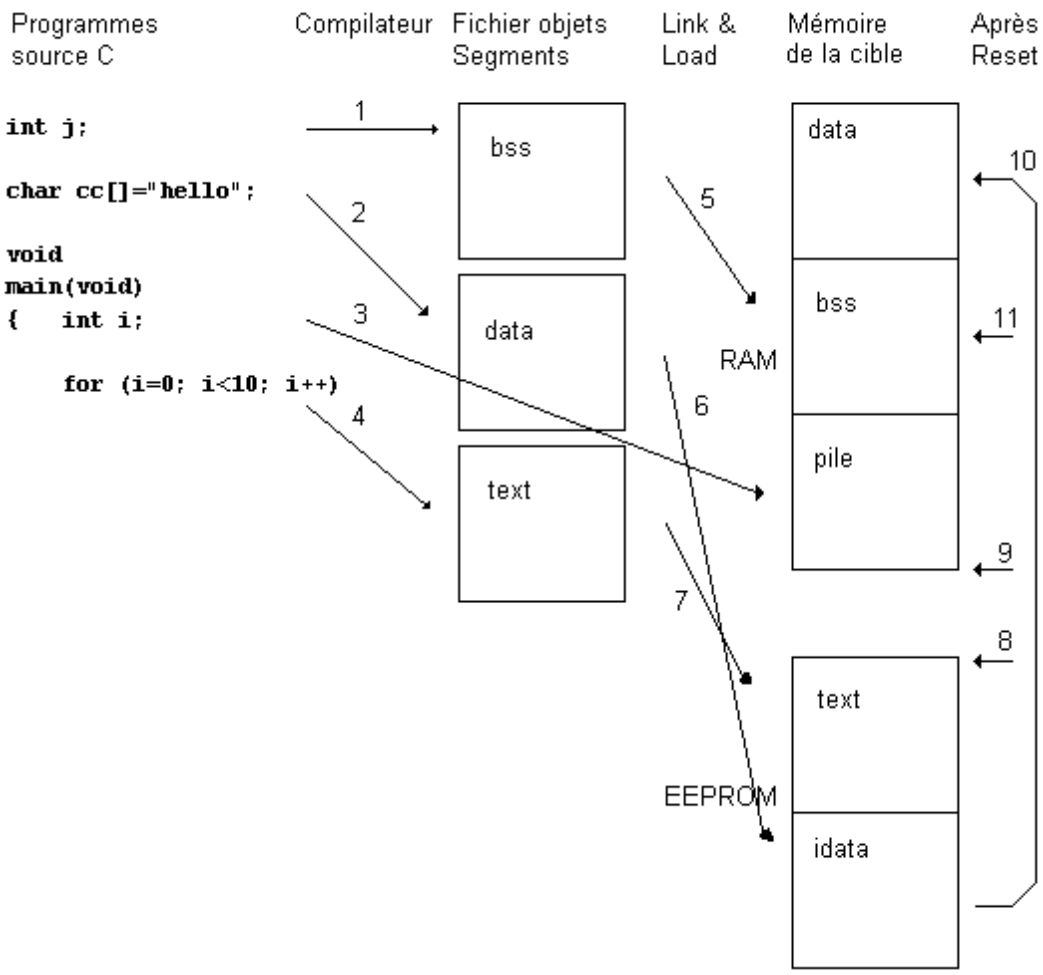
Si votre programme principal s'appelle **`toto.c`**, vous trouverez des informations sur les sections utilisées dans le fichier **`toto.mp`**.

Les variables déclarés **`const`** sont stockés dans le segment TEXT.
Voilà des exemples de déclarations de variables pour les différents segments.

`char a[100];`	La variable se trouve dans le segment BSS dans la RAM. La variable est mise à zéro avant de lancer le programme.
`char b[]="Controlboy";`	La variable se trouve dans le segment DATA dans la RAM. Les données sont stockées dans l'EEPROM et sont copiées dans la RAM avant de lancer le programme.
`const char c[]="Controlboy";`	La variable avec ses données se trouve dans le segment TEXT dans l'EEPROM. Le programme ne peut pas changer la variable.

La directive **`#pragma abs_address`** permet de charger des données à une adresse absolue.

```
#pragma abs_address:0xFFF0
void (*rtiint_vector)() = RtiInt ;
#pragma end_abs_address
```



Vous avez à gauche un programme source en C. Le compilateur attribue les données aux segments comme suivant:

1. Une variable non initialisée est attribuée au segment BSS.
2. Une variable initialisée est attribuée au segment DATA.
3. Une variable locale est attribuée à la pile. Elle ne se trouve pas dans le fichier exécutable.
4. Le programme est attribué au segment TEXT.

Le linker attribue les données à la mémoire cible comme suivant:

5. Le segment BSS est attribué à la mémoire RAM.
6. Le segment DATA a une copie IDATA, qui est attribuée à la mémoire EEPROM.
7. Le segment TEXT est attribué à la mémoire EEPROM.

Le débogueur charge votre programme dans l'EEPROM. Il ne charge rien dans la RAM.
Après le reset le talker et le fichier **`crt11.o`** assurent l'environnement de roulement du programme C:

8. Le talker saute au début de l'EEPROM.
9. **`Crt11.o`** met le SP (pointeur de pile) à la fin de la mémoire RAM.
10. **`Crt11.o`** copie la zone IDATA de l'EEPROM à la zone DATA de la RAM.
11. **`Crt11.o`** met la zone BSS dans la RAM à zéro.
12. **`Crt11.o`** exécute la routine principale **main().**

Types

Le compilateur utilise des types de base suivants.

unsigned char	8 bits
signed char	8 bits
unsigned short	16 bits
signed short	16 bits
unsigned int	16 bits
signed int	16 bits
unsigned long	32 bits
signed long	32 bits
float	32 bit, exposant 8 bits, mantisse 24 bits
double	32 bit, exposant 8 bits, mantisse 24 bits
pointeur	16 bits
void	

Le type **`char`** est égal au type **`unsigned char`**.

Le format des nombres en virgule flottante est le format de la norme IEEE. Le calcul de ces nombres utilise de la mémoire globale. Ces fonctions sont donc non réentrant.

Le fichier **`limits.h`** contient des déclarations concernant les types de base, comme prévu par la norme ANSI.

```
#define CHAR_BIT  8
#define CHAR_MAX  SCHAR_MAX
#define CHAR_MIN  SCHAR_MIN
#define INT_MAX         32767
#define INT_MIN         -32768
#define SCHAR_MAX 127
#define SCHAR_MIN -128
#define SHRT_MAX  INT_MAX
#define SHRT_MIN  INT_MIN
#define UCHAR_MAX 255
#define UINT_MAX  65535u
#define USHRT_MAX UINT_MAX
#define LONG_MAX   2147483647L
#define LONG_MIN  (-2147483647L-1)
#define ULONG_MAX 4294967295UL
```

Le fichier **`float.h`** contient des déclarations concernant les types de virgule flottante.

```
#define DBL_MAX   3.402823466e+38f
#define DBL_MIN   1.175494351e-38f
#define FLT_MAX   3.402823466e+38f
#define FLT_MIN   1.175494351e-38f
```

Routine d'interruption

On peut également écrire une routine d'interruption en utilisant la directive **#pragma** avec la syntaxe.

```
#pragma interrupt_handler <nom> [<nom>]*
```

L'exemple suivant **it.c** se trouve dans le sous-répertoire **exemples**. Il déclare une routine d'interruption pour le timer du temps réel. Il compte le temps écoulé dans les variables second et minute. On doit charger l'adresse de cette routine comme vecteur d'interruption. La directive **#pragma abs_address** permet de charger des données à une adresse absolue. Il faut initialiser le timer par exemple dans la routine main. N'oubliez pas d'autoriser des interruptions.

```
#include <hc11.h>
#include <stdio.h>
int second;
int minute;

#pragma interrupt_handler RtiInt
void RtiInt(void)
{     static int tictac = 0;
      if (++tictac >= 244){         /* 8 Mhz */
            tictac = 0;
            if (++second >= 60) {
                  second = 0;
                  minute++;
            }
      }
      TFLG2 |= 0x40;
}

void
main(void)
{
      PACTL &= 0xFC;          /* selectioner la vitesse */
      TMSK2 |= 0x40;          /* declencher le timer */
      asm(" cli ");           /* autoriser les interruptions */
      for (;;);
}

#pragma abs_address:0xFFF0
void (*rtiint_vector)() = RtiInt ;
#pragma end_abs_address
```

Si vous avez chargé et lancé le programme dans votre cible, tapez dans la fenêtre du débogueur pour vérifier la marche du temps.

```
> d minute, second
```

Interface Assembleur

Sous-programme en Assembleur

On peut écrire des sous-programmes en assembleur. Toutes les données déclarées globales sont directement accessibles. L'interface d'un sous-programme C est exactement définie. Un programme C peut appeler un programme assembleur et un programme assembleur peut appeler un programme C tout en transférant les paramètres et le résultat. Le programme assembleur doit respecter le transfert des paramètres et du résultat de C.

Les paramètres sont transformés avant de les passer au sous-programme.

Type original	Format du paramètre
Char 8 bits	Int 16 bits
Short 8 bits	Int 16 bits
Int 16 bits	Int 16 bits
Long 32 bits	Long 32 bits
Float 32 bits	Double 32 bits
Double 32 bits	Double 32 bits
Pointeur 16 bits	Pointeur 16 bits

Une structure garde sa taille.

Le premier paramètre est passé par le registre D, si c'est possible. Les autres paramètres se trouvent sur la pile. Le pointeur de pile pointe sur l'octet avant l'adresse de retour. Derrière on trouve sur la pile le deuxième paramètre et ensuite les autres successivement. Le sous-programme peut utiliser tous les registres sauf le pointeur de pile et le registre IX et retourne le résultat dans le registre D. Si le résultat dépasse 16 bits, le registre D contient un pointeur sur le résultat.
Exemple:

```
int toto(int a, int b);

void
main(void)
{     int   i;
      i = toto(3, 7);
}
```

Le sous-programme trouve le premier paramètre dans le registre D. Il sauve le registre IX. L'instruction tsx transfère le pointeur de pile dans le registre IX et augmente le registre IX de 1. IX pointe donc sur le registre IX sauvé, suivi par l'adresse de retour, et le deuxième paramètre se trouve encore deux octets plus loin. La routine retourne la somme de ces deux paramètres dans le registre D et restaure le registre IX avant le retour.

```
            .text
_toto::     pshx
            tsx
            addd  4,x   ; 2. parametre
            pulx
      rts
```

La directive ASM

La directive **asm** permet d'introduire directement des instructions assembleur dans un programme C. Par exemple l'instruction assembleur dans le programme ci-dessous autorise les interruptions.

```
TMSK2 |= 0x40;    /* déclencher le timer       */
asm(" cli ");     /* autoriser interruptions   */
```

Les lignes écrites en assembleur sont exclues d'optimisation.

Le compilateur utilise le registre IX pour adresser les variables locales et les paramètres de la fonction. Il affiche dans le programme assembleur où se trouvent ces données. Voilà le début d'un sous-programme C.

```
int
toto(int p1, int p2)
{     int i, j;
```

Ce qui donne après la compilation en assembleur.

```
; IX -> 0,x
;              j -> 2,x
;              i -> 4,x
;             p2 -> 12,x
;             p1 -> 8,x
_toto::
```

On peut également adresser les variables locales et les paramètres dans une directive asm par leurs noms: %<nom de la variable>. On peut écrire plusieurs ligne dans une seule directive. Voilà un exemple d'une directive à plusieurs lignes, qui adresse les données locales du programme.

```
asm(" ldd   %p1\n"
    " std   %i");
```

Ce qui donne en assembleur.

```
      ldd   4,x
      std   2,x
```

Si on utilise dans une directive asm le registre IX, il faut le restaurer.

```
asm(" ldx   #$1000\n"   /* utiliser IX */
      ...
      " tsx");          /* restaurer IX */
```

Linker

Le rôle du linker est de relier votre code avec des librairies pour construire un exécutable. Dans le cas de CC11, le fichier exécutable est un fichier ASCII au format S records de Motorola, contenant le programme et les données. C'est le fichier que le débogueur sait lire et charger dans la carte cible.

Sections

Un fichier exécutable comporte normalement 3 sections:

- les instructions exécutables qui sont placées dans la section code ou zone TEXT,
- les variables globales ou statiques initialisées sont dans la section variable ou zone DATA,
- les variables globales ou statiques non initialisées sont dans la zone BSS (Block Start Section).

Comme il n'y a pas d'OS ou de gestion de mémoire qui protège le système, vous pouvez placer le code et les données où vous voulez. Pendant l'exécution le programme a accès à 2 autres sections:

- la pile pour les données locales et le contrôle de l'exécution,
- le tas pour l'allocation dynamique de la mémoire.

Zone Text

La zone de texte contient le code. C'est généralement une zone de lecture seule comme dans une ROM, PROM ou EPROM. Pendant le développement du programme, on place cette zone en EEPROM ou même en RAM pour une programmation et une mise à jour facile. La puissance du linker vous permet de relocaliser votre code sans le recompiler ce qui facilite le déplacement dans le plan mémoire. Votre programme n'a besoin que d'être linker à nouveau pour s'exécuter correctement (à moins que vous n'usiez de références directes dans vos routines assembleurs).

Zone Data: Variables globales initialisées

Pour exemple:

```
int abc = 1 ;
```

déclare une variable de type entier et l'initialise à 1. Cette variable sera placée dans la section de donnée. Pour être sûre d'initialiser la section de donnée avec des valeurs correctes, les valeurs initiales sont conservées dans une zone tampon IDATA. Au début de l'exécution, la routine d'initialisation copie cette zone IDATA vers la zone DATA. IDATA est située juste après la zone de texte. Elle est donc mise en ROM pour que le contenu soit restitué après un Reset ou une coupure d'alimentation. Normalement vous n'avez pas besoin de spécifier une adresse de départ pour IDATA qui suit par défaut la zone de texte. Comme la routine d'initialisation copie cette zone vers la zone DATA, ces 2 zones doivent être contigues. Si vous utilisez des zones non contigues, vous devrez spécifier l'adresse de base pour la zone IDATA. Sinon, le linker générera un message d'erreur.

Zone BSS

Pour exemple:

```
int déf;
```

déclare une variable globale non initialisée, qui est mise à 0 en valeur initiale (règle ANSI C). Elle sera placée dans la zone BSS. La routine de départ initialise entièrement la section BSS avant d'appeler le main ( ).

Section absolue

Quelquefois, il est pratique d'adresser un objet par son adresse absolue. Par exemple, pour un vecteur d'interruption. En C vous devez utiliser l'instruction pragma comme suit:

```
#pragma abs_address:<address>
#pragma end_abs_address
```

exemple:

```
#pragma abs_address: 0xFFD6 / * vecteur d'interruption pour HC11 */
```

Abs_address spécifie que l'objet qui suit va être mis en mémoire à l'adresse spécifiée (0xFFD6). Vous pouvez utiliser plusieurs instructions abs_address si vous voulez utiliser plusieurs objets de ce type. Un end_abs_address ou une fin de fichier permet de terminer cette partie de déclaration. Pour cette déclaration, la directive d'assemblage générée est:

```
.area memory (abs)
.org <address>
```

La première ligne spécifie à l'assembleur et au linker que l'adresse est absolue et n'a pas besoin d'adresse extérieure relative.

Symboles

L'assembleur traite un nom indéfini comme une référence externe. Par exemple si le fichier contient:

```
ldd #__heap_start
```

et qu'il n'y a pas de définition de __heap_start dans le fichier, l'assembleur place ce symbole comme une référence externe dans le fichier objet. Quand le linker combinera les fichiers objets entre eux, il cherchera dans tous les fichiers la définition de ce symbole. Ce symbole devra être trouvé dans un autre fichier objet. Typiquement le symbole peut être une variable externe et un autre fichier contiendra sa définition. De plus, le linker permet au symbole d'être défini lors de l'édition de lien par une ligne de commande. Par exemple si le symbole init_sp a la valeur 0x7FFF vous pouvez vous servir de la commande:

```
icc11w -dinit_sp:0x7FFF file.o
ou
ilinkw -dinit_sp:0x7FFF file.o
```

De plus, en vous servant d'une adresse numérique vous pouvez vous servir de la valeur définie par le symbole qui a été précédemment défini dans la ligne de commande par exemple:

```
ilinkw -d__regs:0x1000 -bdata:__regs
```

donne la valeur 0x1000 à __regs et place la section DATA à cette adresse.

Plan mémoire

La syntaxe complexe pour spécifier une adresse est

```
-b<name>:<range>[:<range>]
<range> est <begin_address>[.<end_address>]
```

Par exemple, si votre espace EEPROM est de 0x0 à 0x1FFF et de 0x4000 à 0x5FFF, vous devrez écrire:

```
-btext:0x0.0x1FFF:0x4000.0x5FFF
```

Ceci permet d'employer au mieux des espaces mémoire non contigus.

Adresses par défaut

Si vous ne spécifiez pas d'adresse pour une section, le linker se servira de l'adresse de fin de la précédente section comme adresse de départ. Cet ordre est simplement défini à partir des étiquettes des sections dans le fichier objet. Aussi, un moyen simple de définir l'ordre des sections est de le définir dans le fichier de départ. Le fichier de départ par défaut spécifie que la section DATA commence juste après la section TEXT.

ASSEMBLEUR

Le compilateur génère le code assembleur qui est exploité ensuite par l'assembleur. L'assembleur génère un fichier de sortie .S19 relogeable. Vous pouvez écrire aussi des routines en assembleur et les incorporer dans votre programme C. Ce chapitre décrit le format du langage accepté par l'assembleur.

Zone relogeable

Le code assembleur est divisé en sections relogeables et sections absolues. Le linker combine les sections de même nom entre elles vers tous les modules objets. Au moment de l'édition de liens, vous devez spécifier l'adresse de départ de chaque section relogeable. Le linker ajustera les symboles de référence à leur adresse finale. Ce processus est appelé: relocation du code.

Format du code source assembleur

<nom> est une séquence d'au plus 32 caractères alphabétiques pouvant contenir les caractères spéciaux: dollars($), point(.), underscore(_). Un nom ne peut pas commencer par un chiffre.

<nombre> est une séquence de chiffres au format C: Le préfixe 0x signifie que c'est un nombre hexadécimal, le préfixe 0 signifie un nombre octal et pas de préfixe signifie que c'est un nombre décimal. De plus, le préfixe 0b signifie que c'est un nombre binaire. Vous pouvez aussi indiquer le format hexadécimal en vous servant du préfixe $.

<escape sequence> est un caractère spécial comme \n, \t, etc. En plus \0xxx entre un code octal.

<string> est une chaîne de caractères entre guillemet (""). L'utilisation des guillemets dans la chaîne de caractères doit être précédée par le caractère backslash (\) comme distinction avec les guillemets de fin de chaîne.

<expression> est une expression générale. C'est-à-dire qu'elle peut être à la fois:

- un terme c'est-à-dire un nombre, un nom, etc...
- une expression entre ' (' and ')'
- deux expressions jointes avec un opérateur binaire. Ces opérateurs binaires sont les mêmes qu'en C: << >> + - * / % & | ^
- un opérateur unaire appliqué à une expression
  - > l'octet de poids fort d'une expression
  - < l'octet de poids faible d'une expression
  - 'x caractère x
  - "ab deux caractères a et b
  - les opérateurs de C: - + ~

Tous les opérateurs ont la même priorité, vous devrez insérer des parenthèses pour grouper les expressions par priorités.

Référence de la page zéro

Quelques instructions prennent comme opérande l'adresse de page nulle ceci correspond à l'adresse de l'opérande dans les 256 premiers octets de la mémoire. Vous ne pouvez pas spécifier une variable de la page nulle en C, mais vous le pouvez en assembleur en faisant précéder la variable ou l'adresse par le caractère *:

```
bset    *_foo,#0x23
bclr    *0x20,#0x32
```

Vous devez être sûr qu'une variable de ce type est définie dans les 256 premiers octets autrement vous aurez un message d'erreur au moment de l'édition de liens. Vous pouvez aussi mettre quelques variables dans la zone absolue.

```
.area memory(abs)
.org 0
foo::byte 1
```

Format

Un fichier assembleur contient des lignes de texte assembleur dans le format suivant. (Les lignes plus grandes que 128 caractères seront tronquées.)

```
[ label: ] operation [ operand ]
```

Un commentaire peut être introduit n'importe où dans la ligne précédé par le caractère '**;**'. Tous les caractères qui suivront ce caractère de commentaire seront alors ignorés. Un label défini un nom de symbole relogeable. Sa valeur correspond à l'adresse de l'endroit où le label apparaît dans le fichier final. De nombreux labels peuvent exister. Ils doivent être suivis par **:** ou **::**. **::** signifie que le label est un symbole global qui peut être référencé dans un autre module objet. Une opération est à la fois une directive assembleur ou un opcode HC11.

Directives assembleur

Les directives sont des opérations qui ne génèrent pas de code mais affectent l'assemblage. L'assembleur accepte les directives suivantes:

.text spécifie que les instructions ou les données sont ajoutées dans la zone texte

.data spécifie que les instructions ou les données sont ajoutées dans la zone data.

Si vous créez une donnée dans la zone DATA, vous devrez définir la même valeur dans la zone IDATA précédente. Au début du programme, la section IDATA est copiée dans la section DATA et la taille des deux sections doivent être identique. On doit réserver l'espace dans la zone DATA et définir les valeurs dans la zone IDATA correspondante. Par exemple:

```
.data
_mystuff::
.lbkb 5
.area idata
.byte 1,2,3,4,5; _mystuff donne ceci au démarrage.
```

.area <name> spécifie que la donnée ou l'instruction suivante est ajoutée à la section name. .text est un synonyme pour .area text et .data est un synonyme pour .area data. Le compilateur utilise uniquement les sections texte (TEXT), donnée (DATA), et BSS (BSS).
<name> peut être suivi par ''(abs),'' signifiant que cette zone est une section absolue et qu'elle peut contenir la directive .org.

.org <exp> change le compteur du programme par l'adresse spécifiée. Cette directive est valide seulement dans une section absolue.

.byte <exp>[,<exp>]* (ou .db...) alloue des octets de 8 bits et les initialise avec leurs valeurs spécifiées. Par exemple .byte 1,2,3 alloue 3 octets avec la valeur 1,2, et 3.

.word <exp>[,<exp>]* (ou .dw...) alloue des mots de 16 bits et les initialise avec les valeurs spécifiées. Par exemple .word 1,2,3 alloue 3 mots avec la valeur 1,2, et 3.

.blkb <nombre> réserve nombre octets sans initialiser leur contenu.

.blkw <nombre> réserve des mots sans initialiser leur contenu.

.ascii <string> alloue un bloc de la même largeur de string et l'initialise avec string.

.asciz <string> alloue un bloc de la même largeur que string plus 1, et initialise avec la valeur de string suivie par le caractère nulle de fin de chaîne.

.even force le compteur du programme à être pair.

.odd force le compteur du programme à être impair.

.globl <name>[,<name>]* déclare nom comme un symbole global et peut être référencé en dehors de ce module. Ceci a le même effet que de définir le label suivi de **::**

.if <exp>, **.else**, et **.endif** implémentent des directives conditionnelles d'assemblage. Si <exp> n'est pas nulle, alors la directive spécifiée est exécutée et .else ou .endif est ignoré. Si <exp> est nulle, ce sont les directives dans .else ou .endif qui sont exécutées.

.include <string> ouvre le fichier nommé string.

<name> = <exp> donne la valeur de l'expression au nom.

Librairie standard et fichiers d'en-tête

Le compilateur est livré avec la librairie C standard. La plupart des fonctions est conforme au format C standard. Il y a des exceptions comme printf () qui est limité pour réduire la taille de la fonction.

Vous trouvez dans le répertoire **`/cc11/lib`** trois librairies:

`Libc11.a`	La librairie de routines standard. Le printf() est limité et n'affiche ni des variables de type long ni ceux de type float. Le link inclut cette librairie automatiquement.
`Liblng11.a`	Contient un printf() qui affiche les variables de type long. Pour inclure cette fonction, ajoutez dans l'option de link **`-llng11`** .
`Libfp11.a`	Contient un printf() qui affiche les variables de type long et ceux de type float. Pour inclure cette fonction, ajoutez dans l'option de link **`-lfp11`** .

Vous trouvez dans le répertoire **`/cc11/libsrc`**

- Les sources de tous les fichiers de la librairie standard.
- Un fichier makefile **`libc.mak`** pour compiler tous les fichiers de la librairie et installer les librairies dans **`/cc11/lib`**
- Les sources de **`crt11.s`** et **`end11.s`**
- Un fichier **`crt11.mak`** pour compiler ces fichiers et installer leurs fichiers objet.

Assert.h: Assert

La macro assert (test) termine l'exécution du programme si le test de condition est évalué à 0.

Ctype.h: Classification de caractère

int isalnum (int c) retourne non nul si C est un digit ou alphabétique.

int islpha (int c) retourne non nul si C est alphabétique.

int iscntrl (int c) retourne non nul si C est un caractère de contrôle (en général, LF, FF, BELL...etc).

int isdigit (int c) retourne non nul si C est un digit.

int isgraph (int c) retourne non nul si C est un caractère imprimable et non un espace.

int islower(int c) retourne non nul si C est un caractère minuscule.

int isprint(int c) retourne non nul si C est un caractère imprimable.

int ispunct(int c) retourne non nul si C est caractère imprimable sans être le caractère espace ou un digit ou an alphabétique.

int isspace(int c) retourne non nul si C est le caractère espace ou un caractère du type CR, FF, HT, NL, and VT.

int isupper(int c) retourne non nul si C est an caractère majuscule.

int isxdigit(int c) retourne non nul si C est un digit hexadécimal.

int tolower(int c) retourne le caractère C en minuscule.

int toupper(int c) retourne le caractère C en majuscule

Float.h: Caractéristiques de nombres en virgule flottante

Ce fichier défini les propriétés des nombres flottants.

Math.h: Fonctions virgule flottante

Ce fichier déclare les fonctions spécifiques au type flottant.

double exp(double x) retourne l'exponentielle x.

double fabs(double x) retourne la valeur absolue de x.

double fmod(double x, double y) retourne le reste de la division x / y.

double log(double x) retourne le logarithme népérien de x.

double log10(double x) retourne le logarithme décimal de x.

double pow(double x, double y) retourne x élevée à la puissance y.

double sqrt(double x) retourne la racine carrée de x.

double sin(double x) retourne le sinus de x exprimé en radians.

double cos(double x) retourne le cosinus de x exprimé en radians.

double tan(double x) retourne la tangente de x exprimé en radians.

double asin(double x) retourne l'arc sinus de x exprimé en radians.

double acos(double x) retourne l'arc cosinus de x exprimé en radians.

double atan(double x) retourne l'arc tangente de x exprimé en radians.

Setjmp.h: Setjmp

Ce fichier défini le type jmp_buf.

int setjmp(jmp_buf buffer) retourne 0 lors du premier appel, et retourne une valeur non nulle quand il retourne par un longjmp()

void longjmp(jmp_buf buffer, int retval) retourne au point defiéné par setjmp()

Stdargs.h: Arguments variables

stdarg.h permet d'accéder aux variables de type varargs comme dans le printf(char *fmt, …)

va_start(va_list foo, <last-arg>) initialise une variable foo.

va_arg(va_list foo, <promoted type>) accède à l'argument suivant.

va_end(va_list foo) finit l'accès aux variables.

Par exemple printf() peut être utilisé en se servant de vfprintf() comme suit:

```
#include <stdarg.h>
int printf(char *fmt, ...)
{
va_list ap;
```

```
va_start(ap, fmt);
vfprintf(fmt, ap);
va_end(ap);
}
```

Stdio.h: Entrée standard, Sortie standard

L'entrée standard et la sortie standard ne sont pas adaptées pour un microcontrôleur embarqué, aussi beaucoup de ces propriétés ne sont pas applicables ici. Néanmoins quelques fonctions sont supportées. Vous devrez assurer les fonctions getchar() et putchar() pour que ces fonctions marchent.

int getchar() retourne un caractère de la sortie standard par exemple d'un clavier. C'est votre programme principal qui doit inclure cette fonction.

int printf(char *fmt, ..) écrit une chaîne formatée en accord avec les formateurs dans la chaîne fmt. Les formateurs sont issus du standard C:

- %d – écrit l'argument qui suit comme un entier décimal
- %o - écrit l'argument qui suit comme un entier octal non signé
- %x - écrit l'argument qui suit comme un entier hexadécimal non signé
- %u - écrit l'argument qui suit comme un entier décimal non signé
- %s - écrit l'argument qui suit comme une chaîne de caractère C terminé par le caractère nul.
- %c - écrit l'argument qui suit comme un caractère ASCII
- %f - écrit l'argument qui suit comme un nombre à virgule flottante (Vous devrez inclure la librairie libfp.a pour vous servir de ce formateur)
- Si vous ajoutez la lettre 'l' entre % et un des formateurs du type entier, alors l'argument sera interprété comme un long à la place d'un int. (Vous devrez inclure la librairie liblng.a pour vous servir de ce formateur)

int putchar(int c) écrit un seul caractère sur la sortie standard par exemple sur un afficheur LCD ou sur le port SCI. C'est votre programme principal qui doit inclure cette fonction.

int puts(char *s) écrit un type string suivi par NL. Cela utilise la fonction putchar().

int sprintf(char *buf, char *fmt) écrit un texte formaté dans buf en utilisant les formateurs définis dans fmt. Les formateurs sont identiques à ceux employés dans la fonction printf().

De plus, pour faciliter l'exportation du programme vers un autre environnement, stdout et stderr sont définis comme 0, et FILE est typedefed comme void, et fprintf(FILE *, char *fmt, ...) est identique a la fonction printf(char *fmt, ..).

Stdlib.h: Fonctions Standard Library

int abs(int i) retourne la valeur absolue de i.

int atoi(char *s) converti la chaîne de caractère en un entier ou retourne 0 si une erreur est détectée.

double atof(const char *s) converti la chaîne de caractère en un double.

long atol(char *s) converti la chaîne de caractère en un long ou retourne 0 si une erreur survient.

void *calloc(size_t nelem, size_t size) retourne un pointeur sur un block mémoire de largeur contenant nelem objets, chacun de taille "size." Cette mémoire est initialisé à zéro. Cette mémoire est allouée sur le tas (i.e., vous devez appeler la fonction _NewHeap() avant). Cette fonction retourne 0 si elle ne peut allouer cette mémoire.

void exit(status) met fin au programme. La valeur de sortie est écrite dans le registre D.

void free(void *ptr) libère la mémoire allouée sur le tas.

void *malloc(size_t size) alloue un block mémoire de taille “size” sur le tas. Elle retourne 0 si la mémoire ne peut être alloué.

int rand(void) retourne un nombre pseudo-aléatoire compris entre 0 et RAND_MAX.

void *realloc(void *ptr, size_t size) réalloue un block mémoire précédemment alloué mais de nouvelle taille.

Void srand(unsigned seed) initialise la valeur seed pour rand().

long strtol(char *s, char **endptr, int base) convertis le caractère s en un type long int en accord avec la base choisie. Si base vaut 0, alors strtol choisi sa base en fonction du premier caractère de s (en tenant compte d’un éventuel signe moins (-)): 0x ou 0X indique un entier hexadécimal, 0 indique un entier octal, un entier décimal sera choisi sinon. Si endptr n’est pas NULL, alors *endptr sera définie en fin de conversion de s.

unsigned long strtoul(char *s, char **endptr, int base) est identique à la fonction strtol(), sauf que l’entier converti est de type unsigned long et que la valeur retourné est aussi de type unsigned long.

String.h: Fonctions sur caractères

void *memchr(void *s, int c, size_t n) cherche la première occurrence de c dans le tableau s de taille n. Elle retourne l’adresse de l’élément recherché ou le pointeur NULL si aucune occurrence n’est trouvée.

int memcmp(void *s1, void *s2, size_t n) compare deux tableau, chacun de taille n. Cette fonction retourne 0 si les tableaux sont égaux et un nombre positif si le premier élément différent de s1 est plus grand que celui de s2; dans le cas contraire elle retourne un entier négatif.

void *memmove(void *s1, void *s2, size_t n) copie s2 dans s1, chacun de taille n. Cette routine marche correctement même si s1 est plus grand que s2. Elle retourne s1.

void *memset(void *s, int c, size_t n) place c dans tous les éléments du tableau s de taille n. Elle retourne s.

char *strcat(char *s1, char *s2) concatène s2 à la suite de s1. Elle retourne s1.

char *strchr(char *s, int c) cherche la première occurrence de c dans s, incluant le caractère nulle de fin de chaîne. Elle retourne l’adresse de l’élément trouvé ou le pointeur NULL si aucune occurrence n’est trouvée.

int strcmp(char *s1, char *s2) compare deux chaînes de caractère. Elle retourne 0 si les deux chaînes sont identiques, et un entier positif si le premier élément différent de s1 est plus grand que celui correspondant de s2. Dans le cas contraire elle retourne un entier négatif.

char *strcpy(char *s1, char *s2) copie s2 dans s1. Elle retourne s1.

size_t strlen(char *s) retourne la longueur de la chaîne s.

char *strncat(char *s1, char *s2, size_t n) concatène jusqu’à n éléments, sans inclure le caractère nulle de fin de chaîne, de s2 vers s1. Elle replace le caractère de fin de chaîne à la suite de la nouvelle chaîne s1. Elle retourne s1.

int strncmp(char *s1, char *s2, size_t n) est identique à la fonction strcmp() sauf qu’elle compare au plus n caractères.

char *strncpy(char *s1, char *s2, size_t n) est identique à la fonction strcpy() sauf qu’elle copie au plus n caractères.

char *strncpy(char *s1, char *s2, size_t n) est identique à la fonction strcpy() sauf qu'elle copie au plus n caractères.

char *strpbrk(char *s1, char *s2) effectue la même recherche que strcspn() sauf qu'elle retourne un pointeur sur l'élément de s1 et NUL s'il n'y a pas d'occurrence.

char *strrchr(char *s, int c) cherche la dernière occurrence de c dans s et retourne un pointeur dessus. Elle retourne un pointeur NUL si aucune occurrence n'apparaît.

Char *strstr(char *s1, char *s2) cherche la première occurrence de la chaîne s2 dans s. Elle retourne l'adresse de l'élément trouvé ou le pointeur NUL si aucune occurrence n'est trouvée.

10.1 Bibliothèque partagée

A quoi ça sert?

Regardez l'exemple suivant:

```
void lcdinit(void);
void
main(void)
{     lcdinit();
      printf("   10/3= %f", 10.0 / 3.0);
}
```

Ce programme ne contient que 100 octets. Mais avec le printf, le calcul en virgule flottante, le driver pour le LCD, il contient finalement plus de 7000 octets. Une bibliothèque partagée contient les routines les plus courantes. La bibliothèque est chargée dans un endroit fixe dans l'EEPROM de la carte cible. L'application ne contient plus ces fonctions et fait appel aux fonctions de la bibliothèque partagée. Le temps de chargement du programme est nettement réduit.

Créer une bibliothèque partagée

Le fichier **`/cc11/libshare/libshare.txt`** contient une liste de fonctions qui seront intégrées dans la bibliothèque. Cette liste avec le fichier makefile crée une bibliothèque partagée pour la carte Controlboy F1. La bibliothèque occupe l'EEPROM de 0xD000 à 0xFC00 et la RAM de 0x200 à 0x220. Elle contient les fonctions de calcul pour les variables de type long et pour les variables en virgule flottante, un printf, et le driver pour l'afficheur LCD. Le fichier makefile crée le fichier **`LIBSHARE.S19`**. Il faut charger ce fichier dans l'EEPROM de la carte cible. Le fichier makefile crée également un fichier **`LIBSHARE.A`** qui sera installé dans le répertoire ../LIB. L'application doit ajouter **`-lshare`** dans l'option du link pour profiter de la bibliothèque partagée.

Pour utiliser des sections data et bss dans une bibliothèque partagée, il y a trois solutions:

- La bibliothèque n'utilise pas ces sections.
- La bibliothèque n'utilise que le bss et le bss n'est pas mis à zéro avant de lancer l'application. L'application ne doit pas utiliser la RAM de la bibliothèque partagée. C'est la solution dans cet exemple pour le Controlboy F1.
- La bibliothèque utilise ces sections. Il faut adapter le ctr11.s pour initialiser ces sections correctement.

Make

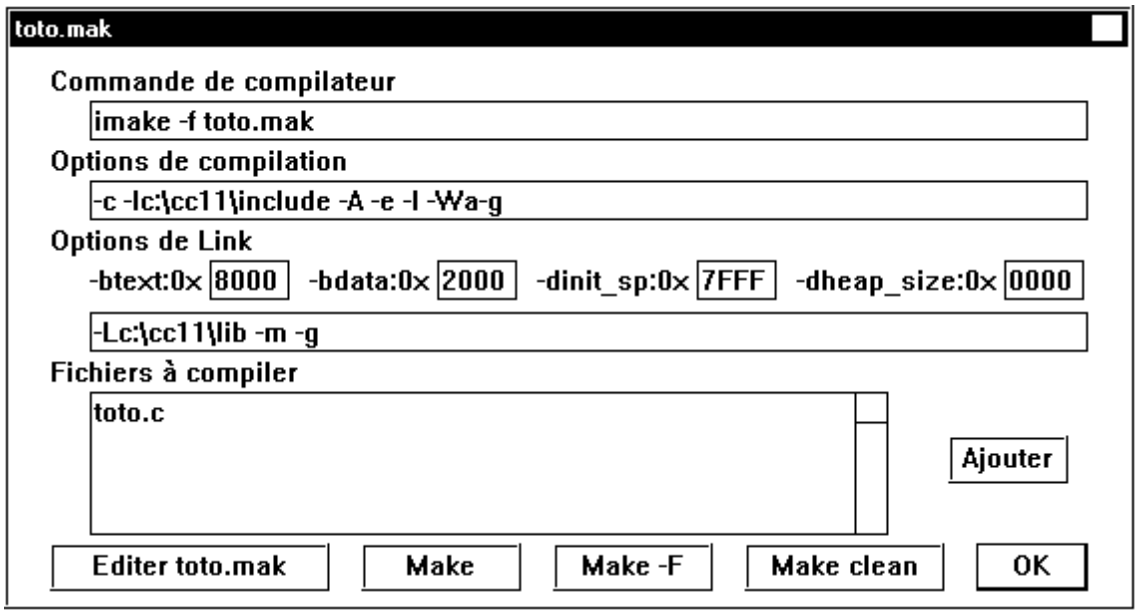
Un programme typique contient des fichiers objets issus de multiples fichiers sources. Il est fastidieux et source d'erreurs de compiler et d'éditer les liens manuellement entre les fichiers et tout particulièrement de se souvenir des fichiers qui ont changé. Aussi, si un fichier d'en-tête change, vous devrez recompiler tous les fichiers sources qui incluent ce fichier d'en-tête. En vous servant de Imake et de makefile, vous laisserez au programme l'exécution de ce travail. Imake est un utilitaire qui manage les dépendances entre les fichiers. Vous créez un fichier descriptif de toutes les dépendances entre les fichiers du programme et vous appellerez imake pour compiler les fichiers qui ont changé ou pour compiler les fichiers qui contiennent un fichier d'en-tête qui a changé depuis la dernière fois que le programme a été construit. Le logiciel CC11 crée automatiquement le fichier **`toto.mak`** pour votre fichier **`toto.c`**. Vous pouvez changer les paramètres de ce fichier ou directement éditer le fichier toto.mak. Ce chapitre s'adresse à ceux qui veulent écrire leur propre makefile.

Imake lit un fichier en entrée contenant une liste de dépendances entre les fichiers et les règles associées pour maintenir ces dépendances. Le format est généralement un nom de fichier cible suivi par une liste de fichiers dont il dépend, suivi par un jeu de commandes qui sera utilisé pour recréer le fichier cible avec ces fichiers associés.

Chaque fichier dépendant est lui-même en général un fichier cible d'un groupe de fichiers dépendants et cela de façon récursive. Si après une opération sur toutes les dépendances, un fichier cible est trouvé manquant ou est plus vieux que tous les autres fichiers dépendants, imake se sert du jeu de commandes ou par défaut de règles implicites pour reconstruire la cible. Si aucune cible n'est spécifiée dans la ligne de commande, imake se sert de la première cible définie dans le makefile.

Exemple d'utilisation de Imake

Dès que vous avez créé un fichier source **`toto.c`**, le CC11 vous crée le fichier **`toto.mak`**. Cliquez sur OPTIONS, TOTO.MAK pour visualiser et changer les paramètres principaux.



Vous pouvez cliquer sur EDITER TOTO.MAK pour visualiser et changer le fichier **`toto.mak`**. En cliquant sur MAKE vous lancez Imake pour créer le fichier exécutable. En cliquant sur MAKE -F Imake va recréer le fichier objet à partir de zéro. Toutes les cibles seront mises à jour. En cliquant sur MAKE CLEAN Imake efface tous les fichiers sauf les fichiers sources. Vous pouvez vous servir du fichier **`toto.mak`** comme point de départ pour écrire votre propre makefile.

```
CFLAGS= -c -Ic:\cc11\include -A -e -l -g -Wa-g
LFLAGS= -Lc:\cc11\lib -m -g -btext:0x8000 -bdata:0x2000 -
dinit_sp:0x7FFF -dheap_size:0x0000
TARGET= toto
SRCFILES= toto.c
OBJFILES= toto.o
.SUFFIXES:  .c .o .s .s19
.RESPONSE:
      icc11w
$(TARGET).s19: $(OBJFILES)
      icc11w -o $(TARGET) $(LFLAGS) $(OBJFILES)
.c.o:
        icc11w $(CFLAGS)  $<
.s.o:
        icc11w $(CFLAGS)  $<
```

Le reste de ce chapitre décrit en détail le fonctionnement de imake.

Utiliser le fichier de description: Makefile

Quand plus d'un argument -f <filename> apparaît, imake concatène tous ces fichiers dans l'ordre d'apparition.

Règle dérivée:

Si une cible n'a pas d'entrée makefile ou si cette entrée n'a pas de règle, imake tente de dériver la règle suivant ces méthodes:

- règle implicite, lit à partir d'un makefile spécifié,
- règle standard, lit à partir d'un fichier par défaut (default.mk),
- règle par la cible spéciale .DEFAULT:

S'il n'y a pas d'entrée dans le makefile pour la cible et pas de règle qui puisse être dérivée pour la construire, et s'il n'y a pas de fichier présent, imake renvoie un message d'erreur et s'arrête.

La procédure de départ pour imake est de lire les définitions d'environnement du MAKEFLAGS et d'enregistrer toutes les options. Alors, il lit la ligne de commande pour la liste d'options, et après cela il lit le makefile par défaut qui contient les macros de définition prédéfinies et les cibles d'entrée pour les règles implicites. Imake se sert des fichiers **`default.mk`** du répertoire courant ou le cherche dans les répertoires par défaut. Finalement, imake lit toutes les macros de définition de la ligne de commande. Ceci réécrit les définitions des macros dans le makefile.

Composition du Makefile

Le makefile peut contenir un mélange de ligne de commentaires, de définition de macro, de lignes d'inclusion (include) et des lignes de cible. Une ligne peut continuer sur une autre ligne en mettant un caractère \ en fin de ligne.

Une ligne de commentaire est une ligne commençant par un #.

Une ligne d'inclusion est utilisée pour inclure le texte d'un autre makefile. Les 7 premières lettres de la ligne sont le mot « include » suivi d'un espace. Le mot qui suit est traduit comme un nom de fichier.

Macro

Une ligne de définition de macro a la forme «WORD=text... ». Le mot à gauche du signe égal est le nom de la macro et celui à droite est la valeur de la macro. Un espace entre le premier mot et le signe égal sera ignoré. Les macros sont référencées avec un $. Les caractères suivants, ou un mot en parenthèse (), ou un mot en accolades {} sont interprétés comme la référence de la macro. Imake étend la référence, en la remplaçant par la valeur de la macro. Si une macro contient une autre macro,

la première est évaluée en premier. A noter qu'il peut y avoir une évaluation infinie si une macro se référence elle-même.

Macro pré-definie

La macro MAKE est spéciale. Elle a la valeur imake par défaut. Il y a des macros qui sont utilisées comme des abréviations dans les règles.

.$* réfère au nom de base de la cible courante pour une utilisation avec une règle implicite
.$< réfère aux noms des fichiers dépendants pour une utilisation avec une règle implicite
.$@ réfère au nom de la cible courante

Parce que imake assigne $< et $* comme pour des règles implicites, cela peut devenir incorrect quand vous vous servez d'une cible explicite.

Une ligne de la forme «WORD += text... » est utilisé pour attendre le texte à la fin de la macro. La chaîne "+=" doit être précédée et suivie d'un caractère d'espace.

Règles de Cible

Une entrée dans le makefile a le format suivant:

```
cible:  dépendances
        commandes
```

La première ligne contient le nom de la cible ou la liste de cibles séparée par un espace ; ceci peut être suivi par une dépendance ou par une liste de dépendances que imake parcours dans l'ordre. La ligne suivante commence par un espace ou une tabulation et contient les commandes pour construire la cible.

Si une cible est appelée dans plus d'une entrée comme nom de cible, les dépendances et les règles sont ajoutées pour former une liste complète de cibles et de règles.

Pour reconstruire une cible, imake appelle les macros et passe chaque ligne de commande à l'exécution comme une commande de DOS.

Cibles spéciales

Quand vous vous servez du makefile, les noms de cible suivants exécutent certaines fonctions.

.DEFAULT: La règle pour cette cible est d'utiliser le processus quand il n'y a pas d'autre entrée pour lui et sans règle pour le construire. Imake ignore toutes les dépendances de cette cible.
.DONE: Imake se sert des cibles et des dépendances après que toutes les autres soient construites.
.IGNORE: Imake ignore les codes d'erreur non nul retournés par les commandes.
.INIT: Cette cible et ses dépendances sont construites avant toutes les autres
.SILENT: Imake n'affiche pas la ligne de commande à exécuter.
.SUFFIXES: Ceci affiche la liste de suffixes pour sélectionner des règles implicites.
.RESPONSE: Ceci indique que la commande suivante doit être prise dans un fichier si la commande devient trop longue. Par exemple

```
.RESPONSE:
        icc11w
```

dit que si cette ligne de commande pour icc11w est trop longue, alors imake pourra mettre cette ligne de commande dans un fichier temporaire et se servir de l'option @ avec cette commande.

Règles

Quand les règles s'exécutent, le premier caractère implique une exécution spéciale. Les lignes commençant par les caractères spéciaux suivants sont interprétées comme suivant ;

-	Imake ignore tous les codes d'erreur non nul, normalement, imake se termine quand une commande retourne non nulle à moins que l'option -i ou la cible .IGNORE est utilisée.
@	Imake n'affiche pas la ligne de commande avant la fin de son exécution. Normalement, chaque ligne est affichée avant d'être exécutée à moins que l'option -s ou la cible .SILENT est utilisée.

Règles implicites

Un nom de fichier cible est fabriqué à partir du nom de base et son suffixe. Quand une cible n'a pas de règle imake cherche une règle implicite à partir de la liste des éléments des suffixes. Une règle implicite se présent comme suivant:

```
.Ds.Ts:
        regle
```

Ts est le suffixe de la cible et Ds le suffixe de la dépendance. La liste des suffixes reconnus est donnée dans la liste de dépendances dans .SUFFIXES.

Ligne de commande du compilateur et des utilitaires

ICC11W Pilote de compilateur

Format

icc11w [options] file1 file2...

Vous pouvez agir sur le pilote du compilateur et vous n'aurez pas besoin de vous servir d'autres outils individuels pour compiler un programme. Le pilote du compilateur prend vos fichiers d'entrée et les compile en suivant vos options spécifiées ou les options par défaut. Quelques options sont passées au compilateur directement comme l'option -D pour définir le nom de la macro. Si un code d'échec est retourné, les drivers du compilateur s'arrêtent sur le programme qui a généré l'erreur. Vous pouvez demander à la compilation de s'arrêter après un certain nombre d'exécution. Par exemple -S signifie au compilateur d'assembler seulement. Les options inconnues et les types de fichier sont passés directement au linker.

-A	Signaler les déclarations de fonction sans prototype et d'autres déclarations non conformes strictement à la norme ANSI C.
-c	crée un fichier objet seulement, sans le linker.
-D<name>[<=def>]	défini un nom de macro. Si aucune des définitions n'est donnée, la valeur par défaut sera 1.
-E	Lance seulement le préprocesseur pour les fichiers C. Ne compile pas, n'assemble pas et ne linke pas. Les fichiers générés ont le même nom que ceux en C, mais avec l'extension .i
-e	le préprocesseur accepte des commentaires du style C++.
-I	passe des informations de ligne de la source C dans le fichier assembleur pour le débogueur
-l<f>	linke dans les fichiers de librairie <f>.a. Par exemple: utiliser -lfp11 pour inclure la librairie des nombres flottants avec la fonction printf
-L<dir>	spécifie le répertoire dans lequel il faut chercher les librairies, crt11.o et end11.o
-o<file>	donne un nom au fichier exécutable. Le fichier aura l'extension par défaut .s19
-R	n'inclut pas les fichiers de départ crt11.o et end11.o
-s	Silencieux. Par défaut, si vous spécifiez plus d'un fichier d'entrée, le pilote affiche chaque fichier et son travail en cours.
-S	crée seulement un fichier assembleur sans le linker.
-U<name>	Annuler la définition d'une macro de préprocesseur.
-v	Verbose. Affiche les lignes de commande pour appeler toutes les étapes et affiche les informations de version. Si vous spécifiez -v plus d'une fois, alors le pilote affiche les commandes, mais ne les exécute pas.
-w	Annule le diagnostic d'avertissement comme des variables non référencées.
-W<pass><arg>	passe un argument <arg> à la commande <pass>. <pass> peut être p pour préprocesseur (icppw), f pour le compilateur (iccomm11w), a pour l'assembleur (ias6811w) ou l pour le linker (ilinkw).

ICCPW Préprocesseur

Format

icppw [options] <input file> [<output file>]

Le préprocesseur lit les fichiers d'entrée et exécute les directives dans le fichier. Les fichiers d'entrée ont couramment une extension .c et les fichiers de sortie, une extension .i. Si le fichier de sortie n'est pas spécifié, la sortie est la sortie standard.

-11	Spécifie une cible HC11: Utilisez les variables d'environnement spécifique icc11w et ses fichiers. Si ceci est spécifié, ce doit être le premier argument de ICPP ; par défaut, les variables d'environnement ICC11 sont utilisées.
-D<name>[<=def>]	Définie un nom de macro. Si aucune définition n'est donnée, alors la valeur 1 est appliquée.
-E	Ignore les erreurs sauf quand un fichier ne peut être écrit.
-e	Accepte des commentaires du style C++.
-I<path>	Répertoire pour les fichiers d'en-tête. Le préprocesseur se sert des chemins d'inclusion pour chercher des fichiers. Vous pouvez vous servir de plusieurs options -I. De plus le préprocesseur chercher dans les répertoires spécifiés dans l'environnement.
-U<name>	Annuler la définition d'une macro

ICCOM11W Générateur de code

Format

iccom11w [options] <input file> [<output file>]

C'est le cœur du compilateur ; il prend en entrée un fichier C préparé et génère en sortie un fichier assembleur. Le compilateur accepte le standard ANSI C mais pas l'ancien style K&R. Généralement, le fichier d'entrée a une extension .i et le fichier de sortie .s. Si vous ne spécifiez pas de fichier de sortie alors celui-ci est écrit sur la sortie standard.

Les options acceptées sont:

- A	Signaler les déclarations de fonction sans prototype et d'autres déclarations non conformes strictement à la norme ANSI C.
-data:<name>	Donne un autre nom à la zone DATA.
-e<number>	Défini le nombre maximum d'erreurs. Le compilateur s'arrête quand le nombre d'erreurs dépasse cette limite. Par défaut, le maximum est à 20.
-l	Passe des informations de ligne de la source C dans le fichier assembleur pour le débogueur
-text:<name>	Donne un autre nom à la zone TEXT.
-w	Désactive les messages d'avertissement.

IAS6811W Assembleur

Format

ias6811w [options]<input file>

L'assembleur crée un fichier objet relogeable.

-l	L'assembleur génère un fichier (.lis) avec le code assembleur et les adresses relatives.
-o<file>	Spécifie le nom du fichier de sortie. Par défaut, le nom est construit sur la même racine que le nom en entrée avec l'extension .o

ILINKW Le Linker

Format

ilinkw [options] <file1><file2>...

ilinkw combine les fichiers objets entre eux pour former un exécutable. Il inclue automatiquement les fichiers crt11.o, end11.o et la librairie libc11.a. Le linker cherche dans les librairies après que le fichier objet soit construit, aussi, vous pouvez placer une librairie n'importe où dans la ligne de commande. Vous pouvez vous servir de l'option -L pour spécifier où les librairies se trouvent

-11	Spécifie une cible HC11, utilise les variables d'environnement et les fichiers spécifiques à ICC11.
-btext:<start address>[.end address] [:<start address>[.<end address>]]*	Défini les plages d'adresse de la section TEXT. Par défaut, la plage est de 0 à 0xFFFF.
-bdata:<start address>[.end address] [:<start address>[.<end address>]]*	Défini la plage d'adresse de la section DATA. Par défaut, cette section suit immédiatement la section TEXT jusqu'à 0xFFFF.
-b<section name>:<start address> [.end address] [:<start address>[.<end address>]]*	Défini la plage d'adresse de la section name. Suit par défaut la dernière section rencontrée.
-d<symbol>:<value>	Défini un symbole de linkage qui va être utilisé pour les références non résolues ou comme valeur pour définir une adresse de section.
-g	Génère des informations pour le débogueur.
-l<f>	Inclure la librairie lib<f>.a
-L<dir>	Défini le répertoire dans lequel les fichiers librairie seront recherchés (incluant crt11.o et end11.o).
-m	Crée un fichier map dont le nom est construit sur le nom de sortie avec une extension .mp et crée un fichier .lst qui regroupe le fichier .lis avec les adresses finales.
-R	N'inclut pas les fichiers crt11.o et end11.o
-o<file>	Défini le nom de sortie qui aura automatiquement une extension .s19. Par défaut, le nom est construit sur le nom du premier fichier en entrée.
-s<old>:<new>	Utilise la section appelée <new> comme si elle était nommée <old>.
-u<startup file>	Défini un fichier de départ. Par défaut, ce fichier est crt11.o.
-w	Désactive les messages d'avertissement.

ILIBW Gestionnaire de librairie

Cet utilitaire permet de créer et de manipuler les librairies.

Format

ilibw [options] <library archive> <object file 1> <object file 2> ...

Cet utilitaire permet de créer et de manipuler les librairies. Les fichiers librairies (par exemple libc11.a) doivent apparaître avant des fichiers objet après les options.

Les options valides sont:

-a	Ajoute le module aux archives, crée l'archive si elle n'existe pas. Si le module existe déjà, il sera remplacé.
-t	affiche les noms des modules de l'archive,
-x	Extrait les modules de l'archive.
-d	Efface les modules de l'archive.

Exemples

Pour remplacer puchar.o dans la librairie par une nouvelle version:

ilibw -a libc11.a putchar.o

Pour connaître le contenu de la librairie:

ilibw -t libc11.a

IMAKE Utilitaire Make

Format

imake [-f filename] [options] [target ...] [macro=value ...]

Cet utilitaire est utilisé pour maintenir, mettre à jour et régénérer un groupe de programme.

Si aucun makefile n'est spécifié avec l'option -f, l'utilitaire lit un fichier appelé « makefile » s'il existe.

Si aucune cible n'est spécifiée sur la ligne de commande, imake se sert de la première cible définie dans le makefile. S'il doit construire une cible, et qu'aucune règle ne peut être dérivée pour la construire, imake retourne une erreur et s'arrête.

-f<makefile>	Utilise le fichier makefile. Le contenu de ce fichier, quand il existe, remplace les définitions des règles implicites et des macros prédéfinis. Quand plus de -f makefile apparaît, imake concatène tous ses fichiers dans l'ordre d'apparence.
-d	Affiche les raisons pour lesquelles imake choisi de reconstruire une cible; affiche toutes les nouvelles dépendances.
-F	Oblige toutes les cibles à être mises à jour, construit les cibles et toutes les dépendances, même si la mise à jour n'est pas nécessaire.
-i	Ignore les codes d'erreur retournés, est équivalent à la cible spéciale .IGNORE.
-k	Stop la construction d'une cible dès qu'une erreur est rencontrée sur celle-ci, continue la construction des autres cibles.
-n	Mode sans exécution, affiche les commandes, mais ne les exécute pas. Cependant, si une ligne de commande contient une référence à la macro $ (MAKE), cette ligne sera toujours exécutée.
-r	Ignore le fichier par défaut (default.mk).
-s	Mode silencieux, n'affiche pas les lignes de commande pendant leur exécution et est équivalent à la cible spéciale .SILENT.
-S	Annule l'effet de l'option -k ce qui signifie que n'importe quel code d'erreur retourné par un processus sous-jacent pendant la construction arrêtera l'exécution et affichera le code d'erreur.
-v	Affiche le numéro de version de imake.
macro=value	Définition de macro: cette définition permet de fixer la valeur pendant l'exécution de imake. Il est prioritaire devant toute autre définition de cette macro dans le makefile.

BASIC11 Compilateur Basic pour 68HC11

Le compilateur vous permet d'écrire des programmes en Basic sur le système P.C. hôte pour des cibles à base de MC68HC11 de Motorola.

Basic (Beginner's All-purpose Symbolic Instruction Code) a été inventé en 1964 par John Kemeny et Tom Kurz au Dartmouth Coolege. Le Basic a été conçu comme un langage de base, facile à apprendre et à maîtriser en peu de temps. Les premiers systèmes Basic étaient des interpréteurs qui tournaient sur des ordinateurs bien avant l'âge des microprocesseurs et des P.C. Sa deuxième vie a vu Basic en forme d'interpréteur qui tournait dans la partie ROM des microprocesseurs.

Basic n'est pas normalisé, même s'il existe un 'ANSI Standard for minimum Basic'. Chaque implémentation de langage Basic a ses propriétés, souvent dûes aux caractéristiques du microprocesseur ciblé, mais aussi aux contraintes de la cible elle même.

On trouve sur les petits et les très petits microprocesseurs des dialectes Basic qui se contentent du strict minimum: l'octet comme seul type de donnée, IF, FOR, GOTO, GOSUB.

Le BASIC11 utilise un dialecte qui est plutôt orienté vers un langage structuré comme le Pascal ou le C. Même si on retrouve les fameux GOTO et GOSUB, il existe bien d'autres éléments pour remplacer ces directives un peut trop basic.

Le BASIC11 n'est pas un interpréteur mais un compilateur qui traduit le programme source en programme objet. Ce programme est prévu pour être brûlé dans une PROM. Il tournera sans l'aide d'un interpréteur ou d'un système d'exploitation sur le microprocesseur 68HC11.

Quelle est la différence entre un interpréteur et un compilateur?

- Le programme exécuté par un interpréteur est interprété, donc très lent. Le programme compilé par un compilateur est largement plus rapide.
- Un interpréteur sur la cible occupe automatiquement de la place dans la mémoire et reste donc limité au niveau du confort de manipulation et de la richesse du langage.
- L'interpréteur tourne directement sur la carte cible. On n'a donc pas besoin d'un P.C. comme hôte pour compiler le programme.
- Le travail sur un système hôte permet de mieux gérer et de conserver la source et la documentation du programme.

Le compilateur BASIC11 et l'assembleur AS11 tournent sous DOS. Le compilateur BASIC11 donne comme résultat un fichier en assembleur qui est traduit par AS11 en fichier format Motorola S-record. Le programme WBASIC11 tourne sous Windows. Ce programme permet d'éditer les fichiers source, lancer le compilateur sous DOS et récupérer les messages du compilateur comme les messages d'erreur. Le débogueur permet de charger le programme dans la mémoire de la cible et de déboguer le programme.

Présentation du logiciel

Le programme WBASIC11 vous présente après le lancement deux fenêtres. La fenêtre en haut est la fenêtre principale. Elle vous permet de lancer un éditeur pour éditer le programme source et compiler le programme. La fenêtre en bas contient le déboguer qui vous permet de communiquer avec la carte cible à base du 68HC11.

Dans le menu FICHIER vous trouvez les outils pour ouvrir et traiter le programme source principal.

COMPILER lance le compilateur. Le curseur d'attente s'affiche pendant la compilation. Après on trouve la sortie des programmes de compilation.

    !E <fichier>(<ligne>)

Une telle ligne affiche une erreur. Cliquer deux fois sur cette ligne ouvre une fenêtre pour éditer ce fichier et positionne la source sur la ligne erronée.

Le menu FENETRES affiche les fichiers source de l'option et tous les fichiers inclus dans ces fichiers par la directive #include. Cliquer sur un fichier ouvre une fenêtre pour éditer ce fichier.

Le dialogue OPTIONS vous permet de spécifier les options pour compiler le programme.

    Commande de compilateur
    Options de compilation
    Fichiers à compiler
    Répertoires des fichiers sources

On peut donner plusieurs options, fichiers, et répertoires, séparés par espace ou virgule. Les options sont enregistrées dans un fichier <nom>.prj. Le nom vient du nom du fichier source principal.

2.3 Exemple: Programme de base

Une fois que vous avez établi une connexion entre le P.C. et votre carte cible, on peut lancer le premier programme.

Cliquez sur FICHIER, OUVRIR, et ouvrez le programme clignot.bas.

Programme Clignot.bas:

```
ProgramPointer  $8000 ' à modifier selon la cible
DataPointer     $0002 ' à modifier selon la cible
StackPointer    $01FF ' à modifier selon la cible

byte DDRD  at $1009
byte PORTD at $1008

int  i

DDRD = $8                   ' PD3 = sortie
ASM cli                     ; enable debugger
do                          ' pour toujours
    if PORTD.3=0 then       ' basculer PD3
          PORTD.3=1
    else
          PORTD.3=0
    end if
    for i = 0 to 30000      ' tempo
    next
loop
```

Ce programme bascule régulièrement une DEL à la sortie PD3.

Il faut adapter les premières lignes du programme comme indiqué dans le chapitre précédent. Vous devrez modifier le programme, si vous avez une DEL à une autre sortie du 68HC11.

Cliquez sur COMPILER pour compiler le programme. Si le compilateur n'affiche pas d'erreurs, vous pouvez maintenant charger le programme.

Il faut que le débogueur dans la fenêtre en bas vous affiche une cible dans l'état STOP. Si la cible tourne, il suffit de cliquer sur STOP pour arrêter la cible. Sinon lancer le talker dans la cible comme c'est indiqué dans le chapitre précédent.

Cliquez sur CHARGER pour charger le programme dans la mémoire de la cible. Cliquez sur GO pour lancer votre programme.

La DEL doit maintenant clignoter.

Vous pouvez cliquer sur STOP pour arrêter le programme. Deux fenêtres vous affichent la source Basic de votre programme et le fichier listing du programme assembleur. C'est le programme créé par le compilateur Basic. Dans chaque fenêtre vous trouvez une ligne sélectionnée qui correspond à l'endroit ou se trouve le registre d'instructions PC de votre programme. Tapez à l'invité du débogueur

```
d i
```

pour afficher la valeur de la variable.

Le langage Basic11

Le Basic11 est un langage sans format strict. Il y a une instruction par ligne. L'instruction peut commencer tout à gauche dans la ligne, mais ce n'est pas obligatoire.

3.1 Les mots-clés

Les mots-clés sont des mots réservés. On ne peut utiliser un mot-clé comme nom d'une variable.

AND	**GOSUB**	**PRINT**
ASM	**GOTO**	**PROGRAMPOINTER**
AT	**IF**	**REM**
BYTE	**INT**	**RETURN**
DATAPOINTER	**INTEGER**	**STACKPOINTER**
DO	**INTERRUPT**	**STEP**
ELSE	**LOOP**	**THEN**
END	**MOD**	**TO**
EXIT	**NEXT**	**UNTIL**
EXTERN	**NOT**	**WHILE**
FOR	**OPTION**	**XOR**
FUNCTION	**OR**	

Les mots-clés peuvent être écrits en majuscule ou en minuscule. Les trois lignes suivantes sont toutes légales.

```
PROGRAMPOINTER    $E000
datapointer       $0002
StackPointer      $01FF
```

3.2 Les commentaires

Un commentaire est précédé du mot-clé REM, d'une apostrophe ou de //. Il est possible d'intégrer des commentaires dans une ligne d'instruction en les isolant d'une apostrophe.

```
REM Le Programme Clignot.bas
‘ Un programme qui fait vachement rien
A = 30              ‘ valeur de début
A = A + 1           // A maintenant 31
```

Les commentaires servent à expliquer le comportement d'un programme. Ils sont surtout utiles, si on veut changer un programme plus tard ou si quelqu'un d'autre que l'auteur doit le modifier. Le compilateur ignore les commentaires. Ils sont donc complètement inoffensifs. Aucun compilateur ne se heurte jamais aux commentaires. Mais aucun compilateur ne vérifie non plus leur conformité avec le programme...

3.3 Les options du compilateur

<prologue>	=	OPTION <string>

Le compilateur accepte des options sur la ligne de commande DOS ou par la directive option dans le programme source. Ces directives se trouvent toujours au début du programme. Une option est entourée par des guillemets. Il faut une directive option pour chaque option. Le compilateur connaît les options suivantes.

b	Les variables non déclarées sont automatiquement déclarées comme BYTE.
i	Les variables non déclarées sont automatiquement déclarées comme INTEGER.
L<répertoire>	L'option spécifie le répertoire des fichiers de la bibliothèque. Ce répertoire est /BASIC11 par défaut. Si le compilateur ne trouve pas un fichier source dans le répertoire du travail, il cherche le fichier dans le répertoire de la bibliothèque.

Par exemple:

```
Option      "b"
```

3.4 La déclaration des pointeurs

<prologue>	=	PROGRAMPOINTER <constant expression>
	|	DATAPOINTER <constant expression>
	|	STACKPOINTER <constant expression>

Les déclarations des pointeurs sont essentielles pour le comportement du programme. Ces déclarations se trouvent au début du programme comme la directive option.

```
ProgramPointer    $E000
```

Le compilateur stocke le programme et les données invariantes dans la mémoire à partir de cette adresse. Cette adresse devrait donc être le début de la mémoire PROM, EPROM ou EEPROM de la carte cible.

```
DataPointer       $0002
```

Le compilateur réserve de la place pour les variables dans la mémoire à partir de cette adresse. Cette adresse devrait donc être le début de la mémoire vive, RAM. La RAM d'un 68HC11 commence toujours à l'adresse $0000. Il est néanmoins conseillé d'éviter les deux premiers octets de la RAM, parce que les programmes erronés ont la fâcheuse tendance d'écrire à l'adresse 0.

```
StackPointer      $00E8
```

Le pointeur de pile devrait être placé à la fin de la mémoire vive libre. Prévoyez surtout assez de place pour la pile du programme. La pile est utilisée pour les appels de fonctions et pour leurs variables locales.

Si le programme ne contient pas la déclaration du ProgramPointer, le compilateur ne prend pas ce programme pour un programme principal. Il va générer un programme assembleur mais ne le traduit pas en programme objet.

3.5 La directive #include et le fichier start.bas

Cette directive commence tout à gauche dans une ligne avec #.

```
#include <nom de fichier>
```

La directive permet d'inclure un autre fichier dans la source. Le compilateur compile ce fichier et ensuite retourne au premier fichier et continue après la commande. Le fichier inclus peut inclure d'autres fichiers.

Il est conseillé d'inclure dans chaque programme Basic le fichier start.bas. Ce fichier inclut les déclarations des pointeurs qui sont essentiels pour le programme. Il faut donc changer les trois premières lignes du fichier start.bas selon la carte cible utilisée. Le petit programme en assembleur qui suit ces déclarations sera exécuté après le lancement du programme et avant l'exécution du programme Basic. Ce programme met toutes les variables à 0, ce qui évite pas mal d'erreurs. Enfin, ce fichier contient les définitions des ports du microprocesseur. Voici le fichier start.bas:

```
ProgramPointer     $E000  ' à modifier selon la cible
DataPointer        $0002
StackPointer $01FF        ' à modifier selon la cible

             sect   text
             cli                ; enable debugger
             ldx    #_data_s
             bra    _crt2
_crt1        clr    0,x         ; clear data area
             inx
_crt2        cpx    #_data_e
             bne    _crt1

             sect   data
_data_s      equ *

byte SCONF         at $0420     ' Controlboy 2 /3
byte PORTB         at $0410     ' avex X68C75
byte PORTBI        at $0430
byte PORTC         at $0408
byte PORTCI        at $0428

'byte PORTB        at $1004     ' Controlboy 1
'byte PORTC        at $1003     ' mode single-chip
'byte DDRC         at $1007

byte PORTA         at $1000
byte PIOC          at $1002
byte PORTCL        at $1005
byte PORTD         at $1008
byte DDRD          at $1009
byte PORTE         at $100A
byte TMSK2         at $1024
byte TFLG2         at $1025
byte PACTL         at $1026
byte PACNT         at $1027
byte BAUD          at $102B
byte SCCR1         at $102C
byte SCCR2         at $102D
byte SCSR          at $102E
byte SCDR          at $102F
byte ADCTL         at $1030
byte ADR           at $1031
byte OPTIONS       at $1039
```

3.6 La déclaration des variables, leurs noms, leurs types

<declaration>	=	<type> <variable> [,<variable>]*
	|	<type> <variable>(<constant expression>)
<type>	=	BYTE | INTEGER | INT

Chaque variable a un nom jusqu'à 31 caractères. Ce nom doit commencer par une lettre ou un blanc souligné(_) suivi par des lettres ou des chiffres ou des blancs soulignés (_).

```
abc
Abc
A1b2c3
s_S_123_S
```

Ces exemples sont tous des noms légaux. Notez que abc et Abc sont des noms différents et représentent deux variables différentes. Un nom doit être différent des mots-clés.

Les types des variables

On doit déclarer une variable avant de l'utiliser.

Le type BYTE (octet) gère des variables entières dans l'intervalle de valeurs

BYTE 0 à 255

Une variable déclaré BYTE est stockée dans 1 octet dans la mémoire.

Le type INTEGER ou INT gère des nombres entiers dans l'intervalle de valeurs

INTEGER -32768 à 32767

Une variable déclarée INTEGER est stockée dans 2 octets.

Notez que le 68HC11 est un petit microcontrôleur qui travaille naturellement dans un mode 8 bits. Il ne faut pas le confondre avec un processeur de grande taille. Les variables BYTE sont donc beaucoup plus adaptées à son unité centrale, qui a peu d'instructions pour le traitement à 16 bits. On utilise des variables INT pour des variables qui risquent de dépasser la valeur qu'on peut stocker dans un octet.

Les lignes suivantes déclarent les variables a, ab, Ab et ab_b comme BYTE et les variables i, j, k et k33 comme INTEGER. Ces variables sont placées dans la mémoire vive RAM.

```
byte  a
byte  ab, Ab, ab_b
int   i, j, k, k33
```

Les tableaux

Tous les tableaux doivent être déclarés comme les variables simples. Une variable de type tableau peut stocker plusieurs données. Les tableaux sont limités à une dimension.

```
byte abc(5)
```

Cette instruction déclare la variable abc comme tableau de 5 octets dans la RAM. Le premier octet est accessible par abc(0) est le dernier par abc(4).

Notez, si vous utilisez le fichier start.bas que toutes les variables sont mises à zéro après le RESET. Sinon le contenu des variables est indéterminé.

Les déclarations en PROM

```
<declaration>   =   <type> <variable>() = <constant expression>
                                          [,<constant expression>]*
                |   <type> <variable>() = <string>
<string>        =   "<asciistring>"
```

Nos premières déclarations étaient des déclarations pour des variables en mémoire vive. On peut également déclarer des variables en mémoire PROM. On ne peut écrire dans ces variables, comme elles se trouvent dans la mémoire morte du microprocesseur. Mais on peut initialiser ces variables.

```
int def() = 1, 3, 7, 9
byte titre() = "Controlboy"
```

La première ligne déclare un tableau de quatre éléments à 16 bits. La deuxième ligne déclare une chaîne de caractères. titre(0) contient le caractère ASCII ‘C’, titre(1) le caractère ‘o’. Après le dernier caractère ‘y’ il y a un dernier élément qui contient la valeur 0 et qui termine le tableau.

Les déclarations des ports entrée sortie

```
<declaration>   =   <type> <variable> AT <constant expression>
```

Il reste à déclarer les variables qui se trouvent à une certaine adresse du microprocesseur. Ce sont normalement des ports, dont les adresses sont connues. On déclare un port suivi du mot-clé AT et l’adresse du port.

```
byte PORTA at $1000
```

La ligne déclare la variable à l’adresse indiquée. Aucun espace dans la mémoire n’est réservé ni dans la RAM ni dans la PROM.

Les déclarations des variables externes

Pour accéder a une variable déjà déclarée dans un autre programme, on déclare la variable externe.

```
extern byte seconde
```

3.7 Les affectations et les calculs

```
<instruction>   =   <lexpression> = <expression>
<lexpression>   =   <variable> | <variable> ( <expression> )
<expression>    =   <primary>
                |   - <primary> | NOT <primary>
                |   <expression> * <primary> | <expression> / <primary>
                |   <expression> MOD <primary>
                |   <expression> + <primary> | <expression> - <primary>
                |   <expression> <compare> <expression>
                |   <expression> AND <primary>
                |   <expression> OR <primary> | <expression> XOR <primary>
<primary>       =   <variable>
                |   <variable> ( <expression> )
                |   <constant>
                |   ( <expression> )
<compare>       =   = | == | <> | != | > | >= | < | <=
```

On affecte des valeurs à une variable à l'aide du signe égal. On peut affecter une constante, le contenu d'une autre variable ou une expression. Voici quelques exemples d'affectations:

```
byte a,b,c(10)
int  i, j(10)

a = 7                   la variable a est mise à 7
i = -100                la variable i est mise à -100
b = a + 1               b = 8
c(3) = b + a * 2        c(3) = 8 + 7 * 2 = 22
c(a) = ( b + a ) * 2    c(7) = ( 8 + 7 )* 2 = 30
b = c(3) / a            b = 22 / 7 = 3
```

Une expression se compose de constantes, de variables et d'opérateurs. Les opérateurs sont appliqués selon leurs priorités. Ils sont classifiés dans quatre groupes.

Dans le premier groupe d'opérateurs on trouve les parenthèses qui ont la priorité la plus élevé. Elles permettent de changer la priorité d'une expression.

Le deuxième groupe est le groupe des opérateurs arithmétiques qui font les calculs classiques. Les opérateurs sont traités dans l'ordre de leur priorité. La multiplication et la division sont appliquées avant l'addition et la soustraction.

Les opérateurs relationnels représentent le troisième groupe. Ils sont utilisés dans les tests: IF. Ils comparent deux valeurs. Un opérateur relationnel renvoie une valeur logique. Tous les bits sont à 1 quand l'opération est vraie. La valeur est à 0 quand l'opération est fausse.

Les opérateurs logiques dans le quatrième groupe sont effectués bit par bit. On utilise ces opérateurs dans les tests mais aussi pour effectuer des opérations sur des variables. Voilà les exemples les plus courants.

```
PORTA = PORTA or $08            mettre le bit 3 du port A à 1
PORTA = PORTA and not $08       mettre le bit 3 du port A à 0
PORTA = PORTA xor $08           inverser le bit 3 du port A
if PORTA and $08 then           examiner le bit 3 du port A
```

On peut adresser directement un bit d'une variable et notamment d'un port.

```
PORTA.3 = 1                 mettre le bit 3 du port A à 1
PORTA.3 = 0                 mettre le bit 3 du port A à 0
if PORTA.3 = 1 then         examiner le bit 3 du port A
```

Priorité	Opérateur	Description
1 (plus élevée)	**( )**	Parenthèses
2	**-**	Négation
	NOT	Négation logique
3	*****	Multiplication
	/	Division
	MOD	Reste de division
4	**+**	Addition
	-	Soustraction
5	**=**	Egalité (également ==)
	<>	Différent de (également !=)
	>	Supérieur à
	>=	Supérieur ou égal à
	<	Inférieur à
	<=	Inférieur ou égal à
6	**AND**	Et logique
7 (plus basse)	**OR**	Ou logique
	XOR	Ou logique exclusif

Les constantes

```
<constant>      =   [0-9]+
                |   $[0-9|A-F|a-f]+
                |   0x[0-9|A-F|a-f]+
                |   %[0|1]+
                |   `<asciicharacter>`
```

On peut entrer des constantes dans une expression d'une des manières suivantes.

```
123             nombre décimal.
$001f           nombre hexadécimal
0x001F          nombre hexadécimal
%0001100        nombre binaire
`y`             caractère ASCII.
```

3.8 La boucle FOR

<instruction>	=	FOR <variable> = <expression> TO <expression> [STEP <constant expression>]
	|	NEXT [<variable>]
	|	EXIT FOR

La boucle FOR utilise une variable de comptage. On définit une valeur de début, une valeur de fin et le pas de l'incrémentation. Si aucun pas n'est indiqué, la valeur 1 est automatiquement utilisée.

La boucle se termine par l'instruction NEXT. On peut ajouter à cette instruction le nom de la variable de comptage de la boucle.

Le premier exemple est une boucle qui initialise les éléments d'un tableau.

```
byte  a, b(10)
int   i, j
for a = 0 to 9
      b(a) = 0
next
```

Le pas de la boucle peut être positif ou négatif. Une boucle peut contenir une autre boucle.

```
for i = 2 to -30 step -2
      ...
      for j=i to i+4
            ...
      next j
      ...
next i
```

L'instruction EXIT FOR permet de quitter prématurément une boucle. Le programme continue après l'instruction NEXT.

```
for a = 0 to 9
      ...
      if b(a) = 3 then exit for
      ...
next a
```

3.9 La boucle DO LOOP WHILE UNTIL

<instruction>	=	DO [WHILE | UNTIL <expression>]
	|	EXIT DO
	|	LOOP [WHILE | UNTIL <expression>]

La directive DO declare une boucle sans variable de comptage. La boucle se termine par l'instruction LOOP.

La boucle suivante ne s'arrête jamais. Un programme sur un système embarqué ne doit jamais s'arrêter. Il a donc besoin d'une boucle sans fin.

```
do
      ...
loop
```

La boucle suivante utilise une condition. Les instructions de la boucle ne seront exécutées que si la condition est vraie avant le début de la boucle. Si la condition n'est plus remplie, la boucle s'arrête et le programme continue après l'instruction LOOP. Si la condition est déjà fausse au début, la boucle ne sera pas exécutée.

```
do while PORTA.3 = 0
      ...
loop
```

La boucle suivante utilise également une condition. Mais le sens de la condition est inversé. La boucle s'arrête dès que la condition soit remplie. (UNTIL = jusqu' à).

```
do until PORTA.3 = 1
      ...
loop
```

La boucle suivante est comme la boucle avant, mais la condition est testée à la fin de la boucle. Le programme va donc exécuter cette boucle en tous cas une fois.

```
do
      ...
loop until PORTA.3 = 1
```

L'instruction EXIT DO permet de quitter prématurément la boucle. Le programme continue après l'instruction LOOP.

3.10 Le test IF

<instruction>	=	IF <expression> THEN <instruction> [ELSE <instruction>]
	|	IF <expression> THEN
	|	ELSE
	|	END IF

Les tests vous donnent un moyen de contrôler le déroulement du programme. Le test utilise souvent une comparaison et exécute l'instruction qui suit le THEN, si la comparaison est vraie et l'instruction qui suit le ELSE si la comparaison est fausse. La partie ELSE est facultative.

```
byte a,b,c
int j(5)
if a<5 then c=0 else c=1
```

Si la variable est inférieure à 5, c est mis à 0, sinon c est mis à 1. S'il faut exécuter plusieurs instructions selon le résultat du test, on choisit la syntaxe suivante. La partie ELSE est toujours facultative.

```
if a<5 then
      c = 0
      ...
else
      c = 1
      ...
end if
```

On peut examiner le bit d'un port ou d'une variable.

```
if PORTA.3 = 1 then
```

La comparaison peut être assez complexe.

```
if j(a)+j(b) < a+b*c then ...
```

Le test peut unir plusieurs comparaisons par des opérateurs logiques, notamment AND et OR. Les deux tests suivants donnent le même résultat.

```
if a>1 and a <4 then ...
if a=2 ou a=3 then ...
```

Les opérateurs relationnels utilisés dans les comparaisons sont les suivants.

=	Egalité (également ==)
<>	Différent de (également !=)
>	Supérieur à
>=	Supérieur ou égal à
<	Inférieur à
<=	Inférieur ou égal à

3.11 Les instructions de saut GOTO, GOSUB

<instruction>	=	GOTO <label>
	|	GOSUB <label>
	|	RETURN [<expression>]

GOTO et GOSUB vous donnent un autre moyen de contrôler le déroulement du programme. GOTO permet un branchement direct vers une autre instruction.

```
if a<5 then c=0 else c=1
```

On peut remplacer cette ligne par le petit programme suivant.

```
if a>=5 then goto ici1
c = 0
goto ici2
ici1: c=1
ici2:
```

GOSUB permet un branchement vers un sous-programme. Le sous-programme doit se terminer par l'instruction RETURN. Le programme continue après le RETURN avec l'instruction qui suit le GOSUB. Mais GOSUB ne peut ni passer des paramètres ni recevoir un résultat. Les fonctions sont donc conseillées pour remplacer le GOSUB.

```
gosub ici
...          ' le programme continue ici après le RETURN

ici: ...     ' sous-programme
...
return       ' fin du sous-programme
```

3.12 La directive ASM

<instruction>	=	ASM <textstring>

La directive ASM permet d'inclure une ligne assembleur dans le programme. La syntaxe de la directive est le mot-clé ASM suivi par l'instruction assembleur. Mais ce mot-clé n'est même pas nécessaire. Le compilateur reconnaît automatiquement des instructions assembleur dans le programme Basic.

Les deux lignes Basic donnent donc le même résultat.

```
asm   cli
cli
```

L'instruction assembleur CLI permet des interruptions de l'unité centrale.

L'instruction assembleur peut accéder aux variables globales et aux fonctions par leurs noms. Les paramètres et les variables locales d'une fonction se trouvent sur la pile. Il faut compiler le programme Basic et chercher la déclaration de la fonction dans le programme assembleur qui est généré par le compilateur. On trouve les adresses sur la pile derrière la déclaration en commentaire.

3.13 L'affichage PRINT

<instruction>	=	PRINT <printelement> [,<printelement>]*
<printelement>	=	<expression>
	|	<string>

L'instruction PRINT affiche des données sur un afficheur à cristaux liquides, sur l'interface série RS232 ou bien sur un autre port périphérique.

L'instruction PRINT est suivie par un ou plusieurs éléments à afficher. On peut afficher des chaînes de caractères, des variables et des expressions des variables. Les variables sont affichées comme nombres décimaux. Un tableau de type BYTE est affiché comme chaîne de caractères.

```
int i, j
i = 5
j = -3
print i," / ", j, " = ", i/j, " R ", i MOD 3
```

vous affichera

```
5 / -3 = -1 R 2
```

La fonction suivante affiche une variable de type byte comme nombre hexadécimal.

```
function displayhexbyte(hexdata)
      byte tohex()="0123456789ABCDEF"
      putchar(tohex((hexdata/16) AND $F))
      putchar(tohex(hexdata AND $F))
      end function
```

Putchar

L'instruction PRINT est basée sur une fonction putchar qui doit être fournie par l'application. Si votre programme doit afficher la sortie du PRINT sur un afficheur à cristaux liquides, la fonction putchar doit afficher un caractère sur l'afficheur. Vous trouvez une telle fonction dans le fichier LCD.BAS. Il suffit d'inclure ce fichier dans votre programme pour que le PRINT s'affiche sur l'afficheur à cristaux liquides. Dans le programme principal il faut néanmoins appeler une fois la routine lcdinit pour initialiser l'afficheur.

Voilà une autre fonction putchar qui affiche la sortie du PRINT sur l'interface série RS232.

```
function putchar(x)
do
loop until SCSR.7 = 1
SCDR = x
end function
```

3.14 Les fonctions

<declaration>	=	FUNCTION <function> ([<variable> [,<variable>]*])
	|	END FUNCTION
<instruction>	=	RETURN [<expression>]
<primary>	=	<function> ([<expression> [,<expression>]*])

Les fonctions ou procédures vous donnent le meilleur moyen pour structurer votre programme. Une fonction est un sous-programme. Un programme peut appeler cette fonction. On peut passer quelques paramètres à la fonction. La fonction exécute le sous-programme avec ces paramètres et peut renvoyer un résultat. Le programme qui a appelé la fonction reçoit le résultat et finit son calcul avec celui-ci. Il y a bien des fonctions, qui n'ont aucun paramètre. Et il y a aussi des fonctions qui ne donnent aucun résultat. On appelle également ces dernières des procédures.

Voici l'exemple d'une fonction qui calcule le maximum de ces deux paramètres:

```
function max(a, b)
if a>b then return a else return b
end function
```

La déclaration de la fonction commence avec le mot-clé FUNCTION suivi du nom de la fonction et la liste des paramètres formels. Le mot-clé SUB peut remplacer le mot FUNCTION. L'instruction RETURN permet de quitter la fonction et de passer un résultat au programme qui a appelé la fonction. L'instruction END FUNCTION finit la déclaration de la fonction. Cette instruction force automatiquement une instruction RETURN.

```
int i, j, k
i = 8
j = 17
k = max(i, j)                ' k sera 17
```

Le programme principal appelle la fonction par son nom et la liste de paramètres actuels qui sont passés à la fonction. Dans notre cas, la fonction max travaille sur le paramètre a, qui est une copie de la variable i, donc 8, et sur le paramètre b, qui est une copie de la variable j, donc 17. Max renvoie donc certainement 17 comme résultat.

Une fonction peut appeler une autre fonction. On peut librement composer une expression à base des fonctions et des autres éléments.

```
k = 100 - max(max(7+i,j-3),max(x,max(i,k)))
```

Paramètres

Les paramètres et le résultat de la fonction sont de type INTEGER. Si vous passez une variable de type BYTE, la variable sera automatiquement transformée en type INTEGER. Les paramètres sont des variables simples ou des éléments d'un tableau. Un tableau est interdit comme paramètre. Les paramètres de la fonction sont passés par valeur. La fonction travaille donc sur des copies des variables et ne change pas les variables du programme qui a appelé la fonction.

```
int i, j
i = -6
j = abs(i)
...

function abs(x)          ‘ x est une copie de i
if x < 0 then x = -x     ‘ change x, ne change pas i
return x
end function
```

Fonctions récursives

Les fonctions peuvent s'appeler elles-mêmes. Ce sont alors des fonctions récursives. Il y a peu d'applications qui utilisent des telles fonctions. L'exemple suivant montre une fonction qui affiche une variable décimale à l'aide d'une fonction putchar qui affiche un caractère. Si la variable (1234) est supérieure à 9, on traite la partie supérieure (1234 / 10 = 123) avant d'afficher le dernier chiffre (1234 MOD 10 = 4). C'est difficile de trouver une solution aussi simple sans récursivité. Mais c'est peut-être aussi difficile de trouver une solution aussi incompréhensible.

```
function printnum(x)
if x <0 then
      putchar(`-`)
      x = -x
end if
if x > 9 then printnum(x/10)
putchar (x MOD 10 + `0`)
end function
```

Les fonctions externes

Pour accéder a une fonction déjà déclarée dans un autre programme, on déclare la fonction externe.

```
extern function lcdinit()
...
lcdinit()
```

3.15 Les domaines de validité des variables globales et locales

Les variables déclarées au début d'un programme Basic sont accessibles dans la totalité du programme. On les appelle des variables globales.

On peut déclarer des variables au début d'une fonction. On les appelle les variables locales. Elles sont initialisées à zéro chaque fois que la fonction démarre. Les paramètres d'une fonction et les variables locales sont stockés dans la pile. Ils sont donc accessibles seulement pendant l'exécution de la fonction. La place qu'ils occupent dans la pile est libérée en quittant la fonction.

Un fichier source Basic contient des éléments suivants.

Déclarations des variables globales
    accessibles au programme principal et aux sous-programmes
...
Instructions du programme principal
...

FUNCTION func1()
Déclarations des variables locales
    seulement accessibles au sous-programme func1
...
Instructions du sous-programme func1
...
END FUNCTION

FUNCTION func2()
Déclarations des variables locales
    seulement accessibles au sous-programme func2
...
Instructions du sous-programme func2
...
END FUNCTION

L'exemple montre les déclarations et l'usage des variables globales et locales.

```
int i,j                 ‘ donnees globales
i = abs(j)
...
function abs(x)         ‘ x sur la pile
int r                   ‘ r sur la pile
if x < 0 then r = -x else r =x
return r                ‘ la place est liberee
end function            ‘ x et r ont disparu
```

3.16 Exemple: Entrée par clavier, sortie par afficheur LCD

Ce programme est écrit pour une carte cible Controlboy 3 avec un afficheur cristaux liquide et un clavier à 12 touches type téléphone.

La fonction putchar qui affiche un caractère sur l'afficheur se trouve dans le fichier lcd.bas qui est inclus à la fin du programme. On trouve dans ce fichier également la fonction keyget qui examine le clavier. Quand on presse une touche la routine renvoie le code ASCII de la touche pressée.

Le programme principal initialise les ports utilisés et appelle la fonction lcdinit pour initialiser l'afficheur. Après il nous affiche gentiment un message d'accueil et entre dans une boucle sans fin. Quand on presse la touche T1 sur la carte, le programme fait klaxonner les six relais de la carte d'une manière aléatoire. La fonction tempo permet de ralentir cette opération. Mais cette fonction surveille aussi le clavier. Quand on presse une touche du clavier, la fonction affiche la touche sur l'afficheur.

```
' programme pour tester LCD et clavier sur Controlboy 3

#include "start.bas"

      DDRD  = 0               ' PD5=PD4=PD3=PD2=input
      SCONF = 0x4C            ' B,C = sorties
      lcdinit()
      print " Controlboy 3  "

      do                            ' pour toujours
      if PORTD.5 = 1 then
            PORTB = 0               ' if T1 = 1
      else
            PORTB = PORTB + 37      ' if T1 = 0
      end if
      tempo(5)
      loop

function tempo(cnt)
      int i, k
      for cnt=cnt to 0 step -1
            for i=0 to 100
                  k = keyget()
                  if k <> 0 then putchar(k)
            next
      next
      end function

#include "lcd.bas"
```

3.17 Routine d'interruption

<declaration>	=	INTERRUPT FUNCTION <function> AT <constant expression>

La déclaration d'une fonction d'interruption permet de traiter des interruptions de l'unité centrale.

L'exemple utilise une routine d'interruption pour l'horloge du temps réel. Le programme principal déclenche l'horloge, autorise les interruptions et continue dans une boucle sans fin. La routine rtiinit est la fonction d'interruption. On doit charger l'adresse de la fonction comme vecteur d'interruption. Le vecteur pour l'horloge du temps réel se trouve à l'adresse $FFF0 comme indiqué dans la déclaration de la fonction. L'horloge appelle la fonction d'interruption régulièrement, 244 fois par seconde. La fonction a ni des paramètres ni de résultat. Le programme compte le temps écoulé dans les variables seconde et minute et affiche ce temps chaque seconde.

```
' demonstration d'une routine d'interruption

#include "start.bas"

      byte s, seconde, tictac
      int   minute

      lcdinit()               ' enable l'afficheur
      PACTL.1 = 0             ' selectioner la vitesse
      PACTL.0 = 0
      TMSK2.6 = 1             ' declencher le timer
      cli                     ' autoriser les inter
      do                      ' pour toujours
      if seconde <> s then    ' si seconde a change ...
            if seconde >= 60 then
                  seconde = 0
                  minute = minute + 1
            end if
            print "   ", minute,":",seconde
            s = seconde
      end if
      loop

interrupt function rtiint at $FFF0
      tictac = tictac + 1
      if tictac >= 244 then   ' 8Mhz: 244, 4,9Mhz: 150
            tictac = 0
            seconde = seconde + 1
      end if
      TFLG2.6 = 1             ' autoriser ints a nouveau
      end function

#include "lcd.bas"
```

3.18 Syntax

```
statement       =  <prologue>
                |  <declaration>
                |  [ <label> : ] <instruction>

<prologue>      =  OPTION <string>
                |  PROGRAMPOINTER <constant expression>
                |  DATAPOINTER <constant expression>
                |  STACKPOINTER <constant expression>

<declaration>   =  <type> <variable> [ ,<variable> ]*
                |  <type> <variable>(<constant expression>)
                |  <type> <variable>() = <constant expression>
                                        [,<constant expression>]*
                |  <type> <variable>() = <string>
                |  <type> <variable> AT <constant expression>
                |  FUNCTION <function> ( [<variable> [,<variable>]* ] )
                |  INTERRUPT FUNCTION <function> AT <constant
                   expression>
                |  END FUNCTION

<type>          =  BYTE | INTEGER | INT

<instruction>   =  <lexpression> = <expression>
                |  FOR <variable> = <expression> TO <expression>
                                        [ STEP <constant expression> ]
                |  NEXT [<variable>]
                |  EXIT FOR
                |  DO [ WHILE | UNTIL <expression> ]
                |  EXIT DO
                |  LOOP [ WHILE | UNTIL <expression> ]
                |  IF <expression> THEN <instruction> [ ELSE <instruction> ]
                |  IF <expression> THEN
                |  ELSE
                |  END IF
                |  GOTO <label>
                |  GOSUB <label>
                |  RETURN [<expression>]
                |  PRINT <printelement> [,<printelement>]*
                |  ASM <textstring>
                |  REM <textstring>

<lexpression>   =  <variable> | <variable> ( <expression> )

<expression>    =  <primary>
                |  - <primary> | NOT <primary>
                |  <expression> * <primary> | <expression> / <primary>
                |  <expression> MOD <primary>
                |  <expression> + <primary> | <expression> - <primary>
                |  <expression> <compare> <expression>
                |  <expression> AND <primary>
                |  <expression> OR <primary> | <expression> XOR <primary>

<primary>       =  <variable>
                |  <variable> ( <expression> )
                |  <function> ( [<expression> [ ,<expression>]* ] )
                |  <constant>
                |  ( <expression> )
```

```
<compare>       =  = | == | <> | != | > | >= | < | <=

<constant>      =  [0-9]+
                |  $[0-9|A-F|a-f]+
                |  0x[0-9|A-F|a-f]+
                |  %[0|1]+
                |  `<asciicharacter>`

<printelement   =  <expression>
>
                |  <string>

<string>        =  "<asciistring>"
<variable>      =  <name>
<label>         =  <name>
<function>      =  <name>
<name>          =  [A-Z|a-z|_] [A-Z|a-z|0-9|_]*
```

4 Préprocesseur

Le préprocesseur et la première étape de la compilation. Le préprocesseur inclut autres fichiers source dans le programme, traite la compilation conditionnelle et les macros.
Il permet de remplacer les lignes

```
if PORTD.3=0 then           ' basculer PD3
          PORTD.3=1
else
          PORTD.3=0
end if
```

Par des lignes suivantes, qui rende votre programme plus lisible

```
#define LED PORTD.3
if LED=0 then          ' basculer LED
          LED=1
else
          LED=0
end if
```

Les commandes préprocesseur commencent tout à gauche dans la ligne avec #.

#define <nom> <text>	definir un macro sans parametre
#define <nom>(params) <text>	definir un macro avec parametres
#undef <nom>	supprimer un macro
#if <ifexpr>	compiler lignes suivantes si <ifexpr> est vrais
#ifdef <nom>	compiler lignes suivantes si <nom> est défini
#ifndef <nom>	compiler lignes suivantes si <nom> n'est pas défini
#else	inverser #if #ifdef #ifndef
#endif	fin de #if #ifdef #ifndef
#include <nom de fichier>	inclure <nom de fichier>

La commande **#include** permet d'inclure un autre fichier dans la source. L'assembleur compile ce fichier et ensuite retourne au premier fichier et continue après la commande. Le fichier inclus peut inclure d'autres fichiers.
<ifexpr> est une expression arithmétique <expr> ou la comparaison des deux expressions.

<ifexpr> = <expr> [==|!=|>|>=|<|<= <expr>]

5 Assembleur

Le compilateur crée à partir de votre programme Basic11 TOTO.BAS un programme assembleur avec l'extension A11. Le compilateur lance l'assembleur qui crée le fichier objet. On peut directement créer un fichier source assembleur et compiler le programme avec l'assembleur. Si votre fichier source a l'extension A11, le programme Windows ne lance que l'assembleur. L'assembleur DOS a la syntaxe

```
as11 [option]* <nom du fichier>
```

L'assembleur traduit le fichier source et génère plusieurs fichiers.

<nom du fichier>**.a11**	fichier source assembleur
<nom du fichier>**.s19**	fichier objet. Motorola S-records
<nom du fichier>**.lst**	fichier listage

Le fichier listage contient la ligne source et l'adresse et les données générées par l'assembleur. Le fichier objet contient des données en format Motorola S-records pour les charger dans la cible. Le fichier contient aussi des informations pour le débogueur.

L'assembleur affiche les segments et leur première et dernière adresses utilisées.

L'assembleur DOS accepte les options sur la ligne de commande. On peut entrer les options aussi par la directive **option**. L'assembleur connaît les options suivantes.

-n	Fichier objet sans informations pour le débogueur.
-b	Ne pas transformer les Branches en Jumps.
-j	Traiter les Jumps comme les Branchs.
-g	Les symboles sont locaux par défaut.
-l	Numéros de ligne dans le fichier listage.
-c	Comptage de cycles du 68HC11 dans le fichier listage.
-2..-9	Passes (défaut: 3)

Si l'option -b n'est pas mise, l'assembleur remplace automatiquement des Branchs trop loin par des Jumps. L'assembleur exécute plusieurs passes pour réduire successivement les Jumps par des Branches.

Si l'option -g n'est pas mise, tous les symboles sont globaux. Si l'option est mise, le secteur de validité d'un symbole est le fichier dans lequel il est déclaré. On peut le déclarer global par la directive **.globl**. Un symbole déclaré avec deux deux-points est aussi global.

Les directives suivantes sont des opérations qui ne correspondent pas aux opérations de l'unité centrale et qui sont directement exécutées par l'assembleur.

	option	[r|n|b|j|g|l|c|2..9]*	mettre option
	org|.org	<valeur>	changer l'adresse d'assemblage
	fcb|.byte	<valeur>[,<valeur>]*	stocker des octets en mémoire
	fcb|.ascii	'<chaîne ASCII>'	stocker des octets en mémoire
	fdb|.word	<valeur>[,<valeur>]*	stocker des mots de 16 bit
	rmb|.blkb	<valeur>	réserver octets en mémoire
<nom>	equ|=	<valeur>	déclarer un symbole
	sect|.area	text|data|bss|none	changement de la section
	.globl	<nom>[,<nom>]*	déclarer symbole global
	end		ignoré

La commande sect vous permet de gérer plusieurs adresses d'assemblage dans le fichier par exemple pour la RAM (**`sect data`**) et pour l'EEPROM (**`sect text`**).

Un symbole a jusqu'à 31 caractères. Les caractères suivants sont reconnus.

a..z A..Z 0..9 _ . $

Un nom ne peut pas commencer avec un chiffre ni avec $.

Une expression arithmétique <expr> peut être composée par des symboles, des constantes et le '*' ou le '.', qui représentent l'adresse d'assemblage actuel. On peut les combiner par des opérations + - * / ( ).

Le premier caractère d'une constante décide son interprétation:

'	caractère ASCII
$	numéro hexadécimal
0x	numéro hexadécimal
%	numéro binaire

Sinon c'est une valeur décimale.

Une étiquette se trouve dans la ligne source tout à gauche. elle peut être suivie par un ou deux deux-points. Ensuite il y a un ou plusieurs caractères espace suivi par l'instruction. Ensuite il y a de nouveau un ou plusieurs caractères espace suivi par les opérandes. Finalement il y a les commentaires. Une ligne qui commence avec * ou avec ; est un commentaire.

étiquette[:[:]] opération [opérandes] [commentaire]

L'assembleur inclut également un préprocesseur. Il ne faut pas le confondre avec le préprocesseur du Basic11. Si on travaille en Basic11, ce son préprocesseur qui traite ces commandes. Des commandes préprocesseur commencent tout à gauche dans la ligne avec #.

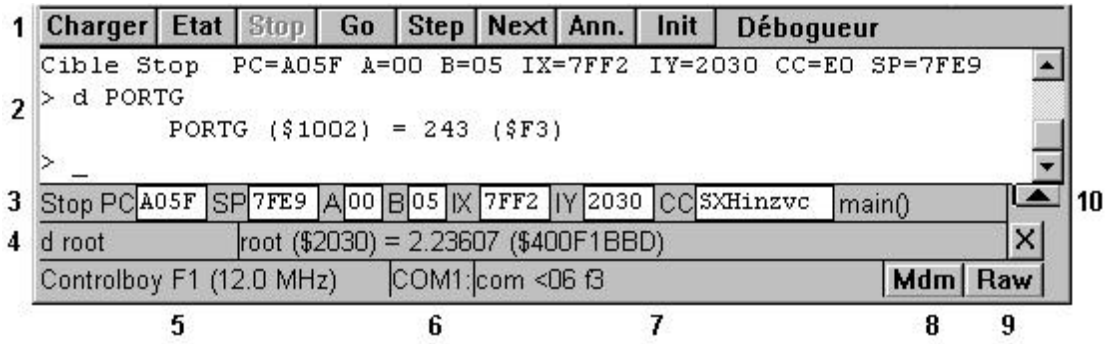
#if <ifexpr>	assembler lignes suivantes si <ifexpr> est vrais
#ifdef <nom>	assembler lignes suivantes si <nom> est défini
#ifndef <nom>	assembler lignes suivantes si <nom> n'est pas défini
#else	inverser #if #ifdef #ifndef
#endif	fin de #if #ifdef #ifndef
#assert <ifexpr>	signaler une erreur si <ifexpr> est faux
#include <nom de fichier>	inclure <nom de fichier>

La commande **`#include`** permet d'inclure un autre fichier dans la source. L'assembleur compile ce fichier et ensuite retourne au premier fichier et continue après la commande. Le fichier inclus peut inclure d'autres fichiers.

<ifexpr> est une expression arithmétique <expr> ou la comparaison des deux expressions.

<ifexpr> = <expr> [==|!=|>|>=|<|<= <expr>]

Débogueur



Le débogueur assure la liaison entre le PC et la carte cible. Le programme tourne essentiellement sur le PC (l'hôte) mais une petite partie qui s'appelle talker tourne dans la cible.

1. Les commandes les plus courantes sont accessibles par boutons.
2. Dans la fenêtre vous entrez la commande et recevez la réponse.
3. L'état de la cible, les registres de l'unité centrale.
4. Une ou plusieurs lignes de visualisation mémoires. Voir Watch.
5. La configuration de la cible. Cliquer là-dessus pour la changer.
6. Le port de communication. SIM pour simulateur.
7. La communication avec le talker sur la carte cible.
8. Modem. Voir Déboguer une carte cible à distance
9. Voir Fenêtre de mode brut.
10. Deux boutons, s'ils sont accessibles, permettent de monter et descendre dans la pile.

Mise en route

Le débogueur tourne dans la fenêtre en bas. Quand le programme démarre il affiche un des messages suivant:

Cible ne répond pas	Le débogueur n'arrive pas à communiquer avec la cible notamment le talker. Voir Préparation du logiciel pour la carte cible
Cible tourne	Un programme tourne sur la cible. Quand un programme tourne sur la cible quelques commandes sont permises. Il y a des commandes comme charger un programme qu'elles ne peuvent pas être exécutées. Pour changer l'état de la cible il faut taper la commande STOP ou cliquer sur le bouton STOP. Ensuite le débogueur affiche …
Cible stop	Le débogueur et le talker sont désormais prêts et à vos ordres. Si le programme ne s'arrête pas, il faut démarrer le talker sans démarrer le programme. Appuyez sur les touches T1 et RESET en même temps, lâchez la touche RESET avant et T1 après.

Le débogueur affiche dans sa fenêtre sur la ligne la plus basse le caractère > pour signaler, qu'il est prêt à accepter une commande.

Le champ d'état affiche toujours l'état actuel de la cible. La barre de menus affiche quelques commandes. Vous pouvez cliquer sur un de ces boutons au lieu de taper la commande à la main. Le débogueur note les dernières commandes. Pour exécuter une commande à nouveau, tapez sur la flèche haute. Quelques commandes continuent quand on tape ENTREE. Les arguments des commandes sont des numéros en hexadécimal sans $. On peut également entrer des symboles du programme d'assemblage s'ils sont chargés.

Charger et comparer

Si le programme est assemblé sans erreur, la commande suivante charge le programme dans l'EEPROM sur la carte:

```
> load
```

Le débogueur vérifie que le programme est bien chargé sans erreur. Le débogueur charge également la table des noms symboliques dans sa propre mémoire. Pour vérifier que le programme dans l'EEPROM correspond bien au programme au P.C., tapez

```
> ver
```

Pour tester un programme qui est déjà chargé et pour obtenir la table de symboles, tapez

```
> loads
```

Entrée et sortie des données

Pour afficher l'état courant de la cible et ces registres, appuyez sur ETAT ou tapez

```
> reg
```

Vous pouvez changer le continu d'un registre, par exemple pour changer le PC à $F800, tapez

```
> reg PC F800
```

La commande D permet d'afficher des variables comme elles sont définies dans le programme de source. Cette commande accepte une forme pour trouver plusieurs variables. D * affiche les variables locales d'une fonction. D ** affiche toutes les variables. Le fichier "debug.def" contient des noms qui sont toujours connus par le débogueur. Ce fichier est cherché dans le répertoire du fichier source et ensuite dans le répertoire du programme en exécution.

```
> d minute, seconde
> d PORTA
> d PORT*
> d *
> d **
```

La commande suivante change la valeur d'une variable.

```
> PORTA = 11
```

La commande M affiche la mémoire de la cible. La commande peut être appliquée sur la totalité d'espace d'adressage. On voit donc la mémoire vive (RAM), la mémoire morte (ROM) l'EEPROM mais également les registre E/S. Les données sont affichées en hexadécimal et en ASCII.

```
> m 0
```

Affiche les premiers 64 octets de la mémoire vive. En entrant ENTREE vous voyez les 64 octets suivants.

```
> m 1000 30
```

Affiche les registres E/S de l'adresse $1000 à l'adresse $102F. Pour afficher le programme

> dis F980 ou
> dis start

Désassemble les données. Vous voyez quelques lignes des données en hexadécimal et interprété comme instruction machine.

La commande MS (memory set) permet de changer la mémoire ou également des registres E/S. La commande suivante change les registres aux adresses $1000, $1001 et $1002.

> ms 1000 = 01 23 45

Watch, Fenêtres de visualisation mémoires

La commande Watch ouvre une ligne supplémentaire entre la ligne d'état de la cible et la ligne de configuration pour afficher 16 octets de mémoire de la cible ou une variable de la cible.

> w 2000
> w TCNT
> w minute

Ces trois commandes ouvrent trois lignes de visualisation mémoires. Dans chaque ligne on a à gauche la commande à exécuter, au milieu les données de la cible et à droite un bouton pour effacer la ligne. On peut en tout moment cliquer sur la ligne pour lire la mémoire de la cible et actualiser les données. Quand le programme s'arrête par un point d'arrêt, après un pas-à-pas ou par le bouton STOP, les données sont automatiquement actualisées.

Démarrage, points d'arrêt et pas à pas

Le programme GO démarre le programme. Si vous entrez une adresse, le compteur de programme PC est mis à l'adresse avant de démarrer.

```
> go F800
```

Pour déboguer un programme, on peut mettre des points d'arrêt:

```
> br temp
```

La commande met un point d'arrêt à l'adresse TEMP. Vous pouvez même placer un point d'arrêt quand le programme tourne. Si le programme arrive au point d'arrêt, il s'arrête et donne la main au talker. Une instruction peut avoir plusieurs octets, il faut toujours mettre le point d'arrêt sur le premier. Pour continuer le programme utilisez la commande GO. La fenêtre source vous affiche les points d'arrêt. Cliquer sur la marge place ou enlève un point d'arrêt. Pour afficher tous les points d'arrêt en vigueur, tapez

```
> br
```

La commande suivante efface tous les points d'arrêt:

```
> brdel
```

Pour déboguer son programme lentement il y a le pas à pas (Trace). Si le programme ne tourne pas, on n'exécute que l'instruction machine suivante par

```
> step        ou
> next
```

NEXT se comporte comme STEP sauf dans le cas d'un saut au sous-programme. STEP entre dans le sous-programme pas à pas tant que NEXT exécute le sous-programme en totalité et s'arrête à l'instruction suivant le saut.

Si vous travailler en langage évalué Basic11 ou CC11, taper sur la touche Majuscule et sur le bouton STEP ou NEXT, exécute le programme en pas-à-pas jusqu'à ce que le programme source change de ligne, ou que vous lâchiez la touche Majuscule. Taper sur la touche Alt et sur le bouton STEP ou NEXT exécute le programme en pas-à-pas jusqu'à ce que vous lâchiez la touche Alt.

Fonctionnement du débogueur et du talker

Le débogueur sur le P.C. et le talker sur la cible, comment travaillent-ils ensemble? Le débogueur ne fait ni de simulation ni d'émulation. Le programme tourne dans son propre environnement. Cela demande une certaine collaboration du programme à tester. Pour communiquer avec le débogueur le talker est installé dans la mémoire de la cible. C'est le collaborateur du débogueur. Pour exécuter une commande par exemple M pour afficher la mémoire, le débogueur envoie une commande au talker. Celui-ci lit la mémoire et envoie le résultat au débogueur qui l'affiche. Toutes les commandes travaillent plus ou moins comme ça. Le talker est responsable d'exécuter la tâche sur place.

Le talker et le programme d'application doivent donc partager l'unique unité centrale et collaborer. Si la cible démarre, le talker prend le contrôle. Ensuite le talker passe le contrôle au programme à exécuter. Le talker est donc inactif. Si le programme atteint un point d'arrêt, il passe par une interruption le contrôle au talker. Idem, si une commande du débogueur arrive par la ligne série. Le programme ne doit en aucun cas détruire les ressources du talker. Comme le talker reçoit ses commandes du débogueur par la ligne série et donc par des interruptions, il peut travailler quasi en parallèle avec le programme d'application. Si une commande du débogueur arrive comme caractère par la ligne série, le programme est interrompu et le talker prend le relais jusqu'à que la commande soit complètement exécutée. Un programme qui ne permet pas d'interruptions ou qui change la configuration de la ligne série ou qui évite l'usage normal de la pile, ou ou ou, casse la collaboration et ne permets plus l'usage du débogueur.

Pour réaliser un point d'arrêt le débogueur met à l'adresse une instruction SWI. L'instruction déclenche une interruption pour passer le contrôle au talker. Le débogueur change par conséquent le programme à déboguer. Les commandes NEXT et STEP sont réalisés par un ou dans le cas d'une saute par deux points d'arrêt. Quand vous demandez au débogueur de démarrer votre programme, il met les points d'arrêt dans la mémoire. Quand le programme s'arrête, il les enlève. N'oubliez pas d'enlever tous points d'arrêt avant de couper la carte cible du débogueur.

Interruptions du talker

Le talker traite les interruptions

RESET (FFFE) pour démarrer le talker et le programme au RESET.

Illegal Opcode (FFF8).

SWI (FFF6) pour les points d'arrêt et les pas à pas.

SCI (FFD6) la ligne série.

Partager la ligne série entre une application et le talker.

Le talker utilise la ligne série pour la communication avec l'hôte. Néanmoins le programme peut aussi bien utiliser la ligne. Si une interruption de la ligne série arrive, l'unité centrale saute à l'adresse $00FA dans la mémoire vive. Un programme qui veut traiter toutes données de la ligne série écrit à l'adresse $00FB l'adresse de son sous-programme qui traite les interruptions. Par conséquent quand ce programme tourne, le talker est inutilisable. Autrement si le talker reçoit un caractère de la ligne série qu'il ne reconnaît pas comme commande du déboguer, il saute à l'adresse $00FD le caractère dans le registre A. Un programme qui écrit l'adresse d'un sous-programme à l'adresse $00FE peut donc traiter toutes les données qui ne sont pas dédiées au talker. Ce sont notamment tous les caractères ASCII qui sont passées au sous-programme.

```
FFD6:          00FA                    EEPROM: vecteur d'interruption SCI

00FA:          jmp     talkerInt       RAM: saute au talker

FE..  talkerint:  lit caractère        EEPROM talker
                  si caractère >$06
                        jmp 00FD
                  sinon traité par le talker

00FD           jmp     talkerend       RAM: saute au talker

FE.. talkerend:  rti                   EEPROM talker
```

Fenêtre du mode brut

La fenêtre du mode brut vous permet de communiquer directement avec votre programme si votre programme reçoit ou envoie des caractères par la ligne série. Vous pouvez choisir entre le mode ASCII et le mode HEXADECIMAL pour la présentation des caractères.

Déboguer une carte cible à distance

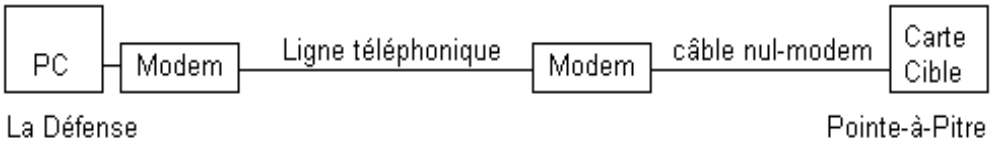
Si vous êtes à la Défense et la carte cible doit tourner à Pointe-à-Pitre, inutile de se déplacer pour chaque changement du programme ou pour récupérer les données enregistrées. Il vous faut une ligne téléphonique et deux modems pour communiquer avec une carte cible à distance.

A Pointe-à-Pitre:

Il faut mettre le modem en mode réponse automatique. Il prend la ligne au deuxième caractère de sonnerie(S0=2). Il faut également enlever le contrôle de flux(&D0&K0). On enregistre ces paramètres dans la mémoire non volatile du modem et sélectionne cette configuration à utiliser après la mise sous tension(&Y0&W0). Connectez le modem de la cible sur le port série du P.C., normalement prévu pour la communication avec la carte cible. Choisissez la boîte de dialogue modem et cliquez sur Modem Cible pour envoyer la commande au modem.

ATS0=2&D0&K0&Y0&W0

Le modem doit répondre dans la grande fenêtre an bas en gauche avec OK. Ajoutez Q1 dans la ligne pour désactiver des messages du modem vers la cible. Mais vous n'avez plus d'OK comme confirmation. Il vous faut également un nul modem entre le modem - connecteur femelle - et la cible - aussi connecteur femelle. Connectez la cible par le nul modem au modem et le modem a la ligne téléphonique. Allumez la carte cible et le modem. Assurez-vous que la cible tourne bien. Prenez l'avion direction Paris.



Retour à la Défense

Connectez le modem au port série que vous avez choisi pour communiquer avec la carte cible. Augmentez dans la boîte de dialogue configuration le communication timeout. Choisissez la boîte de dialogue modem et entrez le numéro de téléphone à Pointe-à-Pitre dans la place qui est prévu pour ce numéro(**********). Le mode de transmission de données est le mode de liaison directe. (\N1). Cliquez sur Enregistrer pour sauvegarder le numéro. Cliquez sur Connecter pour établir la connexion avec la carte cible. Attendez que le modem affiche la communication établie dans la fenêtre en bas à gauche. Si vous travaillez en mode prototypage rapide, cliquez sur la barre qui affiche l'état pour récupérer l'état actuel de la cible. Si vous travaillez en assembleur, cliquez sur Etat dans le menu du débogueur. Vous pouvez maintenant travailler avec la carte cible à distance.

BRDEL — Effacement de point d'arrêt — BRDEL

BRD[EL]
BRD[EL] <Adresse>

Efface un ou tous les points d'arrêt.

Pour effectuer cette commande, la cible doit être dans l'état STOP.

> brd temp	Efface le point d'arrêt à l'adresse *temp*
> brdel	Efface tous les points d'arrêt.

BREAK — Points d'arrêt — BREAK

BR[EAK]
BR[EAK] <Adresse>

La première commande sans adresse affiche les points d'arrêt en vigueur. La commande avec une adresse met un point d'arrêt à l'adresse indiquée. Si le programme atteint l'adresse il passe le contrôle au talker.

Pour effectuer cette commande, le talker doit être capable de lire la commande de la ligne série.

> br	Affiche tous les points d'arrêt
> br temp	Met un point d'arrêt à l'adresse TEMP

D — Affichage des variables — D

D <Variable> [,<Variable>]*
D <forme>*
D *
D **

Lit la mémoire et affiche le continu des variables du programme. D * affiche les variables locales d'une fonction. D ** affiche toutes les variables.

Pour effectuer cette commande, le talker doit être capable de lire la commande de la ligne série.

> d minute, seconde	Affiche les variables dans la RAM

DIS — Désassemblage — DIS

DIS
DIS **<Adresse>**
DIS **<Adresse> <Longueur>**

Lit la mémoire à l'adresse et affiche les données en instruction machine. En entrant ENTREE la commande continue l'affiche.

Pour effectuer cette commande, le talker doit être capable de lire la commande de la ligne série.

> dis loop — Affiche des instructions à partir de l'adresse

GO — Démarrage du programme — GO

G[O]
G[O] **<Adresse>**

Si l'adresse est indiquée met le PC à l'adresse. Ensuite démarre le programme ou continue le programme.

Pour effectuer cette commande, la cible doit être dans l'état STOP.

> go start — Démarre le programme
> g — Continue le programme après un point d'arrêt

INITTALKER — Initialisation du talker — INITTALKER

INIT
INITTALKER

Initialise le talker, si il est accidentellement endomagé. La commande efface l'EEPROM en totalité.

> inittalker — Initialise le talker

LOAD — Chargement du programme — LOAD

L[OAD]

Charge le programme dans l'EEPROM de la cible. Le programme doit être assemblé sans erreur. La commande lit le fichier <nom>.S19. La commande charge le programme dans l'EEPROM et vérifie la programmation correcte. Le débogueur charge la table des noms symboliques dans sa propre mémoire. Le PC est mis à $F800, le SP est mis à $00E8. Tous points d'arrêt sont effacés. La commande charge sur les adresses $F800 à $FE7F et $FFD6 à $FFFF. Il refuse le chargement sur des autres adresses.

Pour effectuer cette commande, la cible doit être dans l'état STOP.

> load	Charge le programme
> ver	Compare le programme avec le fichier .S19

LOADS — Chargement de la table de symboles — LOADS

LOADS

Charge la table des noms symboliques dans la mémoire du débogueur pour déboguer un programme qui tourne déjà.

Cette commande peut toujours être effectuée indépendamment de l'état de la cible. Si le talker est dans l'état STOP, la commande VER est préférable.

> loads	Charge la table de symboles dans le débogueur.
> m tbuf tbufp	Affiche les données de la mémoire vive.

LOG — Journal — LOG

LOG
LOG ON
LOG <Nom de fichier>
LOG ON <Nom de fichier>
LOG OFF

Active ou désactive le journal. Le journal contient toutes les commandes entrées et toutes les données affichées. Si aucun nom est indiqué le journal est écrit dans le fichier CBOY.LOG. Pour désactiver le journal, entrez LOG OFF.

Cette commande peut toujours être effectuée indépendamment de l'état de la cible.

> log	Ecrit le journal dans le fichier CBOY.LOG
> log off	Finit le journal

MEM

Affichage de la mémoire

MEM

M[EM]
M[EM] **<Adresse>**
M[EM] **<Adresse> <Longueur>**

Lit la mémoire à l'adresse et affiche les données en hexadécimal et en caractères ASCII. En entrant ENTREE la commande continue l'affiche.

Pour effectuer cette commande, le talker doit être capable de lire la commande de la ligne série.

> m 0	Affiche 64 octets à l'adresse 0
>	Et les 64 octets suivantes
> m 1000 20	Affiche des registres E/S

MFILL

Remplissage de la mémoire

MFILL

MF[ILL] **<Adresse> <Longueur> [=] <Octet>**

Remplit la mémoire à l'adresse avec des octets. Le deuxième paramètre précise le numéro des octets à remplir ou l'adresse finale.

Pour effectuer cette commande, le talker doit être capable de lire la commande de la ligne série.

> mf 0 E8 = 00	Efface la mémoire vive
> mf F800 FA00 FF	Efface l'EEPROM

MSET

Ecriture mémoire

MSET

MS[ET] **<Adresse> [=] <Octet> [<Octet>]***
MS[ET] **-W <Adresse> [=] <Mot> [<Mot>]***

Ecrit des octets ou des mots de 16 bits dans la mémoire.

Pour effectuer cette commande, le talker doit être capable de lire la commande de la ligne série.

> ms 102C 33	Ecrit $33 dans le registre E/S
> ms -w tbufp = tbuf	Initialise le pointeur tbufp avec la valeur tbuf

NEXT

Pas à Pas

NEXT

NEXT
N

Exécute une instruction du programme et arrête le programme après l'exécution. Si l'instruction et un saut au sous-programme (BSR ou JSR) le sous-programme est exécuté avant l'arrêt du programme. En entrant ENTREE la commande continue l'opération.
Si vous travailler en langage évalué Basic11 ou CC11, taper sur la touche Majuscule et sur le bouton NEXT exécute le programme en pas-à-pas jusqu'à ce que le programme source change de ligne, ou que vous lâchiez la touche Majuscule. Taper sur la touche Alt et sur le bouton NEXT exécute le programme en pas-à-pas jusqu'à ce que vous lâchiez la touche Alt.

Pour effectuer cette commande, la cible doit être dans l'état STOP.

> n	Exécute une instruction du programme
>	Et encore une

REG

Registres

REG

R[EG]
R[EG] A | B | CC | PC | SP | IX | IY [=] <Valeur>

La commande sans arguments affiche l'état et les registres de la cible. Si la cible se trouve dans l'état STOP, la commande affiche les registres sauvés. Sinon, demande au talker l'état et les registres actuels.

La commande avec des arguments change le contenu du registre. La cible doit être dans l'état STOP.

> reg	Obtient et affiche l'état de la cible
> reg PC = start	Change le registre PC

STEP

Pas à Pas

STEP

STEP
S

Exécute une instruction du programme et arrête le programme après l'exécution. Si l'instruction et un saut au sous-programme (BSR ou JSR), entre dans le sous-programme. En entrant ENTREE la commande continue l'opération.
Si vous travailler en langage évalué Basic11 ou CC11, taper sur la touche Majuscule et sur le bouton STEP exécute le programme en pas-à-pas jusqu'à ce que le programme source change de ligne, ou que vous lâchiez la touche Majuscule. Taper sur la touche Alt et sur le bouton STEP exécute le programme en pas-à-pas jusqu'à ce que vous lâchiez la touche Alt.

Pour effectuer cette commande, la cible doit être dans l'état STOP.

> s	Exécute une instruction du programme
>	Et encore une

STOP — Arrêt du programme — STOP

STOP

Arrête le programme en exécution. La commande envoie une commande au talker pour arrêter le programme. Le programme s'arrête et peut être continué par GO, NEXT ou STEP.

Pour effectuer cette commande, la cible doit être en exécution et le talker doit être capable de lire la commande de la ligne série

> stop Arrête le programme en exécution

UPLOAD — Enregistrer le programme dans un fichier — UPLOAD

UPLOAD <Adresse> <Longueur> [<nom de fichier>]

Lit la mémoire de la cible et enregistre les données dans un fichier en format Motorola S-record. La première fois il faut donner le nom du fichier. Des autres Uploads sans nom de fichier sont ajoutés au fichier.

Pour effectuer cette commande, le talker doit être capable de lire la commande de la ligne série.

> upload F800 F900 up.s19 enregistre le programme dans le fichier up.s19

VER — Vérification du programme — VER

V[ER]

Compare le programme dans l'EEPROM avec celui dans le fichier. La commande lit le fichier <nom>.S19. et charge la table des noms symboliques dans la mémoire du débogueur.

Pour effectuer cette commande, la cible doit être dans l'état STOP.

> ver Compare le programme avec le fichier

WATCH — Fenêtres de visualisation mémoires — WATCH

W[ATCH] <Adresse>
W[ATCH] <Variable>

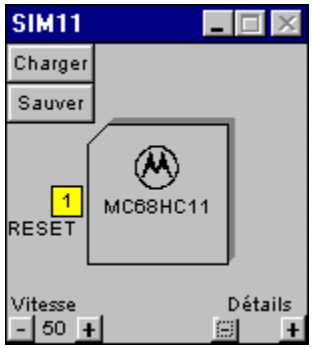
Ouvrir une ligne supplémentaire pour afficher la mémoire de la cible.

> w 2000
> w minute

Dans chaque ligne on a à gauche la commande à exécuter, au milieu les données de la cible et à droite un bouton pour effacer la ligne. Cliquer sur la ligne actualise les données. Quand le programme s'arrête, les données sont automatiquement actualisées.

Simulateur

Le simulateur permet d'exécuter un programme pour un 68HC11 sur un microcontrôleur virtuel.
Pour remplacer la carte cible à base d'un 68HC11 par le simulateur, il suffit de sélectionner dans la configuration le port SIM au lieu d'un port réel COM.
Le débogueur va donc communiquer avec le simulateur au lieu de communiquer avec une cible réelle.
Le microcontrôleur virtuel se présente comme suivant sur votre écran:



Le microcontrôleur virtuel se comporte comme un microcontrôleur réel.
Il faut charger un talker dans la mémoire du microcontrôleur pour que le débogueur puisse communiquer avec lui pour charger et déboguer votre programme.
Mais vous avez de la chance, le microcontrôleur est livré avec un talker déjà chargé en usine.
Vous pouvez le remplacer par un talker de votre choix à tout moment à l'aide de la commande INITTALKER du débogueur.
Le talker chargé, le débogueur peut communiquer avec le microcontrôleur virtuel par l'interface série virtuelle comme avec un 68HC11 réel.

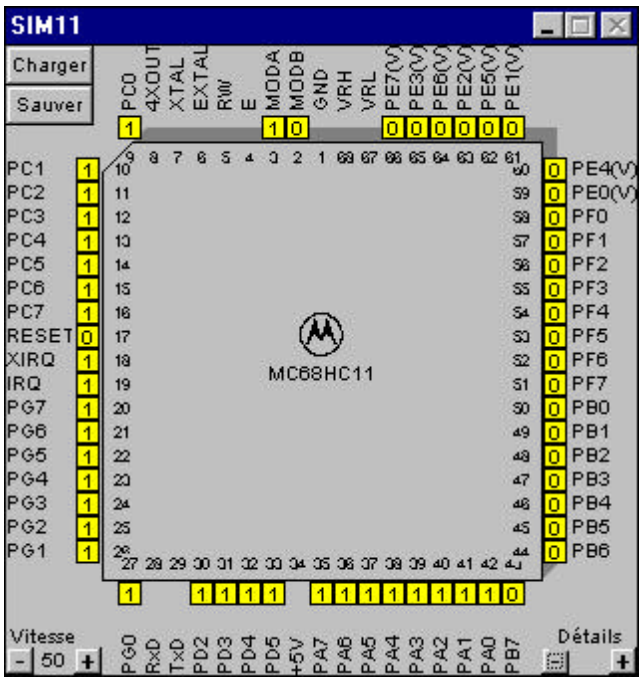
Le **bouton jaune** RESET correspond à l'entrée RESET du microcontrôleur. Cliquez sur le bouton pour changer l'état de l'entrée.

Le microcontrôleur virtuel tourne tout le temps comme un microcontrôleur réel.
Les boutons **Vitesse** permettent d'accélérer ou de ralentir la vitesse du microcontrôleur.
Si vous choisissez une vitesse de 50%, le simulateur prend la moitié de temps de votre P.C. et laisse l'autre moitié pour les autres applications qui tournent sur votre ordinateur.

Le bouton **Sauver** permet d'enregistrer dans un fichier l'état du microcontrôleur. L'état comprend la mémoire, l'unité central, et les entrées et sorties. On recharge l'état avec le bouton **Charger**.

Quand on lance le simulateur, il cherche le fichier **default.sim**. Il cherche ce fichier dans le répertoire de travail, et ensuite dans le répertoire des exécutables BIN.
S'il trouve ce fichier, le simulateur charge l'état qui se trouve dans le fichier.
L'installation du logiciel met un fichier default.sim dans le répertoire des exécutables. Ce fichier contient un talker pour le microcontrôleur virtuel.

Les boutons **Détails** permettent de voir plus ou moins de détails du microcontrôleur.

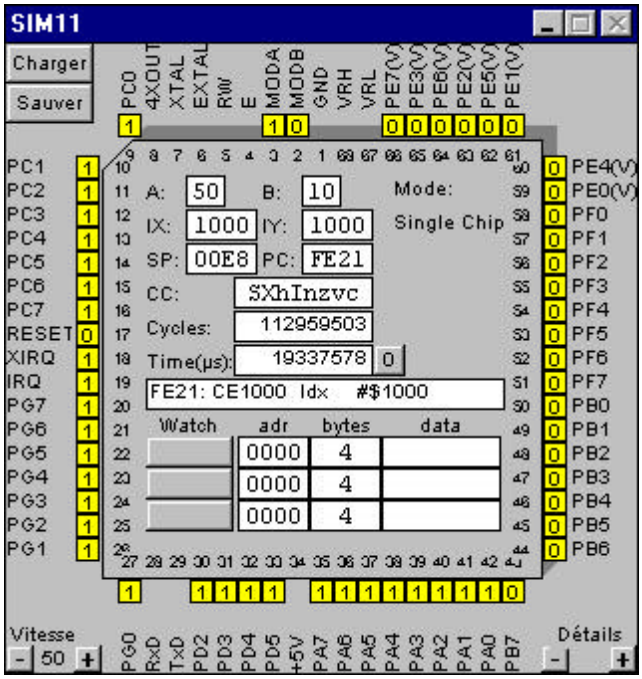


Avec cette fenêtre vous avez accès aux entrées et aux sorties du microcontrôleur.
En cliquant sur une entrée numérique, on bascule son état.
On peut régler de la même manière une entrée analogique de 0 à 5 Volt, parce que si on dépasse le 5V, on risque d'abîmer le convertisseur analogique numérique. Il est donc protégé.
Quand le programme écrit sur une sortie, son état change sur la fenêtre.
Les ports B, C et F (sur un 68HC11F1) ne sont pas disponibles en mode étendu.
Les entrées MODA, MODB, RESET permettent de choisir le mode du 68HC11 et de mettre le microcontrôleur à zéro.
Les entrées IRQ et XIRQ peuvent déclencher des interruptions.
Les ports PD0 et PD1 sont pris pour communiquer avec le débogueur. Votre programme ne peut pas les utiliser, mais il peut envoyer des données par l'interface série et recevoir des caractères.
(Voir débogueur, interruptions du talker)

Exemple

Sélectionnez dans la configuration le port SIM et la carte Controlboy F1. Ouvrez le fichier CLIN.BAS ou le fichier CLIN.C selon votre compilateur préféré. Compilez le fichier. Chargez le programme dans la cible à l'aide de débogueur. Cliquez sur GO pour lancer le programme. Le programme fait clignotez une DEL virtuel à la sortie PG0. Vous voyez cette sortie basculer régulièrement. Si vous mettez l'entrée PG1 à 0, la sortie PG0 bascule plus rapidement.

Les boutons Détails permettent même d'ouvrir tout délicatement le capot du 68HC11, ce qui est déconseillé pour un microcontrôleur réel. On entre donc dans la microchirurgie.



Si le microcontrôleur n'est pas en état RESET, et vous n'avez pas choisi une vitesse zéro, vous voyez bien, que le microcontrôleur tourne toujours. S'il n'exécute pas un programme d'application, il exécute le talker, qui attend une commande par l'interface série.
Vous voyez les registres de l'unité centrale, le mode actuel du microcontrôleur, l'instruction, que l'unité central va exécuter prochainement.
La fenêtre vous affiche aussi les cycles, que l'unité central a exécuté, et le temps virtuel écoulé du microcontrôleur selon le quartz que vous avez choisi. Vous pouvez mettre ces valeurs à zéro.

Si vous choisissez la vitesse 0, le programme s'arrête. Vous pouvez exécuter le programme pas à pas.
Il ne faut pas confondre ce mode pas-à-pas avec le pas-à-pas du débogueur.
Ouvrir le capot et arrêter l'unité centrale est seulement possible avec un microcontrôleur virtuel.

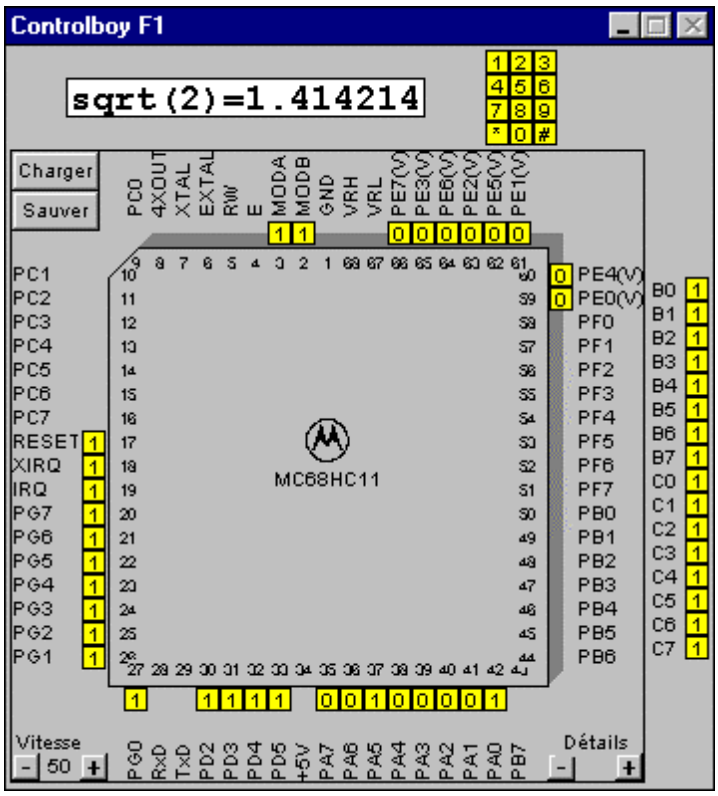
Les lignes **watch** permet d'afficher la mémoire et d'arrêter le microcontrôleur sur une condition.
Les champs adr et bytes sélectionnent une zone de mémoire.
En cliquant plusieurs fois sur le bouton gauche vous pouvez choisir par ligne:

Watch	Afficher la mémoire (4 octets maximum)
Break	Arrêter l'unité centrale, quand le PC entre dans la zone.
Stop Rd	Arrêter quand l'unité centrale a lu dans la zone.
Stop Wr	Arrêter quand l'unité centrale a écrit dans la zone, ou si c'est une entrée, quand l'entrée a changé son état.
Stop RW	StopRd et StopWr.

Pour profiter pleinement d'un microcontrôleur, on le place sur une **circuit imprimée** entourée des autres composants.
Le logiciel fait appel au fichier **board.dll** qui se trouve dans le répertoire BIN.
Le logiciel est livré avec deux cartes:

board0.dll	68HC11 seul
boardcf1.dll	Controlboy F1 avec afficheur LCD et clavier 12 touches

Pour changer la carte il faut copier un de ces fichiers sur board.dll et redémarrer le logiciel.
Voici l'exemple du 68HC11 sur la carte Controlboy F1.



On peut créer son propre fichier board.dll. On trouve les sources des fichiers dans le répertoire BOARDSRC.
Il vous faut en plus une chaîne de compilation pour PC (non fourni).
Le programme doit fournir les fonctions suivantes

LibMain	Ouvrir le DLL
WEP	Fermer le DLL
BoardOpen()	Ouvrir ou fermer la simulation
BoardRead()	Lire un octet sur la carte
BoardWrite()	Ecrire un octet sur la carte

Le DLL peut envoyer un message SIM_PORTCHANGED au simulateur pour mettre les entrées et la mémoire du 68HC11 à jour.

Réalisation

Boîtiers
MC68HC11A0, A1, E0, E1, MC68HC811E2 tous en PLCC.
MC68HC11F1FN en PLCC.
DIL non réalisé.

Modes
Boot, single chip, étendu, selon l'état de MODA et MODB au RESET.
Mode Test non réalisé.
Registres OPTION, BPROT, HPRIO, INIT, CONFIG non réalisés.

Mémoire
64k RAM, non protégée en tous modes.
La ROM du 68HC11A8 sans sécurité apparaît en mode BOOT de l'adresse BF40 à l'adresse BFFF.
(Motorola M68HC11 Reference Manual Rev 3, B-2)

Resets et interruptions

Reset	externe réalisé
XIRQ	réalisé
IRQ	réalisé, mode niveau bas
Illegal Opcode	réalisé, utilisé par le talker
SWI	réalisé, utilisé par le talker
SCI(interface série)	réalisé pour la réception, utilisé par le talker
Real Timer	réalisé
les autres	non réalisé

Unité centrale
Les registres et les instructions sont toutes réalisées selon le Motorola M68HC11 Reference Manual Rev 3, chapitres 6 et A.

Entrées, sorties parallèles.
Réalisé pour les ports A, B, C, D, E, F, G avec leurs registres de direction des données.
Port B, C handshake non réalisé. Port G chip-select non réalise.

Interface serial asynchrone SCI
Réalisé en partie pour communiquer avec le débogueur.

Entrées analogiques, convertisseur analogique numérique.
Réalisé pour le registre ADR1 seulement.

Real Timer
realisé.

Interface serial synchrone SPI, Timer, Pulse Accumulator
Non réalisé.

Les réalisations peuvent être quelque fois un peu naïves.

Programmation en Prototypage Rapide

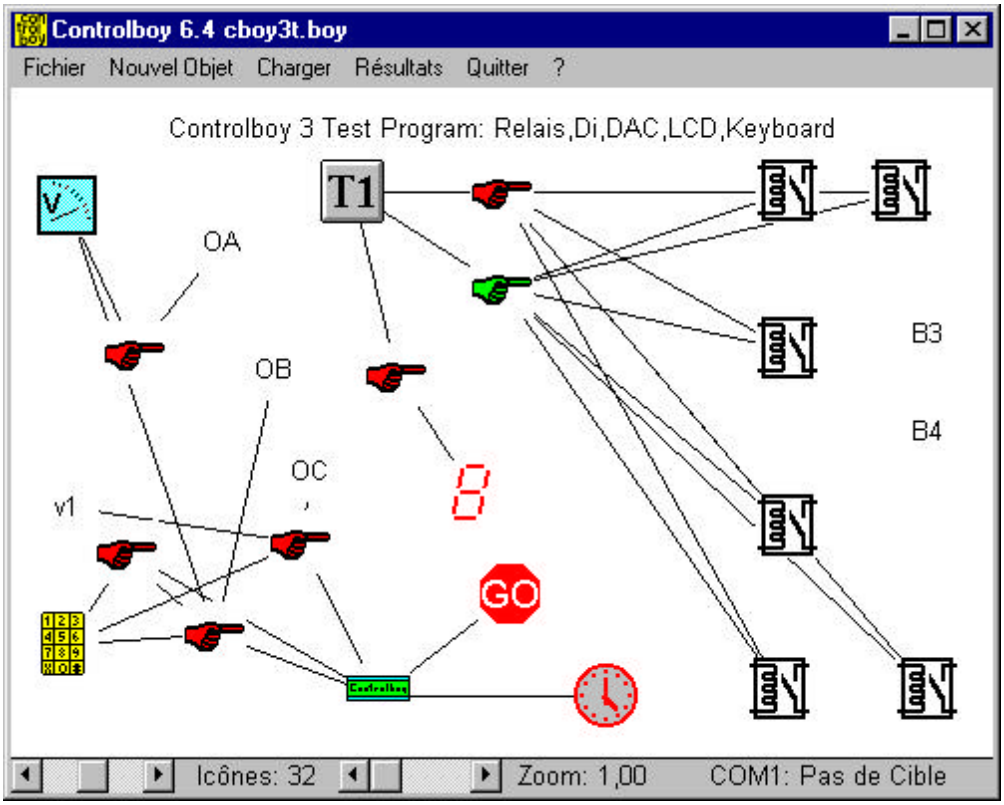
Fenêtre du projet, objets
Fichiers
Imprimer
Entrées
Calibrer
Enregistrement des données
Sorties
Servos
Variables en mémoire vive
Actions
Conditions
Opérations
Chargement
Effacer les données
Régler l'heure
Ressources
Résultats
Insérer un Programme Assembleur dans le Prototypage Rapide

La programmation se fait sous Windows sur une surface graphique. Le générateur de programme n'est certainement pas aussi universel qu'un langage de programmation. La programmation est limitée aux projets faciles de mesure et de réglage. Il est néanmois étonnant de voir le nombre de possibilités du générateur.

Pour ceux qui ne lisent jamais une instruction

Cliquez sur FICHIER. Cliquez sur DEMO. Et faites vous démontrer l'utilisation.

Fenêtre du projet, objets



Vous voyez ici la fenêtre du générateur de programme. Elle correspond à un projet ou un programme. Chaque icône représente un objet. Il y a les objets suivants:

Entrées:
Des objets d'entrée représentent des entrées de la cible. Ils se trouvent dans la fenêtre partie gauche.

Sorties:
Des objets de sortie représentent des sorties de la cible. Il se trouve dans la fenêtre partie droite.

Variables en mémoire vive:
Des Variables en mémoire vive représentent des octets dans la mémoire RAM. Elles sont utilisées pour gérer des informations intérimaires ou pour des tâches plus complexes.

Actions:
Des actions représentent des activités à exécuter par la cible. Des actions sont déclenchées par des événements, par exemple, quand la température franchit 25°. Une action change une sortie ou une variable en mémoire vive. Les actions sont liées par des lignes dans la fenêtre à tous les objets qu'elles contrôlent ou dont elles dépendent.

Pour examiner ou changer un objet, cliquez deux fois sur son icône.

Pour déplacer un objet, cliquez sur l'icône et faites glisser le pointeur.

Fichiers

Un projet ou un programme est géré dans un fichier sur le disque dur. Ce fichier a pour extension .BOY. Cliquez dans la fenêtre du générateur de programme sur FICHIER et vous avez avec

NOUVEAU
OUVRIR
ENREGISTRER
ENREGISTRER SOUS

tous les outils pour ouvrir et enregistrer des fichiers. Vous pouvez garder plusieurs programmes sur votre disque. Bien entendu, sur une cible ne tourne qu'un seul programme.

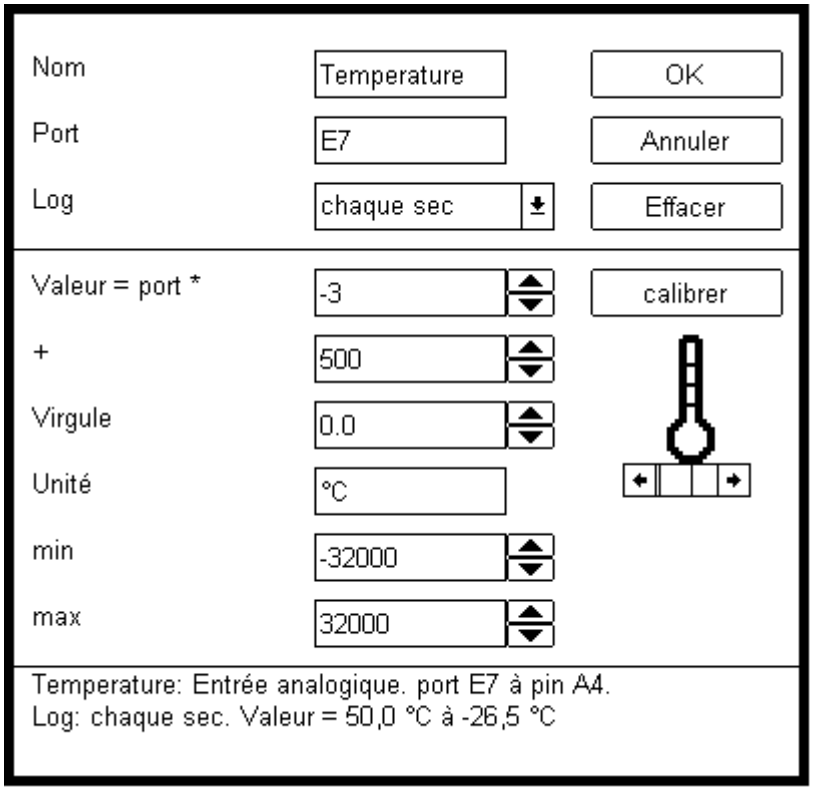
Imprimer

Cliquez sur FICHIER, ensuite sur IMPRIMER

PROJET: Votre programme sera imprimé comme il est affiché dans la fenêtre. Avant d'envoyer les données à l'imprimante, vous pouvez par CONFIGURATION choisir entre plusieurs tailles de papier et entre deux orientations.

PROJET EN MOTS: Votre programme sera enregistré dans un fichier. Après on vous demande, s'il faut imprimer le fichier. Vous pouvez également traiter le fichier avec un éditeur de texte avant l'impression.

Entrées



Un objet d'entrée précise une entrée de la cible. Une entrée comprend les éléments suivants:

Nom: chaque objet est identifié par son nom unique. Donnez-lui un nom de votre choix.

Port: L'entrée du microprocesseur. Quand vous définissez une entrée nouvelle, vous pouvez choisir dans la liste de ports libres.

Log: Précisez, comme l'entrée sera notée.

Valeur, virgule, unité, min, max: précisent la transformation de la valeur brute à la valeur qui s'affiche dans les résultats et sur l'afficheur.

Valeur brute est la valeur (0..255) à l'entrée. Les calculs sont effectués pour transformer la valeur brut à la valeur réelle (par exemple -40..70 °C).

Calibrer: Cette touche vous offre une fenêtre qui affiche constamment la valeur brute et la valeur réelle. Voir chapitre Calibrer.

Calibrer

Pour calculer des valeurs réelles à base des valeur brutes (0..255) il faut choisir la zone d'entrée (par exemple température intérieure 15° à 25 °C). On règle les circuits extérieurs de la cible pour profiter le plus possible du convertisseur analogique/ numérique (p.e. 15° C: entrée=0,5V CAN=25 et 25°C: entrée=4,5V CAN=230).

Après cliquez sur CALIBRER. Il faut mesurer deux valeurs brutes qui sont les plus éloignées que possible (p.e. 15° et 25°). Stockez la première valeur en cliquant sur X1 et la deuxième en cliquant sur X2. Vous pouvez également entrer les valeurs à la main. Précisez la virgule. Dans notre cas une présentation de 0,0 est conseillée. Calibrez maintenant les valeurs qui correspondent aux valeurs brutes par cliquant sur les boutons à droite de X1 et X2. Commencez avec X1. Quand la valeur est juste, calibrez X2. Toutes les valeurs ne sont pas possibles. Affinez les valeurs avec X1.

Enregistrement des données

Le champ Log décide quand et à quelle vitesse la cible enregistra des données. On peut noter des entrées et des variables en mémoire. Les données sont enregistrées dans la mémoire vive (RAM) pendant la journée est sont écrites dans l'EEPROM une fois par jour à minuit.

Chaque Seconde: Les données sont enregistrées dans la RAM. Chaque minute la valeur moyenne des 60 secondes écoulées est enregistrée dans la RAM. Chaque heure la valeur moyenne des 60 minutes écoulées est enregistrée dans la RAM. A minuit la valeur moyenne des 24 heures écoulées est enregistrée dans l'EEPROM. A la fin du mois la valeur moyenne des 30 jours écoulés est enregistrée dans l'EEPROM. Vous pouvez visualiser comme résultats les 60 dernières secondes, 60 dernières minutes, 24 dernières heures, 30 derniers jours et 24 derniers mois.

Chaque Minute: Les données sont enregistrées dans la RAM. Chaque heure la valeur moyenne des 60 minutes écoulées est enregistrée dans la RAM. A minuit la valeur moyenne des 24 heures écoulées est enregistrée dans l'EEPROM. A la fin du mois la valeur moyenne des 30 jours écoulés est enregistrée dans l'EEPROM. Vous pouvez visualiser comme résultats les 60 dernières minutes, 24 dernières heures, 30 derniers jours et 24 derniers mois.

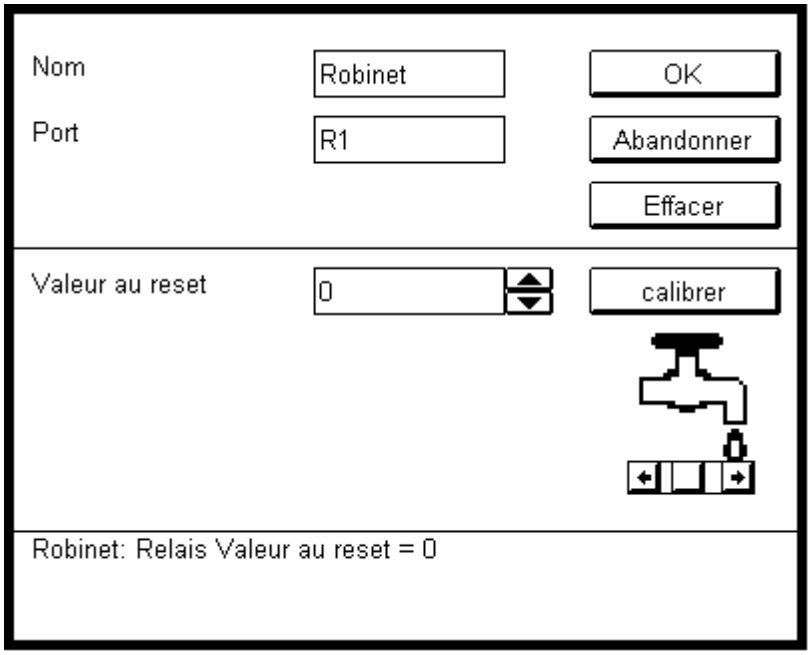
Chaque Heure: Si c'est une sortie, les données sont enregistrées chaque minute et chaque heure la valeur moyenne des 60 minutes écoulées est enregistrée dans la RAM. Si c'est une variable en mémoire, les données sont enregistrées chaque heure dans la RAM. A minuit la valeur moyenne des 24 heures écoulées est enregistrée dans l'EEPROM. A la fin du mois la valeur moyenne des 30 jours écoulés est enregistrée dans l'EEPROM. Vous pouvez visualiser comme résultats les 24 heures, 30 derniers jours et 24 derniers mois.

Chaque Jour: Si c'est une sortie, les données sont enregistrées toutes les 10 minutes et à minuit la valeur moyenne des données est enregistrée dans l'EEPROM. Si c'est une variable en mémoire, les données sont enregistrées à minuit dans l'EEPROM. A la fin du mois la valeur moyenne des 30 jours écoulés est enregistrée dans l'EEPROM. Vous pouvez visualiser comme résultats les 30 derniers jours et 24 derniers mois.

Log =	Base	Résultats	RAM octets	EEPROM octets
Aucun			0	0
Chaque seconde	Chaque seconde	60 sec, 60 min, 24 h, 30 jours, 24 mois	144	54
Chaque minute	Chaque minute	60 min, 24 heures, 30 jours, 24 mois	84	54
Chaque heure	Chaque minute	24 heures, 30 jours, 24 mois	26	54
Chaque jour	Toutes les 10 minutes	30 jours, 24 mois	2	54

On peut changer les paramètres de l'enregistrement. Cliquez sur FICHIER, ensuite sur LOG.

Sorties



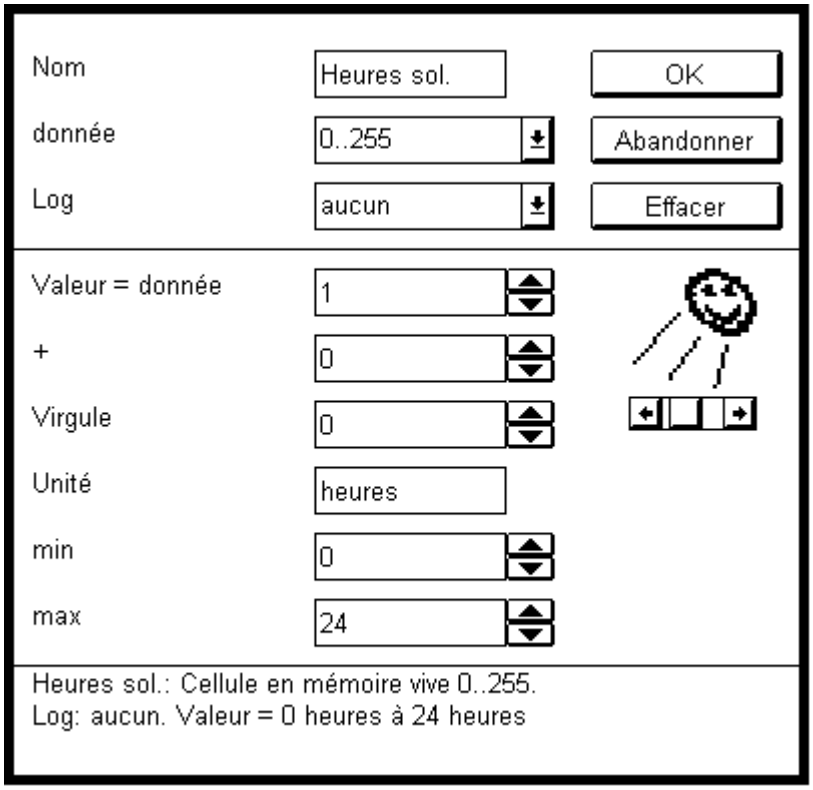
Un objet de sortie précise une sortie de la cible. Une sortie comprend les éléments suivants:

Nom, Port: comme déjà décrit dans le chapitre Entrées.
Valeur au reset: La valeur sera écrite dans la sortie quand le programme démarre.

Servos

Une sortie peut contrôler un servo à l'impulsion positive (compatible Futaba). La sortie attend des valeurs entre 0 et 200. Le programme les transfère en impulsions pour régler le servo.

Variables en mémoire vive



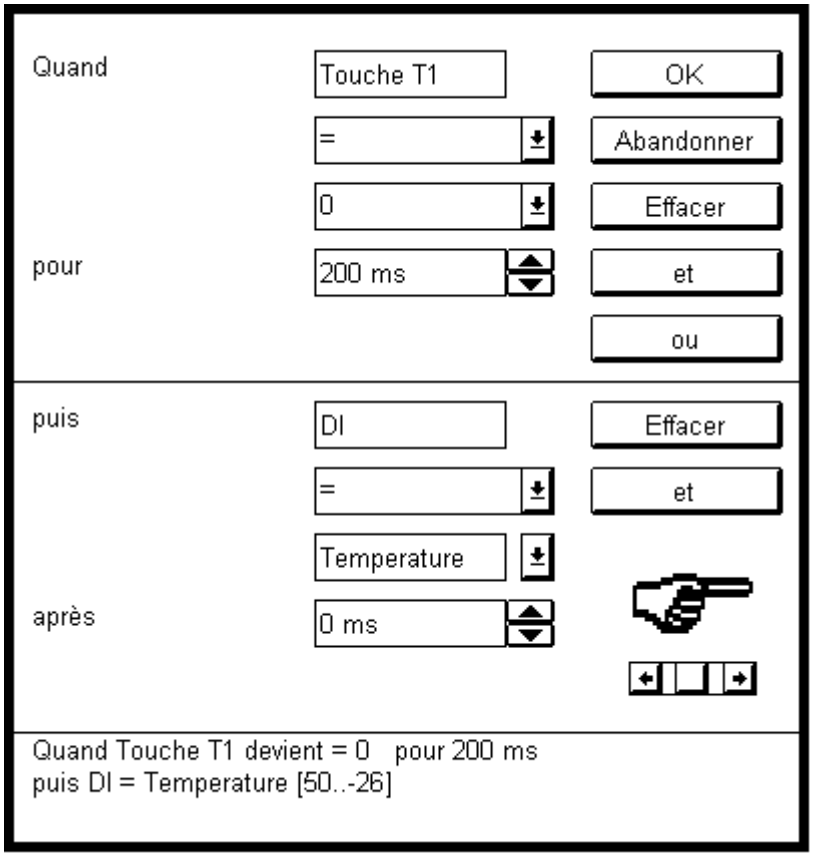
Des Variables en mémoire vive représentent des octets dans la mémoire RAM de la cible. Elles sont utilisées pour gérer des informations intérimaires ou pour des tâches plus complexes. Une variable a un ou deux octets.

Donnée	RAM octets
0..255	1
0..65535	2

Divisé par: Les enregistrements sont limités à 0..255. Une variable qui prend des valeurs 0..65535 doit être divisé avant l'enregistrement pour que la valeur ne franchisse pas 255. Par exemple: Votre programme additionne chaque minute la température à une variable. Chaque heure le programme enregistre la variable divisée par 60, donc la valeur moyenne.

Les autres champs sont déjà expliqués dans le chapitre Entrées.

Actions



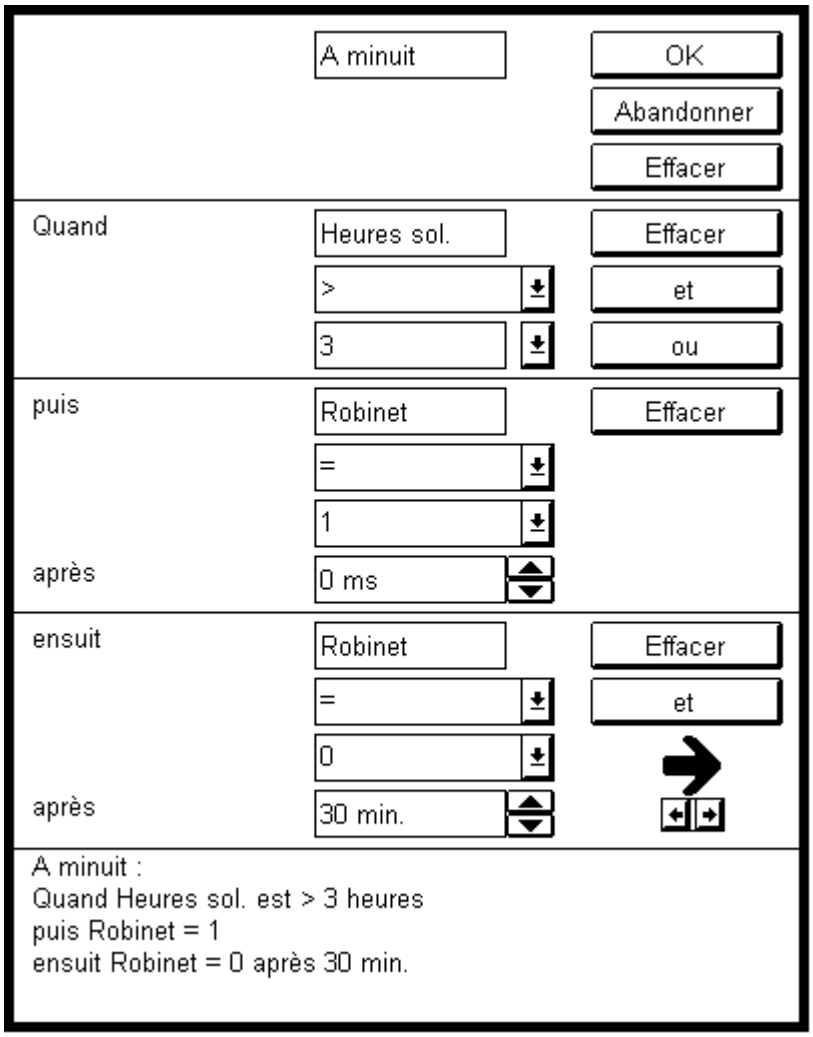
Une action représente une activité à exécuter par le programme. Elle est divisée entre
une ou plusieurs conditions, et
une ou plusieurs opérations.

Conditions

Les conditions précisent QUAND l'action sera déclenchée, par exemple quand une sortie franchit une certaine valeur ou quand une touche est pressée. La première condition peut être régulière, par exemple CHAQUE HEURE, sinon la condition est toujours testée. Dont des conditions qui sont liées par un ET, la première déclenche l'action. Les conditions suivantes sont des clauses complémentaires, qui doivent être remplies pour exécuter les opérations. Dont des conditions qui sont liées par OU, chacune peut déclencher les opérations. Dans le champ du texte vérifier bien, comme une condition est testée. C'est affiché par DEVIENT ou EST.

Toutes les entrées, les variables en mémoire vive et des informations du système comme le temps peuvent être testées et comparées.

Pour: Précise le temps que la condition doit être remplie avant de déclencher les opérations. Ce temps permet à filtrer des parasites.



Opérations

Les opérations précisent, CE QUI va se passer, quand les conditions sont accomplies. Toutes les sorties et les variables en mémoire vive peuvent être changées. Le programme peut leur donner une valeur directe ou les augmenter ou les diminuer par une valeur. La valeur est soit une constante soit une entrée soit une variable en mémoire. Vous pouvez entrer plusieurs opérations qui seront exécutées indépendamment.

Après: Précise le temps après lequel le programme exécute l'opération. Avec deux opérations on peut coller un relais et l'éteindre quelque temps plus tard.

Chargement

Quand le programme est achevé, il faut le charger dans la cible. Cliquez sur CHARGER, ensuite CHARGER DANS LA CIBLE. Le programme est généré, traduit en langage machine, et finalement chargé par la liaison série dans la mémoire EEPROM de la cible. Attendez bien la fin de l'opération. Votre projet est achevé. Vous pouvez déconnecter la cible de l'ordinateur.

Effacer les données

Quand vous chargez un nouveau programme, il est conseillé d'effacer toutes les données qui se trouvent encore dans la mémoire de la cible pour éviter des résultats farfelus. Cliquez sur CHARGER, ensuite EFFACER TOUTES DONNEES.

Régler l'heure

La cible garde l'heure dans sa mémoire. Pour régler l'heure, cliquez sur CHARGER, ensuite REGLER L'HEURE. Cliquez sur la touche du milieu pour transférer l'heure du P.C. dans la mémoire de la cible.

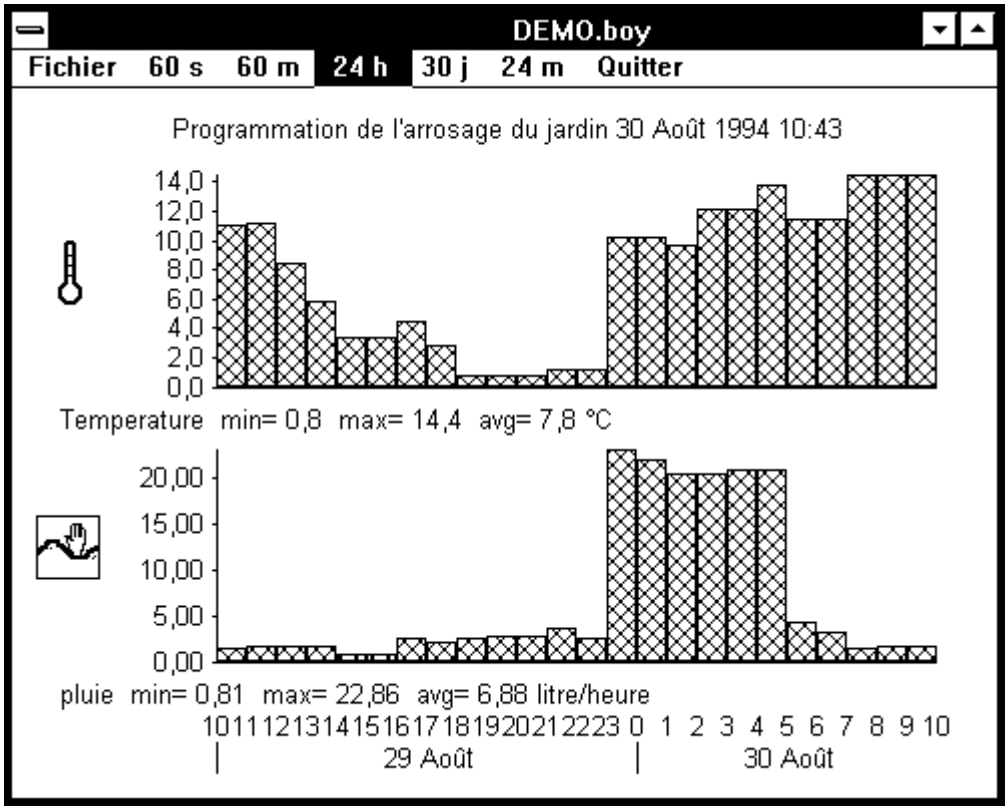
Heure d'été: Le programme dans la cible ne connaît pas l'heure d'été. Si vous changez l'heure de votre ordinateur pendant la période d'heure d'été, ce bouton doit être sélectionné pendant cette période. Le programme calcule automatiquement la différence de temps.

Vitesse: Quand vous observez après quelque temps que l'horloge de la cible ne marche pas correctement, vous avez la possibilité de régler la vitesse d'horloge. Chaque fois à minuit l'horloge est décalée par le temps indiqué.

Ressources

Cliquez sur CHARGER, ensuite RESSOURCES. Le programme vous affiche combien de mémoire RAM et EEPROM est déjà prise, et combien il en reste. Quand le montant de mémoire qui reste est négatif, le programme n'entre pas dans la mémoire de la cible.

Résultats



Quand vous connectez la cible à votre P.C., vous pouvez examiner les données enregistrées. Cliquez dans la fenêtre de programmation sur RESULTATS. Vous voyez maintenant la fenêtre des résultats. Choisissez au menu la période à examiner.

Pour imprimer les résultats, cliquez sur FICHIER, ensuite IMPRIMER. Avant d'envoyer les données à l'imprimante, vous pouvez par CONFIGURATION choisir entre plusieurs tailles de papier et entre deux orientations.

Valeurs brutes: Dans ce mode les valeurs brutes sont affichées au lieu des valeurs calculées. Cliquez sur FICHIER, ensuite sur VALEURS BRUTES pour allumer ce mode. La même procédure annule ce mode.

EMULATION: Dans ce mode les résultats ne sont pas tirés de la mémoire de la cible mais sont générés par des nombres au hasard, ce qui vous permet de tester votre programme sans long enregistrement des données réelles. Cliquez sur FICHIER, ensuite sur EMULATION pour allumer ce mode. La même procédure annule ce mode.

ENREGISTRER LES DONNEES: Cliquez sur FICHIER, ensuite sur ENREGISTRER LES DONNEES pour enregistrer les données dans un fichier. Vous pouvez examiner ce fichier avec un editeur comme le BLOCNOTES.

Insérer un Programme Assembleur dans le Prototypage Rapide

Il y a la possibilité d'insérer des fonctions en assembleur dans un programme prototypage rapide.

Pendant la génération de votre programme, le générateur cherche trois fois un programme assembleur dans votre répertoire de travail. Supposons que votre programme s'appelle TOTO.BOY. Après un RESET, avant que l'automate entame son travail normal, le générateur insère le fichier TOTO_R.A11 (R=Reset). Pendant le travail normal de l'automate, toutes les 20 millisecondes le générateur insère le fichier TOTO_A.A11 (A=Always). Finalement vers la fin du fichier, le générateur insère le fichier TOTO_F.A11 (F=Free).

Prenons par exemple l'endroit pendant le travail normal de l'automate. Si le générateur de programme trouve le fichier TOTO_A.A11, il ajoute une instruction

#include TOTO_A.A11

dans le fichier assembleur. S'il ne trouve pas ce fichier, il cherche également le fichier DEFAUL_A.A11. S'il trouve ce fichier, il l'insère, sinon il n'insère aucun fichier. Le programme dans ce fichier est exécuté du haut à bas tout les 20 millisecondes.

Si le nom de votre programme est très long comme MOULINEX.BOY, assurez que le nom du fichier ne dépasse pas 8 caractères: MOULIN_A.BOY.

Si vous créez un fichier TOTO_A.A11, n'oubliez pas de recompiler votre programme prototypage rapide pour que le générateur de programme se rende compte de votre fichier assembleur.

Utilisez des variables en RAM pour échanger des données entre le programme assembleur et votre programme prototypage rapide.

CONTROLBOY: Dépannage avant l'intervention.

Surtout: Ne vous fâchez pas. Les vraies pannes de cartes Controlboy sont rares.

Vous n'avez jamais communiqué correctement avec la carte cible. Vous n'avez jamais chargé un programme dans la cible.

La carte est livrée avec un programme de test chargé dans l'EEPROM.
Controlboy 1/2: Presser les touches T1 ou T2 affichent des valeurs sur l'afficheur et fait basculer régulièrement les relais.
Controlboy 3: Presser la touche T1 bascule les relais et affiche des chiffres sur l'afficheur.
Controlboy F1: Presser la touche T1 bascule la LED rappidement.

Votre programme se charge correctement. Le programme ne se lance pas après le RESET ou après la mise en tension.

Vérifiez que votre programme commence au début de l'EEPROM:
Controlboy 1: $F800.
Controlboy 2,3: $E000.
Controlboy F1: $8000.

La communication avec la carte cible ne fonctionne pas correctement.

Choisissez la surface d'assembleur, ouvrez la fenêtre du débogueur (FENETRES/ DEBOGUEUR). Pressez la touche RESET de la carte cible.

- Le débogueur affiche CIBLE STOP. Vous pouvez maintenant travailler avec la cible. Le dépannage s'arrête ici.
- Le débogueur affiche CIBLE TOURNE. Cliquez sur STOP dans le menu du débogueur pour arrêter le programme qui tourne sur la cible.
- Il n'y a aucune réaction dans la fenêtre du débogueur.
- Le débogueur affiche CIBLE NE REPONDS PAS ou CIBLE REPONDS MAL.

Si votre programme dans la cible ne s'arrête pas, il faut démarrer le talker sans démarrer le programme. Appuyez sur les touches T1 et RESET au même temps, lâchez la touche RESET avant et T1 après. Si le débogueur affiche CIBLE STOP, c'est gagné.

Vérifiez votre alimentation. Une pile n'est pas assez puissante pour alimenter la carte long temps. Vérifiez que vous avez au moins 7 V à l'entrée de la carte est entre 4,8 et 5,2 V sur la carte.

Vérifiez le port de votre ordinateur, que vous utilisez.
Changez de port du P.C.. Changez d'ordinateur.
Vérifiez que vous travaillez bien sous Windows 95, 98 ou NT.
Vérifiez le cordon entre la carte cible et l'ordinateur.

Vérifiez la configuration (FICHIER/CONFIGURATION). Il suffit de choisir le port du P.C. et le matériel dans la deuxième ligne.

Remplacement du talker dans l'EEPROM de la carte cible.
Cette procédure est nécessaire si votre programme a accidentellement abîmé le talker dans l'EEPROM de la cible. Ne suivez pas cette procédure, si vous n'avez jamais travaillé correctement avec la carte cible. Vérifiez la configuration avant de lancer cette opération. Il faut exécuter la commande INITTALKER dans la fenêtre du débogueur qui vous indique les manipulations à faire pour remplacer le talker. Après avoir suivi tous les pas, que le logiciel a proposé, le débogueur doit afficher une cible en état STOP. Cette procédure est un peu pénible et ne marche pas forcement le premier coup. Essayez-la plusieurs fois si c'est nécessaire.

Pour mieux comprendre

Le Talker
Le talker dans la cible assure la communication entre la carte cible et le P.C.. Il se trouve dans l'EEPROM de l'adresse $FE80 à $FFFF et utilise également la mémoire vive de $00E9 à $00FF. Le RESET lance le talker, qui examine l'entrée touche T1. Si celle-ci est à 1 (touche non pressée), le talker saut au début de l'EEPROM pour exécuter directement votre programme d'application. Si celle-ci est à zéro (touche pressée), le talker attend une commande par la ligne série.

Ce que votre programme d'application doit respecter
Votre programme devrait exécuter l'instruction d'assembleur CLI pour permettre la communication entre le débogueur et le talker sur la cible. Cette instruction se trouve dans chaque programme de prototypage rapide, dans le fichier start.bas du Basic11 et dans le fichier crt11.s du CC11.

L'EEPROM
L'EEPROM n'est pas qu'une mémoire passive, mais aussi un automate. Pour écrire correctement dans l'EEPROM, il faut suivre une certaine procédure, qui protège l'EEPROM - et ainsi le talker - contre des écritures involontaires. Pendant cette procédure l'EEPROM n'est pas accessible. Si votre programme écrit accidentellement dans l'EEPROM, l'automate se mets on route et coupe la mémoire du microprocesseur, ça veut dire, le microprocesseur ne peut plus lire l'EEPROM. Comme la procédure pour écrire dans l'EEPROM est assez compliquée, le programme ne peut la changer accidentellement. Mais l'EEPROM reste inaccessible. L'EEPROM ignore le RESET. Il faut donc couper le courant pour quelques secondes pour remettre l'automate à zéro, et surtout pas relancer votre application après la mise en tension.

Ce qui est toujours utile:
Charger la dernière version de logiciel sur notre serveur Internet www.controlord.fr

Utilisez une pince d'extraction pour extraire un composant en boitier PLCC.

Si vous n'arrivez pas à établir la communication avec la carte Controlboy, veuillez nous appeler devant votre ordinateur avec la carte Controlboy connecté et alimenté à coté.

Sinon, renvoyez la carte Controlboy avec le cordon pour la réparation dans nos locaux. Indiquez le défaut et le numéro de téléphone et le nom de personne à contacter. L'intervention est forfaitaire, les composants endommagés en sus.

Demande d'intervention d'une carte Controlboy

Expéditeur:

Personne à contacter:

Téléphone:

Défaut: